

# Towards Building Flexible and Reliable Electronic Commerce Systems: A Distributed Transactional Activity Approach

Tong Zhou   Ling Liu   Calton Pu  
Department of Computer Science & Engineering  
Oregon Graduate Institute  
P.O.Box 91000  
Portland, OR 97291-1000, USA  
{tzhou,lingliu,calton}@cse.ogi.edu

## Abstract

In electronic commerce over the Internet, business transactions are frequently long-duration activities, exhibit complex interaction dependencies, and are prone to failures or unpredictable situations. As an adaptive engineering approach to developing flexible and reliable electronic commerce systems, we present the distributed transactional activity model (TAM) in this paper and report our experience in designing and implementing TAM. A distinct feature of this approach is the utilization of a family of dynamic activity restructuring operations that allow ongoing transactional activities to bypass some unexpected execution bottlenecks, and the notion of validity that provides correctness guarantees for the activity restructuring operations. Another advantage of TAM is its facilities that help business process designers to deal with business structure changes adaptively and incrementally. Our implementation method is based on component technology and using plug-in adaptors on top of commercial OLTP systems. We argue that TAM, its dynamic activity restructuring operations, and our implementation method together provide a viable and practical foundation towards effectively building and efficiently managing electronic commerce applications.

## 1 Introduction

Electronic commerce (EC) has remarkably reshaped today's business practices. Businesses are taking advantage of the unique characteristics of online trading (e.g., reduced costs in supply distribution and goods delivery,

decreased length of product development cycles, flexible transaction values, etc.) to create modern business models and service offerings. At the same time, the increased flexibility and potential appeals of online business practices have put forward the need for design and development of flexible and reliable EC systems and toolkits. In our view, a flexible and reliable electronic commerce system should provide supporting mechanisms for distributed commerce transactions that not only ensure timing deadlines and direct trusts but also address consistency in concurrent data sharing, resilience to unexpected failures, and efficiency guarantee in the presence of unpredictable situations. More concretely, a modern electronic commerce system needs to address questions such as:

- *How can an EC system provide consistency guarantees that are resilient to unexpected failures and concurrent data sharing?*

In traditional business settings, many business processes are handled by humans based on papers. Traditional transactions carried out by online transaction processing (OLTP) systems only represent a small portion of an entire business process. These classical transactions are typically short in duration, and involves a relatively small number of well-defined operations on corporate data. In contrast, a typical electronic commerce application often spans an entire business practice (e.g., from the procurement of supplies to the delivery of goods to consumers), and involves frequent human and computer interactions. The distinct characteristics of commerce transactions include long-durations, complex interaction dependencies, and vulnerability to failures or unpredictable situations, such as network congestion, connection delays, or server breakdowns. Thus, one of the big challenges for building flexible and yet reliable EC systems is to provide system-automated mechanisms to shield the end-users from any explicit handling of recovery and concurrent task synchronization in the presence of unexpected events or failures.

- *How can a modern EC system assist business process designers in building flexible, reliable, and adaptive business models?*

Electronic commerce has made it possible for new business models to be deployed and new features to be added into existing business practices, introducing greater flexibility into the enterprise business infrastructure than ever before. For example, an online merchant may want to provide both traditional payment schemes (e.g., by credit card) and

various new payment schemes (e.g., digital cash). Similarly, the merchant may want to support multiple payment-delivery relationships, such as payment before delivery (e.g., for ordering a computer) or multiple payment/delivery sequence (e.g., for playing online games). Such dynamic nature of electronic commerce applications adds extra complexity to design and development along two dimensions: On one hand, it is often difficult to foresee all the functionalities or services required and to be required, and thus difficult to build the most effective business structure once and for all. This is especially true in an open information universe, such as the Internet and the World Wide Web (WWW). On the other hand, the highly distributed and interactive nature of electronic commerce applications make them much more vulnerable to resource availability constraints and consistency problems due to failures. For example, typical network congestion, server overload, user intervention or absence, and other exceptions may cause temporary or permanent bottlenecks to any ongoing execution of EC processes. Such unexpected situations may end up affecting the entire business workflow process and its deadline requirements. Thus, it is desirable for a modern EC system to provide flexibility and adaptability to help the potentially delayed business processes to bypass system bottlenecks, for example, by dynamically restructuring the affected execution path at run time.

With these questions in mind, we develop the distributed Transactional Activity Model (TAM) as an extension to the Activity Model [24] for electronic commerce applications. We have incorporated and implemented a family of dynamic activity restructuring operations [26], in the first prototype of TAM. TAM provides simple and effective facilities to allow business process designers to specify the behavioral composition of complex activities and a wide variety of activity interaction dependencies in a high-level and declarative way. Its dynamic activity restructuring operations help process designers to deal with business structure changes adaptively and incrementally. TAM also introduces a formal notion of validity which not only allows the restructuring of ongoing activities to let them bypass execution bottlenecks, but also provides correctness guarantees to the involved activities during such restructuring process. In this paper, we report our experience in designing and implementing TAM as an adaptive engineering approach to developing flexible and reliable EC systems. Our implementation method is based on component technology and uses plug-in adaptors [3, 40]. The ini-

tial implementation can be seen as an extension built on top of the standard OLTP reference architectures [20]. This implementation design allows us to easily port the prototype system to existing TP environments to support enriched functionalities that are particularly useful for building flexible and reliable electronic commerce applications.

The rest of the paper is organized as follows. In Section 2, we outline the Transactional Activity Model and walk through the key concepts with an electronic commerce case study. In Section 3, we describe the functionalities of the dynamic activity restructuring operations and illustrate their use through the running case study. We present an overview of the design and implementation of the TAM prototype system, especially the dynamic activity restructuring operations in Section 4. We review the related work in Section 5, and concludes the paper in Section 6.

## 2 Transactional Activity Model for Electronic Commerce

In this section, we describe the Transactional Activity Model (TAM) [24, 25] to address the transactional and flexibility requirements in electronic commerce applications. We illustrate TAM's features by presenting a case study, which will be used in the rest of this paper.

