Model Checking Electronic Commerce Protocols

Nevin Heintze
Bell Laboratories
Murray Hill, NJ 07974
nch@research.bell-labs.com

J. D. Tygar
Carnegie Mellon Univ.
Pittsburgh, PA 15213
tygar@cs.cmu.edu

Jeannette Wing
Carnegie Mellon Univ.
Pittsburgh, PA 15213
wing@cs.cmu.edu

H. Chi Wong
Carnegie Mellon Univ.
Pittsburgh, PA 15213
hcwong@cs.cmu.edu

Abstract

The paper develops model checking techniques to examine NetBill and Digicash. We show how model checking can verify atomicity properties by analyzing simplified versions of these protocols that retain crucial security constraints. For our analysis we used the FDR model checker.

1 Atomicity Properties

Correctness is a prime concern for electronic commerce protocols. How can we show that a given protocol is safe for use? Here we show how to use model checking to test whether electronic commerce protocols satisfy some given atomicity properties.

For verifying properties of protocols, model checking is a dramatic improvement over doing hand proofs, because it is mechanizable; it is a dramatic improvement over using state-of-the-art theorem provers because it is automatic, fast, and requires no human interaction. Moreover, we found a number of problems in proposed electronic commerce protocols using model checking. Model checking allows us to focus on just those aspects of the protocol necessary to guarantee desired properties. In doing so, we can gain a better understanding of why the protocol works and often can identify places of optimizing it.

For this paper, we have chosen to check atomicity properties. [2] argue that these properties are central to electronic commerce protocols.

In an *atomic* protocol, an electronic purchase either

- aborts with no transfer of money and goods; or
- fully completes with money and goods exchanged.

Moreover, these atomic properties are preserved even if communications fail between some of the parties, because of failure of either a communications link or a node (including the parties participating in the protocol.)

Tygar [22] gave informal descriptions of three protocol properties that appear to be related to atomicity:

money atomicity Money should neither be created nor destroyed by electronic commerce protocols. For example, this protocol is not money atomic:

1. Consumer sends message to consumer's bank: *transfer $value to merchant*;
2. Consumer's bank decrements consumer's balance by *$value*;
3. Consumer's bank sends message to merchant's bank: *increase merchant's bank balance by $value*;
4. Merchant's bank increments merchant's balance by *$value*.

If Message 3 is not received, then the consumer's balance will have lost money without the merchant's bank having received the money. Effectively, money is destroyed.

goods atomicity A merchant should receive payment if and only if the consumer receives the goods. Goods atomicity is particularly relevant in the case of electronic goods (such as binary files) that are delivered over the network. For example this protocol is not goods atomic:

1. Consumer sends credit card number to merchant;
2. Merchant charges consumer's credit card;

This work was supported in part by Defense Advanced Research Project Agency (ARPA contract F33615-93-1-1330), the National Science Foundation (NSF cooperative agreement IR-9411299), and by the US Postal Service. This work is the opinion of the authors and does not necessarily represent the view of their employers, funding sponsors, or the US Government.

3. Merchant sends electronic goods to consumer

Suppose Message 3 is not received by the consumer; then she will not have received the goods for which she was charged.

certified delivery In the case of electronic goods, both the merchant and the consumer should be able to give non-repudiable proof of the contents of the delivered goods. (We do not consider certified delivery in this paper.)

In this paper, we discuss how to use model checking to determine whether money atomicity and goods atomicity hold of two classes of electronic commerce protocols: account-based (e.g., NetBill [5, 21]) and token-based (e.g., a simplified protocol inspired by Chaum's offline Digicash protocol [3, 4]). We used the FDR model checker [15], though other model checkers could have been used.

2 Model Checking

Model checking is a technique that determines whether a property holds of a finite state machine. The expression of the property and the machine could be in different languages where the property is a logical formula and the machine is described as a set of states and a state transition function. Since the machine has a finite number of states, an exhaustive search (through a standard reachability algorithm) is done to check that the logical formula holds at every state. The model checker SMV uses this approach. Alternatively, as with FDR, the expression of the property and the machine could be in the same language. In this case, a test for language inclusion is done to determine whether the property (one set of traces) holds of the machine (the second set of traces). Model checking is completely automatic, and usually fast, at least in comparison to alternative techniques like theorem proving.

An additional benefit that model checking provides over other techniques such as theorem proving is that if the property being checked does not hold, a counterexample is produced. This feedback is an invaluable means of debugging. It also is a way to explore the design space, perhaps finding ways to optimize a design (e.g., by eliminating a message exchanged or an extra encryption step).

In our context of using FDR as our model checker, we describe the protocol that we wish to analyze through a CSP [10] process, Imp. This is the "machine" we want to check. We describe the property, e.g., money atomicity, that we wish to check of the protocol as another CSP process, Spec. To determine whether the protocol satisfies the property we test whether Imp is a *refinement* (in the technical sense used in CSP) of Spec. Roughly speaking, Imp's set of traces should be a subset of Spec's set of traces; thus, the machine is a trace refinement of the property. In reality, for atomicity properties, we need the failure refinement, which extends the trace refinement to handle non-determinism.

Model checking is a demonstrated success in hardware verification. Researchers and industrialists have used checkers like SMV [17], Murphi [6], COSPAN [9] and SPIN [11] to find bugs in published circuit designs, floating point standards, and cache coherence protocols for multiprocessors. It has been adopted by the hardware community to complement the traditional validation method of hardware simulation.

Model checking has also recently gained the attention of the software community. Notably, Atlee and Gannon used SMV to check safety properties for event-driven systems; Jackson uses his model enumeration method in Nitpick to check Z-like specifications [12]; and Vaziri-Farahani and Wing used SMV to check cache coherence in distributed file systems [23].

