

# Transactions in Loosely Coupled Distributed Systems

Alan Fekete†

Paul Greenfield★

Dean Kuo★

Julian Jang★,†

†School of Information Technologies  
University of Sydney 2006  
Australia  
fekete@it.usyd.edu.au

★Network Applications and Technologies Group  
CSIRO Mathematical and Information Sciences  
Locked Bag 17, North Ryde 1670  
Australia

Paul.Greenfield@csiro.au, Dean.Kuo@csiro.au, Julian.Jang@csiro.au

## Abstract

An exciting trend in enterprise computing lies in the integration of applications across an organisation and even between organisations. This allows the provision of services by automated business processes that coordinate activity among several collaborating companies. The best successes in this type of distributed system come through use of Web Services technologies, which allow interoperation between applications and workflows, through open standards based on XML and SOAP. In this talk we discuss some unresolved issues which confront a designer who wants to ensure the consistency of data across several collaborating workflows or applications.

*Keywords:* Web services, consistency, choreography, compensation, composition of services, interface specification, failure atomicity, isolation.

## 1 Loosely Coupled Distributed Systems for Automating Business Processes

Businesses continuously seek methods to automate tasks to reduce costs. With the advent of distributed computing, it is possible to implement Enterprise Application Integration (EAI) and Business-to-Business integration (B2Bi) solutions to automate business processes. The key to success is interoperability between loosely coupled components implemented and hosted on independent platforms. These may be within the same enterprise or indeed they can be owned by different organisations. We say the components are loosely coupled because they are written independently, and they can be combined in multiple ways. In fact, in many cases the component encapsulates a legacy IT system, such as inventory management or financial package. The essence of such systems is the ability of collaborating organisations to allow controlled external access to their internal IT systems; this clearly raises lots of questions familiar to database researchers, such as semantic data conversion, access control, data integrity, etc.

There are a number of distributed computing platforms available for implementing interoperation between application components. Some, like J2EE or COM+ are proving popular and powerful within

a tightly controlled organisation. However, they lack the cross-platform interoperability needed for industry-wide collaboration. Other technologies such as CORBA have proved too heavy-weight and complex for many users.

Web Services is a maturing distributed computing platform, that is currently attracting a lot of attention (Freemantle, Weerawarana & Khalaf 2002). This approach to interoperability relies on both XML messaging and the internet. It is based on a set of standards managed by the vendor-neutral W3C with support from all major IT vendors. These standards include SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery Integration). SOAP is an XML-based protocol for exchanging information between components and is independent on the transport protocol. However, most SOAP implementations today rely on HTTP for transport. SOAP provides the universal interoperability between components that are implemented and hosted on different platforms. WSDL describes a service's static interface — type information and operations of the service — and is similar to interface definition language from J2EE and CORBA. UDDI serves as *white* and *yellow* pages directories. UDDI also provides *green* pages which describe the *technical* information about a service, such as its WSDL description.

The progress of Web Services so far indicate that it will become a broadly accepted distributed computing platform for EAI and B2Bi since it provides universal interoperability, and there is plentiful tool support to expose existing legacy and new applications as Web Services. This approach is lightweight; Vendors such as Microsoft, IBM and Apache already provide support for Web Services technologies.

The Web Services standards mentioned above allow outsiders to invoke a remote application, and provide arguments and receive results in a format that can be understood on both sides. That is, a remote system can provide a callable service. For effective EAI and B2Bi, we need more — a long running business process in one organisation may need to interact with another long running business process, possibly one in another organisation, through a sequence of operations flowing in both directions.

It's important to distinguish between the general structure of a loosely coupled distributed system based on Web Services, and the simpler situation where a long running business process includes calls on remote operations exposed as Web Services. The general situation has partners each of which maintains per-collaboration state throughout a long collabora-

tion; the simpler structure has state in the long running process, but not in the service providers.

Designing a general loosely coupled distributed system thus requires description of long running business processes which exchange messages each of which is done using SOAP. It's important to distinguish between a description of an *interface* or abstract business process, which is the way external collaborators interact with a service, and the *orchestration* or executable business process, which describes when each activity is invoked, how control flows, etc, in sufficient detail for a workflow engine to execute. Of course, the separate activities can themselves be complex business processes.

A number of standards have been proposed to describe orchestration, interfaces and related issues for the interaction of long running business processes. In this talk we will focus on an important issue which we feel has not yet been resolved in these standards: how to ensure that the stateful interacting processes keep consistency in the data they are managing, despite failures and concurrency between activities.

