A FLEXIBLE MODEL FOR PROVIDING TRANSACTIONAL BEHAVIOR TO SERVICE COORDINATION IN AN ORTHOGONAL WAY

Alberto Portilla*, Genoveva Vargas-Solar, Christine Collet
LSR-IMAG Laboratory, BP 72, 38402 Saint Martin d'Hères, France

José-Luis Zechinelli-Martini
CENTIA, Universidad de las Américas Sta Catarina Martir, Puebla, Mexico

Luciano Garcia-Bañuelos
Departamento de Ciencias Básicas, Ingeniería y Tecnología,Universidad Autónoma de Tlaxcala, Mexico

Keywords: Transactional behavior, advanced transactional models, services oriented systems, services based applications, web services.

Abstract: A key step towards consistent services coordination is providing non functional properties. In that sense, transactional properties are particularly relevant because of the business nature of current applications. While services composition has been successfully addressed, transactional properties of services composition have been mainly provided by *ad-hoc* and limited solutions at systems' back end. This paper proposes a transactional behavior model for services coordination. We assume that given a flow describing the application logic of a service based application, it is possible to associate to it a personalized transactional behavior in an orthogonal way. This behavior is defined by specifying contracts and associating a well defined behavior to the activities participating in the coordination. Such contracts ensure transactional properties at execution time in the presence of exceptions.

1 INTRODUCTION

A key step towards consistent services coordination is providing non functional properties. In that sense, transactional properties are particularly relevant because of the business nature of current applications. While services composition has been successfully addressed, transactional properties have been mainly provided to services coordination by *ad-hoc* and limited solutions at systems' back end. In service based applications, a service is a software component, available on a network, that offers some functions by exporting an application programming interface (API). Nowadays, services based applications are built by coordinating interactions among several services.

Figure 1 shows an example of a service based application for trip reservation, composed using services of several providers. In the Figure, boxes represent activities and each activity represents a service method call. Let us consider that this application implements the following logic: given a reservation it is necessary to get a bank authorization for the payment (a_1). If the payment has been guaranteed, the flight reservation (a_2), the hotel reservation (a_3) and the car reservation(a_4) can be done, otherwise the reservation is canceled (a_5). In addition, there are transactional issues that must be considered, some are related to application semantics and others to activities semantics. In this example, we can consider the following aspects: before making a reservation (flight, hotel or car) the payment must be authorized (a_1); a reservation cannot be rejected (a_5) and processed at the same time (a_2, a_3 and a_4). Furthermore, there are several alternatives for executing an activity (e.g. a flight reservation can be completed using Mexicana, Air France or British Airways). Some activities are critical for the success of the application. For exam-

*Supported by the Mexican Education Council through the program for improving the teaching system in mexican public universities (PROMEP-SEP), the Autonomous University of Tlaxcala, México and the Jenkins Excellence Fellowship Program at UDLA.

Portilla A., Vargas-Solar G., Collet C., Zechinelli-Martini J. and Garcia-Bañuelos L. (2007).
A FLEXIBLE MODEL FOR PROVIDING TRANSACTIONAL BEHAVIOR TO SERVICE COORDINATION IN AN ORTHOGONAL WAY.
In *Proceedings of the Third International Conference on Web Information Systems and Technologies - Internet Technology*, pages 104-111
DOI: 10.5220/0001289001040111

ple, while the flight and hotel reservations are critical for the acceptance of the trip reservation, an unsuccessfully car reservation can be tolerated. Some activities can be retried in case of failure. For example, asking for the bank authorization or booking the flight can be retried several times.

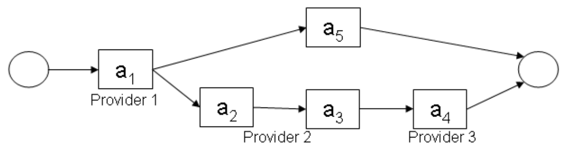

Figure 1: Service based application example.

