A Comparison of WS-BusinessActivity and BPEL4WS Long-Running Transaction

Patrick Sauter[1], Ingo Melzer[2]

[1]Universität Ulm, Fakultät für Informatik, 89069 Ulm, Germany
ps9@informatik.uni-ulm.de
[2]DaimlerChrysler AG Research and Technology, Postfach 2360, 89013 Ulm, Germany
paper@ingo-melzer.de

Abstract. Although WS-BusinessActivity and BPEL4WS Long-Running Transaction (LRT) are conceptually very similar and are both designed to support the execution of complex business transactions, they differ in a large number of aspects. This is particularly true because BPEL4WS, unlike WS-BusinessActivity, was not designed to support distributed coordination. This paper comprehensively discusses the similarities and differences between WS-BusinessActivity and BPEL4WS LRT and demonstrates the two concepts on the basis of a joint example. The proposal is to replace BPEL4WS' concept of compensation handlers with a more comprehensive handler type – coordination handlers – that communicate only via SOAP messages and thus make WS-BusinessActivity redundant.

1 Introduction

Within less than three years since its initial release, the Business Process Execution Language for Web Services (BPEL4WS [1]) has gained wide industry and research acceptance. BPEL4WS' goal is to describe, coordinate, and execute complex business processes by combining Web Services with workflow concepts. BPEL4WS will serve as a general framework for composing existing Web Services into coarser-grained, more complex, and possibly long-running applications. The Long-Running Transaction (LRT) coordination protocol is part of the BPEL4WS specification and is a mechanism for dealing with errors during such a long-running activity.

WS-BusinessActivity, on the other hand, is part of the Web Services Transaction Framework (WSTF) which also consists of WS-Coordination [2] and WS-AtomicTransaction [3]. The main purpose of WS-BusinessActivity [4] is to coordinate long-running, compensation-based activities that may consist of several AtomicTransactions.

As a result, it might seem that WS-BusinessActivity and BPEL4WS LRT are rather unrelated approaches to transactions or activities of long duration. The BPEL4WS specification [1] itself states that “the achievement of distributed agreement is an orthogonal problem outside the scope of BPEL4WS, to be solved by using the protocols described in the WS-Transaction specification”. (Notice that WS-Transaction now is deprecated and has been split into WS-AtomicTransaction and WS-BusinessActivity.) In this paper, however, we will argue that the two concepts are

not that different after all and can be merged to form a single modeling tool for any kind of long-running transaction.

Therefore, the contribution of this paper is

- the discussion why WS-BusinessActivity and BPEL4WS LRT are neither orthogonal nor contradicting approaches to complex business transactions and
- the description of how their differences can be overcome by fully incorporating WS-BusinessActivity into BPEL4WS.

2 Related Work

Both BPEL4WS and WS-BusinessActivity are relatively new specifications that have emerged during the last few years. Although they clearly are not competing specifications for the same purpose, they share both their underlying transaction model of so-called "open nested transactions" and the idea to invoke explicitly coded compensating actions in the event of failure during the execution of a transaction. These concepts will now be introduced briefly:

In short, an **open nested transaction** is a tree (of arbitrary height) of so-called "subtransactions". Open nested transactions are the generalization of nested transactions which are sometimes also referred to as "*closed* nested transactions". The children of a *closed* nested transaction may commit only when the parent commits. As a result, the overall transaction commits when the root commits, with no individual part of work to be completed (committed) earlier [5, 6]. This limitation does not make sense for long-running, distributed transactions, because it would imply that locks on resources have to be kept for a long period of time until the root commits. Therefore, the subtransactions of an *open* nested transaction (which is mainly used for *distributed* transactions) may commit independently of each other without having to wait for the root transaction to commit.

The next important question on open nested transactions is what the parent transaction should do if one of its child subtransactions has failed. Basically, this behavior is left to the implementor of the transaction – he may decide whether the overall transaction should abort or simply ignore the failed subtransaction. For example, an ordering system that chooses the cheapest supplier might still be able to commit successfully if only one of the suppliers fails during the transaction.

