Mining and Improving Composite Web Services Recovery Mechanisms

Sami Bhiri, Walid Gaaloul, Claude Godart

International Journal of Web Services Research , Vol.3, No.2, 2008

Mining and Improving Composite Web Services Recovery Mechanisms

Sami Bhiri*, Walid Gaaloul*, and Claude Godart**

*Digital Enterprise Research Institute
IDA Business Park, Galway, Ireland
{sami.bhiri, walid.gaaloul}@deri.org

**LORIA-INRIA
BP 239, F-54506 Vandoeuvre-lès-Nancy Cedex, France
godart@loria.fr

ABSTRACT:

Ensuring composite services reliability is a challenging problem. Indeed, due to the inherent autonomy and heterogeneity of Web services it is difficult to predict and reason about the behavior of the overall composite service.
Generally, previous approaches develop, using their modeling formalisms, a set of techniques to analyze the composition model and check "correctness" properties. Although powerful, these approaches may fail, in some cases, to ensure CS reliable executions even if they formally validate its composition model. This is because properties specified in the studied composition model remains assumptions that may not coincide with the reality (i.e. effective CS executions).
Sharing the same issue, we present a reengineering approach that starts from CS executions log to improve its recovery mechanisms. Basically, we propose a set of mining techniques to discover CS transactional behavior from an event based log. Then, based on this mining step, we use a set of rules in order to improve its reliability.

KEY WORDS:
Web Services Compositions, Mining, Reliability, Transactional Web Service.

Introduction

Nowadays, enterprises are able to outsource their internal business processes as services and make them accessible via the Web. Then, they can dynamically combine individual services to provide new value-added composite services (CS for short). Due to the inherent autonomy and heterogeneity of Web services, a fundamental problem concerns the guarantee of correct executions of a CS. An execution is correct if it reaches its objectives or fails (properly) according to the designers requirements.

Motivating example Let consider an application for online travel arrangement carried out by a composite service as illustrated in figure 1.The customer specifies its requirements in terms of destination and hotel through the *CRS* service. The application launches in parallel flight and hotel reservation (*FR* and *HR* respectively) (after a study of the local transport accommodations

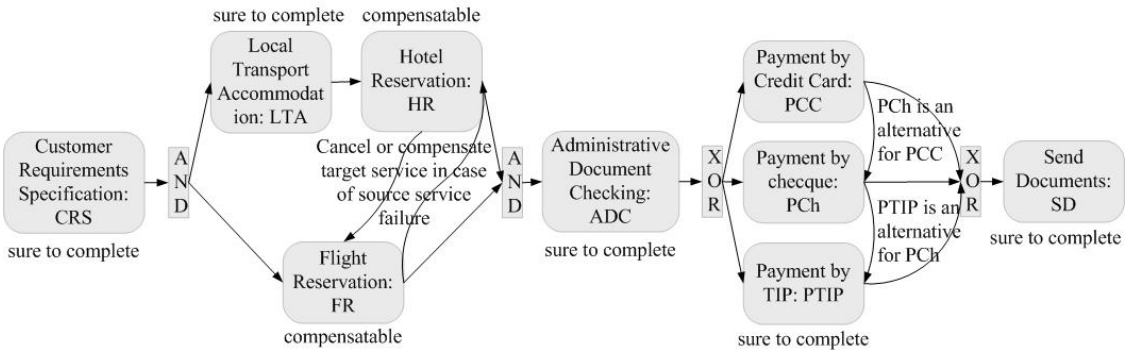


Figure 1. A composite Web service for Online Travel Arrangement (OTA for short)

(*LTA*)). The *ADC* service disposes administrative documents. Then, the customer is requested to pay by credit card (*PCC*), by check (*PCh*), or by TIP (*PTIP*). The Send Documents (*SD*) service ensures the delivery of documents to the customer. To deal with exceptions, designers specify additional mechanisms for failures handling and recovery. First, they specify that the hotel reservation can be compensated (by cancellation for instance) when the *FR* service fails to reserve a flight, and reciprocally. Second, to ensure the payment, they specify the *PCh* service as a payment alternative for the *PCC* service. Similarly, they specify the *PTIP* service as a payment alternative for the *PCh* service with the assumption that the *PTIP* service always succeeds. Finally, designers specify that *CRS*, *LTA*, *ADC* and *SD* services are sure to complete. The main problem at this stage is how to ensure that the specified CS model guaranties reliable executions.

Generally, previous approaches develop, based on their modeling formalisms, a set of techniques to analyze the composition model and check "correctness" properties. Although powerful, these approaches may fail, in some cases, to ensure CS reliable executions even if they formally validate the CS model. This is because properties specified in the studied composition models remains assumptions that may not coincide with the reality.

Back to our example, let us suppose for instance, that in reality (by observation of sufficient execution cases) the *FR* and *PCh* services never fail and the *PTIP* service is not sure to complete. That means, among other, (i) there is no need for the *HR* service to support compensation policies (which can be costly), and (ii) the payment can fail while the hotel and flight reservations are maintained. Formal approaches cannot deal with such anomalies.

Mining the effective transactional behavior allows to detect gaps mentioned above and to improve the application reliability. For instance in our example, mining the transactional behavior allows to improve the CS model by specifying the *PCh* service as a payment alternative for the *PTIP* service (since we notice that *PCh* is sure to complete).

Overview of our approach As explained in section 2, we distinguish between the control flow and the transactional flow of a composite service. The control flow specifies its execution logic (without undesired failures). While the transactional flow defines its recovery mechanisms.

In this paper we present an approach to improve CS recovery mechanisms based on the analysis of its execution history. We proceed in two steps. First, we discover the effective recovery mechanisms (transactional flow) of the composite service. Then, we use a set of rules in order to improve its composition model. Figure 2 gives an overview of the main steps of our approach:

- Collecting execution history: The purpose of this phase is keeping track of the composite service execution by capturing the relevant generated events.

- Analyzing the execution history: The purpose of this phase is mining the effective transactional flow of a composite service. For that we need first to mine its effective control flow and extract its set of termination states.
- Improving the composition model: Based on the execution history analysis we use a set of rules to improve the composite service recovery mechanisms.

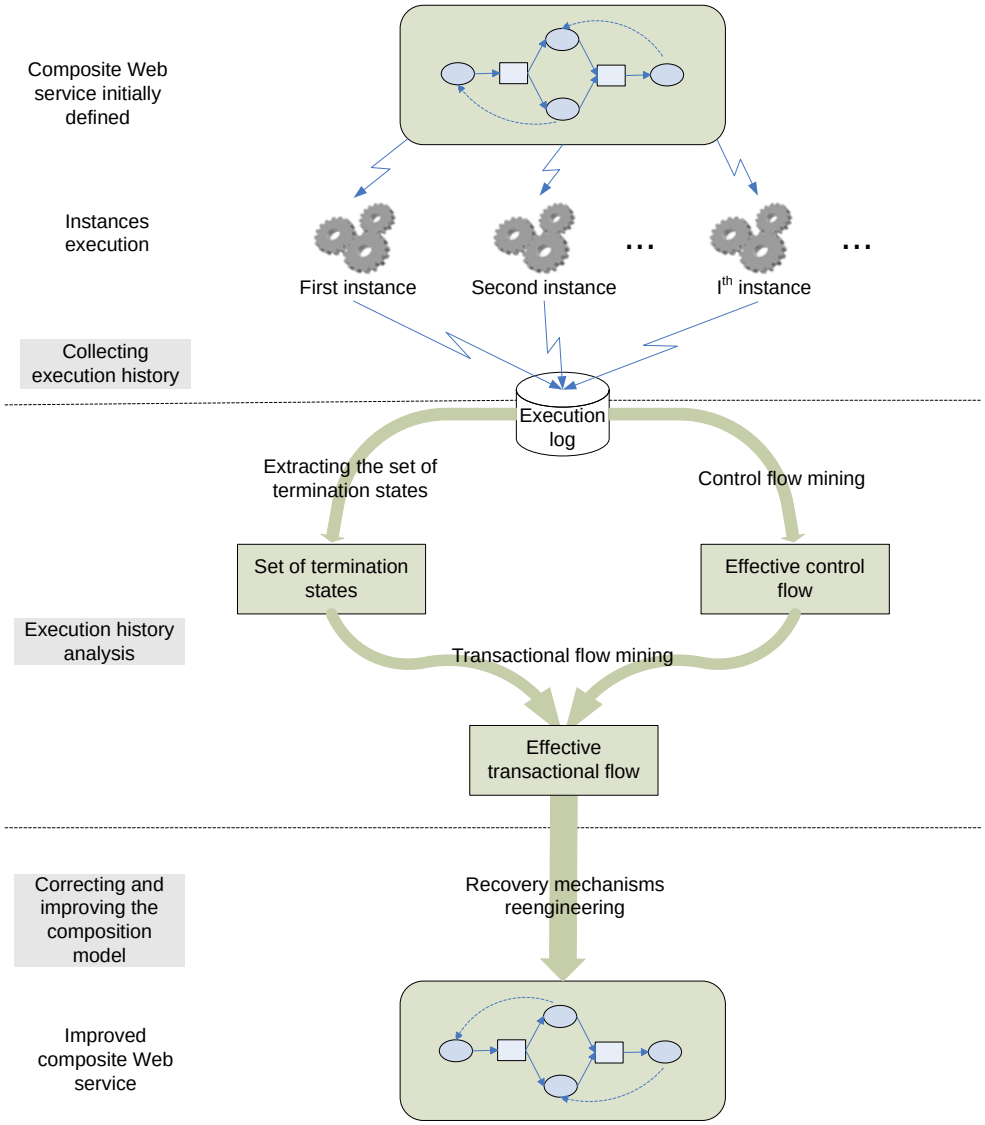


Figure 2. Overview of our approach

The remainder of this paper is organized as follows. In section 2 we introduce our transactional Web service model. Section 3 illustrates how we model a composite Web service according to the presented model. Section 4 discusses Web services logging and shows how we can capture composite service execution history. Section 5 and Section 6 present respectively our control flow and transactional flow mining techniques. In section 7 we show how we proceed to improve CS recovery mechanisms. Section 8 discusses some related work. Section 9 concludes our paper.

Transactional Web Service Model

In this section, we introduce our Web services composition model. We introduce the concept of a transactional Web service (TWS for short). Then we show how we combine a set TWS to define a new value-added service.

Transactional Web Service: TWS

In this paper, by Web service we mean a self-contained modular program that can be discovered and invoked across the Internet. A transactional Web service is a Web service of which the behavior manifests transactional properties.