### 2.1 The Online Shopping Case Study

Our case study focuses on a typical scenario in electronic commerce: shopping over the Internet. Consider an **E-Shopping** activity that captures an online shopping process involving a web-surfer (customer) and an online merchant (see Figure 1). In a simplified case (with respect to a real process), this activity might consist of four subactivities: **TermAgreement**, **Offer**, **Purchase**, and **Delivery**. The activity **E-Shopping** starts when the customer visits the online merchant's web site, and have entered his/her login id to the merchant's system. The activity **TermAgreement** is first launched, in which the merchant outlines some terms and options (e.g., in the form of an HTML page) that apply to the potential trade(s), and the customer either accepts or rejects. The terms/options might include such items like whether the customer pays before selected goods are delivered, whether the invoices are digitally signed, whether receipts are provided, and so on. In case the customer agrees to the terms set forth by the merchant, the

**Offer** activity follows. In this activity, information about certain goods are first furnished by the merchant, like the price information; the customer then makes a selection. After the customer has decided on the goods he/she would like to purchase, the entire **E-Shopping** process proceeds on to the next phase. Depending on the terms both parties have agreed in the **TermAgreement** activity, there are different possibilities for how the activities **Purchase** and **Delivery** are carried out. The activity **Purchase** has a number of subtasks, such as: (1) **PaymentMethods**: in which all the necessary payment methods are determined, e.g., whether some coupons or vouchers or frequent-flyer points can be used/redeemed against the purchase, or whether the customer uses credit card or digital cash to pay; (2) **Payment**: the actual payment process in which value is transferred from the customer to the merchant; and optionally (3) **PaymentReceipt**: in which a receipt is generated by the merchant and given to the customer.

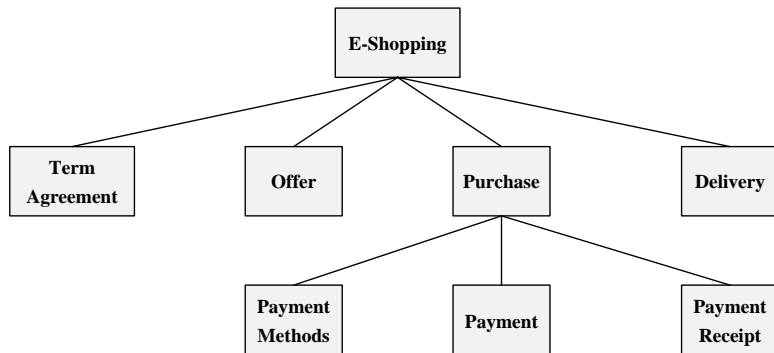


Figure 1: The online shopping activity

## 2.2 Activities, Activity Patterns, and Activity Composition Hierarchy

Activities are structured programs that exchange information with other activities, databases, files, and users. In the Transactional Activity Model (TAM), activities are represented by parameterized *activity patterns* (or *activity specifications*). Informally, an activity pattern consists of objects, messages, message exchange constraints, preconditions, postconditions, and

triggering conditions [24]. Activity patterns capture the structure, flow, interactions, and transactional requirements within business activities. An *activity instance* is an instantiation of its activity pattern or specification. In the rest of the paper, the terms *activity* and *activity pattern* will be used interchangeably when no conflict arises.

TAM distinguishes between two types of activities: *elementary activities* and *composite activities*. An elementary activity is a program that consists of database *Read* and *Write* operations, as well as transaction operations such as *Begin*, *Commit*, or *Abort* [25]. A composite activity consists of a tree of composite or elementary activities, and a set of user-defined activity dependencies. The tree organization of an activity  $\alpha$  is called the *activity composition hierarchy* (or activity hierarchy) of  $\alpha$ . Elementary activities correspond to the leaf nodes in an activity hierarchy. The activities that are components of a composite activity are sometimes referred to as *constituent activities* or *subactivities*; and the set of activity dependencies associated with a composite activity can be seen as the cooperation contracts among its subactivities that collaborate to accomplish a complex workflow task. In our online shopping case study, **Offer** and **Payment** are two example elementary activities; **Purchase** is a composite activity, while being a constituent activity of **E-Shopping**. **E-Shopping** is the top-level activity. A running instance of this activity pattern (or specification) is an **E-Shopping** instance.

Figure 2 and Figure 3 present the TAM activity specifications for our online shopping example. First, each activity is described by specifying the list of children activities and their interaction dependencies through the behavioral aggregation facility, thus all the composite activities are defined in a declarative and incremental fashion. Second, users may add additional activity dependencies that are domain-dependent to the activity specification. TAM classifies application-specific interaction dependencies into different categories, like activity execution dependencies, activity interleaving dependencies, activity state transition dependencies, and activity visibility dependencies.

## 2.3 Activity State Transition Dependencies

An activity has a set of observable states  $\mathcal{S}$  and a set of possible state transitions  $\psi: \mathcal{S} \rightarrow \mathcal{S}$ , where  $\mathcal{S} = \{begin, commit, abort, done, compensate\}$ . The transitions between the states of an activity can be represented by a finite state automation and its transition graph. Figure 4 shows the state

```

begin Activity E-SHOPPING(In: CustomerId:CUSTOMER, Start:TIME, Out: End:TIME)

  Behavioral Aggregation of Component Activities:
  TERMAGREEMENT(In: CustomerId:CUSTOMER, Out: Accepted:BOOLEAN, Term:TERM)
  OFFER(In: CustomerId:CUSTOMER, Out: Accepted:BOOLEAN, Offer:OFFER);
  PURCHASE(In: CustomerId:CUSTOMER, Offer:OFFER, Out: Receipt:RECEIPT);
  DELIVERY(In: CustomerId:CUSTOMER, Receipt:RECEIPT);

  Execution Dependencies:
  ExecR1: TERMAGREEMENT precede OFFER
  ExecR2: OFFER precede {PURCHASE, DELIVERY}

  State Transition Dependencies:
  STR1: {abort(PURCHASE)∨abort(TERMAGREEMENT)∨abort(OFFER)} enable abort(self)

end Activity

```

**Figure 2:** An example specification of composite activity E-Shopping

```

begin Activity PURCHASE(In: CustomerId:CUSTOMER, Offer:OFFER, Out: Receipt:RECEIPT)

  Behavioral Aggregation of Component Activities:
  PAYMENTMETHODS(In: CustomerId:CUSTOMER, Offer:OFFER, Out: Methods:METHODS)
  PAYMENT(In: CustomerId:CUSTOMER, Methods:METHODS)
  PAYMENTRECEIPT(In: CustomerId:CUSTOMER, Offer:OFFER, Methods:METHODS, Out: Receipt:RECEIPT)

  Execution Dependencies:
  ExecR1: PAYMENTMETHODS precede PAYMENT
  ExecR2: PAYMENT precede PAYMENTRECEIPT

  Interleaving Dependencies:
  ILLR1: PAYMENT precede DELIVERY

  State Transition Dependencies:
  STR1: abort(PAYMENT) enable abort(self)

end Activity

```

**Figure 3:** An example specification of composite activity Purchase

transition graph of a TAM activity. When an activity  $T$  is activated, it enters the state *begin* and becomes active. The state of  $T$  is changed from *begin* to *commit* if  $T$  commits, and to *abort* if  $T$  or its parent aborts. If  $T$ 's parent aborts after  $T$  is committed, then its state is changed to *compensate*; if  $T$ 's root activity commits, then its state becomes *done*.