The concept of open nested transactions has not been incorporated into WS-BusinessActivity and BPEL4WS LRT without adaptations. In particular, both are using only a "variant" [1] of open nested transactions and therefore use the terms "nested scopes" (and "nested activities") instead of "subtransactions". A **scope** is the definition of a logical unit of work as well as the smallest unit of error handling and can best be compared with a "try" block in Java. Since scopes can be **nested**, the resulting structure can also be regarded as a tree.

In BPEL4WS, every scope can be assigned a compensation handler. **Compensation** refers to the idea of invoking explicitly coded business logic to undo the effects of a successfully committed action or transaction. A scope's compensation handler therefore contains the appropriate compensation logic, e.g. a WSDL `portType` reference.

Each of these three concepts – open nested transactions, nested scopes, and compensation – has been incorporated into the specifications of both WS-BusinessActivity and BPEL4WS. In particular, the idea of coordinating scopes has been incorporated into BPEL4WS by means of the Long-Running Transaction (LRT) coordination protocol [1, Section 13.2 and Appendix C]. In effect, LRT directly uses the names of states and state transitions of WS-Transaction for coordination among BPEL4WS scopes. The differences of how nested scopes are used by the two specifications will be discussed in the subsequent section.

Apart from WS-BusinessActivity, there are two other important specifications related to transactions for Web Services: WS-Coordination and WS-AtomicTransaction.

In short,

- WS-Coordination provides the protocol for distributing the coordination context of a transaction (e.g. a unique transaction ID) to its participants. For example, WS-Coordination specifies the interface of a transaction manager (a so-called coordinator) for creating a new or joining an already existing transaction. Both WS-AtomicTransaction and WS-BusinessActivity are so-called *coordination types* that are built on top of WS-Coordination.
- A WS-AtomicTransaction is a short-lived (though not necessarily fully ACID-compliant [7]) transaction implementing the two-phase commit (2PC) protocol in terms of Web Services. Typically, an AtomicTransaction is used for locking resources exclusively and sending the `Rollback` notification in the event of failure.

In this context, WS-BusinessActivity can be seen as a framework for putting together a large number of AtomicTransactions (that all share the same `CoordinationContext` as provided by WS-Coordination) that are compensated when the overall BusinessActivity fails. In contrast to an AtomicTransaction, a BusinessActivity is long-running and typically asynchronous. These three specifications are also referred to as the Web Services Transaction Framework (WSTF).

Another noticeable specification related to transactions for Web Services is the Business Transaction Protocol (BTP, [8]) by OASIS. This paper, however, focuses on the WSTF, because it currently is the most widely accepted specification and the two Web Services heavyweights, IBM and Microsoft, are backers of both the WSTF and BPEL4WS.

Moreover, we will analyze the distinction between WS-BusinessActivity and BPEL4WS. The October 2003 paper “The Next Step in Web Services” [9] suggests that these two specifications, among others, should be used in combination. Curbera et al. explicitly consider the combined use of BPEL4WS, WS-Transaction (now: WS-AtomicTransaction and WS-BusinessActivity), and WS-Coordination and state that WS-BusinessActivity (together with WS-Coordination) should be used “in environments where BPEL4WS scopes are distributed or span different vendor implementations”. In other words, several BPEL4WS workflows should register as participants to join a BusinessActivity, i.e. BPEL4WS is “smaller” or finer-grained than WS-BusinessActivity. Figure 1 illustrates this idea.

