Transactional Patterns for Reliable Web Services Compositions

Sami Bhiri
DERI
National University of Ireland
IDA Business Park, Galway, Ireland
sami.bhiri@deri.org

Olivier Perrin
LORIA-INRIA
BP 239, F-54506
Vandœuvre-lès-Nancy Cedex, France
godart@loria.fr

Claude Godart
LORIA-INRIA
BP 239, F-54506
Vandœuvre-lès-Nancy Cedex, France
operrin@loria.fr

ABSTRACT

Reliability is one of the main challenge that encounter Web services compositions. Due to the inherent autonomy and heterogeneity of Web services it is difficult to predict the behavior of the overall composite service.

Current related technologies are unable to resolve this problem efficiently. These technologies rely on two existing strong approaches: transactional processing and workflow systems. In one hand transactional processing ensures reliability. However, they are too rigid to support process based applications like composite Web services. On the other hand, workflow systems focus mainly on coordination and organizational aspects and ignore reliability issues.

In this paper we propose a new solution that combines the business process adequacy of workflow systems and the reliability of transactional processing. We introduce the concept of transactional patterns to ensure reliable composite services. A transactional pattern can be seen as a convergence concept between workflow patterns and advanced transactional models. We show how we use it to define composite services and how we ensure their reliability according to the designers specific needs.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; H.2.4 [**Database Management**]: Systems—*Transaction Processing*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Distributed commercial transactions*

General Terms

Design, Reliability

Keywords

Web services compositions, Reliability, Workflow patterns, Transactional processing.

1. INTRODUCTION

Web services approach is extending the Web from an information support to a B2B middelware. One of the main concept that offers this technology is the ability to define a new composite service using existing services. In this paper, we are interested in how to ensure reliable Web services compositions. By reliable composition we mean a composition where all its executions are correct (from a business point of view). An execution is correct if it reach its objective or fails according to the designers requirements. Due to the inherent autonomy and heterogeneity of Web service it is difficult to predict the overall behavior of a composite service.

Current related technologies are unable to resolve this problem efficiently. These technologies rely on two existing strong approaches: transactional pro-cessing and workflow systems. Transactional processing aim to ensure correct execution of a set of operations encapsulated inside a treatment unit called transaction. Workflow systems deal with coordination and organizational aspect of business processes. Taken separately, these two technologies are unable to ensure reliable Web services compositions.

In one hand, Advanced Transaction Models (ATM) [4], although powerful and providing a nice theoretical framework, are too database-centric, limiting their possibilities and scope [1] in this context (e.g. their inflexibility to incorporate different transactional semantics as well as different behavioral patterns into the same structured transaction [7]). On the other hand, workflow systems [15], as the key technology for business process automation [11], lack sound mechanisms for reliability and correctness.

In this paper, we introduce the concept of transactional patterns a convergence concept between ATM and workflow patterns [14]. Transactional patterns combine the workflow process adequacy and the transactional processing reliability. We show in particular how we use transactional patterns to define composite services and how to ensure their reliability according to designers specific needs.

The remainder of the paper is organized as follows. Section 2 presents a motivating example showing the limits of workflow systems and ATM to ensure reliable composite services. In section 3 we detail the main elements required to model Web services compositions. Sections 4 introduce the concept of transactional patterns. In section 5, we show how we use transactional patterns to define composite services and how to ensure their reliability. Section 6 presents some related work and shows how our approach can complement outgoing current efforts while section 7 concludes.

2. MOTIVATING EXAMPLE

We consider an application for online travel arrangement (OTA for short), carried out by a composite service as illustrated in figure 1. The customer specifies its requirements for destination and hotels. The composite service launches in parallel hotel and flight booking. Then, the customer is requested to pay online. Once this is done, travel documents are sent to the customer. To deal with failures, the designers of the composite service may augment the control flow described above with a set of transactional requirements. For instance, they may require the services FB and TDU to be sure to complete and the service FB to be compensatable. Then, they may specify TDU as an alternative for $TDFE$ fails. They may also require to compensate FB when the hotel booking fails.

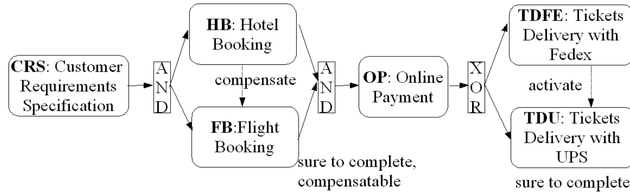

Figure 1: A composite service for online travel arrangement.

Modeling this example with ATM or workflow systems is not easy. In one hand, ATM are too rigid to enable a such control structure, and they do not support bottom-up applications design, starting from predefined business process and using pre-existing systems or services with diverse semantics [7]. On the other hand, workflow systems lack functionalities to assess that the specified transactional behavior ensure the required reliability. In our example, if the service OP may fail, causing the travel arrangement abortion, flight and hotel booking should be undone.