The main transactional properties of a Web service we are considering are *retriable, compensatable and pivot* (Mehrotra, Rastogi et al. 1992). A service *s* is said to be ***retriable*** if it is sure to complete after several finite activations. *s* is said to be ***compensatable*** if it offers compensation policies to semantically undo its effects. Then, *s* is said to be ***pivot*** if once it successfully completes, its effects remains for ever and cannot be semantically undone. Naturally, a service can combine properties, and the set of all possible combinations is {∅; *retriable; compensatable; pivot; (retriable, compensatable); (retriable, pivot)*}.

Every service can be associated to a life cycle state chart that models the possible states through which the executions of this service can go, and the possible transitions between these states. The set of states and transitions depend on the service transactional properties. Each service has a minimal set of states (*initial, aborted, active, cancelled, failed, completed*) and a minimal set of transitions (*abort(), activate(), cancel(), fail(), complete()*). When a service is instantiated, the state of the instance is *initial*. Then this instance can be either *aborted* or *activated*. Once it is *active*, the instance can normally continue its execution or it can be *cancelled* during its execution. In the first case, it can achieve its objective and successfully *completes* or it can *fail*. A compensatable service has in addition, a state compensated and a transition *compensate()*. A retriable service has in addition a transition *retry()*.

Within a transactional service, we distinguish between external and internal transitions. External transitions are fired by external entities. Typically they allow a service to interact with the outside and to specify composite services orchestration (see next section). The external transitions that we are considering are *activate()*, *abort()*, *cancel(),* and *compensate()*. Internal transitions are fired by the service itself (the service agent). Internal transitions we are considering are *complete()*, *fail()*, and *retry()*.We note TWS the set of transactional Web services.

Transactional Composite Web Service: TCS

A transactional composite (Web) service (TCS for short) is a composite Web service of which the component services are TWS. Such a service takes advantage of its component services transactional properties to specify failure handling and recovery mechanisms. We note TCS the set of transactional composite Web services.

Composition of transactional Web service

A TCS defines a set of preconditions on each component service's external transition in order to define the orchestration schema. These preconditions specify for each component service when it will be aborted, activated, canceled, or compensated. For example, the OTA service specifies that

ADC will be activated after the completion of *HR* and *FR*. That means the precondition of the transition *activate()* of *ADC* is the completion of *HR* and the completion of *FR*. Thus, a TCS can be defined as the set of its component services and the set of the preconditions defined on their external transitions. More formally we define a TCS as following.

Definition 1: A transactional composite Web service *tcs* is a couple $tcs = (ES \in TWS, Prec)$ where *ES* is the set of its component Web services and *Prec* is a function that defines for each component service's external transition a set of preconditions for its activation.

Thus, we distinguish for each component service, s, a set of **exclusive** preconditions for each of its external transition, *activate()*, *abort()*, *cancel()*, and *compensate()*. For instance, the OTA service specifies that *PCh* will be activated either after the completion of *ADC* (exclusively) or after the failure of *PCC*. That means $Prec(PCh.activate()) = \{(ADC.completed \wedge PCh$ chosen for delivery), $PCC.failed\}$.

Preconditions express at a higher abstract level relations (successions, alternatives, etc) between component services in form of dependencies. These dependencies express how services are coupled and how the behavior of certain component service(s) influences the behavior of other service(s). For example the precondition on the external transition *activate()* of the *PCh* service express (i) a succession relations (or dependency) between the *ADC* service and the *PCh* service and (ii) an alternative relation (or dependency) between the *PCC* service and the *PCh* service.

Definition 2 Let be *cs* a TCS, s_1 and s_2 two component services of *cs*, $s_1.t_1()$ a transition of s_1, and $s_2.t_2()$ an external transition of s_2, a dependency from $s_1.t_1()$ to $s_2.t_2()$, denoted $dep(s_1.t_1(), s_2.t_2())$, exists if the activation of $s_1.t_1()$ may fire the activation of $s_2.t_2()$.

In our approach, we consider *activation*, *alternative*, *abortion*, *compensation* and *cancellation* dependencies which we detail in the following.

Activation dependency and activation condition: An activation dependency expresses a succession relation between two services. An activation dependency from s_1 to s_2 exists *iff* the completion of s_1 may fire the activation of s_2. Such dependency is defined according to the activation condition of s_2 $ActCond(s_2)$. $ActCond(s)$ specifies when *s* will be activated (as a successor for other(s) service(s)).

For example, the OTA service shown in figure 1 defines an activation dependency from *HR* to *ADC*, and from *FR* to *ADC* such that *ADC* will be activated after the completion of *HR* and *FR*. That means $ActCond(ADC)=\{HR.completed \wedge FR.completed\}$.

Alternative dependency and alternative condition: Alternative dependencies allow defining execution alternatives as forward recovery mechanisms. An alternative dependency from s_1 to s_2 exists *iff* the failure of s_1 may fire the activation of s_2. Such dependency is defined according to the alternative condition of s_2 $AltCond(s_2)$. $AltCond(s)$ specifies when *s* will be activated (as an alternative) for other(s) service(s).

For instance the OTA service shown in figure 1 defines an alternative dependency from *PCC* to *PCh* such that *PCh* will be activated when *PCC* fails. That means $AltCond(PCh) = \{PCC.failed\}$.

Abortion dependency and abortion condition: An abortion dependency allows propagating failures (causing the TCS abortion) from one service to its successor(s) by aborting them. An

abortion dependency from s_1 to s_2 exists iff the failure, cancellation or the abortion of s_1 may fire the abortion of s_2. Such dependency is defined according to the abortion condition of s_2 $AbtCond(s_2)$. $AbtCond(s)$ specifies when s will be aborted after the failure, the cancellation, or the abortion of other(s) service(s).

Compensation dependency and compensation condition: A compensation dependency allows defining a backward recovery mechanism by compensation. A compensation dependency from s_1 to s_2 exists *iff* the failure or the compensation of s_1 may fire the compensation of s_2. Such dependency is defined according to the compensation condition of s_2 $CpsCond(s_2)$. $CpsCond(s)$ specifies when s will be compensated after the failure or the compensation of other(s) service(s).

The OTA service described in figure 1 defines a compensation dependency from *HR* to *FR* such that *FR* will be compensated when *HR* fails. That means $CpsCond(FR) = \{HR.failed\}$.

Cancellation dependency and cancellation condition: A cancellation dependency allows signaling a service execution failure to other service(s) being carried out in parallel by canceling their execution if necessary. A cancellation dependency from s_1 to s_2 exists *iff* the failure of s_1 may fire the cancellation of s_2. Such dependency is defined according to the cancellation condition of s_2 $CnlCond(s)$. $CnlCond(s)$ specifies when s will be canceled after the failure other(s) service(s).

Control and transactional flow of a TCS

We call the activation and abortion dependencies control dependencies. We call the compensation, cancellation and alternative dependencies transactional dependencies. Control and transactional dependencies express at a higher abstract level respectively the `control flow` and the `transactional flow` of a TCS.

Control flow The `control flow` of a TCS specifies the partial ordering of component services activations. Intuitively the control flow of a TCS is defined by the set of its activation dependencies. Formally, we define a control flow as a TCS where its dependencies are only activation dependencies.

Definition 3 A `control flow` is a TCS, $cf = (ES,Prec)$ such that $\forall s \in ES\ AltCond(s) = \bot$; $CpsCond(s) = \bot$; and $CnlCond(s) = \bot$.

We note $CFlow$ the set of all control flows. We define the function *getCFlow* that returns the control flow of a given TCS.

Definition 4 We define the function *getCFlow* that returns the control flow of a TCS.

$getCFlow$: $TCS \rightarrow CFlow$

$sc = (ES,Prec) \mapsto cf = (ES',Prec')$

such that $ES' = ES$ and $\forall s \in ES\ Prec'(s.activate()) = ActCond(s)$; $Prec'(s.cancel()) = \bot$; $Prec'(s.compensate()) = \bot$.

Transactional flow The `transactional flow` of a TCS specifies the recovery mechanisms. Intuitively, a transactional flow of a TCS is defined by its component services transactional properties and its set of transactional dependencies. Formally we define a transactional flow as a TCS where its dependencies are only transactional dependencies.

Definition 5 A `transactional flow` is a TCS, $tf = (ES,Prec)$ such that $\forall s \in ES$ $ActCond(s)=\perp$.

We note $TFlow$ the set of all transactional flows. We define the function *getTFlow* that returns the transactional flow of a given TCS.

Definition 6 We define the function *getTFlow* that returns the transactional flow of a TCS.
$getTFlow: TCS \rightarrow TFlow$
$sc = (ES,Prec) \mapsto tf = (ES',Prec')$
such that $ES' = ES$ and $\forall s \in ES\ Prec'(s.activate()) = AltCond(s)$.

A TCS, *cs*, can be defined as the union of its control flow, *getCFlow(cs)*, and its transactional flow *getTFlow(cs)*. In general, the union of two TCS cs_1 and cs_2 is a TCS where (i) the set of its component services is the union of cs_1's and cs_2's component services (ii) the precondition of an external transition of a component service *s* is the one defined by cs_1 if *s* belongs only to cs_1, the one defined by cs_2 if *s* belongs only to cs_2, or the union of the preconditions defined by cs_1 and cs_2 if *s* belongs to both of them.

Definition 7 Let two TCS cs_1 and cs_2: $cs_1 = (ES_1,Prec_1)$ and $cs_2 = (ES_2,Prec_2)$. The union of cs_1 and cs_2 is the TCS defined as follows: $cs = cs_1 \cup cs_2 = (ES,Prec)$ where

- $ES = ES_1 \cup ES_2$
- $\forall s \in ES$

$$Prec(s) = \begin{cases} Prec_1(s) \text{ if } s \in ES_1 \wedge s \notin ES_2 \\ Prec_2(s) \text{ if } s \notin ES_1 \wedge s \in ES_2 \\ Prec_1(s) \cup Prec_2(s) \text{ if } s \in ES_1 \wedge s \in ES_2 \end{cases}$$

Relation between the control flow and the transactional flow of a TCS

A TCS transactional flow is tightly related to its control Flow. Indeed, the recovery mechanisms (defined by the transactional flow) depend on the execution process logic (defined by the control flow). For example, regarding the OTA composite service, it is possible to define the *PCh* service as an alternative to the *PCC* service because (according to the XOR control flow operator) they are defined on exclusive branches.