### 1.1 Proposed Standards for Web Services

There are now a number of proposed standards for EAI and B2Bi solutions for long running business transactions. Some of these are competing standards while others complement each other.

The standards that are most relevant to our research are those that describe the *choreography* of a service (that is, the dynamic, message ordering aspects). These are all built on top of WSDL which describes the static interface of a Web Service: type information, the signature of each operation supported by the Web Service (that is, the name of the operation, its input and output messages) and binding information.

One set of standards relates to business process orchestration — how to combine a set of Web Services to automate a business process. Initially, IBM proposed the Web Services Flow Language (WSFL) and Microsoft implemented Biztalk with XLANG as its workflow language for describing business process orchestration. Recently, BEA, IBM and Microsoft combined to release three specifications (BPEL4WS, WS-Transactions and WS-Coordination) that represent a convergence of WSFL and XLANG. Collectively, these specifications deal with implementing a process from collaborating activities and workflows.

BPEL4WS is concerned with when each activity is invoked, what data flows between activities, what alternative execution is produced on different failures, etc. WS-Coordination describes extensible mechanisms for providing a protocol that coordinates activity between participants, including the creation and propagation of the context, and registration with coordinators. WS-Transaction (sometimes abbreviated to WS-TX) defines two specific families of protocols. One family (AT) manages a distributed atomic transaction. Members of this family use different commit protocols, all of which are variants of Two Phase Commit. Another family (BA) coordinates a scope (possibly nested) in a long running Business Activity. The coordinator interacts with the participants each of which is performing an activity within the scope. The mechanisms are derived from Sagas, described below. In particular, the coordinator can deal with a problem by causing each participant to run an application-specific compensation for the completed activities.

There are several (competing) standards to describe the choreography aspects of interfaces that is,

dynamic behaviour as seen from the outside. Two of these are WSCL from HP and WSCI from a consortium of BEA, Intalio, SAP, SUN. Both have been submitted to W3C and their status is currently as a *W3C Note*. Both WSCI and WSCL describe the order of operations as a directed acyclic graph. WSCI is much more elaborate than WSCL since the latter only describes the order in which operations are allowed to occur. WSCI goes further by describing the transactional boundaries of operations, and it provides thread management and failure handling through compensation activities. Other standards for Web Services choreography description include BPML from BPMI (BPMI.org) and ebCPP from OASIS.

## 2 Consistency Issues

An important goal for the designers of a loosely coupled distributed system is to maintain consistency for interacting long running business processes in the presence of failures and concurrent activities. The objectives are thus similar to transaction processing for database management systems.

However, the environment of a loosely coupled distributed system is very different from that standard in the OLTP world. The loosely coupled system is constructed from pieces that need to remain autonomous, because they were written, and are run, independently. In many cases, they belong to different organisations which are competitors as well as collaborators; the organisations' goals are not the same, and each can't extend trust to the other. The pieces use many resources and may include human intervention, so each lasts a long time. For these reasons, it is unacceptable for one business process to hold process-duration locks in another business process. The lack of locks means that processes can't be completely isolated from one another; also this means that one can't follow traditional rollback, based on the restoration of before images kept in a log.

A typical example of two collaborating long running business processes is e-procurement where a customer and merchant have a stateful interaction to manage an order. The collaboration is initiated by a customer sending a quote request to the merchant who replies with a quote, the customer then may proceed with the order by sending in a purchase order. The collaboration would include the payment activities: merchant sending an invoice, customer sends payment before its due date, and the merchant sends back a receipt. There would also be interactions concerning the shipment of goods, with a notification from the merchant informing the customer about shipping plans, and a confirmation from the customer when the delivery has occurred. We show a possible pattern of interaction in the sequence diagram in Figure 1; note that time increases down the page, however be aware that the messages concerned with payment can be interleaved arbitrarily with those concerned with shipment, rather than separated as in the diagram.

The ultimate goal is to have systems in which state and data are kept consistent. This will depend on an ability to express what consistency means in each particular application. To begin, we need to identify the variety of consistency conditions in realistic applications. We have classified several distinct sorts of consistency condition.

- Failure atomicity conditions: these refer to the termination status of the activities.

