Supporting Dispute Handling in E-commerce Transactions, a Framework and Related Methodologies

Jian Tang[*] Ada Waichee Fu[+] Jari Veijalainen[++]
*Department of Computer Science
Memorial University of Newfoundland
St. John's, NF, A1B 3X5, Canada
Jian@cs.mun.ca

[+]Department of Computer Science and Engineering
Chinese University of Hong Kong
Shatin, Hong Kong
adafu@cse.cuhk.edu.hk

[++]Department of Computer Science and Information Systems
University of Jyväskylä
Jyväskylä, Finland
veijalai@cs.jyu.fi

Abstract

An E-commerce transaction is a means to conduct particular commercial activities using the global digital E-commerce infrastructure. We concentrate here on business to customer (B-to-C) E-commerce transactions. These transactions are based on protocols offered by the global infrastructure, primarily the Internet. Using electronic means to do business can greatly improve the efficiency of the business transactions. It however poses some problems that were rarely considered to be important before. One class of problems is caused by the behavior of untrusted participants. For reasons such as dishonesty, disputes may arise. In the general case, when a dispute arises an untrustworthy participant may have an arbitrary behavior, and may or may not cooperate with the dispute handling process. In this paper, we study one class of disputes where the participants have some limited willingness to cooperate, and the causes or demands by the initiators are directly related to the actions in the transactions in which the disputes arise. To this end, we first establish a correctness criterion by extending the existing notion of transactional atomicity to a broader context where the behavior of untrustworthy players is taken into consideration. We then introduce the concept of a dispute, and the difficulties to handle it caused by abnormal behavior of players. We introduce a strategy, *benefit set*, based on which a solution is built to circumvent that behavior. Due to the special features of E-commerce transactions, we propose a two-tier arbiter structure as the main mechanism in our solution.

Keywords: E-commerce, protocol, dispute, atomicity, customer, merchant, arbiter, transaction

1 Introduction

E-commerce transactions allow people and/or companies to conduct commercial activities using the global digital E-commerce infrastructure. Using an electronic infrastructure to do business can greatly improve the efficiency of the business transactions. It however poses some problems that were rarely considered to be important before. For example, when a customer is trying to purchase a product through the Internet, how does she know if the person or company she is contacting is really the one who put the advertisement there in the first place? What provisions should be made in an E-commerce transaction protocol if some parties involved in the performance of the transaction are not trusted? On the Internet, it is hardly possible to know the trustworthiness of a party in advance each time one wants to perform a transaction, because anyone can set up an attractive E-commerce site with a little cost and be immediately in global business. Also, it is not possible to have a legally binding facility in electronic messages as easily and at such a low cost as a hand written signature. Third, E-commerce is really a global phenomenon and all local and national legislative instruments are insufficient to guarantee a legal framework for the activity. Due to these factors, among others, E-commerce transaction protocols may not always be followed faithfully by all parties.

So disputes may arise in E-commerce for many reasons. Although many E-commerce transaction protocols have been proposed, very few of them have paid adequate attention to the support of dispute handling. To the best of our knowledge, the only work that studies this issue is reported in [1], where some interesting issues are addressed, such as formulation of dispute claims, architecture of dispute handling systems, etc. (See Section 5 for more detail.) The directive in [6] contains the framework legislation for E-commerce. However, it gives neither any standard structure nor protocol for dispute handling. It regulates at an abstract level the ordering process (place order and confirm it) and everything else is left to market or member countries to decide.

In this paper, we study the aspects of an E-commerce transaction relating to dispute handling in an environment where the participating parties may be untrustworthy. We concentrate on one kind of disputes where the participants have some limited willingness to cooperate, and the causes or demands by the initiators are directly related to the actions in the transactions in which the disputes arise. Our main contributions are the following: (1) We establish a correctness criterion, based on which the dispute handling mechanism is built, by extending the existing notion of transactional atomcity [17] to a broader context where the behavior of untrustworthy players is taken into consideration; (2) We introduce a strategy, called benefit set, to circumvent the untrustworthy behavior of players; (3) We propose a model for an E-commerce transaction

protocol that supports the dispute handling process and introduce a two-tier arbiter structure that resolves a dispute in a semi-algorithmic way.

The rest of the paper is organized as follows. In Section 2, we describe E-commerce transactions, their structures and semantics. In this section, the concept of dispute is also introduced. In Section 3, we introduce the E-commerce transaction protocol and its representation. Then we introduce the concept of a benefit set. After that we introduce a two-tier arbiter structure based on the previous concepts. In Section 4, we discuss some related issues. We conclude the paper by presenting the related work and summarizing the main results.

2 E-commerce Transactions

2.1 Structure

An E-commerce transaction involves a number of *players*. Each player is associated with a *role*. For the purpose of this paper, we always include two roles, *customer* and *merchant*. The role of a customer is to make a payment to the merchant and as a result receive some goods. The role of a merchant is to deliver the goods in exchange for the payment he receives from the customer. The goods can be electronic (in the terminology of [6] these are Information Society Services, ISS) or tangible. There is a third role, called *arbiter*. This role is solely for resolving a dispute, should one occur. (In reality, there may also be other roles involved in a transaction, such as banks or credit card companies, which are responsible for the real transfer of funds between the customer and the merchant, intermediate servers that perform certain authentication for the players, etc. Since the presence of these roles will not alter our discussion in an essential way, we omit them in this paper.) In the following discussion, for ease of references, we will use the term 'player' to also denote its role.

Structurally, an E-commerce transaction is a sequence of three logical operations: *negotiation, purchasing*, and an optional *remediation*, as depicted in Figure 1.

During *negotiation*, customers and merchants try to reach an agreement on the terms and conditions on the goods to be purchased and their prices. A successful negotiation is followed by the *purchasing* operation, which consists of three sub-operations. (The double arrowed line indicates that fund_transfer and goods_delivery can be in either order, depending on the transaction.) Given below are their definitions, where we use *result* to denote the negotiation results, *o* a purchase order, *c* a customer, and *m* a merchant.