More generally, a control flow implicitly tailors all possible recovery mechanisms. We call a potential transactional flow of a given TCS the transactional flow including all possible transactional dependencies (i.e. recovery mechanisms) that can be defined w.r.t to its control flow. More formally each component service, *s*, has according to the TCS control flow:

- *ptCpsCond(s)*: its potential compensation condition that specifies when it may eventually be compensated.
- *ptAltCond(s)*: its potential alternative condition that specifies when it may eventually be activated as an alternative.
- *ptCnlCond(s)*: its potential cancellation condition that specifies when it may eventually be canceled.

Back to our example, according to the OTA service control flow *FR* may be eventually compensated (i) either after the failure of *ADC*, (ii) or after the compensation of *ADC* (ii) or after the failure of *HR*. That means the potential compensation conditions of *FR* are the failure of *ADC*, the compensation of *ADC*, or the failure of *HR*: $ptCpsCond(FR) = \{ADC.failed, ADC.compensated, HR.failed\}$.

Many TCS can be specified according to the same control flow *cf* . Each one of them extends *cf* with a transactional flow included in its (*cf*) potential transactional flow, *potential(cf)*. More formally:

$$\forall \text{TCS } cs \text{ defined according to a control flow } cf$$
$$cs = getCFlow(cs) \cup getTFlow(cs) \text{ such that}$$
$$getCFlow(cs) = cf \text{ and } getTFlow(cs) \subseteq potential(cf).$$

Figure 8 illustrates two TCS defined according to the same control flow. Each of these TCS extend the control flow shown in figure 3 by a transactional flow included in its potential transactional flow defined in figure 4.

TCS set of termination states

Many executions can be instantiated according to the same TCS model. The state at a specific time of a TCS instance composed of n services can be represented by the tuple $(s_1,s_2, \ldots s_n)$, where s_i is the state of the service instance x_i at this time. The set of termination states of a TCS is the set off all possible termination states of all its instances.

We distinguish two kinds of termination state. The first one corresponds to the termination states reached after normal executions (without unexpected failures according to the control flow). We call a termination state of this first type a `termination state without failure`. The set of `termination states without failures` of a TCS is defined by its control flow.

The second kind of termination state corresponds to the ones reached in case of failure(s) of certain component service(s) (according to the transactional flow). We call a termination state of this second type a `termination state with failure`. The set of `termination states with failure` of a TCS is defined by its transactional flow. We define the function $computeTS_{withFailure}$ that returns the set of `termination states with failure` of a given TCS (more precisely given its transactional flow).

Pattern based modeling

In the previous section, we presented our transactional Web service model allowing capturing both the control and the transactional flow of a TCS. In this section, we show how we model a TCS. We adopt an approach based on workflow patterns (van der Aalst, ter Hostede et al. 2003). We extend them in order to specify, in addition to the control flow they are considering by default, TCS' transactional flow.

Pattern based modeling is interesting for many reasons. Patterns are relatively simple (compared to workflow language) thanks to the abstraction they ensure. Patterns are practical since they are deduced from the practice. In addition they enhance reusability and comprehension between designers. Pattern based modeling allows also modular and local processing. In the following, section 3.1 introduces the composition patterns. Section 3.2 shows how we make use of them in order to specify TCS.

Composition patterns

In the following, we present the workflow patterns from the perspective of our model. Then we show how a given workflow pattern implicitly tailors a set of possible transactional flow.

Workflow patterns

As defined in (Gamma, Helm et al. 1995), a pattern "is the abstraction from a concrete form which keeps recurring in specific non arbitrary contexts". Regarding that, a workflow pattern (van der Aalst, ter Hostede et al. 2003) can be seen as an abstract description of a recurrent class of interactions. For example, the `AND-join` pattern (see figure 3) describes an abstract services interactions as follows: *a service is activated after the completion of several other services*.

Regarding our TCS model, the basic workflow patterns consider only the control flow side. Thus, they can be considered as control flow patterns. Formally, we define a control flow pattern as a function that returns a control flow given a set of services.

Definition 8 A control flow pattern, *pat*, is a function *pat*: $P(TWS)$[1] $\rightarrow$ $CFlow$, that returns a control flow *pat(S)* given a set of transactional services *S*. *pat* defines for each service $s \in S$, its activation condition *ActCond(s)*.

In our approach, we consider the following patterns: *sequence, AND-split, OR-split, XOR-split, AND-join, OR-join, XOR-join and m-out-of-n* (van der Aalst, ter Hostede et al. 2003). Our paper (Bhiri, Godart et al. 2006) details how we define each of these patterns according to the definition 8. Figure 3 illustrates the application of the patterns *AND-split, AND-join, XOR-split,* and *XOR-join*.

Patterns transactional potential

A workflow pattern *pat* defines a control flow *pat(S)* given a set of services. As all control flows, *pat(S)* possesses a potential transactional flow. We define for each workflow pattern, *pat*, a function, $potential_{pat}$, that returns given a set of services *S* the potential transactional flow of *pat(S)*.

Definition 9 Let *pat* a pattern. The function $potential_{pat}$: $P(TWS) \rightarrow TFlow$, returns given a set of services *S*, the potential transactional flow of the control flow *pat(S)*. $potential_{pat}$ defines for each service $s \in S$ its potential compensation condition, *ptCpsCond(s)*, its potential alternative condition, *ptAltCond(s)*, and its potential cancellation condition *ptCnlCond(s)*.

Our paper (Bhiri, Godart et al. 2006) details the potential functions of the patterns AND-split, OR-split, XOR-split, AND-join, OR-join, XOR-join and m-out-of-n. Figure 4 illustrates the application of the potential functions of the patterns *AND split, AND join, XOR split,* and *XOR join*.

TCS specification

Specifying a TCS returns to define its control and its transactional flow. In the following we show how we make use of (i) workflow patterns for defining TCS' control flow and (ii) their transactional potential for defining TCS' transactional flow.

[1] Let *S* a set of elements, *P(S)* denotes the set of subsets of *S*.

Control flow specification

We call `pattern instance`, the control flow resulting from the application of a pattern to a set of services. Let *pat* a pattern and *S* a set of services, *pat(S)* is an instance of *pat*. We use `pattern instances` as the basic brick for specifying TCS' control flow (Bhiri, Godart et al. 2006). Indeed, in our approach a control flow is defined as the union of `pattern instances`. More formally:

$$\forall \text{TCS } cs = (ES, Prec) \; \exists \text{ a set of patterns } \{P_1, \ldots, P_n\} \text{ and a partition } S \text{ of } ES\text{: } S = \{S_1, \ldots S_n\}$$
$$(\text{with } ES = \cup_{1 \le i \le n} (S_i) \mid getFControl(cs) = \cup_{1 \le i \le n} P_i(S_i).$$

Figure 3 shows how we define the control flow of the OTA service as a union of pattern instances.

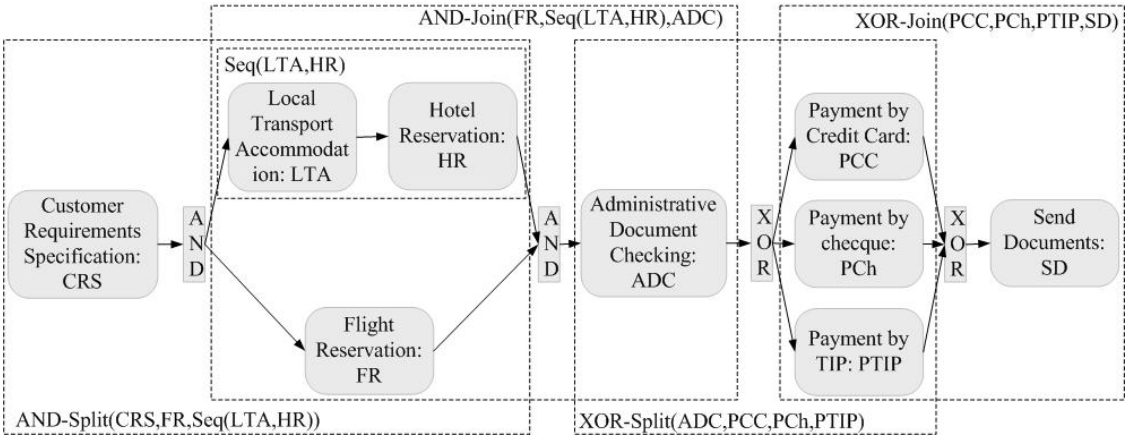


Figure 3. The control flow of the OTA service is defined as a union of pattern instances

Transactional flow specification

The transactional flow of a TCS is included in the potential transactional flow of its control flow. Thus, the first step to define the transactional flow of a TCS is specifying its potential transactional flow. The potential transactional flow of a TCS is the union of the potential transactional flows of its patterns instances. We define the function potential that returns the potential transactional flow of a given control flow.

Definition 10 The function *potential* returns the potential transactional flow of a given control flow:

$$CFlow \rightarrow TFlow$$
$$Cf = \cup_i pat_i(S_i) \mapsto ptf = \cup_i \text{potential}_{pati}(S_i)$$

Figure 4 displays the transactional potential flow of the control flow defined in figure 3. It illustrates how it is the union of the potential transactional flow of the control flow pattern instances.

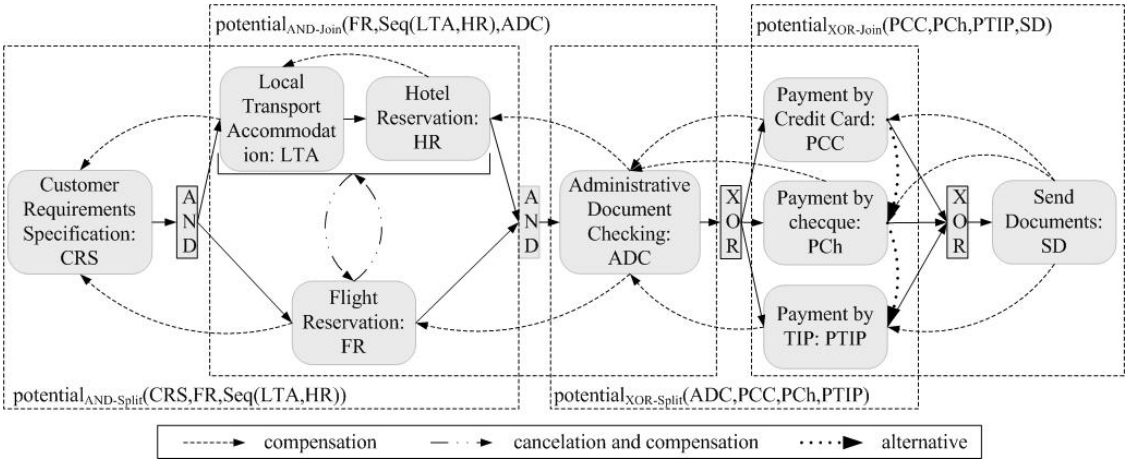


Figure 4. The potential transactional flow of the OTA service is the union of potential transactional flow of its pattern instances

Web Service Logging

Following a common requirement in the areas of business processes and services management, we expect the 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 facilities necessary for the collecting and the logging of Web services execution log (Sahai, Machiraju et al. 2001; Fauvet, Dunas et al. 2002 ; Rouached, Gaaloul et al. 2006). In the following, we examine and formalize the logging possibilities in service oriented architectures which is a requirement to enable the approach described in this paper.

