Examining BPEL's Compensation Construct

Joey W Coleman

School of Computing Science
University of Newcastle upon Tyne
NE1 7RU, UK
email: j.w.coleman@ncl.ac.uk

Abstract. This paper gives a short description of some features of long-running transactions, as well as the language BPEL and its particular implementation of the compensation concept. Two examples are used to illustrate the application of BPEL's compensation construct. These examples, and reference to a structural operational semantics developed elsewhere, are used to help support an argument for the need of a more general implementation of compensation.

1 Introduction

Despite decades of work on ways of modelling long-running activities using transaction schemes, the same basic problems exist now as 25 years ago. Many systems have been designed to address parts of the problem but they tend to be refinements of the usual recovery mechanisms.

Designers of business process languages, in an attempt to model workflow, have taken to including a concept called compensation in their work. Compensation, in general, should be capable of addressing both non-reversible errors and non-erroneous changes in the execution of an activity. Unfortunately, the implemented design of compensations in languages such as BPEL can only conveniently be used to handle a subset of errors.

Possible semantic descriptions of BPEL's compensation mechanism are given in [Col04,BFN04] and others. The pragmatics of BPEL's compensation mechanism, on the other hand, need clarification. This paper aims to demonstrate how the mechanism's purpose as described in the BPEL specification [ACD+03] is left unmet by the constraints given in the same document.

The next section of this paper gives a quick description of properties associated with the Long-Running Transactions (abbreviated as LRT). Section 3 describes BPEL and its compensation construct. Following that, section 4 gives an example that shows how BPEL's compensation model fits with a simple LRT. Section 5 extends that example into a scenario that does not easily fit BPEL's compensation model. The final section concludes this paper with a summary of the points raised by the examples and the motivation for a more general implementation of compensation.

2 Long-Running Transactions

The notion of long-running transactions [ACD+03] arises out of work done in the database community on structuring transactions intended to run over long periods of time — from seconds up through minutes, hours, days and longer. In contrast, the well-studied notion of ACID transactions give desirable properties for transactions that run in short periods of time — nanoseconds, milliseconds, and up to a few seconds in length. The properties of consistency and durability are common to both LRTs and ACID transactions, but atomicity and isolation are very much weakened for LRTs [Gra81]. Synonyms for long-running transaction are long-lived transaction [Gra81,GMS87] and long-running activity [DHL91].

The properties of consistency and durability apply to LRTs in the same manner as they do with short-lived transactions. A LRT must leave the system in a consistent state, and more importantly, any changes made during the execution of the LRT that are visible outside of the LRT must also maintain system consistency. The durability requirement is obvious: having the changes made by a LRT that disappear except through the actions of another LRT is generally not a desirable thing.

Isolation can only be applied to those bits of state that are truly local to the LRT. Any state that does not survive past the end of the LRT should be isolated from everything outside of the LRT. Changes to the overall system state, however, cannot be isolated as the usual techniques used to isolate changes to the system state are unsuitable over long periods of time [GMS87].

Atomicity in the context of a LRT is very much relaxed when contrasted against its meaning for regular database transactions. For a LRT atomicity simply means that any changes during its progress maintain system consistency. The use of the compensation notion implicitly acknowledges the fact that there are cases where it is not possible to put the system state back to what it was before the start of the transaction.

As with short-lived transactions, LRTs have well-defined boundaries for their beginning and completion. They can, and usually should, contain short-lived transactions and even other LRTs.

3 BPEL's Compensation Construct

BPEL[1] [ACD+03] is a relatively recent language that is still under development. Its origins lie in the web services community, and the initiators of its development include BEA Systems, IBM, Microsoft and a number of others. Current activity on the language is now coordinated by the OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee[2].

One of the claims made in the BPEL specification is that it provides the necessary tools and structure to support LRTs that are local to a BPEL process.

[1] Business Process Execution Language for Web Services
[2] Web address: http://www.oasis-open.org/

Central to that claim is BPEL's provision of a compensation construct, which was modelled after ideas in previous work on Sagas [GMS87] and others.

The specific implementation of compensation that BPEL uses is essentially an extension of the usual exception-handling mechanisms seen in languages such as `C++`, Java, and so on. Blocks of code — called scopes in BPEL — may have a compensation handler associated with them. These scopes, and their associated compensation handlers, can be nested to an arbitrary depth. Upon successful completion of the scope the compensation handler and the current state of the process are saved for possible later invocation.

Invocation of a compensation handler can only be done from within one of BPEL's fault (exception) handlers, and when actually invoked, the compensation handler is run on its associated saved state. It is not possible for the compensation handler to access the current state of the process directly, though it is not difficult to imagine a situation where current state is accessed by means of another process.

