

Discovering and Improving Recovery Mechanisms of Composite Web Services

Sami Bhiri
Digital Enterprise Research Institute
IDA Business Park, Galway, Ireland
sami.bhiri@deri.org

Walid Gaaloul and Claude Godart
LORIA-INRIA
BP 239, F-54506 Vandœuvre-lès-Nancy Cedex, France
{gaaloul, godart}@loria.fr

Abstract

One of the main challenges that encounter Web services is how to ensure reliable compositions. In this paper we present an approach that starts from a composite service effective executions to improve its reliability. Basically, we propose a set of mining techniques to discover its model and its transactional behavior from an event based log. Then, based on this mining step, we use a set of rules to improve its recovery mechanisms.

1. 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). However, due to the inherent autonomy and heterogeneity of Web services, a fundamental problem concerns the guarantee of correct executions of a CS.

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.

In this paper, we present a different approach that starts from a CS executions log and uses a set of mining techniques to discover the CS model and the CS transactional

behavior. Then, based on these mined information, we use a set of rules to improve the CS recovery mechanisms.

2. Motivation and overview

Motivating example We consider an application for on-line 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 (*LTA*)). The *ADC* service disposes administrative documents. Then, the customer is requested to pay either 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 cancelation 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.

Previous approaches may fail, in some cases, to ensure CS reliable executions even if they formally validate the CS model correctness. This is because specified properties in the composition model may not coincide with the reality (*i.e.* effective CS executions).

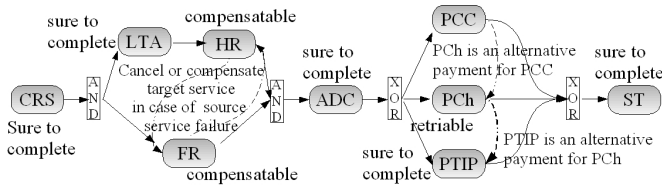


Figure 1. A composite Web service for online travel arrangement

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.

Discovering 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 the section 3, we distinguish between the control flow and the transactional flow of a composite service. Figure 2 overviews our approach. The first step consists in discovering the TCS transactional flow from an event-based log. Then we use a set of rules to improve the TCS recovery mechanisms. The mining phase is itself divided into two steps. First, we mine the TCS control flow and the TCS set of termination states. Then we mine the TCS transactional flow. Due to lack of space, we skip the set of termination states mining step which is trivial.

The remainder of this paper is organized as follows. In section 3 we introduce our transactional Web service model. Section 4 and Section 5 presents respectively our control flow and transactional flow mining techniques. In section 6 we show how we proceed to improve a CS recovery mechanisms. Section 7 concludes our paper and discusses some related and future work.

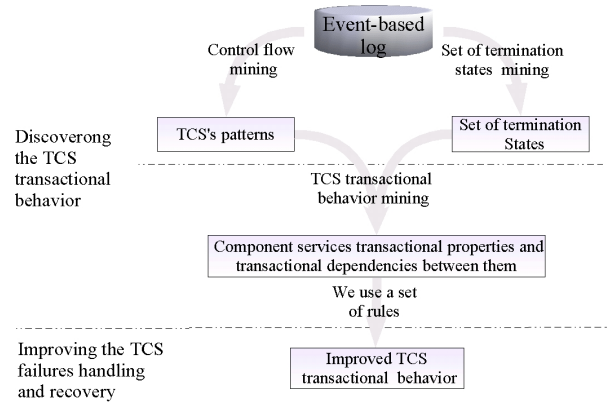


Figure 2. Overview of our approach

3. Transactional Web Services 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.

3.1. 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 *reliable*, *compensatable* and *pivot* [1]. A service *s* is said to be *reliable* 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 $\{r; cp; p; (r, cp); (r, p)\}$.

Every service can be associated to a life cycle state-chart that models the possible statuses through which the executions of this service can go, and the possible transitions between these statuses. 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 retrievable 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()*.

3.2. Transactional composite Web service: TCS

A composite Web service orchestrates a set of services to achieve a common goal. 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.

3.2.1. Composition of transactional Web services 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*.

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* ser-

vice and (ii) an alternative relation (or dependency) between the *PCC* service and the *PCh* service.

