A Unified Mechanism for Improving Advanced Transaction Management in Web Services

Sergio Castro
Université catholique de Louvain (UCL)
Département d'Ingénierie Informatique (INGI)
Louvain-la-Neuve, Belgium
Email: scastro@uclouvain.be

Johan Fabry
INRIA FUTURS
ADAM team
Laboratoire d'Informatique Fondamentale De Lille
Lille, France
Email: johan.fabry@lifl.fr

Kim Mens
Université catholique de Louvain (UCL)
Département d'Ingénierie Informatique (INGI)
Louvain-la-Neuve, Belgium
Email: Kim.Mens@uclouvain.be

Theo D'Hondt
Vrije Universiteit Brussel (VUB)
Programming Technology Lab
Pleinlaan 2,
1050 Brussels, Belgium
Email: tjdhondt@vub.ac.be

***Abstract*—Current web-service composition languages, such as BPEL, provide inferior support for transaction management. Consequently, specifications written in these languages are needlessly complex and hard to reuse. Furthermore, the languages themselves are not extensible, making them hard to adapt to support new forms of web service composition. Using a unified mechanism for dealing with transaction management we are able to simplify BPEL specifications, making them more reusable, and also allowing easy evolution of the language, addressing new forms of composition. In this paper we present DBCF, a framework that implements this unified mechanism. Moreover, we show how DBCF can be used as a basis for defining a BPEL extension that successfully addresses the above problems.**

I. Introduction

The idea of Web Services is to propose mechanisms for standardizing the interfaces of components in loosely-coupled distributed systems. As a result, they provide a solution to the interoperability problems that have been always present in application integration efforts. Web Services are also envisioned to be the basis for a seamless and almost completely automated infrastructure for cross-enterprise application integration [1]. Web Service composition techniques have an important role in the pursuit of this goal. They provide a mechanism for combining, according to business rules, the functionality provided by a group of services in a single service. Such compositions are usually defined in a Web Service composition language. In this context, the BPEL language [2] (Version 1.1, also known as BPEL4WS or WS-BPEL) has become the de facto standard in industry for Web Services composition, superseding other similar specifications (e.g. BPML [3] and WSFL [4]).

BPEL follows a hierarchical composition of activities. In BPEL it is possible to define basic activities that perform simple tasks (e.g. web service invocations), and also to define structural activities that contain or compose more than one activity. This allows for hierarchies of activities to be created. Structural activities define the order (control flow) in which nested activities are executed, or the conditional execution of such nested activities. Examples of structural activities in this language are *sequence* or *flow* activities, for sequential and parallel execution of nested activities.

One of the main concerns that BPEL needs to express is related to transaction management. This is because a composition defined in this language must coordinate different distributed services. As a result of this, the well known transactional ACID [5] properties traditionally used in tightly-coupled systems are insufficient. The underlying reason for this is that their enforcement could create performance penalties or even worse problems. To address this, more advanced forms of transaction management [6] [7] need to be used. In other words, additional transactional concepts developed in research on advanced transaction models should be taken into account, e.g., the notion of compensation, functional replication, synchronization of activities and timing management [8].

The BPEL strategy for expressing these transactional properties is however limited, non-extensible, and relies on different unrelated constructs. We believe that these concerns should be managed using a more straightforward, extensible and unified set of constructs, relying on proven existing formal models for advanced transaction management. In this paper we introduce our solution to these problems, in the form of DBCF (Dependency-Based Composition Framework). DBCF proposes an alternative simplified transaction management mechanism for Web Services Composition to the one used in BPEL. Furthermore, we show how we can employ DBCF as a basis for defining a BPEL extension that includes its main strategies for transaction management. This significantly improves the transaction handling facilities of the industry's state-of-the-art language for Web Services composition. These improvements are: first, a unified and straightforward mechanism for dealing with transaction management concerns; second, the production of less tangled code; third, reduction in the amount of transactional related constructs; fourth, the use of an extensible mechanism for transaction management.

This paper starts with a summary of how transactional concerns are currently managed in BPEL. After, we expose the ACTA formalism for advanced transaction models [6][7] and the KALA language [9] for the implementation of advanced transactions. They serve as a main source of inspiration for defining the main features of DBCF. Later, we illustrate the advantages of the DBCF transaction management mechanisms

to the ones used by BPEL. Finally, we describe our BPEL extension that reifies the main ideas for advanced transaction management developed in DBCF.

II. Advanced Transaction Management in BPEL

A. Implementation of Transaction Management Concerns

We briefly discuss here the six transactionally-related concerns that are supported by BPEL. These are synchronization of activities, compensation of activities, functional replication of activities, management of timing, exception handling and serializable scopes. We give a brief overview of each of these next, summarized from the BPEL specification [2].

Synchronization of activities: This concern is about the necessity of synchronizing, at a fine-grained level, the activities that are part of the business process. Some of these synchronization issues can be solved using control flow structures for specifying the order in which activities should be executed. However in other cases, particularly where high concurrency is present, more sophisticated synchronization mechanisms are needed. BPEL solves this with the notion of logical *links* and their different transactional outcomes associated with activities. Links are declared at the beginning of a *flow* activity representing parallel execution. Later, activities can associate these links with transactional outcomes using the *source* construct. Finally, activities that have synchronization dependencies with activities nesting *source* constructs, can evaluate such outcomes in boolean logic expressions, using the *target* construct, in order to determine if they can be executed or not.

Although this strategy permits the expression of sophisticated synchronization dependencies, it is complex and implies the use of at least three different constructs for each synchronization concern: the *link*, *source* and *target* constructs, which are scattered among the main business composition code.

