Extending workflow patterns with transactional dependencies to define reliable composite Web services

Sami Bhiri, Olivier Perrin, Claude Godart

Extending workflow patterns with transactional dependencies to define reliable composite Web services.

Sami Bhiri, Olivier Perrin, Claude Godart
LORIA-University Henri Poincaré, Nancy, France
claude.godart@loria.fr

***Abstract*— In this paper, we introduce the concept of a *transactional pattern* for specifying flexible and reliable composite Web services. A transactional pattern is a convergence concept between workflow patterns and advanced transactional models; thus it combines workflow flexibility and transaction reliability. Designers can simply connect together transactional patterns to define a composite Web service. In relation, we provide techniques to ensure control and transactional consistency of such services.**

I. Introduction

SOA provides the key enabling infrastructure required to integrate supply chains and end-to-end e-Business applications at a variety of different levels, from applications and data, to business processes, and across (and within) organizations. Thus, the applications built by composing or coordinating the execution of several Web services can be very complex for the following reasons: there are many components services, the execution can last a long time, the Web services environment is loosely coupled and asynchronous by nature. Because machines may fail, processes may be cancelled, or services may be moved or withdrawn, handling potential failures is mandatory. However, traditional transactional model has been proven to be unsuitable for long running, asynchronous, and decentralized activities.

An increasing number of applications are being constructed by combining or coordinating the execution of multiple Web services, each of which may represent an interface to a different underlying technology. The resulting applications can be very complex in structure, with complex relationships between their constituent services. Furthermore, the execution of such an application may take a long time to complete, and may contain long periods of inactivity, often due to the constituent services requiring user interactions. In the loosely coupled environment represented by Web services, long running applications will require support for recovery and compensation, because machines may fail, processes may be cancelled, or services may be moved or withdrawn. Web services transactions also must span multiple transaction models and protocols native to the underlying technologies onto which the Web services are mapped.

However, handling failures using the traditional transactional model for long running, asynchronous, and decentralized activities has been proven to be unsuitable. Advanced Transaction Models (ATMs) [1] have been proposed to manage failures, but, although powerful and providing a nice theoretical framework, ATMs are too database-centric, limiting their possibilities and scope [2] in this context (e.g. their inflexibility to incorporate different transactional semantics as well as different behavioural patterns into the same structured transaction [3]).

In the same time, workflow [4] has became gradually a key technology for business process automation [5], providing a great support for organizational aspects, user interface, monitoring, accounting, simulation, distribution, and heterogeneity.

In this paper, we propose an approach to specify and orchestrate flexible and reliable Web services compositions based on the concept of *transactional patterns* which combines workflow flexibility and transactional reliability.

Section II presents a motivating example. Section 3 and 4 specify the transactional composite Web services and workflow patterns concepts. Section 5 introduces the transactional patterns, and shows how to use them to specify composite Web services, while guaranteeing the transactional consistency. Section VI presents related work, while section VII concludes.

II. Motivating example

We consider an application for online travel arrangement, 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 want to 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. They may also require the service FB to be compensatable (when an hotel booking is cancelled, or when it fails). Then, they may specify that service TDU is an alternative for $TDFE$.

Modeling this example with ATM or workflow systems is not easy because 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 [3]. On the other hand, workflow systems lack functionalities to assess

Proceedings of the Advanced International Conference on Telecommunications and
International Conference on Internet and Web Applications and Services (AICT/ICIW 2006)
0-7695-2522-9/06 $20.00 © 2006 IEEE

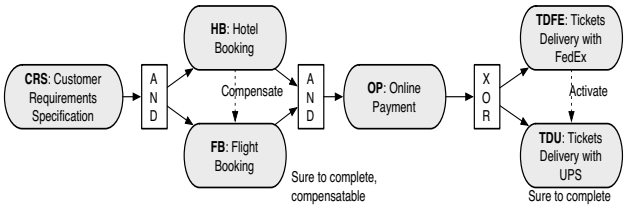

Fig. 1. A composite service for online travel arrangement.

that the specified transactional behavior ensure the required reliability. In our example, if the service OP fails, causing the travel arrangement abortion, flight and hotel booking should be undone.

