Failure Analysis of an E-commerce Protocol using Model Checking

Indrakshi Ray Indrajit Ray
Department of Computer and Information Science
University of Michigan-Dearborn
Email: {iray, indrajit}@umich.edu

Abtract: The increasing popularity of electronic commerce (e-commerce) has necessitated the development of e-commerce protocols. These protocols ensure the confidentiality and integrity of information exchanged. In addition, researchers have identified other desirable properties, such as, money atomicity, goods atomicity and validated receipt, that must be satisfied by e-commerce protocols. This paper shows how model checking can be used to obtain an assurance about the existence of these properties in an e-commerce protocol. It is essential that these desirable properties be satisfied, even in the presence of site or communication failure. Using the model checker we evaluate which failures cause the violation of one or more of the properties. The results of the analysis are then used to propose a mechanism that handles the failures to make the protocol failure resilient.

Keywords: *e-commerce, failure analysis, automated verification, model checking, transaction management*

Contact Address: Indrakshi Ray Email: iray@umich.edu
Department of Computer and Information Science
University of Michigan-Dearborn
Dearborn, MI 48188
Phone: (313) 593-1794, Fax: (313) 593-9967

Failure Analysis of an E-commerce Protocol using Model Checking

Abstract

The increasing popularity of electronic commerce (e-commerce) has necessitated the development of e-commerce protocols. These protocols ensure the confidentiality and integrity of information exchanged. In addition, researchers have identified other desirable properties, such as, money atomicity, goods atomicity and validated receipt, that must be satisfied by e-commerce protocols. This paper shows how model checking can be used to obtain an assurance about the existence of these properties in an e-commerce protocol. It is essential that these desirable properties be satisfied, even in the presence of site or communication failure. Using the model checker we evaluate which failures cause the violation of one or more of the properties. The results of the analysis are then used to propose a mechanism that handles the failures to make the protocol failure resilient.

1 Introduction

The growing popularity of the world wide web has resulted in an increased interest in e-commerce. Consequently a number of e-commerce protocols have been proposed. Most of these protocols ensure that the information that is exchanged between the parties involved in the e-commerce, is protected from unauthorized disclosure and modification. Moreover, researchers have identified several other desirable properties of e-commerce protocols. Examples of these properties include money atomicity and goods atomicity [21], and validated receipt [16]. Money atomicity ensures that money is neither created nor destroyed in the course of an e-commerce transaction. Goods atomicity ensures that a merchant receives payment if and only if the customer receives the product. Validated receipt ensures that the customer is able to verify the contents of the product about to be received, before making the payment. Although such properties have been identified, a major problem is verifying if a given e-commerce protocol satisfies these properties, specially in the presence of network and site failures. In this paper we address the problem of protocol verification using existing software verification techniques. In particular, we use model checking [1, 9, 13, 14] to the modeny atomicity, goods atomicity and validated receipt properties of the secure e-commerce protocol proposed in [16]. In [16] the authors have informally shown that, in the absence of failures, their protocol has the money atomicity, goods atomicity and validated receipt properties.

The reasons for using model checking are as follows. First, model checking is a completely automated technique and considerably faster than other approaches, such as, theorem proving [2, 3, 7, 15]. Second, if a property does not hold, a counter example is produced by the model checker which helps in understanding why the property does not hold. Last, but not the least, model checking has previously been used successfully to verify security protocols [9, 10, 11, 12]. In this paper we use the Failure Divergence Refinement (FDR) model checker [8]. The protocol that is analyzed is expressed as a communicating sequential process (CSP) [18], which we call SYSTEM. Each property that we wish to check is expressed as another CSP process, which we call SPEC. If SYSTEM is a refinement of SPEC (that is, the set of behaviors generated by SYSTEM is a subset of those generated by SPEC), we can infer that the protocol satisfies the property.

An e-commerce protocol, being distributed in nature, is subject to site and/or communications failures. Are the desirable properties satisfied in the event of a failure? Which properties are valid when some site fails? Our formal analysis helps answer these questions. From the analysis we find that, if there is a failure, most properties do not hold for the protocol described in [16]. Thus some extra mechanism needs to be incorporated into the protocol that will ensure the properties even in the face of failure. In this paper we show how to extend the basic protocol to incorporate these mechanisms and make the protocol fault tolerant.

Briefly, our contribution is as follows. First, we show how to model the e-commerce protocol and the desirable properties in a specification language, and then verify these properties using an existing model checker. Formal specification and verification give an increased assurance that the properties are indeed satisfied by the protocol. Second, we show how to model site and communication failures. We use the FDR model checker to identify which failures preserve the properties and which ones do not. Finally, we propose a mechanism to handle the identified failures that do not preserve the desirable properties; integrating this mechanism with the protocol ensures that the resulting protocol satisfies money atomicity, goods atomicity and validated receipt even in the presence of failures.

The rest of the paper is organized as follows. Section 2 briefly describes the basic protocol. Section 3 describes some related work in this area. Section 4 describes the FDR model checker and then discuss how the basic protocol and the desirable properties can be modeled and verified by the FDR model checker. Section 5 describes how the different forms of failures can be modeled by the protocol and illustrates which properties get violated by site or communication failures. Section 6 proposes the mechanism that is incorporated into the basic protocol so that the properties are preserved in the event of a failure. Section 7 concludes the paper with a brief description of some of the future directions we would like to pursue. Appendix A gives the full specification in CSP of the protocol without failures while Appendix B gives the complete listing of the protocol with failures.

2 Basic Protocol

We begin with a brief description of the e-commerce protocol of [16]. The protocol is designed for electronic transactions involving digital products and is based on the theory of cross validation that is proposed by the authors of [16]. In that work the authors demonstrate informally that their protocol satisfies money atomicity, goods atomicity and validated receipt properties in the absence of failure.

Table 1 shows the different steps in the protocol. The protocol works by exchanging messages between a customer (C), a merchant (M) and a trusted third party (TP). The notation $X \rightarrow Y : P$ is used to denote X sends message P to Y. A merchant has several products to sell. The merchant places a description of each product on an on-line catalog service with the trusted third party together with an encrypted copy of the product. If the customer is interested in a product, he downloads the encrypted version of the product (step 1 in table 1) and then sends a purchase order to the merchant (step 2). Note that the customer cannot use the product unless he has decrypted it. Now the merchant does not send the decrypting key unless the merchant receives payment. The customer does not pay unless he is sure that he is getting the right product. This is handled as follows: the merchant sends the product (step 3) encrypted with a second key, K_2, such

1. $TP \rightarrow C$: *download of product encrypted with key* K_1
2. $C \rightarrow M$: *purchase order*
3. $M \rightarrow C$: *product encrypted with a second key* K_2
4. $M \rightarrow TP$: *the decrypting key* $\hat{K}$ *for the product and the approved purchase order*
5. $C \rightarrow TP$: *the payment token and copy of the purchase order*
6. $TP \rightarrow C$: *the decrypting key*
7. $TP \rightarrow M$: *payment token*

Table 1: Messages Exchanged in the E-commerce Protocol

that K_2 bears a particular mathematical relation with the key, K_1, where K_1 is the key the merchant used when uploading the encrypted product on the trusted third party. Additionally, the merchant escrows the decryption key, $\hat{K}$, corresponding to K_2, with the trusted third party (step 4). The mathematical relation between the keys K_1 and K_2, is the basis for the theory of cross validation that has been proposed (see [16] for complete details). Briefly the theory of cross validation states that the encrypted messages compare if and only if the unencrypted messages compare. Thus, by comparing the encrypted product received from the merchant with the encrypted product that the customer downloaded from the trusted third party, the customer can be sure that the product he is about to pay for is indeed the product he wanted. At this stage the customer is yet to obtain the actual product because he does not have the key, $\hat{K}$, to decrypt the encrypted product. Once the customer is satisfied with his comparison, he sends his payment token to the third party (step 5). The third party verifies the customer's financial information and forwards the decrypting key to the customer (step 6) and the payment token to the merchant (step 7).