When  $T$  is a composite activity,  $T$  enters the *commit* state if all its component activities  $A_i$  ( $i = 1, \dots, n$ ) legally terminate (i.e., commit or abort). In other words, the activity  $T$  may commit even if one of its component activities is aborted. Similar to the saga transactions [17, 9], the *commit* of a subactivity in TAM is independent of the *commit* of its parent activ-

ity. If an activity aborts, then all its children that are in *begin* state are aborted; and its *committed* children, however, are compensated for. We call this property *termination-sensitive* dependency between an activity  $A_C$  and its parent activity  $A_P$ , denoted by  $A_P \rightsquigarrow A_C$ . This termination-sensitive dependency, inherent by an activity hierarchy, prohibits a child activity instance from having more than one parent, ensuring the hierarchically nested structure of active activities. When the abort of all *active* subactivities of an activity is completed, the compensation for *committed* subactivities is performed by executing the corresponding compensations in an order that is the reverse of the original order.

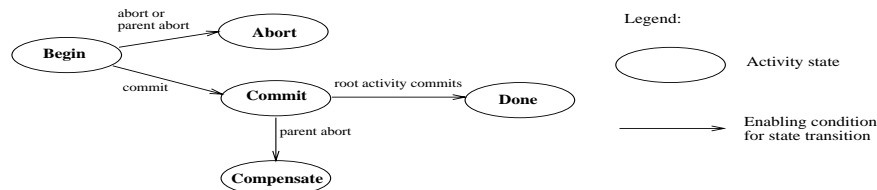


Figure 4: Activity state transition graph

The activity state transition dependencies discussed in this section are implicitly defined for an activity through its activity composition hierarchy. In TAM we provide a number of constructs (see Section 2.5) to allow users to add additional state transition dependencies that are application-specific. For example, in our case study, an additional state transition dependency is defined for activity **Purchase** (see Figure 3). It requires that the **Payment** subactivity’s outcome be critical to **Purchase**. If **Payment** is *aborted*, then the **Purchase** activity has to be *aborted* as well. This dependency is in addition to the implicit dependencies defined with the hierarchical relationship between **Purchase** and **Payment**.

## 2.4 Activity Visibility Dependencies

The visibility of an activity refers to the ability of an activity to see the results of other activities while it is executing. In TAM, the visibility of an activity is defined in terms of the object access-set of its parent activity, which forms a visibility dependency between the given activity and its parent activity.

TAM uses the multiple object version scheme [29] to define the notion of visibility in the presence of concurrent execution of activities, that is, object versions are created and maintained at different points in the activity hierarchy. Each activity has a local set of object versions, which is accessible to any of its children subactivities in the hierarchy. Thus, there may be several versions of an object scattered throughout the activity hierarchy. For a specific object, the version at the *Root* activity is the oldest version, and the ones further down in the hierarchy are causally dependent on it. With multiple object version scheme, a notion of visibility is defined naturally according to the hierarchy. For a given activity  $T$ , when an object in  $T$ ’s

data access-set receives an access request from a child subactivity of  $T$ , a new version is automatically created for this subactivity. The new version of the object is written back to the parent activity’s access-set when the subactivity that initiated the access to the object commits or comes to a breakpoint. A *breakpoint* [5, 16] of an activity is a point in its execution where it can interleave with other activities. With the multiple object version scheme, the root activity of an activity hierarchy contains the most stable version of each object, and guarantees that it can recover its copies of objects in the event of system failure.

The rule governing the visibility of object versions between a composite activity  $A_P$  and its child subactivity  $A_C$  is described as follows:

- **Visibility:**  $A_C$  has access to all objects that  $A_P$  can access, i.e., it can read objects that  $A_P$  has modified.

Important to note is that both default activity state transition dependencies and activity visibility dependencies are implicitly defined by the activity composition hierarchy. They can be modified either by additional programmer-specified activity dependencies (e.g., the state transition dependencies in Figure 3), or through programmer-guided or system-triggered activity restructuring operations (i.e., Activity-split and Activity-join) to be discussed in Section 3.

## 2.5 Basic Constructs for Specification of Activity Dependencies

In TAM application-specific activity dependencies are user-defined and are associated with composite activity patterns. They capture complex interaction dependencies between activities that have no hierarchical composition relationship with each other. Examples include constraints on the occurrence of a subactivity execution and the temporal precedence of execution of subactivities at the same abstraction level and across different levels in the activity composition hierarchy [25].

TAM provides four constructs for specification of user-defined activity dependencies. They are **precede**, **enable**, **disable**, and **compatible**. Figure 2 and Figure 3 have already illustrated some use of these constructs. The construct **precede** is designed to capture the temporary precedence dependencies and the existence dependencies between two activities. For example, “**Purchase precede Delivery**” specifies a *begin-on-commit* execution dependency between the two activities: “**Delivery** can not begin

before **Purchase** *commits*". The constructs **enable** and **disable** are utilized to specify the enabling and disabling dependencies between activities. One of the critical differences between these two constructs and the construct **precede** is that **enable** or **disable** specifies a triggering condition and an action being triggered, whereas **precede** only specifies an execution precedence dependency as a precondition that needs to be verified before an action can be activated — it is not an enabling condition that, once satisfied, triggers the action. The construct **compatible** declares the compatibility of two concurrently running activities that possibly have shared data. It is provided solely for specification convenience, since two TAM activities are compatible when there is no execution precedence dependency between them. For reference convenience, the semantics of each construct is formally described in Figure 5. Note that the semantics given in Figure 5 shows that **enable** and **disable** are not derivable from each other.

In specifying activity dependencies, one important note is that the activity composition hierarchies in TAM have already implicitly captured the *transactional dependencies* among its constituent activities that are caused by concurrent data sharing and failure recovery. For example, in above online shopping case, the business process might have transactional requirements like: if the **Purchase** does not succeed, the complete shopping activity **E-Shopping** has to be aborted. Or, the **Payment** activity can commit even if the receipt is not properly produced in **PaymentReceipt**. TAM's special treatment for transactional dependencies compared with other normal execution dependencies relieves process designers from explicitly specifying the actions that deal with concurrent synchronization and failure recovery, which are complex and might not be familiar to the designers.

Another feature of TAM is that the specification scope of the set of activity dependencies associated with each activity pattern is regulated to encourage incremental specifications of hierarchically complex activities. For instance, in the online shopping example, to define the activity **Purchase**, the activity designer needs to focus on (specify) only the execution dependencies between its children activities (i.e., **PaymentMethods**, **Payment**, and **PaymentReceipt**), and restrict the activity interleaving dependencies to be only the interaction dependencies between its children activities or between its children activities and its siblings' children activities (e.g., **Payment precede Delivery**). Furthermore, although an activity designer may not override the termination-sensitive state transition dependencies (recall Section 2.3) between a composite activity and its children

activities, they can add additional application-specific state transition dependencies, such as making the commit of a child activity to be critical to the commit of its parent activity (see Figure 3).