Web service collecting solutions

The first step in the Web Service mining process consists of gathering the relevant Web data, which will be analyzed to provide useful information about the Web Service behavior. We discuss how these log records could be obtained by using existing tools or specifying additional solutions. Then, we show that the mining abilities are tightly related to the information provided in web service log and depend strongly on its richness.

Existing logging solutions provide a set of tools to capture web services logs. These solutions remain quite "poor" to mine advanced web service behaviors. That is why **advanced logging solutions** should propose a set of developed techniques that allows us to record the needed information to mine more advanced behavior. This additional information is needed in order to be able to distinguish between web services composition instances.

Existing logging solutions

There are two main sources of data for Web log collecting, corresponding to the interacting two software systems: data on the Web server side and data on the client side. The existing techniques are commonly achieved by enabling the respective Web server's logging facilities. There already exist many investigations and proposals on Web server log and associated analysis techniques. Actually, papers on Web Usage Mining WUM (Punin, Krishnamoorthy et al. 2001) describe the most well-known means of web log collection. Basically, server logs are either stored in the

Common Log Format[2] or the more recent *Combined Log Format*[3]. They consist primarily of various types of logs generated by the Web server. Most of the Web servers support as a 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 and customer electronic behavior in this new Web environment. Since the Web server log is derived from requests resulting from users accessing pages, it is not tailored to capture service composition or orchestration. That is why, we propose in the following a set of advanced logging techniques that allows to record the additional information to mine more advanced behavior.

Advanced logging solutions

Identifying web service composition instance

Successful mining for advanced architectures in Web Services models requires composition (choreography/ orchestration) information in the log record. Such information is not available in conventional Web server logs. Therefore, the advanced logging solutions must provide an identifier for both choreography and orchestration and a case identifier in each logged interaction.

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 parties 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.

Since all interactions between Web Services happen through the exchange of SOAP message (over HTTP), another alternative is to use SOAP headers that provides additional information on the message's content concerning **choreography**. Basically, we modify SOAP headers to include and gather the additional needed information capturing **choreography** details. Those data are stored in the special `<WSHeaders>`. This tag encapsulates headers attributes like: `choreographyprotocol, choreographyname, choreographycase` and any other tag inserted by the service to record optional information; for example, the `<soapenv:choreographyprotocol>` tag, may be used to register that the service was called by *WS − CDL* choreography protocol. The SOAP message header may look as shown in Figure 5. Then, we use SOAP intermediaries (Anbazhagan and Arun 2002) which are an application 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 either a Web service sender or captures SOAP response messages from either 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 the previous solution).

[2] http://httpd.apache.org/docs/logs.html
[3] http://www.w3.org/TR/WD-logfile.html

```
< soapenv : Header >
        < soapenv : choreographyprotocol
                soapenv : mustUnderstand = ”0”
                xsi : type = ”xsd : string” >WS−CDL
        < /soapenv : choreographyprotocol >
        < soapenv : choreographyname
                soapenv : mustUnderstand = ”0”
                xsi : type = ”xsd : string” > OTA
        < /soapenv : choreographyname >
        < soapenv : choreographycase
                soapenv : mustUnderstand = ”0”
                xsi : type = ”xsd : int” > 123
        < /soapenv : choreographycase >
< /soapenv : Header >
```

Figure 5. The SOAP message header

Concerning **orchestration** log collecting, since the most web services 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 others web services to accomplish a choreography, or discovering the set of interactions exchanged within the context of a given choreography or composition.

Collecting Web service composition instance

The focus in this section is on collecting and analyzing **single** web service composition instance. The issue of identifying several instances has been discussed in the previous section. The exact structure of the web logs or the event collector depends on the web service execution engine that is used. In our experiments, we have used the engine bpws4j[4] that uses log4j[5] to generate logging events. Log4j is an Open Source 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. Figure 6 shows some example of log4j 'logging event' generated by bpws4j engine. The event extractor captures logging event and converts it to a unique TCS log format. These expressions are described in next section.

```
2006-03-13 10:40:39,634    [Thread-35]    INFO    bpws.runtime - Outgoing
```

[4] http://alphaworks.ibm.com/tech/bpws4j
[5] http://logging.apache.org/log4j

```
response: [WSIFResponse:serviceID =
'{http://tempuri.org/services/CRS}CustomerRegServicefb0b0-fbc5965758--8000'operationName
= 'completed'
        isFault = 'false' outgoingMessage = 'org.apache.wsif.base.WSIFDefaultMessage@
        1df3d59 name:null parts[0]:[JROMBoolean: : true]'
        faultMessage = 'null' contextMessage = 'null']
2006-03-13 10:40:39,634        [Thread-35]     DEBUG        bpws.runtime.bus -
Response
        for external invoke is[WSIFResponse:serviceID='{http://tempuri.org/services
        /CCRS}CustomerRegServicefb0b0-fbc5965758--8000'
        operationName = 'authenticate' isFault = 'false'        outgoingMessage =
        org.apache.wsif.base.WSIFDefaultMessage@1df3d59 name:null parts[0]:
        [JROMBoolean: : true]'faultMessage = 'null'     contextMessage = 'null']
2006-03-13 10:40:39,634        [Thread-35]     DEBUG        bpws.runtime.bus -
Waiting
for request
```

Figure 6. Example of log4j 'logging event'

Web mining log structure

The UML class diagram in figure 7 represents a TCS log structure. This log structure represents syntheses through a unique format the information captured by log4j. The conversion from log4j to this format is given in more details in our paper (Rouached, Gaaloul et al. 2006).

As shown, a `TCSLog` (see definition 8) is composed of a set of `EventStreams`. Each `EventStream` traces the execution of one case (instance). It consists of a set of `Events` that capture the services life cycle performed in a particular TCS instance. An `Event` is described by the service identifier that it concerns, the current service state (*aborted, failed, cancelled, completed and compensated*) and the time when it occurs (`TimeStamp`).

Definition 11 A `TCSLog` is considered as a set of `EventStreams`. Each `EventStream` represents the execution of one case. More formally, an `EventStream` is defined as a quadruplet ***EventStream***: (*beginTime, endTime, sequenceLog, SOccurence*) where:

- (*begin: TimeStamp*) and (*end: TimeStamp*) are the moment of log beginning and end,
- *sequenceLog* : `Event*` is an ordered `Event` set belonging to one TCS case,
- (*SOccurence:int*) is the instance number.

So, ***TCSLog***: (*TCSID*, {*ServiceStream*$_i$: ***EventStream***; $0 \leq i \leq$ number of TCS instantiations}) is a TCS log where *ServiceStream*$_i$ is the `EventStream` of the i^{th} TCS execution case.

An example of an **EventStream** extracted from our TCS model example is given below:

EventStream(5, 20, [**Event**(*CRS*, 5, completed), **Event**(*LTA*, 6, completed), Event(*FR*, 8, completed), **Event**(*HR*, 9, completed), Event(*ADC*, 12, completed), **Event**(*PCC*, 13, failed), **Event**(*PCh*, 15, completed), **Event**(*SD*, 20, completed)],1)

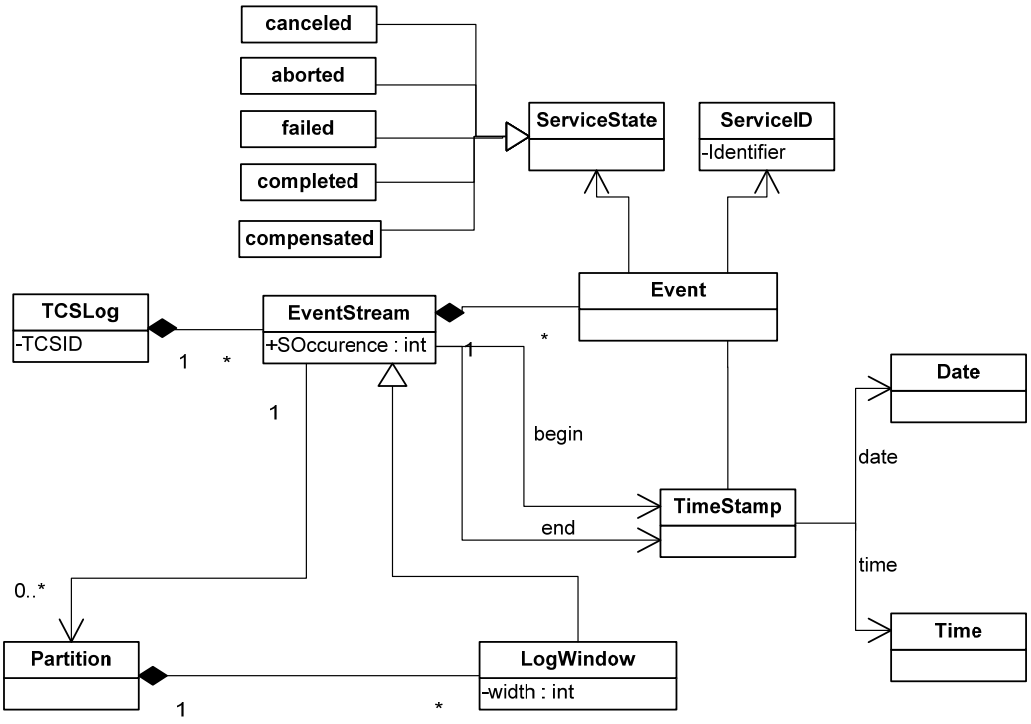


Figure 7. Structure of a TCS log

Control Flow Mining

In this section, we are interested in discovering "elementary" TCS patterns: Sequence, AND-split, OR-split, XOR-split, AND-join, OR-join, and M-out-of-N Join patterns inspired from workflow patterns (van der Aalst, ter Hostede et al. 2003). Our control flow mining approach proceeds in three steps : Step (i) the construction of statistical dependency table SDT, Step (ii) the statistical specifications of patterns' sequential, conditional and concurrent behaviors, and Step (iii) the mining of TCS patterns through a set of rules using these statistical specifications.