Three things characterize BPEL's compensation:

- the mechanism is intended to be a form of backward recovery [ACD+03];
- despite saving the state of the process at scope completion, the mechanism only provides a convenient means to manipulate the process' control flow within the bounds of an exception handler;
- compensation handlers are named to facilitate control flow modification.

It is, in fact, possible to give an operational semantics that shows BPEL's compensation mechanism to be a primitive named procedure call [Col04]. Though the compensation "procedures" do not directly allow parameters, it could be argued that the saved state could be used as a parameter passing mechanism. The use of names to identify specific compensation handlers allows for an arbitrary, programmer-defined ordering, including parallel execution.

Categorizing BPEL's compensation feature as intended for backwards recovery comes directly from the BPEL specification [ACD+03] which mentions the use of compensation to 'reverse' and 'undo' previous activities. The specification also goes so far as to restrict the invocation of a compensation handler to within a fault handler. Compensation in BPEL has been relegated to the realm of abnormal behaviour.

The assessment of BPEL's compensation mechanism as a convenient means to alter control flow relies on the fact that the language specification explicitly restricts compensation to only have meaning in a local sense. Saving the process state at scope completion is only intended to save the contents of the process' variables, *not* the underlying state of the BPEL processing engine [ACD+03]. Saving those variables could be done manually and would give the compensation handler the added ability of being able to access the current state of the process. This leaves the single bit of control flow modification that BPEL's implementation of compensation usefully achieves: a simple mechanism to partition the actions of a traditional `try/catch`-style exception handler so that the handler need not try to figure out which parts of the body have executed.

4 The Bookshop

One of the common examples used to illustrate LRTs is that of a buyer-seller-shipper situation. Here we will consider a bookshop example similar to that used in [BFN04].

The example starts with the seller accepting an order for books in stock; an order for books not in stock is rejected immediately. Accepting the order reduces the seller's available inventory. The seller then attempts to fulfill the order by doing the following in parallel: a) arranging for the books to be shipped, b) packing the order, and c) checking the buyer's credit.

In this example we are not including the pickup of the books from the seller by the shipper, the receipt of the books by the buyer, the actual payment of the seller by the buyer, nor the payment of the shipper by the seller.

Included in the example are compensation actions for accepting the order, packing the order, and booking the shipper. Since checking the buyer's credit was just a 'read', there is no compensation required for that action. Of course, should the buyer's credit rating be insufficient for the order, then the order will be canceled.

It is straightforward to cancel this LRT at any point during its execution. If the seller in the LRT had only just completed accepting the order, canceling it involves merely throwing the order away, making the books available for sale, and notifying the buyer that the order was canceled. If the order had been accepted and the parallel tasks were in progress, then canceling involves unbooking the shipper and unpacking any packed books, then throwing away the order and making the books available for sale.

This example translates into a BPEL process in a straightforward manner. Accepting an order would exist as a BPEL `scope` object (with its compensation handler). Unbooking the shipper would also be in its own `scope` object. However, to correctly model the required compensation for the parallel tasks, each action that packs a book would need to be in its own `scope`, thus allowing only the compensation handlers for the packed books to run.

5 The Bookshop, Extended

The previous example is fairly straightforward, and perhaps even matches the most common behaviour that a bookseller might follow. There is, however, a more complex behaviour that shows the limitations of BPEL's compensation model.

For this example, imagine that the seller carries no stock, such that all books must be pre-ordered. In this case, the seller accepts any order for any collection of books that it believes it can get. For the sake of simplicity, the seller also knows the correct final price for any book that it believes to be available.

The seller would then generate, in parallel, an expected delivery time for the buyer, and charge the buyer for the books. This charge may simply be a deposit or the full cost of the book, but it would be a charge rather than a credit

check. After the buyer has been charged, the seller then places an order with their supplier for the desired books. When the seller receives the books it would perform the parallel tasks of arranging for the books to be shipped and packing all of the books to be shipped to the client.

If the seller's suppliers were completely reliable, then the mechanism for canceling this order is a straightforward extension of the previous example. Since it is unlikely that the suppliers would be completely reliable, we will assume that suppliers will occasionally be unable to supply certain books, and that the suppliers will notify the seller of this when the seller places their order from the supplier.

Assume, instead, at some point between when the seller told the buyer when to expect the books and when the seller should have received the books, that the seller receives some notification that one of the books in the order is no longer available. This requires that things be corrected so that the unavailable book is no longer a part of the order.

If there was only one book in the order then handling this situation is easy: just cancel the whole LRT. For cases where there were several books in the order, especially when the seller has already received some of the books, then the seller will still need to ship the rest of the books and record things appropriately.

