Compensation is Not Enough

Paul Greenfield[1], Alan Fekete[2], Julian Jang[1,2], Dean Kuo[1]

[1]CSIRO Mathematical and Information Sciences, Lock Bag 17, North Ryde NSW 1670 Australia
[2]School of Information Technologies, University of Sydney, Sydney NSW 2006 Australia
Emails: {paul.greenfield, julian.jang, dean.kuo}@csiro.au fekete@it.usyd.edu.au

Abstract – An important problem in designing infrastructure to support business-to-business integration (B2Bi) is how to cancel a long-running interaction (either because the user has changed their mind, or in response to an unrecoverable failure). We review the fault-handling and compensation mechanism that is now used in most workflow products and business process modelling standards. We then use an e-procurement case-study to extract a set of requirements for an effective cancellation mechanism, and we show that the standard approach using fault-handling, and compensation transactions is not adequate to meet these requirements.

***Index Terms*—middleware, workflow, business process modelling, atomicity, compensation, transactions.**

I. INTRODUCTION

Enterprise application integration (EAI) and business-to-business integration (B2Bi) automate business processes by composing autonomous components (services) into loosely coupled distributed applications. These composed applications can be represented as workflows in business process modelling languages such as BPEL4WS [2] and WSCI [3], and implemented using workflow technologies such as BizTalk Server [1] and WebSphere MQ Workflow [4].

Defining a business process and modelling it in a workflow language would be a straightforward exercise if every activity always completed successfully and could never be cancelled. Reality is somewhat different though, as shown in [5] where the authors report that nearly 80% of the time spent on implementing a business process is spent on handling exceptions (failures). There are many reasons why handling failures is difficult including:

- Failures can happen at anytime, even while handling another failure.
- Any of the activities making up a business process can fail in many ways, and there are failures which are not local to one party, but rather in the way peer processes interact.
- Interactions between business partners are often best handled asynchronously, making it difficult to keep the overall state of the combined systems consistent.
- Business processes are often affected by other concurrent business processes as they access and update shared resources.

Failures which take a business process away from its normal path and prevent from completing normally need to be handled to ensure that overall system consistency is maintained and data integrity is not compromised. A simple example from e-procurement is that a failure, if not appropriately handled, could easily lead to a situation where ordered goods are delivered to a customer but the merchant never requests or receives payment.

In this paper, we first discuss the common failure and cancellation mechanisms found in current business process modelling languages. These languages all support a model that is an amalgamation of:

- Traditional short-duration ACID transactions [10].
- Concepts from advanced transaction models [9], in particular Saga's use of compensation transactions [8].
- Exception handling mechanisms derived from programming languages such as PL/I, CLU, Ada, C++ and Java.

The main contribution of this paper is a critique of this 'standard model for handling failures and cancellation. This critique is based on observations derived from examining an e-procurement scenario. Using this e-procurement example, we first describe the types of behaviour required when a failure or other cancellation occurs in a business process. This enables us to then evaluate the appropriateness and adequacy of failure and cancellation handling capabilities found in current business process modelling languages. We observe that there are many classes of required behaviour that cannot be succinctly specified using this standard model – that is, the combination of ACID transactions, compensation transactions and exception handling is inadequate when specifying how to handle a failure or cancellation event.

The paper is organised as follows. In Section II, we describe related work – reviewing ACID transactions [10], the Saga advanced transaction model [8], exception handling in programming languages and the business process modelling language BPEL4WS [2] from BEA, IBM and Microsoft. In Section III, we describe our e-procurement scenario and in Section IV we describe the types of behaviour required when a failure or cancellation event occurs. Section V is a critique of failure and cancellation handling capabilities in business process modelling languages. Section VI presents a discussion of what is missing in business process modelling languages for

failure and cancellation handling as well as a conclusion and discussion of our future work.