Transactional behavior for services coordination is currently addressed in three ways: it is explicitly defined within the application logic by adding sequences of activities to its workflow (e.g. (Vidyasankar and Vossen, 2004), (Schuldt et al., 2002), etc.); it is implemented by transactional protocols (e.g. BTP (Furniss, 2004), WS-Tx (Cox et al., 2004), etc.); or it is specified by transactional policies expressed using transactional coordination languages (e.g., πt-calculus (Milner et al., 1992), (Bhiri et al., 2005), etc.). However, this leads to complex service based applications difficult to maintain and hardly adaptable to different application requirements (Hagen and Alonso, 2000; Vargas-Solar et al., 2003; Derks et al., 2001).

This paper proposes a model for specifying transactional properties for services coordination. Given a coordination describing the application logic of a service based application it is possible to associate it with a transactional behavior. Our approach addresses atomic behavior based on atomicity contracts for associating atomicity properties and recovery strategies to sets of activities (i.e., spheres (Davies-Jr, 1978; Hagen and Alonso, 2000; Leymann and Roller, 1997)). Our model separates the specification of coordination and transactional behavior as follows:

- The application logic is captured by using a coordination approach based on workflows.
- Transactional behavior is defined by contracts according to the behavior of services participating in a coordination and the semantic of the target service based application.

Our approach proposes recovery strategies for ensuring transactional behavior of services coordination. Such strategies can be chosen according to specific application requirements. Thus, the same type of application can have different associated behaviors according to given application requirements.

The remainder of the paper is organized as follows. Section 2 introduces the main concepts of the transactional behavior model that we propose. Section 3 illustrates the specification of transactional behavior for a target application through a use case. Section 4 compares our work with existing approaches that provide transactional properties to services coordination. Finally, Section 5 concludes this paper and discusses future work.

2 TRANSACTIONAL COORDINATION MODEL

A coordination model can be used for expressing an application logic defined by an orchestration. An orchestration specifies the workflow whereby activities are synchronized using ordering operators. Acoording to its role within the application logic, an activity has an associated behavior that specifies the way it should be handled under the presence of exceptional situations.

2.1 Service

A service represents a set of functions provided by an autonomous software component through a port. It is formally defined as a quadruple $\langle S\text{-}name, FM, NM, P\rangle$ where:

- $S\text{-}name$ is the name of the service.
- FM and NM are interfaces consisting of a finite set of methods that provide respectively functional and non functional aspects, for example XA/XOPEN (The-Open-Group, 1991). A method is defined by a triple M=$\langle M\text{-}name, In, Out\rangle$ where:
  - $M\text{-}name$ is the name of the method.
  - In and Out are respectively the sets of input and output parameters.
  - A parameter is defined by a tuple $\langle P\text{-}name, Type\rangle$, where:
    - $P\text{-}name$ is the name of the parameter.
    - $Type$ represents a predefined type such as boolean, string, integer, etc.
- P is the communication port used for interacting with the service.

The trip reservation application previously introduced uses three service providers (see Figure 1): *bank* that controls financial operations executed on accounts (*Provider 1*), *travel agency* that rates and manages flight and hotel reservations (*Provider 2*), and car rental company that manages car rental (*Provider 3*).

2.2 Orchestration

An orchestration describes an application logic through control and data flows. In our work an orchestration is defined using a Petri net, where a place models the states among two transitions. A transition represents a task, hereinafter, refered as activity. A token represents an execution state. A directed arc is used to link places and transitions. A Petri net is defined as a triple $\langle PL, A, F \rangle$ (van der Aalst and van Hee, 2004) where:

- PL is a finite set of places.
- A is a finite set of transitions, $PL \cap A = \phi$.
- $F \subseteq (PL \text{ x } A) \cup (A \text{ x } PL)$ is a set of arcs.

