Just What Could Possibly Go Wrong In B2B Integration?

Dean Kuo¹, Alan Fekete², Paul Greenfield¹, Julian Jang¹ and Doug Palmer¹

¹CSIRO Mathematical and Information Sciences
GPO Box 664 ACTON ACT 2601 Australia
dean.kuo, paul.greenfield, julian.jang, doug.palmer}@csiro.au

²School of Information Technologies, Madsen Building F09
University of Sydney NSW 2006 Australia
fekete@it.usyd.edu.au

Abstract

One important trend in enterprise-scale IT has been the increasing use of business-to-business integration (B2Bi) technologies to automate business processes that cross organisational boundaries, such as the interactions between partner companies along the supply chain. It is relatively easy to describe a pattern of interaction, or choreography, in the case where everything proceeds smoothly. However, the abnormal cases, such as where a process fails or a message is lost, are much more complicated, and risk introducing data and process inconsistencies into computer-based systems. Current B2Bi technologies do not supply an infrastructure that can provide reliability without considerable sophistication from the architects and developers, since performance and organisational autonomy rule out using traditional ACID transactions which ensure consistent data in OLTP systems. As a first step towards guiding architects to the design of B2Bi systems that maintain consistency despite failures, this paper describes a variety of types of failure that can arise in practice, based on a realistic e-procurement scenario. We describe these failures in terms of the different types of state that naturally occur within the distributed system. Different types of failure can lead to different types of inconsistencies if they are not appropriately handled. Understanding the types of failure that need to be handled, or prevented, is essential to an architect or developer who must design and write handlers for all the exceptions that can occur in their workflows.

1 Introduction

Reliability is becoming a key requirement as businesses move towards business-to-business integration (B2Bi) and enterprise application integration (EAI). Reliability requires that all failures and concurrent activities be handled appropriately; otherwise, data may be corrupted and unacceptable system behaviour will occur. Providing reliability is difficult as there are many types of failures, and they can happen at just about anytime, even while handling previous failures. Dealing with failure scenarios is a major source of complexity in the architecture and implementation of a B2Bi and EAI application.

A typical B2Bi application requires the integration of a number of autonomous components that belong to different organisations. This type of application is commonly referred to as a loosely-coupled distributed system. The interactions within such a system are often peer-to-peer, stateful and they may last from minutes to months or beyond. For example, in e-procurement and supply chain management systems, an order may take weeks to complete, from its start when the customer first requests a quote to its finish when the customer has paid and received their goods. In this paper, we refer to these types of applications as long running business transactions.
In this paper, we describe the types of situations that can cause a long running business transaction to deviate from its normal processing paths, and then indicate how these deviations need to be handled in order that the resulting system behaviour remains acceptable to all involved parties. Our descriptions use examples derived from a realistic e-procurement scenario to illustrate the type of situations that can arise in practice.

This description can help architects and developers ensure that their designs and implementations will maintain consistency in the presence of failures and concurrent activities. The description also provides a way to evaluate current and proposed B2Bi and EAI technologies on their support for failure handling (often referred to as transactional support). Current technologies have limited support for handling failures and the example scenarios show what is still lacking in these technologies, and may lead to enhanced fault handling capabilities in the future.

The structure of the paper is organised as follows. In Section 0, we describe related work. Section 3 describes normal processing of the e-procurement scenario. In Section 4, we define the different types of state in loosely coupled distributed environment followed by a description of the types of failures that can occur, with respect to states, in Section 5. Conclusions and future work are presented in Section 6.

2 Background and Related Work

2.1 Web Services

There are powerful commercial pressures towards the use of business-to-business integration (B2Bi). Partner organisations can reduce costs and improve their efficiency by developing computer-based systems that automate their interactions, for example by connecting applications along an entire supply chain. The maturing Web Services distributed computing platform provides an ideal platform for this integration by providing the low-level mechanisms for connectivity, data interchange and cross-application integration invocation through standards such as XML, SOAP [4], WSDL [7] and UDDI [5]. Web Services are relatively light-weight and simple, and extensive support is offered by most IT vendors.

Some B2B applications are built around the stateful interactions between a set of autonomous components that occur over an extended period of time – from minutes to months or beyond. An example is e-procurement where a customer initiates an order by sending a quote request to the merchant who replies within a specified time period with a quote. If the customer decides to proceed with the order, then a purchase order is sent to the merchant; otherwise, the interaction terminates. If the order proceeds, the merchant sends an invoice to the customer, the customer then sends payment to the merchant followed by the merchant sending a receipt. The merchant sends a message to the customer when ordered goods have been shipped (goods in transit) and the customer sends the merchant an acknowledgment when goods have been received. We refer to this type of interaction as a B2B protocol and contract. In Web Services terminology, this is referred to as Web Service choreography.