II. Related Work

The database community had considerable early success in dealing with failure and unusual events through the concept of ACID transactions. This work was expressed in a very elegant mathematical theory of serializability. The influence of the theory and the practical mechanisms was recognized in a Turing Award to Jim Gray in 1998. The key insight was that the appropriate mechanisms in the database infrastructure could completely hide both failures and interleaving from the application programmer. The developer could now simply write code that performed meaningful business activities, and enclose this code within a transaction construct. The system would then ensure that the code would always run as if it were a single, indivisible and instantaneous unit of work, totally unaffected by any other activities that may be taking place concurrently with its execution. If part of the code had executed and a failure prevented the rest from running, the transaction infrastructure would rollback the incomplete activity, that is, it would restore the database to its state at the time the activity started. Furthermore, when the activity completed successfully, the infrastructure would ensure that its changes persisted in the database despite any possible subsequent crash. The term "commit" is used to describe the successful completion of the transaction, and a failure leading to rollback is called "abort".

The acronym ACID refers to the essential properties provided for a piece of code defined as a transaction. It is atomic, so it either commits having executed completely, or else no-one can detect that anything happened at all. It is isolated, so even when several transactions run concurrently, each appears to run as if no other code were executing at the same time. It is durable, with changes made by committed transactions persistent despite crashes or other failures. The other letter stands for consistent. This is separate from the other aspects, since it is the application programmer's responsibility to make the code keep the database state consistent if the code is considered by itself with no failures. The other aspects (A, I and D) will then guarantee that the state is consistent even for executions where failures and concurrency are present.

The mechanisms used to support the ACID transaction concept are locking and logging [10]. The infrastructure places locks on certain items whenever they are accessed by application programs. These locks are held until the transaction completes, preventing concurrent transactions from seeing inconsistent data coming from an incomplete transaction. Each modification to the database is recorded in a log table, which is stored on disk separately from the data itself. Rollback can be performed by re-installing the previous value for every item found in the log; similarly after a crash the up-to-date state can be restored from the log. We note that locking is essential for logging to work correctly, since if several activities have all modified one item, then it would not be clear which state should be restored when one activity needs undoing.

Almost all commercial database systems use these standard transaction mechanisms and they have been refined and tuned to be extremely efficient for the typical business application of the 1980s and 1990s, such as recording an order at a warehouse, or a payment in a bank. This class of application (called OLTP) was the bread-and-butter of enterprise computing in the past. However, it has been clear for at least a decade that these mechanisms cannot be used to meet other enterprise requirements such as enterprise application integration (EAI) or business-to-business integration (B2Bi). One critical characteristic of these newer enterprise computing situations are that the activities being automated have long durations (seconds to months) since they require the collaboration of many participants, and often involve human interaction for some steps. It is infeasible to hold locks on data items for such long periods, as this could delay other, concurrent, activities for days or longer!

Another important characteristic of traditional OLTP is that the transaction acts entirely within one administrative unit's control (originally, it was in a single database, later extended to activity distributed among several databases but managed by a single transaction processing monitor). In contrast, modern enterprise computing can cross boundaries, for example one long-running process may involve the warehouse, the financial controller, and the logistics section. In B2Bi, the process must even run between different companies, which may be competitors as well as collaborators. Again, this makes locking unworkable, as no organisation would give a competitor a lock on its data (this would invite a simple denial of service attack).

It is generally acknowledged that long-running business processes cannot simply be treated as an extended ACID transaction, but at least some of the ease of programming and support for consistency provided by ACID transactions is still needed. This requirement has led to extensive research [9] on extended transaction models that relax full ACID guarantees but still provide some infrastructure support for handling failure and concurrency. The most influential of these models for business process models and workflow management systems has been the Saga approach [8].

Garcia-Molina and Salem in [8], proposed to structure a long running process (a "Saga") as a sequence of smaller tasks, each of which would be done as an ACID transaction. The underlying mechanism would then ensure that each task ran without interference, but the tasks of one process could interleave with the tasks of another process.