3. TRANSACTIONAL WEB SERVICES MODEL

In this section, we introduce our Web services composition model. We distinguish in particular between the coordination and the transactional aspects of a composite Web service (CWS for short). In one hand a CWS can be seen as a flow of autonomous and heterogenous services. On the other hand, it can be considered as a structured transaction where the component services are the subtransactions and the interactions are the dependencies.

The section 3.1 introduces the concept of a transactional Web service. We present the transactional properties we are considering and we show how we model a Web service behavior according to its transactional properties. The section 3.2, illustrates how we combine a set of transactional Web services to create a new value-added service. We show how we model the orchestration schema at different levels of abstraction. We distinguish, in particular, the `control flow` (coordination aspects) and the `transactional flow` (transactional aspect) of a CWS. The section 3.3 details the relation between the control flow and the transactional flow of a CWS.

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 *retriable, compensatable and pivot* [12]. 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 $\{r; cp; p; (r, cp); (r, p)\}$.

Every service can be associated to a life cycle statechart that models the possible statuses through which the executions of this service can go, and the possible transitions between these statuses [3]. 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 continues 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 $\mathcal{TWS}$ the set of all transactional Web services.

3.2 Transactional composite Web service: TCS

A composite Web service is a conglomeration of existing Web services working in tandem to offer a new value-added service [11]. It 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 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 in figure 1 specifies that OP will be activated after the completion of HB and FB. That means the precondition of the transition $activate()$ of OP is the completion of HB and the completion of FB.

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 $tcs = (ES \subset \mathcal{TWS}, \mathcal{P}rec)$ where ES is the set of its component Web services and $\mathcal{P}rec$ is a function that defines for each component service's external transition a precondition for its activation.*

Preconditions on services' external transitions specify for each how it reacts to the other states change and how it acts on their behaviors. Actually, the function $\mathcal{P}rec$ defines for each component service's external transition $t()$ a set of preconditions to activate it. It is worthy to note that these preconditions are **exclusive**. 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 TDU will be activated either after the completion of OP [1] (exclusively) or after the failure of $TDFE$. That means $\mathcal{P}rec$*(TDU.acti-vate())* = {*(OP.completed* $\bigwedge$ *TDU* chosen for delivery), *TDFE.failed*}.

We note $\mathcal{TCS}$ the set of all transactional composite Web services. We define the function $services$: $\mathcal{TCS} \longrightarrow \mathcal{P}^2(\mathcal{TWS})$ that returns the set of component services of a given TCS.

3.2.2 Dependencies between a TCS's component services

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 one(s). For example the precondition on the external transition *activate()* of *TDU* expresses (i) a succession relations (or dependency) between *OP* and *TDU* and (ii) an alternative relation (or dependency) between *TDFE* and *TDU*. More formally:

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 $s_2.t_2()$, noted dep$(s_1.t_1(), s_2.t_2())$, exists if the activation of $s_1.t_1()$ may fire the activation of $s_2.t_2()$.*

Dependencies express relation between services, however they do not describe precisely interactions between services. A dependency $dep(s_1.t_1(), s_2.t_2())$ does not specify when $s_2.t_2()$ will be activated (following $s_1.t_1()$ activation). $dep(s_1.t_1(), s_2.t_2())$ is defined according to $\mathcal{P}rec(s_2.t_2())$.

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. More formally and according to the definition 2:

$$depAct(s_1, s_2) \stackrel{\text{def}}{=} dep(s_1.complete(), s_2.activate()).$$

An activation dependency from s_1 to s_2 expresses only a succession relation between them. However, it does not specify when s_2 will be activated (following the termination of s_1). Regarding its definition, an activation dependency $depAct(s_1, s_2)$ is defined according to $\mathcal{P}rec(s_2.activate())$ and more precisely according to the activation condition of s_2. The activation condition of a service s (as a successor) determines when it will be activated as a successor for other(s) service(s). We note the activation condition of a service s $ActCond(s)$.

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

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. More formally and according to the definition 2:

$$depAlt(s_1, s_2) \stackrel{\text{def}}{=} dep(s_1.fail(), s_2.activate()).$$

Regarding its definition, an alternative dependency $depAlt(s_1, s_2)$ is defined according to $\mathcal{P}rec(s_2.activate())$ and more precisely according to the alternative condition of s_2. The alternative condition of a service s, $AltCond(s)$, specifies when s will be activated as an alternative of other(s) service(s).

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

Note that the activation condition of the transition $activate()$ of a service s is defined by s activation condition (as a successor), $ActCond(s)$, and by s alternative condition $AltCond(s)$:
$\mathcal{P}rec(s.activate()) = ActCond(s) \bigcup AltCond(s)$.