A business process specifies the behaviour of one of the systems participating in a B2B protocol. Business processes can be specified using workflow modelling languages such as BPEL4WS [2] and WSCI [6] and they can be implemented in workflow technologies such as BizTalk Server [1] and WebSphere MQ Workflow [9]. Business processes can also be implemented using programming languages such as C#, C++, Java.
Defining a B2B protocol and implementing business processes is a straightforward exercise if nothing ever goes wrong. Reality is somewhat different though, as shown in [12] 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 arise from the way that 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 impacted by other concurrent business processes as they access and update shared resources.

Existing products and standards provide some support for dealing with failures and avoiding the resultant inconsistencies. Most choreography and orchestration standards support defining a step of a business process as an ACID transaction, with locking and logging ensuring that the step runs indivisibly and that it is rolled back in case of failure. Furthermore there is widespread support for extended transaction models where the developer can define a compensator to perform a semantic inverse for a completed activity. This style of failure handling derives from Sagas [14] which are explained below. There are also emerging standards, such as WS-Transactions [8] and BTP [3], for coordinating the orderly completion of an activity across partner sites. This allows some level of consistency in the outcome of a distributed activity by ensuring that either all parties complete successfully or else all terminate in a failure state.

There are still numerous open research questions that need to be addressed before architects and developers can systematically design and implement reliable B2Bi applications in a loosely coupled distributed environment. They include:

- Design of mechanisms for expressing what consistency means for a particular application domain;
- Deciding on enhancements to current business process modelling languages such as BPEL4WS [2] and WSCI [6], and to workflow technologies such as BizTalk Server [1] and WebSphere MQ Workflow [9], to reduce the effort required to handle failures;
- Development of a theory to be able to reason about a design and implementation to show that it leads to a reliable system (one which handles failures without producing inconsistencies).

An important step towards addressing these open research questions is the description of the types of failures that can occur and how they need to be handled. This is the contribution of this paper, providing a thorough understanding of the issues involved. This description also leads to the outline of a framework which can be used to evaluate the failure handling capabilities of current B2Bi and EAI technologies.

2.2 Database transactions

The database community had considerable early success in dealing with problems arising from failures and concurrent activities through the adoption of mechanisms based on ACID transactions [16]. The key insight underlying this work 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 context. 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. The mechanisms for concurrency control and recovery used by ACID transactions rely on resource locking. It has been well accepted that locking is inappropriate for processes that last a long time – even minutes, let alone for months – because other activities can be prevented from executing for excessively long periods of time. In addition, in B2Bi, the interaction runs between different companies, and probably over the public Internet. Locking in such an environment can lead to simple denial of service attacks, as well as unacceptable dependencies between organisations.

A large number of advanced/extended transaction models were proposed in the 1980s, targeting applications such as Computer Aided Design and other domains requiring support for long duration transactions without the need for all of the ACID properties. Many of these models are collected in the book [11]. Some of the mechanisms (protocols) introduced from advanced/extended transaction models are useful for handling certain types of failures. The most influential is Sagas [14] which decomposes a long running transaction (a Saga) into a sequence of ACID transaction with the extra property that a failure anywhere during the sequence causes compensation activities to be issued for each completed ACID transaction of the sequence in reverse chronological order. The application programmer is expected to write a compensator for each activity. One significant assumption made in this work is that a compensation activity is guaranteed to successfully execute. The significance of this assumption is discussed later in this paper. Commercial workflow products designed for B2Bi long running transactions (applications) such as BizTalk Server from Microsoft [1] and business process modelling languages such as BPEL4WS [2] and WSCI [6] already include support for compensation transactions.

There are, however, significant differences in the requirements of B2Bi and EAI applications that make ACID transactions and advanced/extended transactions inadequate to the task of ensuring consistency. ACID transactions, as used in the On-Line Transaction Processing (OLTP) world make several assumptions:

- Transactions are client-server, with one party starting and controlling the transaction
- Transactions are of short duration (milliseconds to a second or two)
- Components and databases all lie within a single trusted domain under the control of the same administrator group
- Both the client and server can unilaterally abort a transaction while the transaction is active.