To provide confidentiality, integrity and non-repudiability, the messages exchanged are appropriately signed, encrypted and (when required) sent along with a cryptographic checksum. Since we do not focus on the cryptographic aspects of the protocol in this work, such details are abstracted away and not shown in table 1.

3 Related Work

Lowe [10, 11, 12] have used the FDR model checker to find attacks on cryptographic protocols. Roscoe et al. [17] have used the FDR model checker together with data independence techniques to prove that some security protocols are free from attacks.

Heintze et al. [9] focus on the non-security aspects of e-commerce protocols and use the FDR model checker to verify the money and goods atomicity properties of two e-commerce protocols – NetBill [4, 19] and Digicash [5]. The important contribution of the work is that it illustrates how to model the e-commerce protocols and the properties of interest in CSP. This work confirms the claims that the NetBill protocol does

have money and goods atomicity and provides a counter-example to illustrate why the Digicash protocol does not have money atomicity.

Heintze et al. [9] assume that neither the NetBill server nor the communication links to the NetBill server ever fail. The authors substantiate this assumption by arguing that banks (the NetBill server provides the services of a financial instituition in this work) provide fail-safe service to customers and, in the worst case, communication with the bank can be made possible using hand-delivery. The authors further assume that the customer and/or the merchant do not fail arbitrarily; if they do so, the atomicity properties are violated. The authors argue that, in this case, even if the atomicity properties are violated, only the failing agent suffers and no harm is caused to the other party.

We also use the FDR model checker to analyze an e-commerce protocol. However, we give a more comprehensive treatment of site and communication failures than given by Heintze et al. [9]. We allow the customer, the merchant, the trusted third party and the communication links to fail arbitrarily. We propose additional mechanisms that ensure that despite failures, the desirable properties are still preserved. In this regard our work is different from the work of Heintze et al.

4 Using FDR to Verify the Properties of the Basic Protocol

FDR [8] is a model checker, based on the theory of CSP, that allows one to check properties of finite state systems. In CSP, processes are described using events and operators. Events cause a process to change state. The representation of states is implicit. A process may be composed of component processes that require synchronization on some events; each component must be willing to participate in an event before the process can make a state transition. This is how the component processes interact with each other.

In FDR, the finite state system is modeled as a process, say SYSTEM, and the property of interest is modeled as another process, say SPEC. If SYSTEM is a refinement of SPEC (that is, the set of the possible behaviors of SYSTEM is a subset of the set of possible behaviors of SPEC), then the property modeled by SPEC holds. If some property does not hold, FDR generates a counter-example that illustrates under what scenario the property is violated. The counter-example generated is very useful for analyzing and debugging purposes.

In the following paragraphs we describe how we model our protocol. The complete FDR specification of the protocol is provided in the appendixes. Figure 1 describes the CSP notations used in our model.

4.1 Modeling the Communication between the Processes

The communication between processes is synchronized in CSP. To model asynchronous communication we follow the approach of Heintze et al. [9]. A sender sends messages over a unique channel (the sender's "out" channel for a particular receiver) while the receiver receives the message over another channel (the receiver's "in" channel corresponding to a particular sender). Thus when two agents are communicating, two channels are associated with each agent for a total of four channels. Additionally, for each pair of channel – sender's "out" and receiver's "in" – we have a process that reads data from the sender's "out" channel and writes the data into the receiver's "in" channel – for a total of two processes.

1. `a?x`

   Input `x` on channel `a`

2. `a!x`

   Output `x` on channel `a`

3. `{|a|}`

   Set of events associated with channel `a`

4. `a -> P`

   Defines the process waiting for event `a`, after `a` it behaves like `P`.

5. `[]x:X @ a?x -> P`

   Defines the process that accepts any element `x` of `X` on channel `a` and behaves like `P`.

6. `P |~| Q`

   Defines the process that non-deterministically can behave like either `P` or `Q`.

7. `P [] Q`

   Defines the process that can behave like either `P` or `Q`. However, preference is given to the unblocked process.

8. `P [|{}|] Q`

   Defines the process in which processes `P` and `Q` run completely independently of each other.

9. `P [|X|] Q`

   Defines the process where all events in `X` must be synchronized and the events outside `X` can proceed independently.

Figure 1: List of CSP Notations

For example, to model the communication between the merchant and the customer four channels, (minc, coutm, cinm, moutc) and two processes (COMMmc and COMMcm) are involved. The process COMMmc reads a data from channel coutm and writes the data to channel minc and the process COMMcm reads a data from channel moutc and writes it to channel cinm. The process COMMcm is modeled in CSP as follows:

```
COMMcm = []x: {po} @(coutm ?x -> (minc !x -> COMMcm))
```

where $\{po\}$ – the purchase order – is the set of all data that is communicated directly from the customer to the merchant. Similarly, we have four channels and two processes for modeling the communication between merchant and trusted third party – channels moutt, tinm, toutm and mint and processes COMMmt and COMMtm – and another set of four channels and two processes for representing the communication between customer and trusted third party – channels coutt, tinc, toutc and cint and processes COMMct and COMMtc. We model the process COMMmc, COMMmt, COMMtm, COMMct, COMMtc in a manner similar to COMMcm.

4.2 Modeling the Customer, Merchant and the Trusted Third Party Processes

4.2.1 Modeling the Customer Process

The protocol starts when the customer browses the catalog hosted on the trusted third party and downloads the encrypted product from there. The downloading of the encrypted product is modeled as the sending of the encrypted product by the trusted third party and the receipt of the product by the customer. Thus, we can say that, initially the customer waits for an encrypted product from the trusted third party.

```
CUSTOMER = cint ?x -> DOWNLOADED_EGOODS(x)
```

Once the customer has downloaded the product, it sends a purchase order to the merchant. This is modeled as:

```
DOWNLOADED_EGOODS(x) = coutm !po -> PO_SENT(x)
```

The customer then waits for the encrypted product from the merchant. On receiving a message from the merchant, the customer checks to see if the message is indeed some encrypted product sent by the merchant. If so, the customer proceeds to the next step, otherwise it continues to wait for the encrypted product. The specification for this event is as follows:

```
PO_SENT(x) = cinm ?y  ->  if (y==encryptedGoods1 or y==encryptedGoods2)
                          then RECEIVED_EGOODS(x,y)
                          else PO_SENT(x)
```

The next step involves comparing the encrypted product received from the merchant with those downloaded from the third party. If the two do not match, the customer terminates the protocol.

```
RECEIVED_EGOODS(x,y) = if (x==y) then RECEIVED_CORRECT_GOODS
                       else ABORT
```

When the customer is satisfied with the encrypted product, he sends the payment token to the third party.

```
RECEIVED_CORRECT_GOODS = coutt !paymentToken -> TOKEN_SENT
```