In the security domain, Roscoe [19] used FDR to prove noninterference of a simple security high-low hierarchy (after Bell-LaPadula's model) and Lowe [14] recently used FDR to debug and prove the correctness of the Needham-Schroeder authentication protocols [18]. Our work is the first to use model checking for analyzing electronic commerce protocols and for checking atomicity properties[1].

3 Two Case Studies

We investigate NetBill and a simplified digital cash protocol with respect to money atomicity and goods atomicity. Money atomicity is concerned with the conservation of money in the context of account balances and electronic coins. That is, electronic coins should not be arbitrarily created or destroyed, and

[1]Use of formal methods to demonstrate protocol security has attracted wide attention – we cannot survey all the results in a brief paper such as this. However, electronic commerce protocols have received less attention – two important papers on formal (non-model checking) methods for electronic commerce are [1, 13].

fund transfers and conversions between funds and coins should be consistent. In other words, if we have a system formed by a consumer C and a merchant M who have accounts at a bank, the sum given by the formula

$$\begin{aligned} &\text{C's account balance} + \text{C's coins} \\ &+ \text{M's account balance} + \text{M's coins} \end{aligned} \qquad (1)$$

should be conserved. In the context of electronic coins, another component of money atomicity is that rightful possession of a coin should entitle the owner to spend the coin or deposit it in an account. This "cash property" will play an important role in our analysis of a simplified digital cash protocol.

Goods atomicity is concerned with the integrity of a sale: we want to guarantee that goods are transferred exactly when money is transferred. A consumer only wants to pay for goods received. A merchant wants to be payed for goods delivered.

Assumptions

Our analysis focuses on the atomicity aspects of protocols. We do not, for example, consider the cryptographic details of the protocol—in fact our modeling of a protocol completely hides these details. We also do not model multiple interleaved runs of a protocol (in which, for example, a single agent could participate as a consumer in one run and a merchant in another). Instead we consider a single run of the protocol with one consumer, one merchant, and one bank. We discuss these abstractions further in Section 4.

Perhaps the most important assumption we make is about the failure model used in our analysis. First, consider the bank. In the context of bank failure, few if any atomicity properties can be guaranteed. In practice, banks go to great lengths to ensure reliable, fail-safe service. We model this by assuming that the bank never fails. Next, consider communication with the bank. This may take place over some unreliable medium such as a telephone line or the Internet. However, as a last resort, anyone can physically go to the bank to deposit funds or present purchase orders. In effect, every agent has a fail-safe communication line with the bank.

Now consider agents other than the bank. We allow communication between non-bank agents to fail arbitrarily. However, we only allow limited failures at non-bank agents because arbitrary failure would compromise atomicity properties. For example, suppose that a merchant receives an electronic coin in exchange for goods, and then immediately fails before depositing the coin at the bank. The coin is effectively lost and money atomicity fails. Note, however, that the only party to suffer was the party that failed; there is no loss to the consumer nor the bank.

Our failure model for agents, other than banks, will be based upon the notion of commitment points, as used in standard database transactions [7, 16, 8]. We assume that each agent (other than the bank) has a particular point in the protocol at which that agent commits. Before this point is reached, we allow an agent to abort the protocol freely. After the commitment point, we consider only failures in an agent if the failure can potentially affect the outcome of the protocol for another agent. In particular, we ignore failures that can affect only the agent's own outcome. In Section 5 we outline a more comprehensive failure model that expands these ideas.

3.1 NetBill

NetBill [5, 21] is designed to support very low-cost transactions involving electronic goods. One central and distinguished claim of the NetBill protocol is that it satisfies goods atomicity, and this will be the focus of our analysis. In NetBill, all money-related activities are centralized at the bank and take the form of transfers between accounts; consequently, arguing money atomicity is straightforward.

Here, we use an abstracted and simplified version of the protocol that captures atomicity properties. For full details on the protocol see [5, 22].

The Protocol

The consumer C starts the protocol (Figure 1) by sending the merchant M a goods request, to which M responds with the goods encrypted with a one-time key K. At step 3, C sends M an electronic payment order (EPO) signed with C's private key. This EPO constitutes a fund transfer authorization, and sending it to M marks C's commit point. M checks the validity of this EPO, endorses it, appends K to it, and sends it to the bank B. This is the point where M commits to the transaction. Including K with the endorsed EPO is central to ensuring goods atomicity. At step 5, B sends to M a receipt of the fund transfer (which includes K). Then M forwards this message to C. In case M does not forward the message (either because of failure, bad management,

1. $C \to M$: *goods request*
2. $M \to C$: *goods, encrypted with a key K*
3. $C \to M$: *signed EPO (electronic payment order)*
4. $M \to B$: *endorsed signed EPO, signed K*
5. $B \to M$: *signed receipt (including K)*
6. $M \to C$: *signed receipt (including K)*

If C does not receive the signed receipt, C may contact B directly:

7. $C \to B$: *transaction inquiry*
8. $B \to C$: *signed receipt*

Figure 1: The simplified NetBill protocol.

or attempted fraud), C can go to the bank for a copy of this message, and hence obtain K[2].

NetBill in FDR

To model NetBill, we view each agent as a finite state machine, and use CSP processes to encode them. A CSP process denotes a set of sequences of events, where an event represents a state transition of the state machine; states are implicit.

Figures 2, 3, and 4 present simplified versions of the consumer, the merchant, and the bank processes respectively. Note that `->` is a form of sequential composition, `||` and `[]` are choice operators (with `||`, the choice is completely arbitrary, whereas `[]` gives preference to unblocked processes), `STOP` denotes process termination, `!` and `?` are the communication primitives (for example, `coutm!goodsReq` sends the message `goodsReq` on the channel `coutm`, and `cinm?x` receives a message from channel `cinm`, and binds the variable `x` to the message).