III. Transactional composite Web services

A. Transactional Web service

A Web service is a self-contained modular program built with XML, SOAP, WSDL and UDDI specifications that can be discovered and invoked across the Internet [6], [7]. A *transactional Web service* is a Web service that emphasizes transactional properties for its characterization and correct usage.

The main transactional properties of a Web service we are considering are *retriable, compensatable, pivot* [8]. A service s is said to be ***retriable*** (s^r) if it is sure to complete after a finite number of activations. s is said to be ***compensatable*** (s^{cp}) if it offers compensation policies to semantically undo its effects. Then, s is said to be ***pivot*** (s^p) if once it successfully completes, its effects remains for ever and cannot be undone. A service can combine properties; the set of all possible combinations is $\{\emptyset; r; cp; p; (r, cp); (r, p)\}$.

For each service, a set of operations is available, depending the transactional properties of the service. For instance, a service defined as *pivot* has a minimal set *abort(), activate(), cancel(), fail(), complete()* allowing respectively its abortion before activation, its activation, its cancellation during its execution, its failure and its successful completion. A compensatable service has in addition a *compensate()* operation for its compensation. A retriable service has a *retry()* operation allowing to activate it after each failure.

B. Transactional composite Web service

A composite Web service is a conglomeration of existing Web services working in tandem to offer a new value-added service [5]. It coordinates a set of services as a cohesive unit of work to achieve common goals. A *Transactional Composite (Web) Service* (TCS) emphasizes transactional properties for composition and synchronization of component Web services. It takes advantage of services transactional properties to specify mechanisms for failure handling and recovery. A TCS defines orchestration between its services using dependencies to specify how services are coupled and how the behavior of some given services influences the behavior of some others. These dependencies are used to express the relationships (sequence, alternative, compensation,...) that exist between component services.

A dependency specifies the relationships existing between services. It does not specify when a service is executed, but rather how the behavior of a service impacts another service. Thus, a dependency defines when a service will be activated, aborted, cancelled, or compensated. In the composite service illustrated in figure 1, the service OP will be activated only after the completion of both HB and FB. This means that it exists an activation dependency between these services, and that the precondition associated to the $activate()$ operation of OP is: $HB.completed \wedge FB.completed$.

C. Types of dependencies

In our proposition, we consider different types of dependencies: activation, alternative, abortion, compensation, cancellation. We classify these dependencies in two classes: activation dependencies and transactional dependencies (compensation, cancellation and alternative).

1) Activation dependency: It exists an activation dependency between a service s_1 and a service s_2 if the completion of s_1 can fire the activation of s_2. An activation condition $Activate_{Cond}(s_2)$ is associated to s_2, where $Activate_{Cond}(s)$ specifies when a service s will be activated.

In our example (figure 1), activation dependencies between HB and OP, and between FB and OP are defined such that OP will be activated after the completion of HB and FB. That means that $Activate_{Cond}(OP) = HB.completed \wedge FB.completed$.

2) Alternative dependency: It exists an alternative dependency between a service s_1 and a service s_2 if the failure of s_1 can fire the activation of s_2. An alternative condition $Alternative_{Cond}(s_2)$ is associated to s_2, where $Alternative_{Cond}(s)$ specifies when a service s will be activated (as an alternative of an other service).

In figure 1, an alternative dependency exists between $TDFE$ and TDU such that TDU will be activated when $TDFE$ fails. That means that $Alternative_{Cond}(TDU) = TDFE.failed$.

3) Abortion dependency: It exists an abortion dependency between a service s_1 and a service s_2 if the failure, the cancellation or the abortion of s_1 can fire the abortion of s_2. An abortion condition $Abortion_{Cond}(s_2)$ is associated to s_2, where $Abortion_{Cond}(s_2)$ specifies when s will be aborted after the failure, the cancellation, or the abortion of other(s) service(s).

4) Compensation dependency: It exists a compensation dependency between a service s_1 and a service s_2 if the the failure or the compensation of s_1 can fire the compensation of s_2. A compensation condition $Compensation_{Cond}(s2)$ is associated to s_2, where $Compensation_{Cond}(s)$ specifies when s will be compensated after the failure or the compensation of other(s) service(s).