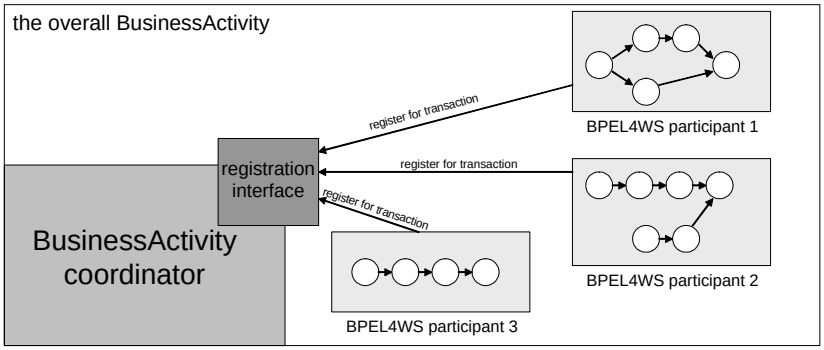

Figure 1 shows the notion of Curbera et al. Here, many "smaller" BPEL4WS instances register for participation in a "large" BusinessActivity.

In the subsequent sections, we will show that there are alternative views on the relationship between BPEL4WS and WS-BusinessActivity.

3 Differences between WS-BusinessActivity and BPEL4WS LRT

There are several similarities as well as an even greater number of differences between WS-BusinessActivity and BPEL4WS Long-Running Transaction. This section rigorously lists these differences and similarities (the latter are denoted in *italics*) as a table and then discusses the most crucial deviations. We will later argue that the differences can be overcome by simply prescribing the manner in which a BPEL4WS LRT communicates with its nested (child) scopes.

Table 1 compares the key features and characteristics of BPEL4WS Long-Running Transaction with those of WS-BusinessActivity.

	BPEL4WS LRT	WS-BusinessActivity
paradigm	orchestrate a flow of Web Services towards a coarser-grained (higher-level) service; act as a wrapper for a flow of "smaller" services	coordinate a set of distributed Web Services (e.g. AtomicTransactions) to reach a mutually agreed outcome; also includes WS-Coordination
number of participants	pre-determined; all potentially involved types of partners (their WSDL descriptions) are known at binding-time	dynamic; participants may join or leave the BusinessActivity at any time as long as they implement the BusinessActivity WSDL interface
error messages	errors (fault and compensation) are handled internally, and no explicit error messages are sent; instead, e.g. a `compensate()` method is called	explicitly described by the specification: error messages (so-called "notifications", e.g. `Compensate`) to the participants are sent as SOAP messages
business error handling concept	*nested scopes and compensation; the unit of error handling is a scope*; a scope is a set of local activities	*nested scopes and compensation; the unit of error handling is a scope*; a scope is a set of distributed activities

fatal error handling concept	the occurrence of a BPEL4WS fault causes the entire scope to exit; the already completed activities are compensated and a fault handler is invoked; similar to try-catch-blocks	go to state `Faulting`; this state cannot be reached directly from the `Completed` state, because compensation has to be tried first; `Faulting` means that a compensation attempt has failed
place to implement the compensation handler	each scope is assigned a dedicated compensation handler that is invoked if the entire scope has to be compensated; therefore, a service might have multiple compensation handlers	the BusinessActivity-compliant service itself must understand and be able to process the `Compensate` notification
short-running transaction support	*may consist of several AtomicTransactions*	*may consist of several AtomicTransaction*
designed for…	*complex long-running transactions* (so-called "activities")	*complex long-running transactions*
scope of the specification	136 pages; complex workflow semantics described; defines its own coordination protocol (LRT)	21 pages; focus not on describing semantics, but on states and state transitions; defines two slightly different coordination protocols
order of activities/ steps	described in detail by the process description, i.e. the `.bpel` file	has to be defined by other means, e.g. has to be hard-coded or the order may even be arbitrary; only the coordination message flow of the overall activity/transaction is pre-defined

Many of the differences listed in Table 1 are implied by the differing intended purposes of the two specifications – graph-oriented workflow description and execution on the one hand, distributed transactions on the other hand. However, because a coordinated and mutually agreed outcome of an activity is also an important quality aspect of business-critical (BPEL4WS) workflows, the LRT protocol was added to BPEL4WS. Unfortunately, one thing was forgotten: the ability to coordinate *distributed* scopes. LRT supports only the coordination of scopes that are local within the same BPEL4WS engine. As a result, WS-BusinessActivity is still required for all distributed long-running transactions. Figure 2 illustrates this relationship.

