To appear in Proc. 3rd Int'l Workshop on Adv. Issues of E-Commerce and Web-Based Info. Sys. (WECWIS 2001), San Jose, Calif., 21-22 June 2001

Guaranteeing Recoverability in Electronic Commerce

Cris Pedregal-Martin Krithi Ramamritham

Computer Science Department, University of Massachusetts
Amherst, Mass. 01003 USA
E-mail: cris@cs.umass.edu

Abstract

Electronic commerce systems (retail, auction, etc.) are good examples of data-based systems that operate under correctness and resilience requirements of a transactional nature but go beyond conventional databases, as they are formed by the aggregation of heterogeneous, autonomous components. In this paper we introduce a framework to specify, analyze, and reason about the behavior of such systems, focusing on how they are designed to make consistent progress in spite of failures. The contributions of this paper are: (a) the introduction of the Guarantee *abstraction to deal with transactional applications; (b) a framework based on guarantees and protocols to* specify *the behaviors of systems and their components and* reason *about the properties of systems and their components; and (c) application of the framework to a common e-commerce scenario. The framework allows the hierarchical composition of transactional systems and their properties, as well as the proofs of these properties: we specify a system's behavior at its most abstract level, and proceed to decompose the specification mirroring the structure of the system's components, considering the role of guarantee-preserving component systems and recovery in each case. In particular, we show how the lower-level properties are supported by the component systems, which we also characterize within the same framework.*

1. Introduction

Many data-based systems go beyond conventional databases, as they are formed by the aggregation of heterogeneous, autonomous components, but operate under correctness and resilience requirements of a transactional nature. In this paper we introduce a framework to specify and reason about the requirements (and how to satisfy them) for such systems to make consistent progress towards their goals in spite of failures. Ensuring progress in spite of failures is the task of recovery, broadly understood to mean the infrastructure that deals with the consequences of failures in a manner that enables the progress of the system. In order to achieve this end-to-end property even in the face of failures, each of the components must observe certain behaviors which we describe in terms of *protocols*, and keep certain promises, which we describe in terms of *guarantees*. The abstractions of *guarantees* and *protocols* are stated in terms of predicates on *events* and on the state of the system as produced by a particular *history* of these events.

Contributions of the Paper. The primary contributions of this paper are: (a) the introduction of the *Guarantee* abstraction to deal with transactional applications; (b) the development of a framework based on guarantees and protocols to *specify* the behaviors of systems and their components and *reason* about the properties of systems and their components; and (c) application of the framework to a commonly occurring e-commerce scenario. An interesting aspect of the framework is that it allows the hierarchical composition of transactional systems and their properties, as well as the proofs of these properties.

A Specific E-Commerce Scenario. Our scenario consists of all the parties in an e-commerce retail (business) transaction, e.g., a customer, a merchant, supplier, and a bank. We describe the system, at the highest level, in terms of how the behaviors and relationships between these components contribute to the progress of an e-commerce (business) transaction. Specifically, the property, or goal, of the e-commerce system is that (the right) goods be exchanged for (the correct amount of) money [9]. In e-commerce, as in most real systems, this end-to-end property must hold in spite of failures; in other words, once the merchant confirms and order to a client, the goal of completing the exchange of goods for money must be attained eventually.

Throughout the paper, we consider the specific scenario (illustrated in Figures 1 and 2) of an online Merchant selling to a Customer who pays with a credit card[1] issued by a Bank, which functions as a trusted third party for the transfer of money between the Merchant and the Customer. A Supplier provides the goods to the Customer as instructed

[1] We chose this well-known scenario for pedagogical reasons – to capture enough real world details without overwhelming the reader.

Table 1. Example Guarantees.

Example	enable_action	disable_action	trigger_action	discharge_action
E-Commerce	Merchant obtains authid from Bank	Merchant cancels authid with Bank	Merchant tells Bank to pay based on prior authorization	Bank pays Merchant
(Bank's) Database System	transaction T's update p to ob put in persistent store	Abortion of T	Commitment of T	ob reflects T's update p
DB's Recovery System	redo log record for update p to ob put in persistent store	Abortion of T	redo update p on ob (during recovery)	update p redone on object ob

by the Merchant. (M stands for the Merchant, C for the Customer, B for the Bank, and S for the Supplier.) [2]

Each party to the e-commerce system is in turn a system of its own, typically dependent on existing components. For example, the Merchant is dependent on business rules that involve managing inventory, billing, and communicating through interactions with the Customer, a Supplier, and Bank. To complete its part of the business transaction in spite of failures, the Merchant relies (a) on the guarantees provided by the Bank, the Supplier, and the Customer, (b) on following certain protocols in its interactions with these entities, and (c) on the recovery properties of its own database system. Figure 1 reflects the actions taken by the parties involved, under the assumption that all requests are granted and the sale succeeds. The scenario is explained in detail in Sections 2 and 3.

Guarantees allow Compositional Treatment of Recoverability Properties. We provide a specification framework which exposes only the abstract properties of the underlying components that play a role in supporting the current level's functional and recovery-related goals. For example, a Merchant should not confirm an order unless it is sure that it will be able to complete it (i.e., arrange successful shipping and charging) in spite of failures. Specifically, to process an order, the Merchant needs to secure a payment promise from the Bank. This is satisfied by obtaining a credit card charge authorization, in effect a guarantee that the Merchant will be paid later by presenting the authorization to the Bank. When the Merchant externalizes the order confirmation, it is making a promise to the Customer that the goods will be shipped, which relies on the guarantee from the Bank that it will make payments that correspond to prior authorizations and on the guarantee from the Supplier that the goods have been reserved from the stock. The Bank, in turn may rely on (i) the Bank's contract with the Customer, and (ii) the recoverability of the Bank's internal database. The above example is characterized by the precedence constraints governing the actions (the interactions with its components) of the Merchant and the following guarantees: the Merchant's order confirmation is a guarantee to the Customer that the order will be shipped and requires (a) a guarantee from the Supplier that the ordered goods will be shipped, (b) a guarantee from the Bank that (the corresponding) authorized charge will be honored, and (c) a guarantee from the Customer that it will recognize (to the Bank) the charges arising from this sale. Notice that this does not specify how *each* component supports its guarantee, but does impose the requirement that it support it (for instance, the Bank cannot forget about authorizations given and the corresponding escrows on accounts). This example illustrates that our ideas are suitable for rich hierarchical treatment of the kind of systems we are interested in.

Within a component, a database transaction system offers the recovery support for its guarantees, relying in turn, on the guarantee of persistence its own database recovery system supports. For example, by updating its authorizations and accounts data structures within a transaction, the Bank obtains atomicity and durability which enable it to implement guarantees, both external and internal. That an authorized charge will be paid is an external guarantee offered to the Merchant and supports directly the end-to-end property of exchange of money for goods. The guarantee that the Bank will be able to bill the Customer for the charge is internal and necessary for the Bank's own recovery from failures. Each level has protocols that guide its interactions with the components it uses, which together with the guarantees at that level allow the system to achieve its goals.