Definition 1 *Dependency from a transition $s_1.t_1()$ to an external transition $s_2.t_2()$*

Let be sc a TCS, s_1 and s_2 two component services of sc , $s_1.t_1()$ 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 *cancelation* 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 to define execution alternatives as a 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 to propagate 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, cancelation 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 fail-

ure, the cancellation, or the abortion of other(s) service(s).

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

The OTA service described in figure 1 defines a cancelation dependency from HR to FR such that FR will be canceled when HR fails. That means $CnlCond(FR) = HR.failed$.

It is worthy to note that each of the above conditions are a set of exclusive sub-conditions.

3.2.2. Control and transactional flow of a TCS We call transactional dependencies the compensation, cancelation and alternative dependencies. Activation 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.

We use (workflow-like) patterns to define a composite service control flow. A workflow pattern can be seen as an abstract description of a recurrent class of interactions. For example, the AND-join pattern [2] describes an abstract services orchestration by specifying services interactions as

following: *a service is activated after the completion of several other services*. In our approach, we consider the sequence, AND-split, OR-split, XOR-split, AND-join, OR-join and XOR-join patterns [2].

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 the set of its transactional dependencies. Defining a TCS transactional flow returns to define for each component service its transactional properties and its compensation, alternative and cancelation conditions. It is worthy to note, as shown in the next paragraph, that a transactional flow is defined according to a control flow.

3.3. 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) depends 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 (semantics). 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 cancelation 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) (exclusively) or after the compensation of ADC (ii) (exclusively) or after the failure of HR . That means the potential compensation conditions of FR are the failure of ADC , the compen-

sation of *ADC*, or the failure of *HR*: $ptCpsCond(FR) = \{ADC.failed, ADC.compensated, HR.failed\}$.

3.4. TCS set of termination states

Many executions can be instantiated according to the same TCS model. The state at specific time, of a TCS instance composed of n services can be represented by the tuple (x_1, x_2, \dots, x_n) , where x_i is the state of the service instance x_i at this time. The set of termination states of a TCS is the set of all possible termination states of all its instances.

We distinguish two kind of termination states. The first one corresponds to the termination states reached after normal executions (without unexpected failures according to the control flow). We call such a termination states 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 states 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).

4. Control Flow Mining

4.1. TCS Event log

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 [3].