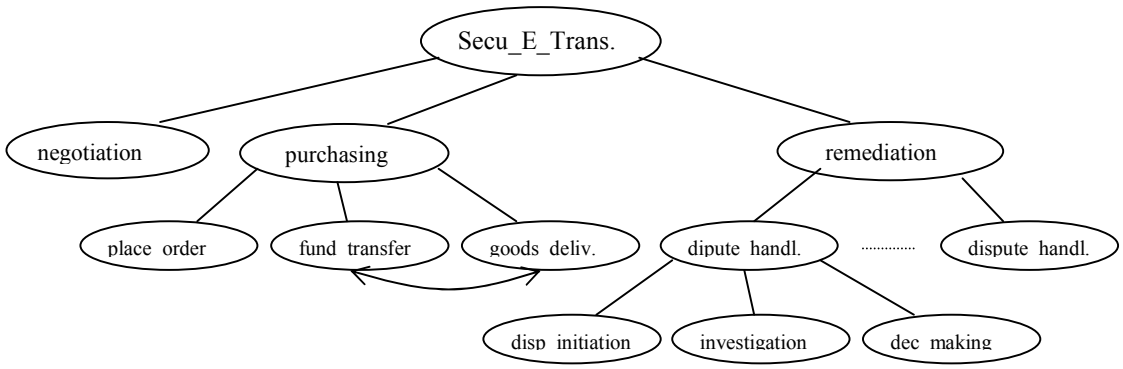


Figure 1. Structure of Secure E-Commerce Transaction.

place_order(Order *o*, Negotiation_Result *result*, Customer *c*, Merchant *m*) {
    c places a purchase order *o* from *m* based on *result*
}

fund_transfer(Money *x*, Customer *c*, Merchant *m*) {
    pay(*x*, *c*, *m*) // *c* makes a payment for an amount of *x* to *m*
    get_money(*x*, *m*, *c*) // *m* receives a payment for an amount of *x* from *c*
}

goods_delivery(Goods_Description *x*, Merchant *m*, Customer *c*) {
    ship(*x*, *m*, *c*) // *m* ships goods described by *x* to *c*
    get_goods(*x*, *c*, *m*) // *c* receives goods described by *x* from *m*
}

The operations *place_order( )*, *pay( )*, *get_money( )*, *ship( )* and *get_goods( )* are primitive operations, and therefore are executed atomically. That is, if they succeed, the prescribed work is done; otherwise, nothing has happened effectively. Note that we take a *state-oriented* view here, i.e., if a work is done, some state of interest is changed, otherwise the state remains intact. For example, *pay(x, c, m)* succeeds if and only if an amount of *x* is deducted from the account of *c*, and get_money(x, m, c) succeeds if and only if an amount of *x* is credited to the account of *m*.

If operation *place_order( )* succeeds, a purchase order *o* will be generated. The generated purchase order legally binds *c* and *m* by describing the goods to be purchased, such as the name, model, quantity, manufacturer, etc., and the price to be paid. (It may also specify additional constraints such as the delivery date, the deadline for the payment, etc.) We use *price(o)* and *goods_des(o)* to denote the price and the description of the goods specified in *o*. We can now define *purchasing* operation as:

purchasing(Negotiation_Result *result*, Customer *c*, Merchant *m*) {
    if place_order(*o*, *result*, *c*, *m*) succeeds {
    customer_prepare (*x*, *price(o)*); // prepare an amount of money *x* such that *x = price(o)*
    merchant_prepare(*y*, *goods_des(o)*);// prepare goods described by *y* such that *y = goods_des(o)*
    fund_transfer(*x*, *c*, *m*);
    goods_delivery(*y*, *m*, *c*);
  }
}

Note that the *customer_prepare( )* and *merchant_prepare( )* both simulate human actions, and, therefore, can not be atomic. When they fail, some unexpected results may be generated, for example, $x < price(o)$ and $y \neq goods\_des(o)$.

The operation *remediation* refers to a process to resolve disputes that may subsequently arise following the purchasing operation. We will discuss it in more detail in Section 2.4.

2.2 Atomicity

An E-commerce transaction must possess certain properties to be practically useful. In this section, we discuss these properties. The following definitions 1 and 2 rephrase the money and goods atomicity proposed in [17].

Definition 1: The operation *fund_transfer(x, c, m)* preserves ***money atomicity*** if either both *pay(x, c, m)* and *get_money(x, m, c)* succeed, in which case we say it succeeds, or both operations fail, in which case we say it fails.

Definition 2: The operation *goods_delivery(x, m, c)* preserves ***goods atomicity*** if either both *ship(x, m, c)* and *get_goods(x, c, m)* succeed, in which case we say it succeeds, or both operations fail, in which case we say it fails.

Definition 3: The operation *purchasing(result, c, m)* preserves ***purchase atomicity*** if
(1) *fund_transfer(x, c, m)* preserves money atomicity and *goods_delivery(y, m, c)* preserves goods atomicity, and
(2) for the sub-operations in *purchasing( )*, either all of them succeed, or *place_order( )* fail and *fund_transfer( )* transfers no money and *goods_delivery( )* delivers no goods.

Since the theme of this paper is to handle the anomalies directly related to the players in the execution of transactions, we will assume that fund transfer and goods delivery operations always succeed. Then, purchase atomicity can possibly be violated only in two cases:

(1) the order is placed, but the customer (merchant) has prepared incorrect payment (goods), or
(2) no order is placed, but there are some none-zero fund transferred/goods delivered.

The second case is rare. Hence, we will mainly consider the first case.

2.3 States and Actions

The notion of atomicity has a simple representation in terms of ***transaction state*** (or simply, 'state'). We can view a state as a mapping from {***order, money, goods***} to {−1, 0, 1}. *order* takes a value of 1 if an order has been placed, and 0 otherwise. *money* takes a value of 1 if the full payment is transferred, −1 if an insufficient payment is transmitted, and 0 if no payment has been made. We store into data item *goods* a value of 1 if the goods are delivered that meet the conditions as specified in the purchase order, −1 if the delivered goods do not meet those conditions, and 0 if no goods have been delivered. We can see that if the value of <*order, money, goods*> is <1,1,1> or <0,0,0>, then purchase atomicity is preserved.

Actions are the physical operations that implement the logical operations in an E-commerce transaction. For example, 'customer made a payment to merchant', and 'bank sent a confirmation to merchant' are actions. A proposition may claim occurrence or non-occurrence of an action.

An action that is related to a state variable is termed a ***state-sensitive*** action. As will be seen in later sections, state-sensitive actions play an important role in dispute handling.

Each action has an ***initiator,*** which is the subject part in the action statement. In particular, if an action is a message delivery, the sender of the message is always the initiator.

2.4 Remediation Revisited

Disputes are usually caused by abnormal behavior of humans. Human anomalies are harder to handle than machine anomalies. For example, in the current literature, most practical methods dealing with machine anomalies assume fail-stop behaviors. This is because even for very limited non-fail-stop machine malfunctions, the solutions mainly have paper value only. (Consider Byzantine General problems [10], for example.) For human related anomalies, however, assuming fail-stop behavior is not realistic. Moreover, for intentional human anomalies, the offenders may try to cover it up. This may include, for example, supplying false information to show their innocence, and denying wrong doing when no one can prove otherwise. In most cases, a malfunctioning machine does not have this capability. In the face of these difficulties, dispute handling provides an alternative in attacking human anomalies.