After sending the payment, the customer waits for a message from the trusted third party. The third party either sends the customer the key or an abort message, depending on the outcome of the protocol. Once the customer has received the message from the third party, the protocol stops. Otherwise the customer continues to wait for the message.

```
TOKEN_SENT =  cint ?y ->
              if (y==key) then SUCCESS
              else (if y== transactionAborted) then ABORT
              else TOKEN_SENT
```

4.2.2 Modeling the Merchant Process

On the merchant side, the protocol begins with the merchant waiting to receive a purchase order from a customer.

```
MERCHANT = minc ?x -> if (x==po) then PO_REC
                      else MERCHANT
```

The merchant in response must send an encrypted product to the customer. The merchant can act in two ways – either he sends the correct encrypted product (denoted by `encryptedGoods1`) or an incorrect encrypted product (denoted by `encryptedGoods2`). This non-deterministic choice is modeled as follows:

```
PO_REC = (moutc !encryptedGoods1 -> ENCRYPTED_GOODS_SENT) |~|
                      (moutc !encryptedGoods2 -> ENCRYPTED_GOODS_SENT)
```

Once the merchant has sent the encrypted product, he must send the decryption key to the trusted third party.

```
ENCRYPTED_GOODS_SENT  = moutt !key -> KEY_SENT
```

After sending the key, the merchant waits to receive the payment token from the trusted third party. The third party either sends the payment token or a transaction abort message if the transaction was aborted. The merchant process terminates once it receives a message, otherwise it continues to wait for the message.

```
KEY_SENT = mint ?x -> if (x==paymentToken)
                      then SUCCESS
                      else if (x==transactionAborted)
                           then ABORT
                           else KEY_SENT
```

4.2.3 Modeling the Trusted Third Party

The customer downloading the encrypted product, is modeled from the third party's end, as the trusted third party sending the encrypted product to the customer.

```
TP = toutc !encryptedGoods1 -> WAIT_TOKEN_KEY
```

The next step involves the third party waiting to receive the payment token from the customer and the key from the merchant. When the third party receives a message it checks if the message is a payment token or key or neither. Note that, it is not known whether the key or payment token will arrive first. If the payment token arrives first, the third party must wait for the key. On the other hand, if the key arrives first, the third party must wait for the payment token. This aspect of the protocol is modeled as follows:

```
WAIT_TOKEN_KEY = (tinc ?a -> if (a==paymentToken) then WAIT_KEY(a)
                             else WAIT_TOKEN_KEY) []
                 (tinm ?b -> if (b==key) then WAIT_TOKEN(b)
                             else WAIT_TOKEN_KEY)

WAIT_KEY(a) = tinm ?b -> if (b==key) then CHECK_TOKEN(a,b)
                         else WAIT_KEY(a)

WAIT_TOKEN(b) = tinc ?a -> if (a==paymentToken) then CHECK_TOKEN(a,b)
                           else WAIT_TOKEN(b)
```

Once the third party has received both the key and the payment token, it proceeds to the next step of validating the payment token with the customer's financial institution. The details of the validation process is ouside the scope of the original protocol and, consequently, we omit these steps from the current model. Instead, the model non-deterministically chooses between the options: (i) token okay or (ii) token not okay.

If the payment token is okay, the trusted third party proceeds to send out the key to the customer and the token to the merchant. If the payment token is not okay an abort message is sent to the customer and the merchant, and the protocol terminates.

```
CHECK_TOKEN(a,b)  = OK_TOKEN(a,b) |~| NOK_TOKEN
OK_TOKEN(a,b) = SEND_TOKEN_KEY(a,b)
NOK_TOKEN = SEND_ABORT_MESSAGE
```

The process of sending an abort message to customer and merchant is modeled by the following step.

```
SEND_ABORT_MESSAGE = toutc !transAborted -> toutm !transAborted -> STOP
```

The processes of sending out the key to the customer and the token to merchant can occur in any order. Thus the sending out of key and token by the trusted third party is modeled as follows.

```
SEND_TOKEN_KEY(a,b) = toutc !b -> SEND_TOKEN(a) |~|  toutm !a -> SEND_KEY(b)
SEND_TOKEN(a) = toutm !a -> SUCCESS
SEND_KEY(b) = toutc !b -> SUCCESS
```

4.3 Modeling the Desirable Properties

In this section we describe how we model the properties of money atomicity, goods atomicity, and validated receipt.

4.3.1 Modeling Money Atomicity

Our protocol does not deal with the details of how money is actually transferred from the customer's account to the merchant's account. We assume that the merchant receiving the validated token is an acceptable form of payment. Note that, in reality the process is far more complex; the merchant probably has to deposit this token in a bank which will communicate with the customer's bank and debit customer's account and credit merchant's account. These issues and the details about methods of payments will be addressed in a future work.

The money atomicity property states that money is neither created nor destroyed in the process of an electronic transaction. This property is satisfied when the payment token, sent by the customer, is received by the merchant. However, the payment token may not always be received by the merchant. Consider, for example, the case when the payment token cannot be validated by the third party because of insufficent funds or some other error. In this case, the customer receives a transaction abort message which tells him that the transaction was aborted and the payment token was not forwarded to the merchant. Thus money atomicity is satisfied when one of the following things happen: (i) the customer sends the payment token and the merchant receives it or (ii) the customer sends the payment token and then receives a transaction abort message. This is modeled in CSP as follows:

```
SPEC1 = STOP |~| ((coutt.paymentToken -> mint.paymentToken -> STOP)
                 [] (coutt.paymentToken -> cint.transAborted -> STOP))
```

The next step in the verification of money atomicity involves forming a new process called SYSTEM1 by combining the merchant, the customer and the trusted third party processes with the appropriate communication processes, and hiding all the irrelevant events (that is, all events other than those specified in SPEC1). To verify that SYSTEM1 satisfies SPEC1 we need to check that SYSTEM1 is a failure/divergence refinement [18] of SPEC1. This check is done automatically by FDR, confirming that money atomicity is indeed satisfied by the protocol.

4.3.2 Modeling the Goods Atomicity Property

The goods atomicity property ensures that a customer receives the product if and only if he has paid. Note that both the correct encrypted product and the key are required by the customer in order to get the product. Thus in our model, the product comprises both the correct encrypted product and the key. The payment is made in the form of a token. The merchant receiving the token suffices as the customer having paid.

In the protocol, the customer may receive the correct encrypted product from the merchant (denoted by encryptedGoods1) or some incorrect one (denoted by encryptedGoods2). The goods atomicity property requires one of the following things to happen: (i) the customer receives both the correct encrypted product

and the keys and the merchant receives the token, or (ii) the customer receives just the encrypted product and neither the merchant gets the payment token nor the customer the keys. For (i), note that the customer may receive the key before the merchant receives the payment token or vice-versa. Thus goods atomicity is specified as follows:

```
SPEC2 = STOP |~| ((cinm.encryptedGoods1 -> STOP) []
                  (cinm.encryptedGoods2 -> STOP)  []
       (cinm.encryptedGoods1 -> cint.key -> mint.paymentToken -> STOP) []
        (cinm.encryptedGoods1 -> mint.paymentToken -> cint.key -> STOP))
```

We create another process called SYSTEM2 by combining the customer, merchant, third party and the communication processes and hiding the unrelated events (the events not associated with the specification of the goods atomicity property). FDR proves that SPEC2 is a failure divergence refinement of SYSTEM2 confirming that the protocol does indeed have goods atomicity.