Thus, by exposing how guarantees depend on each other via abstract requirements, we formalize expectations that are enforced *within* each component, as these systems are typically aggregations of autonomous components. For example, in asserting that it will have the goods shipped, the Merchant relies on the Supplier's guarantee of reserving the necessary stock, without the ability to see, much less impose, how the Supplier implements the promise.

Reasoning about End-to-End Properties of Systems. So far, we examined the framework with respect to its ability to *specify* the protocols and guarantees involved at each level of the system. Our framework is also designed to support *reasoning* about the system, for example, (a) to show

[2] For simplicity, we assume that a single Bank takes care of the Customer and the Merchant's payment details; in the real world, there would be at least two banks involved, with mutual guarantees for the transfer of charges and payments.

Table 2. Basic Notation.

Notation	*Name*	*Description*
H	(System) history	partial order (p.o.) on events
event	event/action	operation op; also $\varepsilon, \delta, \ldots$ in formulas
$\rightarrow$ e.g., $\varepsilon \rightarrow \delta$	happens-before	denotes order: ε happens before δ in history H
$(\mathsf{X}, \mathsf{Y}, \mathsf{action} : parlist)$	action invocation message sending	subsystem X invokes action w/parameters $parlist$ on Y X sends Y message action w/parameters $parlist$

that a guarantee expected at one level can indeed be delivered by the component that gives the guarantee, (b) to prove correctness properties such as those involving the correct exchange of goods for money, and (c) to understand the consequences of a change in the system, and thus ensure that the desired goal is still achieved. Regarding the latter, consider again the case of the charge authorization the Merchant secures from the Bank in order to commit an order, and suppose that the Merchant passes on the credit card details to the Bank, which returns an authorization for that charge. An advantage of this method is that the Merchant can discard the credit card data immediately and reduce its security requirements. In a subsequent redesign, the Bank may want to change its agreement with the Merchant to allow the Merchant to "self-authorize" some charges off-line, i.e., without contacting the Bank. This self-authorization may apply to charges under a certain threshold, or for periods in which the Bank is unavailable. Given our specification of Bank's guarantee, it is straightforward to identify the component that needs to accommodate the new functionality, and how, in order to ensure the correct exchange of goods for money.

In summary, the specification and reasoning capabilities of our framework apply to systems which operate in a loosely defined transactional nature, i.e., that:

- manage valuable transactional data, representing money, goods, or other resources,
- require robust behavior in the face of failures, i.e., require support for recovery,
- but, unlike conventional DB transaction management systems, these systems are generally formed by a distributed aggregation of heterogenous and autonomous components, which only guarantee certain results given certain protocols but do not allow access to their internals.

Overall, the challenge here is that the system needs transactional support as a whole yet it is composed of autonomous components.

Outline for the paper. The rest of this paper is organized as follows. We begin in Section 2 by introducing the building blocks of our formalism: actions, guarantees and protocols, and discussing the kind of properties we characterize. We show the application of our framework by studying an electronic commerce example in detail, in Section 3. We then apply our ideas to the analysis of one of the e-commerce components, the bank, in Section 4, and we show how one external guarantee offered by the bank is supported by its internal properties. Finally, we present related work in Section 5 and offer some conclusions in Section 6. Throughout the paper we introduce notation as and when we need it and provide the details in tables off the main text; we do likewise with details of the system.

2. Building Blocks: Guarantees and Protocols

In this section we introduce the building blocks of our framework, namely the notions of actions or events, guarantees, and protocols and give motivating examples. We regard a system as consisting of interrelated subsystems, whose activities and interactions take place in the form of *actions*. Constraints on how those actions may happen in the history of the system are expressed by *protocols*, and recovery and persistence properties are captured by *guarantees*.

2.1. Actions and Events

In the context of the system's history, the occurrence of events corresponds to actions and so we use the terms interchangeably (see Table 2). An action may be the sending of a message between components, or the execution of an operation or step within a component. Actions are defined to be atomic.

Notation: We denote an action with a tuple containing the initials of the subsystem which invokes the action (the sender), the subsystem on which the action is invoked (the receiver), and a description of the action with parameters as appropriate. For example $(\mathsf{M}, \mathsf{B}, \mathsf{auth} : p, custid, ordid)$ is the action in which Merchant asks the Bank for a charge authorization for amount p on the account of customer $custid$ (see Table 3 for further details[3]). Actions may have parameters which we omit when possible to reduce clutter.

2.2. Guarantees

In its simplest form, a guarantee involves two subsystems (components): the *requestor* requests a guarantee

[3] We assume that message transmission always succeeds, i.e., the underlying infrastructure reliably delivers all messages.

from the *guarantor* and in response the guarantor enables the guarantee. Later, the requestor invokes an action that triggers the guarantee.[4] The guarantor *must* (eventually) perform the invoked triggering action and discharge action,[5] which delivers its promise. More precisely, associated with a guarantee G is a 4-tuple: $(\mathsf{enable\_action_G}, \mathsf{disable\_action_G}, \mathsf{trigger\_action_G}, \mathsf{discharge\_action_G})$ With each of the above actions is associated a predicate which becomes true iff the corresponding action takes place. Thus, for a guarantee G, $enabled_G$ is true only after the action $\mathsf{enable\_action_G}$ occurs. When $\mathsf{disable\_action_G}$ occurs, predicate $disabled_G$ becomes true. Similarly, for the other two predicates. For simplicity, we have two independent predicates, $enabled$ and $disabled$; as we shall see, an enabled guarantee can be discharged, only if it has not been subsequently disabled. The semantics of guarantees, with actions as events in the system's history, is defined by the following three statements.

(I) $(enabled_G \ \wedge \ \mathbf{not}\ disabled_G)$
$\Rightarrow\ [triggered_G \Rightarrow\ \Diamond discharged_G]$

The statement means that, once the enable_action action takes place, (unless disable_action happens) if the trigger_action is invoked, eventually the discharge_action will occur, i.e., the guarantee will be discharged. Each action may in turn have preconditions.

We want the actions in the guarantee to capture the intended sequence, so we add the following protocols to the definition of a well-formed guarantee:

(II) $\mathsf{discharge\_action} \in H\ \Rightarrow$
$(\mathsf{enable\_action} \rightarrow \mathsf{trigger\_action} \rightarrow \mathsf{discharge\_action})$

This says that if a guarantee has been discharged, it must have been enabled first, and then triggered, before the discharge happened.

(III) $\mathsf{disable\_action} \in H\ \Rightarrow$
$(\mathsf{enable\_action} \rightarrow \mathsf{disable\_action} \wedge \mathsf{discharge\_action} \notin H)$

This says that if a guarantee has been disabled, it had to be first enabled, and it cannot have been discharged.

Notation. We name guarantees with a letter G followed by either the initials of the requestor and the guarantor, for example GBM, or a brief name related to their use, for example GXauth (see tables 4 and 6).

We illustrate guarantees with the following (simplified) examples (see Table 1). Detailed treatment of guarantees outlined in 1 and 2 below appears in subsequent sections.

1. Once an e-commerce merchant obtains an authorization code *authid* for a specific amount from the bank (that issues the customer's credit card), the merchant has the guarantee that if the Merchant later requests payment of the charge associated with *authid*, the Bank *will* pay the charged amount.