In contrast, long running business transactions for many B2Bi applications are long in duration, interactions are peer-to-peer and stateful, in most situations no one can unilaterally abort (cancel) a transaction and the components belong to different organisations (trust domains). In addition, cancelling a long running business process is often very complex and is not as simple as rolling back an ACID transaction. As a result, mechanisms from ACID transactions, such as 2 phase locking, 2 phase commit [16], and advanced/extended transaction models are insufficient to completely deal with failures for B2Bi applications.
Our e-procurement scenario shows that existing methods are not sufficient to deal with all the types of failures and inconsistency that can occur in a long business transaction. The approach in this paper is not to propose yet another advanced/extended transaction model. Instead, we offer a better appreciation of the problem by extracting the types of failures that can occur, and showing how they should be handled and what sort of unacceptable behaviours will occur if they are not handled appropriately. We plan in future to define, formally, what correctness means for B2Bi applications. We will then be in a position to propose extensions to existing standards and develop tools to enable architects and developers to systematically design and implement reliable B2B applications.

3 E-procurement Case Study

In this section, we describe briefly a merchant’s business process in an e-procurement scenario that was derived from a consultancy project. The business process is very simple and is shown in Figure 1. The customer’s business process is not described since it mirrors/complements the merchant’s business process and can be easily derived from this process.

The merchant’s 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, they are registered and have no overdue payments. For invalid customers, a message informing the customer that they are no longer a valid customer is sent and the business process terminates. For valid customers, a quote is calculated and sent to the customer. The merchant will then receive a purchase order if the customer proceeds with the order. Goods are then reserved and the payment and delivery processes are started. These two processes 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. The workflow terminates after the delivery and payment processes have both completed.
If nothing can ever go wrong in this procurement application, the design and implementation of both the merchant's and customer's business processes will be trivial. The real world is much more complex, and section 5 describes just what could possibly go wrong (failures), which we refer to as a deviation from normal processing, in terms of the different types of state identified in Section 4. This will confirm the folk wisdom that failure handling is much more complex to specify and implement than the normal processing paths.

4 States

The main contribution of this paper is a classification of the types of situations that can cause a long running business transaction deviate from a normal processing path. The classification is based on the state of the distributed system.

We have identified three types of state – abstract state, business process state and real world state. We will now describe each of these types of state in detail and the relationships between them, and then describe a set of failures that can occur in terms of these state types.

4.1 Real World State

The real world state is simply the state of the physical world, such as goods on hand and financial agreements.

In our e-procurement scenario, the quantity of each product stored in the warehouse, the physical location in the warehouse where the goods are stored, the conditions of the goods (damaged or in good condition) and the locations of the warehouses are all part of the real world state.

4.2 Abstract State

The abstract state is a computer-based representation of the real world state.

Each component in a loosely coupled distributed computing system has state. This state is an abstract state as it is based on the data held within these computer-based models of the real world. This model includes information such as the expected availability of each product, the location in the warehouse where the goods are supposed to be stored and the customer’s delivery address.

The abstract state of a component is not necessarily exposed externally. This means that a component in the distributed environment may have no direct knowledge of the internal abstract state of other components. For example, a customer would not normally know the level of stock available for any particular product; in airline reservations, a passenger would not know the exact number of seats available for a particular flight. However, it is possible for an external component to derive partial information about the abstract state of a component by considering the component's behaviour – e.g. if a merchant accepts a purchase order then the customer knows that the merchant believes it will have at least the ordered quantity available at the time of shipment. The merchant may provide the customer with direct interfaces to query the value of the internal state, but this can never be more than a snapshot and so is potentially inaccurate.

In an ideal world, the abstract state would correspond precisely to the real world state but this is often not the case. The accuracy of the abstract state is dependent on the abstract state being updated correctly every time the real world state changes. Data entry errors can occur, there may be unanticipated changes in the
world that are not captured in the computer system, and the costs to maintain accuracy could be too
great. For all these reasons, we must consider the possibility of mismatch between the abstract state
and the real world state. The inaccuracy may cause exceptions to be thrown at various stages in the
business process. For example, in the e-procurement scenario, the abstract state in the database may
show that there are sufficient stocks available for an order but this may be found not to be so at the
time of pickup for the delivery.

In other circumstances, the mismatch may arise because the real world state is unknown. For example,
the abstract state does not know the exact location of a truck unless a GPS is installed and linked back
to the computer system in real-time, so typically the abstract state will show merely that the truck is in
transit.