CSP uses a synchronous model of communication between processes. Since we are modeling a distributed system, we need to simulate asynchronous communication. For communication from agent `a` to agent `b`, we use two channels `aoutb` and `bina`, and a process that reads anything from the first channel and writes it to the second. This process can be easily modified to introduce communication failures as needed.

Processes `ABORT`, `SUCCESS`, `ERROR`, `NO_FUNDS`, `NO_TRANSACTION`, `END` and `FAIL` are mapped to the CSP `STOP` process. We use them to improve readability of the code.

[2]What if a corrupt merchant sends a bogus K? In the Netbill protocol, in step 4, the merchant sends a signed version of K to the bank. Previously, in step 2, the merchant has sent the goods (encrypted with K) to the consumer, and the hash of those has been included in the EPO and signed by both the merchant and the consumer. Hence, after the fact, the fraud and responsible party can be detected and proven to other parties. This is treated at length in [5, 22]. Our analysis in this paper does not consider the impact of bogus keys.

Money and Goods Atomicity for NetBill

Recall the money conservation property given by the sum in Formula 1 at the start of this section. Since we do not have electronic coins, and we have only have one run of the protocol, this property is satisfied exactly when a debit is matched by a credit, or vice-versa. In CSP, this can be specified as:

```
SPEC1 = STOP ||
((debitC -> creditM -> STOP) []
 (creditM -> debitC -> STOP))
```

Note that the third component of the specification, `creditM -> debitC -> STOP`, could be omitted, since in our specification of NetBill a `debitC` always happens before `creditM`, if they ever happen. To check this specification using FDR, we first combine the consumer, merchant and the bank processes with an appropriate communication process, and then hide all of the irrelevant events (the Appendix gives the details of the communication process, process combination and event hiding); call the resulting system `SYSTEM1`. Finally, we check that `SYSTEM1` is a refinement of `SPEC1` with the FDR command line:

```
CONSUMER = ABORT || coutm !goodsReq  -> GOODS_REQ_SENT

GOODS_REQ_SENT = ABORT || cinm ?x ->
                           (if x==encryptedGoods then ENCRYPTED_GOODS_REC
                            else ERROR)

ENCRYPTED_GOODS_REC = ABORT || coutm !epo -> EPO_SENT

EPO_SENT = (cinm ?x -> (if (x==paymentSlip) then SUCCESS
                           else if (x==noPayment) then NO_FUNDS
                                else ERROR)) []
           (timeoutEvent -> coutb !transactionEnquiry -> BANK_QUERIED)

BANK_QUERIED = cinb ?x -> (if (x==paymentSlip) then SUCCESS
                           else if (x==noPayment) then NO_FUNDS
                                else (if (x==noRecord) then NO_TRANSACTION
                                      else ERROR))
```

Figure 2: The consumer process.

```
MERCHANT = ABORT || (minc ?x -> (if x==goodsReq then GOODS_REQ_REC
                                 else ERROR))

GOODS_REQ_REC = ABORT || (moutc !encryptedGoods -> ENCRYPTED_GOODS_SENT)

ENCRYPTED_GOODS_SENT = ABORT || minc ?x -> (if x==epo then EPO_REC
                                            else ERROR)

EPO_REC = ABORT || (moutb !endorsedEpo -> ENDORSED_EPO_SENT)

ENDORSED_EPO_SENT = FAIL ||
          (minb ?x ->
             (if (x==paymentSlip) or (x==noPayment) then
                (FAIL ||
                 moutc !x -> END)
              else ERROR))
```

Figure 3: The merchant process.

```
BANK = WAIT_ENDORSED_EPO

WAIT_ENDORSED_EPO = binm ?x ->
                        (if x==endorsedEpo then
                           (OK_TRANSACTION ||
                            NOK_TRANSACTION)
                         else WAIT_ENDORSED_EPO)

OK_TRANSACTION = debitC -> creditM ->
                 boutm !paymentSlip -> FINAL_BANK(paymentSlip)

NOK_TRANSACTION = boutm !noPayment -> FINAL_BANK(noPayment)

FINAL_BANK(x) = binc ?y ->
                  (if y==transactionEnquiry then
                      boutc !x -> FINAL_BANK(x)
                   else FINAL_BANK(x))
```

Figure 4: The bank process.

```
Check1 "SPEC1" "SYSTEM1"
```

`Check1` is a two argument command that determines whether its second argument is a failure/divergence refinement of its first.

Goods atomicity for NetBill is more complex and involves reasoning about the messages' send and receive synchronization events: `cinm.encryptedGoods` (this indicates C's receipt of the message with the `encyptedGoods`), `cinm.paymentSlip` (this indicates C's receipt of the `paymentSlip` message), and `cinb.paymentSlip` (this indicates M's receipt of the `paymentSlip` message). Now, in order for C to "receive" the goods, C must receive both the `encryptedGoods` message and the encryption key (included in the `paymentSlip` message). We can now specify goods atomicity as follows:

```
SPEC2 =
   STOP ||
   (cinm.encryptedGoods ->
      (STOP ||
       (debitC ->
          creditM ->
            ((cinb.paymentSlip -> STOP) ||
             (cinm.paymentSlip -> STOP)))))
```

For simplicity, this specification is somewhat less general than our previous definition of goods atomicity (in particular, the above specification says that goods must be paid for before they are received, whereas the previous definition stated that goods are received if and only if they are paid for and that the order of receipt and payment does not matter).

Our NetBill model satisfies both `SPEC1` and `SPEC2`; the full FDR code and excerpts from the model checking session appear in the Appendix.

3.2 A Simplified Digital Cash Protocol

The second example we investigate is a simplified digital cash protocol based on the offline Digicash protocols [3, 4]. (We have abstracted away a number of crucial components from the Digicash protocol.) In this protocol, electronic coins are withdrawn from and deposited into bank accounts and used for payments. Moreover, these payments are anonymous for the consumer, because coins are "blinded" during withdrawal. In contrast to NetBill, money atomicity for this protocol is non-trivial, because money is not centralized at the bank. We will focus our analysis on money atomicity. We do not provide a formal analysis of goods atomicity since the protocol actually violates this property (which we explain below).