The key insight of this work is in the way to respond to failure during a Saga. If a particular task fails, it can be aborted and rolled back, and then retried. However, if the Saga as a whole gets into an irretrievable difficulty, and needs to abort, what should happen? The answer proposed in [8] is that the application developer should design, for each task, a corresponding compensator. The compensator executes an operation which does the inverse of the original task. For example, the compensator for inserting a record might delete the record; the compensator for depositing to a bank account might withdraw from the same account, and a read-only task

has empty compensator. The compensator should correct any flow-on effects from the original activity. For example, if a sale has been used to determine the amount of a bonus payment, then the compensator ("cancel the sale") might need to recalculate the bonus.

To abort a Saga, the system will abort any active task of the Saga, and then invoke the compensators for each task within the Saga that had previously committed. The compensators are run in the reverse order from the order in which the tasks ran originally. Subsequent work extended the proposal of [8] to allow nested Sagas, where the tasks within a Saga might each be Sagas themselves, rather than ACID transactions. However the essential idea remains the same.

While the database community used the ACID transaction and its generalizations as the basis for dealing with failure, a different evolution took place in the programming language community. Here there was no question of shielding the programmer from having to work out how to respond when a failure happened, whether that failure was due to a user-initiated change of mind, or to an unexpected state of the data, or to a system error such as trying to read a non-existent file. Instead, the focus was on supporting the programmer by separating the code for abnormal cases from the simpler normal case flow. The idea of exception handlers arose in the late 1970s in PL/I and CLU. In these languages, the developer can name particular situations (called exceptions), and write code to deal with each named exception. The runtime environment will terminate normal activity when the exception is "raised" and transfer flow to the exception handler code.

This model of exception handling was picked up by later languages, such as Ada, C++ and Java. There have been some refinements appropriate for object-oriented languages, such as treating Exceptions as first-class objects within an inheritance hierarchy, and allowing an Exception to propagate to larger and larger blocks until a handler is found.

Early prototype workflow systems dealt with the issue of failure in several ways. IBM's FlowMark system [7] used the workflow database to deal with communication losses, errors in the workflow definition, or with system crashes. In contrast, "semantic failures" were not the system's concern. The authors showed how one could encode a variety of advanced transaction models, including Sagas, within the normal workflow definition. In essence, the application designer had to write the compensators, and also write control features in the workflow definition in order to invoke each compensator in the appropriate situations. Meteor2 [6] provided a more seamless system, where a general error-handling mechanism allowed handlers to be written for three levels of error: semantic errors, task manager errors, and errors in the WFMS engine. An unhandled error of one level eventually was reported at the next level.

The way failures are dealt with in workflow management tools and in business process models owe much to both exception handling from programming languages, and Saga's compensation model. For concreteness, in this paper we will use the detailed description in section III of the BPEL4WS standard [2] as the exemplar of this approach; the same concepts are found in a variety of alternatives such as WSCI or Microsoft BizTalk Server.

In BPEL4WS, a business process is described as a structure using XML syntax. This structure contains a nested arrangement of activities, some of which are basic activities, such as receipt of a message, and others are structured activities, such as the interleaving of parallel threads, each running a sequence of basic activities. Fault and compensation handlers can be defined for activities. Such activities are called scopes. Scopes can contain nested scopes.

When error occurs within a scope, a particular kind of fault is generated. There are faults which occur on receipt of a message with incorrectly formatted data, and others which are defined by the application programmer and thrown by the activity's' own code, for example to indicate that the warehouse has none of the required item on hand. A fault can also be thrown for a scope from code in another thread, or even from another instance of the BPEL4WS process. A fault can have a container of data associated with it. Whenever a fault occurs, the innermost enclosing scope has all its ongoing activities automatically terminated. The appropriate fault handler code is then executed. No matter what the fault handler does, the scope is regarded as terminated abnormally.

If the programmer has not provided a fault handler for a particular fault, then a default fault handler is used. The default handler acts as prescribed by the Sagas model [8]: for every enclosed scope that has terminated successfully, run its compensation handler (doing so in reverse chronological order of completion of these scopes), and then re-throw the fault to the enclosing scope. Similarly, if a scope doesn't have an explicit compensation handler, the default is to run the compensators for every enclosed scope that completed, in reverse chronological order.