   Note that the Merchant need just present the authorization code *authid* to the bank to trigger the guarantee. Thus the onus of retaining information about the authorized charge so that it can later deliver on the guaranteed payment lies with the bank. For this, the Bank relies on the guarantee (discussed next) given by its database that operations performed by committed transactions (in this case, the transaction invoked by the Bank to record the details of the authorized charge) will persist.

2. Once a database system records in its persistent store (the details of) a specific operation p performed by a transaction T on an object ob, it is guaranteed that, once T commits, the operation p's effects will be reflected in ob.

   Note that the enabling action *transaction T's update p to ob is put in persistent store* can be considered to have been completed if either (a) the state of ob in persistent store reflects the update p or (b) the details of the update p are stored in persistent store, say as a log record. To support this guarantee, the database system in turn relies on the recovery system's guarantee (discussed next).

3. Once a recovery subsystem writes the redo log record for an operation to persistent store, we have the guarantee that, if called upon, the recovery subsystem has the necessary information to perform the operation again.

   This guarantee example can be used to illustrate that a guarantee may never have to be discharged. For example, if the object reflecting the operation's effects becomes persistent (e.g., if the object is "stolen" from the buffer) before the need to redo arises, say, due to a crash, the trigger action *may not occur*. (For instance, ARIES [6] optimizes its redo phase by redoing an operation according to a given redo log record only if the page does not already reflect the operation.)

The three examples given above reflect several aspects of guarantees:

- An enabled guarantee need be discharged only if the triggering action is invoked. This has ramifications for the requestor of the guarantee. For example, the Merchant must persistently store the authorization code given by the Bank so that later, if necessary, it can invoke the triggering action, providing the authorization code.

- An enabled guarantee must be discharged once the triggering action is invoked. This has ramifications for

[4] The enable and trigger actions are typically messages exchanged between the requestor and the guarantor.

[5] The discharge action is typically an actual change in the state of the world effected by the guarantor.

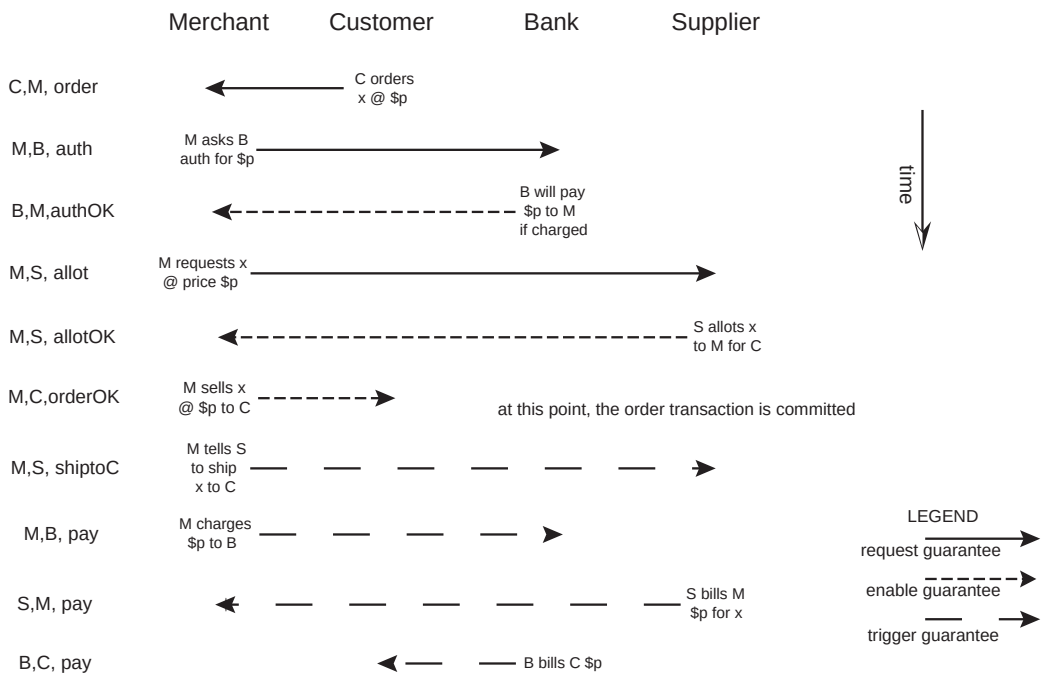

Figure 1. An Electronic Commerce Scenario: Component Subsystems and Actions.

the guarantor. For example, the Bank who gave the authorization must persistently store the authorization details so that later, once invoked, the triggered action can be done (and subsequently the guarantee can be discharged, i.e., the charge paid).

- A guarantee may be disabled before being triggered, which means the guarantor is released from the obligation to discharge the guarantee. For example, the Merchant might abort its sale to the Customer after obtaining the authorization (e.g., if it is unable to secure stock to fill the order), in which case it may explicitly disable it, thus releasing the Bank of the obligation. (This in turn enables the Bank to release the amount escrowed from the Customer's credit line.)

- Guarantees at a certain level can be used, for example, to reason about the functioning of that level of a system without worrying about how the guarantee is achieved by the guarantor. We will be using this feature to reason about the e-commerce system purely based on the guarantees provided by its components C, M, B, S, without resorting to the details of the innards of these component systems.

- Guarantees provided by a certain component typically will be based on the guarantees provided by the subsystems of that component. For instance, a Bank's guarantee to the Merchant is based on the guarantees of the Bank's database which in turn are based on the guarantees provided by the recovery subsystem of the database.

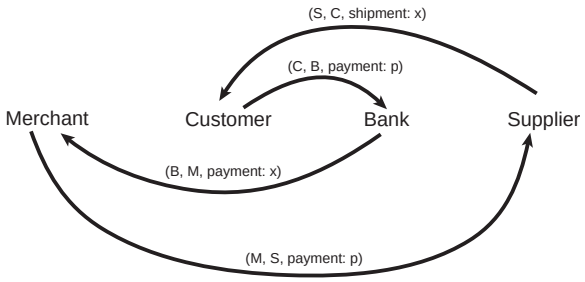

Figure 2. Discharge of Guarantees achieves Goods-Money Atomicity in E-Commerce.

2.3. Protocols

The purpose of a protocol is to dictate correct behavior, by placing conditions on the occurrence of certain events (that is, conditions for the presence of some event in the system's history) or on the precedence relationships between events, which represent actions. For example, $\mathsf{ship} \rightarrow \mathsf{charge}$ (read: ship precedes charge in the system's history) specifies that a (for example) supplier must ship the goods

Table 3. Electronic Commerce Actions.