WS-BusinessActivity, in turn, has two main problems associated with it:

- It does not offer much additional functionality for coordinating long-running transactions compared to BPEL4WS LRT.
- WS-BusinessActivity is a very simple mechanism intended to support very complex operations. It does not provide mechanisms for dealing with complex activity flows. Consequently, a lot of work is left to the implementor and much of the transaction's business logic has to be hard-coded.

The more complex a transaction is, the more does it make sense to model it as a workflow. For example, a very complex financial transaction that involves debiting and crediting a large number of accounts in a particular order can be modeled more easily as a BPEL4WS workflow than as a BusinessActivity. This is because WS-

BusinessActivity does not provide constructs such as sequences, branches, iterations, etc. that make the actual flow of the transaction's steps more explicit.

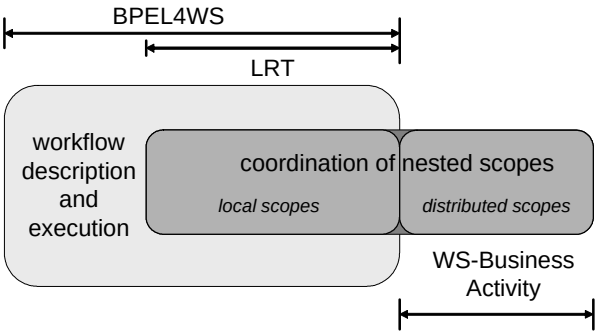

Figure 2 depicts the relationship between BPEL4WS, LRT and BusinessActivity in a set-style notation.

The next section demonstrates the implications of BPEL4WS LRT's lacking distributed coordination support for the implementation of a typical transactional Web Service. Later, in Section 5, we will conclude that this limitation of BPEL4WS can be overcome with little effort, thereby making WS-BusinessActivity redundant.

4 Usage Scenarios and Associated Problems

Reaching distributed agreement is the most important aim of WS-BusinessActivity. In this paper, we will use the example of a patient tracking system in a hospital to demonstrate a typical usage scenario for distributed coordination.

Consider the following situation: A hospital wants to give every stationary patient a unique number, possibly the combination of a person ID with a residence number, and coordinate the treatment by always referencing this number. Part of the treatment might be e.g. appointments in the rehabilitation center or the department of cardiology. Since a hospital essentially is a set of distributed wards interacting with each other, the hospital decides to use Web Service standards for its patient tracking system. They start with providing every patient with a WS-Coordination `CoordinationContext` [2] that consists of his person ID and residence number.

To avoid schedule collisions, every arrangement of an appointment is coordinated by means of WS-AtomicTransaction. For example, whenever a patient must undergo some treatment in multiple wards, the involved wards start an atomic (possibly even ACID) transaction. Furthermore, the treatment within an individual ward might be complex and consist of several sequences, branches, and other typical workflow elements. For example, if the patient's blood pressure is too high, some additional examination steps might have to be performed. Therefore, the wards internally use BPEL4WS implementations (possibly of different vendors) to coordinate the process of treating the patients.

On top of WS-Coordination, WS-AtomicTransaction, and BPEL4WS, the hospital eventually uses WS-BusinessActivity to coordinate the multiple wards' BPEL4WS implementations. For example, if the patient suffers a heart attack during his appointment in the rehabilitation center, the emergency ward (more precisely, its BPEL4WS instance) joins the overall activity as a participant by registering with the WS-Coordination service and starts the emergency treatment. Moreover, some compensating actions have to be invoked to cancel the (previously successfully committed) appointment with the department of cardiology.

The resulting architecture of the hospital's patient tracking system might be very similar to Figure 1. For example, the participants 1 and 2 shown in Figure 1 might be the department of cardiology and the rehabilitation center, whereas the emergency ward might be the third participant joining the overall BusinessActivity only later, shortly after the patient has suffered the heart attack. In this architecture, all coordination messages are transmitted using dedicated SOAP messages and are WS-BusinessActivity notifications as described in [4].