Compensation of activities: Compensating activities allow to make early commits on subtransactions that belong to a long-lived transaction. This allows resources to be released early, improving performance. In case that the main transaction fails, this work has to be undone, which is performed by compensating transactions. To undo committed work, the compensating transactions of all already invoked subtransactions are executed. BPEL provides the dedicated construct *compensationHandler* for expressing the idea of compensation.

Functional replication of activities: When the request for a service fails, frequently an alternative service provider can be selected. From the point of view of the service invocator, both alternatives are functionally equivalent. This has been called *Functional Replication* in other work ([10]). In BPEL, this concern can be implemented as a special case of fault management. When a fault occurs in the execution of a Web Service invocation, an alternative service with the same functionality, can be invoked from a fault handler.

Management of timing: In distributed systems the management of timing is a fundamental issue. A component has to react appropriately if a request is not answered in a determined amount of time (timeout configuration). Alternatively we would like to be able to schedule an activity to begin at a particular date or after a specific amount of time. BPEL provides dedicated constructs, such as *pick*, *onAlarm* and *wait* for timing management.

Exception handling: BPEL provides constructs for dealing with exceptional cases or irrecoverable problems, i.e., a fault handling mechanism. Also, mechanisms for capturing special events that can occur asynchronously with respect to the control flow of the process are provided.

Serializable Scopes: BPEL uses the notion of *scope* for nesting an activity with its own associated variables, fault handlers and compensation handlers. When the activities present in two or more nested scopes need to concurrently access shared variables, a mechanism for concurrency control for such accesses is required. To implement this, BPEL provides special attribute *variableAccessSerializable* for the *scope* construct. Using this attribute ensures that accesses to all the shared variables will be *serializable*. (The semantics of serializable scopes are very similar to the standard isolation level "serializable" [11] used in database transactions.) Suppose two concurrent serializable scopes, S1 and S2, access a common set of variables. Serializability ensures that the results of their behavior would be no different if all accesses of shared variables were first performed by S1 before S2 starts accessing these variables.

B. General observations about the BPEL constructs for transaction management

Our investigation of BPEL has yielded the following general observations of its constructs for transaction management:

- Each synchronization management concern produces tangled code, since each case needs to be managed with at least three constructs (*link*, *source* and *target*) which are scattered into the main business concern.
- The model is complex, and involves a lot of unrelated different constructs. (e.g., three constructs for synchronization management, one construct for compensation, more constructs for timing management, and the functional replication concern is managed as a special case of fault handling).
- It is not extensible. This makes it hard to deal with new transactional problems of a particular business if it is not covered by the language.

After analyzing an alternative for managing transactional concerns in Web Services composition, we come back to these problems later, in order to propose a BPEL extension that overcomes these limitations.

III. The ACTA formal model and the KALA Language

This section briefly describe two proposals that have been the main source of inspiration for defining an alternative transactional model to the one used in BPEL. The information presented here is a summary of [7] and [9].

A. ACTA

The objective of ACTA [6][7] is to provide a formal model that allows us to specify a large number of Advanced Transaction Mechanisms (ATMS). In ACTA, databases are considered as repositories of objects and transactions are considered as invocations of operations on these objects (also called *object events*) or on transaction management primitives, e.g., begin, commit or abort (also called *significant events*). In the model, all these events can be executed concurrently and such executions are logged in the sequence in which they are occurred, creating a history. ACTA can define the properties of an ATMS by reasoning about the properties of event histories [6].

ACTA allows for three kinds of constraints on the transaction history to be declared, they are:

- An event *X* can be constrained to occur only after an event *Y*.
- A particular event *can only* occur in a history if a certain condition is satisfied.
- A particular event *has to* occur in a history if a certain condition is satisfied.

Transaction models specified only in terms of axioms that declare these constraints, although powerful for specifying a great amount of ATMS, have an important drawback: the related specifications are very large and complex. To address these problems, a number of abstraction mechanisms have been defined in ACTA: *dependencies*, *views* and *delegation*. For a complete explanation of these mechanisms, we refer to [6] [7]. Here we solely discuss the concept of dependencies, as they are a key element of our current work. An analysis of the employment of views and delegation in our domain will be done as part of our future research.

Dependencies are constraints on the histories produced by the concurrent execution of interdependent transactions [12]. These constraints can be expressed precisely in terms of the significant events associated with such transactions [6]. Twelve kind of dependencies have been defined in [6] and [7]. We describe three of them as an example, since they are sufficient for expressing the BPEL transactional concerns that will be of our interest in this work. They are:

Begin-on-Commit Dependency.- If a transaction *Tj* has a Begin on Commit Dependency with a transaction *Ti* (or: *"Tj BCD Ti"*) then *Tj* cannot begin executing until *Ti* commits.

Begin-on-Abort Dependency.- If a transaction *Tj* has a Begin on Abort Dependency with a transaction *Ti* (or: *"Tj BAD Ti"*) then *Tj* cannot begin executing until *Ti* aborts.

Compensation Dependency.- If a transaction *Tj* has a Compensation Dependency with a transaction *Ti* (or: *"Tj CMD Ti"*) then if *Ti* aborts, *Tj* must commit.

B. The KALA language

KALA [9] is an aspect language [13] based on the ACTA formal model. It stands for Kernel Aspect Language for ATMS and, since it is based on ACTA, it assures that a wide variety of ATMS can be implemented using this language.