Precondition	*Action*	*Postcondition*	*Comments*
TRUE	$(\mathsf{C}, \mathsf{M}, \mathsf{order}: x, p, custid)$	$enabled_{\mathsf{GCB}}$	C places order with M
$enabled_{\mathsf{GBM}} \wedge$ $enabled_{\mathsf{GSM}}$	$(\mathsf{M}, \mathsf{C}, \mathsf{orderOK}: x, p, ordid)$	$enabled_{\mathsf{GBM,GSM}} \wedge$ $enabled_{\mathsf{GCM}}$	M confirms order to C (this is the commit point)
$enabled_{\mathsf{GCB}}$	$(\mathsf{M}, \mathsf{B}, \mathsf{auth}: p, custid, ordid)$	$enabled_{\mathsf{GCB}}$	M awaits B's authorization
$enabled_{\mathsf{GCB}} \wedge$ $p \leq Limit(custid)$	$(\mathsf{B}, \mathsf{M}, \mathsf{authOK}: ordid, authid)$	$enabled_{\mathsf{GCB,GBM}} \wedge$ $Limit(custid) -= p$	B guarantees future payment and B escrows amount auth
TRUE	$(\mathsf{M}, \mathsf{S}, \mathsf{allot}: x, p, ordid)$	$enabled_{\mathsf{GMS}}$	M requests stock
$enabled_{\mathsf{GMS}} \wedge$ x in stock	$(\mathsf{M}, \mathsf{C}, \mathsf{allotOK}: ordid, x, C)$	$enabled_{\mathsf{GMS,GSM}} \wedge$ x reserved from stock	S guarantees future shipment S reserves x from stock
$enabled_{\mathsf{GSM}} \wedge$ $(\mathsf{M}, \mathsf{C}, \mathsf{orderOK}) \in H$	$(\mathsf{M}, \mathsf{S}, \mathsf{shiptoC}: x, ordid)$	$discharged_{\mathsf{GSM}} \equiv$ $\Diamond(\mathsf{S}, \mathsf{C}, \mathsf{shipment}: x)$	M tells S to ship goods
$enabled_{\mathsf{GBM}} \wedge$ $(\mathsf{M}, \mathsf{C}, \mathsf{orderOK}) \in H$	$(\mathsf{M}, \mathsf{B}, \mathsf{pay}: p, authid)$	$discharged_{\mathsf{GBM}} \equiv$ $\Diamond(\mathsf{B}, \mathsf{M}, \mathsf{payment}: p)$	M presents charge to B
$enabled_{\mathsf{GCB}} \wedge$ $(\mathsf{M}, \mathsf{B}, \mathsf{pay}) \in H$	$(\mathsf{B}, \mathsf{C}, \mathsf{pay}: p, ordid)$	$discharged_{\mathsf{GCB}} \equiv$ $\Diamond(\mathsf{C}, \mathsf{B}, \mathsf{payment}: p)$	B bills C for the charge it received from M
$enabled_{\mathsf{GMS}} \wedge$ $(\mathsf{M}, \mathsf{S}, \mathsf{shiptoC}) \in H$	$(\mathsf{S}, \mathsf{M}, \mathsf{pay}: p, ordid)$	$discharged_{\mathsf{GMS}} \equiv$ $\Diamond(\mathsf{M}, \mathsf{S}, \mathsf{payment}: p)$	S bills M for the shipment M ordered

before charging for them. Another classic example of a protocol is the Write-Ahead Logging (WAL) protocol that mandates that an update to the database must be preceded by the logging of this update.

Notation: We name protocols with an initial P and a subscript. We specify them via the relation $\rightarrow$ on events in the system's history.

Clearly, protocols are neither novel nor hard to follow, hence they do not deserve elaborate explanations. We would like to point out however, the *forced progress* assumption made about protocols: Suppose X *happens before* Y $(X \rightarrow Y)$, where X and Y are steps in an activity. According to the forced progress assumption, once step X executes, Y will be forced to eventually execute *provided the data needed for* Y *to execute are (eventually) available*. The forced progress assumption underlies progress of normal as well as recovery processing in most systems. For example, in workflow systems, it is the job of a workflow engine to ensure the progress of steps by scheduling a step once its predecessor is complete.

By stating a forced-progress protocol, we are indicating that the component's underlying engine will ensure that the next step will eventually happen, without indicating how. With that protocol specification we can prove properties about the component. In turn, using the protocol specification as a requirement, we can separately show how the underlying recovery supports the forced-progress at the appropriate steps. For example, consider protocol
$\mathsf{P}_7 : (\mathsf{C}, \mathsf{M}, \mathsf{order}: x, p, custid) \rightarrow (\mathsf{M}, \mathsf{B}, \mathsf{auth}: p, authid, ordid)$
(see Table 5). According to the forced progress assumption, once the Merchant receives this order, it is forced to perform the action (M, B, auth), provided all the data needed for the latter are available. Often, it is in the satisfaction of the latter condition that guarantees come into play. In Section 3, when we prove the correctness of the e-commerce scenario, we show that the guarantees specified are sufficient to ensure that appropriate actions will occur.

2.4. End-to-end Correctness Properties

Besides using our framework to show how component systems deliver their guaranteed properties, we can also use it to reason about the correctness properties of the top-level system. In particular we are interested in *Money-Goods Atomicity* [9]. For example, we can show for the E-commerce system that

- the correct exchange of goods for money is guaranteed. That is, no one loses or gains in the process (in this scenario, to keep things simple, we do not model profits, commissions, etc.): the Customer pays money for goods, the Supplier receives money for goods, the Bank and the Merchant neither gain nor lose goods or money; and
- after a Customer has committed his/her order, the rest of the actions, namely, payment, shipping, etc. are guaranteed to take place, even if failures occur within any of the parties involved.

Thus, once (a) the protocols observed by the interactions between the components and (b) the guarantees provided by one component to another are specified in the framework, we can show both the functionality as well as recoverability properties of the scenario using the framework.

These high-level properties can be viewed as guarantees provided to a Customer, and encoded as the following 4-tuple: ((C,M,order), (C,M,orderOK), (C,M,orderNO), *Customer will receive and pay for goods*)
This guarantee is the view the user gets of the e-commerce

system as a whole. (The disable action (C,M,orderNO) expresses the contingencies of item not in stock or customer's insufficient credit.)

3. An Electronic Commerce Scenario

In this section we examine in detail the Electronic Commerce scenario depicted in Figure 1. We begin with an informal description of the system and its components and we establish the guarantees and protocols governing the system and its components. We then show how they yield the end-to-end property of exchanging goods for money. In the next section, when we unravel the details of one of the components, the Bank, we show how the Bank's guarantee holds, making use of the guarantees and the protocols that govern the the Bank's database.

3.1. E-Commerce Actions and Timeline

Here we outline how an order is placed and processed[6] indicating the actions that take place and the guarantees enabled and triggered. See Figure 1.

- GCB: The Customer has a standing agreement with the Bank that it will pay the Bank for goods charged to the credit card.
- GMS: The Merchant has a similar agreement with the Supplier that it will pay for goods (if and when they are delivered to Customer).
- (C,M, order): Customer browses, places an order for goods with credit card.
- (M,B, auth): Merchant requests credit card authorization from Bank.
- (B,M,authOK): Bank authorizes the charge, enabling its guarantee GBM that it will accept charges (for the goods) if and when requested.[7]
- (M,S, allot): Merchant asks Supplier to reserve the ordered goods if they are in stock.
- (M,S, allotOK): Supplier allots goods to this order, guaranteeing (GSM) to Merchant that it is ready to ship goods to Customer.
- (M,C, orderOK): Merchant confirms order to Customer. This causes the Merchant to insert a record in its own database about the details of the order and the details of the guarantees obtained from the Supplier and the Bank.