Construction of the statistical dependency table SDT

We use statistical calculus that extracts activation dependencies between services executed without "exceptions" (*i.e.* they reached successfully their **completed** state). There is no need to use others `EventStreams` relating to failure executions containing *failed* or *aborted* or *compensated* or *canceled* states. In fact, these cases concern only TCS transactional behavior which tailors the mechanisms for failures handling and recovery. For these reasons, we need to filter TCS log and take only `EventStreams` of instances executed without failures. We denote by $TCSLog_{completed}$ this TCS log selection.

Thus, the minimal condition to discover TCS patterns is to have TCS logs containing at least the *completed* event states. This feature allows us to mine control flow from "poor" logs which contain only *completed* event state. Any information system using transactional systems offer this information in some form (van der Aalst, Weijters et al. 2003).

From $TCSLog_{completed}$ we extract, for each service A, the following information in the statistical dependency table (SDT): (i) The overall frequency of this service (denoted $\#A$) and (ii) The

activation dependencies to previous B_i services (denoted $P(A/B_i)$). The size of SDT is N*N, where N is the number of TCS services. The *(m,n)* table entry (notation *P(m/n)*) is the frequency of the n^{th} service **immediately preceding** the m^{th} service. The table 1 represents a fraction of the SDT of our motivating example. For instance, *P(HR/LTA)*=0.69 expresses that if *HR* occurs then we have 69% of chance that *LTA* occurs directly before in the TCS log.

As it is computed, the initial SDT presents some problems to express correctly services dependencies especially relating to concurrent and parallel behavior. In the following, we detail these issues and propose solutions to correct them.

P(x,y)	*CRS*	*LTA*	*HR*	*FR*	*ADC*
CRS	0	0	0	0	0
LTA	**0.54**	0	0	0.46	0
HR	0	**0.69**	0	0.31	0
FR	**0.46**	0.31	0.23	0	0
ADC	0	0	**0.77**	**0.23**	0
#P=#*CRS*=#*LTA*=#*HR*=#*FR*=#*ADC*=#ST=100 #*PCC*=#*PCh*=#*PTIP*=35					

Table 1. Fraction of Statistical Dependencies Table SDT (P(x,y)) and Services Frequencies (#)

Erroneous dependencies

If we assume that each `EventStream` from `TCSLog` comes from a sequential (i.e. no concurrent behavior) TCS, a zero entry in SDT represents a causal independence and a non-zero entry means a causal dependency (*i.e.* sequential or conditional relations). But in case of concurrent behavior, `EventStreams` may contain interleaved events sequences from concurrent threads. As consequence, some entries, in initial SDT, can indicate nonzero entries that do not correspond to dependencies. For example the `EventStream` given in section 4 "suggests" erroneous activation dependencies between *LTA* and *FR* in one side and *FR* and *HR* in another side. Indeed, *LTA* comes just before *FR* and *FR* comes immediately before *HR*. These erroneous entries are reported by *P(FR/LTA)* and *P(HR/FR)* in SDT which are different to zero. These entries are erroneous because there are no activation dependencies between these services as it was suggested. Underlined values in SDT report this behavior for other similar cases.

Formally, two services *A* and *B* are in concurrence *iff P(A/B)* and *P(B/A)* entries in SDT are different from zero. Based on this definition, we propose an algorithm (Gaaloul, Baina et al. 2005) to discover services parallelism and then mark the erroneous entries in SDT. This algorithm scans the initial SDT and marks concurrent services dependencies by changing their values to (−1). Through this marking, we can eliminate the confusion caused by concurrent behaviors producing these erroneous non-zero entries.

Undetectable dependencies

For concurrency reasons, a service might not depend on its immediate predecessor in the `EventStream`, but it might depend on another "indirectly" preceding service. As an example of this behavior, *FR* is logged between *LTA* and *HR* in the `EventStream` given in section 4. As consequence, *LTA* does not occur always immediately before *HR* in `TCSLog`. Thus we have only *P(HR/LTA)=0.66* that is an under evaluated dependency frequency. In fact, the right value

between these services is 1 because the execution of *HR* depends exclusively on *LTA*. Similarly, values in bold in SDT report this behavior for other cases.

To discover these indirect dependencies, we introduce the notion of service concurrent window (definition 5.1). A service concurrent window (CW) is related to the service of its last event and covers its directly and indirectly preceding services. Initially, the CW width of a service (i.e. the number of services within) is equal to 2. Every time this service is in concurrence with another service we add 1 to this width. If this service is not in concurrence with other services and has preceding concurrent services, then we add their number to CW width. For example *FR* is in concurrence with *LTA* and *HR*, the width of its CW is equal to 3. Based on this we give an algorithm (Gaaloul, Baina et al. 2005) that calculates the CW width for each service and regroups them in the CW table. This algorithm scans the "marked" **SDT** and updates the CW table in consequence.

Definition 12 A `Window` (see figure 7) defines a log slide over an `EventStream` S:***EventStream*** (*bStream, eStream, sLog, TCSocc*). Formally, we define a log window as a triplet ***window***(*wLog, bWin, eWin*) :

- (*bWin*: `TimeStamp`) and (*eWin*: `TimeStamp`) are the moment of the window beginning and end (with $bStream \leq bWin$ and $eWin \leq eStream$),
- $wLog \subset sLog$ and $\forall$ e: **event** $\in S.sLog$ where $bWin \leq e.TimeStamp \leq eWin) \Rightarrow e \in wLog$.

After that, we proceed through an EventStream partition (definition 5.2) that builds a set of partially overlapping Windows over the EventStream using the CW table. Finally, we give an algorithm (Gaaloul, Baina et al. 2005) that computes the final SDT. For each CW, it computes for its last service the frequencies of its preceded services. The final SDT will be found by dividing each row entry by the frequency of its service.

Definition 13 A `Partition` (see figure 7) builds a set of partially overlapping `Windows` partition over an `EventStream`. *Partition*: `TCSLog` → (`Window`)*
S : *EventStream*(*bStream, eStream, sLog, TCSocc*) → $\{w_i$:`Window`; $1 \leq i \leq n\}$:

- $w_1.bWin = bStream$ and $w_n.eWin = eStream$,
- $\forall w$: *window* $\in$ *Partition*, *e*:`Event` = the last event in *w*, *width(w)*= *CW[e.serviceID]*,
- $\forall\ 0 \leq i \leq n;\ w_{i+1}.wLog$ - {the last *e*:`Event` in $w_{i+1}.wLog\} \subset w_i.wLog$ and $w_{i+1}.wLog \neq w_i.wLog$.

By applying previous algorithms, we have computed the final SDT (table 2) which will be used to discover TCS patterns. Note that, our approach adjusts **dynamically**, through the CW width, the process calculating services dependencies. Indeed, this width is sensible to concurrent behavior: it increases in case of concurrence and is "neutral" in case on concurrent behavior absence.

P(x,y)	*CRS*	*LTA*	*HR*	*FR*	*ADC*
CRS	0	0	0	0	0
LTA	**1**	0	0	-1	0
HR	0	**1**	0	-1	0
FR	**1**	-1	-1	0	0
ADC	0	0	**1**	**1**	0

Table 2. Fraction of new calculated SDT

Statistical specifications of sequential, conditional and concurrent properties

We have identified three kinds of statistical properties (sequential, conditional and concurrent) which describe the main behaviors of TCS patterns. Then, we have specified these properties using SDT's statistics. We use theses properties to identify separately TCS patterns from log. This behavior provides a dynamic algorithm that builds global solution (i.e. global WS composition) based on local solutions (i.e. TCS patterns) iteratively. We begin with the statistical exclusive dependency property (property 1) which characterizes, by the way, the sequence pattern.

Property 1 Mutual exclusive dependency property (as P1): A ***mutual*** exclusive dependency relation between a service S_i and its immediately preceding previous service S_j specifies that the enactment of the service S_i depends only on the completion of service S_j and the completion of S_j enacts only the execution of S_i. It is expressed in terms of:

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

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

Property 2 Concurrency property (as P2): A ***concurrency*** relation between a set of services $\{S_i, 0 \leq i \leq n\}$ belonging to the same workflow specifies how, in terms of concurrency, the enactment of these services is performed. This set of services is commonly found after a ''`fork`" operator or before a ''`join`" operator. We have distinguished three services concurrency behaviors:

- ***P2.1: Global concurrency*** where in the same instantiation the whole services are performed simultaneously : $\forall 0 \leq i \neq j < n; \#S_i = \#S_j \wedge P(S_i/S_j) = -1$
- **P2.2:** ***Partial concurrency*** where in the same instantiation we have at least a partial concurrent execution of services : $(\exists\ 0 \leq i \neq j < n; P(S_i/S_j) = -1)$
- **P2.3:** ***No concurrency*** where there is no concurrency between services: $\forall(0 \leq i \neq j < n; \wedge P(S_i/S_j) \neq -1)$

Property 3 Choice property (as P3): A ***choice*** is a relation between the two operands before and after the ''`join`" and the ''`fork`" operator. It specifies, in terms of control flow, how the workflow instance performs the choice of services' operands activations (i.e. which services are executed after a ''`fork`" operator or before a ''`join`" operator). The two operands of the "`fork`" operator (respectively the "`join`" operator) performing this relation are: (operand 1) a service S from which comes (respectively to which) a single thread of control which splits (respectively converges) into (respectively from) (operand 2) multiple services $\{S_i, 0 \leq i < n\}$. We have distinguished three services choice behaviors:

- **P3.1:** ***Free choice*** where a part of services from the second operand are chosen. We have in terms of services frequencies $(\#S \leq \sum_{i=0}^{n-1} (\#S_i)) \wedge (\#S_i \leq \#S)$ and in terms of services dependencies we have :
    - In "`fork`" operator (S_i occurs certainly after S occurrence): $\forall 0 \leq i < n; P(S_i/S) = 1$
    - In "`join`" operator (S occurs certainly after some S_i occurrences "1 <", but not always after all S_i "< n") : $1 < \sum_{i=0}^{n-1} P(S/S_i) < n$

- **P3.2:** ***Single choice*** where only one service is chosen from the second operand. We have in terms of services frequencies ($\#S = \sum_{i=0}^{n-1} (\#S_i)$) and in terms of services dependencies we have:
  - In "fork" operator (S_i occurs certainly after S occurrence): $\forall 0 \leq i < n$; $P(S_i/S) = 1$
  - In "join" operator (S occurs certainly after only one of S_i occurrences): $\sum_{i=0}^{n-1} P(S/S_i) = 1$
- **P3.3:** ***No choice*** where all services in the second operand are executed. We have in terms of services frequencies $\forall 0 \leq i < n$, $\#S = \#S_i$ and in terms of services dependencies we have:
  - In "fork" operator (S_i occurs certainly after S occurrence): $\forall 0 \leq i < n$; $P(S_i/S) = 1$
  - In "join" operator (S occurs certainly after all S_i occurrences): $\forall 0 \leq i < n$; $P(S/S_i)=1$

Patterns mining

Using statistical specifications of sequential, conditional and concurrent behaviors, the last step is the identification of TCS patterns through a set of rules. In fact, each pattern has its own statistical features which abstract statistically its activation dependencies, and represent its unique identifier.