Using Petri nets the control flow of an orchestration is expressed by order operators (sequential routing, parallel routing, selective routing, and iteration) that relate activities. The data flow is embedded within the control flow. Figure 2 shows the orchestration of the trip reservation example: places are represented as circles (p_1, p_2, p_3, p_4, p_5 and p_6), transitions are represented as boxes (a_1, a_2, a_3, a_4 and a_5), tokens are represented as black circles, and flow relationships are represented as arrowed lines.

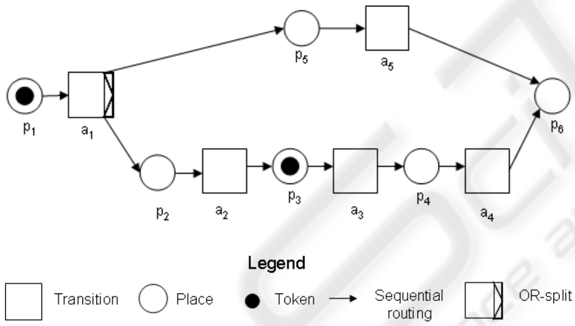

Figure 2: Service based application with Petri nets.

2.3 Activity

Depending on its role within an application logic, an activity can have different behaviors at execution time in the presence of failures (Gray, 1981; Eder and Liebhart, 1996).

2.3.1 Behavior

There are three possible scenarios for characterizing the behavior:

1. An activity can be rolled-back/undone, but possible side effects must be taken into account. In the trip reservation example (see Figure 1), undoing the flight reservation (a_2) can imply an extra charge.
2. A committed activity can be compensated by a so called compensation activity that semantically undoes its actions but it does not necessarily gets back the application to the previous state before the activity was committed. For example, in the trip reservation application, canceling a flight reservation can be undone with an extra charge which does not return the balance account to the previous state. However, there are some activities that cannot be undone, for example personalizing an item using laser engraving.
3. The execution of an activity can be retried several times within the same coordination. In our example, hotel reservation can be executed several times. However, retrying an activity can be associated to other constraints according to its behavior, such as the number of times that an activity can be retried.

Considering the above scenarios an activity can be[2]:

- *Non vital*. The activity does not need to be compensated if it has to be undone/rolled back after having committed. For example, let us assume an activity that uses the trip reservation information for sending commercial information. It can be defined as non vital because the reservation can be done even if the customer does not receive the publicity.
- *Critical*. There is no way to compensate or undo the effects of an activity once it has been committed. Recalling the trip reservation example, the activity bank authorization is critical because once a reservation has been authorized the bank cannot change its opinion. So the execution of the reservation cannot be stopped for this reason.
- *Undoable*. A commited activity can be undone by a compensating activity without causing side-effects. Once it has been compensated the activity can be retried. In our example the activity hotel reservation can be defined as undoable, assuming that a hotel reservation can be canceled at most three days before the date of arrival.
- *Compensatable*. A commited activity can be undone by a compensating activity with associated side-effects. Once it has been compensated the activity can be retried but with side effects such as extra costs. In our trip reservation example the activity flight reservation can be undone by an activity that cancels the flight and that reimburses a

[2]For the time being, this classification is non exhaustive.

percentage of the original amount to the costumer account.

2.4 Sphere

A sphere groups together sequences of activities or spheres. In the following let A be a set of activities. This set may be obtained from an orchestration defined as a Petri net $\langle PL, A, F\rangle$. The set of spheres associated to A denoted $S(A)$ is defined by following rules:

1. If $a_i \in A$, then $(a_i) \in S(A)$ is a simple sphere and a_i is called the component of the sphere. For example s_2 in Figure 3, where $s_2 = (a_5)$.
2. If s_1, s_2, ...,s_n $\in$ $S(A)$, (s_1,s_2,...,s_n) $\in$ $S(A)$ is a complex sphere and s_1, s_2, ...,s_n are called the components of the sphere. For example s_1 in Figure 3 where $s_2 =$ ((a_2), (a_3), (a_4)). Similarly, S_3 is a complex sphere where $s_3 =$ (s_1, s_2).