4.3.3 Modeling the Validated Receipt Property

The validated receipt property ensures that the customer makes the payment only after he is satisfied that the product he is about to receive is the one that he is paying for.

In other words, the validated receipt property ensures one of the following things happen: (i) the customer receives some encrypted product and does not make payment (either because he has received incorrect product or decides not to purchase the product), or (ii) the customer makes the payment after receiving the correct encrypted product. This is modeled as follows:

```
SPEC3 = STOP |~| ((cinm.encryptedGoods2 -> STOP) []
                  (cinm.encryptedGoods1 -> STOP) []
              (cinm.encryptedGoods1 -> coutt.paymentToken -> STOP))
```

Next we create a process SYSTEM3 by combining the processes of the merchant, customer, third party and the communications processes and use FDR to check that SYSTEM3 is a failure divergence refinement of SPEC3. In this manner we are ensured that the protocol also has the validated receipt property.

5 Detecting the Violation of Properties in the Presence of Failures

Our protocol is not resilient to failures. If we allow the merchant, the customer, the third party and the communication medium to fail arbitrarily, the properties get violated. However there also exist some other failures that have no effect on the properties. In order to build a failure resilient protocol, we need to identify which failures result in the destruction of the properties. The model checker aids us in this process. In the following paragraphs we describe how we detect the failures that do not preserve the desirable properties of money atomicity, goods atomicity, and validated receipt.

5.1 Introducing Unreliability over the Communication Medium

The protocol given in Appendix A assumes that the channels through which the merchant, customer and the third party communicate are reliable. In other words, any data send by one party to another will eventually be received by the second party and is not lost in transmission.

Recall from Section 4.1 that the process modeling the reliable communication channel that transfers data from the customer to the merchant is given by:

```
COMMcm = []x: {po} @(coutm ?x -> (minc !x -> COMMcm))
```

In an unreliable channel the data may get lost. That is, data transmitted by the customer may not be received by the merchant. So after the event of the customer transmitting the data, one of two things may happen: (i) the data is lost or (ii) the merchant receives the data. Note that the choice of what is going to happen is not known and happens non-deterministically. Thus the unreliable channel is modeled as follows:

```
COMMcm = []x: {po} DATAcm @(coutm ?x -> (COMMcm |~|  (minc !x -> COMMcm)))
```

With this modification, we recheck the properties using FDR. It turns out that this unreliable channel has no effect on the property. Incrementally we introduce unreliability in the other channels as well and test whether the properties are preserved. Our experiments indicate that the properties are violated when the following channels are made unreliable: (i) the channels connecting the third party and the customer and (ii) those connecting the third party to the merchant.

5.2 Introducing Failure in the Customer, Merchant and Trusted Third Party Processes

The protocol given in Appendix A, does not, in general, allow the customer, merchant and third party process to abort. Now suppose we allow these processes to abort arbitrarily, will the properties still hold?

5.2.1 Introducing Failures in the Customer Process

First, let us consider the customer process given in Appendix A. We allow the customer process to abort only if he is not satisfied with the encrypted product he received. However, we believe that the customer should be able to abort at certain other points in the protocol, without putting the protocol to risk.

As an example, consider the first step for the customer process:

```
CUSTOMER = cint ?x -> DOWNLOADED_EGOODS(x)
```

Suppose we allow the customer to abort in this step (either willfully or because of a system crash). The question then is, does any property get violated? To find out, we need to model the possibility of the customer aborting in the first step:

```
CUSTOMER = ABORT |~| (cint ?x -> DOWNLOADED_EGOODS(x))
```

The above specification says that the customer may abort or wait for the downloading of the encrypted product in a non-deterministic manner. After making the above alteration to the customer process, we use FDR to check for the satisfaction of the properties. As expected, the customer aborting in the first step, has no effect on the properties.

Similarly, we make modifications to the specification to allow the customer process to abort in its second step (that is, the customer sending the purchase order) and check the properties. The modifications to the second step also does not result in the violation of the properties. We make similar modifications to each step in the customer process and check for the violation of the properties.

Our results indicate that such modification to any step, except in the last step of the customer process (that is after the customer has sent the payment token), preserves all the properties. Allowing the customer to abort in the last step violates both money atomicity and goods atomicity. To illustrate how the money atomicity property is violated, we use the FDR debugger which generates the following sequence of events.
`toutc.encryptedGoods1, cint.encryptedGoods1, coutm.po, minc.po, moutc.encryptedGoods1, moutt.key, tinm.key, cinm.encryptedGoods1, coutt.paymentToken, tinc.paymentToken, toutc.transAborted, toutm.transAborted, mint.transAborted`

The above sequence tells us that the following actions are executed. The customer downloads the encrypted product from the third party, then sends a purchase order to the merchant. On receiving the purchase order, the merchant sends the encrypted product to the customer and the key to the third party. The third party receives the key. The customer receives the encrypted product and validates it. The customer then sends the payment token to the third party. At this point, it appears that the customer aborts since we do not see any more messages sent or received by the customer. The third party receives the key from the merchant, the payment token from the customer, and then validates the token. The token turns out to be invalid and an abort message is sent by the third party to the customer and the merchant. Since the customer has aborted in the mean time, he does not get the transaction abort message from the third party. The merchant, however, receives the abort message. In the above scenario, the customer sends out the payment token, but neither the merchant received the payment token nor the customer the transaction abort message. Thus money atomicity is violated. Similarly, a counter example is generated illustrating how goods atomicity was violated. For the sake of brevity, we omit this counter example from the paper.

Thus, our conclusion is that, the customer cannot abort after sending out the payment token and before receiving the key; if the customer does indeed abort we will no longer have money atomicity or goods atomicity.

5.2.2 Introducing Failures in the Merchant Process

Next, we consider the merchant process. Like the customer process, the merchant process can abort in its first two steps without violating the properties. Allowing the merchant to abort in its third step, that is after the merchant has sent the key to the trusted third party results in the violation of money atomicity property. The counter-example generated by FDR is as follows.
`toutc.encryptedGoods1, cint.encryptedGoods1, coutm.po, minc.po, moutc.encryptedGoods1, cinm.encryptedGoods1, coutt.paymentToken`

The above trace indicates that the following actions were executed: The customer downloaded the encrypted product from the third party, and then sends the purchase order to the merchant. The merchant in response sends the encrypted product to the customer. The customer, after receiving the encrypted product, sends the payment token. Then it waits for a response from the third party. The merchant aborts after sending the encrypted product to the customer. The third party has received the payment token from the customer and is awaiting the key from the merchant who has already aborted. Thus, in this case, neither the merchant receives the payment token nor the customer, the abort message. Hence money atomicity is violated.

A minor modification to the protocol handles this problem. The third party associates a timeout event when waiting for responses from customer or merchant. If the third party is waiting for a long time, the time out event occurs and it sends an abort message to the customer and merchant.

```
WAIT_KEY(a) =  (tinm ?b -> if (b==key) then CHECK_TOKEN(a,b)
                           else WAIT_KEY(a))
               [] (timeOut -> SEND_ABORT_MESSAGE)

WAIT_TOKEN(b) = (tinc ?a -> if (a==paymentToken) then CHECK_TOKEN(a,b)
                            else WAIT_TOKEN(b))
                [] (timeOut -> SEND_ABORT_MESSAGE)
```

The above modification to the third party process allows the merchant to unilaterally abort in the third step without violating money atomicity.