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

We divided the TCS patterns in three categories : sequence, fork and join patterns. Note that the rules formulas noted by : (P1) fingers the statistical exclusive sequential property, (P2) fingers the statistical concurrency property and (P3) fingers the statistical choice property.

Sequence pattern: In this category, we find only the sequence pattern (table 3). In this pattern, the enactment of the *B* service depends only on the completion of the *A* service. So we have used the statistical exclusive dependency property to ensure this relation linking *B* to *A*.

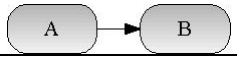
Rules	TCS patterns
(P1) (#B = #A)	Sequence Pattern
(P1) (P(B/A) = 1)	

Table 3. Rules of sequence TCS pattern

Fork patterns: The three patterns of this category (table 4) have a "fork" point where a single thread of control splits into multiple threads of control which can be, according to the used pattern, executed or not. The `AND-split` and `OR-split` patterns differentiate themselves through the no choice and free choice properties. Effectively, only a part of services are executed in the `OR-split` pattern after a "fork" point, while all the B_i services are executed in the `And-split` pattern. The non-parallelism between B_i in the `XOR-split` pattern are ensured by the no concurrency property while the partial and the global parallelism in `OR-split` and `AND-split` is identified through the application of the statistical partial and global concurrency properties.

Join patterns: The three patterns of this category (table 5) have a "join" point where multiple threads of control merge in a single thread of control. The number of necessary branches for the activation of the *B* service after the "join" point depends on the used pattern. The single choice and the no concurrency properties are used to identify the `XOR-join` pattern where two or more alternative branches come together without synchronization and none of the alternative branches is ever executed in parallel. As for the `AND-join` pattern where multiple parallel services converge into one single thread of control, the no choice and the global concurrency are both used to discover this pattern. In contrary of the `M-out-of-N-Join` pattern, where we need only the termination of M of the N incoming concurrent services to enact the *B* service, the concurrency between A_i would be partial and the choice is free.

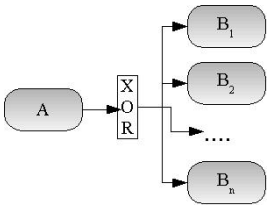
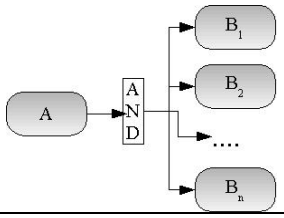
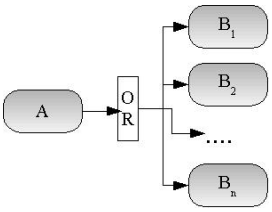
Rules	TCS patterns
(P3) ($\sum_{i=1}^{n}$ (#B_i)= #A)	XOR-split Pattern 
(P3) ($\forall$ 1 ≤ i ≤ n; $P(B_i/A) = 1$) ∧ (P2)($\forall$ 1 ≤ i, j ≤ n; $P(B_i/B_j) = 0$)	
(P3) ($\forall$ 1 ≤ i ≤ n; #B_i = #A	AND-split Pattern 
(P3) ($\forall$ 1 ≤ i ≤ n; $P(B_i/A) = 1$) ∧ (P2)($\forall$ 1 ≤ i, j ≤ n; $P(B_i/B_j) \neq 0$)	
(P3) (#A ≤ $\sum_{i=1}^{n}$ (#B_i)) ($\forall$ 1 ≤ i ≤ n; #B_i ≤ #A)	OR-split Pattern 
(P3) ($\forall$ 1 ≤ i ≤ n; $P(B_i/A) = 1$) ∧ (P2)($\exists$ 1 ≤ i, j ≤ n; $P(B_i/B_j) \neq 0$)	

Table 4. Rules of fork TCS patterns

Transactional flow mining

In this section, we show how we proceed to discover a TCS transactional flow given its control flow and its set of termination states. Regarding our motivating example, we suppose that the two previous mining steps lead to discover the TCS control flow as defined initially by the designers and the TCS set of termination states shown in table 4.

Key Idea

A `termination state with failure` is reached after certain component service(s) failure(s). Such a kind of termination states keeps track of failure(s) produced during the execution and the applied recovery mechanisms. For instance, the `termination state with failure` ts_4 (see table 4) is reached following *HR* failure. In addition, the recovery mechanism applied consists in compensating *FR* and aborting the overall execution.

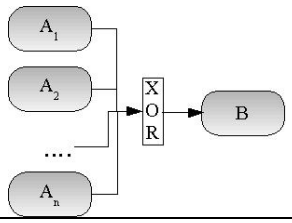
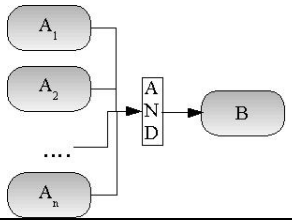
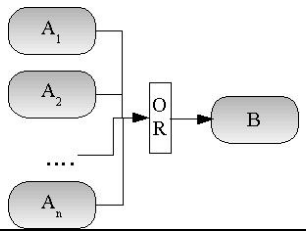
Rules	TCS patterns
(P3) $(\sum_{i=1}^{n} (\#A_i) = \#B)$	XOR-join Pattern 
(P3) $(\sum_{i=1}^{n} P(B/A_i) = 1) \wedge$ (P2)($\forall\ 1 \le i, j \le n$; $P(A_i/A_j) = 0$)	
(P3) ($\forall\ 1 \le i \le n$; $\#A_i = \#B$	AND-join Pattern 
(P3) ($\forall\ 1 \le i \le n$; $P(B/A_i) = 1) \wedge$ (P2)($\forall\ 1 \le i, j \le n$; $P(A_i/A_j) \ne 0$)	
(P3) $(m*\#B \le \sum_{i=1}^{n} (\#A_i))$ ($\forall\ 1 \le i \le n$; $\#A_i \le \#B$)	M-out-of-N-join Pattern 
(P3) $(m \le \sum_{i=1}^{n} P(B/A_i) \le n) \wedge$ (P2)($\exists\ 1 \le i, j \le n$; $P(B_i/B_j) \ne 0$)	

Table 5. Rules of join TCS patterns

	CRS	*LTA*	*HR*	*FR*	*ADC*	*PCC*	*PCh*	*PTIP*	*SD*
ts_1	completed	completed	completed	completed	completed	completed	initial	initial	completed
ts_2	completed	completed	completed	completed	completed	initial	completed	initial	completed
ts_3	completed	completed	completed	completed	completed	initial	initial	completed	completed
ts_4	completed	completed	failed	compensated	aborted	aborted	aborted	aborted	aborted
ts_5	completed	completed	completed	completed	failed	aborted	aborted	aborted	aborted
ts_6	completed	completed	failed	canceled	aborted	aborted	aborted	aborted	aborted
ts_7	completed	completed	completed	completed	completed	failed	completed	initial	completed
ts_8	completed	completed	completed	completed	completed	initial	initial	failed	aborted

Table 4. The extracted set of termination states of the OTA service

For the following, we argue that the control flow, *cf* , is known and fixed. Let $STS_{withFailure}$ a set of `termination states with failure` of a composite service *cs* (of which we know only its control flow *cf*). The transactional flow induced by $STS_{withFailure}$ is the transactional flow (defined according to *cf*) leading to $STS_{withFailure}$ as a set of termination states with failure. It is defined by the reverse function of $computeTS_{withFailure}$: $computeTS^{-1}_{withFailure}$. This function defines for each component service *s*: its transactional properties and its compensation, cancellation, and alternative conditions induced by $STS_{withFailure}$. These conditions specify respectively when s shall be compensated, canceled, or activated as an alternative according to $STS_{withFailure}$.

Thereafter to compute a transactional flow of a TCS given its control flow and its set of termination states it suffices to implement the function $computeTS^{-1}_{withFailure}$. Implementing this function returns to implement how to compute the transactional properties and the transactional conditions induced by $STS_{withFailure}$.

Computing services transactional properties induced by $STS_{withFailure}$

Given the set of termination states of a composite service we can, easily, deduce for each of its component services, *s*, its set of termination states *STS(s)*. For example, given the set of termination states of the service OTA we can deduce that the set of termination states of *FR* is *STS(FR)*={*completed, compensated, cancelled*}. We use the following rules to compute the transactional properties of a component service. ∀ component service, *s*

1. By default *s* is retriable and not compensatable,
2. if *s.failed* ∈ *STS(s)* then *s* is not retriable,
3. if *s.compensated* ∈ *STS(s)* then *s* is compensatable.

The first and second rules allow deducing if a service is retriable or not. The first and third rules allow deducing if a service is compensatable or not. By applying these rules we can deduce, among others, that *FR* is retriable and compensatable. Figure 8.a summarizes these computed transactional properties of component services. Bold properties are the ones that do not match with the initial model.

Computing transactional conditions induced by STSwithFailure

In the following we show how we proceed to compute the compensation condition induced by $STS_{withFailure}$ for a given component service *s*. We proceed similarly to compute the cancellation and alternative conditions. Algorithm 1 allows computing the compensation condition of *s* induced by $STS_{withFailure}$: $CpsCondSTS_{withFailure}(s)$. The principle is: a potential compensation condition of *s* becomes a compensation condition induced by $STS_{withFailure}$ if it occurs in a termination state (with failure) where *s* is compensated.
Thus, the algorithm will go through the set of termination states (line 4 to line 14). For each termination state where *s* is compensated (line 5), the algorithm looks for the potential compensation condition of *s* that holds in this state (line 6 to line 13). Line 7 and line 13 allows going through the potential compensation conditions of *s*. The Boolean variable "satisfied" (line 6 and line 11) enables to mark if the current potential compensation condition holds or not in the current termination state (variable *ts*). A potential compensation condition that holds in *ts* is considered as a compensation condition of *s* induced by $STS_{withFailure}$ (line 10). This condition is retrieved from the set of potential compensation condition of *s* in order to not to be examined again in the other termination states (line 12).