A TCSLog is composed of a set of EventStreams. Each EventStream traces the execution of one case (instance). It consists of a begin and end time, and 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* or *compensated*) and the time when it occurs. 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)])
```

Our control flow mining approach proceeds in two steps : (i) the construction of statistical dependency table SDT, and (ii) the mining of TCS patterns through a set of rules applied on the calculated SDT table.

4.2. Construction of the statistical dependency table SDT

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 [4].

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 component 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 3 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 executions. In the following, we detail these issues and propose solutions to correct them.

P(x,y)	CRS	LTA	HR	FR	ADC	PCC	PCh	PTIP	SD
CRS	0	0	0	0	0	0	0	0	0
LTA	0.54	0	0	<u>0.46</u>	0	0	0	0	0
HR	0	0.69	0	<u>0.31</u>	0	0	0	0	0
FR	0.46	<u>0.31</u>	<u>0.23</u>	0	0	0	0	0	0
ADC	0	0	0.77	0.23	0	0	0	0	0
PTIP	0	0	0	0	0	0	0	0	0
SD	0	0	0	0	0	0	0	0.35	0

#CRS=#LTA=#HR=#FR=#ADC=#SD=100
#PCC=23 #Pch=42 #PTIP=35

Figure 3. Fraction of the Initial SDT

4.2.1. Erroneous dependencies We argue that a zero entry in SDT represents a causal independence and a non-zero entry means a causal dependency. But in case of concurrent executions, 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 dependencies. For example the EventStream given in section 4.1 “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 is 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 [5] 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.

4.2.2. 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, FR is logged between LTA and HR in the EventStream given in section 4.1. As consequence, LTA does not always occur 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. A component service concurrent window (CW) defines a log slide over an EventStream and is related to the service of its last event and covers its directly and indirectly preceding services. Initially, the width of a service s CW (*i.e.* the number of services within) is equal to 2. Every time s is in concurrence with an other service we add 1 to this width. If s 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 propose an algorithm [5] that calculates the CW width for each service and regroups them in a CW table.

Then, we proceed through an EventStream partition that builds a set of partially overlapping Windows over the EventStream using the CW table. Finally, we use an algorithm [5] 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. Table 4 illustrates the final SDT after correction.

P(x,y)	CRS	LTA	HR	FR	ADC	PCC	PCh	PTIP	SD
CRS	0	0	0	0	0	0	0	0	0
LTA	1	0	0	-1	0	0	0	0	0
HR	0	1	0	-1	0	0	0	0	0
FR	1	-1	-1	0	0	0	0	0	0
ADC	0	0	1	1	0	0	0	0	0
PTIP	0	0	0	0	0	0	0	0	0
SD	0	0	0	0	0	0	0	0.35	0

Figure 4. Fraction of Final SDT

4.3. Patterns mining

The second step is the identification of TCS patterns through a set of rules. Actually, each pattern has its own statistical features which abstract its activation dependencies, and represent its unique ID. We divide the TCS patterns into three categories (c.f figure 5): sequence, split and join patterns. Table 1 details their respective statistical ID rules. These ID rules ensure a “local” patterns discovery.