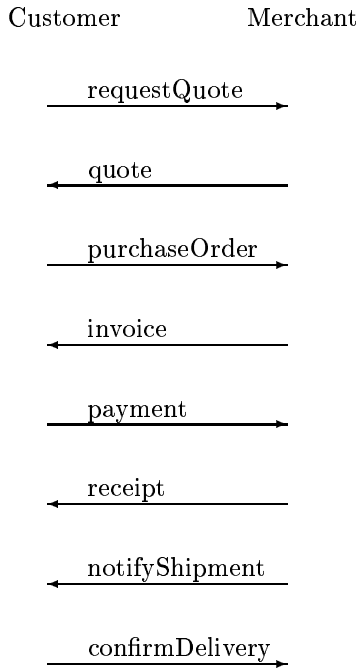


Figure 1: E-procurement interactions

- Isolation properties: these refer to the way state changes during the flow of control through a process, due to interleaved activities of the same or other processes.
- Timeliness properties: these refer to the maximum and minimum period between the occurrence of different situations.

Here, we shall describe, intuitively, what each variety of consistency can mean for the e-procurement example.

For failure atomicity, we need to describe how the various activities can terminate. We need to distinguish between *local* conditions and *global* ones. Local conditions deal with the state of the flow of, and the data accessed by, the activities within one long running business process. Global conditions relate the state and data between collaborating processes.

As an example of a local failure atomicity condition in the e-procurement example, the merchant process which is handling an order must terminate properly — that is, either a sale is made (goods delivered and payment received) or the order is cancelled (the net result that goods have not been delivered and the customer has not paid for the order). From the merchant’s perspective, if the order goods have been delivered but has not received payment by its due date, then this is an inconsistency.

The other level of consistency we need to address is at a global level. Even if all the activities of the merchant end in a state acceptable to the merchant, and all the activities of the customer end in a state acceptable to the customer, this does not imply that the overall state is jointly acceptable. The global state of the system is inconsistent if the merchant terminates believing that a sale has been successfully completed but the customer terminates believing that the order has been successfully cancelled. Another example that highlights the complexity of the problems is that

even if both customer and merchant believe that the order has been successfully cancelled, this is not consistent if one party believes the cancellation has been accompanied by a cancellation fee and the other party believes that no fee was charged.

Isolation properties are also important for consistency. In a long running business process, it is important to make sure that some properties of data are true when particular activities run. For example, a customer needs to be valid when an order is shipped. However, the activity that checks validity may be done separately from the shipment of goods; for example, the check might occur much earlier, when a quote is issued. In this case, the system needs to ensure that no concurrent business process changes the customer’s validity in between the check and the shipment. Of course, holding locks would ensure this, but it would also be intolerable: the time gap between the validity check and the shipment could be days. No system could allow a collaborator (who may also be a competitor) to hold a lock for this duration: the risk of a denial of service attack is too great. Instead techniques are used such as flagging data that should not change beyond a specific range.

Timeliness has traditionally been seen as a performance issue in the database community, where applications are designed to run very rapidly (for example, by excluding all human interaction from within a transaction). However, in a loosely coupled distributed system, the business logic may be crucially dependent on the time when events occur. For example, the merchant may follow a policy where it will honor a quoted price as long as the order is placed within 7 days, but not for longer. This system is inconsistent if the customer treats the quote as valid for 14 days, as the customer might send an order which the merchant rejects but the customer assumes that the order will necessarily succeed.

While we have described the classes of consistency condition separately, we should notice that they may interact with one another. For example, the different outcomes of an isolation property check can result in different possibilities for termination status, or the way interleaved activities can make changes to an observed state may vary depending on the elapsed time since the condition was checked.

In general, the issue that we feel needs further research is how to define the allowable *stateful interactions* between collaborating processes. This is much harder than defining the allowable situations for one long running business process that interacts with external services which do not maintain per-collaboration state. We use the term *contract* to refer to the set of allowable interactions between collaborating parties each of which is a stateful process.

### 3 Transaction Ideas from the Database Community

The concurrency control and recovery mechanisms that ensure preservation of consistency between data items within a single database, were important discoveries of early database research. Indeed, the very idea of an “ACID transaction”, that is, the recognition that one could provide mechanisms in the infrastructure and relieve the application programmer from worrying about failure and interleaving, was a major contribution for which Jim Gray won a Turing Award in 1998. Single database locking and logging are mature technologies and there has been little published research with real impact in this area for the past decade.