It is not possible to implement this scenario using only BPEL4WS, because coordination of *distributed* scopes is not supported. Only local scopes, i.e. sets of service invocations, can be coordinated using the LRT protocol of BPEL4WS. In the next section, we will show how BPEL4WS has to be extended to be able to fully implement distributed coordination scenarios such as the hospital example.

5 Fully Incorporating WS-BusinessActivity into BPEL4WS

5.1 The State of the Art

Every BPEL4WS instance is a Web Service that invokes other Web Services. These “smaller” services can therefore also be classified as “nested” or “finer-grained” Web Services. For example, the department of cardiology's BPEL4WS workflow might invoke the Web Service that calculates the costs of the treatment. This computation can be a complex task that might involve several legacy database queries, and so the treatment calculation Web Service might be a (finer-grained) BPEL4WS workflow itself.

The main question now is: Why couldn't the program that coordinates the distributed scopes (up to now: the BusinessActivity coordinator) be a BPEL4WS workflow as well? The answer is quite simple: It could, if only the way in which BPEL4WS coordination signals (e.g. the `compensate()` method invocation) are sent was prescribed to be SOAP notifications and not left to the implementor. If this was the case, no difference would have to be made between local and distributed coordination. To achieve this, we will introduce a new dedicated handler type for coordinating distributed scopes in Section 5.2. BPEL4WS already includes three handler types – event handlers, fault handlers, and compensation handlers.

The first idea might be to use one of the already existing handler types for coordinating distributed scopes, but it turns out that neither of them is suitable for this purpose, because all of them are only required to accept some kind of local “signals” such

as implementation-defined method invocations, e.g. `compensate()`. Distributed coordination, however, requires explicit (SOAP) messages instead of local “signals”.

The second idea might be to use the most suitable handler type for distributed coordination, the compensation handler, and change the BPEL4WS specification so that the “compensate” signal has to be transmitted by means of a `Compensate` SOAP notification. However, compensation alone is not yet full-fledged distributed coordination.

5.2 The Coordination Handler Concept

Accordingly, a new type of handler is required to process all notifications related to coordination. This handler type will be called a *coordination handler*. Like each of the other handler types, a compensation handler can be attached to either the entire process or to an individual scope. The crucial difference, however, is that coordination handlers accept signals only as explicit SOAP notifications (and not as internal method invocations) that can also be sent from outside the local BPEL4WS instance. As a result of introducing a handler type for all coordination messages including `compensate`, BPEL4WS' compensation handler will become redundant.

Every coordination handler must be able to process the notifications as defined in WS-BusinessActivity. The set of BusinessActivity notifications is subdivided into coordinator-generated and participant-generated messages. In this context, the “coordinator” is the BPEL4WS activity that *calls* a (local or distributed) coordination handler, e.g. the emergency ward's BPEL4WS instance shown in Figure 3. Consequently, these coordinator-generated messages have to be *processed* by every coordination handler, namely: `Completed`, `Fault`, `Compensated`, `Closed`, `Canceled`, `Exit`, `GetStatus`, and `Status`. In turn, every coordination handler must be able to *send* WS-BusinessActivity's participant-generated notifications, more precisely: `Close`, `Cancel`, `Compensate`, `Faulted`, `Exited`, `GetStatus`, `Status`, and maybe also `Complete` (which is only part of the BusinessAgreementWithCoordinatorCompletion protocol of WS-BusinessActivity [4]). These reply messages flow in the reverse direction of the arrows shown in Figure 3.

In order to fully support distributed coordination, it is essential to support not only the set of SOAP notifications specified by WS-BusinessActivity, but also the basic commands of the underlying WS-Coordination specification [2]. These are in particular `Register` and `RegisterResponse`, `CreateCoordinationContext` (to start a new transaction) and `CreateCoordinationContextResponse`. Since probably not all BPEL4WS instances actually require transactional behavior, providing a scope with a coordination handler should be optional.