sequence	Rules		
	$(\#B = \#A) \wedge (P(B/A) = 1) \wedge \forall A_{0 \leq i < n} \neq A; P(B/A_i) = 0 \wedge \forall B_{0 \leq j < n} \neq B; P(B_j/A) = 0$		
split	Rules	join	Rules
(xor)	$(\sum_{i=0}^{n-1} (\#B_i) = \#A) \wedge$ $(\forall 0 \leq i < n; P(B_i/A) = 1) \wedge$ $(\forall 0 \leq i \neq j < n; P(B_i/B_j) = 0)$	(xor)	$(\sum_{i=0}^{n-1} (\#A_i) = \#B) \wedge$ $\sum_{i=0}^{n-1} P(B/A_i) = 1 \wedge$ $\forall 0 \leq i \neq j < n; P(A_i/A_j) = 0$
(and)	$((\forall 0 \leq i < n; \#B_i = \#A) \wedge$ $(\forall 0 \leq i < n; P(B_i/A) = 1) \wedge$ $(\forall 0 \leq i \neq j < n; P(B_i/B_j) = -1)$	(and)	$(\forall 0 \leq i < n; \#A_i = \#B) \wedge$ $(\forall 0 \leq i < n; P(B/A_i) = 1) \wedge$ $(\forall 0 \leq i \neq j < n; P(A_i/A_j) = -1)$
(or)	$\#A \leq \sum_{i=0}^{n-1} (\#B_i) \wedge$ $(\forall 0 \leq i < n; \#B_i \leq \#A) \wedge$ $(\forall 0 \leq i < n; P(B_i/A) = 1) \wedge$ $(\exists 0 \leq i \neq j < n; P(B_i/B_j) = -1)$	(M-out -of-N)	$(m * \#B \leq \sum_{i=0}^{n-1} (\#A_i))$ $\wedge (\forall 0 \leq i < n; \#A_i \leq \#B)$ $(m \leq \sum_{i=0}^{n-1} P(B/A_i) \leq n)$ $\wedge (\exists 0 \leq i \neq j < n; P(A_i/A_j) = -1)$

Table 1. Rules of sequence, split and join patterns

Indeed, to discover a particular TCS pattern we need only events relating to its elements (services). Thus, even using only fractions of TCS logs, we can correctly discover corresponding TCS patterns (which their events belong to these fractions).

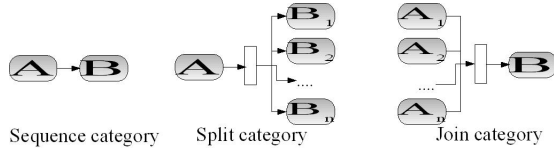


Figure 5. Workflow patterns categories

Sequence pattern : In this category, we find only the sequence pattern. In this pattern, the enactment B depends only on the completion of A and the completion of A enacts only the execution of B .

Fork patterns : The three patterns of this category have a “fork” point where a single thread of control splits into multiple threads of control (that can be, according to the used pattern, executed or not). The three patterns share the causality between A and the services B_i ; that means $(\forall 1 \leq i \leq n; P(B_i/A) = 1)$. In the xor-split pattern, the non-parallelism between B_i , is identified by the statistical property $(\forall 1 \leq i, j \leq n; P(B_i/B_j) = 0)$. The difference between the or-split and the and-split patterns is the frequencies relation between A and the services B_i . Effectively, in the or-

split pattern only a part of these services are executed after the “fork” point. However, all the services B_i are executed in case of the and-split pattern.

Join patterns : The three patterns of this category has a “join” point where multiple threads of control merge in a single thread of control. The three patterns differ in the number of necessary branches to activate B . The and-join pattern requires the execution of all the A_i activities; which can be identified by $(\forall 1 \leq i \leq n; P(B/A_i) = 1)$. The M-out-of-N-Join pattern supports a “partial” parallelism between A_i services; that means $(\exists 1 \leq i, j \leq n; P(A_i/A_j) = -1)$. Finally the no parallelism between A_i in the xor-join pattern can be caught by $((\forall 1 \leq i, j \leq n; P(B_i/B_j) = 0))$.