In figure 1, a compensation dependency exists between HB and FB such that FB will be compensated when HB fails. That means that $Compensation_{Cond}(FB) = HB.failed$.

Proceedings of the Advanced International Conference on Telecommunications and
International Conference on Internet and Web Applications and Services (AICT/ICIW 2006)
0-7695-2522-9/06 $20.00 © 2006 IEEE

5) Cancellation dependency: It exists a cancellation dependency between a service s_1 and a service s_2 if the failure of s_1 can fire the cancellation of s_2. A cancellation condition $Cancellation_{Cond}(s)$ is associated to s_2, where $Cancellation_{Cond}(s)$ specifies when s will be cancelled after the failure other(s) service(s).

In figure 1, a cancellation dependency could exist between HB and FB such that FB will be cancelled when HB fails. That means that $Cancellation_{Cond}(FB) = OP.failed$.

6) Constraints upon dependencies: As said before, we distinguish between activation dependencies and transactional dependencies. However, it is clear that the two classes of dependencies are not independent. Thus, we can express the following list of constraints:

- an abortion dependency between s_1 and s_2 exists if and only if there is an activation dependency between s_1 and s_2.
- a compensation dependency between s_1 and s_2 exists if and only if there is an activation dependency between s_2 and s_1, or s_1 and s_2 execute in parallel and are synchronized.
- a cancellation dependency between s_1 and s_2 exists if and only if s_1 and s_2 execute in parallel and are synchronized.
- an alternative dependency between s_1 and s_2 exists if and only if s_1 and s_2 are exclusive.

D. Control flow and transactional flow of a TCS

Using the different classes of dependencies, we separate the TCS control flow and the TCS transactional flow. The **control flow** specifies the partial ordering of component services activations, and it is defined as the set of the TCS activation dependencies.

Definition 1: The control flow of a TCS is a TCS in which, for each component service s, the following dependencies are: $Alternative_{Cond}(s) = \emptyset$, $Compensation_{Cond}(s) = \emptyset$ and $Cancellation_{Cond}(s) = \emptyset$. We note $\mathcal{CF}low$ the set of all control flows.

The **transactional flow** specifies interactions for failures handling and recovery, and it is defined as the set of its transactional dependencies (compensation, cancellation and alternative).

Definition 2: The transactional flow of a TCS is a TCS in which, for each component service s, $Activation_{Cond}(s) = \emptyset$. We note $\mathcal{TF}low$ the set of all transactional flows.

As underlined in the previous paragraph, the set of transactional dependencies depends on the set of activation dependencies. Thus, a *transactional flow* is always defined according to a given *control flow*.

IV. Workflow patterns

As defined in [9], a pattern "is the abstraction from a concrete form which keeps recurring in specific non arbitrary contexts". A workflow pattern [10] can be seen as an abstract description of a recurrent class of interactions based on activation dependencies. For example, the *AND-join* pattern (see figure 2.b) describes an abstract services orchestration by specifying services interactions as follows: *a service is activated after the completion of several other services*. Thus, a pattern explicitly defines activation dependencies (i.e. the control flow) between a given set of services.

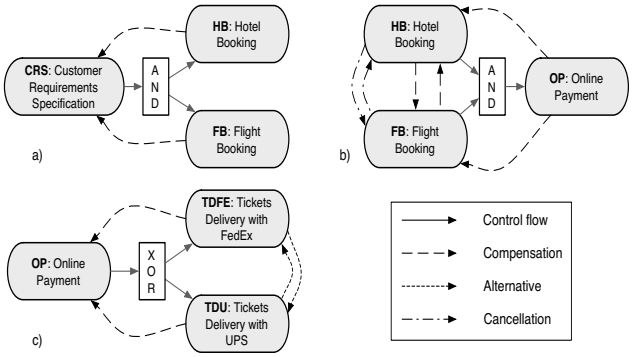

Fig. 2. AND-split, AND-join, and XOR-split patterns and their corresponding potential dependencies.