2.4.1 Structure of Dispute Handling

Remediation is a collection of zero or more operations, called *dispute resolutions*. Each dispute resolution consists of three parts, *dispute initiation, investigation* and *decision making.*

Dispute Initiation

Normally, a dispute results from failure of some players to fulfill certain obligations. We formulate a dispute initiation as a five tuple ***<initiator, target, complaint, request, Tid>.***

- The *initiator* identifies the player who initiates the dispute.
- The *target* indicates the player (or set of players) against whom the dispute is initiated.
- The *complaint* is a pair of sets of *statements* (Y, N). The interpretation of this pair is as follows: the dispute initiator claims that all the statements in Y are true and all the statements in N should also be true but in reality they are not.
- The *request* is a set of statements, each of which states an action that the initiator wishes to be taken, in order to repair the damages that he thinks the target has done to him.
- Finally, the *Tid* identifies the transaction for which the dispute is initiated.

A statement can be represented by a variety of forms. One of the most commonly used is the *prefix form*, whereby a statement is represented as <*operation, subject, object*>, meaning that subject performs operation on object. Note that, in the general case, the operation part may be associated with some attributes. For example, `pay by credit card before April 10, 2000' where`by credit card' and `before April 10, 2000' are two attributes.

The essential parts of a dispute initiation are the complaint and the request. In the general case, the dispute initiator needs to establish two assertions:

(1) all the statements in set Y are true and those in set N are not.

(2) when all the statements in Y are true the statements in set N represent the obligations of the associated subjects. (In the following, we call this assertion a ***YN-Obligation***.)

We can use purchase atomicity as a criterion to check for the fulfillment of obligations. Since the request, not the complaint, is the ultimate goal of the initiator, if a request contributes to the preservation of purchase atomicity, then it is an obligation to be fulfilled, and, therefore, should be honored. In this case, we do not even need to consider the complaint.

From the above discussion, whether or not the request statements contribute to the preservation of purchase atomicity is the very first thing the arbiter needs to know. This in turn requires him to have knowledge about the current transaction state in which the dispute is initiated.

Investigation

In this phase, the arbiter tries to *construct* the current transaction state. Let $X \rightarrow_s Y$ be a state transition via the state sequence *s* where X is the initial state and Y the ending state. Suppose a dispute arises when the transaction is in state Y. The arbiter can construct state Y if he knows sequence *s*. Unfortunately, this is not always possible. This is because all the players may not be fully cooperative in telling the truth about what actions they have/have not taken. The issue here is to maximize the likelihood that they will be cooperative.

Decision Making

This is where the arbiter decides if the request by the dispute initiator will be honored or rejected. The arbiter tries to construct the current state, and then uses purchase atomicity as the base for his decision. If the arbiter can indeed construct the state, a decision can possibly be made algorithmically; otherwise, the arbiter has to exercise his or her discretion.

3 E-commerce Transaction Protocols

3.1 Concepts an Definitions

An E-commerce transaction protocol is a collection of rules that govern the way the three operations (funds transfer, goods delivery, remediation) are executed in an E-commerce transaction. Shown below is such a protocol.

Protocol 1

1. The customer sends a purchase order form to the merchant, which includes the specification of the goods, price, and her credit account information.

2. The merchant checks if the information in the order form is acceptable. If it is, he sends a confirmation to the customer, otherwise, he sends a negative confirmation to the customer and stops.

3. The merchant checks the quantities of the goods ordered. If they are small, he delivers the goods immediately, and sends a delivery slip (e.g. by post, email or fax) which uniquely identifies the goods delivered.

4. The merchant submits the account information to the payment transfer infrastructure for money transfer.

5. If the goods have not been delivered, the merchant delivers the goods and sends a delivery slip that uniquely identifies the goods delivered.

6. After she receives the goods, the customer sends back a receipt to the merchant that uniquely identifies the goods she has received.

We assume that once the delivery slip (receipt) has been sent, the recipient will get it. We now formally describe a protocol.

Definition 4: A ***normal protocol*** is a quadruple $<M, P, F, (S, \prec)>$ where M is a set of message types, P is a set of players, S is a set of symbols, F is a mapping, $F: S \rightarrow M \times P \times P$, and $(S, \prec)$ is a partial order, such that

1. $\exists p_1, p_2 \in P, o \in M, r \in S, \forall s \in S, [p_1 \neq p_2 \ \& \ F(r) = <o, p_1, p_2> \ \& \ (s \neq r \Rightarrow F(s) \neq <o, p_1, p_2> \ \& \ r \prec s)]$

2. $\forall s_1, s_2 \in S, [s_1 \prec s_2 \Rightarrow F(s_1) \neq F(s_2)]$

In condition 1, *o* is a special message type, called *order,* and *r* is called *initial order transmission (IOT).*

The symbols in S are *ids* of message transmissions, which are associated with the tuples in set $M \times P \times P$ as their *values* through the mapping F. A tuple $<m, p, q> \in M \times P \times P$ indicates that player p sends a message of type m to player q. The partial order implies the protocol executions may follow different execution paths. Condition 1 requires that any message transmission follows the *IOT* that possesses a unique value. Condition 2 states that two different message transmissions in any execution path have different values. Using the terminology in Definition 4 to formulate Protocol 1, we have: M = {pay, delivery, slip, receipt, order, order_confirmation, order_rejection}, P = {customer, merchant}. S is a set of distinct symbols, one for each occasion where a message is sent in the protocol. F specifies the message type sent, the sender and the receiver at each occasion. $\prec$ represents the temporal orders between messages.

A normal protocol, however, deals only with normal executions. Since the protocol is the most important reference for an arbiter to pin-point the current state in dispute handling, we would like it to contain maximum information. For this purpose, we will extend a protocol to also contain certain exceptions. A message that does not meet the specification is termed a ***bad message***, otherwise, it is a ***good message***. (In case no confusion is possible, we omit the preceding 'good'.) Insufficient payments and goods with defects are examples of bad messages. In the following, we use *bad t* to denote the type for a bad message of type *t*.