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. More formally and according to the definition 2:

$$depAbr(s_1, s_2) \stackrel{\text{def}}{=} dep(s_1.abort(), s_2.abort()) \bigvee dep(s_1.fail(), s_2.abort()) \bigvee dep(s_1.cancel(), s_2.abort()).$$

An abortion dependency $depAbr(s_1, s_2)$ is defined according to $\mathcal{P}rec(s_2.abort())$. $\mathcal{P}rec(s.abort())$ defines the abortion dependency of s, $AbrCond(s)$, which determines when s will be aborted.

Compensation dependency and compensation 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. More formally and according to the definition 2:

[1] where TDU is chosen for delivery
[2] $\mathcal{P}(S)$ denotes the set of all subsets of S

$$depCps(s_1, s_2) \stackrel{\text{def}}{=} dep(s_1.fail(), s_2.compensate()) \bigvee dep(s_1.compensate(), s_2.compensate()).$$

A compensation dependency $depCps(s_1, s_2)$ is defined according to $\mathcal{P}rec(s_2.compensate())$. $\mathcal{P}rec(s.compensate())$ defines the compensation condition of s, $CpsCond(s)$, which determines when s will be compensated.

The composite service in figure 1 defines a compensation dependency from HB to FB such that FB will be compensated when HB fails. That means *CpsCond(FB)* = {*HB.failed*}.

Cancelation dependency and cancelation 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. More formally and according to the definition 2:

$$depCnl(s_1, s_2) \stackrel{\text{def}}{=} dep(s_1.fail(), s_2.cancel())$$

A cancelation dependency $depCnl(s_1, s_2)$ is defined according to $\mathcal{P}rec(s_2.fail())$. $\mathcal{P}rec(s.fail())$ defines the cancelation condition of s, $CnlCond(s)$, which specifies when s will be canceled.

3.2.3 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. Formally, we define a control flow as a TCS of which the only dependencies are activation dependencies.

DEFINITION 3. *A* `control flow` *is a TCS,* $cf = (ES, \mathcal{P}rec)$ *such that* $\forall s \in ES$ $CondAlt(s) = false$; $CondCps(s) = false$; $CondCnl(s) = false$.

We note $\mathcal{CF}low$ 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.*

$$\begin{array}{llcl} getCFlow: & \mathcal{TCS} & \longrightarrow & \mathcal{CF}\text{low} \\ & sc = (ES, \mathcal{P}rec) & \longmapsto & fc = (ES', \mathcal{P}'rec) \end{array}$$

such that $ES' = ES$ *and* $\forall s \in ES$ $\mathcal{P}'rec(s.activer()) = CondAct(s)$; $\mathcal{P}'rec(s.annuler()) = false$; $\mathcal{P}'rec(s.compenser()) = false$.

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 of which the only dependencies are transactional dependencies.

DEFINITION 5. *A* `transactional flow` *is a TCS,* $tf = (ES, \mathcal{P}rec)$ *such that* $\forall s \in ES$ $CondAct(s) = false$.

We note $\mathcal{TF}low$ 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.*

$$\begin{array}{llcl} getTFlow: & \mathcal{TCS} & \longrightarrow & \mathcal{TF}\text{low} \\ & sc = (ES, \mathcal{P}rec) & \longmapsto & fc = (ES', \mathcal{P}'rec) \end{array}$$

such that $ES' = ES$ *and* $\forall s \in ES$ $\mathcal{P}'rec(s.activer()) = CondAlt(s)$.

A TCS, tcs, is well defined by its control flow, $getCFlow(tcs)$, and its transactional flow $getTFlow(tcs)$.

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

The transactional flow is tightly related to the 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 TDU as an alternative to $TDFE$ because (according to the `XOR-split` control flow operator) they are defined on exclusive branches. Similarly, it is possible to define a compensation dependency from HB to FB because (according to the `AND-join` control flow operator) the failure of HB requires the compensation of the partial work already done which is the flight booking.

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

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

Back to our example, according to the OTA service control flow FB may eventually be compensated (i) either after the failure (exclusively) or the compensation of OP (ii) (exclusively) or after the failure of HB. That means *ptCpsCond(FB)* = {*OP.failed, OP.compensated, HR.failed*}.

Given a control flow cf, several TCSs can be defined according to it. Each of these TCS will adopt cf as its control flow and will extend it by a transactional flow included in cf potential transactional. More formally, given a TCS tcs the following holds:

$$\forall s \text{ a component service of } tcs,\ CpsCond(s) \in PtCpsCond(s),$$
$$CpsCnl(s) \in PtCnlCond(s) \text{ and}$$
$$AltCond(s) \in PtAltCond(s)$$

For example, the transactional flow of the OTA service is included in its potential transactional flow. For instance the compensation condition of FB is the failure of HB which is included in its potential compensation condition.