This is a point of no return: the sale has committed. All parties to the transaction have made the necessary guarantees for the rest of the e-commerce transaction, namely, shipment, charges, payments, etc. to proceed successfully. The parties involved now trigger the guarantees provided to them, which eventually get discharged (see Figure 2):

- (M,S,shiptoC): Merchant tells Supplier to ship to Customer (i.e., triggers GSM).
- (M,B,pay): Merchant asks Bank to pay for the goods (via GBM).

When the Supplier receives (M,S,shiptoC), it triggers

- (S,M,pay): Supplier asks Merchant to pay (using GMS).

When the Bank receives (M,B,pay), its internals cause:

- (B,C,pay): Bank asks Customer to pay (using GCB).

At this point, all the guarantees have been triggered and hence what remains is for the parties to deliver on their guarantees. These are indicated by the events (C,B,payment), (B,M,payment), (M,S,payment), and (S,C,shipment).

3.2. Actions, Guarantees, and Protocols

For ease of explanation, we consider a situation where the order processing proceeds smoothly, i.e., the Bank's authorization succeeds and goods allocation by the Supplier is possible. Thus tables 3 and 5 are not complete, as they only list the actions and protocols occurring in the simpler case. We discuss how to complete the characterization in Section 3.2.1.

Table 4 summarizes the guarantees given and used by the E-commerce components. Table 3 summarizes the actions with their pre- and post-conditions. Table 5 summarizes the protocols followed by the E-commerce components. (For notation see Table 2.) The last five protocols (P_7-P_{11}) are the least interesting –they appear to complete the specification of the subsystems. For example, P_{10} simply says that the Bank only grants authorizations previously requested. The last one, P_{11}, is redundant as it follows from P_8, P_9 and P_2. We focus on the first six protocols (P_1-P_6).

Protocol P_1 expresses the Merchant's business requirement of securing an authorization before confirming the order to the Customer. Protocol P_2 expresses a similar requirement for the goods.

Protocol P_3 indicates that once the order is confirmed, the Merchant *will* tell the Supplier to ship the goods to the Customer. In turn, the Supplier *will* bill the Merchant by protocol P_5. Similarly with protocols P_4 and P_6. These four protocols P_3, P_5, P_4, P_6 each respectively cause the triggering of the guarantees GSM, GMS, GBM, GCB, which in turn ensure that eventually the respective shipping of goods to C, and payments of M to S, B to M, and C to B take place, completing the transaction.

Notice that protocols P_3, P_4, P_5, P_6 are described in Table 5 as "forcing" the action on their right hand side. Although the forced-progress assumption means that each subsystem must strive to advance following the protocols, these four protocols are *guaranteed* to make progress by the guarantees enabled at the time of the action orderOK.

3.2.1 Additional Actions and Protocols

In the treatment of this scenario we have omitted specifying the behavior of the system when the order cannot be

[6] We assume the requests succeed, i.e., the Bank authorizes the charge, and the Supplier allots the goods requested.

[7] The Bank's refusal of an authorization is expressed instead with (B,M,authNO), which does not enable GBM; similarly, the Supplier's refusal is action (M,S, allotNO). See Section 3.2.1.

Table 4. Electronic Commerce Guarantees.

Name	enable_action	disable_action	trigger_action	discharge_action
GCB	(C, M, order :$x, p, custid$)	FALSE	(B, C, pay :$p, ordid$)	(C, B, payment :p)
GMS	(M, S, allot :$x, p, ordid$)	FALSE	(S, M, pay :$p, ordid$)	(M, S, payment :p)
GBM	(B, M, authOK :$ordid, authid$)	(M, B, authOFF :$authid$)	(M, B, pay :$p, authid$)	(B, M, payment :p)
GSM	(S, M, allotOK :$x, ordid$)	(M, S, allotOFF :$ordid$)	(M, S, shiptoC :$x, ordid$)	(S, C, shipment :x)

Table 5. Some Electronic Commerce Protocols.

Name	*Action*	$\rightarrow$	*Action*	*Comments*
P_1	(B, M, authOK :$ordid, authid$)	$\rightarrow$	(M, C, orderOK :$x, p, ordid$)	order confirmed only if auth granted
P_2	(S, M, allotOK :$x, ordid$)	$\rightarrow$	(M, C, orderOK :$x, p, ordid$)	order confirmed only if goods allotted
P_3	(M, C, orderOK :$x, p, ordid$)	$\rightarrow$	(M, S, shiptoC :$x, ordid$)	order confirmed forces shipping
P_4	(M, C, orderOK :$x, p, ordid$)	$\rightarrow$	(M, B, pay :$p, authid$)	order confirmed forces M charge B
P_5	(M, S, shiptoC :$x, ordid$)	$\rightarrow$	(S, M, pay :$p, ordid$)	goods shipped forces S charge M
P_6	(M, B, pay :$p, authid$)	$\rightarrow$	(B, C, pay :$p, ordid$)	M charged B forces B charge C
P_7	(C, M, order :$x, p, custid$)	$\rightarrow$	(M, B, auth :$p, authid, ordid$)	auth. requested only if order in
P_8	(C, M, order :$x, p, custid$)	$\rightarrow$	(M, S, allot :$x, p, ordid$)	goods requested only if ordered
P_9	(M, S, allot :$x, p, ordid$)	$\rightarrow$	(S, M, allotOK :$x, ordid$)	allotted only if requested
P_{10}	(M, B, auth :$p, custid, ordid$)	$\rightarrow$	(B, M, authOK :$ordid, authid$)	authorized only if requested
P_{11}	(C, M, order :$x, p, custid$)	$\rightarrow$	(M, C, orderOK :$x, p, ordid$)	order confirmed only if started

completed, but the reader can easily see how that is done. Consider the situation in which the Bank decides to deny a credit authorization, which in turn will cause the order to be refused. To model that we need to add the corresponding actions and protocols (but no new guarantees). The additional actions mirror the actions which "ok" requests to the Bank and the Merchant, as well as an action to cancel the guarantee from the Supplier if it is already enabled, and the protocols that ensure the added actions happen in the correct sequences. The new actions are:

- (B, M, authNO :$p, ordid$) to signal the denial of the authorization to the Merchant;
- (M, C, orderNO :x, p) to signal the denial of the order to the Customer;
- (M, S, allotOFF :$ordid$) with which the Merchant notifies the Supplier it no longer wants the allotted goods (this is disable_action$_{GSM}$). This action has the precondition that the order be cancelled, i.e., that orderNO happened.

Other protocols of interest are those expressing the behavior of a negative outcome, complementary to the OK protocols (see Table 5):

- (B, M, authNO :$ordid$) $\rightarrow$ (M, C, orderNO :$x, p,$), which indicates the order will be refused if charge is not authorized (cf. P_1);
- (M, C, orderNO :x, p) $\rightarrow$ (M, C, allotOFF :$ordid$), which effects the cancellation (cf. P_3).

A symmetric set of actions and protocols takes care of the case when the supplier does not allot the requested goods. If we want to specify a situation where the Merchant aborts an order, it will simply abide by the protocols to undo the partial work, adding the abort action.
That protocols and guarantees are useful, modular abstractions can be appreciated in the following examples.