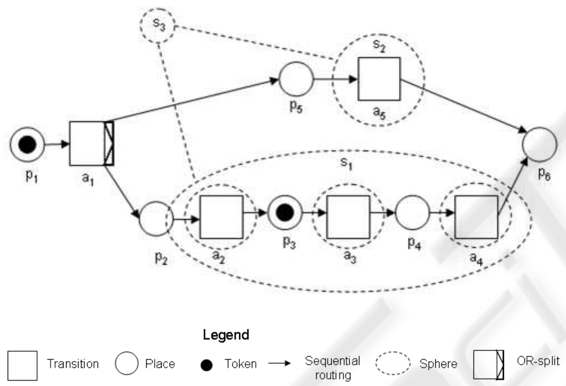

Figure 3: Spheres example.

2.4.1 Behavior

The behavior of a sphere at execution time can be deduced from the behavior of its components. Given a sphere S its behavior is deduced according to the following rules:

- S is *non vital*, if its components are non vital. For example, assuming that components (a_2), (a_3) and (a_4) are non vital the sphere s_1 is non vital (see Figure 3).
- S is *critical*, if it contains at least one critical component that has been committed. For example assuming that the component (a_2) is critical and it has already been committed, then sphere s_1 is critical (see Figure 3). Note that a token in place p_3 indicates that (a_2) is committed for a given execution state.
- S is *undoable*, if all its components are undoable or non vital. For example assuming that (a_2) is non vital and (a_3) and (a_4) are undoable, sphere s_1 is undoable (see Figure 3).
- S is *compensatable*, if all its components are non vital, undoable or compensatable. For example assuming that (a_2) is non vital, (a_3) is undoable, and (a_4) is compensatable, sphere s_1 is compensatable (see Figure 3).

Note that the behavior of a sphere is determined at run time since it depends on the behavior of its components and on their execution state. For example, consider sphere s_1 in Figure 3 with component (a_3) defined as critical. Then s_1 is compensatable before (a_3) commits, and it becomes critical after (a_3) has been committed.

2.4.2 Well Formed Sphere

A sphere S is well formed if it contains at most one critical component, denoted as c_{cr}, or all components within the sphere are not critical. Note that when S contains a critical activity, once c_{cr} commits S must commit because c_{cr} cannot be compensated or undone. In such a case, for ensuring that S commits when c_{cr} commits, it is necessary that the set of components to be executed after c_{cr} must be retriable. For example, sphere s_1 in Figure 3 is well formed if (a_2) is compensatable, (a_3) is critical and (a_4) is undoable.

2.5 Contract

A contract specifies a transactional behavior for a sphere (e.g. atomicity, isolation, etc.). It specifies a recovery strategy that must be executed in case of failure according to the sphere behavior. In order to define such strategies we introduce the concept of safe point.

2.5.1 Safe Point

Given a simple sphere S its safe point is either the most recent critical component committed within S or the activity/sphere before the first component of S. The safe point represents the last consistent state of a coordination before the execution of S. If S fails, it is possible to cancel the components within S and restart the execution from the safe point. For example, a_1 can be a safe point for sphere s_1. Assuming that a_1 is the safe point of s_1, it is also the safe point for sphere s_3 if sphere s_2 is well formed and sphere s_2 is undoable (see Figure 3).

2.5.2 Recovery Strategies

There are three main recovery strategies that can be implemented in case of failure: forward execution, backward and forward recovery. Recovery strategies help to decide wether the execution of a coordination can continue after the occurrence of a failure. Given a well formed sphere S with its associated safe point SP:

- Forward execution. If a retriable or non vital component in S fails the execution of S can proceed anyway. For example considering that all components of s_1 are non vital (see Figure 3), its execution can proceed even if one of its components fails.
- Backward recovery. If a component in S fails and forward recovery cannot be applied then previously committed components in S are undone by their corresponding compensating activities until a safe point is reached. For example, assume that the components in s_1 are defined as follows: (a_2) and (a_3) are compensatable and (a_4) is critical. If during the execution of s_1, (a_3) fails, then (a_2) and (a_3) are succesively compensated until a_1 is reached.
- Forward recovery. It combines backward recovery and forward execution. If forward execution cannot be applied because a critical or compensatable component has failed then apply backward recovery until a safe point is reached and forward execution with a different execution path can be applied.