The compensation handler for a scope has access to a snapshot of the data of that scope, as it was at the instant that the scope completed. Note that this refers to data held in the workflow engine ("Containers" in BPEL4WS terminology) and not to the values stored in databases by the activities that ran. The workflow engine provides concurrency control on access to Containers, if a scope is declared to have the attribute containerAccessSerializable set to "yes".

Note that compensation handlers are only run for completed successfully scopes, and only from a fault-handler for some enclosing scope. Programmers cannot invoke the compensation handler during normal activity.

A very different aspect of cancellation is treated in [11], where the emphasis is on the feedback given to the user concerning their request for cancellation, and on the software architecture needed to provide sensible feedback.

III. An E-procurement Scenario

In this section, we describe briefly an e-procurement scenario that was derived from a consultancy project. The business process, itself, is very simple and is shown in Figure 1. In section IV, we use this example scenario to illustrate the different classes of cancellation behaviour.

The business process is initiated when it receives a quote request from a customer. The merchant first checks that the customer is a valid customer – that is, registered and with no overdue payments. For invalid customers, a message informing the customer that he/she is no longer a valid customer is sent before the business process terminates. For valid customers, a quote is calculated before it is sent to the customer. The merchant will receive a purchase order if the customer proceeds with the order. Goods are then reserved followed by the payment and delivery process which can run concurrently. The payment process consists of sending an invoice, receiving payment and sending a receipt. The delivery process consists of arranging transportation, shipment of goods, sending notification to the customer that ordered goods are now in transit and receiving an acknowledgement that goods have been received by the customer. After the delivery and payment processes have completed, the workflow terminates.

At any point in time, the customer can send a request to the merchant to cancel an active order. The correct behaviour depends on the state of the business process as well as the state of the merchant – e.g. the amount of stock available in its warehouse. The execution of an activity may also fail. This does not necessarily require that the order be cancelled which is unlike ACID transactions [10] and Advanced/Extended transactions models [9] including Sagas [8]. For example, if one transportation company cannot deliver the order then the merchant can find an alternative. If there is a network or server problem, then one can wait and then retry. For long running business processes, only when failures become unrecoverable should the order be cancelled – that is, a long running transaction is only aborted as a last resort.

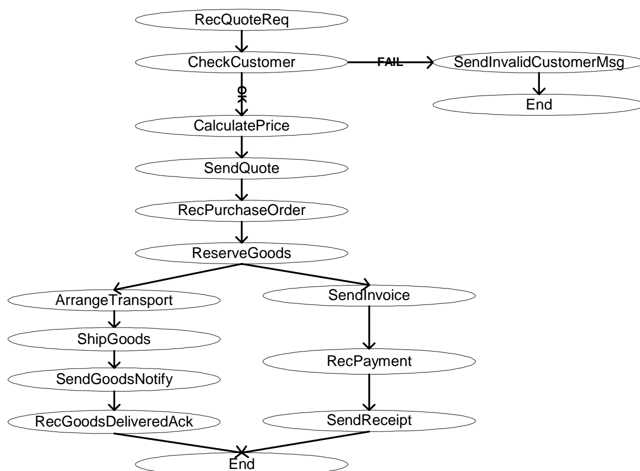

Figure 1 - Merchant's Workflow

IV. Classification

This section describes the different types of behaviour required to handle an abnormal situation which does not fall within the normal processing path for a long running business process. Examples from the e-procurement scenario are used to illustrate each class of behaviour.

Section A describes the types of behaviours required when an event causes a business process to deviate from normal processing so seriously that it is desirable to cancel the original processing entirely. We focus particularly on situations where a cancellation request is received, but there is a lot of similarity with the types of processing needed following an error such as loss of communication or the crash of some application code.

Section B describes the types of behaviour required when an event causes a business process to deviate from normal process but in a less serious way that does not require cancellation. In view of the effort already invested in a long running process, and the number of collaborators involved, a business process should only be cancelled as a last resort.