However, allowing the merchant process to abort in the last step, that is, after sending the key but before receiving the payment token, violates both money atomicity and goods atomicity.

5.2.3 Introducing Failures in the Trusted Third Party Process

Finally, we consider the third party process. The third party process can abort only at its first step. In the rest of the protocol, the third party must be up and running. Note that the trusted third party is providing a service to its clients (customer and merchant) and so this is not an unrealistic demand on the third party.

6 Ensuring Failure Resilence of the Protocol

From the above discussion we can summarize that

1. The customer cannot abort (for any reason) after he has sent the payment token to the trusted third party.
2. The merchant cannot abort (for any reason) after he has sent the product decryption key to the trusted third party.
3. The trusted third party cannot abort unilaterally after its first step.

Note that the customer (merchant) will never make a conscious decision to abort after sending out the payment token (after sending out the product decryption key) as this is detrimental to the customer's (merchant's) interests. However, system failures may cause the customer (merchant) to abort. In this case, after system recovery, the customer (merchant) should be able to continue with the rest of the protocol; that is, the customer (merchant) should be able to receive the product decryption key (payment token) from the trusted third party. Note that this is equally true for communication link failure between the customer (merchant) and the trusted third party or failure of the trusted third party itself.

To ensure that the e-commerce protocol is resilient to failures we propose the following extension to the basic protocol. We assume that each party involved in the transaction, keeps a copy of the information that it sends to another party – for example purchase order, payment token and so on – in its stable storage till such time as the information is no longer needed. Writes to the stable storage are atomic and durable until intentionally purged.

1. The customer, the merchant and the trusted third party uses a system-wide unique identifier, T_i, to denote the current e-commerce transaction. The identifier is a tuple of the form $\langle\ PID,\ C,\ M\ \rangle$, where PID is the identifier for the product the customer, C purchases from the merchant, M. The customer stores a log record of the form $\langle\ T_i,\ INITIATE\ \rangle$ to its stable storage and then sends the purchase order to the merchant.

2. When the merchant receives the purchase order,it writes a log record $\langle\ T_i,\ INITIATE\ \rangle$ to its stable storage; then the merchant checks to see if the purchase order is to its satisfaction. If it is not, the merchant writes an abort record in its log – $\langle\ T_i,\ ABORT\ \rangle$ and aborts the transaction. It informs the customer of this decision. Otherwise it sends the encrypted product to the customer and the product decryption key and the approved purchase order to the trusted third party. Finally, it writes a log record to its stable storage of the form $\langle\ T_i,\ KEY-SENT\ \rangle$. At this stage the merchant enters a point of no return; it cannot abort unilaterally.

3. After receiving a message from the merchant the customer checks to see if it is an abort message or the encrypted product. If it is an abort, the customer aborts the transaction and writes a log record of the form $\langle\ T_i,\ ABORT\ \rangle$. Otherwise the customer validates the encrypted product. If validated, the customer sends the payment token and purchase order to the trusted third party and then writes a log record to its stable storage. The log record is of the form $\langle\ T_i,\ PAYMENT-SENT\ \rangle$. This is the point of no return for the customer. If the encrypted product is not validated the customer can either request the product from the merchant, or abort the transaction.

4. One of the messages - either the message containing the payment token and purchase order from the customer or the message containing the product decryption key and approved purchase order from the merchant - will arrive at the trusted third party before the other message. On receiving the message, the trusted third party associates the unique identifier T_i to this current transaction and writes a log record to its stable storage of the form $\langle\ T_i,\ INITIATE\ \rangle$. The third party starts a timer at this point. If the third party does not receive the other message before the timer expires, it writes a log record $\langle\ T_i,\ ABORT\ \rangle$ and sends abort messages to both the customer and the merchant

5. After receiving the payment token from the customer, the third party validates the token with the customer's financial institution. If the validation fails the third party writes a log record $\langle T_i, ABORT \rangle$ and informs both the customer and the merchant. Otherwise, after the third party has received both – the product decryption key from the merchant and the payment token from the customer – the third party sends the payment token to the merchant and writes a log record $\langle T_i, PAYMENT-FORWARDED \rangle$, and sends the decryption key to the customer and writes a log record $\langle T_i, KEY-FORWARDED \rangle$.

6. After the customer receives the product decryption key from the trusted third party and successfully decrypts the product, the customer writes the log record $\langle T_i, FINISH \rangle$. At this time the customer can remove the messages it sent to the other parties from its stable storage.

7. The merchant also writes a log record $\langle T_i, FINISH \rangle$, after it has received the payment token from the trusted third party. At this stage the merchant can also remove copies of the messages that it sent to other parties, from its stable storage.

6.1 Protocol Failure Analysis

Let us consider each of the possible failure scenarios individually and see why the protocol is failure resilient. Recall that we are interested in the cases after the customer has sent the payment token or the merchant has sent the product decryption key.

1. **Merchant fails after sending product decryption key but before writing log record** $\langle T_i, KEY-SENT \rangle$**.** After recovery from failure the merchant finds from its log that T_i has been initiated but the product decryption key has not been sent out (no information about the key having been sent is recorded). Consequently, it queries the third party to find out the status. If the status is abort, the merchant aborts. If the trusted third party has not received the key, the merchant resends the key and write the appropriate record. If the trusted third party cannot provide a status, the merchant resends the encrypted product to the customer, and the key and approved purchase order to the trusted third party and writes the appropriate log records. It then waits for the payment token from the trusted third party. Finally, as a result of the status query the merchant may receive the payment token. It then finishes by writing the appropriate log record.

2. **Merchant fails after writing log record** $\langle T_i, KEY-SENT \rangle$**.** After recovery from failure, the merchant finds that it has not received the payment token. It asks the third party for the payment token. The third party responds either by sending the payment token or an abort message. If it is an abort message, the merchant write $\langle T_i, ABORT \rangle$ in its stable storage and aborts the transaction. If payment token is received the merchant writes $\langle T_i, FINISH \rangle$ to log.

3. **Merchant fails before writing** $\langle T_i, FINISH \rangle$**.** After recovery, merchant assumes that payment token has not been received. This is then equivalent to the scenario merchant failing after writing log record $\langle T_i, KEY-SENT \rangle$.

4. **Customer fails after sending payment token but before writing log record** $\langle\ T_i,\ PAYMENT-SENT\ \rangle$**.** After recovery, the customer notes from log that T_i has been initiated but no other information (such as, information about the product received or payment token sent) is recorded in the log. The customer, in this case, gets in touch with the merchant and asks for the product. The merchant either sends the encrypted product or an abort message. If the customer receives the encrypted product, the customer validates it, sends the payment token and writes the appropriate log record.

5. **Customer fails after writing log record** $\langle\ T_i,\ PAYMENT-SENT\ \rangle$**.** After recovery the customer notes that the decryption key has not been received. So it requests the trusted third party for the product decryption key. The trusted third part responds with either an abort message or the decryption key. If it is an abort message, the customer writes $\langle\ T_i,\ ABORT\ \rangle$ to its log and aborts. If it is the decryption key, the customer writes $\langle\ T_i, FINISH\ \rangle$ to the log and finishes.