5. 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. We suppose, for our motivating example, 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 figure 6.

5.1. 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 fail-

	CRS	LTA	HR	FR	ADC	PCC	PCh	PTIP	SD
ts ₁	(cpl.	cpl.	cpl.	cpl.	cpl.	cpl.	int.	int.	cpl)
ts ₂	(cpl.	cpl.	cpl.	cpl.	cpl.	int.	cpl.	int.	cpl)
ts ₃	(cpl.	cpl.	cpl.	cpl.	cpl.	int.	int.	cpl.	cpl)
ts ₄	(cpl.	cpl.	fld.	cps.	abr.	abr.	abr.	abr.	abr)
ts ₅	(cpl.	cpl.	cpl.	cpl.	fld.	abr.	abr.	abr.	abr)
ts ₆	(cpl.	cpl.	fld.	cnl.	abr.	abr.	abr.	abr.	abr)
ts ₇	(cpl.	cpl.	cpl.	cpl.	fld.	cpl.	int.	int.	cpl)
ts ₈	(cpl.	cpl.	cpl.	cpl.	cpl.	int.	int.	fld.	abr)

cpl=completed, fld=failed, ps=compensated,
cnl=canceled, int=initial, abr=aborted

Figure 6. The discovered set of termination states of the online travel arrangement TCS

ure(s) produced during the execution and the applied recovery mechanisms. For instance, the termination state with failure ts_4 (c.f figure 6) is reached following HR failure. In addition, the recovery mechanism applied consists in compensating FR and aborting the overall execution.

Let $STS_{withFailure}$ the set of termination states with failure of a composite service cs (of which we know its control flow). The transactional flow induced by $STS_{withFailure}$ is defined by the reverse function of $computeTS_{withFailure}$: $computeTS_{withFailure}^{-1}$. This function defines for each component service s : its transactional properties and its compensation, cancelation, 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 $computeTS_{withFailure}^{-1}$. Implementing this function returns to implement how to compute the transactional properties and the transactional conditions induced by $STS_{withFailure}$.

5.2. 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

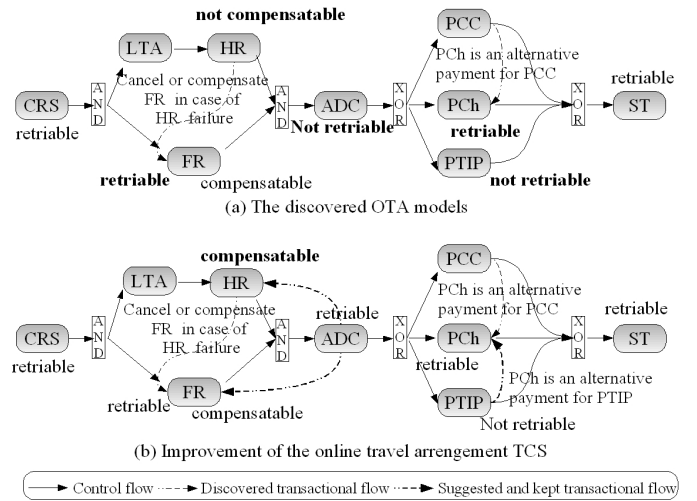


Figure 7. Discovering and improving the online travel arrangement TCS

the following rules to compute the transactional properties of a component service. \forall component service, s

1. By default s is *retriabale* and *not compensatable*
2. if $s.failed \in STS(s)$ then s is *not retriabale*
3. if $s.compensated \in STS(s)$ then s is *compensatable*

The first and second rules allow to deduce if a service is retriabale or not. The first and third rules allow to deduce if a service is compensatable or not. By applying these rules we can deduce, among others, that FR is retriabale and compensatable. Figure 7.b summarizes these computed transactional properties of component services. Bold properties are the ones that do not match with the initial model.

5.3. Computing transactional conditions induced by $STS_{withFailure}$

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 cancelation and alternative conditions.

The algorithm shown in figure 8 allows to compute the compensation condition os s induced by $STS_{withFailure}(s)$: $CpsCondSTS_{withFailure}(s)$. Due to lack of space we give only the idea of the algorithm: The compensation condition of s induced by $STS_{withFailure}$ is a subset of its potential compensation condition; $CpsCondSTS_{withFailure}(s) \subseteq PtCpsCond(s)$. To compute $CpsCondSTS_{withFailure}(s)$ it

suffices to find which elements in $PtCpsCond(s)$ occur in the termination states in which s is compensated.