KALA was created as a tool for specifying the transactional properties of Java methods in a separate description. A KALA aspect declaratively specifies the transactional properties of a Java method, using constructs that directly reflect concepts from ACTA. Therefore, the ACTA concepts of dependencies, views and delegation can be easily associated with a method (considered as a transaction) using KALA. The declarations of the transactional properties of methods, can also be set to coincide with any of the begin, commit and abort significant events of a transaction.

In addition, this language adds some complementary notions that have to be taken into account when expressing the ACTA formalism in a real life programming language, such as the idea of *secondary transactions*. The concept of secondary transactions is useful for abstracting the fact that multiple ATMS require to run some transactions, that conceptually are separated from the main control flow of the application, when certain constraints are satisfied. We describe as an example the notion of *compensating transactions*. In some transactional models (e.g., Sagas [14]), the results of a subtransaction belonging to a long-lived transaction are *partially committed* (also referred as *externalized* in [15]). In case that the main transaction aborts, the work done by its subtransactions must be compensated. This compensation is accomplished by compensating transactions that perform a semantic undo. Compensating transactions are not part of the main business control flow, but must be executed only when a certain event occurs (the abort of the main transaction). Therefore, we refer to them as secondary transactions. KALA eases the use of secondary transactions using a special construct provided for indicating that a particular method should be invoked as a secondary transaction.

Finally, at the implementation level, this language is complemented by *ATPMos*, a sophisticated transactional processing monitor. ATPMos, in addition to support traditional transactions, implements support for ACTA primitives. The capability of ATPMos for supporting ACTA abstractions, specially dependencies, makes this TP monitor a fundamental keystone of the design of our solution.

IV. DBCF

Web Services composition using BPEL raises a number of issues related to transaction management, as we have discussed above: production of tangled code in each synchronization management concern, the use of a complex mechanism that involves a lot of unrelated different constructs and the lack of extensibility of the transactional mechanism.

In our research, in order to improve the management of transactions in this domain, we have combined notions from ACTA and KALA. The product of this synthesis is *DBCF*, which stands for *Dependency-Based Composition Framework*. Service composition applications implemented with it have, in addition to the capability of expressing transactional properties of BPEL-like languages in a more straightforward way, added power and extensibility. As a result, these applications can be implemented quickly, with an improved separation of

concerns and are more adaptable to deal with new transactional scenarios.

DBCF follows an architecture based on a hierarchical composition of activities, where each activity can be defined independently of the others. It borrows from ACTA the notion of transactional dependencies and the dependency types already defined in [6] [7]. From KALA it reuses the notion of secondary transactions and ATPMos, the transactional processing monitor that is embedded in the core of our proposal. In addition, it extends these ideas with the notion of *implicit dependencies*, as a mechanism for expressing complex concepts that involve more than a pair of activities.

DBCF has been implemented as a proof-of-concept that these separate ideas can be successfully combined. It defines a powerful, straightforward and unified mechanism for advanced transaction management that is adequate and correct in the context of Web Services composition.

A. Extending the concept of ACTA dependencies

ACTA is a general framework for expressing transactional models. Since its objective is being able to define a great amount of models, its concepts are not related to a specific one. However, our domain is the composition of Web Services, where a hierarchical composition of activities is the model to be used. The knowledge of the model constraints is an advantage to us, since it allows us to make assumptions that cannot be done in ACTA. For example, we always know that every activity in the process has a parent activity, and that possibly more than one activity ancestor is present (since a parent activity could also has a parent). We also know that certain dependencies are frequently used together, in order to express common transactional constraints. As described later in this section, one example of this occurs when expressing the notion of compensation.

Therefore, we can modify in our domain the concept of dependencies used in ACTA. We define that:

- Dependencies are established among pairs of transactions, as in the ACTA framework.
- A dependency can implicitly abstract more than one ACTA dependency. This is because it eases the use of dependencies that are frequently used together in our domain.
- The constraints enforced by a dependency, are however not restricted to the significant events of the pair of transactions declared in the dependency. This is because significant events of additional transactions not mentioned explicitly in the dependency can be considered. Such other transactions can be inferred implicitly in our scenario of services composition, since we know that the relationships among transactions follow a hierarchical composition model.

B. Overview of the design of DBCF

DBCF has been implemented in Java. The main reason for choosing this language is that we already had a proven transaction monitor with support for dependencies among transactions written in this language (ATPMos, mentioned in III).

```
1  ...
2  @WebService(
3  serviceName="ReservationService",
4  name="Reservation")
5  public class ReservationImpl {
6    @WebMethod()
7    public void reserve(...) {}
8    ...
9  }
```

Listing 1. Definition of a Web Service interface

```
1  ...
2  public class AReserveHotelAndCar
3    extends OnMethodCallActivity{
4    ...
5    @WorkflowState
6    ClientInfo clientInfo;
7    ...
8    public boolean correlation(...) {
9      ...
10   }
11   @MethodHandler(methodHandled="reserve")
12   public void onReserve(...) {
13     ...
14   }
15 }
```

Listing 2. Definition of an activity in DBCF

A programmer in DBCF must do the following tasks in order to define a Web Service composition:

- Define the Web Service composition business interface using JAX-WS [16]. Listing 1 is an example of a Web Service interface defined in DBCF. The framework transparently links the Web Service methods defined in this class and the DBCF event listener.
- Define each activity as an individual module. Listing 2 shows an example of an activity definition which represents the reception of a remote Web Service invocation.
- Define the composition of the activities. Listings 3, 4, 5 and 6 are examples of composition of activities.
- Deploy the application as a normal Web Service application in a JAX-WS compatible engine. For our tests, we used Tomcat [17], version 5.0.