6. **Customer fails before writing** $\langle\ T_i, FINISH\ \rangle$**.** After recovery, customer assumes that it has not received the product decryption key from the trusted third party. This is then equivalent to the scenario customer failing after writing log record $\langle\ T_i,\ PAYMENT-SENT\ \rangle$.

7. **Trusted third party fails before writing log record** $\langle\ T_i,\ INITIATE\ \rangle$**.** At this stage the trusted third party is not aware of the transaction T_i. Consequently the trusted third party does nothing. At some point of time either the customer or the merchant will get in touch asking for the product decryption key or a status query. At this stage the trusted third party will write the log record $\langle\ T_i,\ INITIATE\ \rangle$ and ask the customer for the payment token and purchase order, and the merchant for the product decryption key and the approved purchase order. It then starts the timer.

8. **Trusted third party fails after writing** $\langle\ T_i,\ INITIATE\ \rangle$**.** After recovery the trusted third party notes that T_i has been initiated. It asks the customer for the payment token and purchase order, and the merchant for the product decryption key and the approved purchase order. Once the third party has received a response from the customer or the merchant, it starts the timer and waits for the other message.

9. **Trusted third party fails before writing either log records** $\langle\ T_i,\ PAYMENT-FORWARDED\ \rangle$ **or** $\langle\ T_i,\ KEY-FORWARDED\ \rangle$**.** This the same scenario as the third party failing after writing $\langle\ T_i,\ INITIATE\ \rangle$. So it will take the steps corresponding to this situation.

10. **Trusted third party fails after writing one of the records** $\langle\ T_i,\ PAYMENT-FORWARDED\ \rangle$ **or** $\langle\ T_i,\ KEY-FORWARDED\ \rangle$ **but before writing the other.** After recovery the third party sends the message that was not sent out prior to failure and writes the appropriate record.

6.2 Discussion

In the modified protocol we have assumed that the trusted third party keeps indefinitely, a copy of all information exchanged with itself during the transaction, as well as the log records it writes during the transaction.

On the other hand, the customer and the merchant are able to discard such messages after each has written the $\langle T_i, FINISH \rangle$ record in its log. The trusted third party is required to keep this messages because the customer and the merchant do not communicate to the third party the successful completion of their respective portions of the transaction. This approach has two advantages:

1. it reduces the number of messages exchanged in the protocol, and
2. it reduces the involvement of the trusted third party in the protocol.

However, in practice, it will not be possible for the third party to store copies of all messages exchanged in such transactions for an indefinite period of time.

This problem can be solved in a number of ways.

Approach 1: One solution is that the third party keeps a record of the transactions for a specified amount of time which is known to the customer and the merchant. For the third party to respond to the queries of the customer (merchant) regarding the transaction, the customer (merchant) must place the query within the specified time period.

Approach 2: After the customer has written the log record $\langle T_i, FINISH \rangle$, the customer sends a acknowledgment to the trusted third party. On receiving this message the trusted third party writes a log record $\langle T_i, CUSTOMER-ACK \rangle$. The merchant also does likewise and the trusted third party writes $\langle T_i, MERCHANT-ACK \rangle$ in its log. After both records have been written, the trusted third party discards copies of the messages.

If the trusted third party fails before receiving one or both acknowledgments, it will ask the customer and/or the merchant about the status of transaction T_i at the respective sides.

Approach 3: When the trusted third party is ready to expunge transaction messages it approaches the customer and the merchant for the status of transaction T_i at their respective sides. Depending on the status the trusted third party writes the log records $\langle T_i, CUSTOMER-ACK \rangle$ and/or $\langle T_i, MERCHANT-ACK \rangle$. After both records have been written, the trusted third party discards copies of the messages.

Note that, both approach 2 and 3 require the protocol to be augmented by an extra round of messages.

7 Conclusion and Future Work

In this paper we have illustrated how model checking can be used to get assurance that an e-commerce protocol does satisfy the properties of money atomicity, goods atomicity, and validated receipt. We have also shown how model checking can be used to detect violation of properties in the presence of site and communication failures. Using the analysis, we proposed a mechanism that preserves the properties even in the event of sites or communications failures.

A number of issues still remain to be investigated. In this paper our focus was on the non-security aspects of the e-commerce protocol. In future we plan to investigate the security issues of the e-commerce protocol using model checking. Specifically, we need to investigate whether our protocol is prone to attacks.

This paper considered a single run of the protocol involving one merchant and one customer. In real life, a number or merchants and customers will be executing the protocol, concurrently. We plan to model multiple runs of the protocol using multiple merchants and customers and see whether properties hold or not. We also plan to investigate the security issues involved in multiple runs of the protocol.

In this protocol, we assume that the customer provides the third party with a payment token who validates it and then forwards it to the merchant. However, in reality, the process is more complex. For example, the customer's bank and the merchant's bank may also be involved and each of these entities and the communication channels are prone to failures. We need to model these aspects in details and verify whether money atomicity holds or not in this scenario. We also need to investigate how the many different forms of electronic payment schemes that are available today [6, 20] can be incorporated into our protocol and the resulting solutions be verified for the existence of the properties.

In future, we plan to improve upon the basic protocol. One such improvement involves reducing the involvement of the trusted third party. Not only is the performance of the trusted third party an issue, but also its vulnerability to denial of service attacks. We plan to investigate two approaches to reduce this problem. The first approach involves modifying the protocol to reduce the interactions with the third party. The second approach involves distributing the multiple roles played by the trusted third party over a number of (possibly) semi-trusted third party.

References

[1] J. M. Atlee and J. D. Gannon. State-based Model Checking of Event Driven Systems Requirements. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[2] D. Bolignano. An Approach to the Formal Verification of Cryptographic Protocols. In *Proceedings of the 3rd ACM Conference on Computer and Communications Security, New Delhi, India*, pages 106–118. ACM Press, March 1996.

[3] D. Bolignano. Towards the Formal Verification of Electronic Commerce Protocols. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, June 1997.

[4] B. Cox, J. D. Tygar, and M. Sirbu. NetBill Security and Transaction Protocol. In *Proceedings of the First USENIX Workshop in Electronic Commerce*, pages 77–88, July 1995.

[5] A. Fiat D. Chaum and M. Naor. "untraceable electronic cash". In *Advances in Cryptology – Proceedings of CRYPTO '88*, pages 200–212. Springer-Verlag, 1990.

[6] S. Dukach. SNPP: A Simple Network Payment Protocol. Technical report, MIT Laboratory for Computer Science, 1992. Available from `ftp://ana.lcs.mit.edu/pub/snpp/snpp-paper.ps`.

[7] B. Dutertre and S. Schneider. Using a PVS Embedding of CSP to Verify Authentication Protocols. In *Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, 1997.

[8] Formal Systems (Europe) Ltd. *Failure Divergence Refinement - FDR2 User Manual*, version 2.64 edition, August 1999.

[9] N. Heintze, J. Tygar, J. Wing, and H. Wong. Model Checking Electronic Commerce Protocols. In *Proceedings of the 2nd USENIX Workshop in Electronic Commerce*, pages 146–164, November 1996.

[10] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-key Protocol Using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems: Second International Workshop, TACAS 96* , pages 147–166, 1996.

[11] G. Lowe. Some New Attacks Upon Security Protocols. In *Proceedings of the 1996 IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1996.

[12] G. Lowe and A. W. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, 23:659–669, 1997.