4.3 Business Process State

A business process is defined by a set of activities and a specification of the order in which the
activities are required to execute. In the e-procurement example, the business process for the merchant
includes receiving a quote, verifying that the customer is a registered customer with no overdue
payments, calculating a quote and sending the quote to the customer. The business process state is the
point that the process is up to in its execution. A business process may contain forks (sets of activities
that execute in parallel), thus a business process state can point to multiple positions in a process.

Examples of process state in e-procurement include states such as quote request received, quote sent, invoice
sent, payment received and receipt sent.

There are two types of business processes. One type defines the internal activities of a component and
the other defines the externally visible behaviour. In the e-procurement example, the merchant would
not expose to its partners (e.g. customers and suppliers) the internal processes but would expose a sub-
process which consists entirely of activities that interact with its partners. In BPEL4WS [2], these are
respectively referred to as (executable) business processes and abstract business processes/business protocol. There
are thus two types of process state – internal and external. An external business state encapsulates or
summarizes a set of internal states. For example, the external process state payment received encapsulates
the internal state of received payment, awaiting the validation (of payment) and received validation.

5 What Could Go Wrong?

This section lists a number of situations where the business process deviates from normal processing as
a result of a failure or exception during execution. Some of these deviations are recoverable while
others are not. This means an architect must ensure that there are processes defined to recover from
recoverable deviations and to prevent unrecoverable deviations; otherwise, unacceptable behaviour will
occur that may result in adverse outcomes, such as financial loss. The events that cause a deviation
from the normal processing paths can occur at any time, even when handling previous deviations,
making it more difficult to ensure correct behaviour under all circumstances.

5.1 Failures due to inaccurate abstract State

An abstract state is a representation of a real world state. The real world state of a warehouse is the physical
state of the warehouse – that is, what products, and in what quantity, are stored in the warehouse as
well as their storage location in the warehouse. An inventory system is an abstract representation of the
real world state of a warehouse.
In an ideal world, the abstract state and the real world state would be consistent with each other. Unfortunately, this is not feasible due to the cost and effort required to keep them synchronised at all times, and in some cases, this is actually not physically possible. Difference between the real world state and the abstract state can also be caused by real world events that are not captured within the computer system. For example, goods can become damaged in a warehouse or can be stolen and are thus no longer available for sale. This state mismatch will persist until the inventory system is reconciled with the actual physical goods in the warehouse, something that may not happen until the next stock take.

In the e-procurement scenario, a failure will occur when there are actually insufficient goods in the warehouse to satisfy an order but the inventory states otherwise – the abstract state is not consistent with the real world state.

As it is not feasible to keep the abstract and real world state synchronised at all times, these types of failures are unavoidable. Thus, to ensure correctness (that is, avoid unacceptable behaviours), we must be able to handle exceptions caused by inaccurate abstract state.

Correct handling of exceptions arising from inaccurate abstract state is application dependent. In the e-procurement scenario, if there is insufficient stock available for delivery then there are various ways the exception can be handled. They include:

- Delay the order until a backorder arrives and reschedule delivery;
- For orders that include other products, send all available goods as scheduled and send unavailable goods when they become available – that is, partial fulfilment of an order;
- Cancel the order.

Depending on the circumstances, how this exception is handled may depend on the decisions and policies of the merchant and/or customer.

Notice that when something goes wrong, it is not the case here that the system should by default cancel the business process (abort the transaction), unlike the standard error handling behaviour found in ACID transactions and advanced/extended transaction models. It is often also not possible for either party to unilaterally decide to cancel when a failure occurs. This is also the case in the other types of failures described below.

If this exception is not appropriately handled then unacceptable behaviour may result. For example, goods may never be delivered to the customer but the customer is still invoiced and sends payment to the merchant; or the business process may never terminate.

5.2 Phantom failures due to inaccurate abstract state

Inaccurate abstract state can also cause phantom failures to be thrown. An example is when the abstract state believes that the transportation for a delivery of an order has not arrived by its deadline and this causes an exception to be thrown but in the real world state, the transportation has actually arrived, and it is just that the abstract state has not been updated to reflect the new real world state in time to avoid the exception.

Incorrect handling of this type of exception can cause a system to thrash. In a heavily loaded network/system, there may be a long delay before the initial exception is handled which may cause
further instances of the same exception to be thrown over and over again, thus increasing an already overloaded system which then grinds the system and network to a halt.

5.3 Prohibited Abstract and Real World States

Integrity constraints define, via the abstract state, that certain real world states are prohibited. Examples of integrity constraints for e-procurement include the requirement that each customer does not exceed their credit limit, and that available stock for a particular product is not below a specified amount unless there is an active backorder.