Self-authorization. Consider a variant of the E-commerce scenario in which the Merchant is allowed to authorize some charges without consulting with the Bank. For example, the Merchant can authorize a charge if the amount is below a certain threshold. To accommodate this change, we simply add a disjunctive clause to the enabling of the guarantee, e.g.: enable_action$_{GBM}$ is now:
(B, M, authOK :$ordid, authid$) $\vee\ p \leq maxSelfAuth$

Merchant charges after delivery confirmation. Another variant is to add the extra constraint that the Merchant will charge only after receiving delivery confirmation from the Supplier. One way to specify this is by adding another element to the precondition of the charging action, e.g.:
Precondition((M, B, pay :$p, authid$)) $\Rightarrow$ discharge_action$_{GSM}$
Alternatively, we can add the action (S, M, shiptoCOK :$x, ordid$) for the Supplier to notify the Merchant that the delivery of the goods succeeded, and make that action the new trigger_action$_{GMB}$.

3.3. Proof of the exchange of goods for money

Goods and money atomicity: None of the parties involved lose money, either they get their costs reimbursed or they get goods in exchange for their money. Intuition: consider Figure 2 as a graph, where the nodes are the subsystems M, B, S, C, and there is an edge from X to Y if there is a discharge action of the form (X,Y, pay) or (X,Y,ship). This graph has a cycle, which shows that there is a net exchange of money p for goods x between S and C, with M and B disbursing and receiving the same amount p each. To prove the property, we need to prove that all four transfers eventually take place (if the order has been confirmed); the added benefit of our approach is that we prove it to hold in spite of failures.

We need to prove that all four transfers (money, goods, guaranteed to happen by the discharge of GBM, GSM, GCB,

GMS) *will eventually* take place if the Merchant commits the order. This requires two things: (1) all four guarantees are enabled; and (2) all four guarantees are triggered.

We prove that $(\mathsf{M}, \mathsf{C}, \mathsf{orderOK}) \in H \Rightarrow (\mathsf{B}, \mathsf{C}, \mathsf{pay}) \in H$. The proofs that the other guarantees are discharged are similar, and all hinge on the fact that the $(\mathsf{M}, \mathsf{C}, \mathsf{orderOK})$ action is the point of no return for the sale business transaction. This detail ensures the atomicity, i.e., that one transfer happens iff all transfers happen. This payment action is the discharge of guarantee GCB, so we prove it by showing that the guarantee GCB is enabled, and that the triggering action takes place.[8]

1. $\mathsf{enable\_action}_{\mathsf{GCB}} \in H$: This requires proving that $(\mathsf{C}, \mathsf{M}, \mathsf{order}) \in H$, which follows trivially from (the well-formedness) protocol P_{11} which applies to the hypothesis $(\mathsf{M}, \mathsf{C}, \mathsf{orderOK}) \in H$.
2. $\mathsf{trigger\_action}_{\mathsf{GCB}} \in H$: The trigger action is $(\mathsf{B}, \mathsf{C}, \mathsf{pay})$. We use the forced-progress hypothesis to prove $(\mathsf{B}, \mathsf{C}, \mathsf{pay}) \in H$. This requires identifying a protocol whose right-hand side is $(\mathsf{B}, \mathsf{C}, \mathsf{pay})$, and proving that its left-hand side happens, i.e., that the preceding action took place, and the precondition to $(\mathsf{B}, \mathsf{C}, \mathsf{pay})$ holds. Such a protocols is P_6.
   (a) $(\mathsf{M}, \mathsf{B}, \mathsf{pay}) \in H$. This follows from $(\mathsf{M}, \mathsf{C}, \mathsf{orderOK}) \in H$, protocol P_4, and the precondition to $(\mathsf{M}, \mathsf{B}, \mathsf{pay})$ which requires $enabled_{\mathsf{GBM}}$, which holds by $(\mathsf{M}, \mathsf{C}, \mathsf{orderOK})$ and P_1.
   (b) $Pre((\mathsf{B}, \mathsf{C}, \mathsf{pay}))$ holds. It is a conjunct of two conditions, both of which hold, as follows. $(\mathsf{M}, \mathsf{C}, \mathsf{orderOK})$ and P_{11} yield $enabled_{\mathsf{GCB}}$. $(\mathsf{M}, \mathsf{B}, \mathsf{pay}) \in H$ results from the previous part of the proof.

□

The proof above illuminates where the recovery support is used to guarantee the desired system behavior. This support appears in two guises: persistence and forward progress. Persistence gives a component ability to preserve state in spite of failures, which supports the guarantees, by allowing a component to keep information necessary to honor it.

A component's failure recovery algorithm, which enables it to keep trying the step following the last step completed before the failure (which is supported in part by the atomicity of steps), is what underlies the forced-progress hypothesis. Protocols only prescribe necessary conditions, i.e., if a right-hand-side (rhs) happened, then its left-hand-side (lhs) must have happened. Forced-progress endows the protocols with the extra semantics akin to those of rule-based systems, in that the occurrence of a rhs of a protocol always eventually happens if the lhs happened *and* the precondition of the rhs holds.

[8] We use here guarantees that appear in Table 4, protocols in Table 5, and actions and their pre- and postconditions in Table 3; we omit parameters to reduce clutter.

Finally, we also rely on the persistence of the effects of each step in spite of failures. For example, the Bank must remember authorizations it has issued, and the Merchant must remember authorizations it has received, in spite of failures. In each subsystem, this is typically supported by a database transaction system, whose recovery-related behavior can also be expressed also in terms of our actions, guarantees, and protocols. In the next section we look into the details of the Bank, to exemplify the fact that our formalism can be hierarchically applied.

4. A subsystem: The Bank

In the previous Section, we examined how the E-commerce system achieves its end-to-end goods atomicity property by examining the protocols and guarantees of its high-level components. In this section we examine how the Bank relies on its internal database to support the guarantee it offers when it issues an authorization. In particular, we show how the Authorization guarantee GBM follows from the guarantees of the Bank's database to the Bank.

When the Bank receives a request for a charge authorization, it checks its accounts, and if appropriate grants the authorization, recording the particulars in its accounts. When the Bank receives a charge request accompanied by an authorization, it consults its accounts and if appropriate makes the payment, again recording the particulars. The Bank survives failures because its accounts reside in a database and are updated by transactions.

4.1. Bank Subsystem and Timeline

We view the Bank as composed of two subsystems: the banking Application A and the Database system D. A is responsible for communicating with the Bank's clients, i.e., the Merchant and the Customer, and implementing the Bank's business rules via execution of transactions on D. D is responsible for manipulating the accounts data within a transactional framework. Following the scheme of Section 3.2, we list the actions within the Bank in Table 8, guarantees in Table 6, and some interesting protocols in Table 7. The protocols table is not complete, but as sample, note that protocols P_{B2} and P'_{B2} cover the two possibilities (of whether the authorization is approved not); also note that P_{B4} and P_{B5} correspond to the alternate implementations of action Xpay (see column 2, row 2 in Table 8). An interesting detail in this scenario is Guarantee GXauth (see Table 6) whose enabling and triggering action are the same, signifying that the guarantee is obtained and triggered as a result of the same requesting action.