[13] W. Marrero, E. Clarke, and S. Jha. A Model Checker for Authentication Protocols. In *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols.* Rutgers University, NJ, September 1997.

[14] J. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murphi. In *Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA*, pages 141–151, 1997.

[15] L. C. Paulson. Proving Properties of Security Protocols by Induction. Technical Report TR409-lcp, Computer Laboratory, University of Cambridge, December 1996. Available from `http://www.cl.cam.ac.uk/ftp/papers/reports/`.

[16] I. Ray, I. Ray, and N. Narasimhamurthy. A Fair-Exchange Protocol with Automated Dispute Resolution. Technical report, University of Michigan-Dearborn, 1999. Submitted for publication.

[17] A. W. Roscoe. Proving Security Protocols with Model Checkers by Data Independence Techniques. In *Proceedings of the 1998 IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, 1998.

[18] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, Great Britain, 1998.

[19] M. Sirbu and J. D. Tygar. NetBill: An Internet Commerce System Optimized for Network Delivered Services. *IEEE Personal Communications*, pages 34–39, August 1995.

[20] L. H. Stein, E. A. Stefferud, N. S. Borenstein, and M. T. Rose. The Green Commerce Model. Technical report, First Virtual Holdings Incorporated, 1994. Available from `http://www.fv.com/tech/green-model.html`.

[21] J. D. Tygar. Atomicity in Electronic Commerce. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 8–26, May 1996.

Appendix A – Initial Specification of the Protocol

```
-- This is the first specification of the protocol that
-- verifies money atomicity, goods atomicity and validated receipt
-- The protocol assumes that the merchant, customer or the
-- trusted third party processes do not abort at any step
-- in the protocol. Further it assumes that communication
-- channels are reliable.

datatype KEY = key
datatype MESSAGE = transAborted
datatype PO = po
datatype ENCRYPTEDGOODS = encryptedGoods1|encryptedGoods2
datatype TOKEN = paymentToken

-- Channel declarations
channel coutm : {po}
channel moutc : {encryptedGoods1,encryptedGoods2}
channel moutt : {key}
channel toutc : {key,encryptedGoods1,encryptedGoods2,transAborted}
channel toutm : {paymentToken,transAborted}
channel minc : {po}
channel tinc : {paymentToken}
channel cinm : {encryptedGoods1,encryptedGoods2}
channel tinm : {key}
channel cint : {key,encryptedGoods1,encryptedGoods2,transAborted}
channel mint : {paymentToken,transAborted}
channel coutt : {paymentToken }


-- The customer process

CUSTOMER = cint ?x -> DOWNLOADED_EGOODS(x)

DOWNLOADED_EGOODS(x) = coutm !po -> PO_SENT(x)

PO_SENT(x) = cinm ?y  ->  if (y==encryptedGoods1 or y==encryptedGoods2)
                          then RECEIVED_EGOODS(x,y)
                          else PO_SENT(x)
```

```
RECEIVED_EGOODS(x,y) =  if (x==y) then RECEIVED_CORRECT_GOODS
                        else ABORT

RECEIVED_CORRECT_GOODS = coutt !paymentToken -> TOKEN_SENT

TOKEN_SENT =  cint ?y ->
              if (y==key) then SUCCESS
              else if (y==transAborted) then ABORT
                   else TOKEN_SENT


--The merchant process

MERCHANT = minc ?x -> if (x==po) then PO_REC
                      else MERCHANT

PO_REC =  (moutc !encryptedGoods1 -> ENCRYPTED_GOODS_SENT) |~|
              (moutc !encryptedGoods2 -> ENCRYPTED_GOODS_SENT)

ENCRYPTED_GOODS_SENT = moutt !key -> KEY_SENT

KEY_SENT = mint ?x -> if (x== paymentToken) then SUCCESS
                      else if (x== transAborted) then ABORT
                      else KEY_SENT


--The Trusted Third Party Process

TP = toutc !encryptedGoods1 -> WAIT_TOKEN_KEY

WAIT_TOKEN_KEY = (tinc ?a -> if (a==paymentToken) then WAIT_KEY(a)
                                 else WAIT_TOKEN_KEY) |~|
                 (tinm ?b -> if (b==key) then WAIT_TOKEN(b)
                                 else WAIT_TOKEN_KEY)

WAIT_KEY(a) = tinm ?b -> if (b==key) then CHECK_TOKEN(a,b)
                         else WAIT_KEY(a)

WAIT_TOKEN(b) = tinc ?a -> if (a==paymentToken) then CHECK_TOKEN(a,b)
```

```
                              else WAIT_TOKEN(b)

CHECK_TOKEN(a,b)  = OK_TOKEN(a,b) |~| NOK_TOKEN

OK_TOKEN(a,b) = SEND_TOKEN_KEY(a,b)

NOK_TOKEN = SEND_ABORT_MESSAGE

SEND_ABORT_MESSAGE = toutc !transAborted -> toutm !transAborted -> STOP

SEND_TOKEN_KEY(a,b) = toutc !b -> SEND_TOKEN(a) |~| toutm !a -> SEND_KEY(b)

SEND_TOKEN(a) = toutm !a -> SUCCESS

SEND_KEY(b) = toutc !b -> SUCCESS


SUCCESS = STOP
ABORT = STOP

COMMcm = []x: {po} @(coutm ?x -> (minc !x -> COMMcm))
COMMct = []x: {paymentToken} @(coutt ?x ->  (tinc !x -> COMMct))
COMMmc = []x: {encryptedGoods1,encryptedGoods2} @(moutc ?x
                                        -> (cinm !x -> COMMmc))
COMMmt = []x: {key} @(moutt ?x ->  (tinm !x -> COMMmt))
COMMtc = []x: {key,encryptedGoods1,encryptedGoods2,transAborted} @(toutc ?x
                                       ->  (cint !x -> COMMtc))
COMMtm = []x: {paymentToken,transAborted}@(toutm ?x ->  (mint !x -> COMMtm))

COMM = (((( COMMct [|{}|] COMMmt) [|{}|]
             COMMtc) [|{}|] COMMtm) [|{}|] COMMcm) [|{}|] COMMmc

CIO = {| coutm, coutt, cinm, cint |}
MIO = {| moutt, moutc, minc, mint |}
TIO = {| toutm, toutc, tinm, tinc |}


COMMIO = union(CIO, union (MIO, TIO))
```

```
SYSTEM1 = (((CUSTOMER [|{}|] MERCHANT) [|{}|] TP) [|COMMIO|] COMM) \
         diff(COMMIO,{coutt.paymentToken,cint.transAborted, mint.paymentToken})

SYSTEM2 = (((CUSTOMER [|{}|] MERCHANT) [|{}|] TP) [|COMMIO|] COMM) \
        diff(COMMIO,{cinm.encryptedGoods1,cint.key,mint.paymentToken})

SYSTEM3 = (((CUSTOMER [|{}|] MERCHANT) [|{}|] TP) [|COMMIO|] COMM) \
diff(COMMIO,{cinm.encryptedGoods1,cinm.encryptedGoods2,coutt.paymentToken})

-- This is the money atomicity property

SPEC1 = STOP |~| ((coutt.paymentToken -> mint.paymentToken -> STOP)
                 [] (coutt.paymentToken -> cint.transAborted -> STOP))

-- This is the goods atomicity property

SPEC2 = STOP |~|
  ((cinm.encryptedGoods1 -> STOP) [] (cinm.encryptedGoods2 -> STOP)
 [] ( cinm.encryptedGoods1 -> cint.key -> mint.paymentToken -> STOP)
 [] ( cinm.encryptedGoods1 -> mint.paymentToken -> cint.key -> STOP))


-- This is the validated receipt property

SPEC3 = STOP |~|
      ((cinm.encryptedGoods2 -> STOP) [] (cinm.encryptedGoods1 -> STOP) []
       (cinm.encryptedGoods1 -> coutt.paymentToken -> STOP))
```