Definition 5: Let $PT = <M, P, F, (S, \prec)>$ be a protocol, and $N \subseteq M$. Let $N_B = \{bad\ x: x \in N\}$. An ***extended protocol for PT on N,*** denoted as *extend(PT, N),* is a quadruple $<M \cup N_B, P, F', (S \cup S_B \cup$

S', $\prec_e$)> where S, S_B, and S' are pair-wise disjoint, and F' is a mapping, F': $S \cup S_B \cup S' \rightarrow (M \cup N_B) \times P \times P$ such that

1. $\prec \subseteq \prec_e$ *and* $\forall s_1 \in S, s_2 \in S_B, [F(s_1) = F'(s_1) \& F'(s_2) \in N_B \times P \times P]$
2. $\forall s \in S, <y, p, q> \in N \times P \times P, \exists r \in S_B, [F'(s) = <y, p, q> \Rightarrow F'(r) = <bad\ y, p, q>]$
3. $\forall s \in S_B, <bad\ y, p, q> \in N_B \times P \times P, \exists r \in S, [F'(s) = <bad\ y, p, q> \Rightarrow F'(r) = <y, p, q>]$
4. $\forall s_1, s_2 \in S, s_3 \in S_B, <y, p, q> \in N \times P \times P, [(F'(s_2) = <y, p, q> \& F'(s_3) = <bad\ y, p, q>) \Rightarrow (s_1 \prec s_2 \leftrightarrow s_1 \prec_e s_3)]$
5. $\forall s_1 \in S', \exists s_2 \in S, r \in S_B, [F'(s_1) = F(s_2) \& r \prec_e s_1]$
6. $\forall s_1, s_2 \in S \cup S_B \cup S', [s_1 \prec_e s_2 \Rightarrow F'(s_1) \neq F'(s_2)]$

Set N is called the ***generating set,*** and set N_B is called the ***generated set***.

The first condition implies that *extend(PT, N)* preserves the orders and ids for all the good message transmissions in *PT*. Conditions 2, 3 and 4 together establish that, in *extend(PT, N)*, for each message transmission of a type in N, there is a bad message transmission of the corresponding type in N_B, and they are preceded by the same set of the other transmissions. Condition 5 tells that a bad message transmission may be followed by good message transmissions. Finally, Condition 6 states that *extend(PT, N)* retains the requirement that any path does not duplicate message transmission values.

3.2 Protocol tree

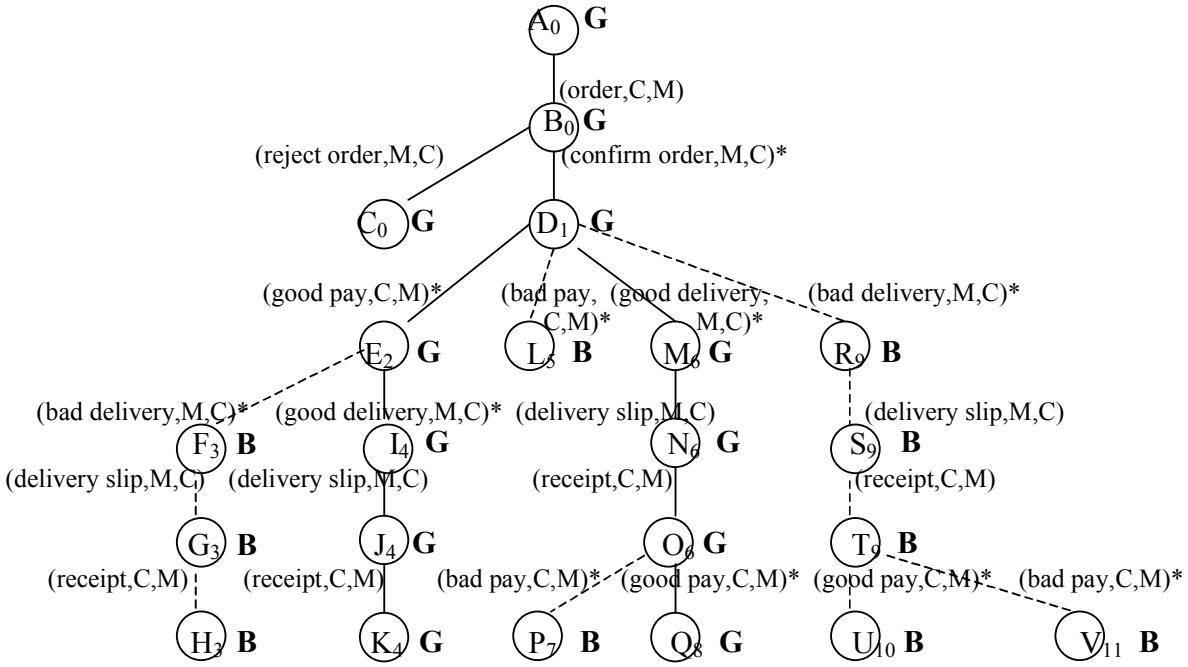
A protocol, either normal or extended, implies a partial order of message transmissions. We shall represent it in a tree structure.

Definition 6: Let $T = <M, P, F, (S, \prec)>$ be a protocol, either normal or extended, and $\prec_{immd}$ be a subset of $\prec$ that denotes 'immediately precede'. The ***protocol tree*** of T, denoted as *tree(T)*, is an arc-labeled non-empty tree defined as follows. For any node n,

- if n is the root, then for each minimal element $r \in S$ (i.e., wrt. $\prec$), we generate a child c of n, and label the arc (n, c) with *F(r)*, and
- if n is not the root, let p be its parent and $s \in S$ be the element for which n is generated. Then for each $r \in S$ such that $s \prec_{immd} r$, we generate a child c for n and label the arc *(n, c)* with *F(r)*.

Shown in Figure 2 is the protocol tree for the extended protocol for Protocol 1. We extend Protocol 1 by defining the generating set as {*pay, delivery*}, and generated set as {*bad pay*, *bad delivery*}. We then predict the messages that might be transmitted after these bad messages. In the tree, each node is an execution stage that is entered after all of the preceding message transmissions have taken place. The subtree containing all the solid arcs is the protocol tree for Protocol 1, and the set of subtrees containing broken arcs are the extensions. For each node in the tree for Protocol 1, we attach a letter 'G' to indicate that it is preceded by good message

transmissions, and for each node in the extended parts we attach a letter 'B' to signify that it follows a bad message transmission. Since a path describes a class of similar executions[1], if it ends at a G-node then all the messages transmitted are good. In particular, if it ends at a G-node that is a leaf node, atomicity is preserved. On the other hand, no B-node preserves atomicity.



Legend: C: customer
M: merchant
(m,x,y): x sends m to y

Figure 2. Protocol tree for Extended Protocol 1.

Each node is associated with a state, which is a mapping from {*order, money, goods*} to {0, 1, −1}. (Refer to Section 2.3.) In the extended protocol tree, the state change is indicated by the subscripts of the identifying letters of nodes *in the same path*. In each path, starting from the root, the letters have the same subscripts if and only if their nodes possess the same states.

3.3 Supporting Dispute Handling

3.3.1 Benefit Set

[1] We are interested only in paths originating from the root, since other paths do not represent transactions in our model.