A. *Cancellation*

1) *Terminate all processes and simple activities*

The simplest type of behaviour is to stop; that is the abnormal situation calls for the system to terminate whatever activities are underway. Within this case, we allow for the cancellation to also execute some additional simple activities after terminating whatever is active.

An example of this type of behaviour is if the customer decides to cancel the order before the merchant has performed any significant activities that impact on the real world. Suppose the cancellation request arrives before ReserveGoods has started. The merchant may be currently calculating a price, or it may have just received the PurchaseOrder, but it has not reserved goods from the inventory system or arranged for delivery. The business process can be terminated and all that might be needed is, optionally, executing a simple activity which updates the status of the order from *active* to *cancelled*.

For this class of behaviour, notice cancellation of a business process does not necessarily return the system back to its exact original state. For example, the customer database may have been updated. Furthermore, activities are not undone via compensation transactions – for example, sending and receiving quotes, and calculating the quote are not compensated. All that is required is basically to terminate the business process.

In BPEL4WS [2], a business process can be terminated without executing any further activities via its `terminate` tag. It is also possible to write a fault handler that executes simple activities and then terminates the scope.

2) *Undo in Reverse Chronological Order*

A slightly more complex class of behaviour requires cancelling a scope by undoing activities that have successfully executed, in the reverse of the chronological order in which they

completed. This is the behaviour provided by the Saga transaction model [8].

In the e-procurement scenario, if an order needs to be cancelled after goods have been reserved and transportation arranged but before an invoice has been sent and the goods shipped, then the order can be cancelled by running compensators in reverse chronological order for those activities that have successfully executed. As mentioned previously, activities such as *calculate quote* and *send/receive quote* need not be undone. To fit the Saga transaction model, these activities have the `null compensator`.

Numerous business process modelling languages, such as BPEL4WS, and commercial workflow products, such as BizTalk Server [1], already provide direct support for this pattern of cancellation, as the default way of handling any exceptional situation.

3) Compensation in Application Defined Order

In the Saga transaction model, if a Saga aborts, the activities are undone in reverse chronological order.

In the e-procurement scenario, certain circumstances place different constraints on the order of compensators. For example, if a cancellation is received after payment has been received and transport arranged, there may be some cancellation charges from the shipper which must be passed on to the customer. Thus the compensator for receiving payment (which refunds money to the customer) should only execute after the compensator for arranging shipment has executed. Since the original processing of ArrangeTransport was concurrent with RecPayment, the completion could have been in either order. That is, in this case compensators do not necessarily occur in reverse of the original chronological order.

There are also cases where one activity has to execute before another but their compensators can execute in any order. In the general case, the compensators may be required to execute in an order that is application specific.

In BPEL4WS, an application programmer can specify an arbitrary process in its compensation handler. Thus, it can support the type of behaviour described here.

4) Executing Independent Activities

There are circumstances where the required behaviour is not to follow the traditional view of rolling back. To handle a cancellation, a business process may need to execute a process whose processes are independent of the activities that have executed in the original forward processing.

For example, in the e-procurement scenario, if ordered goods have already been shipped then a cancellation process will require the invocation of a return goods process. It would arrange the delivery for the unwanted goods back from the customer (either to the original warehouse or perhaps to an alternate storage site). It would also involve special checks to make sure that the goods returned were the ones originally delivered, that the goods have not been damaged, and so on. Notice that the return goods process may itself fail and this needs to be appropriately handled.

In BPEL4WS, as an application programmer can specify an arbitrary process in its compensation and fault handlers, it is possible to define a cancellation mechanism that does not resemble the original process.

5) Activities dependent on state

The correct behaviour to cancel a long running process may depend on the state of other activities and data. The status may not be known at the time when a fault needs to be handled.

For example, if an order is cancelled then the cancellation fee is dependent on the state of the delivery. If there is a fee for the cancellation for delivery, then the costs are passed onto the customer. If an invoice has not been sent, then an invoice for the cancellation fee is sent to the customer; if an invoice has been sent and payment has been received, then a partial refund is sent to the customer.