Fig. 1 describes the architecture of DBCF. At the lowest level, ATPMos is in charge of dependency management, which constitutes the core of our model. DBCF basically is comprised of a Workflow Manager, an Event Manager, a collection of classes that define the different base-cases of activities of the application, and an aspect library. This aspect library is in charge of abstracting the complexities of two things: first is linking the Web Service composition business interface with our framework; second is the variable lookup that each activity must do every time that it needs to reference state that is declared and initialized in an upper activity.

As we mentioned before, to instantiate the framework, specific activities are implemented by the developer, extending

the provided activities. Next to this, a workflow composition model is defined. This model states how the business activities should be combined. On top of the layer of activities and composition definition, the objects related to JAX-WS are placed. These provide the Web Service interface of our workflow system and a collection of remote and local stubs.

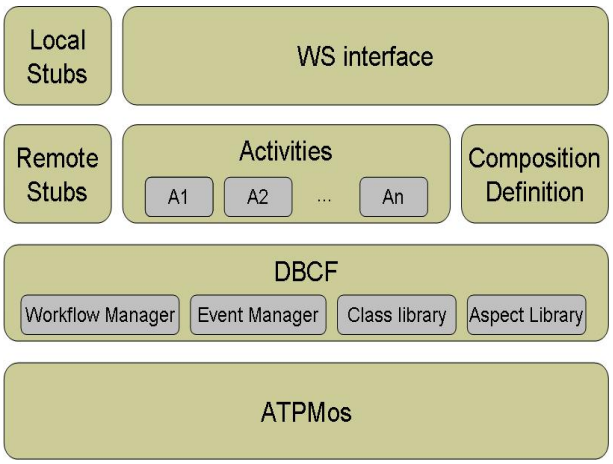

Fig. 1. The DBCF architecture

DBCF implements the basic infrastructural requirements that a BPEL-like composition engine must provide in order to be functional and testable: management of workflow instances, life cycle management, management of external events, management of correlation issues and management of instance state. A detailed discussion on the implementation of the main infrastructural services is however outside of the scope of this paper, for more detail we refer to [8].

DBCF also implements the transactional concerns of synchronization of activities, compensation, functional replication, timing management and exception handling present in BPEL. However, currently only the concerns of synchronization of activities, compensation and functional replication have been implemented using ACTA dependencies as proposed by DBCF. In the current version, timing management has also been implemented, but using alternative mechanisms that mainly simulate the equivalent constructs present in BPEL, and exception handling has been simplified to the common exception handling mechanisms of the Java language. Serializable scopes have not yet been implemented. As part of our future work, we will propose to extend the notion of transactional dependencies, in order to include relevant ideas related to timing management and exception handling. In addition, we would like to explore the possibility of employing the ACTA notions of views and delegation for implementing the BPEL concept of serializable scopes.

We finish this overview with a short description of how the extensibility of the ACTA framework is carried over to DBCF. Two kind of extensions are possible:

- The definition of a new dependency that must be registered in ATPMos. If a new dependency must be recognized by the framework, this is straightforwardly registered in ATPMos, with the specification of a description file. That file declares the name and the constraints of the new dependency.
- The definition of a new dependency that is a composition of already existing dependencies. This is done at a layer on top of ATPMos. When the use of a dependency is declared by the framework, the name is checked before delegating the message to ATPMos. If the name corresponds to a composed dependency, multiple messages for creating a dependency are sent to ATPMos (one for each single dependency present in the composition).

```
1 ...
2 ScopeActivity sa = new ScopeActivity(...);
3 ...
4 AReserveHotel rh = new AReserveHotel(sa, ...);
5 ...
6 AReserveCar rc = new AReserveCar(sa, ...);
7 ...
8 rc.setDependency(rc, "BCD", rh);
9 ...
```

Listing 3. Synchronization handling in DBCF

C. Implementing transaction management concerns in DBCF

To demonstrate the advantages of the use of DBCF, we now discuss how the BPEL transaction management concerns of synchronization of activities, compensation and functional replication can be implemented in our framework, using a more straightforward, unified, and extensible mechanism.

Synchronization of activities: Listing 3 shows an example of a synchronization concern using DBCF. Two invocation activities, *rh* (line 4) and *rc* (line 6) are defined, and a *Begin on Commit Dependency* among them is declared in line 8.

The method *setDependency* can be invoked by any activity, and receives three parameters: the source of the dependency, the identifier of the kind of dependency, and the destination of the dependency. The dependency invoked by this method is applied when the declaring activity starts its life cycle (in this case: *rc*). It is possible also to apply dependencies at *begin*, *commit* or *abort* times.

In our example, the result of applying the aforementioned dependency, is that the activity *rc* will not begin until activity *rh* has begun. This shows how this dependency is useful for specifying a sequential execution order among a collection of activities.

Note that we needed only one construct for expressing a simple synchronization concern (a call to the *setDependency* method in line 8), instead of three BPEL constructs (*link*, *source* and *target*) that would be scattered across the code, for expressing the same idea.

Compensation of activities: Compensation of activities involves the employment of one additional concept: the notion of secondary transactions. Listing 4 shows an activity that plays the role of nesting transaction (*sa* in line 2) and two activities that plays the role of nested transaction (*rh* in line 4 and *ch*

```
1  ...
2  ScopeActivity sa = new ScopeActivity(...);
3  ...
4  AReserveHotel rh = new AReserveHotel(sa, ...);
5  ...
6  ACancelHotel ch = new ACancelHotel(sa, ...);
7  ch.setExecutionMode(
8    ExecutionMode.SECONDARY_ACTIVITY);
9  ch.setDependency(ch, "BCD", rh);
10 ch.setDependency(ch, "BAD", sa);
11 ch.setDependency(ch, "CMD", sa);
12 ...
```

Listing 4. Compensation handling in DBCF with ACTA dependencies

in line 6). This nesting is specified at the moment the two last activities are instantiated (i.e., in lines 4 and 6 respectively).