### 3 Dynamic Restructuring Operations for EC applications

Businesses are taking advantage of the unique characteristics of online trading (e.g., the reduction in supply distribution or goods delivery cost, flexible transaction values, etc.) to differentiate their products and service offerings in the market. What these mean, however, is extra requirements on the systems and tools that support these flexible business practices. For example, the transaction system might be required to support multiple payment schemes, like credit card or digital cash. Or, the merchant software might be required to support multiple pay-deliver relationships, like payment before delivery (e.g., ordering a computer) or multiple payment/delivery sequence (e.g., playing online games). Beyond these, the fact that it's usually difficult to foresee all the functionalities or services (and the most effective business structure) makes the requirements on system support even more demanding. On the other hand, during their execution, the distributed and highly-interactive nature of these electronic commerce applications make them vulnerable to resource availability constraints and consistency problems due to failures.

In order to support the dynamic business structures as well as to improve their execution performance in anticipation of various uncertainties, we have defined, on top of TAM, two groups of operations: *Activity-Split* operations and *Activity-Join* operations for dynamic restructuring of ongoing activities [26]. These operations can help an EC system bypass temporary or permanent bottlenecks in both data flow and control flow in commerce processes, like those caused by node/network unavailability, user intervention or absence, and other exceptions. Furthermore, These activity restructuring operations build on a formal notion of validity so that they guarantee a rich set of correctness criteria (that extend serializability) of the resulting activities.

Below, we explain these restructuring operations and illustrate their use with our case study example.

Construct	Usage	Synopsis
<b>precede</b>	$A_1$ <b>precede</b> $A_2$ condition( $A_1$ ) <b>precede</b> $A_2$ condition( $A_1$ ) <b>precede</b> condition( $A_2$ )	$A_2$ can <i>begin</i> if $A_1$ <i>commits</i> $A_2$ can <i>begin</i> if <i>condition</i> ( $A_1$ )='true' holds. If <i>condition</i> ( $A_1$ )='true' then <i>condition</i> ( $A_2$ ) can be 'true'
<b>enable</b>	condition( $A_1$ ) <b>enable</b> $A_2$ condition( $A_1$ ) <b>enable</b> condition( $A_2$ )	<i>condition</i> ( $A_1$ )='true' $\rightarrow$ <i>begin</i> ( $A_2$ ) If <i>condition</i> ( $A_1$ )='true' then <i>condition</i> ( $A_2$ ) will be 'true'
<b>disable</b>	condition( $A_1$ ) <b>disable</b> $A_2$ condition( $A_1$ ) <b>disable</b> condition( $A_2$ )	<i>condition</i> ( $A_1$ )='true' $\rightarrow$ <i>abort</i> ( $A_2$ ) If <i>condition</i> ( $A_1$ )='true' then <i>condition</i> ( $A_2$ ) cannot be 'true'
<b>compatible</b>	<b>compatible</b> ( $A_1, A_2$ )	<i>true</i> if $A_1$ and $A_2$ can be executed in parallel, and <i>false</i> if the order of $A_1$ and $A_2$ is important

Figure 5: Constructs for activity dependency specification in TAM

### 3.1 Activity-Split Operations

For our case study, consider this scenario during an instance **E-Shopping** activity’s execution: suppose the customer entered an online video store, accepted the terms and agreements, picked two videos that he/she likes to watch online (e.g., two episodes of a comedy series), and has just paid using his/her smart-card. Now the video store is about to live-feed the video/audio stream to the customer. However, the store (hopefully the system, but could be human as well) detects a server overload problem at this moment (or predicts that an overload would occur soon, based on recent customer payment activities). In order to alleviate the problem, the system (or a human agent) suggests to the customer to overlap his/her delivery with other customers’ and use multiple deliveries to fulfill the store’s delivery obligation. After the user has agreed to this change, the **Delivery** activity is split into two activities **Delivery1** and **Delivery2**, each responsible for delivering one episode of the video the customer ordered. An additional activity dependency is enforced on these two new activities that if the **Delivery** activity commits, both of them must commit/succeed. This scenario is shown in Figure 6.

A two-way *Activity-Split* operation on an ongoing activity instance  $C$ , denoted by  $ASplit(C, \varphi, C_1, C_2)$ , produces two new activity instances  $C_1$  and  $C_2$ <sup>1</sup>. The synchronization constraint  $\varphi$  is a function of  $(C_1, C_2)$  that captures the dependencies between them after the split. For example, suppose  $C_1$  and  $C_2$  are independent of each other, we refer to such type of Activity-Split operations as *parallel Activity-Split*, denoted by  $ASplit(C, p, C_1, C_2)$ .

<sup>1</sup>It is possible to reuse the name of the original activity  $C$  to name one of the resulting activities.

N-way Activity-Split can be easily constructed using the two-way Activity-Split [26]; and for presentation clarity, we restrict our discussion to two-way Activity-Split in this paper. We say that *ASplit* is a *meta* operation, since the definition and implementation of *ASplit* are specialized for each synchronization constraint. Figure 7 presents a group of instance Activity-Split primitives that are currently supported in TAM, along with their inherent semantics. Each of these specialized primitives represents one typical case of Activity-Split.

An Activity-Split operation is *valid* if and only if the resulting activities are: (1) TAM activities, i.e., satisfying the implicit activity dependencies implied in the activity composition hierarchy, such as the transactional dependency; (2) satisfying all the existing user-defined activity dependencies, and (3) they do not introduce any conflicting activity dependencies. In TAM, we consider all valid Activity-Splits as legal operations and all invalid Activity-Splits as illegal operations.

Activity-split is useful in several ways. For example, it supports adaptive recovery from permanent or temporary unavailability of nodes, users, or constituent activities. Another benefit is added concurrency, by releasing early-committed resources through a split activity to allow accesses by other concurrent activities.