The final case is a more awkward to handle as it introduces an extra dimension to the problem – the exact state is unknown. In this case the merchant has sent the customer an invoice but has not received payment, and the merchant does not know if the customer's payment is in transit or not. The merchant has to send the customer an invoice for the cancellation fee but a payment for the original invoice may arrive after the merchant has sent the invoice. In this case, the merchant has to assume that if the customer receives an invoice for the cancellation fee and has already sent payment that the customer would ignore the invoice for cancellation fee. The customer would then wait for a partial refund from the merchant.

Care needs to be taken in defining the protocol between customer and merchant; otherwise, inconsistencies could occur where the customer never receives the correct refund or the merchant never receives due payment.

The final case can be more simply handled if the merchant does not allow orders to be cancelled, and it only sends an invoice after the goods have been shipped. That is, we can place extra constraints on the business process to avoid being in unknown states when a fault occurs.

6) Human Intervention

It is not realistic to expect that all cancellation can be handled without human intervention since there may be very complex/special circumstances for the cancellation. Flexibility for handling cancellation is greatly increased if humans handle the most complex and rare situations.

It is straightforward to initiate and receive notification of the outcome of human intervention activities via simple mechanisms such as sending and receiving emails. The difficulty is the business process (workflow), such as BizTalk Server, is having access to sufficient state information to be able to send them to the human operator.

B. *Non-Cancellation*

1) *Return to Normal Processing*

In the e-procurement example, a customer can request that an order be cancelled but in some situations, the merchant can choose whether to accept or reject the request. For example, many bookshops do not allow cancellation of "special orders", where the merchant had to bring in titles which it does not normally carry. If the cancellation request is rejected, the process may notify the customer of the rejection, and then both parties continue as if cancellation request never happened. In the first case – cancellation accepted – the required behaviour falls into one of the categories described in Section A.

In the standard model, fault handling is begun only when all normal activity has been terminated. Thus we need to make sure that a fault is generated only after the merchant is sure that the cancellation request is acceptable. That way, if the cancellation is rejected, the normal activity has been killed and will complete.

2) *Pause Processing*

From the above example, the most appropriate scheme to handle a cancellation request would require a separate thread to receive the request and determine if the request should be accepted or rejected. Only if the cancellation request is accepted should a fault be thrown. This, however, introduces another interesting problem. The thread handling the cancellation request may take some time to decide whether to accept or reject the request – for example, seek approval from a (human) manager – but the main process may continue making forward progress and change the state significantly – e.g. to a state where cancellation is no longer permissible. What is really required is the ability to pause a process when determining if a cancellation request is accepted or rejected.

3) *Alternatives*

Activities in a business process may fail – e.g. a transportation company may not be able to deliver the goods at the required time. When a failure occurs, it is often inappropriate to take the drastic action of aborting the business process. It is possible in certain situations to execute alternative activities, and if they are successful, the business process can continue as normal – e.g. the merchant finds another shipper.

An activity is optional if a business process can continue normally even if an activity fails. Optional is a specialisation of alternative since it can be implemented as an activity with a null activity as its alternative.

4) *Retry*

An activity may fail due to a network or remote server failure. In this case, the most practical handling of the failure is to periodically retry the activity until it succeeds. If the number of reties exceeds a specify number, then one could possibly try an alternative such as sending the equivalent message by fax. Only as a last resort and if the activity is critical for the business process should the process be cancelled.

5) *Rollback to a earlier point in the process and redo*

An activity may fail and the most appropriate cause of action is to undo, via compensation, some activities; thus returning to an earlier point in the business process (savepoint) and then restarting from the savepoint. Furthermore, some activities may not need to be redone when the process resumes – e.g. those activities that were not undone.

A concrete example where this type of behaviour is required is if the delivery of goods were sent to the wrong address. The shipment is returned; the merchant determines the correct address and then resends the shipment. The activities in the payment process need not be undone or redone.