Once the nesting issues are addressed, the next step is to establish the compensation dependency. Since *ch* compensates *rh*, the former should be defined as a secondary activity. To do this lines 7 and 8 invoke the method *setExecutionMode* and pass the constant value *SECONDARY_ACTIVITY*. Now we need to establish adequate dependencies among the compensated and compensating activity. How these dependencies interact is illustrated in Fig. 2. Line 9 declares that the compensating activity has a *Begin On Commit Dependency* with the compensated activity. As a result the former will not be invoked if the latter has not committed. In addition, a *Begin On Abort Dependency* (line 10) and a *Compensation Dependency* (line 11) are established among the compensating activity and the nesting activity. This ensures that the compensating activity will begin only if the nesting activity aborts and, if the nesting activity aborts, the compensating activity will commit. Note that in this combination of dependencies, the last dependency states that the compensating activity must commit if the parent transaction has aborted. If the parent transaction commits, the compensating activity can commit or abort. However, since a Begin on Abort dependency is also present, this second alternative will never be chosen. This is because the condition for starting the compensating activity is precisely that the parent activity aborts.

As an example of the extensibility of our framework, we define a new kind of dependency for compensation that is equivalent to the three dependencies mentioned before, since they are frequently used together. We call it *CPD* in order to distinguish it from the ACTA compensation dependency *CMD*. This simplified dependency can be defined thanks to the fact that we know the structural relationships between activities in our process (recall that we use a hierarchical composition of activities). Therefore, it is possible to infer all the transactions that are not mentioned in the dependency, but that have implicit relationships with the activities defined on it (in our example, the *sa* activity can be inferred as the parent of *rh*).

Listing 5 shows an example of this dependency, where the dependency declaration in line 9 substitutes the three dependencies in lines 9,10 and 11 of listing 4.

Functional replication of activities: The implementation of a functional replication concern requires, as in the previous case,

```
1  ...
2  ScopeActivity sa = new ScopeActivity(...);
3  ...
4  AReserveHotel rh = new AReserveHotel(sa, ...);
5  ...
6  ACancelHotel ch = new ACancelHotel(sa, ...);
7  ch.setExecutionMode(
8    ExecutionMode.SECONDARY_ACTIVITY);
9  ch.setDependency(ch, "CPD", rh);
10 ...
```

Listing 5. Compensation handling in DBCF with a customized dependency

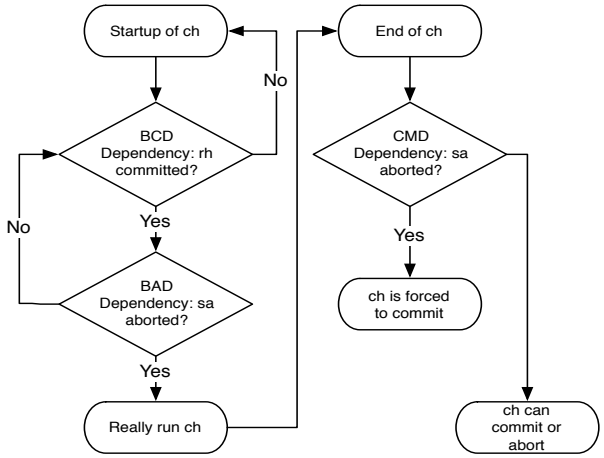

Fig. 2. Flow chart of the ACTA Dependencies of a compensating activity. Note that in the *CMD* dependency the 'Yes' choice will always be taken, as the flow of execution can only enter this dependency if *sa* has aborted, due to the *BAD* dependency.

the definition of a secondary transaction. We show in listing 6 an example of this concern, where the activity *a* is replicated by the activity *b*. Because *b* is an activity that implements a functional replication concern, it is declared as secondary in lines 5 and 6 . Line 7 establishes a *Begin On Abort Dependency* with the primary activity, in such a way that the secondary activity *b* will be executed only if the primary activity *a* fails.

D. Comparing DBCF with BPEL

To conclude this section, we compare DBCF and BPEL, discussing the advantages of each over the other.

Advantages of DBCF over BPEL:

- DBCF is based on an accepted formal model within the advanced transactions community (ACTA). BPEL is not.

```
1  ...
2  ScopeActivity a = new ScopeActivity(sa, ...);
3  ...
4  ScopeActivity b = new ScopeActivity(sa, ...);
5  b.setExecutionMode(
6    ExecutionMode.SECONDARY_ACTIVITY);
7  b.setDependency(b, "BAD", a);
8  ...
```

Listing 6. Functional Replication handling in DBCF

- DBCF inherits the extensible properties of the ACTA framework. Transactional concerns can be expressed with dependencies already defined in previous work [6] [7], or new kind of dependencies can be defined if needed, which eases the use of new transactional models.
- The model proposed in DBCF does not require the tangled code that is produced in BPEL when applying synchronization concerns. In BPEL, a simple synchronization concern needs the scattering of different constructs over the main business concern of the application. In DBCF, only one ACTA dependency has to be defined for expressing the same concern.
- DBCF reduces the number of constructs that are needed for expressing transaction management concerns. The basic concepts of ACTA dependencies with implicit constraints and secondary transactions are well suited for straightforwardly expressing the concerns of *synchronization of activities*, *functional replication* and *compensation management*, as we have shown above.
- BPEL on the other hand, needs three different constructs for expressing *synchronization of activities*, another construct for *compensation management*, and the *functional replication* concerns are tangled with other concerns related with exception handling.