Simply canceling then restarting the whole LRT without the unavailable book is inappropriate: that would likely cost the seller extra in transaction fees. It is also an unnecessary repetition of effort. What should happen is that the unavailable book is removed from the order, the buyer is refunded the appropriate amount, and then the LRT proceeds as though the unavailable book had not been ordered in the first place.

Having the supplier's notification about the unavailability of a book forces the seller to perform compensatory actions so as to avoid having to cancel the entire LRT. It would seem appropriate to use a compensation mechanism to allow the LRT to proceed, but the actual implementation in BPEL would be absurdly complex.

Despite BPEL's restriction that compensation may only be called from within fault handlers, it is possible to model this behaviour using compensation. The design would have each book ordered in its own thread — as with the process given above — but after the individual book order is complete, a busy-wait loop would prevent the thread and associated compensation scopes from exiting. If one of the book orders needs to be compensated, then a flag would be set and the busy-wait loop would throw a fault which in turn would cause a handler to invoke the compensators.

The problem with this solution — aside from its inelegance — is that it seems to run against the pragmatics of a compensation construct to use nested scopes just to isolate fault handlers whose only purpose is to invoke a compensator for the innermost scope.

So why don't we just use the fault handler to directly correct the problem? Leaving aside the convenience of a mechanism that automatically only corrects the actions that have completed, this solution is not much better. The busy-wait

loops are still required, leaving a collection of live threads that are doing nothing. Compare this against a compensation mechanism where, when the compensator is installed, the individual threads are finished and cleaned up normally.

6 Conclusions

From the standpoint of the structural operational semantics mentioned above and developed in [Col04], BPEL's compensation mechanism is only a primitive procedure call. Indeed, the compensation mechanism doesn't even require the full structure of a procedure call, but just that of a block-structure language. The only restriction in the semantic model that prevents the compensation mechanism from from being used outside of a fault handler is a well-formedness condition on the abstract syntax of the language. In light of this it is unfortunate that the BPEL specification states this condition as it needlessly precludes a more general use of compensation.

The initial example does show that the compensation model used in BPEL is applicable to some situations. The use of named compensators has advantages, allowing programmer-defined compensation ordering. Also, in general, the usual implementations of compensation are extremely useful to simplify the programmer's task of only reversing those actions that have completed successfully.

The use of compensation as implemented in BPEL to handle changes to the LRT during its execution is inconvenient at best. An argument can be raised that any changes to the LRT should be kept well defined and incorporated into the main flow of the process. This argument has the same problems as arguing that fault handling should be done inline with the main code rather than separated out into fault handlers.

Some models of compensation have posited a fairly strong property: if an action and its compensation are independent of all of the actions between the original action and the compensation, then the action and its compensation is equivalent to a null action [BHF04]. While this is certainly true, it has been pointed out that the likelihood of independence is rather low [GFJK03]. First, there is the difficulty of designing your process so that the interleaving of local actions maintains this independence. Second, due to the long-running nature of LRTs, and the requisite lack of lock-based synchronization, it is extremely likely that another executing LRT will do something that is not completely independent. One might observe that part of the point of grouping actions together in a transaction is because those actions are not independent.

The extended example gives an argument that the particular compensation model used in BPEL is not applicable to the full range of situations where compensation would seem to be an ideal tool to structure a long-running transaction's fault tolerance. If BPEL's compensation mechanism could be invoked outside of a fault handler then the extended bookshop example would no longer be an issue.

Acknowledgments
The author would like to acknowledge the support of the RODIN (Rigorous

Open Development Environment for Complex Systems) Project funded by the European IST and the DIRC (Dependability Interdisciplinary Research Collaboration) project funded by EPSRC/UK.

References

[ACD+03] Tony Andrews, Franciso Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services, version 1.1. http://www.ibm.com/developerworks/webservices/library/ws-bpel/, May 2003.

[BFN04] Michael Butler, Carla Ferreira, and Muan Yong Ng. Precise modelling of compensating business transactions and its application to BPEL. Technical report, University of Southampton, Electronics and Computer Science, 2004.

[BHF04] Michael Butler, Tony Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In A. Abdallah and J. Sanders, editors, *Proceedings of 25 Years of CSP (in press)*, London, 2004.

[Col04] Joseph W Coleman. Features of BPEL modelled via structural operational semantics. MPhil thesis, University of Newcastle Upon Tyne, November 2004.

[DHL91] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. A transactional model for long-running activities. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 113–122. Morgan Kaufmann Publishers Inc., 1991.

[GFJK03] Paul Greenfield, Alan Fekete, Julian Jang, and Dean Kuo. Compensation is not enough. In *EDOC '03: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*, page 232. IEEE Computer Society, 2003.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM Press, 1987.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *VLDB*, pages 144–154. IEEE Computer Society, September 1981.