As mentioned before, in order for the arbiter to make a judgment, players may need to prove that they or other players have or have not taken certain actions. The issue here is to maximize the likelihood that they will be cooperative.

Definition 7: Let T be an extended protocol tree, and p be a player. The ***benefit set*** of p w.r.t. T, denoted as benefit(p,T), is a set of propositions such that $r \in$ benefit(p,T), if and only if, all the following conditions are true:
(1) r claims a state-sensitive action a
(2) (initiator(a) = p) $\Rightarrow$
(r claims occurrence of a and the arc for action a points to a G-node) or
(r claims non-occurrence of a and the arc for action a connects a G-node to a B-node)
(3) (initiator(a) $\neq p$) $\Rightarrow$
(r claims occurrence of a and the arc for action a connects a G-node to a B-node) or
(r claims non-occurrence of a and the arc for action a points to a G-node)

Simply put, the propositions in the benefit set of a player claim that he/she has (has not) taken actions contributing to preserving (destroying) atomicity, or other players have (have not) taken actions that contribute to destroying (preserving) atomicity. The motivation here is that a player should have little worry to prove any proposition with a truth value of 'true' in his/her benefit set if he/she is able to do so. Let T be the extended protocol tree in Figure 2. Based on Definition 7, the benefit sets of the customer and the merchant with respect to T are as follows[2].

benefit(CUSTOMER, T):
C1: MERCHANT did not confirm the order
C2: CUSTOMER made good payment
C3: CUSTOMER did not make bad payment
C4: MERCHANT did not deliver good goods
C5: MERCHANT delivered bad goods

benefit(MERCHANT, T)
M1: MERCHANT confirmed the order
M2: CUSTOMER did not make good payment
M3: CUSTOMER made bad payment
M4: MERCHANT delivered good goods
M5: MERCHANT did not deliver bad goods

[2] For easy understanding, we still use natural language, instead of prefix form, to present a statement in a benefit set here.

Note that some propositions are strictly stronger than some others, such as C2 than C3, C5 than C4, etc. Nonetheless, in each pair both propositions must be included. This is because they generate different states. (For example, C2 always assigns 1 while C3 may assign 0 to *money.*) Omitting either one may result in losing state information, and hence, cannot guarantee that a complete state will be constructed after all the propositions with truth values of 'true' are proven. (Refer to the discussion following Theorem 1 in the next subsection.)

3.3.2 Using Benefit Sets to Construct Transaction States

Maximizing the likelihood that players are willing to provide proofs is the motivation for the way we defined benefit sets. The question is: can we obtain a complete transaction state after the propositions in the benefit sets are proven? The result is contained in the following theorem. For easy presentation, we will use the phrase 'positive (negative) proposition on action *a*' to refer to a proposition that claims the occurrence (non-occurrence) of action *a*. We say that a proposition *is true at a node* in an extended protocol tree, iff, the actions that have/have not occurred by the time when the transaction execution reaches that node make the proposition true. We say an action *a* is in a path if the arc labeled *a* is in the path.

Theorem 1: Let R–D be a path where R is the root in an extended protocol tree. Then for any state-sensitive action *a*: (1) it is in this path iff there is a positive proposition on it in some benefit set that is true at node D, and (2) it is not in this path, iff, there is a negative proposition on it in some benefit set that is true at node D.

Proof: First note that from the way a benefit set is constructed, for any state-sensitive action *a* present in the protocol tree, there is both a positive and a negative proposition on it in some benefit sets. Notice that *a* is in R–D iff it has occurred when the transaction execution reaches node D. By definition, the positive proposition on *a* is true at node D iff action *a* has occurred when the transaction execution reaches node D. Thus, Assertion 1 is true. Assertion 2 directly follows from Assertion 1 and the first statement in this proof. □

Assume a dispute is initiated at node D. Theorem 1 states that the set of all the propositions in the benefit sets that are true at node D completely identifies the state-sensitive actions that have or have not occurred when the transaction execution reaches D. This implies that the state of node D can be constructed using these propositions. Consider node E_2 in Figure 2, for example. The propositions that are true at this node are M1, M5, C2, and C4. They give the state: the merchant has confirmed the order, the customer paid in full, but the merchant has not yet delivered either good or bad goods. As can be seen from the figure, this is indeed the state for node E_2. (Notice that

based on the propositions that are true in a state one cannot in general reconstruct completely the path which was followed to reach the state. To see this, assume that the protocol allows operations to commute (e.g. pay =a_k and deliver =a_i can happen in any order). Let SS(D) = <a_1,..a_s,a_k,a_i> and SS(D') = <a_1,..,a_s,a_i,a_k> be the paths that lead to nodes D and D', respectively. In both nodes the same propositions are true. Consequently, the states are the same. Still, the paths followed are different. However, since both paths lead to the same state, such a difference will not affect the arbiter's decision.)

3.4 Realizing Arbiter Using a Two-Tier Structure

From the previous discussions, if the arbiter has complete knowledge about the current state, then it can immediately make a judgment on whether or not the request by the dispute initiator installs atomicity for the current state, otherwise, human involvement is necessary. From this perspective, it is appropriate to implement the arbiter using a two-tier structure, a computer software and a human. The software arbiter first tries to generate an algorithmic solution. It does this by first constructing the benefit set for each player. It then sends these benefit sets to the players for proving. The players then try to prove those propositions and inform the software arbiter of the results. Based on these results, the software arbiter may or may not be able to make a decision. In case it cannot, it turns the case over to the human arbiter for arbitration.

3.4.1 Constructing Benefit Sets

The algorithm can be developed almost entirely based on Definition 7. Let X be a player, and P(X) be the benefit set for him/her. For each action A, let I(A) denote its initiator.

Algorithm 1 – constructing the benefit sets

1. Construct the extended protocol tree for the protocol;
2. Mark each action whose incident nodes contain differing subscripts as state-sensitive;
3. For each path in which all the arcs are good message transmissions, mark a G on each node in the path;
4. Mark the remaining nodes with B;
5. For each state-sensitive action A which ends at a G-node, include into set benefit(I(A)) a statement saying "I(A) has taken action A", and into set benefit(J) a statement saying "I(A) did not take action A", where J is a different player from I(A);
6. For each state-sensitive action A that starts from a G-node and ends at a B-node, include into set benefit(I(A)) a statement saying "I(A) did not take action A", and into set benefit(J) a statement saying "I(A) has taken action A";

It is easy to verify that when we apply the above algorithm to the protocol tree in Figure 2 we obtain exactly the benefit sets given in Section 3.3.1. Note that from the way the benefit set is constructed, at any stage of a transaction, every player knows the truth value of every proposition in his/her own benefit set. This point is essential for the effectiveness of the arbiter system.