As a recapitulation of this section it is worthy to maintain that:

- A TCS is well defined by its control flow and its transactional flow.
- Defining a TCS control flow returns to define for each component service s, its activation condition $ActCond(s)$.
- Defining a TCS transactional flow returns to define for each service s, its transactional properties, its compensation condition $CpsCond(s)$, its cancelation condition $CnlCond(s)$, and its alternative condition $AltCond(s)$.
- A TCS transactional flow is included in the TCS potential transactional flow.
- A TCS potential transactional flow depends on the TCS control flow. A TCS potential transactional flow is defined by the potential compensation, cancelation and alternative conditions of each component service. These potential conditions are defined w.r.t the TCS control flow.

4. TRANSACTIONAL PATTERNS

In this section, we introduce the concept of transactional pattern, a new paradigm we propose to ensure reliable Web services compositions. Transactional patterns extend workflow patterns with transactional dependencies, thus allowing to bridge their transactional lack.
As defined in [6], a pattern "is the abstraction from a concrete form which keeps recurring in specific non arbitrary contexts". Regarding that, a workflow pattern [14] can be seen as an abstract description of a recurrent class of interactions. For example, the `AND-join` pattern [14] (see figure 2.b) describes an abstract services interactions as follows: *a service is activated after the completion of several other services*.
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 modelling allow also modular and local processing.
In the section 4.1 we present the workflow patterns and put them in the context of our TCS model. Then we show, in the section 4.2, how we extend them to define transactional patterns.

4.1 Workflow patterns

Regarding our TCS model, the basic workflow patterns [14] 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 7. *A control flow pattern pat, is a function pat: $\mathcal{P}(\mathcal{TWS}) \longrightarrow \mathcal{CF}low$, that returns a control flow $pat(S)$ given a set of transactional services S.*

We note $\mathcal{P}attern$ the set of all control flow patterns. 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* [14]. Due to the lack of spaces, we put emphasis on the *AND-split*, *AND-join* and *XOR-split* patterns (we are using in our illustrative example).

4.1.1 AND-split pattern

[14] defines an `AND-split` pattern as a point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order. According to our approach, an `AND-split` pattern is a function that specifies that a set of services are activated after the completion of another service.

DEFINITION 8. *We define the AND-split pattern as the function:*
`AND-split`: $\mathcal{P}(\mathcal{TWS}) \longrightarrow \mathcal{CF}low$
$S = \{s_0, s_1, \dots, s_n\} \longmapsto cf = (ES, \mathcal{P}rec)$
such that
• $ES = \{s_0, s_1, \dots, s_n\}$,
• $ActCond(s_0)$ = external event to cf
• $\forall i, 1 \le i \le n$ $ActCond(s_i)$ = $s_0.completed$.

Figure 2.a illustrates the control flow result of the application of `AND-split` pattern to the set of services $\{CRS, HB, FB\}$.

4.1.2 AND-join pattern

[14] defines an `AND-join` pattern as a point in the workflow process where multiple parallel subprocesses/activities converge into one single thread of control, thus synchronizing multiple threads. According to our approach, an `AND-Join` pattern is a function that specifies that a service is activated after the completion of a set of other services.

DEFINITION 9. *We define the AND-join pattern as the function:*
AND-join: $\mathcal{P}(\mathcal{TWS}) \longrightarrow \mathcal{CF}low$
$S = \{s_1, \dots, s_n, s_0\} \longmapsto cf = (ES, \mathcal{P}rec)$
such that
• $ES = \{s_0, s_1, \dots, s_n\}$,
• $ActCond(s_0) = \bigwedge_{i=1..n} s_i.completed$
• $\forall i, 1 \le i \le n$ $ActCond(s_i)$ = external event to cf.

Figure 2.b illustrates the control flow result of the application of `AND-join` pattern to the set of services $\{HB, FB, OP\}$.

4.1.3 XOR-split pattern

[14] defines an `XOR-split` pattern as a point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen. According to our approach, an `XOR-split` pattern is a function that specifies that a service, among many others, is activated after the completion of another service.

DEFINITION 10. *We define the XOR-split pattern as the function:*
XOR-split: $\mathcal{P}(\mathcal{TWS}) \longrightarrow \mathcal{CF}low$
$S = \{s_0, s_1, \dots, s_n\} \longmapsto cf = (ES, \mathcal{P}rec)$
such that
• $ES = \{s_0, s_1, \dots, s_n\}$,

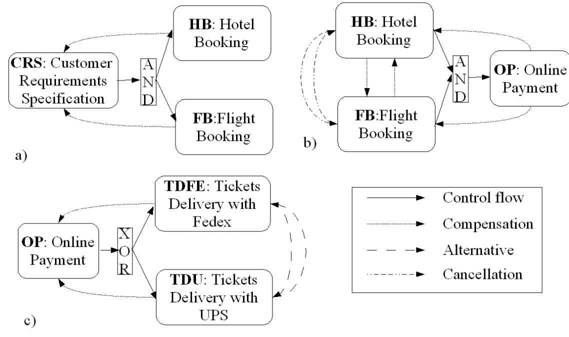

Figure 2: AND-split, AND-join and XOR-split patterns and their corresponding potential functions applied to a given sets of services.