The Protocol

Figure 5 contains our simplified digital cash protocol. The protocol consists of three parts: withdrawal

1. $C \rightarrow B$: *withdrawal request*
2. $B \rightarrow C$: *coin*
3. $C \rightarrow M$: *goods request with blinded coin*
4. $M \rightarrow C$: *challenge for the blinded coin*
5. $C \rightarrow M$: *response for the challenge*
6. $M \rightarrow C$: *goods*
7. $M \rightarrow B$: *blinded coin + response for the challenge*
8. $B \rightarrow M$: *deposit slip*

Figure 5: Simplified digital cash protocol.

of the coin (steps 1 – 2), spending of the coin (steps 3 – 6), and coin deposit (steps 7 – 8). We now describe each step in turn. The consumer C starts the protocol by requesting a withdrawal from the bank. The bank B responds with an electronic coin of the requested value. Before spending it, C "blinds" the coin to prevent the bank from tracing her payments. To spend the coin, C sends the coin to merchant M, and then responds to a challenge randomly selected by M (importantly, C maintains certain secret information about the coin so that only C can correctly respond to a random challenge). M locally verifies the consistency of the challenge/response pair, and then sends the goods. Finally, M deposits the coin by sending the coin and challenge/response pair to B, who responds with a deposit slip, assuming the coin is valid.

Observe that the essential part of spending the coin is not sending the coin to M, but responding to M's challenge. A consumer must take care not to respond to two different challenges for the same coin, because this will be considered evidence of fraudulent double spending: two challenge/response pairs for one coin are (with very high probability) sufficient for the bank to recover the identity of the consumer.

The protocol is clearly not goods atomic because M can omit step 6 but still deposit the coin. Also, note that the withdrawal part of the protocol (the first two messages) actually consists of a cut-and-choose protocol that involves a large number of message exchanges. These details are irrelevant for our analysis and are omitted.

The Simplified Digital Cash Protocol in FDR

Figures 6, 7, and 8 present the consumer, merchant, and bank processes in FDR. To provide a more realistic modeling of the operation of the protocol, we have expanded the protocol behavior outlined in Figure 5 to include:

- coin returns: the consumer may choose to return coins to the bank for refund,
- fraud: the consumer and merchant can attempt double spending and multiple presentation of the same coin to the bank, and
- coin retention: the consumer may choose not to spend a coin and instead keep it for future use.

Unfortunately, in the context of this slightly more realistic system, a serious ambiguity arises. Consider the following scenerio. A consumer withdraws a coin from the bank and attempts to use the coin to pay a merchant. However, as the consumer's response to the merchant's challenge was in transit, the communication network fails. The consumer is left in an uncertain situation. Has the coin been spent? If the merchant actually receives the response, then the consumer should consider the coin spent, but if not, then the coin is unspent. This is a critical issue for money atomicity, because if the consumer makes the wrong guess, then either money will be lost or she could be accused of double spending. To establish money atomicity, we allow the consumer to go to the bank in this situation and see if the coin has been spent; if it has not, she is eligible for a refund on the coin. Of course this leads to a problem: the consumer can spend the coin and then immediately go to the bank and claim the coin may have been lost in transit and obtain a refund, and then moments later the merchant appears coin in hand. In practice this issue could be addressed by timeout/coin-lifetime management. In our model, we abstract the details of how this is solved and enter an "arbitration" state.

There are, however, two well-defined kinds of fraud that are detected and resolved in our model. The first is when a consumer attempts to double spend a coin, and the second is when a merchant attempts to deposit a coin twice. Both cases are detected by the bank and respective events **cFraud** and **mFraud** are triggered by the bank process. This is important, because it allows us to talk about money atomicity properties: in short, money atomicity holds when there are no **cFraud** and **mFraud** events. We elaborate further when we discuss money atomicity.

Our model includes only two challenge/response pairs, whereas there are really billions of possible such pairs. However, the specific identities of challenge/response pairs are immaterial: the critical property is the number of different challenges to which the customer responds (in fact there are only three important cases corresponding to zero, one, or more than one consumer response). Hence we consider just two "symbolic" challenge response pairs. We also abstract the statistical arguments, and simply state that if both of the symbolic challenge/response pairs are sent to the bank, then the bank has proof of consumer double spending.

We remark that the bank process is somewhat complicated because the bank must record information as it proceeds. This is somewhat cumbersome in FDR, and involves using process parameters. The main process involved here is **WAIT**, which has three parameters, the first indicating whether the coin has been deposited or returned, and the second and third indicating which challenge/response pairs have been seen.

Money Atomicity for Simplified Digital Cash Protocol

Recall the money conservancy property given by the sum in Formula 1. The following CSP specification expresses this property in the context of the simplified digital cash protocol:

```
SPEC3 = STOP ||
        (debitC -> ((depositC -> STOP) ||
                    (cKeepsToken -> STOP) ||
                    (depositM -> STOP)))
```

This specification holds in the presence of non-bank communication failures and limited non-bank agent failures. Surprisingly, it even holds in the presence of consumer and merchant fraud.