Since the coordinator and the participant must be able to communicate with each other during the course of the transaction, we suggest using the WS-Notification specification family [10] for establishing the two-way connection of SOAP notifications between the coordinator and the participant. Therefore, it would in particular be required for the WS-Coordination `Register` and `RegisterResponse` notifications to include the contents of a WS-Notification `Subscribe` message, i.e. the coordinator must explicitly register with the participant for the coordinator's coordination messages and vice versa.

5.3 Impact on the Hospital Example

The capability of distributed coordination implies several architectural changes to the hospital example described in Section 4. First of all, the most important simplification is that WS-BusinessActivity, which had been responsible for coordinating the distributed BPEL4WS instances, is no longer used. Instead, the functionality of the Business-Activity-compliant coordination service, probably a hard-coded service surveying the invocations of the registered BPEL4WS instances, is taken over by an explicitly modeled BPEL4WS workflow. Figure 3 shows what this coordinating workflow might look like for the hospital example. Some important aspects of this implementation will now be discussed.

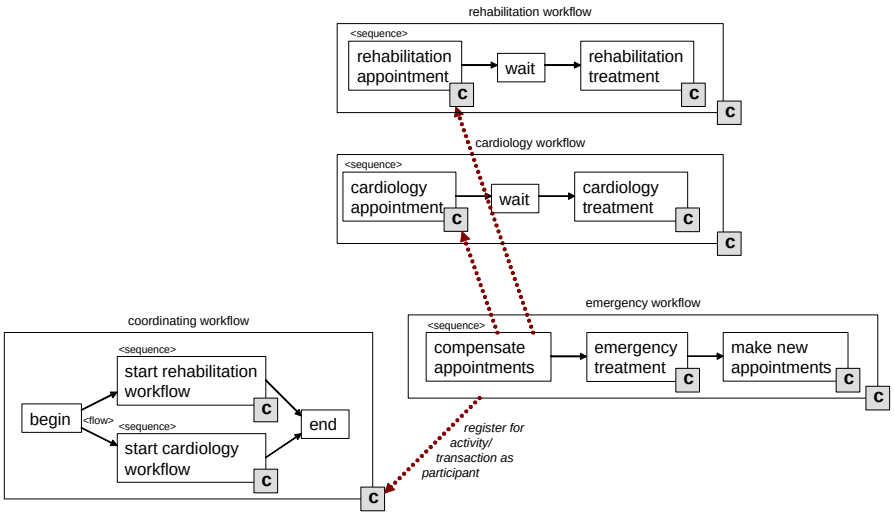

Figure 3 shows a possible architecture of the hospital example as a BPEL4WS workflow with coordination handlers. Boxes with "c" represent the coordination handlers; the dashed arrows indicate the flow of the emergency workflow's SOAP notifications.

Let's first consider the coordination handler of the coordinating (i.e. overall patient treatment) workflow. In order to support the ad-hoc change to its set of active partners when the emergency ward joins the transaction, the overall treatment workflow instance must be able to process the `Register` notification (i.e. "join the existing transaction/activity as a participant"). The emergency ward's workflow then has to compensate all successfully made appointments, call the appropriate heart attack treatment BPEL4WS activities (not shown in Figure 3), and possibly make new appointments.

An important difference to an implementation on the mere basis of BPEL4WS' already specified compensation handlers is as follows: The emergency workflow itself is now able to compensate individual steps of the cardiology and rehabilitation workflows (see the two upper arrows). This would not be possible with compensation handlers, because they only accept signals from within the same workflow engine. As a result, BPEL4WS scopes can now be coordinated even if they are distributed.