• $ActCond(s_0)$ = external event to cf
•$\forall i, 1 \leq i \leq n$ $ActCond(s_i) = s_0.completed \bigwedge c_i$ | *there is always a one only c_i evaluated to true after the completion of s_0.*

Figure 2.c illustrates the control flow result of the application of `XOR-split` pattern to the set of services $\{OP, TDFE, TDU\}$.

4.2 Extending workflow patterns with transactional dependencies

A workflow pattern pat defines a control flow $pat(S)$ given a set of services. As all control flow, $pat(S)$ 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 $pat(S)$. $potential_{pat}$ defines for each service its potential compensation, alternative and cancelation conditions according to the semantics of the control flow defined by pat. In the following, we detail the potential functions of the patterns `AND-split`, `AND-join`, and `XOR-split`.

4.2.1 AND-split potential function

The potential function of the pattern `AND-split`, $potential_{AND-split}$ defines for a given set of services $\{s_0, s_1, \ldots, s_n\}$ the following transactional dependencies: a compensation dependency from s_i ($1 \leq i \leq n$) to s_0 according to the synchronization policy of s_1, …, s_n.

DEFINITION 11. *The potential function of the pattern* AND-split *is defined as follows:*

$$potential_{AND-split}: \quad \mathcal{P}(\mathcal{TWS}) \longrightarrow \mathcal{TF}low$$
$$\{s_0, s_1, \ldots, s_n\} \longmapsto tf = (ES, \mathcal{P}rec)$$

where tf is the potential transactional flow of `AND-split`(S) *and such that $ES = S$ and $\mathcal{P}rec$ is defined as follows:*
• $PtAltCond(s_i) = \emptyset\ \forall 0 \leq i \leq n$
• $PtCpsCond(s_0) = \{PtCpsCond(s_i) \mid i = 1..n\}$
• $\forall\ 1 \leq i \leq n$ $PtCpsCond(s_i)$ = *defined by the used synchronization pattern,*
• $PtCnlCond(s_0) = false$,
• $\forall\ 1 \leq i \leq n$ $PtCnlCond(s_i)$ = *defined by the used synchronization pattern*

Figure 2.a, illustrates the potential function of the pattern `AND-split` applied to *(CRS,HB,FB)*.

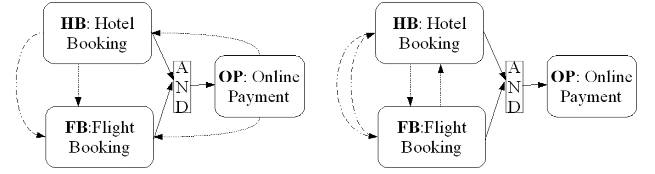

Figure 3: Two transactional patterns derived from the `AND-join` pattern.

4.2.2 AND-join potential function

The potential function of the pattern `AND-join`, $potential_{AND-join}$ defines for a given set of services $\{s_1, \ldots, s_n, s_0\}$ the following transactional dependencies: each service s_i will be compensated or canceled (according to its current state) when a service s_j fails (where $1 \leq i, j \leq n$ and $i \neq j$). Each service s_i ($1 \leq i \leq n$) will be compensated when s_0 fails or is compensated.

DEFINITION 12. *The potential function of the pattern* AND-join *is defined as follows:*

$$potential_{AND-join}: \quad \mathcal{P}(\mathcal{TWS}) \longrightarrow \mathcal{TF}low$$
$$\{s_1, \ldots, s_n, s_0\} \longmapsto tf = (ES, \mathcal{P}rec)$$

where tf is the potential transactional flow of `AND-join(S)` *and such that $ES = S$ and $\mathcal{P}rec$ is defined as follows:*
• $PtAltCond(s_i) = false\ \forall 0 \leq i \leq n$
• $PtCpsCond(s_0)$ = external event,
• $\forall\ 1 \leq i \leq n$ $PtCpsCond(s_i)$ =$\{s_0.failed$, $s_0.compensated$, $s_j.failed \mid 1 \leq j \leq n, j \neq i\}$
• $PtCnlCond(s_0) = false$,
• $\forall\ 1 \leq i \leq n$ $PtCnlCond(s_i) = \{s_j.failed \mid 1 \leq j \leq n, j \neq i\}$.

Figure 2.b illustrates the potential function of the pattern `AND-join` applied to *(HB,FB,OP)*.