6) *Continue processing and create additional process*

Partial fulfilment may be required when a merchant can not provide all the goods at the time of delivery – e.g. if a backorder is delayed. The merchant thus ships the available goods and later it arranges transport of the other goods when they become available.

This class requires that the process spawn another process to handle currently unavailable goods; meanwhile it must continue normal processing for the goods that are already on hand. There is also a need to modify other processes – e.g. the invoice to the customer is not for the full amount but only for goods that have been shipped. A later invoice is sent when the unavailable goods are shipped.

V. CRITIQUE OF STANDARD MECHANISM

In this section, we explore in more depth the issues which were raised in the introduction, concerning the intrinsic shortcomings in the standard approach to dealing with failures and cancellation requests in systems composed from Web Services. This standard approach uses an exception-handling mechanism similar to that in programming languages, together with application defined compensators which semantically undo completed activities. While we concentrate on BPEL4WS as a way to be precise about the standard approach, similar issues arise in other standards such as WSCI.

A fundamental assumption made by the standard model is that that every completed activity can be semantically undone. That is, the model assumes that application designers can always write a correct compensator for each activity. However we saw above that it may not be possible in all cases to undo the effects of shipping some goods. The only way the standard model can deal with activities that cannot be undone is to define an *empty* compensator, but this is unsatisfactory because it is treated as successful compensation and thus enclosing scopes are not aware that the activity has not been undone correctly. Interesting work in the database community has shown the importance of identifying activities that cannot be rolled back (for example, because they take place in a resource manager that does not support the prepared state of two-phase commit). In [12], the term pivot activity is used for such situations, and it is shown that correctness depends on having at most one pivot sub-activity in each larger process.

Even if some aspects of an activity can be undone, it is not always the case that we can return exactly to the original state. The compensator for an activity, such as reserving goods, is to remove the reservation. One might believe that such a

compensator is guaranteed to successfully execute, but the reservation may have triggered off a back order. If the backorder plus the original reservation together would leave the merchant with an excessive quantity of goods, then this simple compensation might be unacceptable.

Furthermore, the standard model does not seem to take account of possible state-dependence in how compensation should occur. At least in the BPEL4WS expression of the model, the compensator has access to the stored state in databases etc, and to the state captured in containers by the original activity, but it does not have access to the current state of running concurrent activities. However we have seen that the correct way to rollback a completed shipment of goods can depend on the status of the concurrent payment process.

Another flaw in the standard approach to cancellation arises from the assumption that fault-handling should involve the immediate termination of all running activities within the scope that has suffered the fault. This assumption makes sense in the traditional object-oriented programming languages where the exception-handling concept arose, but it is not valid for all cases of faults within a long-running business process. In contrast, we believe that proper handling may sometimes require the application-specific fault-handler to intervene intelligently in the running activities. It should examine the current state of the scope and then act on different activities in different ways: some may be allowed to reach a stable point (eg a checkpoint sub-activity should continue until the database is a consistent snapshot), some may wish to take special preparations before termination (for example, freeing computational resources such as memory or file handlers), some may need to be killed, and others maybe should proceed to normal completion, unaffected by the fault.

As defined in the BPEL4WS standard, the fault-handler and compensation approach does not provide sensible code reuse between the default handlers and customised ones written by the application developer. The programmer must either rely entirely on the default (which simply runs compensators in reverse chronological order), or they must write the entire handler from scratch. There is no opportunity to do some preliminary activity and then invoke the default handler, nor can a programmer access information used by the default handler, such as the order of completion of the sub-activities. For example, one couldn't write a customized handler which runs the compensator of the last completed sub-activity, but not any others.

Even in those situations where the standard model allows the application designer to express their intentions for dealing with a need to cancel a scope, there is almost no basis on which they can reason about the correctness of their handlers. In the traditional Saga model, there is an elegant and useful result which states that provided the compensator X^{-1} for activity X commutes with every possible other activity (from this or any other scope) which runs between X and X^{-1}, and also X^{-1} commutes with the compensator of each such activity, and provided that X followed by X^{-1} is equivalent to a null operation, then a whole execution is equivalent to one where only the successful Sagas ran, and the others took no steps at all. The strong hypotheses of this theorem are unlikely to apply in a B2Bi setting. While there is a clear need for some way to identify the possible states of a more typical system once a fault has occurred and been handled, we know of no theory to facilitate this reasoning.