Using one of the above recovery strategies depends on the type of component that fails and the type of previously executed components.

2.6 Atomicity contracts

A classic example of a transactional behavior is atomicity. We used the notion of contract and recovery strategies for defining so called atomicity contracts. We show in the following the definition of a strict atomicity contract[3]. Intuitively, an atomicity contract specifies that given a sphere S all its components must be successfully executed or no component at all. The strict atomicity contract is defined by the following rules. Given a well formed sphere S:

- *Rule 1.* If a critical component in S fails, the sphere fails and backward recovery is applied.
- *Rule 2.* If a component in S fails and the critical component of S has already been committed then forward execution is applied until S commits.
- *Rule 3.* If a component in S fails and the critical component of S has not yet been committed, forward execution is applied until S commits. If the component has failed repeatedly S fails and backward recovery is applied.

[3] In fact, we are relaxing the classic atomicity concept.

3 DEFINING TRANSACTIONAL BEHAVIOR FOR AN E-COMMERCE APPLICATION

Consider the e-commerce application illustrated in Figure 4. A costumer first provides an order and her/his account information, then the system sends commercial information. If the purchase is authorized by the bank, the purchase can proceed; otherwise the order is canceled. For processing authorized orders, the stock is checked and the purchase process starts. If products are available the purchase is applied to an account, an invoice is sent and the order is completed; at the same time the packet is wrapped and delivered. On the other hand, if products are not available, the payment and the shipment are canceled. Thus the application logic (see Figure 4) of such an application specifies activities for defining and authorizing an order (Get_Order(), Get_Payment_Information(), Bank_Authorization()); and four other execution paths for:

- Sending commercial information: Send_Commercial_Info();
- Stop a rejected order: Cancel_Order();
- Processing an authorized order: Process_Order(), Check_ Stock(), Wrap() and Deliver() (if there is enough stock); or Cancel_Shipment(), Cancel_Payment() (if there is not enough stock).
- Executing the purchase: Apply_Charges(), Send_Invoice(), Finish_Order().

3.1 Activities

The company that uses this application has specific business rules that must be respected. For instance:

- BR1: the bank authorization can be only executed once during the process of an order.
- BR2: once a package has been delivered it cannot be returned to the company.
- BR3: The company has implemented packages recycling strategies, so if a package has been