Next consider the cash property component of money atomicity. This states that possession of a coin gives the possessor the right to spend and/or deposit the coin. For C, this can be stated as:

```
SPECcashc =
   STOP ||
   (cinb.token -> ((tokenSpent -> STOP) ||
                   (cKeepsToken -> STOP) ||
                   (depositC -> STOP))).
```

and for M we have:

```
SPECcashm =
   STOP ||
   (mGetsToken ->
          ((depositM-> STOP) ||
           (mGetsRefundSlip -> STOP))).
```

C's cash property does in fact hold in the presence of fraud (that is, fraud by M cannot affect C's cash property; M can fail to deliver the goods, but that is not a violation of the cash property, but of goods atomicity). However, M's cash property does not hold: it can be violated by C's fraud. When FDR is applied to this specification, it generates the following counterexample:

```
coutb.tokenReq, binc.tokenReq, debitC,
boutc.token, cinb.token, coutm.goodsReq,
minc.goodsReq, moutc.challengeA,
cinm.challengeA, coutm.responseA,
minc.responseA, mGetsToken, moutc.goods,
moutb.responseA, binm.responseA, depositM,
boutm.depositSlip, minb.depositSlip,
cinm.goods, coutm.goodsReq, minc.goodsReq,
moutc.challengeB, cinm.challengeB,
coutm.responseB, minc.responseB, mGetsToken,
moutc.goods, moutb.responseB,
binm.responseB, boutm.alreadyDeposited,
cFraud, minb.alreadyDeposited, cinm.goods,
tokenSpent
```

This sequence of events corresponds to the scenario where a consumer double spends a coin: after finishing a successful transaction with the merchant (shown by events **mGetsToken**, **depositM**, and **cinm.goods**), the consumer uses the coin again (**mGetsToken**), gets the goods (**cinm.goods**), but instead of successfully

```
CONSUMER = ABORT || (coutb !tokenReq -> TOKEN_REQ_SENT)

TOKEN_REQ_SENT = cinb ?x ->
                   if x==token then (USE_TOKEN [] RETURN_TOKEN [] KEEP_TOKEN)
                   else ERROR

USE_TOKEN = coutm !goodsReq -> GOODS_REQ_SENT

KEEP_TOKEN = cKeepsToken -> END

GOODS_REQ_SENT = cinm ?x -> (if (x==challengeA) then
                                (coutm !responseA -> TOKEN_USED)
                              else (if (x==challengeB) then
                                      (coutm !responseB -> TOKEN_USED)
                                    else RETURN_TOKEN)) []
                  timeoutEvent -> RETURN_TOKEN

TOKEN_USED = (cinm ?x ->
                 (if x==goods then (tokenSpent -> C_MAY_BE_FRAUD)
                  else RETURN_TOKEN)) []
              (timeoutEvent -> RETURN_TOKEN)

RETURN_TOKEN = coutb !token -> cinb ?x ->
                  (if x==refundSlip then REFUND_RECEIVED
                   else if x==depositSlip then (tokenSpent -> ARBITRATION)
                        else ERROR)

C_MAY_BE_FRAUD = END || USE_TOKEN
```

Figure 6: The consumer process for the simplified digital cash protocol.

```
MERCHANT = ABORT || WAITING_GOODS_REQ(none)

WAITING_GOODS_REQ(previousResponse) = minc ?x ->
  (if (x==goodsReq) then
     (if previousResponse==none then (CHALLENGE_A [] CHALLENGE_B)
      else if previousResponse==responseA then CHALLENGE_B
           else CHALLENGE_A)
   else ERROR)

CHALLENGE_A = moutc !challengeA-> WAIT_FOR_RESPONSE(responseA)

CHALLENGE_B = moutc !challengeB-> WAIT_FOR_RESPONSE(responseB)

WAIT_FOR_RESPONSE(response) =
  minc ?x -> (if x==response then (mGetsToken -> SEND_GOODS(x))
              else moutc !badResponse -> NO_TRANSACTION)

SEND_GOODS(response) = (moutc !goods -> DEPOSIT_TOKEN(response)) []
                        DEPOSIT_TOKEN(response)

DEPOSIT_TOKEN(response) = moutb !response -> WAIT_FOR_BANK(response)

WAIT_FOR_BANK(response) = minb ?x ->
        if x==depositSlip then M_MAY_BE_FRAUD(response)
        else if x==refundSlip then (mGetsRefundSlip -> STOP)
             else if x==alreadyDeposited then FRAUD_DISCOVERED
                  else ERROR

M_MAY_BE_FRAUD(response) = END ||
                           DEPOSIT_TOKEN(response) ||
                           WAITING_GOODS_REQ(response)
```

Figure 7: The merchant process for the simplified digital cash protocol.

```
BANK = binc ?x ->
         (if x==tokenReq then (boutc !badBalance -> STOP []
                              debitC -> boutc !token -> WAIT(0, 0, 0))
          else ERROR)

WAIT(cashedFlag, responseA, responseB) =
   binc ?x ->
     (if (x==token) then
         (if (cashedFlag==0) then
            (depositC -> boutc !refundSlip -> WAIT(1, 0, 0))
          else if (responseA==1 or responseB==1) then
                 (arbitration -> boutc !depositSlip ->
                  WAIT(cashedFlag, responseA, responseB))
               else WAIT(cashedFlag, responseA, responseB))
       else WAIT(cashedFlag, responseA, responseB)) []
    binm ?x ->
      (if (x==responseA) then
         (if (cashedFlag==0) then
             (depositM -> boutm !depositSlip ->
              WAIT(1, 1, responseB))
          else if (responseA==1) then
                 (boutm !alreadyDeposited -> mFraud ->
                  WAIT(cashedFlag, responseA, responseB))
               else if (responseB==1) then
                       boutm !alreadyDeposited-> cFraud ->
                       WAIT(cashedFlag,responseA,responseB)
                     else (arbitration -> boutm !refundSlip ->
                           WAIT(cashedFlag, responseA,responseB)))
       else if (x==responseB) then
              (if (cashedFlag==0) then
                 (depositM -> boutm !depositSlip ->
                  WAIT(1, responseA, 1))
               else if (responseB==1) then
                      (boutm !alreadyDeposited -> mFraud ->
                       WAIT(cashedFlag, responseA, responseB))
                    else if (responseA==1) then
                           (boutm !alreadyDeposited -> cFraud ->
                            WAIT(cashedFlag,responseA, responseB))
                         else (arbitration -> boutm !refundSlip ->
                               WAIT(cashedFlag,responseA,responseB)))
            else WAIT(cashedFlag, responseA, responseB))
```

Figure 8: The bank process for the simplified digital cash protocol.

depositing the coin, the merchant receives a message indicating that the coin in question had been spent before (**boutm.alreadyDeposited**), and consumer fraud (**cFraud**) is revealed. (For further details, see the Appendix.)