4.2.3 XOR-split potential fucntion

The potential function of the pattern `XOR-split`, $potential_{XOR-split}$ defines for a given set of services $\{s_0, s_1, \ldots, s_n\}$ the following transactional dependencies: each service s_i is an alternative for s_j where $1 \leq i, j \leq n$ and $i \neq j$. Each service s_i (1≤i≤n) will compensate s_0 when it fails or is compensated.

DEFINITION 13. *The potential function of the pattern* XOR-split *is defined as follows:*

$$potential_{XOR-split}: \quad \mathcal{P}(\mathcal{WS}) \longrightarrow \mathcal{TF}low$$
$$\{s_0, s_1, \ldots, s_n\} \longmapsto tf = (ES, \mathcal{P}rec)$$

where tf is the potential transactional flow of `XOR-split`(S) *and such that $ES = S$ and $\mathcal{P}rec$ is defined as follows:*
• $PtAltCond(s_0) = false$
• $\forall\ 1 \leq i \leq n$ $PtAltCond(s_i) = \{s_j.failed \mid j = 1..n, i \neq j\}$
• $PtCpsCond(s_0) = \{s_j.failed \mid j = 1..n, i \neq j\}$
• $\forall\ 1 \leq i \leq n$ $PtCpsCond(s_i)$ = external event to *tf,*
• $PtCnlCond(s_0) = false$,
• $\forall\ 1 \leq i \leq n$ $PtCnlCond(s_i) = false$

Figure 3.c illustrates the potential function of the pattern `XOR-split` applied to *(PL,SDF,SDD)*).

$\mathcal{G}_i$	$\longrightarrow$	s | $\texttt{Sequence}(\mathcal{G}_i, \mathcal{G}_j)\ \mathcal{A}$ |
		$\texttt{AND-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \mathcal{B}$ |
		$\texttt{OR-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \mathcal{C}$ |
		$\texttt{XOR-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \mathcal{D}$ |
$\texttt{Sequence}(\mathcal{G}_i, \mathcal{G}_j)\ \mathcal{A}$	$\longrightarrow$	$\texttt{Sequence}(\mathcal{G}_i, \mathcal{G}_j)\ \texttt{Sequence}(\mathcal{G}_j, \mathcal{G}_k)\ \mathcal{A}$ |
		$\texttt{Sequence}(\mathcal{G}_i, \mathcal{G}_j)\ \texttt{AND-split}(\mathcal{G}_j, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \mathcal{B}$ |
		$\texttt{Sequence}(\mathcal{G}_i, \mathcal{G}_j)\ \texttt{OR-split}(\mathcal{G}_j, b_1, \ldots, b_n)\ \mathcal{C}$ |
		$\texttt{Sequence}(\mathcal{G}_i, \mathcal{G}_j)\ \texttt{XOR-split}(\mathcal{G}_j, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \mathcal{D}$ |
		$\texttt{Sequence}(\mathcal{G}_i, \mathcal{G}_j)$
$\texttt{AND-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \mathcal{B}$	$\longrightarrow$	$\texttt{AND-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \texttt{AND-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \mathcal{E}$ |
		$\texttt{AND-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \texttt{OR-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \mathcal{E}$ |
		$\texttt{AND-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \texttt{XOR-join}\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \mathcal{E}$ |
		$\texttt{AND-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \texttt{m-out-of-n}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \mathcal{E}$ |
$\texttt{OR-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \mathcal{C}$	$\longrightarrow$	$\texttt{OR-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \texttt{OR-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \mathcal{E}$ |
		$\texttt{OR-split}(\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n)\ \texttt{XOR-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \mathcal{E}$ |
$\texttt{XOR-split}(\mathcal{G}_0, \ldots, \mathcal{G}_n)\ \mathcal{D}$	$\longrightarrow$	$\texttt{XOR-split}(\mathcal{G}_0, \ldots, \mathcal{G}_n)\ \texttt{XOR-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \mathcal{E}$ |
		$\texttt{XOR-split}(\mathcal{G}_0, \ldots, \mathcal{G}_n)$
$\texttt{-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \mathcal{E}$	$\longrightarrow$	$\texttt{-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \texttt{Sequence}(\mathcal{G}_{n+1}, \mathcal{G}_0)\ \mathcal{A}$ |
		$\texttt{-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \texttt{AND-split}(\mathcal{G}_{n+1}, \mathcal{G}_{n+2}, \ldots, \mathcal{G}_{n+p})\ \mathcal{B}$ |
		$\texttt{-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \texttt{OR-split}(\mathcal{G}_{n+1}, \mathcal{G}_{n+2}, \ldots, \mathcal{G}_{n+p})\ \mathcal{C}$ |
		$\texttt{-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})\ \texttt{XOR-split}(\mathcal{G}_{n+1}, \mathcal{G}_{n+2}, \ldots, \mathcal{G}_{n+p})\ \mathcal{D}$ |
		$\texttt{-join}(\mathcal{G}_1, \ldots, \mathcal{G}_n, \mathcal{G}_{n+1})$

Table 1: Left contextual grammar defining the language of consistent control flows.