Advantages of BPEL over DBCF. BPEL does possess some transaction-related features that could be used to significantly improve DBCF. These are:

- Exception management is more powerful than what is currently implemented in DBCF. Exceptions in BPEL are propagated following the hierarchical composition of activities, and can be handled by any of the activities that receive the exception. In DBCF, exception management is limited to the Java exception handling mechanism, and if an exception is not caught for an activity, it stops the life cycle of the process instead of being thrown to an upper activity.
- Although most of the timing management features of BPEL have been implemented in DBCF, in BPEL it is possible to express certain sophisticated ideas that in DBCF cannot be expressed. An example is that an activity can be scheduled to be executed on a particular time, but only if the associated scope where the activity is defined still has not finished its execution.
- In addition, BPEL has already defined a mechanism for expressing temporal predicates, while in DBCF this has not yet been implemented.
- The synchronization model of BPEL can use boolean logic expressions in terms of the outcomes of transactions, for expressing conditions in which a specific activity can begin its execution. Though the only constraints that can be expressed are sophisticated *begin on commit* or *begin on abort* dependencies (instead of the broader kind of dependencies that can be specified with DBCF), the possibility of including logic predicates in the expression of dependencies should be taken into account as a future direction of research for DBCF.

V. Extending BPEL to simplify the specification of advanced transaction management

We have discussed a number of important problems and limitations of the BPEL constructs for transaction management, and shown how these are addressed by DBCF. In this section we sketch how our solution can be used to improve BPEL. We recommend alternatives to BPEL constructs, based on the main ideas of DBCF. We provide here semantic explanations of the constructs, using the same notation used in the BPEL specification [2], instead of providing lengthy XSD [18] or DTD [19] formal descriptions.

With our extension, BPEL will have the following advantages:

- It will use a common model with a simple unified set of constructs for transactional management, bringing simplicity to the language, reducing its code size, and accelerating the language learning process.
- Well-defined extension points in the language for transaction management will be provided. In this way, BPEL could evolve more easily in order to deal with new transactional problems.
- The scattering of code related to synchronization of activities will be reduced. Therefore, the final code will be more clear and raise less issues with regard to evolution.

To accomplish this, our proposed extension provides support for the following concepts:

- The notion of associating state with activities, where possible states can be: *non-initiated*, *began*, *committed* or *aborted* (this is an open list of possible states, since more sophisticated transactional models can have additional states).
- The possibility of establishing transactional dependencies between different activities, using the known and extensible dependency model of the ACTA framework.
- The notion of activities that are not part of the business case that the application is trying to solve, but that are related to other non-functional concerns like transaction management.
- The inclusion of the concept of implicit dependencies, in such a way that a single statement can declare complex dependency relationships among more than the two activities explicitly declared, as in DBCF.

We discuss in the rest of this section the constructs that should be added to BPEL in order to implement these ideas.

A. Simple dependencies between activities

Once we have adopted the notion of activities with state in BPEL, the next step proposed in our model is establishing relationships between activities present in a particular BPEL scope using ACTA dependencies. This is a two-part process that consists in first solving identification issues and second setting ACTA dependencies.

Identification issues: The ACTA framework proposes a way of declaring dependencies between pairs of activities (transactions). Therefore, to define ACTA dependencies in a programing language like BPEL, we need an identification mechanism for representing the activities that are declared in the dependencies. We take advantage of the fact that the *name* attribute is part of the collection of attributes defined as *standard attributes* in the BPEL specification. Standard attributes are defined as all such attributes that can be present in any BPEL activity. We have preferred to use this identifier for defining ACTA dependencies between two activities, instead of defining a new dedicated attribute.

Setting ACTA dependencies: We propose to create a construct for defining dependencies between activities, using activity names for source and destination of the dependency, and a string representing the type of the ACTA dependency that is being established between the source and destination activities. The following code shows how the structure of a dependency is written:

```
<dependency type="ncname" source="ncname"?
  destination="ncname" />
```

The *source* attribute represents the source activity, and the *destination* attribute represents the destination activity. The question mark near the *source* attribute, denotes that it is optional, since when omitted, it represents by default the activity that declares the dependency.

The possible values of the type attribute can be the traditional ACTA dependencies that have been discussed in other work [6] [7], or extended to any new sort of dependency that could be defined between BPEL activities. For example, synchronization concerns in BPEL could be expressed with a single dependency construct, instead of the *link* constructs that are currently used.

B. Secondary Activities

Secondary activities, as defined in section III, are activities that are not part of the business concern of the application, but related to other concerns like transaction management. They are associated with a BPEL scope, and they express transactional relationships with the *primary activity* of a scope (or activities nested in it) using ACTA dependencies.

Secondary activities can be represented inside a scope, using the following construct:

```
<scope>
  ...
  <secondaryActivities>
    ...
  </secondaryActivities>
</scope>
```

BPEL scopes nest only one activity, called the *primary activity*. However this activity can be a complex structured activity, which allows for the nesting of more than one activity in a single scope. Normally, a scope is considered as finished when its primary activity has finished.

```
1  <scope name="parent">
2    <invoke name="requestShipping" ... />
3    <secondaryActivities>
4      <invoke name="cancelShipping" ... >
5        <dependency type="BCD"
6          destination="requestShipping"/>
7        <dependency type="BAD"
8          destination="parent"/>
9        <dependency type="CMD"
10         destination="parent"/>
11     </invoke>
12   </secondaryActivities>
13 </scope>
```

Listing 7. Compensation handling using ACTA dependencies

```
1  <scope>
2    <invoke name="requestShipping" ... />
3    <secondaryActivities>
4      <invoke name="cancelShipping" ... >
5        <dependency type="CPD"
6          destination="requestShipping"/>
7      </invoke>
8    </secondaryActivities>
9  </scope>
```

Listing 8. Compensation handling with implicit dependencies

In our extension, we consider that a scope cannot be considered as finished, if secondary activities are running at the moment of the completion of its primary activity. If that is the case, the scope has to wait for the completion of these activities before it is considered as finished. If no secondary activities are running at the moment of the completion of the primary activity, the scope is considered as finished.