However, in the absence of revealed fraud (i.e. in the case where there are no (**cFraud**) or (**mFraud**) events), M's cash property is satisfied. We express this as follows:

```
SPECcashm' =
   STOP ||
   ((mGetsToken ->
           ((depositM -> STOP) ||
             (mGetsRefundSlip -> STOP))) ||
    FRAUD)
```

where **FRAUD** denotes processes that contain at least one fraud event, **cFraud** or **mFraud**, and are otherwise arbitrary. Using FDR we checked that indeed the unmodified protocol satisfies this modified property.

4 Summary and Discussion of Our Contributions

We have presented a model checking approach for verifying atomicity properties of electronic commerce protocols. Until now, such properties have been reasoned about using informal and *ad hoc* methods. However, these methods have not been adequate and numerous significant errors have been made in the design of electronic commerce protocols. Not only are the protocols themselves moderately complex and subtle, but the properties expected and/or desired are often only partially specified and not fully understood. Model checkers can address both aspects of this problem: we can write precise definitions of the behavior of a protocol (at any desired level of abstraction) and then formulate protocol properties and test that they are satisfied. If a property is not satisfied, a model checker will give a counterexample, which we can use to step through the execution of the protocol to better understand its behavior. This kind of interactive experimentation is a very powerful tool for debugging and modifying both protocols and the properties we expect to hold. In our experience, the most obvious specification of a property is often incorrect or inadequately expresses the property, and that by experimenting, we frequently obtain more precise and stronger properties.

We have discussed here two properties: money atomicity and goods atomicity. We believe that these techniques will extend to other properties such as anonymity, transactional properties (consistency, isolation, durability), nonrepudiation, certified delivery, etc. Similarly, though we demonstrated our approach on only two protocols, they are radically different from each other; model checking atomicity and other properties should easily be applicable to other electronic commerce protocols.

In our modeling of NetBill and the simple digital cash protocol, we have employed a number of abstractions. For example, we have ignored the low-level details of the the underlying cryptographic mechanisms and just treated them as a blackbox (this is the standard "perfect encryption" assumption). In fact our model goes one step further: we have chosen not to even mention encryption/decryption/signature operations so that we could develop as simple a model as possible and focus on atomicity properties.

A second example of a reasonable abstraction we applied is in modeling the challenge/response pairs in the electronic cash protocol: many billions of different pairs were represented as just two pairs. As a third example, recall that we modeled NetBill and the electronic cash protocol assuming just a small number of players: there was only one bank, one consumer, and one merchant; moreover, we consider only one run of the protocol. In practice, we would expect these protocols to be used in huge networks with large numbers of consumers, merchants and banks, with multiple interleaved runs of the protocols. Roscoe and MacCarthy justify similar simplifications in their work using FDR to model check *data-independent* properties of concurrent processes [20].

In summary, finding the right abstractions is essential to finding an effective representation of a protocol for model checking. The goal is to map an intractable problem into a tractable abstracted problem in such a way that proving something about the abstracted problem says something meaningful about the real problem at hand.

5 Future Work

We plan to investigate some of the assumptions made and abstractions used in our modeling of NetBill and the electronic cash protocol. For example, suppose we consider multiple merchants, consumers, and banks? Multiple runs of a protocol? Multiple transaction values? What number (of players, runs, values) is too large for current model checking technology to handle?

Could we provide a formal justification for some of the abstractions we used? For example, can we prove that if goods atomicity holds for one merchant and one consumer, then it holds for multiple merchants and consumers? We treated the cryptographic component of the protocol as orthogonal to our analysis for atomicity. We doubt that analyzing an enriched FDR model to include the cryptographic component would be tractable; however, we believe that it may be possible to use other model checking methods to address some cryptographic aspects. Then, we may be able to factor the problem of whether, say, NetBill is goods atomic, into two problems: (a) determining whether the cryptographic aspects of NetBill are secure, and (b) determining whether Netbill is goods atomic, assuming its cryptographic aspects are secure (which is essentially what we have proved in this paper).

Finally, we plan to provide more comprehensive failure modeling. In this paper we have used the following informal principle: failures by one agent can interfere with that agent's atomicity properties, but they must not interfere with another agent's properties. We can formulate this is a more precise manner as follows. First, we analyze the atomicity properties that we wish to establish, and associate components of these properties with individual agents. For example, goods atomicity can be stated as: "if the consumer pays then the consumer gets the goods" and "if the consumer gets the goods then the merchant gets paid". The first part of this statement is the consumer's property, and the second is the merchant's. Then, for each agent, we consider a model in which the agent does not fail but other agents fail arbitrarily, and we seek to establish those components of the atomicity properties associated with the non-failed agent. Even more ambitiously, limited bank failure is another important—and realistic—aspect to model for future work.

References

[1] Ross Anderson. UEPS - a second generation electronic wallet. In *The Second European Symposium on Research in Computer Security*, pages 411–418, 1992.

[2] L. Jean Camp, Marvin Sirbu, and J. D. Tygar. Token and notational money in electronic commerce. In *Proceedings of the First USENIX Workshop in Electronic Commerce*, pages 1–12, July 1995.

[3] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, October 1985.

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

[5] Benjamin Cox, J. D. Tygar, and Marvin Sirbu. NetBill security and transaction protocol. In *Proceedings of the First USENIX Workshop in Electronic Commerce*, pages 77–88, July 1995.