The concept and implementation of ACID transactions assumes a particular type of environment, where activity runs very quickly and remains within a trust boundary. Thus in this sort of environment it is feasible to hold locks on data till a transaction completes, and it is reasonable to delay, or to rollback and retry, whenever two transactions seek to use the same data concurrently.

It became clear very early on that the ACID approach is not appropriate for many activities that manipulate data in environments where these assumptions are not valid.

A range of "extended transaction models" were proposed in the 1980s, aimed at cooperative processes like collaborative design, or long running business processes. The 1990s saw many of the leading researchers focus on suitable models and mechanisms for ensuring consistency within workflow infrastructure. This research is summarised in (Alonso, Agrawal, El Abbadi, Kamath, Gunthor & Mohan 1996, Kamath & Ramamritham 1996, Worah & Sheth 1997, Kamath & Ramamritham 1998).

In many ways, the issues in loosely coupled distributed systems are more complex variations on what arises for workflow infrastructure. The extra issues arise from the fact that several workflows are interacting, and each must keep per-collaboration information about the state of its peers.

Of course, ACID transactions still play an important role in a loosely coupled system, since many of the separate activities are written as transactions that access one or more databases. Sometimes, several activities will run rapidly enough within a trust domain, accessing databases that have the hooks for Two Phase Commit. If this is so, the activities can be encapsulated as a distributed transaction. The overall system certainly needs to know which activities have these transactional properties. It will propagate transaction context appropriately, and it will be able to undo these activities by standard database mechanisms, rather than in more complex application-specific ways.

In this section we will summarise and critique two early works that seem to us to be very significant for supporting consistency within loosely coupled distributed systems. A notion of compensation was defined in Sagas (Garcia-Molina & Salem 1987), and has influenced some of the standards (such as BPEL4WS and WSCI) as a mechanism for managing when a process fails. The other mechanism we discuss is the use of assertions as invariants on entry and exit from activities; this comes from ConTracts (Wachter & Reuter 1992).

### 3.1 Compensation

In (Garcia-Molina & Salem 1987), it was 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. Thus the underlying mechanism would 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 (Garcia-Molina & Salem 1987) 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.

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 (Garcia-Molina & Salem 1987), 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.

It is easy to prove that a concurrent execution of Sagas will be serialisable provided that each compensator commutes with every task and with every compensator that executed between the task and its compensator. However it is almost impossible for many tasks to write compensators with such a strong property. In general the drawback of the compensation approach lies in the difficulty of writing a compensator that winds back the original task even from states that have changed significantly. The current BPEL4WS specification restricts the compensator to seeing only the state of any variables as recorded at the end of the original workflow; future specification versions will certainly relax this limitation, but still the whole compensation approach gives no guidance to the developer on how to write the compensator, or even on how to test whether it has been written correctly.

A compensator ought to remove not only the direct effect of the original task, but also any indirect effect through other activities which read the data now discovered to be inappropriate. For example, if the merchant has recorded a large order, and this has been used to calculate a bonus for the relevant region manager, then the compensator for the order task ought to recalculate the manager's bonus (or rather, to maintain modularity, the compensator should somehow trigger a recalculation in the bonus process).

It may also be the case that the execution of one compensator ought to influence the activity of another compensator. This influence may not be possible when the compensators are run in the reverse chronological order of the original tasks. For example, in the original workflow the merchant might arrange shipment then receive payment from the customer. During compensation, any charges incurred by the merchant as it cancels the shipment, need to be deducted from the payment amount before the customer gets sent the refund.

Another difficulty with compensation-based systems lies in their assumption that the compensator always runs successfully. In real systems we have to deal with the case where we want to compensate for an overpayment, but the recipient has already spent the money! A business process should not lead to inconsistent data when a compensator itself can't be executed.

### 3.2 Invariants on tasks

In a traditional ACID transaction, each individual operation operates on data which is unchanged from that seen by earlier steps in the transaction. Thus, if one step checks the validity of a customer, then we can be sure the customer is still valid when all later steps use the customer information. In a long running business process, locks can't be held for more than a few seconds, so it is harder to ensure that the customer's validity is preserved once it has been

checked. In (Wachter & Reuter 1992), a general workflow description approach was introduced. As well as providing a language to express the sequence of steps, including the input and output parameters, a ConTract made explicit the conditions each task needed to complete successfully. These are called *entry invariants* of the task. The ConTract also expresses the conditions that are true at the end of a task, as *exit invariants*. For example, if a shipment task should only be attempted when the customer is valid, the application developer needed to state that customer validity is an entry invariant for the shipment task. The developer could also write that customer validity is an exit invariant for the validity checking task. The syntax of ConTracts also allowed each exit invariant to indicate how it was to be preserved: several possibilities were defined.