A scope should not wait for secondary activities that have not begun after completion of the primary activity. This is firstly because some of these activities may never be executed at all as a result of their dependencies. Secondly these could begin in an undetermined moment in the future, when their begin dependencies with activities not belonging to the current scope are satisfied. Therefore, if the scope waits for such non-crucial activities in order to be considered as completed, it would indefinitely block the execution of other activities that are waiting for this activity to finish.

C. Implicit dependencies between activities

The implicit dependencies concept we developed for DBCF can be also added to BPEL. This accomplishes a significant simplification to the way transactional dependencies can be declared. We demonstrate this next, through an example.

Listing 7 shows the basic implementation of a compensation management concern. We can see here that only two simple activities are declared: the primary activity *requestShipping* declared in line 2, and the secondary activity *cancelShipping* declared in lines 4 to 11. This secondary activity declares three ACTA dependencies (lines 5 to 10) that configure *cancelShipping* to be executed only as a compensation transaction of *requestShipping*.

Listing 8 proposes a simpler implementation, using an

implicit dependency (lines 5 and 6). This dependency replaces the three dependencies showed in listing 7.

D. Suppressing failures

To be fully compatible with the idea of associating state with BPEL activities, the only additional concept that should be added to the language is the notion of "failed activities". We have defined a failed or aborted activity as an atomic or structural activity in BPEL that has already been unsuccessfully executed. This lack of success can be due to various circumstances. In the case of an invocation activity for example, it can be the presence of a fault in the output message of a web service invocation. In other more complex cases, such as structural activities (activities that enclose other activities), a failure can be due to the throwing of a fault in one of the internal activities.

The standard behavior in BPEL when this kind of problems arise, is to manage the fault if a fault handler exists in the scope where such a fault happened. If no appropriate fault handler is available, the fault is thrown to the enclosing scope. This behavior is not compatible with our model, where sometimes a fault occurrence only means assigning the aborted state to the activity where this fault occurred, without implying the throwing of the fault notifying a problem to the enclosing scope.

We propose a simple syntactic construct that expresses this idea, and that could be added to the set of BPEL *standard-elements*. The BPEL specification defines the standard elements as a set of constructs that can be potentially included in any activity. This set of elements is currently limited to *source* and *target* links. The additional element we propose is:

```
<suppressFailure>
  boolean
<suppressFailure/>
```

The inclusion of a *suppressFailure* element with a true value inside a BPEL activity would be semantically equivalent to the inclusion of an implicit scope immediately containing the activity, with a dummy fault handler. The following code illustrates this:

```
<scope>
  <activity ... />
  <faultHandlers>
    <catchAll />
  </faultHandlers>
<scope/>
```

An example of the use of *suppressFailure* is functional replication. In this setting, more than one service can be invoked in order to supply an equivalent behavior (from the client point of view). Listing 9 defines a *requestShipping* service as a first invocation alternative (lines 2 to 4), but in case of problems, there is the alternative of invoking the *requestShippingAlt* service (lines 6 to 9). We can express this transactional relationship with the ACTA *begin on abort* dependency previously discussed (lines 7 and 8).

```
1  <scope>
2    <invoke name="requestShipping" ... >
3      <suppressFailure>true</suppressFailure>
4    <invoke>
5    <secondaryActivities>
6      <invoke name="requestShippingAlt" ... >
7        <dependency type="BAD"
8          destination="requestShipping"/>
9      </invoke>
10   </secondaryActivities>
11 </scope>
```

Listing 9. Functional replication with a *suppressFailure* element

Here, the *requestShipping* service plays the role of primary activity, and *requestShippingAlt* is the only secondary activity declared in the scope. In this example we consider that *requestShipping* throws a fault when invoked. Since the activity declares a *suppressFailure* element (line 3), the fault is not propagated to an outer scope, but the state of this activity is changed to *aborted*. After the scope has finished the execution of its primary activity, it checks the state of its secondary activities before being able to change its state to *committed*. Since one of its secondary activities, namely *requestShippingAlt* has already started (because its begin on abort dependency has been satisfied), the scope should wait for the finalization of this activity before completing. If *requestShippingAlt* succeeds, the scope changes its state to *committed*. Conversely, if this activity fails, the scope changes its state to *aborted* and the fault is thrown to the outer scope.

VI. Summary

The objective of this work is to advance the current mechanisms for transaction management in Web Service composition languages. Using a unified mechanism of ACTA dependencies we were able to achieve significant improvements with regard to avoiding tangling of concerns of BPEL code, conciseness of BPEL code and the extensibility of the BPEL transaction mechanism. We discussed these improvements in this paper.

First, we demonstrated the flexibility of the ACTA framework, as a base for developing a mechanism based on transactional dependencies. This system unifies the different existing BPEL mechanisms for dealing with advanced transaction management concerns into one element.

Second, we adapted and extended the dependency model of ACTA in such a way that it better fits our domain. We used the notion of secondary transactions [9] and we added the concept of implicit dependencies between activities.