4.2.4 Definition

A transactional pattern derived from a workflow pattern pat is an instance of pat extended by a transactional flow included in its potential transactional flow.

DEFINITION 14. *Let pat a pattern, we call a transactional pattern derived from pat each TCS cs such that:*

$$getCFlow(cs) = pat(services(cs)) \text{ and}$$
$$getTFlow(cs) \subseteq potential_{pat}(services(cs))$$

Many transactional patterns can be derived from the same control flow pattern. Figure **??** illustrates two transactional patterns derived form the `AND-join` pattern. Both extend an instance of the `AND-join` pattern with a transactional flow included in its potential transactional flow.

5. RELIABLE WEB SERVICES COMPOSITIONS USING TRANSACTIONAL PATTERNS

We use transactional patterns as the basic brick to specify composite Web services. A TCS can be defined as a set of transactional patterns connected together (having common component services) in an eventual nested way. Figure 4 illustrates how we can specify the OTA service as the following transactional patterns composition: $trans_{AND-split}\ trans_{AND-join}\ trans_{XOR-split}$.

Connecting together a set of transactional patterns can lead to a control and transactional (flow) inconsistency. Control flow inconsistency can raise, for instance, when an `XOR-split` is followed by an `AND-join`. Transactional inconsistency can raise, for instance, when a component service can eventually fail, causing the entire TCS abortion, without compensating the partial work already done.

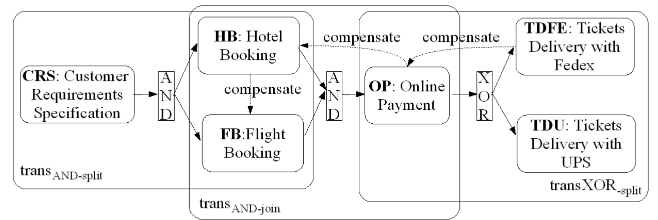

Figure 4: The OTA service defined as a connection of a set of transactional patterns.

A TCS is reliable if its control flow and transactional flow are consistent. For example, the OTA service defined in figure 4 is not retriable since OP can eventually fail without compensating FB. In the following we show how we ensure a TCS reliability.

5.1 Ensuring control flow consistency

To ensure the control flow consistency, we propose the left contextual grammar described in table 1. This grammar defines the language of consistent control flows. It ensures consistent connection between the patterns. It postulates that (i) a consistent control flow should start with either a sequence, or a split pattern, (ii) a `sequence` pattern can be followed by any split pattern, (iii) an `AND-split` pattern can be followed by any join pattern, (iv) an `OR-split` pattern can be followed by an `OR-join` or an `XOR-join` pattern, and (v) an `XOR-split` pattern can be followed only by an `XOR-join` pattern. In addition, a component service in a given TCS can be itself a composite service where its control flow is consistent (respects the above grammar); thus allowing to use patterns in a nested way inside a composition.

5.2 Ensuring transactional flow consistency

The first step to ensure consistent transactional flow consists in determining component services transactional properties. If these transactional properties are not known, we apply the following rules (in the given order) to compute them:
• each service is by default retriable and pivot,
• each service target of a compensation dependency is compensatable,
• each service source of a compensation, cancelation, or alternative dependency is not retriable.

By applying these rules to the OTA services shown in figure 4 we can deduce that TDU is retriable contrary to OP and $TDFE$. We can also deduce that HB and FB are compensatable.

The second step consists in ensuring an `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 composite service shown in figure 4 is not `intuitively valid` since it does not respect, among others, the property P_1 for the service $TDFE$ and the property P_2 for the service OP.

We propose a set of rules to 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. $\forall\, ptAltCond_i(s) \in AltCond(s)$,
   $\Diamond(ptAltCond_i(s)) \bigwedge ptAltCond_i(s) \notin AltCond(s) \Rightarrow$
   $AltCond(s) = AltCond(s) \bigcup ptAltCond_i(s)$.
2. $\forall\, ptCpsCond_i(s) \in ptCpsCond(s)$,
   $\Diamond(ptCpsCond_i(s)) \bigwedge ptCpsCond_i(s) \notin CpsCond(s) \Rightarrow$
   (a) s must be compensatable and
   (b) $CpsCond(s) = CpsCond(s) \bigcup ptCpsCond_i(s)$.
3. $\forall\, ptCnlCond_i(s) \in ptCnlCond(s)$,
   $\Diamond(ptCnlCond_i(s)) \bigwedge ptCnlCond_i(s) \notin CnlCond(s) \Rightarrow$
   $CnlCond(s) = CnlCond(s) \bigcup ptCnlCond_i(s)$.