3.4.2 Generating a Decision

We focus our attention on how the software arbiter works. The method is described by the following Algorithm.

Algorithm 2 — decision making

Input: Extended protocol tree EPT
Dispute initiation <*initiator, target, (yes_set, no_set), request, tid*>
Benefit set *benefit(player, EPT)* for each player *player*

Output: A decision on whether *request* will be honored, or turn the case over with a message to human arbiter

Initialization: *statement_set* ← Φ
1. for each player *player* and *proposition* ∈ *benefit(player, EPT)*
2. if **CanProve1***(player, proposition, tid)* then
3. *statement_set* ← *statement_set* ∪ {*proposition*}
4. *statement_set* ← **Refine(***statement_set)* // discard redundant propositions
5. **Mark***(statement_set, EPT)* // mark the arcs for occurred/not-occurred actions
6. if **CanConstructState***(statement_set, EPT)* = true then // can construct a complete state
7. *G* ← **GetNode***(EPT, statement_set)* //get a node with the complete state
8. if **Atomicity***(G, EPT)* then // G preserves atomicity
9. if **CanProve2***(initiator, yes_set, no_set, tid)* then //can prove *yes_set,* ¬ *no_set,* and YN-obligation
10. human judgement(on the request)
11. else // the initiator cannot prove what he has complained
12. no action is taken
13. else // the state does not preserve the atomicity
14. *new_statement_set* ← **GetNewSet***(statement_set, request)* //incorporate *request*
15. **Mark***(new_statement_set, EPT)*
16. *G′* ← **GetNode***(EPT, new_statement_set)*
17. if **Atomicity***(G′, EPT)* then // *request* reinstall atomicity
18. accept *request*
19. else // the request does not reinstall atomicity
20. if **CanProve2***(initiator, yes_set, no_set, tid)*
21. human judgement (on the request)
22. else // the initiator cannot prove what he has complained
23. no action is taken
24. else // cannot construct a complete state
25. human judgement

Eight functions are used in Algorithm 2.

- *Mark(statement_set, EPT)* simply marks the arcs in *EPT* for the actions that are claimed to have occurred or not occurred by the propositions in *statement_set.* The marked tree will be referenced by some other functions.
- *CanConstructState(statement_set, EPT)* returns *true* if the propositions in *statement_set* can determine a complete state in *EPT*, and *false* otherwise.
- *CanProve1(player, proposition ,tid)* should be viewed as a communication interface (such as a remote procedure call). It returns *true* if *player* can prove that *proposition* is true for transaction *tid*, and *false* otherwise. (See [4] on theorem proving.)
- *CanProve2(initiator, yes_set, no_set, tid)* is also a communication interface. It returns *true* if *initiator* can prove that the propositions in *yes_set* are true, those in *no_set* are false, and the YN-Obligation for transaction *tid* is valid, and *false* otherwise.
- Function *Refine(statement_set)* discards from *statement_set* those propositions that are strictly weaker than some other proposition in the same set.
- The function *GetNode(EPT, statement_set)* returns a node where every proposition in *statement_set* is true.
- The function *Atomicity(G, EPT)* works on node *G* whose state is given. It returns *true* if this state possesses atomicity, and *false* otherwise.
- The function *GetNewSet(statement_set, request)* integrates *statement_set* and *request* by removing every proposition in the former that conflicts with a proposition in the latter and then unionizing the two.

We list the pseudo code for some of the above functions in Figure 3 - 8. We use the phrase 'complete path' to refer to a path that starts from the root and ends at a leaf.

```
Void Mark(StatementSet SS, ExtendedProtocolTree EPT)
1.   if any arc is already marked remove the mark
2.   for each complete path x in EPT
3.       for each p ∈ SS
4.           if p claims occurrence of an action then
5.                for any arc in x that is labeled with p, mark that arc with Y
6.           if p claims non-occurrence of an action then
7.                for any arc in x that is labeled with ¬p, mark that arc with N
```

Figure 3. Marking the arcs in the extended protocol tree.

```
Boolean CanConstructState(StatementSet SS, ExtendedProtocolTree EPT)
1.  for each complete path y that contains an arc for each action claimed in SS
2.          if in y all the state-sensitive arcs are marked and
```

every arc marked with Y precedes every arc marked with N
3. then return (*true*)
4. return (*false*)

Figure 4. Determining if the current state can be constructed

StatementSet Refine(StatementSet *SS*)
1. $SS' \leftarrow \Phi$
2. for each $h \in SS$
3. for each $h' \in SS$ such that $h' \neq h$
4. if *h implies h′* then
5. $SS' \leftarrow SS' \cup \{ h' \}$
6. return $(SS - SS')$

Figure 5. Discard unnecessary propositions

StatementSet GetNewSet(StatementSet *SS*, RequestSet *RS*)
1. for each $r \in RS$
2. for each $s \in SS$
3. if *r* implies $\neg s$ then
4. remove *s* from *SS*
5. return $(SS \cup RS)$

Figure 6. Incorporating requests

Node GetNode(ExtendedProtocolTree *EPT*, StatementSet *SS*)
1. Find a path R–D–L where R is the root and L is a leaf such that:
all the state-sensitive arcs in R–D are marked with Y &
all the state-sensitive arcs in D–L are marked with N &
R–D contains an arc for each positive proposition in *SS*
2. return (D)

Figure 7. Get a node for a statement set

Boolean Atomicity(Node *D*, ExtendedProtocolTree *EPT*)
1. Find the complete path *y* that contains *D*
2. if *D* is a B-node then
3. return (*false*)
4. else if at least one arc in *y* is marked with N then
5. return (*false*)
6. else
7. return (*true*)

Figure 8. Determining if a node preserves atomicity

In Algorithm 2, in line 7, a node with the current state is returned. When control reaches line 9, the current state preserves atomicity, and hence the initiator's request will either destroy or be unrelated to atomicity. In the former case, it is a compensation, and in the latter case, it is ad-hoc. If the initiator can show that the target indeed has some obligation to fulfill, the software arbiter will ask the human arbiter to decide if the request is reasonable, otherwise it will not consider that request. When control reaches line 18, the current state does not preserve atomicity but the initiator's request reinstalls it, the software arbiter immediately accepts the request. When the control reaches line 19, the initiator's request does not reinstall atomicity. But if he can prove the target failed to fulfill an obligation, then his request will still be considered, otherwise, it will be rejected immediately.