### 3.2 Activity-join Operations

Following the above scenario, in order to make sure the customer is not dissatisfied, the store (again, either the system or an on-duty clerk) follows up the previous change of delivery method with an email right before the

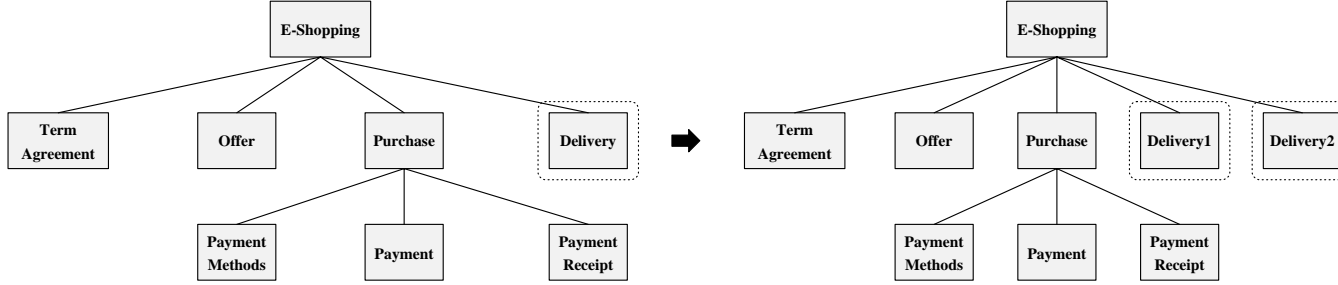


Figure 6: An example application using Activity-Split

Usage	Synchronization Constraint	Synopsis
$ASplit(C, p, C_1, C_2)$	$compatible(C_1, C_2)$	parallel activity-split operation ( <i>p-Split</i> )
$ASplit(C, s, C_1, C_2)$	$C_1 \text{ precede } C_2$	serial activity-split operation ( <i>s-Split</i> )
$pa-Split(C, C_1, C_2)$	$commit(C_1) \text{ disable } C_2$	parallel-alternative activity-split operation $C_1, C_2$ may begin together and abort $C_2$ if $C_1$ commits
$sa-Split(C, C_1, C_2)$	$begin(C_1) \text{ precede } begin(C_2)$ $\wedge abort(C_1) \text{ enable } C_2$	serial-alternative activity-split operation $C_2$ cannot begin unless $C_1$ aborts
$cc-Split(C, C_1, C_2)$	$commit(C_1) \text{ precede } commit(C_2)$	commit-on-commit activity-split operation $C_2$ cannot commit unless $C_1$ commits
$ca-Split(C, C_1, C_2)$	$abort(C_1) \text{ precede } commit(C_2)$	commit-on-abort activity-split operation $C_2$ cannot commit unless $C_1$ aborts
$ASplit(C, u)$	none	activity-split by <i>unnesting</i> ( <i>u-Split</i> ) if $Children(C) \neq \emptyset$ , modify $Children(Parent(C))$ by $Children(Parent(C)) \cup Children(C)$ , remove $C$

Figure 7: Instance Activity-Split primitives in TAM

delivery is about to finish or right after that, apologizing for the inconvenience and asking for the customer’s feedback. The email might state that in case the customer is not quite satisfied, the store could offer some compensation (like in forms of certain credit points) to the customer. This email initiates a new activity **DeliveryFeedback**. This new activity is *joined* with the previous two **Delivery** activities to become a new subactivity of **E-Shopping**. The activity might last until the store hears back from the customer. In case the customer is not satisfied, the store’s system might have to *compensate* the **Purchase** activity with certain actions. These kinds of activity dependencies (more transaction-related) are added

into **E-Shopping**’s specification. The scenario is illustrated in Figure 8.

The inverse operation of Activity-Split, called *Activity-Join*, combines results of several ongoing subactivities together and releases them atomically, as if they had been a single activity. Activity-Join is also a meta operation, with two specialized versions. The first type of Activity-Join operation is called “*join-by-group*”, denoted by  $AJoin(C, g, C_1, C_2)$ . It groups two activities  $C_1$  and  $C_2$  by creating a new activity  $C$  as their parent activity, while preserving the activity composition hierarchy of each input activity. It is also called synchronous join. An Activity-Join by grouping is considered *valid*, if the two input activities  $C_1$  and  $C_2$  are sibling activities or inde-

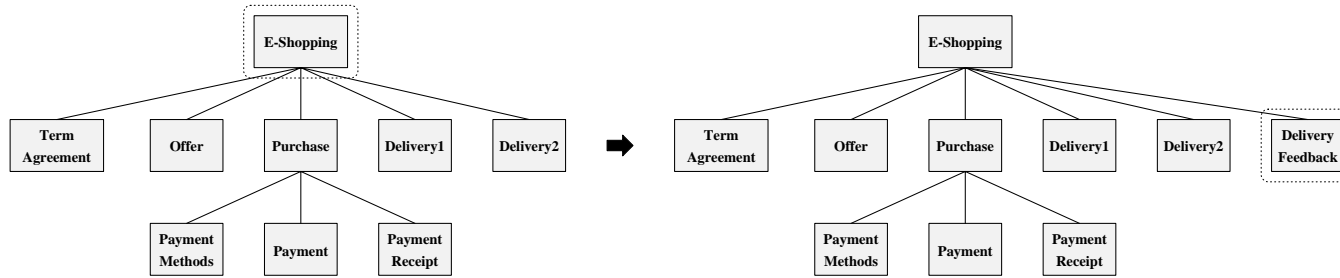


Figure 8: An example application using Activity-Join

pendently ongoing activities. The second type of Activity-Join operations is referred to as “*join-by-merge*”, denoted by  $AJoin(C, m, C_1, C_2)$ . It physically converges two activities in progress into a single activity, and also called asynchronous join. An Activity-Join by merge is considered *valid*, if the two input activities  $C_1$  and  $C_2$  are sibling activities, or one is a parent activity of another, or two independently ongoing activities, i.e., they do not have common parent activity.

For grouping or merging activities that are across different abstraction levels, a combination of Activity-Split and Activity-Join operations are used. Note that we do not support Activity-Join of arbitrary activities. For example, a call to join activity  $C$  with an activity that is a sibling activity of  $C$ ’s parent activity or  $C$ ’s grandparent activity is not acceptable due to potential inconsistency. Our experience shows that the set of Activity-Join operations we support cover most of the practical requirements for grouping or merging ongoing activities in a cooperative group working environment.

In summary, Activity-Split and Activity-Join can be combined in any formation. By allowing the release of early-committed resources or ownership transfer of uncommitted resources, these dynamic activity restructuring operations bring into complex activities a number of advantages, such as adaptive recovery, added concurrency, dynamic process evolution, and enhanced cooperation, providing a cost-effective framework for organizing and restructuring activities in an electronic commerce environment.