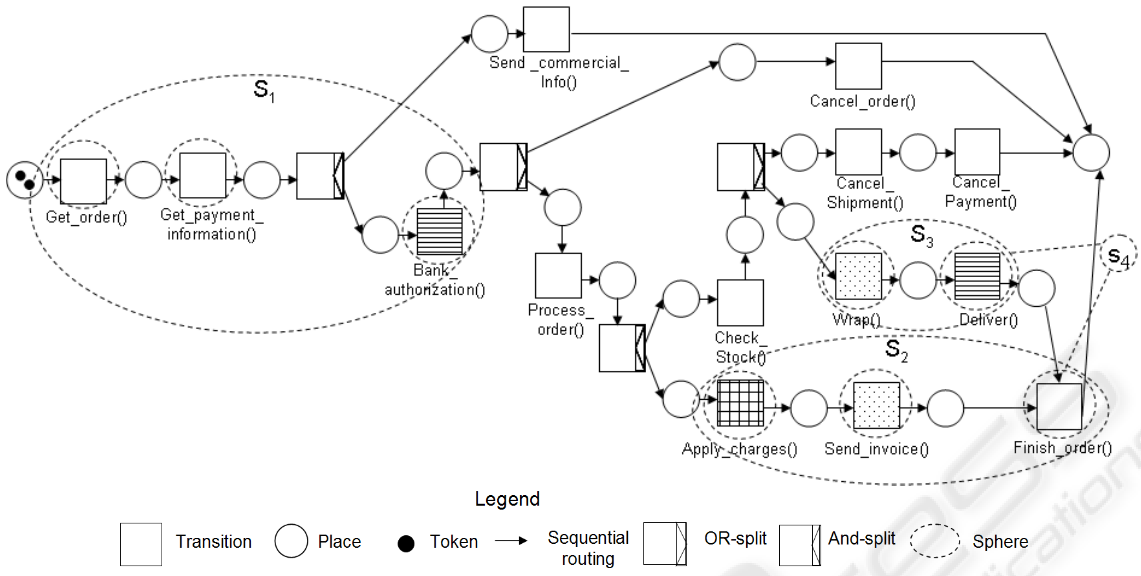

Figure 4: E-commerce application with transactional requirements.

wrapped, the wrapping paper can be removed and used for other packages.

- BR4: Invoices can be canceled.
- BR5: If clients are unsatisfied with the products they can return it to the company but extra charges are applied.

Considering such business rules, we associated the following behaviors to activities (see Figure 4):

- Bank_authorization() and Deliver() are critical activities (BR1, 2).
- Wrap() and Send_invoice() are undoable activities (BR3, 4).
- Apply_charges() is a compensatable activity (BR5).
- The remaining activities are non vital ones because no specific business rule is specified in the application.

It must be noted that activities behavior depends on the application semantics. If the application changes its business rules, activities behavior can change. An advantage of our approach is that this can be easily done because the application logic and the transactional behavior aspects are defined separately.

3.2 Spheres

The business rules defined in the previous section concern activities behavior. Still, the following business rules concern more complex patterns that complete the application semantics. The authorization of the bank is requested once the costumer information has been obtained (BR6). An order is finished only when charges have been applied to an account and the corresponding invoice has been sent (BR7). Once packages have been packed they must be shipped (BR8). The purchase is finished once the package is delivered and the order has been processed (BR9). Thus the following spheres and their associated behaviors must be defined:

- S_1 groups components involved in the order authorization ((Get_Order()), (Get_Payment_Information()), (Bank_ Authorization())). S_1 is a critical sphere because it contains the critical component (Bank_authorization()).
- S_2 groups activities used for implementing the order processing ((Apply_Charges()), (Send_Invoice()), (Finish_ Order())). S_2 is compensatable because (Apply_Charges()) is a compensatable component.
- S_3 groups activities for delivering the package ((Wrap()), (Deliver())). S_3 is critical because it contains a critical component (Deliver()).
- S_4 groups spheres S_2 and S_3 for ensuring that the purchase will finish only if the package is wrapped and delivered and the payment has been executed and the invoice has been sent. S_4 is a critical sphere because S_3 is critical.

Note that all spheres are well formed because they contain at most one critical component.

Once spheres have been defined, they must be associated to contacts that specify strategies to be undertaken at execution time in the presence of failure. In the case of our example, we have identified spheres requiring strict atomicity properties in order to fulfil BR7-9. Coupled with the strict atomicity contract that we defined in Section 2.6, S_1 ensures BR6, S_2 ensures BR7, S_3 ensures BR8, and S_4 ensures BR9.

4 RELATED WORKS

Several research projects have addressed transactional behavior for information systems, first in the context of DBMS and after for process oriented systems.

In DBMS, transactional behavior has been tackled successfully to data through the concept of ACID transactions (Delobel and Adiba, 1982; Ozsu and Valduriez, 1999; Doucet and Jomier, 2001; Gray and Reuter, 1993) and advanced transactional models (García-Molina and Salem, 1987; Elmagarmid et al., 1990; Wachter and Reuter, 1992). These approaches are well suited when data reside in one site and transactions lifetime is short. Besides they operate with homogeneous execution units and therefore they are not applicable directly to others environments such as Internet.