In the function *CanConstructState( )*, if *SS* is the set of all the propositions in the benefit sets that are true at some node in EPT, the existence of a complete path that meets the conditions in lines 1 and 2 is guaranteed. The following is the justification. By Assertion 1 of Theorem 1, and from the way the arcs are marked, there exists a path R–D where R is the root such that the arcs marked with Y in this path correspond to the positive propositions in *SS*. In any path D–L where L is a leaf, if it contains a marked arc, then that arc is labeled with a state-sensitive action, say *b*. By Condition 6 in Definition 5, any arc in R–D cannot be labeled by *b*. By Assertion 2 of Theorem 1, a negative proposition on *b* is in *SS*. Thus that arc is marked with N.

In function *GetNode( )*, note that although the state is transaction-specific, since the execution of the transaction has been reflected by the truth values of the statements in set *SS*, we do not need the transaction id as a parameter. Note also that there can be multiple nodes that meet the conditions in line 1, but all have a state that is consistent with the truth values of the statements in *SS*. (Refer to the illustration at the end of Section 3.3.2.)

It is worth mentioning here about function *CanProve*2*( )*. This function is called in lines 9 and 20 in the main algorithm. In both cases, the dispute initiators are considered to have requested for compensations. In these cases the initiators will have to prove YN-obligations to keep their cases open. In our model, compensations are not included in the purchase orders, and therefore their justifications can not be made through the use of atomicity. Thus proving YN-obligations in order to justify the compensation requests in these cases will most likely require external evidence, such as oral promises, written promises via e-mails, or faxes, etc. A detailed discussion of this issue is beyond the scope of this paper.

3.5 Scenarios

In this section we give two scenarios which serve as examples to illustrate how a two-tier structured arbiter functions. These scenarios are based on the extended protocol tree in Figure 2. The benefit sets constructed in each scenario are the ones shown in Section 3.3.1.

Scenario 1: The customer initiates a dispute complaining 'I paid in full, but the merchant delivered bad goods to me' and thus requesting 'exchange of goods '. We have: complaint = ({(good pay, c, m), (bad delivery, m, c)}, {(good delivery, m, c)}, and request = {(good delivery, m, c)}. In line 2 of Algorithm 2, the software arbiter requests each player to prove any proposition in the benefit set when the dispute is initiated. Suppose it is true that the customer has paid in full but the merchant made a bad delivery. Also, suppose that the customer has received the delivery slip and sent back her receipt. Now five propositions, C2, C3, C4, C5 and M1 are true at the time when the dispute is initiated. The customer proves C2 by showing the bank statement. This also proves C3. She proves C5 by showing that the goods identified on the delivery slip are indeed bad. This also proves C4. The merchant proves M1 by showing the signed order form. Thus in Algorithm 2 when control reaches line 4, we have *statement_set* = {(good pay, C, M), (bad delivery, M, C), (confirm order, M, C), ¬(bad pay, C, M), ¬(good delivery, M, C)}, which is then refined as *statement_set* = {(good pay, C, M), (bad delivery, M, C), (confirm order, M, C)}. During the execution of function *Mark(statement_set, EPT)*, the arcs for the propositions in *statement_*set are marked. At line 6, the function *CanConstructState(statement_set, EPT)* returns *true.* This is because there is a path, say (A_0, H_3) in which all the state-sensitive arcs, (B_0, D_1), (D_1, E_2) and (E_2, F_3) are marked with Y. (Another path that also meets the condition is (A_0, U_{10})). Then in line 7, *GetNode(EPT, statement_set)* returns a node, say H_3. This node does not preserve atomicity, since it is a B-node. When the control reaches line 16, the function *GetNode(EPT, new_statement_set)* is called again, with *new_statement_set* = {(good pay, C, M), (good delivery, M, C), (confirm order, M, C)}. It will return K_4. The test at line 17 evaluates to *true.* Hence, the customer's request is honored.

Scenario 2: The customer initiates a dispute with complaint 'The merchant promised to reimburse me 25% of the total price if I paid in full before June 12, but he did not keep his promise even though I paid on June 11 ' and request ' refund 25% of the amount I have paid '. Thus the complaint is ({(good pay: before June, 12, C, M)}, {(refund: 25%, M, C)}) and the request is {(refund: 25%, M, C)}. Suppose the customer is right. The software arbiter again constructs the

benefit sets and requests the players to prove propositions that are true. The following propositions are true when the dispute is initiated: C2, C3, M1, M4 and M5. Again, all these propositions are proved by the players respectively, which are therefore all included into *statement_set*. After the refinement, we have *statement_set* = {(good pay, C, M), (good delivery, M, C), (confirm order, M, C)}. The test at line 8 is true. The arbiter now knows that the current state preserves atomicity, but the customer's request effectively destroys it. He then asks the customer to prove that the statements in the *yes_set* of her complaint are true, as well as the YN-Obligation, which reads: if the customer paid in full before June 12, then the merchant is obliged to refund 25%. The customer can prove the statement in the *yes_set* by showing the bank statement. If she can also prove the YN-Obligation to the satisfaction of the arbiter, then the arbiter will make a suggestion to the human arbiter; otherwise, her request will be rejected.

4 Discussion

Our discussion so far has assumed that the players are not always trustworthy. In reality, however, some players may be generally viewed as trusted. One example is a bank. When trusted players are present, the general framework discussed in the paper is still applicable, except that we do not need to preclude from the benefit set of a trusted player any proposition relating to messages sent to or received from it. Neither do we need to include these propositions into the benefit sets of other players.

Another related aspect concerns the practical significance of the decisions made by arbiters. In the real world, a decision made by a judge will be enforced. Can we say the same for the arbiter here? Solving theses issues requires a cooperative effort from other sectors of our society. For example, a dispute handling mechanism for E-commerce transactions must, first, gain the recognition of the governments of all the countries that are willing to participate in such a system. Then it must obtain a proper legal status in order for the decision made by an arbiter to be legally binding. Although these issues are by no means easy, and still require a great deal of investigation, yet they are important.

It is worth noting that any message transmission in an E-commerce transaction may be associated with a temporal constraint. Our model can be extended to include them in a straightforward manner. The only change we need to make is: if in the purchase order a message type is associated with a temporal constraint we will consider it as a special attribute which does not have an impact on whether the message itself is 'good' or 'bad'. (For example, if a message with a type of 'goods' is associated with a deadline, then it is still a good message even if its

delivery does not meet that deadline.) Now our arbiter system functions as is. The initiator of a dispute may now make a stronger case than before by establishing a YN -Obligation that would be impossible without a temporal constraint. For example, suppose the customer complains 'the merchant did not deliver before April 15' and requests 'refund 50% of the total price', where 'April 15' has been written on the purchase order as the deadline for the goods delivery. Now the customer can formulate a YN-Obligation as: Y = {(payment, customer, merchant)}, N = {(deliver: before April 15, merchant, customer)}. She can show that the merchant did not fulfill his obligation, and hence add some weight to her request. She would not be able to establish this YN-Obligation if 'April 15' had not been written on the purchase order.