Exceptions are thrown during a business process when an integrity constraint is violated. They must be appropriately dealt with; otherwise unacceptable behaviour can occur. How an exception should be handled is application dependent. For example, if a customer exceeds his or her credit limit while placing a new order, then there are a number of ways to handle the exception. They include: increasing their credit limit (maybe temporarily); requesting that the customer deposit funds into their account; or cancelling the order which caused their credit limit to be exceeded. Notice again that there are a number of possibilities to handle the exception and that the business process does not have to be cancelled as a result of the exception.

The more interesting example relates to integrity constraints that contain disjunctions – e.g. available stock is not below a specified level unless (or) there is an active backorder. If an order reduces the available stock to something below the acceptable level, the exception should definitely not cancel the order but instead trigger off a new backorder. If the backorder throws an exception, perhaps the supplier no longer stock the ordered product, then the merchant can try to find an alternative supplier and if no such supplier can be found, remove the integrity constraint for this product and update the inventory to reflect that this product will no longer be available once all remaining stocks have been sold.

In the above examples, a business transaction’s activity causes an exception. It is also possible for exceptions to be thrown due to an activity not occurring as shown in Section 5.4.

5.4 Prohibited Time-Based Internal Process State

An interesting type of prohibited process state arises from expected events and when they are supposed to occur. A business process may specify when an activity in a business process has to occur, and if it doesn’t happen by the specified time then an exception should be thrown.

A good example is the requirement that payment from the customer should be received by its due date. When this deadline is missed, there are many ways to handle the exception. They include notifying the customer of overdue payment and extending the deadline for payment; charging an additional late fee and sending a new invoice; cancelling the order if the goods have not been shipped, (and possibly charging the customer a cancellation fee); and as a last resort, initiating legal action (a human oriented activity) to recover costs.

Additional complexity can be caused by the asynchronous nature of the interactions between the merchant and customer’s business processes. If the merchant sends a new invoice, including the late fee, in response to an overdue payment then it is possible that payment for the initial invoice will then be received. The merchant would then wait for payment for the late fee payment only.
If exceptions such as overdue payment are not appropriately handled then the merchant’s business process may never terminate, and more importantly, the merchant might never get paid. Care must also be taken when defining how to handle an exception between the customer and merchant; otherwise it is easy to end up with unacceptable outcomes. For example, customer may pay the original invoice, then receive another invoice (which covers the original charge plus a cancellation fee) and pay that in full as well. Inconsistency can also occur if, after the merchant sends a second invoice which includes a late fee, they receive payment from the customer for the original invoice but then forget about the late fee which is still outstanding.

5.5 Mismatch between Internal Process State and Abstract State

Successful execution of an activity from a particular internal process state may depend on the abstract state having appropriate values at that time. For example, there must be sufficient funds available at the time the customer wishes to send payment to the merchant and there have to be sufficient stocks available at the warehouse for delivery when transportation arrives. The most intuitive scheme for specifying the conditions required for successful completion of an activity is by attaching predicates (pre-conditions) to activities in a long running business transaction [13] and [15].

In a long running business transaction, if an activity \( A \) depends on a condition to successfully execute, the business transaction will typically execute an earlier activity \( A' \) in the business transaction to ensure that the condition will be true when the activity \( A \) executes – for example, in e-procurement, the merchant will typically reserve the quantity of goods required by an order so that there will be sufficient goods available at the time of delivery. However, just because goods have been successfully reserved does not mean that the goods are already stored in the warehouse since the reserved goods may be goods that are scheduled to arrive from the supplier before the delivery date – that is, the predicate may not be actually true when the activity \( A' \) executes but is expected to be true by the time \( A \) executes. Furthermore, if \( A' \) successfully executes, there is actually no guarantee that the predicate will be true when activity \( A \) executes since the goods may not have arrived from the supplier in time or there may be a mismatch between the external process state and real world state. However, in most cases the predicate will be true when activity \( A \) executes.

There are multiple schemes that can handle the exception which occurs in those cases where the predicate fails at the time \( A \) is due to execute, and they are similar to those described in Section 5.1

If the exception is not appropriately handled, then unacceptable behaviour can occur; for example the merchant may never deliver the goods while the customer has paid for the goods.

5.6 Mismatches between External Process States

B2B integration often requires autonomous components and long stateful interactions between numerous participants. The e-procurement scenario is a good example; the participants include a merchant, customer, bank and a shipper. Each component exposes its external business process state to its partners. However, the external process state of one component may not be compatible with another component’s external process state.