The timeline for the Bank is simple: essentially, the Bank reacts to messages it receives, and also runs internal actions periodically to ensure bills are sent and charges paid. For example, reception by the Bank of (M,B,auth) causes the

Table 6. Bank Internal and External Guarantees.

Name	enable_action *state change*	disable_action *state change*	trigger_action *state change*	discharge_action
GXauth	Xauth($p, custid, ordid$) $(p, custid, ordid, \_) \in authSet$	FALSE	Xauth($p, custid, ordid$) $(p, custid, ordid, \_) \in authSet$	lookup($authid, authSet$)
GXbill	Xauth($p, custid, ordid$) $(p, custid, ordid, \_) \in authSet$	XauthOFF($authid$) $(p, custid, ordid, \_) \notin authSet$	Xpay($p, authid, M$) $(p, ordid, custid) \in billSet$	send(B, C, pay :$p, orderid$)
GXpayout	Xauth($p, custid, ordid$) $(p, custid, ordid, \_) \in authSet$	XauthOFF($authid$) $(p, custid, ordid, \_) \notin authSet$	Xpay($p, authid, M$) $(p, authid, merchid) \in payoutSet$	send(B, Mpayment :p)
GBM	(B, M, authOK :$ordid, authid$)	(M, B, authOFF :$authid$)	(M, B, pay :$p, authid$)	(B, M, payment :p)

Table 7. Some Bank Protocols.

Name	*Action*	$\rightarrow$	*Action*
P_{B1}	(M, B, auth : $p, custid, ordid$)	$\rightarrow$	Xauth $(p, custid, ordid)$
P_{B2}	Xauth $(p, custid, ordid)$	$\rightarrow$	(B, M, authOK : $ordid, authid$)
P'_{B2}	Xauth $(p, custid, ordid)$	$\rightarrow$	(B, M, authNO : $ordid$)
P_{B3}	(M, B, pay : $p, authid$)	$\rightarrow$	Xpay $(p, authid, merchid)$
P_{B4}	Xbill	$\rightarrow$	(B, C, pay :$p, ordid$)
P_{B5}	Xpay $(p, authid, merchid)$	$\rightarrow$	(B, C, pay :$p, ordid$)

start of internal transaction Xauth (see Table 8) whose successful commit records the authorization details and causes the Bank's response (B,M,authOK) (or (B,M,authNO) as appropriate). More to the point, the commit of Xauth enables guarantee GXauth[9] (see Table 6) that the authorization information will be there when transaction Xpay looks it up (via lookup, using $authid$).

4.2. Actions, Guarantees, and Protocols

The actions within the Bank are embodied by transactions which update accounts in the Bank's internal database, and by receiving and sending messages. Actions are summarized in Table 8, in which we use some ad-hoc notation, as follows. The construct: `receive` *message* `start` *action* `end` denotes that when the *message* is received by the Bank, its Application subsystem will start *action*, a database transaction. The construct: `transaction` *program code* `commit end` denotes an atomic transaction, as supported by the database transaction system, which updates the data structures as indicated in *program code*. Finally, the construct: `eventually start` *action* `end` denotes that the application will make sure that *action* is executed eventually (for example, by executing it periodically), provided that any applicable preconditions on *action* are satisfied. With this notation in mind and the remarks at the bottom row of Table 8 we can examine the actions themselves.

[9] All guarantees internal to the Bank are given by the Database.

Action Xauth checks if there is enough credit left for the customer for whom the Merchant requests a charge authorization to cover the amount requested, and if so escrows the amount from the customer's credit line, records the details of the authorization, and grants it by returning a message with an authorization code to the Merchant. Because Xauth is a database transaction, the Database grants a guarantee, enabled by the commit of Xauth, that the updated states of the data structures *creditline, authSet* will survive failures. This enables the Bank to support both its external guarantee GBM and its own internal business rules. An example of the latter is that the Bank needs to keep track of all outstanding authorizations for each Customer, lest it authorize over the credit limit.

Xpay relies on the discharge of guarantee GXauth to validate a request for payment (i.e., to trigger guarantee GBM, identified by *authid*), to prepare the a payment to the Merchant. Xpay also uses GXauth to prepare the corresponding charge to the Customer. When Xpay commits, it triggers guarantees GXbill and GXpayout.

The Bank application starts transactions Xpayout and Xbill to effect the payments and charges set up by Xpay. That these transactions will find all the pending payments and charges is again guaranteed by the database system via the commit of Xpay. The discharge of guarantee GXpayout, and the protocol of eventually executing Xpayout support the discharge of external guarantee GBM. We could have folded Xpayout and Xbill into Xpay, but we wanted to emphasize the decoupling between the Merchant's request to be paid (served by Xpay), which is the triggering action for guarantee GBM, and the effects, i.e., the eventual discharge of guarantee GBM.

4.3. Proof of external guarantee from internals

For the proof in Section 3.3 that the Electronic Commerce scenario satisfies the end-to-end property of goods and money atomicity, we took as a given the guarantees offered by each of the subsystems. Now we turn our attention to the problem of proving that indeed those guarantees hold, given a specification of the guarantor subsystem. With our specification of internals of the Bank we prove that the

Table 8. Bank internal Actions.

Bank's processing of Merchant auth request	Bank's processing of Merchant pay request	Bank's paying and billing

`receive (M,B, auth: p,custid,ordid)`
`  start Xauth (p, custid, ordid)`
`end`
	`receive (M, B, pay: p, authid)`
	`  start Xpay (p, authid, M)`
	`end`
		`do eventually`
		`  start Xpayout`
		`  start Xbill`
		`end`

`transaction Xauth (p, custid, ordid)`
` if credit[custid] < p`
`  then send (M, B, authNO: ordid)`
`  else creditline[custid] -= p`
`  escrow[custid] += p`
`  authid = newkey (authSet)`
`  authSet += (p, custid,`
`  ordid, pending)`
`  send (M, B, authOK:`
`  p, ordid, authid)`
` commit`
`end`
`// all auth info is persistent`
	`transaction Xpay (p,authid,merchid)`
	` if (aid = lookup(authid, authSet))`
	`  and (p =< aid.p)`
	`  then payoutSet +=`
	`  (p, authid, merchid)`
	`  billSet +=`
	`  (p, ordid, cid)`
	` [OR send`
	`  (B,C,pay: p, ordid)]`
	`// else: drop bad authid silently`
	`  commit`
	`end`
	`// info pay and bill is persistent`
		`transaction Xbill`
		` forall z in billSet`
		`  send (B, z.cid,`
		`  pay: z.p, z.ordid)`
		`  billSet -= z`
		`  commit; end`
		
		`transaction Xpayout`
		` forall z in payoutSet`
		`  send (B, z.merchid,`
		`  payment: z.p)`
		`  payoutSet -= z`
		` commit; end`

Xauth commits all the authorization-related information, guaranteeing it will be possible to honor it later.	Xpay does *not* pay the Merchant, but simply ensures that the payment, and the charge to the Customer, are queued for processing later (or, Customer may be billed immediately, see bracketed alternative code).	Xpayout actually discharges the guarantee GBM; Xbill triggers the guarantee GCB (unless OR alternative is used, see left).

Bank's (external) guarantee GBM holds.