- Locks can be held, preventing any change to the data that was checked
- The exit condition can be preserved throughout the period, by having a check run on each interleaved activity; this other activity would be rejected if it could violate the truth of the exit condition
- The exit condition could be allowed to become false, and the system would re-run the check at the time when another task needs this condition for entry. In this situation, the developer would need to describe how to proceed if the revalidation check failed, by a “Conflict resolution” step on the task with the entry invariant. For example, the developer could indicate a procedure to call that would restore the invariant.

These techniques have not been fully implemented, as far as we know. They require sophisticated manipulation of logic by the workflow engine, but we see these as offering a powerful framework to express the isolation conditions needed in application consistency.

#### 4 A Research Agenda

In this section, we describe some research topics we plan to investigate, in the Network Applications and Technologies group at CSIRO, and the Middleware research group at Sydney University. Readers seeking ideas in this space should also watch closely other researchers such as the Advanced Enterprise Middleware group at IBM ([www.research.ibm.com/AEM/](http://www.research.ibm.com/AEM/)) who are proposing some object-oriented frameworks for expressing sophisticated transaction management alternatives within a single workflow engine (Bennett, Hahm, Leff, Mikalsen, Rasmus, Rayfield & Rouvellou 2000).

The overall target of our research is to make it possible for developers to routinely, and easily, construct EAI and B2Bi applications which maintain the state and data consistently, even in executions with failure and unexpected interleaving.

In traditional database applications, ACID transactions meet this need. They provide a clean structure, where the application developer merely determines the transaction boundaries, and writes each piece of application code to transform the data from the transaction begin to the commit. The developer does not consider failure scenarios or concurrency. The system infrastructure then ensures that the complete application works correctly.

Even in single organisation workflow systems, it is not feasible to have an entire process within a single ACID transaction: the individual components could

not risk holding locks for the duration of a business process, and indeed some legacy components may not support locking/logging at all. In a wide scale loosely coupled distributed system the need for autonomy is even greater, with collaboration between parties who are competitors, and might well wish to block one another's services if they could! Thus we can't rely on ACID transactions as the sole structuring technique for collaborating business processes. No other technique is known that provides the same level of ease for the application developer, to completely ignore failure handling and concurrency. Thus the most we can aim for is to allow system developers to make business-specific decisions about what particular consistency conditions are important, and then they would design interactions between workflows that provide this level of consistency and also adequate performance.

The current mechanisms for describing collaborating business processes are very far even from this reduced goal. They provide at best ad hoc support for particular coordination mechanisms, but they do not guide the developer on how to choose the appropriate mechanisms for the application domain, or on how to write the processing to deal with failures or interleaving. It is our hope to remedy this. We want to show that one can provide infrastructure that allows the developer to define consistency for their situation, and allows them to write a system as a matter of routine such that consistency is maintained. The infrastructure we want to offer includes

- a language to express consistency conditions,
- tools to check when the system maintains consistency,
- guidance in using the mechanisms properly.

The first stage of our research has been the informal expression of consistency conditions that arise in realistic systems, as illustrated in this paper. The next goal is to have a precise framework in which the conditions can be expressed. We see three separate levels at which we need to work: the contract, the interface, and the orchestration.

The contract should describe the business requirements for global consistency between collaborating parties. In order that the conditions should be useful in practice, we feel that the conditions should be in a declarative style, so they can be considered individually, and without implementation detail biasing the design.

The description of the externally visible pattern of interaction of a business process is given in the interface. We feel that this interface should be enriched, with extra information which goes beyond the syntax of invocation (as in WSDL) and beyond the sequence of interactions (as in WSCL), to address the failure atomicity properties, the isolation properties, and the performance properties (especially timeliness). Initial ideas for this were presented in (Kuo, Fekete, Greenfield & Jang 2002). For example, for failure atomicity properties, we feel that each process needs to identify situations that lead to termination with success, those that lead to “acceptable failure” (for example, an order that was cancelled before any goods were shipped, or one that was cancelled after shipment but the goods were returned and the required cancellation fees have been paid), and others that are unacceptable (goods have been shipped but not accepted at the destination, with no return arrangement made).