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 TDU, $TDFE.failed$ is eventually true (since $TDFE$ is not retriable) and is not considered as one of its alternative conditions. By applying this rule we can generate the suggestion to add an alternative dependency from $TDFE$ to TDU. Similarly by applying the second rule, we can also generate the suggestion to add a compensation dependency from OP to FB.

It is worthy to note that during the first and the second step the designers have the final decision about component services transactional properties or which suggestions consider and which ones refuse. Like this, our approach allows to take into account designers specific requirements that may violate the well behavior properties introduced above.

6. RELATED WORK

We classify the current related technologies in two classes, workflow based like WSBPEl [2] and WS-CDL [8] and transactional based like WS-AtomicTransaction [9], WS-BusinessActivity [10] and WS-TXM (Acid, BP, LRA) [5].

We can say that these technologies are standardized versions (using XML as an exchange format and the Web as an invocation infrastructure) of the workflow approach or ATM adapted to work in a peer to peer environment. Consequently, they inherit the limitation of these two approaches: ensure reliability on behalf of process adequacy or the opposite. We believe that our approach can complement these efforts.

In one hand, WSBPEL and WS-CDL follow a workflow approach to define services compositions and services choreographies. Like workflow systems these two language meet the business process need in term of control structure. However, they are unable to ensure reliability especially according to the designers specific needs. Our approach can be used on top of them. We can use our approach to define reliable compositions. Then the defined model can be described either using WSBPEL or WS-CDL. Obviously, we need to extend these two languages to support cancelation and alternative interactions.

On the other hand, WS-AtomicTransaction, WS-BusinessActivity and WS-TXM rely on ATM to define transactional coordination protocols. Like ATM these protocols are unable in most cases to model Business process due to their limited control structure. Our approach allows to extend these protocols to support complex structure while preserving reliability. Indeed, a transactional pattern taken alone as a composition of transactional patterns can be considered as a transactional protocol.

A one important step to integrate our approach is to extend Web services description to describe their transactional properties. This is possible thanks to the sematic Web services languages. For instance, the transactional properties we are considering can be easily described in the non functional block of a WSMO service [13].

7. CONCLUSION

In this paper, we propose an approach to ensure reliable Web services compositions. The main idea is to combine the process adequacy of workflow systems and the reliability of transactional processing. We introduce the concept of transactional patterns. Transactional patterns extend workflow patterns with transactional dependencies, thus allowing to bridge their transactional lack. We show how we use them to define composite Web services and how we ensure their reliability.

The main contribution of our approach is the convergence of workflow approach and transactional processing. Our approach presents also the advantages of any pattern based modeling like simplicity, practice and modularity. As a future work we plan to consider more workflow patterns especially those supporting repetitive processing and multi instantiation. We aim also to integrate the concept of scope of services to allow different processing levels.

8. REFERENCES

[1] G. Alonso, D. Agrawal, and A. E. Abbadi. Process Synchronisation in Workflow Management Systems. In *8th IEEE Symposium on Parallel and Distributed Processing (SPDS'97)*, New Orleans, Louisiana, October 1996.

[2] I. BEA and Microsoft. Business process execution language for web services (bpel4ws). 2003.

[3] S. Bhiri, O. Perrin, and C. Godart. Ensuring required failure atomicity of composite web services. In *WWW*, pages 138–147, 2005.

[4] A. Elmagarmid. *Transaction Models for Advanced Database Applications*. Morgan-Kaufmann, 1992.

[5] D. B. et al. Web services transaction management (ws-txm) version 1.0. In *Arjuna, Fujitsu, IONA, Oracle, and Sun*, July 28 2003.

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

[7] N. Gioldasis and S. Christodoulakis. Utml: Unified transaction modeling language. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering*, pages 115–126. IEEE Computer Society, 2002.

[8] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon. Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10, October 2004.

[9] D. Langworthy and al. Web services atomic transaction (ws-atomictransaction).

[10] D. Langworthy and al. Web services business activity framework (ws-businessactivity).

[11] B. Medjahed, B. Benatallah, A. Bouguettaya, A. H. H. Ngu, and A. K. Elmagarmid. Business-to-business interactions: issues and enabling technologies. *The VLDB Journal*, 12(1):59–85, 2003.

[12] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. A transaction model for multidatabase systems. In *ICDCS*, pages 56–63, 1992.

[13] 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*.

[14] W. M. P. van der Aalst, P. Barthelmess, C. Ellis, and J. Wainer. Workflow Modeling using Proclets. In O. Etzion and P. Scheuermann, editors, *5th IFCIS Int. Conf. on Cooperative Information Systems (CoopIS'00)*, number 1901 in LNCS, pages 198–209, Eilat, Israel, September 6-8, 2000. Springer-Verlag.

[15] W. M. P. van der Aalst and K. M. van Hee. *Workflow Management: models, methods and tools*. Cooperative Information Systems. MIT Press, 2002.