Appendix B – Specification with the Introduction of Failures

```
-- This is the modified specification of the  protocol that
-- assures money atomicity, goods atomicity and validated receipt
-- in the face of failures.

datatype KEY = key
datatype PO = po
datatype EGOODS = encryptedGoods1|encryptedGoods2
datatype TOKEN = paymentToken
datatype MESSAGE = transAborted

-- Channel declarations
channel coutm : {po}
channel moutc : {encryptedGoods1,encryptedGoods2}
channel moutt : {key}
channel toutc : {key,encryptedGoods1,encryptedGoods2,transAborted}
channel toutm : {paymentToken,transAborted}
channel minc : {po}
channel tinc : {paymentToken }
channel cinm : {encryptedGoods1,encryptedGoods2}
channel tinm : {key}
channel cint : {key,encryptedGoods1,encryptedGoods2,transAborted}
channel mint : {paymentToken,transAborted}
channel coutt : {paymentToken }

channel timeOut

-- The customer process

CUSTOMER = ABORT |~| ( cint ?x -> DOWNLOADED_EGOODS(x))

DOWNLOADED_EGOODS(x) = ABORT |~| ( coutm !po -> PO_SENT(x))

PO_SENT(x) = ABORT |~|
              (cinm ?y  ->  if (y==encryptedGoods1 or y==encryptedGoods2)
                            then RECEIVED_EGOODS(x,y)
                            else PO_SENT(x))

RECEIVED_EGOODS(x,y) =
```

```
 ABORT |~| (if (x==y) then  RECEIVED_CORRECT_GOODS
                        else ABORT)

RECEIVED_CORRECT_GOODS = ABORT |~| (coutt !paymentToken -> TOKEN_SENT)

TOKEN_SENT =  cint ?y ->
              if (y==key) then SUCCESS
              else if (y==transAborted) then ABORT
                       else TOKEN_SENT


--The merchant process


MERCHANT = ABORT |~| ( minc ?x -> if (x==po) then PO_REC
            else MERCHANT)

PO_REC = ABORT |~| (moutc !encryptedGoods1 -> ENCRYPTED_GOODS_SENT) |~|
           (moutc !encryptedGoods2 -> ENCRYPTED_GOODS_SENT)

ENCRYPTED_GOODS_SENT =  ABORT |~| (moutt !key -> KEY_SENT)

KEY_SENT = mint ?x -> if (x== paymentToken) then SUCCESS
                        else if (x==transAborted) then ABORT
                           else KEY_SENT


--The Trusted Third Party Process


TP =  ABORT |~| (toutc !encryptedGoods1 -> WAIT_TOKEN_KEY)

WAIT_TOKEN_KEY = (tinc ?a -> if (a==paymentToken) then WAIT_KEY(a)
                                 else WAIT_TOKEN_KEY) []
                 (tinm ?b -> if (b==key) then WAIT_TOKEN(b)
                                 else WAIT_TOKEN_KEY)

WAIT_KEY(a) =  (tinm ?b -> if (b==key) then CHECK_TOKEN(a,b)
                         else WAIT_KEY(a))
```

```
                [] (timeOut -> SEND_ABORT_MESSAGE)

WAIT_TOKEN(b) = (tinc ?a -> if (a==paymentToken) then CHECK_TOKEN(a,b)
                            else WAIT_TOKEN(b))
                 [] (timeOut -> SEND_ABORT_MESSAGE)

CHECK_TOKEN(a,b)  =  OK_TOKEN(a,b) |~| NOK_TOKEN

OK_TOKEN(a,b) =  SEND_TOKEN_KEY(a,b)

NOK_TOKEN = SEND_ABORT_MESSAGE

SEND_ABORT_MESSAGE = toutc !transAborted -> toutm !transAborted -> STOP

SEND_TOKEN_KEY(a,b) =  toutc !b -> SEND_TOKEN(a) |~| toutm !a -> SEND_KEY(b)

SEND_TOKEN(a) = toutm !a -> SUCCESS

SEND_KEY(b) = toutc !b -> SUCCESS


SUCCESS = STOP
ABORT = STOP


COMMcm = []x: {po} @(coutm ?x -> (COMMcm |~|  (minc !x -> COMMcm)))
COMMct = []x: {paymentToken} @(coutt ?x -> (tinc !x -> COMMct))
COMMmc = []x: {encryptedGoods1,encryptedGoods2}@(moutc ?x ->
                                ( COMMmc |~| (cinm !x -> COMMmc)))
COMMmt = []x: {key} @(moutt ?x -> (COMMmt |~| (tinm !x -> COMMmt)))
COMMtc = []x: {key,encryptedGoods1,encryptedGoods2,transAborted}@(toutc ?x
                                ->  (cint !x -> COMMtc))
COMMtm = []x: {transAborted,paymentToken}@(toutm ?x ->  (mint !x -> COMMtm))

COMM = (((( COMMct [|{}|] COMMmt) [|{}|]
             COMMtc) [|{}|] COMMtm) [|{}|] COMMcm) [|{}|] COMMmc

CIO = {| coutm, coutt, cinm, cint |}
MIO = {| moutt, moutc, minc, mint |}
```

```
TIO = {| toutm, toutc, tinm, tinc |}

COMMIO = union(CIO, union (MIO, TIO))

SYSTEM1 = (((CUSTOMER [|{}|] MERCHANT) [|{}|] TP) [|COMMIO|] COMM) \
             union(diff(COMMIO,{coutt.paymentToken,mint.paymentToken,
                   cint.transAborted}),{timeOut})

SYSTEM2 = (((CUSTOMER [|{}|] MERCHANT) [|{}|] TP) [|COMMIO|] COMM) \
             union(diff(COMMIO,{cinm.encryptedGoods1,encryptedGoods2,
                   cint.key,mint.paymentToken}),{timeOut})

SYSTEM3 = (((CUSTOMER [|{}|] MERCHANT) [|{}|] TP) [|COMMIO|] COMM) \
              union(diff(COMMIO,{|cinm,coutt|}),{timeOut})


-- This is the money atomicity property


SPEC1 = STOP |~| (coutt.paymentToken -> ((cint.transAborted -> STOP) |~|
 (mint.paymentToken -> STOP)))

-- This is the goods atomicity property


SPEC2 = STOP |~| ((cinm.encryptedGoods1 -> STOP) []
      (cinm.encryptedGoods2 -> STOP) []
( cinm.encryptedGoods1 -> cint.key -> mint.paymentToken -> STOP)
 [] ( cinm.encryptedGoods1 -> mint.paymentToken -> cint.key -> STOP))

-- This is the validated receipt property

SPEC3 = STOP |~| ((cinm.encryptedGoods1 -> STOP)
                 [] (cinm.encryptedGoods2 -> STOP)
                 [] (cinm.encryptedGoods1 -> coutt.paymentToken -> STOP))
```