Finally, the implementation details of a business process will need to take existing orchestration languages and incorporate additional mechanisms by which consistency can be maintained.

With a declarative representation for the consistency conditions at each level, there is scope for “formal methods” techniques which can reason about these descriptions. This reasoning would ideally be supported by automated tools. For example, one would like to check that two processes, described by particular interfaces, are compatible with each other in composing according to a particular contract. This requires that each will respond appropriately to meet the expectations of their partner, with regard to the state of each party in the interaction. For example, a customer who is unwilling to pay a cancellation fee when an order is cancelled before delivery, can’t interact correctly with a merchant that requires a cancellation fee once the shipment has been sent. An extension of this could be incorporated into resource discovery services like UDDI, so that a party can seek a service which will collaborate correctly in a given contract.

We also envisage runtime detection of inconsistency, so that workflow engines can report when particular consistency conditions fail. This would be a great improvement on current infrastructure, which continues to run silently as the data gets more and more corrupted. We would also like to extend existing workflow tools so that they help the designer ensure that the workflow they build has properties expressed in the interface. For example, the tool might indicate where different sorts of cancellation messages must be awaited, and where particular coordination protocols should be employed.

A different research direction would be to consider how systems can be composed with existing technology, that is, without infrastructure support for declarative failure, isolation or timeliness properties. Without such support, the designers will need standard patterns that interact properly.

## 5 Conclusions

In this talk, we have explained some new technologies for building loosely coupled distributed systems, that are currently attracting a lot of attention in industry. These systems execute through a coordinated pattern of interaction between pre-existing applications in multiple organisations. The interoperability of these applications is made convenient through the wide acceptance of “Web Services” standards such as SOAP and WSDL, and this should improve further as the dust settles between competing standards for higher level aspects such as business process description.

While it is easy to connect the pieces to make a large-scale system, it is very hard to do so in ways which will keep the data and process state aligned properly between separately written and autonomously managed applications that collaborate while keeping per-collaboration state in each party. We have identified several of the difficult issues in defining consistency for particular applications, based on a realistic e-procurement example.

We’ve shown some of the ideas from the database community, such as compensation, which have influenced the existing standards. We demonstrated deficiencies in the existing work, and we have outlined a research program to address these. There is an exciting opportunity for database researchers to investigate these issues and make a major contribution to the state-of-practice of commercial software design.

## References

- Alonso, G., Agrawal, D., El Abbadi, A., Kamath, M., Gunthor, R. & Mohan, C. (1996), Advanced Transaction Models in Workflow Contexts, in ‘IEEE International Conference on Data Engineering’ pp. 574–581.
- Bennett, B., Hahm, B., Leff, A., Mikalsen, T., Rasmus, K., Rayfield, J. & Rouvellou, I. (2000), A distributed object-oriented framework to offer transactional support for long running business processes, in ‘IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)’ pp. 331–348.
- Freemantle, P., Weerawarana, S. & Khalaf, R. (2002), ‘Enterprise Services’ *Communications of the ACM* **45**(10), 77–82.
- Garcia-Molina, H. & Salem, K. (1987), Sagas, in ‘ACM International Conference on Management of Data (SIGMOD)’ pp. 249–259.
- Kamath, M. & Ramamritham, K. (1996), ‘Correctness Issues in Workflow Management’ *Distributed Systems Engineering* **3**(4), 213–221.
- Kamath, M. & Ramamritham, K. (1998), Failure Handling and Coordinated Execution of Concurrent Workflows, in ‘IEEE International Conference on Data Engineering’ pp. 334–341.
- Kuo, D., Fekete, A., Greenfield, P. & Jang, J. (2002), Towards a framework for capturing transactional requirements of real workflows, in ‘Second International Workshop on Cooperative Internet Computing’, Hong Kong, pp. 113–122.
- Wachter, H. & Reuter, A. (1992), The ConTract Model, in ‘Database Transaction Models for Advanced Applications’ (edited by A. Elmagarmid), pp. 219–263. Reprinted in ‘Readings in Database Systems, 3rd ed’ (edited by M. Stonebraker and J. Hellerstein).
- Worah, D. & Sheth, A. (1997), Transactions in Transactional Workflows, in ‘Advanced Transaction Models and Architectures’ (edited by S. Jajodia and L. Kerschberg), pp. 3–34.