5 Related Work and Concluding Remarks

E-commerce transactions have been studied extensively in the last decade. A comprehensive review is given in [9]. Many protocols with varying levels of security guarantee have been proposed [2, 3, 5, 7, 11, 14, 21]. Atomicity is an issue that has been thoroughly explored in database transactions in the last two decades [8, 12, 20]. Its relevance to E-commerce transactions is introduced in [17, 18]. In [13,19], the authors give a more detailed analysis on the transactional properties of E-commerce transactions. However, these work have not paid attention to dispute handling, except for [1].

The salient feature in [1] is a language that can be used to express some sophisticated claims by a dispute initiator and an architecture for a dispute handling mechanism in payment systems. However, in their formulation, there is no mention of a correctness criterion based on which a decision can be made. Thus, it is unclear how such a system can determine whether or not any request by the dispute initiator should be honored.

An E-commerce transaction is very different from traditional transactions. In this environment, not only can machines fail, but some players may also act maliciously. We study one of the strategies to handle human related anomalies. That is to let a player raise a dispute, which will then be handled by a dispute handling system, in an attempt to post-protect his interest if the player is indeed the victim. To this end we study a model for E-commerce transactions, and introduce both the theoretical and the implementation aspects. Among these are the concept of benefit set for circumventing the untrustworthiness of players, and a two-tier arbiter structure.

Some issues deserve further study. Our benefit set model assumes the willingness of cooperation by all the players, including untrustworthy ones, on proving things that 'do not conflict' with their interests. If some players are not as cooperative then the effectiveness of the

two-tier arbiter system deteriorates. A possible approach would be to make it possible also for a player to do the proving of propositions in other players' benefit sets. How this affects the performance needs to be analyzed. Another issue is concerned with the capability of the software arbiter. In the current system, a software arbiter is not capable of handling any request for compensation, neither can it handle disputes not related to atomicity. This is because the structures and semantic contents of these requests or disputes usually are more irregular than those under the framework of purchase atomicity. A possible solution is to let the software arbiter do supervised learning. It can be trained first with a set of predefined requests as well as the answers to these requests. It will then classify the new requests based on the learned knowledge. Here an important issue is how to formulate a training instance using a request. Exactly how the supervised learning is structured in the context of arbiters is an interesting topic to explore.

REFERENCES

1. N. Asokan, E. Herrweghen and M. Steiner, "Towards a Framework for Handling Disputes in Payment Systems", Proc. of 3rd USENIX Workshop on Electronic Commerce, 1998.

2. M. Bellare, J. Garay, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik and M. Waidner, "iKP – a Family of Secure Electronic Payment Protocols", Proc. of the 1st USENIX Workshop on Electronic Commerce, 1995.

3. D. Chaum, "Achieving Electronic Privacy", Scientific American, 1992, pp 96 – 101.

4. D. Duffy, "Principles of Automated Theorem Proving", John Wiley & Sons Ltd., 1991.

5. B. Cox, J. D. Tygar and M. Sirbu, "NetBill Security and Transaction Protocol", Proc. of the 1st USENIX Workshop on Electronic Commerce, 1995.

6. Directive 2000/31/EC of the European Parliament and of the Council of 8 June 2000 on certain legal aspects of information society services, in particular electronic commerce, in the Internal Market ("Directive on electronic commerce"). Official Journal of the European Communities, Vol. 43, L178 (17.7.2000).

7. E. Gabber and A. Silberschatz, "Agora: a Minimal Distributed Protocol for Electronic Commerce", Proc. of the 2nd USENIX Workshop on Electronic Commerce, 1996.

8. J. Gray and A. Reuter, "Transactions Processing: Techniques and Concepts", Morgan Kaufmann, San Mateo, CA, 1994.

9. S. Kimbrough and R. Lee, "Formal Aspects of Electronic Commerce: Research Issues and Challenges", *International Journal of Electronic Commerce,* Vol. 1, No. 4, 1997, pp 11 – 30.

10. L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem", ACM Transactions on Programming Languages & Systems, Vol. 4, No. 3, July 1982, pp 382 – 401.

11. J. H. Lee, "A Resilient Access Control Scheme for Secure Electronic transactions", Proc. of the 2nd USENIX Workshop on Electronic Commerce, 1996.

12. N. Lynch, M. Merritt, W. Weihl and A. Fekete, "Atomic Transactions", Morgan Kaufmann, San Mateo, CA, 1994.

13. H. Schuldt, A. Popovici and H. J. Schek, " Execution Guarantees in Electronic Commerce", Proc. of 8th Intl. Workshop on Foundations of Models and Languages for Data and Objects In: Gunter Saake, Kerstin Schwarz, Can Türker (eds.), Transactions and Database Dynamics. Lecture Notes in Computer Science Nr. 1773, Springer Verlag, Berlin, December 1999, pp. 193 –202.

14. D. Steves, C. Yurkanan and M. Gouda, "A Protocol for Secure Transactions", Proc. of the Second USENIX Workshop on Electronic Commerce (EC96) Nov. 18-21, 1996. http://www.sage.usenix.org/publications/library/proceedings/ec96/full_papers/steves/html/index.html.

15. SWIFT see http://www.swift.com.

16. J. Tang, Ada Fu and J. Veijalainen: "On Dispute Handling in E-Commerce", Technical Report, CSE, CUHK 2001.

17. J. D. Tygar, " Atomicity in Electronic Commerce", PODC 96, pp. 8 – 26.

18. J. D. Tygar, "Atomicity versus Anonymity: distributed Transaction Electronic Commerce", Proc. of the 24th VLDB Conference, 1998, 1 – 12.

19. J. Veijalainen, "Transactions in Mobile Electronic Commerce", Proc. of 8th Intl. Workshop on Foundations of Models and Languages for Data and Objects. In: Gunter Saake, Kerstin Schwarz, Can Türker (eds.), Transactions and Database Dynamics. Lecture Notes in Computer Science Nr. 1773, Springer Verlag, Berlin, December 1999, pp. 208- 229.

20. J. Veijalainen, A. Wolski, Transaction-based Recovery. Chapter 11 in: A.Elmagarmid, M. Rusinkiewicz and A. Sheth (eds.), Management of Heterogeneous and Autonomous Database Systems, Morgan Kaufmann Publishers, San Francisco, CA, USA, October 1998, pp. 301-350.

21. VISA, Mastercard, et.al, "Secure Electronic Transaction Specification", Book 2, Technical Specification, http://www.visa.com/cgi-bin/vee/sf/set/settech.html?2+1, 1996.