Input: *STS*: the TCS set of termination states

PtCpsCond(s): The potential compensation condition of s defined by the control flow
Output: *CpsCondSTSwithFailure(s)*: the compensation condition of s induced by *STSwithFailure*
Data: *ts*: the current termination state in *STS*
$PtCpsCond_i(s)$: a potential compensation condition of s
satisfied: a Boolean variable set to true when $PtCpsCond_i(s)$ is satisfied in *ts*

```
1  begin
2    CpsCondSTS_withFailure(s) ← ∅
3    ts ← the next ts in STS
4    while ts ≠ null do
5      if the state of s in ts is compensated then
6        satisfied ← false
7        PtCpsCond_i(s) ← the next PtCpsCond_i(s) in PtCpsCond(s)
8          while none satisfied and PtCpsCond_i(s) ≠ null ≠
9            if PtCpsCond_i(s) is satisfied in ts then
10             CpsCondSTSwithFailure(s) ← CpsCondSTSwithFailure(s) PtCpsCond_i(s)
11             satisfied ← true
12             PtCpsCond(s) ← PtCpsCond(s) - PtCpsCond_i(s)
13         PtCpsCond_i(s) ← the next PtCpsCond_i(s) in PtCpsCond(s)
14     ts ← the next ts in STS
15 end
```

Algorithm 1. Extracting the compensation condition of a service *s* induced by $STS_{withFailure}$

For example, the potential compensation condition of *FR*, *HR.failed*, becomes a compensation condition because it is satisfied in ts_4 (in which the state of *FR* is compensated). Figure 8.a illustrates the discovered TCS after the control flow and transactional flow mining.

Improving a TCS recovery mechanisms

To improve TCS recovery mechanisms, we introduce the concept of `intuitively valid` transactional flow. An `intuitively valid` transactional flow is, as its name stands, a transactional flow that respects the following properties of well transactional behavior: (P_1) following a service failure, it tries first to execute an alternative if it exists, (P_2) otherwise (in case of a fatal failure causing the overall composite service failure) it compensates the work already done and (P_3) cancel all running executions in parallel. For example, the discovered transactional flow shown in figure 8.a is not `intuitively valid` since it does not respect, among others, the property P_1 for the service *PTIP* and the property P_2 for the service *ADC*.

To improve a TCS recovery mechanisms, we propose a set of rules that generate suggestions to designers in order to define an `intuitively valid` transactional flow (given the computed transactional properties). We suppose that ◊F means F is eventually true: ∀ component service, *s*

1. $\forall\ ptAltCond_i(s) \in AltCond(s), \Diamond(ptAltCond_i(s)) \wedge ptAltCond_i(s) \notin AltCond(s) \Rightarrow$ *suggest that* $AltCond(s) = AltCond(s) \cup ptAltCond_i(s)$.
2. $\forall\ ptCpsCond_i(s) \in CpsCond(s), \Diamond(ptCpsCond_i(s)) \wedge ptCpsCond_i(s) \notin CpsCond(s) \Rightarrow$ *suggest that*
    a. s must be compensatable and
    b. $CpsCond(s) = CpsCond(s) \cup ptCpsCond_i(s)$.

3. $\forall$ $ptCnlCond_i(s) \in ptCnlCond(s)$, $\Diamond$ $(ptCnlCond_i(s)) \wedge ptCnlCond_i(s) \notin CnlCond(s) \Rightarrow$ *suggest that* $CnlCond(s) = CnlCond(s) \cup ptCnlCond_i(s)$.

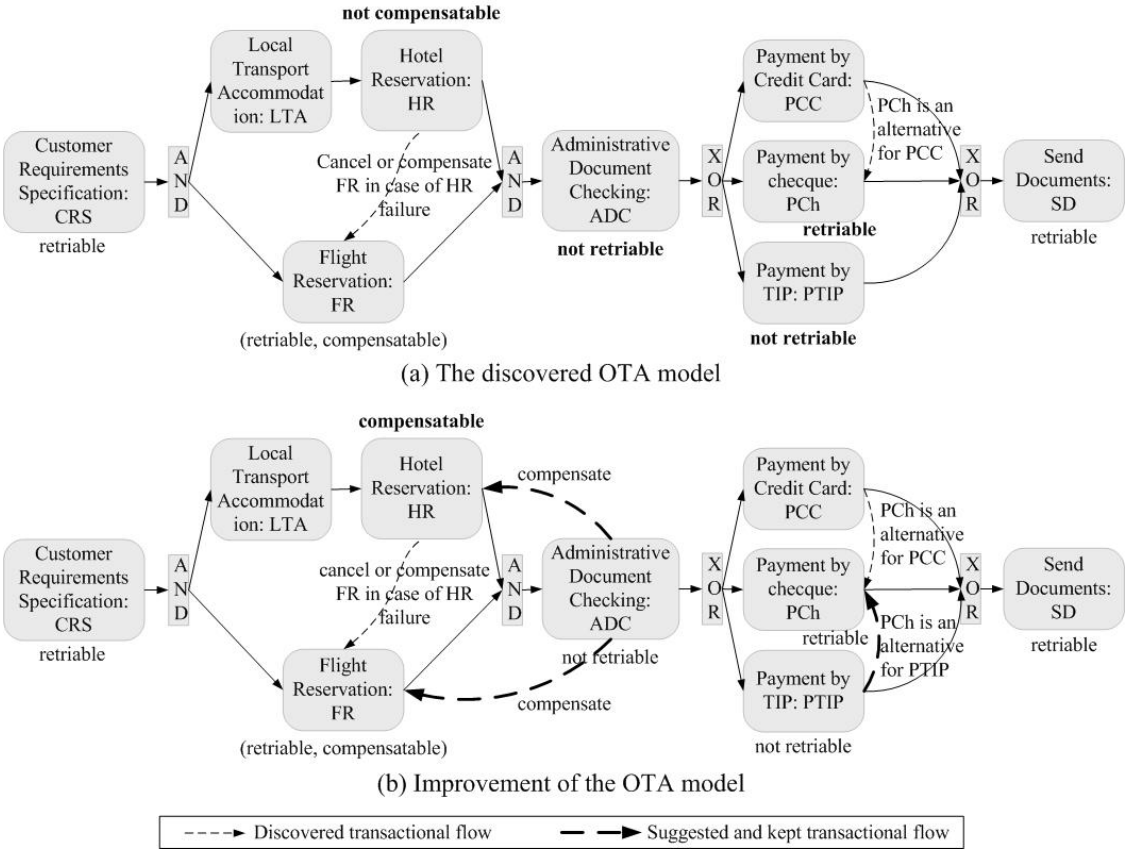


Figure 8. Discovering and improving the OTA service

The first rule aims at ensuring the above property P_1. It postulates that each potential alternative condition of s, $ptAltCond_i(s)$, eventually true must be considered as an alternative condition of s. For example, the potential alternative condition of *PCh* (and *PCC*), *PTIP.failed* is eventually true (since *PTIP* is not retriable) and is not considered as one of its alternative conditions. By applying this rule we generate the following suggestion: $\mathbf{S_1}$: add an alternative dependency from *PTIP* to *PCh* and $\mathbf{S_2}$: add an alternative dependency from *PTIP* to *PCC*.

The second rule aims at ensuring the property P_2. It postulates that each potential compensation condition of s, $ptCpsCond_i(s)$, eventually true must be considered as a compensation condition of s. For instance, the potential compensation condition of *HR* (and *FR*), *ADC.failed*, is eventually true (since *ADC* is not retriable) and is not considered as one of its compensation condition. By applying this rule we generate the suggestion $\mathbf{S_3}$: add two compensation dependencies from *ADC* to *FR* and from *ADC* to *HR*. Similarly by applying this rule we generate the suggestion $\mathbf{S_4}$: add a compensation dependency from *HR* to *LTA*. The third rule aims at ensuring the property P_3. It postulates that each potential cancellation condition of s, *ptCnlCondi(s)*, eventually true must be considered as a cancellation condition of s.

It is worthy to note that the designers have the final decisions about which suggestions consider and which refuse. For instance, designers may reject the above suggestions $\mathbf{S_2}$ and $\mathbf{S_4}$ because *PCC* is not retriable and *LTA* is without effect. Like this, our approach allows to take into account

designers specific needs that may violate the well behavior properties introduced above. Figure 8.b illustrates the OTA service after improvement.

Related work

In this paper we presented an original approach for ensuring reliable Web services compositions. Different from previous works, our approach starts from a composite service (CS for short) executions log and uses a set of mining techniques to discover its control flow and its transactional flow. Then, based on this mining step, we use a set of rules to improve its recovery mechanisms according to designers' specific needs.

Generally, formal previous approaches develop, using their modeling formalisms, a set of techniques to analyze the composition model and check some properties. (Bultan, Fu et al. 2003) proposes a formal framework for modeling, specifying and analyzing the global behavior of Web services compositions. This approach models web services by mealy machines (finite state machines with input an output). Based on this formal framework, authors illustrate the unexpected nature of the interplay between local and global composite Web services. In (Hamadi, Benatallah et al. 2003), authors propose 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.

Other works follow transactional approaches. Emerged standards such as WS-TXM (Acid, BP, LRA) (Doug, Martin et al. 2003), WS-Atomic-Transaction (Little and Wilkinson 2007) and WS-Business-Activity (Freund and Litlle 2007) define transaction protocols between composed services. These approaches rely on advanced transactional models (Elmagarmid 1992). (Limthanmaphon and Zhang 2004) presents a transaction management model based on the tentative hold and compensation concepts. (Pires, Benevides et al. 2002) presents a framework composed of a multilayered architecture, an XML-based language, and a transactional model. (Bhiri, Perrin et al. 2005) proposes a transactional approach to ensure the failure atomicity required by the designers.