5.4 General Advantages

In general, the main advantages of using BPEL4WS with coordination handlers instead of WS-BusinessActivity for coordinating distributed scopes can be listed as follows:

- No distinction has to be made between the implementation of local and distributed scopes; both can be implemented by means of BPEL4WS. A local scope differs from a distributed scope only by the fact that there is no complete implementation of the coordination handler, because accepting the `Compensate` notification is sufficient for local coordination.
- Code redundancy is minimized, because the compensation business logic is kept at the respective scope's coordination handler itself and does not have to be copied to the invoking activity (as this is the case for compensation handlers).
- The coordination service does not have to be hard-coded. Instead, it can be described as an explicit BPEL4WS “coordinating” workflow, thereby minimizing the number of mechanisms for implementing long-running business workflows.
- Compensation of individual activities can be triggered by activities that are not within the same workflow engine, e.g. by the emergency workflow.
- Since also the messages defined by the WS-Coordination specification are supported by every coordination handler, even WS-AtomicTransaction could be incorporated into BPEL4WS. In order to support atomic distributed coordination, a coordination handler would only have to be prescribed to additionally support the set of SOAP notifications described in the WS-AtomicTransaction specification.

As a result, replacing compensation handlers with coordination handlers leverages BPEL4WS LRT to support distributed transactions. Basically, doing so does not change anything about BPEL4WS' underlying workflow and nested scope concept, but only makes BPEL4WS scopes more flexible. When looking at Table 1 again, it becomes obvious that all differences have been overcome by adding the coordination handler concept to BPEL4WS. In particular, the business and fatal error handling concepts have been consolidated, and the number of participants of a BPEL4WS instance has become dynamic since the process' coordination handler now is able to process the `Register` notification.

6 Conclusions

Initially, BusinessActivity and BPEL4WS Long-Running Transaction seemed to be rather orthogonal concepts with different or even incommensurate goals and paradigms. But when having a closer look at the two specifications, the main difference turns out to be BPEL4WS' lacking ability to support *distributed* coordination, although its Long-Running Transaction protocol already supports the coordination of *local* BPEL4WS scopes.

The suggestion of this paper is that BPEL4WS’ compensation handlers should be replaced by more powerful *coordination handlers* which accept coordination signals among nested scopes only as SOAP messages. As a result, WS-BusinessActivity is not needed any longer, and we have demonstrated the advantages of using the approach without WS-BusinessActivity on the basis of a hospital's patient tracking system.

Moreover, coordination handlers could also be used for reaching atomic agreement among distributed, short-running BPEL4WS scopes and thereby also including the functionality of WS-AtomicTransaction. Eventually, we believe that a single aim – in this case, the implementation of long-running transactional activities – should be pursued by a single powerful mechanism only.

Acknowledgement

This paper was written as part of a Web Services research project at DaimlerChrysler Research and Technology in Ulm, Germany and a diploma thesis at the Department of Applied Information Processing (SAI) of Prof. Schweiggert at the University of Ulm.

References

1. S. Thatte et al. Business Process Execution Language for Web Services. Version 1.1. May 2003. Available at http://www.ibm.com/developerworks/library/ws-bpel/
2. D. Langworthy et al. WS-Coordination specification. September 2003. Available at http://www-106.ibm.com/developerworks/library/specification/ws-tx/#coor
3. D. Langworthy et al. WS-AtomicTransaction specification. September 2003. Available at http://www-106.ibm.com/developerworks/library/specification/ws-tx/#atom
4. D. Langworthy et al. WS-BusinessActivity specification. January 2004. Available at http://www-106.ibm.com/developerworks/library/specification/ws-tx/#ba
5. J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Series in Data Management Systems. 1992.
6. F. Leymann, D. Roller. Production Workflow: Concepts and Techniques. Prentice Hall. 2000.
7. J. Gray. The Transaction Concept: Virtues and Limitations. In Proceedings of the 7th International Conference on Very Large Data Bases. Pages 144-154. September 1981.
8. A. Ceponkus et al. Business Transaction Protocol (BTP). BTP Committee specification. April 2002. Available at http://www.oasis-open.org/committees/business-transactions/
9. F. Curbera, R. Khalaf, N. Mukhi, S. Tai, S. Weerawarana. Service-oriented computing: The next step in Web services. Communications of the ACM, Volume 46 Issue 10. October 2003.
10. S. Graham et al. WS-Notification specification. March 2004. Available at http://www-106.ibm.com/developerworks/library/specification/ws-notification/