Examples of such incompatibility include:

- The customer is in the received receipt external business process state while the merchant is in the invoiced state. These two external states are incompatible since the customer could not have possibly
received a receipt if the merchant has not even sent the invoice. However, the external business process state paid for the customer is compatible with the merchant’s external business process state invoiced since payment may be in transit.

- The customer is in the external business state cancelled state while the merchant is in the successfully completed state.
- The merchant is in an external business state which is awaiting payment but the customer is in a state which indicates that its business process has terminated.

These incompatibilities can be prevented by ensuring that the components’ (dynamic) behaviour is compatible with respect to a B2B orchestration protocol. That is, when one component sends a message to another component, the destination component is expecting that message, and whenever a component is awaiting the arrival of a message, some other component will (eventually) send a message of the correct type.

In a correct design and implementation of a B2Bi application, the business processes participating in the B2B interaction should never be in a situation where one business process is waiting for events (messages) that will never happen, neither should any component receive unexpected events. Such incompatibilities will cause business processes to never terminate and cause messages to be lost or queued somewhere, never to be properly processed.

We are starting to investigate whether model checking [10] can be used to automate the reasoning needed to verify that a set of business processes are compatible with respect a given B2B protocol. There is existing work [11] that shows the use of such techniques for simple cases with normal processing only, but it is not clear whether the ideas will scale to the complex protocols where each party has many failure-handling mechanisms. Unlike the exceptions described in Sections 5.1 to 5.5, this type of situation should never occur in a correctly designed and implemented system.

5.7 Incompatible Abstract States

Even though abstract states are internal to a component, two or more components’ internal state may be incompatible in a B2B interaction. In e-procurement, examples of incompatible abstract states include the following:

- The amount payable for an order differs in the customer and merchant’s abstract state. Similarly, if an order is cancelled, the merchant and customer may differ in their understanding of the cancellation fee that is payable.
- Similarly, the products ordered in an order differ in the merchant and customer’s abstract state.

Incompatible abstract state become evident when one of the business processes throws an exception after receiving a message from another component – e.g. the merchant would throw an exception when it receives payment from the customer in which the amount is incorrect.

The exception needs to be appropriately handled and the incompatibility between the abstract states resolved; otherwise, the environment would be left in an inconsistent state. This is where non-repudiation of the initial purchase order is required to resolve the problem.
6 Conclusion and Future Work

Handling exceptions is critical in the design and implementation of reliable B2B applications in a loosely coupled distributed environment. In this paper, we described the types of exceptions that can occur. We also gave realistic examples, derived from an e-procurement system, of each type of exception and describe how they need to be handled to maintain consistency. From these examples, it becomes evident that existing commercial workflow products and standards for Web Service orchestration and choreography do not provide sufficient support for handling executions which cause deviations from normal processing.

Mechanisms based on ACID transactions are inappropriate for B2B applications as ACID transactions are intended for transactions that are short in duration (milliseconds), the components are tightly coupled and have to be within a single trusted domain. In contrast, B2B applications, such as e-procurement, require long running transactions (minutes to months and beyond), the components are loosely coupled and belong to different trust domains. Other advance/extended transaction models are also inappropriate as they can not handle all the different types of exceptions (deviations) that can occur.

An open research question is how to systematically design and implement reliable B2B applications – that is, how to maintain consistency even when exceptions (failures) occur. The list of different types of exceptions described in this paper is an important step towards our goal as it distinguishes between the types of exceptions can and cannot already be handled by existing standards and products, thus indicating where extensions are required. The list can also be used to evaluate exception handling capabilities (transactional capabilities) for existing products and standards in this space.

It may be unrealistic to be able to prevent all inconsistencies to occur in loosely coupled distributed systems, and another open research question is how to design B2B applications that are capable of self-monitoring, so that inconsistencies are quickly recognised and corrected. Subtle differences in contract interpretation can lead, over time, to a number of the inconsistencies described in this paper. Since these differences are not explicit, appropriate diagnostic tools and techniques need to be developed to assist in detecting such inconsistencies.

Our future work includes the development of a semantic model which will lead us to be able to formally define what consistency means. This model will enable us to formally verify that the components participating in a B2B protocol will not introduce inconsistencies even if failures occur. We will also be investigating whether model checking [10] can automate the process of verification.

These activities will provide us with a better appreciation of the real difficulties in maintaining consistencies and should lead us to architectural design patterns, protocols and tools that will assist architects and software engineers in their efforts to systematically design and implement reliable B2B applications in the future.

Reference List