[6] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[7] J. Gray and A. Reuter. *Transaction Processing.* Morgan-Kauffman, 1993.

[8] James N. Gray. A transaction model. Technical Report RJ2895, IBM Research Laboratory, San Jose, California, August 1980.

[9] Zvi Har'El and Robert P. Kurshan. Software for analytical development of communications protocols. In *AT&T Technical Journal*, pages 45–60, January/February 1990.

[10] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[11] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[12] Daniel Jackson. Nitpick: A checkable specification language. In *Proc. Workshop on Formal Methods in Software Practice*, San Diego, CA, January 1996.

[13] Rajashekar Kailar. Reasoning about accountability in protocols for electronic commerce. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 236–250, May 1995.

[14] Gavin 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, March 1996.

[15] Formal Systems (Europe) Ltd. *Failures Divergence Refinement—User Manual and Tutorial*, 1993. Version 1.3.

[16] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions.* Morgan Kaufmann, San Mateo, CA, 1994.

[17] K. L. McMillan. Symbolic model checking: An approach to the state explosion problem. Technical Report CMU-CS-92-131, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1992. Ph.D. thesis.

[18] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978. Also Xerox Research Report, CSL-78-4, Xerox Research Center, Palo Alto, CA.

[19] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 114–127, May 1995.

[20] A.W. Roscoe and H. MacCarthy. A case study in model-checking CSP. submitted for publication, October 1994.

[21] Marvin Sirbu and J. D. Tygar. NetBill: an internet commerce system optimized for network delivered services. *IEEE Personal Communications*, pages 34–39, August 1995.

[22] J. D. Tygar. Atomicity in electronic commerce. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 8–26, May 1996.

[23] Jeannette Wing and Mandana Vaziri-Farhani. A case study in model checking software systems. *Science of Computer Programming*, January 1996. To appear. Preliminary version in *SIGSOFT Proc. of Foundations of Software Engineering*, October 1995.

A Appendix

We present the full code for our model of NetBill and the digital cash protocol (lines beginning with "-" are comments). We also give some excerpts of the FDR verifications.

A.1 NetBill

```
-- The data types DATAxy are the set of data
-- that are transmitted from the principal x
-- to the principal y;
DATAmc = {noPayment, paymentSlip,
          encryptedGoods}
DATAbc = {noPayment, paymentSlip, noRecord}
DATAcm = {goodsReq, epo}
DATAcb = {transactionEnquiry}
DATAbm = {paymentSlip, noPayment}
DATAmb = {endorsedEpo}

-- The names of the channels are of the form
-- x(in/out)y, where in/out refers to the
-- direction (relative to x), and y is the
-- other party of the communication;
pragma channel coutm : DATAcm
pragma channel coutb : DATAcb
pragma channel moutc : DATAmc
pragma channel moutb : DATAmb
pragma channel boutc : DATAbc
pragma channel boutm : DATAbm
pragma channel minc : DATAcm
pragma channel binc : DATAcb
pragma channel cinm : DATAmc
pragma channel binm : DATAmb
pragma channel cinb : DATAbc
pragma channel minb : DATAbm

pragma channel creditM, debitC, timeoutEvent

[The consumer process CONSUMER:
 as given in the paper.]

[The merchant process MERCHANT:
 as given in the paper.]

[The bank process BANK:
 as given in the paper.]

SUCCESS = STOP
ERROR = STOP
```

```
NO_FUNDS = STOP
NO_TRANSACTION = STOP
END = STOP
ABORT = STOP
FAIL = STOP

-- The communication channels: only those
-- involving the bank are reliable.

COMMcm = []x: DATAcm @
            (coutm ?x ->
               (COMMcm []
                (minc !x -> COMMcm)))
COMMcb = []x: DATAcb @
            (coutb ?x ->
               ((binc !x -> COMMcb)))
COMMmc = []x: DATAmc @
            (moutc ?x ->
               (COMMmc []
                (cinm !x -> COMMmc)))
COMMmb = []x: DATAmb @
            (moutb ?x ->
               ((binm !x -> COMMmb)))
COMMbc = []x: DATAbc @
            (boutc ?x ->
               ((cinb !x -> COMMbc)))
COMMbm = []x: DATAbm @
            (boutm ?x ->
               ((minb !x -> COMMbm)))

COMM = COMMcm [|{}|] COMMcb [|{}|] COMMmc
       [|{}|] COMMmb [|{}|] COMMbc
       [|{}|] COMMbm

-- The Whole Netbill System
CIO = {| coutm, coutb, cinm, cinb |}
MIO = {| moutc, moutb, minc, minb |}
BIO = {| boutc, boutm, binc, binm |}
BINT = {debitC, creditM}
BTOT = union(BIO, BINT)
COMMIO = union(CIO, union(MIO, BIO))
COMMIO' = diff(COMMIO,
              {cinm.encryptedGoods,
               cinm.paymentSlip,
               cinb.paymentSlip})

SYSTEM1 =
   ((CONSUMER [|{}|] MERCHANT [|{}|] BANK)
    [| COMMIO |] COMM)
   \ union(COMMIO, {timeoutEvent})
```

```
SYSTEM2 =
   ((CONSUMER [|{}|] MERCHANT [|{}|] BANK)
    [| COMMIO |] COMM)
   \ union(MIO,
       union(BIO,
         union({|coutm, coutb |},
               {cinm.noPayment,
                cinb.noPayment,
                cinb.noRecord,
                timeoutEvent})))

SPEC1 = STOP |~|
       ((debitC -> creditM -> STOP) []
        (creditM -> debitC -> STOP))

SPEC2 = (STOP |~|
        (cinm.encryptedGoods -> (STOP |~|
           (debitC -> creditM ->
           ((cinb.paymentSlip -> STOP) |~|
            (cinm.paymentSlip -> STOP))))))
```