In workflow systems there are several approaches that aim at ensuring consistency among computations using process as execution units. WAMO (Eder and Liebhart, 1995) introduces a complex transactional language for workflows. (Hagen and Alonso, 2000) introduces an approach to add atomicity and exception handling for IBM-FlowMark. (Grefen et al., 2001) proposes a workflow definition language to implement the saga model. (Derks et al., 2001) addresses atomic behavior for workflows by means of spheres. (Schuldt et al., 2002) proposes a model based in flexible transaction model for handling concurrency and recovery in process execution. However these approaches address transactional behavior in an *ad-hoc* fashion.

In Web services, transactions are used to ensure sound interactions among business process. Web services transactions (WS-Tx) (Cox et al., 2004) and business transaction protocol (BTP) (Furniss, 2004) remain as the most accepted protocols for coordinating Web services. A coordination protocol is a set of well-defined messages that are exchanged between participants of a transaction scope. However, these approaches do not offer mechanisms to ensure correctness in the specification because there is no reference model and the developer implements transactional behavior.

Other approaches provide transactional frameworks. (Fauvet et al., 2005) introduces a model for transactional services composition based on an advanced transactional model. (Bhiri et al., 2005) proposes an approach that consists of a set of algorithms and rules to assist designers to compose transactional services. In (Vidyasankar and Vossen, 2004) the model introduced in (Schuldt et al., 2002) is extended to web services for addressing atomicity. These approaches support the definition of atomic behavior based in the states of termination of activities with execution control flow defined a priori, which makes them not adaptable.

There are other approaches, transaction policies (Tai et al., 2004), πt-calculus (Bocchi et al., 2005) and transactional Web services orchestration (Hrastnik and Winiwarter, 2006), that address transactional behavior to web services based on existing protocols (BTP and WS-Tx) and therefore implement partially transactional behavior without a clear separation among application logic and transactional requirements.

In contrast to Web services protocols and frameworks, we consider that transactional aspects can be separated from the coordination specification and tackled using a general point of view. In this way our model fulfills the current spirit of reusing practices existing in software engineering.

5 CONCLUSIONS AND FUTURE WORK

The main contribution of this paper is an approach that adds transactional behavior to services coordination. For modeling transactional behavior, we introduce a model based on behavior of activities and set of activities (spheres). The contract concept is used for addressing a transactional behavior to spheres. Our model can be added to existing coordination approaches with minor changes but must important respecting the application logic. Our behavior model can be scaled because spheres can be defined recursively. Different from existing proposals our approach enables the definition of transactional behavior without implying the modification of the original application logic. Besides, our model is not based in any advanced transactional model, but enables such kind of behavior if necessary.

We are currently formalizing our model to provide a general framework for specifying transactional properties for services coordination and for ensuring such properties at execution time. We are also conducting the implementation of a contract management engine that can be plugged in existing orchestration engines to enact transactional orchestrations.

Finally, we will explore how our behavior model can be extended to address other properties such as isolation, and durability.

ACKNOWLEDGEMENTS

This research is partially related to DELFOS project of the Franco-Mexican Laboratory of Informatics (LAFMI) of the French and Mexican governments.

REFERENCES

Bhiri, S., Godart, C., and Perrin, O. (2005). Reliable web services composition using a transactional approach. In International, I., editor, *O. e-Technology, e-Commerce and e-Service*, volume 1 of *eee*, pages 15–21.

Bocchi, L., Ciancarini, P., and Rossi, D. (2005). Transactional aspects in semantic based discovery of services. In Jacquet, J.-M. and Picco, G. P., editors, *COORDINATION*, volume 3454 of *Lecture Notes in Computer Science*, pages 283–297. Springer.