## 4 Implementation

### 4.1 Overview of Implementation Method

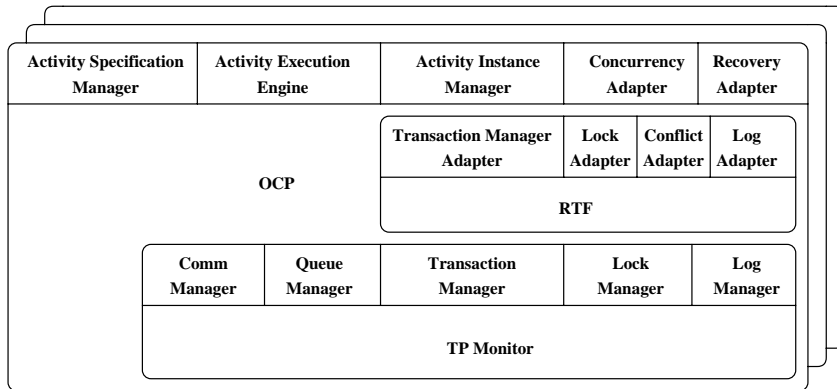
In implementing the distributed Transactional Activity Model (TAM) and the set of activity restructuring operations, we have adopted a carefully designed implementation method. First, following a component-based and open-system approach, we combine microprotocol composition [31, 40], open implementation [23, 40], and incremental specialization [35, 40] in the design of our system, to achieve modular, adaptable, and efficient primitive implementation. Second, we utilize the Reflective Transaction Framework (RTF) [3] and the Open Coordination Protocol (OCP) [40], which we previously developed to support distributed extended transactions on commercial transaction processing (TP) systems, to address the transactional and distribution requirements in workflow activities. Third, we build adaptors on top of RTF, OCP, and commercial TP systems, to make our implementation of activity restructuring operations practically available. In the following paragraphs, we explain the above techniques in more detail.

The first building block of our implementation method is the careful design of system functions (e.g., activity restructuring, recovery handling) to facilitate modular, adaptable, and efficient implementation. We achieve this by combining microprotocols, open implementation, and incremental specialization.

The second building block of our implementation method is the system components developed in our previous effort on supporting distributed ex-

tended transactions [33]. In particular, the Reflective Transaction Framework (RTF) supports different extended transaction models on top of production transaction processing (TP) systems; the Open Coordination Protocol (OCP) is a flexible coordination facility for building different distributed coordination protocols for distributed extended transaction processing. By utilizing and extending these system components, we guarantee the transactional properties of distributed workflow activities even in presence of failures, concurrent data sharing, or dynamic activity restructurings.

The third building block of our implementation method is a set of transactional activity management adapters (*TAM adapters*), which are *add-on* modules to RTF, OCP, or an existing transaction processing (TP) monitor. Each TAM adapter encapsulates and extends the functionalities of its underlying layer. Like the *transaction adapters* in RTF, TAM adapters are based on the commands and functionality of the well-documented TP monitor reference architecture [20] for practicality. Furthermore, our implementation of the TAM adapters relied only on a small but widely supported set of API in the TP monitor reference architecture [20].



**Figure 9:** Transactional Activity Processing Framework

Figure 9 illustrates the relationships among TAM adapters, RTF, OCP, and TP monitors in a distributed setting. Each outer-most bounding frame in the figure represents an entire transactional activity processing component on one distributed node. Within each component (i.e., each frame), three layers are present: TP monitor layer, RTF and OCP layer, and the

TAP layer. OCP and the transaction adapters (e.g., transaction manager adapter, lock adapter, etc.) in RTF utilize and extend the functionality of TP monitors to handle distributed extended transactions. TAM adapters utilize and extend the functionalities of RTF, OCP, and TP monitors to handle dynamic transactional activities. Together, these adapters and the underlying TP facility form a framework for dynamic transactional activity processing.

By applying the above implementation method, each of TAM’s system functions is composed of a suite of microprotocols and invariants/quasi-invariants. The microprotocols utilize RTF, OCP, and the TAM adapters in the distributed transactional activity processing framework. The invariants and quasi-invariants guide the specialization process at primitive instantiation time to build efficient primitive instances. Furthermore, the adapters make our implementation easy to build on top of production TP systems.

## 4.2 System Prototype

We have implemented a prototype system for transactional activity management. The system covers a family of dynamic activity restructuring operations, as well as other important features of the Transactional Activity Model (TAM) [25]. The complete system is client-server oriented. The backend server features a set of transactional activity management primitives, and is built on Transarc’s transaction processing (TP) monitor Encina. The frontend features several web-friendly graphical user interfaces (GUIs) for activity control and administration at both specification and instance levels, and is built with a mixed use of Java applets, JavaScript, and HTML for portability and reusability. Currently, the frontend and backend server communicate via some CGI gateway programs that mediate between HTTP and Encina TRPC (transactional RPC based on DCE-RPC).

Figure 10 is a screen capture showing a running telecommunication activity instance [26] in our prototype system. There are two drawing canvases in the window: the left canvas shows both the activity composition hierarchy of this activity instance and the execution dependencies between its descendant activities; the right canvas, in contrast, merely represents hierarchy and transactional dependency information, but in a canonical way. Only the left canvas is used for restructuring operation specifications, as will be described later in this section. The use of two canvases allows a user to clearly observe what would happen to this activity instance prior to