Third, we demonstrated the validity of these ideas for implementing transaction management in Web Service composition, through the implementation of the DBCF framework. DBCF is a prototype that makes use of the main directions of the above model.

Fourth, we proposed an extension to BPEL, the current state of the art for Web Services composition. Our extension uses the main ideas developed in DBCF and significantly improves on the BPEL mechanisms for managing the transactional

concerns of activity synchronization, functional replication and activity compensation.

DBCF solves three BPEL problems: First, the tangling of code in synchronization management concerns. Second, the complexity of the model, created by the use of an unrelated group of constructs for each transaction management concern. Third, the lack of extensibility of the transactional mechanism used by BPEL.

These problems are solved as follows:

- The proposal that transaction management concerns in Web Service composition can be managed using the core ideas of dependencies and secondary transactions. As a consequence there is a reduction in the number of constructs and the amount of concepts that need to be memorized by a programmer.
- DBCF also proposes an extensible mechanism for transaction management inherited from ACTA. This results in a mechanism that is adaptable, it will be able to deal with new transactional issues in the future.
- DBCF introduces the notion of implicit dependencies, as a mechanism for accomplishing further simplification.

VII. Conclusions

Transaction management in loosely coupled environments is not a trivial task. Web Services are the current state of the art for interoperability for such kind of systems, and the BPEL language is the current de-facto standard for web services composition. BPEL support for expressing transactional properties is however limited, non extensible, and relies on different unrelated constructs for managing transaction management concerns. In this paper we have demonstrated that these concerns can be managed using a simpler, extensible and unified set of constructs, relying on a proven existing formal model for advanced transaction management. Furthermore, we have shown how BPEL can be extended to incorporate our solution. As a result of this the BPEL programmer reaps the benefits of more concise code that is less tangled, and the BPEL language is more extensible towards new transactional models in the future.

VIII. Future work

Possible avenues for future work are mainly oriented to improve DBCF, so that transaction management concerns that are currently managed by BPEL are better handled. As we mentioned before, these are mainly related to timing management, exception handling and the use of multiple activities to express synchronization management.

In order to solve these problems, additional extensions to the concept of ACTA dependencies can be defined in such a way that a dependency statement can also be expressed with:

- Temporal logic elements.
- Inclusion of elements related to exception handling.
- The use of more than two activities in dependencies, relating their transactional outcomes with boolean logic expressions.

Also, we would like to explore the possibility of employing the ACTA notions of views and delegation for implementing the BPEL concept of serializable scopes. These improvements involve the modification of ATPMos, the transactional monitor taken from the KALA implementation and used by DBCF.

Another interesting research direction in DBCF, is the definition of a method for formally establishing implicit relationships between transactions. This should allow the concept of implicit dependencies to be used in different scenarios, and not only in the context of Web Services composition.

Finally, a transformation engine could be developed that can transform BPEL processes written in XML, to modules written in DBCF, such that these processes can be executed with the services of our framework. To achieve this, the XML document that defines a BPEL process should be parsed, and each BPEL activity should be mapped to a corresponding DBCF activity.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services. Concepts, Architectures and Applications*. Springer, 2004.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, and J. Klein, "Business Process Execution Language for Web Services (version 1.1)," http://www.ibm.com/developerworks/library/specification/ws-bpel/, May 2003.

[3] A. Arkin, "Business Process Modeling Language- BPML 1.0," BPMI Consortium, Tech. Rep., June 2002.

[4] F. Leymann, "Web Services Flow Language (version 1.0)," http://www.ebpml.org/wsfl.htm, May 2001.

[5] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[6] P. K. Chrysanthis and K. Ramamritham, "A formalism for extended transaction model," in *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 103–112.

[7] ——, "ACTA: The SAGA continues," in *Database Transaction Models for Advanced Applications*, 1992, pp. 349–397.

[8] S. Castro, "Acta Dependencies as a Unified Mechanism for Compensation, Activities Synchronization and Functional Replication in BPEL4WS," Master's thesis, Vrije Universiteit Brussel, 2006.

[9] J. Fabry and T. D'Hondt, "KALA: Kernel aspect language for advanced transactions," in *Proceedings of the 2006 ACM Symposium on Applied Computing Conference*, vol. 2. ACM Press, 2006, pp. 1615 – 1620.

[10] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz, "A multidatabase transaction model for interbase," in *Proceedings of the sixteenth international conference on Very large databases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 507–518.

[11] K. Ramamritham and P. K. Chrysanthis, "A taxonomy of correctness criteria in database applications (*)," *VLDB Journal: Very Large Data Bases*, vol. 5, no. 1, pp. 85–97, 1996.

[12] P. K. Chrysanthis and K. Ramamritham, "Synthesis of extended transaction models using ACTA," *ACM Trans. Database Syst.*, vol. 19, no. 3, pp. 450–491, 1994.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[14] H. Garcia-Molina and K. Salem, "Sagas," in *ACM Conference on Management of Data, pages 249-259*, May 1987.

[15] H. Wachter and A. Reuter, "The ConTract model," *Database transaction models for advanced applications*, pp. 219–263, 1992.

[16] "The JAX-WS Project," https://jax-ws.dev.java.net/, Sun.

[17] "Apache Tomcat," http://tomcat.apache.org/, The Apache Software Foundation.

[18] E. van der Vlist, *XML Schema*. O'Reilly Media, Inc., 2002.

[19] P. Flynn, *Understanding SGML and XML Tools*, 1st ed., ser. Electronic Publishing Series. Kluwer Academic Publishers, August 1998, no. 0-7923-8169-6.