A. Workflow patterns in terms of control flows

Given the definitions of section III, we can expressed the control flow of a workflow pattern in terms of activation dependencies. In this paper, due to the lack of space, we put emphasis on the following three patterns *AND-split*, *AND-join*, and *XOR-split* to explain and illustrate our approach, but the concepts presented here can be applied to the other patterns[1]. As an example, figure 1 illustrates the patterns *AND-split* applied to *(CRS, HB, FB)*, *AND-join* applied to *(HB, FB, OP)*, and *XOR-split* applied to *(OP, TDFE, TDU)*.

[10] 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.

Definition 3: We define the AND-split pattern as the function:

$AND - split(S) : \quad S = \{s_0, s_1, \ldots, s_n\} \longrightarrow \mathcal{CF}low$ such that $\forall i,\ 1 \leq i \leq n,\ Activation_{Cond}(s_i) = s_0.completed$.

By analogy, the AND-join pattern, defined as a point in the workflow process where multiple parallel subprocesses/activities converge into one single thread of control, thus synchronizing multiple threads, can be specified as follows:

Definition 4: We define the AND-join pattern as the following function:

$AND - join(S) : \quad S = \{s_1, \ldots, s_n, s_0\} \longrightarrow \mathcal{CF}low$ such that $Activation_{Cond}(s_0) = \bigwedge_{i=1..n} s_i.completed$.

Then, the XOR-split pattern, defined as a point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen, can be specified as follows:

[1]Our approach also considers the following list of patterns: sequence, *AND-split*, *OR-split*, *XOR-split*, *AND-join*, *OR-join*, *XOR-join* and *m-out-of-n* [10].

Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006)
0-7695-2522-9/06 $20.00 © 2006 IEEE

Definition 5: We define the XOR-split pattern as the following function:
$XOR-split(S)$: $S = \{s_0, s_1, \ldots, s_n\} \longrightarrow \mathcal{CF}low$ such that $\forall i,\ 1 \leq i \leq n,\ Activation_{Cond}(s_i) = s_0.completed \bigwedge c_i$ | there is always a unique $c_j (1 \leq j \leq n)$ which is evaluated to true after the completion of s_0.

We argue that once defined, a workflow pattern implicitly defines a new class of dependencies, called *potential transactional dependencies*, i.e. a set of transactional dependencies not initially defined by the pattern, and that can be used/added in order to tailor (or to augment) the control flow (see figure 2). In fact, these dependencies are directly related to the semantics of the activation dependencies of the pattern. This can be easily explained by the constraints we had presented in the previous section.

Once a pattern is defined, a control flow is specified, but if we consider the transactional dependencies, it is possible to refine the pattern in order to take into account not only the control flow, but also the transactional dependencies. We are using the term potential as it can exist several transactional dependencies, but the designer of the composite service can only select a few of them.

Enhancing the workflow patterns with transactional aspects is very interesting, as it makes it possible to handle failures in a better way, i.e. to ensure the reliability of the execution of an instance of the pattern. In our case, we are considering transactional composite Web services, and we model the composition using the powerful concept of patterns, and we add reliability to the composition using transactional aspects.

B. Workflow patterns revisited with transactional dependencies

In the previous paragraph, we presented potential transactional dependencies. As the transactional flow we are considering is made of alternative, cancellation, and compensation, we distinguish between three different types of potential dependencies which are:

- $Potential - Compensation_{Cond}(s)$: the potential compensation condition specifies when s can be eventually compensated,
- $Potential - Cancellation_{Cond}(s)$: the potential cancellation condition specifies when s can be eventually cancelled,
- $Potential - Alternative_{Cond}(s)$: the potential alternative condition specifies when s can be eventually activated as an alternative.

As an example, if we consider the *AND-join* pattern previously introduced, we obtain the following set of potential transactional dependencies. First, there is no alternative dependencies. Each service s_i will be compensated or cancelled (according to its current state) when a service s_j fails ($1 \leq i, j \leq n$ and $i \neq j$). Then, each service s_i ($1 \leq i \leq n$) will be compensated when s_0 fails or will be compensated. This can be formally defined as follows:

Definition 6: The potential transactional flow of the pattern *AND-join* applied to the service s_0 is defined as:

$Potential - \mathcal{TF}low_{AND-join}(S)$:
$S = \{s_1, \ldots, s_n, s_0\} \longrightarrow \mathcal{TF}low$
where:

- $Potential - Alternative_{Cond}(s_i) = \emptyset, \forall\ 0 \leq i \leq n$
- $Potential - Compensation_{Cond}(s_i) = s_0.failed \bigoplus s_0.compensated \bigoplus_{1 \leq j \leq n, j \neq i} s_j.failed,\ \forall\ 1 \leq i \leq n,$
- $Potential - Cancellation_{Cond}(s_0) = \emptyset,$
- $Potential - Cancellation_{Cond}(s_i) = \bigoplus_{1 \leq j \leq n, j \neq i} s_j.failed,\ \forall\ 1 \leq i \leq n.$

Figure 2.b illustrates the potential transactional flow of the *AND-join* pattern applied to *(HB,FB,OP)*. The service FB can be compensated when HB fails or when OP is compensated (or when OP fails). That means $Potential - Compensation_{Cond}(FB) = OP.failed \bigoplus OP.compensated \bigoplus HB.failed$, and $Potential - Cancellation_{Cond}\ (FB) = HB.failed$.

We proceed analogously to add transactional dependencies to other patterns we consider.

V. Web services composition using transactional patterns

A. Motivations

The use of workflow patterns appears to be an interesting idea to compose Web services. However, current workflow patterns do not take into account the transactional properties (except the very simple cancellation patterns category [11]). It is now well established that the transactional management is needed for both composition and coordination of Web services. That is the reason why we augment the original workflow patterns with transactional dependencies, in order to provide a reliable composition (or coordination). We called this new type of patterns the transactional patterns. In the following, we will show how a composite Web service can be composed using these patterns, and what are the properties that can be infered by the application of these patterns.

B. Transactional patterns

Given the transactional properties of a service (pivot, compensatable, retriable), and a workflow pattern augmented with transactional dependencies for a composite service, we are able to deduce a new pattern, called a *transactional pattern*, that will be used to specify both the control and transactional flows. The control flow $\mathcal{CF}low$ is inherited from the workflow pattern (i.e. the activation dependencies), while the transactional flow $\mathcal{TF}low$ is specified using the set of transactional dependencies for managing alternatives, compensation, or cancellation given by the $Potential - \mathcal{TF}low$ function introduced in definition 6.

From a transactional pattern, one can define several *transactional pattern instances* which are the application of a transactional pattern to a given set of services, and where the transactional dependencies of the instance is a subset of the set of the potential transactional dependencies of the transactional pattern.

Proceedings of the Advanced International Conference on Telecommunications and
International Conference on Internet and Web Applications and Services (AICT/ICIW 2006)
0-7695-2522-9/06 $20.00 © 2006 IEEE

For instance, on figure 2.c, a designer have the choice between the alternative or the compensation dependency. His/her choice may depend on the properties of the Web services he/she will use. If the OP service has the compensatable transactional property, the designer the compensation dependency. Conversely, the designer may choose to keep the alternative dependency for the delivery if the OP is not compensatable. To summarize, the choice depends not only on the designer, but also on the transactional properties of the component services.

C. Composition

As said earlier, transactional patterns are interesting to compose a set of Web services to obtain a transactional composite service (TCS) which is reliable from an execution point of view. Thus, we specify a TCS as a set of transactional patterns instances connected together (sharing some component services). Figure 3 shows how we can specify the online travel arrangement service using the following transactional patterns composition: *Trans*$_{AND-split}$*(CRS, HB, FB)*, *Trans*$_{AND-join}$*(HB, FB, OP)*, *Trans*$_{XOR-split}$*(OP, TDFE, TDU)*.

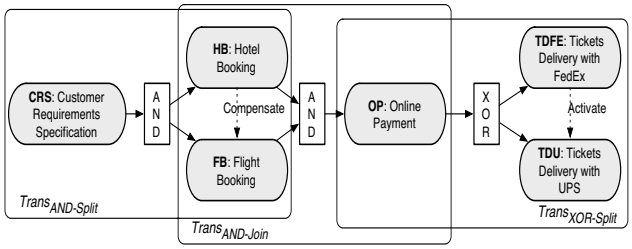

Fig. 3. A TCS is defined as a composition of a set of transactional patterns instances.