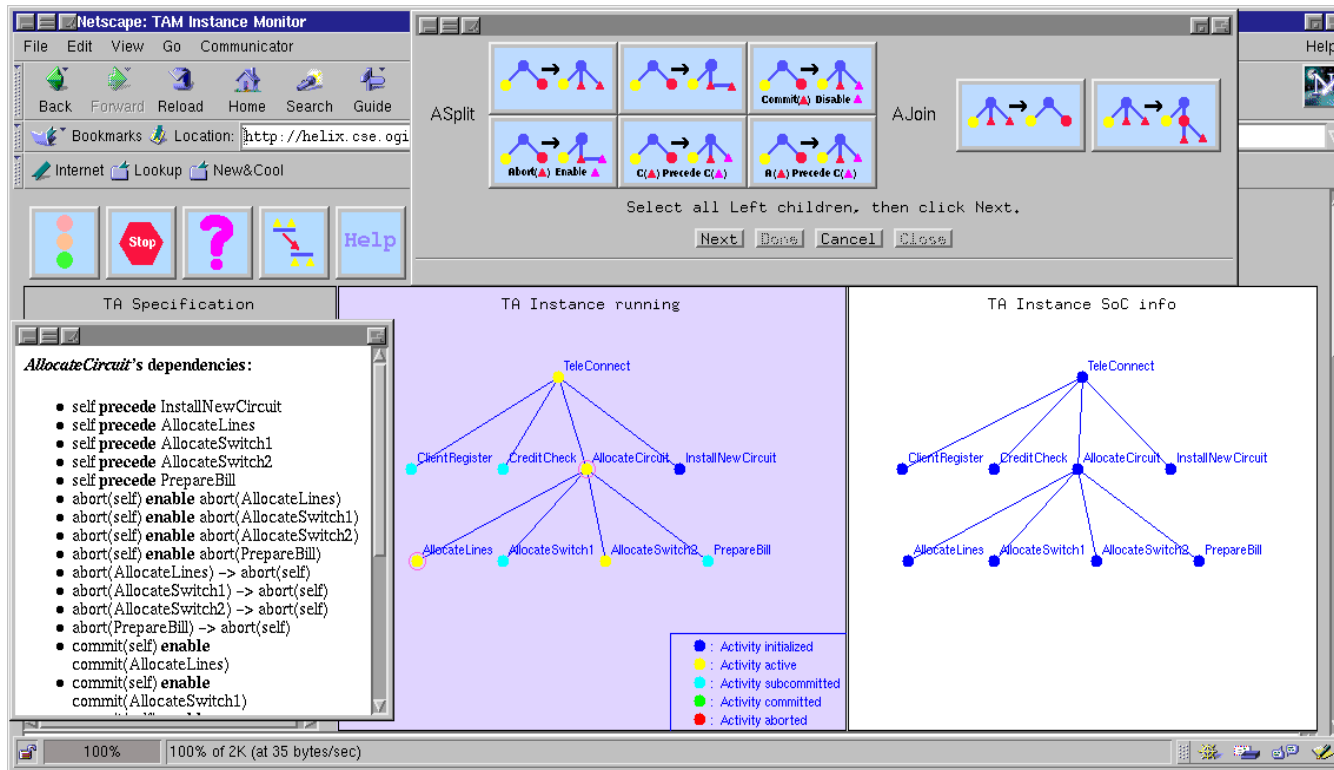


Figure 10: A Screen Capture of Our Prototype System

and after any dynamic restructuring operation. For each node in the left canvas, different colors are used to represent that activity instance's different states: blue indicates **Initiated** state, yellow indicates **Active**, cyan indicates **Subcommitted**, green indicates **Commit**, and red indicates **Abort**. These colors would change in sync with activity instances' state changes during their execution.

The pop-up window at the left-side shows the activity dependency information for those activity or subactivity instances running and monitored by the system. For example, dependencies that guard the transactional outcome for an activity instance, like commit-on-commit, commit-on-abort, etc. These dependencies were discussed in Section 2.

The small pop-up window at the top of Figure 10 is the graphical user interface to activity restructuring operations in our system. In the window, the top portion contains some image buttons, each representing an activity restructuring operation. They are grouped into two categories: Activity-Split and Activity-Join, as described in Section 3. Each operation has its particular transactional semantics to associate with, and would affect activity instances' dependency information differently. The bottom portion navigates the user (administrator) in the context of each selected operation, with a set of text messages and action buttons. A user interacts with the system by a mixed use of mouse actions (click, drag-and-drop, etc.) on the canvases and button clicks.

## 5 Related work

Electronic commerce (EC) is of growing interest to many of today's businesses, like merchants, financial institutions, and technology providers [22, 1, 4]. There have been many coordinated efforts in setting up industry standards for EC, ranging from specifications that address a single step (e.g., payment protocols) within a potentially complex electronic commerce activity [37, 13, 38], to initiatives that try to capture an entire business process (e.g., shopping protocols) [32, 21, 30]. However, these efforts are largely in their infancy, although some have already built pilot programs testing initial products.

Several activity models have been proposed [11, 10, 36] to support declarative specification of control flows within activities. Features of long running activities [11, 10] include an automatic compensation capability and the use of ECA-rules for monitoring activities. The cooperative model [36] achieves cooperation through controlled data exchanges in private workspaces and controlled sharing of a common database among users. Although previous activity models (e.g., [11, 10, 36]) and workflow systems (e.g., [19, 27]) have been successful in capturing the business processes, they have limitations in the prediction of correctness of concurrent activities. For example, the ECA-rules are very powerful tools. Their expressive power on the other hand makes it difficult to explore the application-specific activity execution dependencies that are critical for reasoning about the correctness of concurrent executions of activities, from the generic form of ECA-rules specification. Moreover, very few activity models or workflow systems proposed so far, to our knowledge, provide the adequate support for dynamic split and join of activities of deeply nested structures, ensuring the correctness of resulting activities.

A number of *extended transaction models* (ETMs) [12, 15, 14, 17, 2, 28, 29, 34, 39] have been proposed, each targeting at a particular subset of the whole spectrum of interactions possible in advanced application environments. Although each ETM (such as nested transactions, sagas, split- and join-transactions, cooperative transactions) provides a set of well-defined transactional properties for modeling its target applications, the support for prediction of correctness properties of extended transactions is limited only to those interactions that can conform to the pre-defined transaction structures specific to the chosen ETM. Therefore, an ETM alone is not sufficient for organizing complex cooperative activities that may require the whole spectrum of interactions [8]. Furthermore, most of the ETMs

proposed so far are system supporting built-in ETMs, rather than system supporting user-defined ETMs. The demand for defining new ETMs and redefining existing ETMs as needed becomes increasingly critical towards supporting for a variety of workflow applications with diverse or possibly conflicting business requirements [19, 27].

The TAM development is mostly inspired by the early work on split-transactions [34], Cooperative Transactions [2], and Transaction Groups [29]. Although cooperation among sibling transactions is supported by Cooperative Transactions or transaction groups [2, 34, 29], the interactions among siblings are either limited to static and one-shot design of transaction groups [29, 2], or restricted to leaf node transactions in order to support the isolation-based serializable split-transactions using read-sets and write-sets in a split call [34]. TAM differs from these models by three distinct features: (1) TAM activity specification language allows the users to specify activity composition hierarchy and activity dependencies declaratively and incrementally. Instead of restricting to interaction semantics among siblings, TAM captures interaction dependencies among activities across activity boundaries (e.g., interactions among child activities of different siblings). (2) To the best of our knowledge, no previous activity models have considered the transactional support for activity-split and activity-join operations to allow dynamic restructuring of activities that are hierarchically complex and have sophisticated interaction dependencies at the same time. (3) The third distinct feature is the systematic analysis of the benefits and validity of activity-split and activity-join operations. By specializing the activity restructuring meta operations, the efficient implementation of various split and join operations can also be explored.

Our implementation effort also leverages and builds on our previous experience and mechanisms. Specifically, our transactional activity processing framework builds on RTF [3], OCP [40], and MARS [7], which systematically extend commercial TP monitors to make the implementation of a wide range of distributed extended transaction models practically available [33]. The Reflective Transaction Framework (RTF) is a software framework that systematically extends the functionality of a conventional TP system to implement extended transaction models. The Open Coordination Protocol (OCP) is a flexible coordination facility for systematically building optimized coordination protocols for distributed extended transactions and transactional workflows. The Modular Architecture for Recovery Systems (MARS) is a software architecture for building flexible and efficient recovery systems to support extended transactions. The design of these

foundation components is also related to previous research on using model-independent building blocks to implement different extended transaction models [8, 6, 18].