VI. Discussion

In this paper, we have identified a variety of different styles of processing that are needed to deal with abnormal situations in a system composed from Web Services across organisational boundaries. We illustrated these styles by examples from a realistic e-procurement scenario. We then used this collection of styles to identify some key weaknesses in the standard approach to fault-handling, which combines programming-language constructs for exceptions with application-defined compensators that semantically undo completed sub-activities. The newly emerging standard BPEL4WS is based on this flawed standard approach. The shortcomings we identified include some where sensible handling is clumsy but possible to write, and others where we cannot see how to do it at all. We have also shown that even when the handlers are written, the developer has no proper support for reasoning about the correctness of the system in those executions where faults have occurred.

The weaknesses in existing standards and infrastructure lead us to our own research agenda: we wish to make it possible for developers to routinely, and easily, construct EAI and B2Bi applications which maintain state and data consistently, even in executions with failures. We want to discover how to provide infrastructure that allows developers to define the consistency needed for their application domain, and allows them to write a system as a matter of routine such that consistency is maintained. Infrastructure should include

- A language to express consistency conditions. For example, one should be able to express declaratively that the merchant will pass on to the customer any costs incurred in cancelling an order
- A language to express system designs, with more powerful and expressive ways to manage the interactions between the normal processing, and the code written by the designer to handle various unusual situations, including failures and cancellation requests. We think that a key idea is to treat cancellation and failure as events, just like message arrival, rather than under a separate and inflexible fault-handling regime. In particular, there must be effective mechanisms for one event handler to inspect and intervene in a controlled way, in the execution of concurrent handlers. This is in contrast to the standard model automatically terminating current activity when a fault occurs in a scope. (We note that WSCI already has a general event handling notation; however it doesn't seem to allow for fault handling which intervenes intelligently in ongoing processing). Another essential aspect of any infrastructure support

is notation to distinguish activities that can always be rolled back, from those that must continue forward under some conditions.

- Tools to check when the system maintains consistency. This means that there must be a foundation for reasoning about system designs, and the states to which they can move the system. We expect that tools will be based on model checkers or other techniques from the formal methods community.

Reference List

[1] BizTalk Server. http://www.microsoft.com/biztalk/

[2] Business Process Execution Language for Web Services (BPEL4WS), Version 1.0. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[3] Web Service Choreography Interface (WSCI) 1.0 Specification. http://wwws.sun.com/software/xml/developers/wsci/

[4] WebSphere MQ Workflow. http://www-3.ibm.com/software/integration/wmqwf/

[5] C. Peltz .http://devresource.hp.com/drc/technical_white_papers/WSOrch/WSOrchestration.pdf

[6] D. Worah and A. Sheth. Transactions in Transactional Workflows. In: *Advanced Transaction Models and Architectures*, Anonymous1997.pp. 3-34.

[7] G. Alonso, A. El Abbadi, M. Kamath, R. Gunthor, D. Agrawal, and C. Mohan, 1994, Technical Report IBM RJ9913.

[8] H. Garcia-Molina and Salem, K., "Sagas," *ACM International Conference on Management of Data (SIGMOD).,* pp. 249-259, 1987.

[9] H. Wächter and A. Reuter. The ConTract Model. In: *Database Transaction Models for Advanced Applications*, ed. A. K. Elmagarmid. Morgan Kaufmann, 1992.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.

[11] L. Bass and B. John, Technical Note CMU/SEI-2002-TN-021, 2002.

[12] S. Mehrotra, R. Rastogi, A. Silberschatz, and H. Korth, "A Transaction Model for Multidatabase Systems," *International Conference on Distributed Computing Systems*.