However, connecting a set of transactional patterns instances can lead to a control flow and/or a transactional flow inconsistencies. For instance, control consistency problem can raise when instances are disjoined (no shared services allowing to connect the instances) or when an *XOR-split* instance is followed by an *AND-join* instance. Likewise, transactional inconsistency can raise when a component service fails, causing the entire TCS abortion, with remaining effects of the partial execution. For example, if we suppose that OP is not retriable (it can fail) in the TCS defined in figure 3, this means that FB should be compensated in order to be sure that it does not exist a remaining effect (a flight is booked) after the abortion of the TCS. This implies that it exists the compensate transactional dependency between OP and FB, and that FB is compensatable.

The first step to detect and correct transactional inconsistencies consists in determining component services transactional properties. We use the following rules (applied in the given order) to determine the component services transactional properties:

1) each service is by default retriable, pivot and effectless.
2) each service which is a target of a compensation dependency is compensatable and with effects.
3) each service which is a source of a cancellation, alternative or compensation (in case of failure) dependency is not retriable.

Potential conditions depend on the transactional properties of the component Web services. For instance, the potential compensation condition $OP.failed$ of the service FB depends if OP is retriable or not. By analogy, the compensation dependency from HB to FB (specified by the TCS in figure 3) implies that HB is not retriable (since it can fail) and FB is with effects (since it is compensated). We can also deduce for this TCS that the service TDU is retriable since it is not the source of any compensation, cancellation or alternative dependency. This is not the case of the service $TDFE$ which is not retriable (source of an alternative dependency).

To manage these problems, we define a TCS as *valid* if it ensures both the control flow consistency and the transactional flow consistency. In order to guarantee the reliability of the TCS, we are using a set of rules to check both the control flow and the transactional flow consistency. These rules are described in [12]. Briefly summarized, the algorithm we are using is as follows:

- after a component service failure, we are looking for an alternative dependency, if it exists,
- after a composite service failure, we try to compensate what can be compensated given the transactional properties of the component services,
- after a composite service failure, we cancel all the current executions of the TCS.

In order to compute the transactional consistency, we need not only these rules, but also the transactional properties of each service introduced in III-A.

Thus, there are two different ways to use our approach, i.e. to use transactional patterns for Web services composition. On one hand, the designer chooses a set of transactional patterns, connects them, and fixes the transactional dependencies of the TCS. Then, we have a matching algorithm which is looking for a set of Web services with transactional properties which are able to match both the patterns and the dependencies of the TCS. On the other hand, the designer defines a TCS using transactional patterns, and a set of Web services with their respective transactional properties. Then, the algorithm tries to infer and select the transactional dependencies according to the potential transactional properties. If the algorithm fails, that means that transactional inconsistencies have been found.

VI. Related Work

Coordinating a set of treatment units (or tasks) to achieve a common goal has been tackled by both ATM [1] and workflow systems [4]. Due to historical reasons, each of these two technologies addressed the problem from a different point of view. Emerged from data base transactional processing, ATM ensures reliability and correctness for the supported application executions. Evolved from office automation, workflow has became the key technology for business processes automation.

Unfortunately, the workflow flexibility is missed in ATM and the transactional reliability is not supported by workflow systems. To overcome these limitations, [13] proposed a transactional Workflow system supporting multitask and multisystem activities where: (a) different tasks may have different execution behaviors or properties, (b) designers define the coordination of the different tasks, and (c) specify their required failure atomicity. In this approach, failure atomicity requirement is defined by specifying a set of Accepted Termination States (ATS). A transaction is committed if it reaches an accepted termination state, otherwise it is aborted. The fact of proceeding so limits its application to database applications.

If the combination of reliability and flexibility is important for EAI, it is essential in the context for Web services due to the cross boundaries and lack of trust aspects. Thus, many standards have emerged to address this problem by adapting previous solutions to the context of Web services.