Although powerful, the above formal approaches may fail, in some cases, to ensure CS reliable executions 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 transactional web services correction based on event-based logs, and in general there are very few contributions in this area. Prior art in this field is limited to estimating deadline expirations and exceptions prediction (Sayal, Casati et al. 2002; Grigori, Casati et al. 2004). They describe a tool set on top of HPs Process Manager which includes a so-called ''BPI Process Mining Engine". It supports business and IT users in managing process execution quality by providing several features, such as analysis, prediction, monitoring, control, and optimization. However, they neither discuss the correctness of transactional interactions nor address the issue of failures handling and recovery. Indeed, our approach differs from the above: we discover and prevent transactional anomalies and also propose solutions to enhance the CS modeling. We start from a CS log and analyze it in order to reengineer the CS model.

A number of research efforts in the area of workflow management have been directed for mining workflows models. This issue is close to that we propose in terms of discovery. There are

practically no approaches to transactional behavior mining except in (Gaaloul, Bhiri et al. 2004a ; Gaaloul, Bhiri et al. 2004b). Indeed, previous works in workflow discovery focus mainly in control flow mining. The idea of applying process mining in the context of workflow management was first introduced in (Agrawal, Gunopulos et al. 1998). 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, which are inspired by workflow products such as IBM MQSeries workflow (formerly known as Flowmark) and InConcert. Cook and Wolf (Cook and Wolf 1998a) investigated similar issues in the context of software engineering processes. These works are limited to sequential processes. Cook and Wolf extended their work, in (Cook and Wolf 1998b), to concurrent processes. Herbst et al. (Herbest, 2000a; Herbest, 2000b) present an inductive learning component used to support the acquisition and adaptation of sequential process models, generalizing execution traces from different workflow instances to a workflow model covering all traces. Starting from the same kind of process logs, van der Aalst et al. (van der Aalst, Weijters et al. 2003) explore also the area of workflow process mining. They propose techniques to discover workflow models expressed in their own desired workflow modeling notation, which is based on Petri nets. Compared to these work we focus on concurrent behavior in our control flow mining. Indeed, we give a better specification of concurrency behavior through the discovery of CS patterns witch are well-formed structures giving an abstract description of recurrent class of control flow interactions. In besides, we propose a set of control flow mining rules that are characterized by a ”local” CS patterns discovery. These rules are context-free, they proceed through a local log analyzing enabling us to recover correctly partial results even if we have only fractions of CS log.

Conclusion

In this paper we presented a reengineering approach in order to improve recovery mechanisms of Composite Web services (CS for short). Starting from a CS executions log, we use a set of mining techniques to discovering its control and transactional flow. Then, based on this mining step, we use a set of rules to improve its composition model. The originality of our approach is that we correct and improve CS composition models based on their effective execution and behavior.

We introduced a transactional Web service model that integrates the workflow adequacy modeling (rich and complex control structure) and the transactional models reliability (transactional semantics and dependencies with sound recovery mechanisms). In addition, we discussed the existing Web logging solutions and give advanced solutions in order to capture CS execution history.

Our control flow mining approach is original regarding other proposed techniques. It is characterized by a ”local” discovery techniques that allows to recover partial results. In besides, it discovers more behavioral complex features with a better specification of ”fork” point and ”join” point.

However, the work described in this paper represents an initial investigation. In our future works, we hope to discover more complex patterns by using more metrics (e.g. entropy, periodicity, etc.) and by enriching the TCS log. We are also interested in the modeling and the discovery of more complex transactional characteristics of cooperative TCSs.

ACKNOWLEGMENT

This material is based upon works supported by the EU funding under the SUPER project (FP6-026850).

REFERENCES

Agrawal R., Gunopulos D., Leymann F. (1998). Mining Process Models from Workflow Logs, *Proceedings of the 6th International Conference on Extending Database Technology*, Valencia, Spain, March 1998, 469-498.

Anbazhagan, M., Arun, N. (2002). Use SOAP-based intermediaries to build chains of Web service functionality. Retrieved September 1, 2002 from http://www.ibm.com/developerworks/java/library/ws-soapbase/?loc=dwmain.

Bhiri, S., Godart, C., Perrin, O. (2006). Transactional patterns for reliable web services compositions, *Proceedings of the 6th international conference on Web engineering*, Palo Alto, July 2006, 137-144.

Bhiri, S., Perrin, O., Godart, C. (2005). Ensuring required failure atomicity of composite Web services, *Proceedings of the 14th International World Wide Web Conference*, Chiba, Japan, May 2005, 138-147.

Bultan, T., Fu, X., Hull, R., Su, J. (2003). Conversation specification : a new approach to design and analysis of e-service composition, *Proceedings of the 12th International World Wide Web Conference*, Budapest, Hungary, May 2003, 403-410.

Cook, J. E., Wolf, A. L. (1998a). Discovering models of software processes from event-based data, ACM Transactions on Software Engineering and Methodology (TOSEM), v(7), n° 3, 215-249.

Cook, J. E., Wolf, A. L. (1998b). Event-based detection of concurrency, *Proceedings of 6th ACM SIGSOFT FSE conference*, Florida, USA, November 1998, 35-45.

Doug, B., Martin, C., Oisin, H., Mark L., Jeff M., Eric N., Jim W., Keith S. (2003). Web Services Composite Application Framework (WS-CAF). Retrieved July 28, 2003 from http://www.arjuna.com/standards/ws-caf/.

Elmagarmid, A. K. (ed.). (1992). *Database transaction models for advanced applications*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Fauvet, M.-C., Dumas, M., Benatallah, B. (2002). Collecting and Querying Distributed Traces of Composite Service Executions, *Proceedings of the 14th International Conference on Cooperative Information Systems*, California, USA, October-November 2002, 373-390.

Freund, T., Little M. (ed.). (2007). Web Service Business Activity Version 1.1. Retrieved April 16, 2007 from http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec-os.pdf.

Gaaloul, W., Baïna, K., Godart, C. (2005). Towards Mining Structural Workflow Patterns, *Proceedings of DEXA*, Copenhagen, Denmark, August 2005, 24-33.

Gaaloul, W., Bhiri, S., Godart, C. (2004a). Discovering Workflow Patterns from Timed Logs, Proceedings of Informationssysteme im E-Business und E-Government, Beiträge des Workshops derGI-Fachgruppe EMISA, LNI, Luxemburg, October 2004, 84-94.

Gaaloul, W., Bhiri, S., Godart, C. (2004b). Discovering Workflow Transactional Behavior from Event based Log, *Proceedings of the 12th International Conference on Cooperative Information Systems*, Cyprus, October 2004, 25-29.

Gamma, E., Helm, R., Johnson, R., Vlisside, J. (1995). *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.

Grigori, D., Casati, F., Castellanos, M., Dayal, U., Sayal, M., Shan, M.-C. (2004). Business process intelligence, *Computers in Industry*, v(53), n° 3, 321-343.

Hamadi, R., Benatallah, B. (2003). A Petri net-based model for web service composition », *Proceedings of the Australasian Database Conference,* Adelaide, Australia, February 2003, 191-200.

Herbest, J. (2000a). A Machine Learning Approach to Workflow Management, *Proceedings of the 11th European Conference on Machine Learning*, Barcelona, Spain, May – June 2000, 183-194.

Herbest, J. (2000b). Dealing with Concurrency in Workflow Induction, *Proceedings of the European Concurrent Engineering Conference*, Leicester, United Kingdom, April 2000.

Limthanmaphon, B., Zhang, Y. (2004). Web service composition transaction management, *Proceedings of the 15th Australasian Database Conference*, Dunedin, New Zealand, January 2004, 171-179.

Little, M., Wilkinson, A. (ed.). (2007). Web Service Atomic Transaction Version 1.1. Retrieved April 16, 2007 from http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-os.pdf.

Mehrotra, S., Rastogi, R., Korth, H. F., Silberschatz, A. (1992). A Transaction Model for Multidatabase Systems, *Proceedings of International Conference on Distributed Computing Systems*, Yokohama, Japan, June 1992, 56-63.

Pires, P. F., Benevides, M. R. F., Mattoso, M. (2002). Building ReliableWeb Services Compositions, *Proceedings of Web, Web-Services, and Database Systems*, Erfurt, Germany, October 2002, 59-72.

Punin, J., Krishnamoorthy, M., Zaki, M. (2001). Web Usage Mining: Languages and Algorithms, *Proceedings of Studies in Classification, Data Analysis, and Knowledge Organization*, Springer-Verlag, 2001.

Rouached, M., Gaaloul, W., van der Aalst, W. M. P., Bhiri, S., Godart, C. (2006). Web Service Mining and verification of Properties: An approach based on Event Calculus. *Proceedings of the International Cooperative Information Systems*, Montpellier, France, October 2006, 408-425.

Sahai, A., Machiraju, V., Ouyang, J., Wurster K. (2001). Message tracking in SOAP-based Web services, Technical Report HPL-2001-199, Retrieved 2001 from http://www.hpl.hp.com/techreports/2001/HPL-2001-199.html.

Sayal, M., Casati, F., Shan, M., Dayal U. (2002). Business Process Cockpit, Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China, August 2002, 880-883.

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski B., Barros A. P. (2003). Workflow Patterns, *Distributed and Parallel Databases*, v(14), n° 1, 5-51.

van der Aalst, W. M. P., Weijters, T., Maruster, L. (2004). Workflow Mining: Discovering Process Models from Event Logs, IEEE Trans. Knowl. Data Eng, v(16), n° 9, 1128-1142.

ABOUT THE AUTHOR(S)

Dr. Sami Bhiri is a postdoctoral researcher at DERI - the National University of Ireland, Galway, where he is involved in managing several EU projects. Before joining DERI, he was a research and teaching assistant in the University of Nancy 1 and in the ECOO team of the LORIA-INRIA research laboratory. His research interests are in the area of applying semantics to B2B Integration, Service Oriented Computing and Business Process Management.

Dr. Walid Gaaloul is a postdoctoral researcher at the National University of Ireland, Galway, where he is involved in several EU projects. Before joining DERI, he was a research in the ECOO team of the LORIA-INRIA research laboratory and teaching assistant in the University of Nancy 1. His research interests lie in the area of Business Process Management, Process intelligence, Process reliability, Service Oriented Computing and semantics for B2B Integration.

Prof. Dr. Claude Godart is full time Professor at Nancy University, France and scientific director of the INRIA ECOO project. His centre of interest concentrates on the consistency maintenance of the data mediating the cooperation between several partners. This encompasses advanced transaction models, user centric workflow and web services composition models. He has been implicated in several transfer projects with industries (France, Europe, and Japan) for a wide range of applications including e-commerce, software processes and e-learning.