## 6 Summary

Electronic commerce have definitely brought many businesses into the on-line arena. These new business applications bear distinctive characteristics like being distributed, highly-interactive, and long in duration, subjecting them to both development and execution uncertainties.

In this paper, we have presented the distributed transactional activity model (TAM) and reported our experience in designing and implementing TAM as an adaptive engineering approach to developing flexible and reliable electronic commerce systems. TAM features a set of dynamic activity restructuring operations with formal correctness guarantees, and a set of high-level declarative constructs for incremental activity specifications. The activity restructuring operations allow ongoing EC activities to bypass some unexpected execution bottlenecks, and the specification constructs allow business process designers to deal with changes adaptively. Our implementation method is based on component technology and uses plug-in adaptors on top of commercial online transaction processing (OLTP) systems. We believe that these modeling facilities and system mechanisms together provide a viable and practical foundation for effectively and efficiently supporting reliable EC applications.

We are currently using our prototype system to evaluate the performance of activity restructuring operations and the benefits they bring to business workflow activities. On the theoretical side, we are interested in strengthening our results on the assurance of correctness properties of concurrent activity execution despite the restructuring by Activity-Splits and Activity-Joins.

## References

[1] N. R. Adam and Y. Yesha, editors. *Electronic Commerce: Current Research Issues and Applications*, volume 1028 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

- [2] F. Bancilhon, W. Kim, and H. Korth. A model for CAD Transactions. In *Proceeding of the 11th International Conference on Very Large Databases*, pages 25–33. Morgan Kaufman, 1985.
- [3] R. S. Barga and C. Pu. A practical and modular method to implement extended transaction models. In *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995.
- [4] C. Beam and A. Segev. Automated Negotiation in Electronic Commerce. In *Proceedings of the 3rd International Workshop on Next Generation Information Technology and Systems*, Israel, 1997.
- [5] P. A. Bernstein, J. Rothnie, N. Goodman, and C. Papadimitriou. The concurrency control mechanism of sdd-1: A system for distributed databases (the full redundant case). *IEEE Trans. on Software Engineering*, 4(3), May 1978.
- [6] A. Biliris, S. Dar, N. Gehani, H. Jagadish, and K. Ramamritham. ASSET: A system for supporting extended transactions. In *Proceedings of 1994 ACM SIGMOD*, pages 44–53, May 1994.
- [7] S.-W. Chen. *Recovery for Extended Transaction Models*. PhD thesis, Department of Computer Science, Columbia University, February 1997.
- [8] P. Chrysanthis and K. Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of SIGMOD Conference on Management of Data*, pages 194–203, June 1990.
- [9] P. Chrysanthis and K. Ramamritham. Acta: The saga continues. In *Elmagarmid [15]*, pages 349–397, 1992.
- [10] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of the ACM SIGMOD*, 1991.
- [11] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long-running activities. In *Proceedings of the 17th Very Large Databases*, pages 113–122, 1991.
- [12] A. Deacon, H. Schek, and G. Weikum. Semantic-based multilevel transaction management in federated systems. In *Proceedings of International Conference on Data Engineering*, pages 452–461, 1994.
- [13] DigiCash. <http://www.digicash.com/>.
- [14] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the 16th International Conference on Very Large Data Bases*, 1990.
- [15] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

- [16] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. on Database Systems*, 8(3), June 1983.
- [17] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM SIGMOD Int. Conference on Management of Data*, pages 462–473, 1987.
- [18] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a programmable transaction environment. In *Proceedings of the 1994 IEEE Conference on Data Engineering*, pages 462–473, February 1994.
- [19] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, (2):119–153, 1995.
- [20] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [21] Joint Electronic Payment Initiative (JEPI). <http://www.w3.org/ECCommerce/specs/>.
- [22] R. Kalakota and A. B. Whinston. *Frontiers of Electronic Commerce*. Addison-Wesley, 1996.
- [23] G. Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA '92 Workshop on Reflection and Meta-level Architectures*, 1992. See <http://www.xerox.com/PARC/sp1/eca/oi.html> for updates.
- [24] L. Liu and R. Meersman. The Basic Building Blocks for Modeling Communication Behavior of Complex Objects: an Activity-driven Approach. *ACM Transactions on Database Systems*, 21(3):157–207, 1996.
- [25] L. Liu and C. Pu. A Transactional Activity Model for Organizing Open-ended Cooperative Activities. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS-31)*, Big Island of Hawaii, HI, January 1998.
- [26] L. Liu and C. Pu. Methodical Restructuring of Complex Workflow Activities. In *Proceedings of the 1998 IEEE Conference on Data Engineering*, Orlando, Florida, February 1998.
- [27] C. Mohan. *Advanced Transaction Models - Survey and Critique*. Tutorial presented at the ACM SIGMOD international conference, 1994.
- [28] C. Mohan, G. Alonso, R. Gunthor, and M. Kamath. Exotica: A research perspective on workflow management systems. In *IEEE Bulletin of the Technical Committee on Data Engineering*, pages 19–26, March 1995, Vol.18, No.1.
- [29] M. Nodine and S. Zdonik. Cooperative transaction hierarchies: a transaction model to support design applications. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 83–94, 1990.
- [30] Open Buying on the Internet (OBI). <http://www.supplyworks.com/obi/>.
- [31] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.
- [32] Internet Open Trading Protocol (OTP). <http://www.otp.org:8080/>.
- [33] C. Pu, R. Barga, T. Zhou, and S.-W. Chen. Implementing Extended Transaction Models. In *High Performance Transaction Systems (HPTS) Workshop 1997*, Pacific Grove, California, September 1997.
- [34] C. Pu, G. E. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the 14th International Conference on Very Large Data Bases*, 26-37, August 1988.
- [35] C. Pu, T. Autrey, A. Black et al. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Colorado, December 1995.
- [36] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wasch, and P. Muth. Towards a cooperative activity model - the cooperative activity model. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 194–205, 1995.
- [37] Secure Electronic Transaction (SET). <http://www.mastercard.com/set/specs.html>.
- [38] Visa Cash. <http://www.visa.com/cgi-bin/vee/nt/cash/main.html?2+0>.
- [39] J. Wasch and A. Reuter. The ConTract model. In *Elmagarmid [15]*, pages 219–264, Chapter 7, 1992.
- [40] T. Zhou, C. Pu, and L. Liu. Adaptable, Efficient, and Modular Coordination of Distributed Extended Transactions. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996.