Emerging standards such as BTP [14], WS-transaction (WS-AtomicTransaction [15], [16] define models to support a two-phase coordination of web services. These proposals are based on a set of extended transactional models to specify coordinations between services. Participants agree to a specific model before starting interactions. Then the corresponding coordination layer technologies support the appropriate messages exchange according to the chosen transactional model. These propositions inherit the extended transactional models rigidity. In addition, there is a potentiel problem of transactional interoperability between services implemented with different approaches. Our approach can complement these efforts and overcome these two gaps. Indeed, our approach allows for reliable, more complex, and more flexible compositions. In addition, it can coordinate services implemented with different technologies since we use only services transactional features (and not interested in how they are implemented). So, we can use our approach to specify flexible and reliable composite services, while component services can be implemented by one of the above technologies. Once a valid TCS is reached, it can be considered as a coordination protocol and can be plugged in one of the existing coordination technology to be executed.

In our previous work [12], we propose an approach to ensure reliable Web services compositions. We distinguish between control flow and transactional flow of a TCS, but we proceed differently. First, the designers have to define the global composite service structure, and second, the designers have to specify the required Accepted Termination States (ATS) as a correctness criteria (like in [13]). Then, we use a set of transactional rules to assist designers to compose a valid TCS with regards to the specified ATS. However, specifying ATS can be a non trivial and a trouble task. The new approach presented in this paper shares the same purpose, but it tries to be more practical. To avoid the painfull specification of the ATS, this new approach encapsulates within a transactional pattern a default ATS which takes into account some situations like "*each service with effects need to be compensated when the global execution aborts*". But to permit to new approach to consider specific contexts, one can modify/adapt the transactional behavior of a pattern. Thus, we keep the strengths of the previous approach as we are taking into account the needs of the designers.

VII. Conclusion

In this paper, we propose a solution to ensure reliable and flexible a Web service composition. The main idea of our approach is to combine workflow flexibility and transactional processing reliability. We introduce an extension of workflow patterns, the *transactional patterns*, which can be seen as a convergence concept between workflow systems and transactional models to easily define flexible and reliable composite Web services. Then, we propose a set of rules in order to avoid inconsistencies which can result from the composition of the patterns.

References

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

[2] 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.

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

[4] W. M. P. van der Aalst and K. M. van Hee, *Workflow Management: models, methods and tools*, ser. Cooperative Information Systems, J. W. S. M. Papazoglou and J. Mylopoulos, Eds. MIT Press, 2002.

[5] 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*, vol. 12, no. 1, pp. 59–85, 2003.

[6] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: An introduction to soap, wsdl, and uddi," *IEEE Internet Computing*, vol. 6, no. 2, pp. 86–93, 2002.

[7] P. F. Pires, M. R. F. Benevides, and M. Mattoso, "Building reliable web services compositions." in *Web, Web-Services, and Database Systems*, 2002, pp. 59–72.

[8] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz, "A transaction model for multidatabase systems." in *ICDCS*, 1992, pp. 56–63.

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

[10] W. M. P. van der Aalst, P. Barthelmess, C. Ellis, and J. Wainer, "Workflow Modeling using Proclets," in *5th IFCIS Int. Conf. on Cooperative Information Systems (CoopIS'00)*, ser. LNCS, O. Etzion and P. Scheuermann, Eds., no. 1901. Eilat, Israel: Springer-Verlag, September 6-8, 2000, pp. 198–209.

[11] W. M. P. van der Aalst and A. H. M. ter Hofstede, "Yawl: yet another workflow language." *Inf. Syst.*, vol. 30, no. 4, pp. 245–275, 2005.

[12] S. Bhiri, O. Perrin, and C. Godart, "Ensuring required failure atomicity of composite web services." in *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, 2005, pp. 138–147.

[13] M. Rusinkiewicz and A. Sheth, "Specification and Execution of Transactional Workflows," in *Modern Database Systems, The Object Model Interoperability and beyond*, W. Kim, Ed. Addison Wesley, ACM Press, 1995, pp. 592–620.

[14] "Business Transaction Protocol, Version 1.0," in *OASIS Committee Specification*, 2002.

[15] D. Langworthy and al., "Web services atomic transaction (ws-atomictransaction)," *BEA, IBM, Microsoft, 2003.*

[16] ——, "Web services business activity framework (ws-businessactivity)," *BEA, IBM, Microsoft, 2003.*