Cox, W., Cabrera, F., Copeland, G., Freund, T., Klein, J., Storey, T., and Thatte, S. (2004). Web services transaction (ws-transaction). Technical specification, BEA Systems, International Business Machines Corporation, Microsoft Corporation, Inc.

Davies-Jr, C. T. (1978). Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198.

Delobel, C. and Adiba, M. (1982). *Bases de données et systèmes relationnels*. Dunod, Informatique.

Derks, W., Dehnert, J., Grefen, P., and Jonker, W. (2001). Customized atomicity specification for transactional workflows. In *Proceedings of the International Symposium on Cooperative Database Systems and Applications*, pages 155–164. IEEE.

Doucet, A. and Jomier, G., editors (2001). *Bases de données et internet, Modèles, languages et système*. Hermes, Informatique et Systèmes d'information. Lavoisier, first edition.

Eder, J. and Liebhart, W. (1995). The workflow activity model WAMO. In *Conference on Cooperative Information Systems*, pages 87–98.

Eder, J. and Liebhart, W. (1996). Workflow recovery. In *Proceedings of the International Conference on Cooperative Information Systems*. CoopIS'96.

Elmagarmid, A. K., Leu, Y., Litwin, W., and Rusinkiewicz, M. (1990). A multidatabase transaction model for interbase. In *Proceedings of the sixteenth international conference on Very large databases*, pages 507–518, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Fauvet, M.-C., Duarte, H., Dumas, M., and Benatallah, B. (2005). Handling transactional properties. In LNCS, S.-V., editor, *WISE 2005: 6th International Conference on Web Information Systems Engineering*, volume 3806, pages 273–289.

Furniss, P. (2004). Business transaction protocol. Technical specification, OASIS.

García-Molina, H. and Salem, K. (1987). Sagas. In ACM, editor, *9th Int. Conf. on Management of Data, San Francisco, California, USA*, pages 249–259.

Gray, J. (1981). The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society.

Gray, J. and Reuter, A. (1993). *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers.

Grefen, P., Vonk, J., and Apers, P. (2001). Global tansaction support for workflow management systems: from formal specification to practical implementation. *Very Large Data Base Journal*, 10(4):316–333.

Hagen, C. and Alonso, G. (2000). Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958.

Hrastnik, P. and Winiwarter, W. (2006). Twso transactional web service orchestrations. *Journal of Digital Information Management*, 4(1):–.

Leymann, F. and Roller, D. (1997). Workflow-based applications. *IBM Systems Journal*, 36(1).

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, part i/ii. *Journal of Information and Computation*, -(100):1–77.

Ozsu, M. T. and Valduriez, P. (1999). *Principles of distributed database systems*. Prentice Hall, second edition.

Schuldt, H., Alonso, G., Beeri, C., and Schek, H.-J. (2002). Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116.

Tai, S., Mikalsen, T., Wohlstadter, E., Desai, N., and Rouvellou, I. (2004). Transaction policies for service-oriented computing. *Data Knowl. Eng.*, 51(1):59–79.

The-Open-Group (1991). *Distributed Transaction Processing: The XA Specification*. X/Open Company Ltd., U.K.

van der Aalst, W. and van Hee, K. (2004). *Workflow Management, Models, Methods, and Systems*. The MIT Press, first edition.

Vargas-Solar, G., García-Banuelos, L., and Zechinelli-Martini, J.-L. (2003). Toward aspect oriented services coordination for building modern information systems. In *Encuentro Internacional de Computacion 2003*. ENC-SMCC, IEEE.

Vidyasankar, K. and Vossen, G. (2004). A multi-level model for web service composition. In *ICWS*, pages 462–. IEEE Computer Society.

Wachter, H. and Reuter, A. (1992). The contract model. In Elmagarmid, A. K., editor, *Database Transaction Models for Advanced Applications*, chapter 7, pages 219–263. Morgan Kaufmann Publishers.