We can now formalize the intuition that the guarantee GXauth by the Database system to the Bank, combined with the forced-progress protocol of the banking application, yields via GXpayout the external bank guarantee GBM. Specifically, we need to prove that, given the Bank's internal guarantees and protocols and that $(\mathsf{B}, \mathsf{M}, \mathsf{authOK}:ordid, authid)$ occurs, the occurrence of $(\mathsf{M}, \mathsf{B}, \mathsf{pay}:p, authid)$ ensures that eventually $(\mathsf{B}, \mathsf{M}, \mathsf{payment}:p)$[10] takes place.

1. $\mathsf{Xauth}(p, custid, ordid) \in H$: We have $(\mathsf{B}, \mathsf{M}, \mathsf{authOK}:ordid, authid) \in H$ by hypothesis ($\mathsf{enable\_action}_{\mathsf{GBM}}$ happens). By protocol P_{B1} and the forced-progress assumption, we have that $\mathsf{Xauth}(p, custid, ordid) \in H$. (Recall that the precondition of Xauth is TRUE.)
2. $\mathsf{Xpay}(p, authid, merchid) \in H$: We have by hypothesis ($\mathsf{trigger\_action}_{\mathsf{GBM}}$ happens) that $(\mathsf{M}, \mathsf{B}, \mathsf{pay}:p, authid) \in H$, and by protocol P_{B3} and forced-progress, we get $\mathsf{Xpay}(p, authid, merchid)$
3. $\mathsf{send}(\mathsf{B}, \mathsf{M}, \mathsf{payment}:p) \in H$: The preceding items (1) and (2) are the $\mathsf{enable\_action}_{\mathsf{GXpayout}}$ and $\mathsf{trigger\_action}_{\mathsf{GXpayout}}$ of guarantee GXpayout, whose $\mathsf{discharge\_action}_{\mathsf{GXpayout}}$ is $\mathsf{send}(\mathsf{B}, \mathsf{M}, \mathsf{payment}:, p) \in H$. GXpayout holds as it is supported by the D database subsystem of the Bank [11] Note that the persistence of the set *payoutSet*, ensures that the information recorded by Xpay is found by transaction Xpayout
4. $(\mathsf{B}, \mathsf{M}, \mathsf{payment}:p)$ follows from the meaning of send □

The proof above sheds light on how a higher-level property (GBM) offered by a subsystem (the Bank) can be decomposed in fairly abstract manner in terms of the subsystem's abstract internal properties, which in turn are built on still-lower level mechanisms. The proof is simple because we do not concern ourselves with how the semantics of transactions is supported within the Bank. Such semantics we characterize abstractly via internal guarantees and protocols (and the forced-progress assumption).

5. Related Work

Our focus has been the development of a framework that can be applied across all levels of abstraction in systems that employ transactions as building blocks. Many extended transaction models have been developed over the years to handle activities that have transactional components [4]. In most of these, for example, those built from the saga model [5], the basic building block for dealing with failures is the notion of compensations.Whereas compensations allow completed steps of a failed activity to be rolled back in an application-specific way, they do not provide any abstractions for forward recovery. Some transaction models had the notion of *vital transactions* to allow an activity to commit even if some of the steps fail. What we have shown is that by obtaining recovery-related guarantees, one step of an activity can get the necessary assurance to commit the effects of the activity (from a user's perspective) while continuing with the rest of the activity even after a failure.

[10] These actions are $\mathsf{enable\_action}_{\mathsf{GBM}}$, $\mathsf{trigger\_action}_{\mathsf{GBM}}$, and $\mathsf{discharge\_action}_{\mathsf{GBM}}$, respectively.

[11] In turn, a proof of guarantee GXpayout can be done in terms of the recovery system of the database transaction system.

In retrospect, the vital transactions can be designed to provide the necessary recovery guarantees for the activity to be completed even after a failure.

Related work includes research on long-running activities such as workflows, long-running transactions in general and e-commerce in particular (see [2, 3, 8]). Of necessity, however, most of the descriptions are complex and rich in details specific to the infrastructure, and lacking in abstraction; this makes it difficult to distinguish essential from incidental aspects of the problem and its solution. Redressing this difficulty is one of the goals of the framework developed in this paper.

6. Conclusions and Future Work

The need for a way to specify and reason about recovery scenarios is pressing: new applications and recovery schemes arise today that do not conform to the conventional database recovery methods and assumptions. Examples include e-commerce applications, discussed at length in this paper, as well as mobile applications,workflows, and robust applications such as Phoenix[1]. Understanding, analyzing, and designing recovery are challenging tasks, because recovery permeates many aspects of a system, from low-level buffer management to high-level application semantics. It is in this context that the value of our contributions lie.

In this paper, we presented a framework to specify and reason about recovery by focusing on a classical e-commerce application. We described the behavior and interaction of an e-commerce system's components in terms of protocols (prescriptions of correct behavior) and guarantees (promises of future behavior one component makes to another). Additionally, we precisely showed similarities and differences across levels of abstraction, which show the expressive power and broad applicability of our methodology. Although not discussed here, our framework can also be used to deal with the detailed protocols and guarantees that govern a traditional transaction processing system [7].

Our specification framework exposes the abstract properties of the components that play a role in supporting the current level's end-to-end requirements. Each level is governed by protocols, which together with the guarantees at that level allow the system to achieve its goals. Within the components of a level, its subsystems, e.g., a database transaction system of a Bank, offers the support for the guarantees made by the component. The database transaction system in turn relies on the guarantee of persistence its recovery system supports. For example, by updating its authorizations and accounts data structures within a transaction, the Bank obtains atomicity and durability which allow it to offer its own guarantees which in turn contribute to achieving the end-to-end property of exchange of money for goods in the E-commerce scenario.

We believe that this work can lead towards to a theory of recovery in the broad sense of the term, including: how abstract requirements of recovery can be mapped onto different infrastructures, how different recovery requirements affect requirements on infrastructure, and generally what are the essential and what are the incidental components of recovery.

References

[1] R. S. Barga and D. B. Lomet. Phoenix: Making applications robust. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proc. ACM SIGMOD Conference*, pages 562–564, Philadephia, Penn., USA, June 1999.

[2] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Trans. on Database Systems*, pages 405–451, Sept. 1999.

[3] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proc. of the ACM SIGMOD Conference*, pages 204–214, May 1990.

[4] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan-Kaufmann, San Mateo, Calif., 1991.

[5] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of the ACM SIGMOD Conference*, pages 249–259, May 1987.

[6] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging. *ACM Trans. on Database Systems*, 17(1):94–162, Mar. 1992.

[7] C. Pedregal-Martin and K. Ramamritham. Analyzing, Specifying, and Reasoning to Guarantee Recoverability. Technical Report 01-03, Computer Science, University of Massachusetts, Amherst, Mass., Apr. 2001.

[8] H. Schuldt, A. Popovici, and H.-J. Schek. Automatic generation of reliable e-commerce payment processes. In *Proc. 1st Int. Conf. on Web Information Systems Engineering (WISE'00)*, Hong Kong, China, June 2000.

[9] J. D. Tygar. Atomicity in electronic commerce. In *Proc. of the 15th Annual ACM Symposium on Principle of Distributed Computing*, pages 8–26, May 1996.