```

Input: STS: the TCS set of termination states
PtCpsCond(s): The potential compensation condition of s
Output: CpsCondSTSwithFailure(s): the compensation condition of s induced by STSwithFailure
Data: ts: the current termination state in STS
PtCpsCondi(s): a potential compensation condition of s
satisfied: a boolean variable set to true when PtCpsCondi(s) is satisfied in ts
begin
  CpsCondSTSwithFailure(s) = ∅
  ts = the next ts in STS
  While ts ≠ null
    If the state of s in ts is compensated then
      satisfied = false
      PtCpsCondi(s) = the next PtCpsCondi(s) in PtCpsCond(s)
      While none satisfied and PtCpsCondi(s) ≠ null
        If PtCpsCondi(s) is satisfied in ts then
          CpsCondSTSwithFailure(s) = CpsCondSTSwithFailure(s) ∪ PtCpsCondi(s)
          satisfied = true
          PtCpsCond(s) = PtCpsCond(s) - PtCpsCondi(s)
          PtCpsCondi(s) = the next PtCpsCondi(s) in PtCpsCond(s)
        ts = the next ts in STS
  end

```

Figure 8. 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 $ts4$ (in which the state of FR is compensated). Figure 7.b illustrates the discovered TCS after the control flow and transactional flow mining.

6. Improving a TCS recovery mechanisms

To improve a TCS recovery mechanisms, we introduce the concept of intuitive valid transactional flow. An intuitive valid transactional flow can be characterized by the following three properties: (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 behavior shown in figure 7.b 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 intuitive valid transactional flow (given the computed transactional properties). We suppose that $\diamond F$ means F is eventually true: \forall component service, s

1. $\forall ptAltCond_i(s) \in AltCond(s),$
 $\diamond(ptAltCond_i(s)) \wedge ptAltCond_i(s) \notin AltCond(s) \Rightarrow$
 $AltCond(s) = AltCond(s) \cup ptAltCond_i(s).$
2. $\forall ptCpsCond_i(s) \in ptCpsCond(s),$
 $\diamond(ptCpsCond_i(s)) \wedge ptCpsCond_i(s) \notin CpsCond(s) \Rightarrow$
 (a) s must be compensatable and
 (b) $CpsCond(s) = CpsCond(s) \cup ptCpsCond_i(s).$
3. $\forall ptCnlCond_i(s) \in ptCnlCond(s),$
 $\diamond(ptCnlCond_i(s)) \wedge ptCnlCond_i(s) \notin CnlCond(s) \Rightarrow$
 $CnlCond(s) = CnlCond(s) \cup ptCnlCond_i(s).$

Due to lack of space we explain only the first rule. The second and third rules can be understood similarly. The first rule aims to ensure 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 retrievable) and is not considered as one of its alternative conditions. By applying this rule we can generate the following suggestion: s_1 : add alternative dependencies from $PTIP$ to PCh and s_2 : from $PTIP$ to PCC .

By applying the other two rules to our example, we can also generate the following suggestions s_3 : add two compensation dependencies from ADC to FR and from ADC to HR , and s_4 : add a compensation dependency from HR to LTA .

It is worthy to note that the designers have the final decision which suggestions consider and which refuse. For instance, designers may reject the above suggestions s_2 and s_4 because PCC is not retrievable and LTA is effectless. Like this, our approach allows to take into account designers specific needs that may violate the well behavior properties introduced above. Figure 7.c illustrates the OTA service after improvement.

7. Discussion

In this paper we presented an original approach for ensuring reliable Web services compositions. Different from previous works, our approach starts from a TCS 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 the TCS recovery mechanisms according to designers specific needs.

Generally, previous approaches like [6–8] develop, based on their modeling formalisms, a set of techniques to analyze the composition model and check "correctness" properties. [6] proposes a formal framework, based on mealy machines, for modelling, specifying and analyzing the global behavior of Web services compositions. [7] proposes a Petri net-based algebra for composing Web services. [8] proposes a transactional approach to ensure the failure atomicity required by the designers. Although powerful, these approaches may fail, in some cases, to ensure TCS reliable executions even if they formally validate the TCS model. This is because properties specified in the studied composition models remains assumptions that may not coincide with the reality.

Previous works in workflow discovery focus mainly in control flow mining [4, 9–11]. They do not deal with the transactional flow. In our approach we proposed not only how to discover a TCS transactional flow, but in addition how to improve it according to designers specific needs. Furthermore regarding these approaches, our control flow mining approach is original. It is characterized by a "local" discovery techniques that allows to recover partial results [5]. In besides, it discovers more behavioral complex features with a better specification of "fork" point and "join" point and adjusts dynamically in case of concurrence the control flow mining process. We have implemented our presented patterns mining algorithms within a prototype [5].

To the best of our knowledge, there are practically no approaches to transactional Web services mining based on log. Our approach can be seen as the first work in this field. Our current work is about integrating the transactional enrichment of our approach in semantic Web services approaches. The transactional properties we are using can be, easily, integrated and described in the non functional properties block in a WSMO [12] service. We are also interested in discovering more complex transactional characteristics of

composite Web services by enriching more the TCS log.

References

- [1] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz, "A transaction model for multidatabase systems." in *ICDCS*, 1992, pp. 56–63.
- [2] W. M. P. van der Aalst, A. P. Barros, A. H. M. ter Hofstede, and B. Kiepuszewski, "Advanced workflow patterns," in *CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings*, O. Etzion and P. Scheuermann, Eds. Springer, 2000, pp. 18–29.
- [3] M.-C. Fauvet, M. Dumas, and B. Benatallah, "Collecting and querying distributed traces of composite service executions." in *CoopIS/DOA/ODBASE*, 2002, pp. 373–390.
- [4] W. van der Aalst and L. Maruster, "Workflow mining: Discovering process models from event logs," in *QUT Technical report, FIT-TR-2003-03*, Brisbane, 2003.
- [5] W. Gaaloul, K. Baina, and C. Godart, "Towards mining structural workflow patterns." in *DEXA*, 2005, pp. 24–33.
- [6] T. Bultan, X. Fu, R. Hull, and J. Su, "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, 2003, pp. 403–410.
- [7] R. Hamadi and B. Benatallah, "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., 2003, pp. 191–200.
- [8] S. Bhiri, O. Perrin, and C. Godart, "Ensuring required failure atomicity of composite web services." in *WWW*, 2005, pp. 138–147.
- [9] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining process models from workflow logs," *Lecture Notes in Computer Science*, vol. 1377, pp. 469–498, 1998.
- [10] J. E. Cook and A. L. Wolf, "Event-based detection of concurrency," in *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Press, 1998, pp. 35–45.
- [11] J. Herbst, "A machine learning approach to workflow management," in *Machine Learning: ECML 2000, 11th European Conference on Machine Learning, Barcelona, Catalonia, Spain*, vol. 1810. Springer, Berlin, May 2000, pp. 183–194.
- [12] D. Roman, H. Lausen, and U. Keller(eds), "Web service modelling ontology, wsmo deliverable d2 version 1.1," in <http://www.wsmo.org/2004/d2/v1.1>.