The check of SPEC1 generated the following FDR output:

```
fdr> Check1 "SPEC1" "SYSTEM1";
SPEC1 ....(5 states)
CONSUMER ..........(10 states)
MERCHANT ...............(15 states)
BANK .........(10 states)
COMMcm .(3 states)
COMMcb .(2 states)
COMMmc .(4 states)
COMMmb .(2 states)
COMMbc .(4 states)
COMMbm .(3 states)
readalphabet:
  Alphabet contains 3 events [up to 998]
6 configuration masks in 6 transitions
3
5 reachable configurations
0
nfcompact :
  compacting 4 state normal form:
    now 4 states
87 configuration masks in 67 transitions
139
199 reachable configurations
1 The implementation does indeed
  refine the normal form.
Checked 199 pairs.
Refinement check succeeded
```

```
No failure in this context!
val it = - : (label,selector,cause) Context
fdr>
```

A.2 A Simplified Digital Cash Protocol

```
-- The data types DATAxy are the set of data
-- that are transmitted from the principal x
-- to the principal y;
DATAcb = {tokenReq, token}
DATAcm = {responseA, responseB, goodsReq}
DATAbc = {token, badBalance, badToken,
          depositSlip, refundSlip}
DATAbm = {refundSlip, depositSlip,
          alreadyDeposited}
DATAmc = {goods, badResponse, challengeA,
          challengeB}
DATAmb = {responseA, responseB}

[Communication channels:
 as given for NetBill.]

pragma channel goodsReceived, debitC,
       depositC, depositM, mFraud, cFraud,
       mGetsRefundSlip, tokenSpent,
       mGetsToken, cKeepsToken,
       timeoutEvent, arbitration

[The consumer process CONSUMER:
 as given in the paper.]

[The merchant process MERCHANT:
 as given in the paper.]

[The bank process BANK:
 as given in the paper.]

ABORT = STOP
END = STOP
REFUND_RECEIVED = STOP
ERROR = STOP
ARBITRATION = arbitration -> STOP
NO_TRANSACTION = STOP
FRAUD_DISCOVERED = STOP

[COMM: as given for NetBill.]

-- The communication events
CIO = {| coutm, cinm, coutb, cinb |}
MIO = {| moutc, minc, moutb, minb |}
BIO = {| boutc, boutm, binc, binm |}
COMMIO = union(CIO, union(MIO, BIO))
COMMIO' = diff(COMMIO, {cinb.token})

-- The model for Money Atomicity

SYSTEM3 =
   ((CONSUMER [|{}|] MERCHANT [|{}|] BANK)
    [| COMMIO |] COMM)
   \ union(COMMIO,
          {goodsReceived, mGetsRefundSlip,
           mGetsToken, tokenSpent, mFraud,
           cFraud, timeoutEvent,
           arbitration})

SYSTEMc =
   ((CONSUMER [|{}|] MERCHANT [|{}|] BANK)
    [| COMMIO |] COMM)
    \ union(COMMIO',
           {goodsReceived, debitC,
            depositM, mGetsToken,
            mGetsRefundSlip, timeoutEvent,
            arbitration, mFraud, cFraud})
SYSTEMm =
   ((CONSUMER [|{}|] MERCHANT [|{}|] BANK)
    [| COMMIO |] COMM)
   \ union(COMMIO,
          {goodsReceived, debitC, depositC,
           tokenSpent, cKeepsToken,
           timeoutEvent, arbitration,
           mFraud, cFraud})

SYSTEMm' =
   ((CONSUMER [|{}|] MERCHANT [|{}|] BANK)
    [| COMMIO |] COMM)
   \ union(COMMIO,
          {goodsReceived, debitC, depositC,
           tokenSpent, cKeepsToken,
           timeoutEvent, arbitration})

SPEC3 =
   STOP |~|
   (debitC -> ((depositC -> STOP) |~|
               (cKeepsToken -> STOP) |~|
               (depositM -> STOP)))

SPECcashc =
   STOP |~|
   (cinb.token -> ((tokenSpent -> STOP) |~|
                     (cKeepsToken -> STOP) |~|
                     (depositC -> STOP)))
SPECcashm =
   STOP |~|
   (mGetsToken ->
      ((depositM-> SPECcashm) |~|
       (mGetsRefundSlip -> SPECcashm)))

SPECcashm' =
   STOP |~|
   ((mGetsToken ->
       ((depositM-> STOP) |~|
        (mGetsRefundSlip -> STOP)))
    |~| FRAUDM)

FRAUDM = (mGetsToken -> FRAUDM) |~|
         (mGetsRefundSlip -> FRAUDM) |~|
         (depositM -> FRAUDM) |~|
         (mFraud -> ANYM) |~|
         (cFraud -> ANYM)

ANYM = (mGetsToken -> ANYM) |~|
       (mGetsRefundSlip -> ANYM) |~|
       (depositM -> ANYM) |~|
       (mFraud -> ANYM) |~|
       (cFraud -> ANYM) |~|
       STOP
```

The check of SPEC1 generated the following FDR output (and counter-example):

```
fdr> Check1 "SPECcashm" "SYSTEMm";
SPECcashm ....(6 states)
CONSUMER ..................(16 states)
MERCHANT ......................(24 states)
BANK .................(19 states)
COMMcm .(4 states)
COMMcb .(3 states)
COMMmc .(5 states)
COMMmb .(3 states)
COMMbc .(6 states)
COMMbm .(4 states)
readalphabet:
  Alphabet contains 4 events [up to 998]
7 configuration masks in 7 transitions
3
6 reachable configurations
0
nfcompact :
  compacting 2 state normal form:
    now 2 states
174 configuration masks in 149 transitions
120
1002 reachable configurations
1

SYSTEMm
Interface={|{|depositM,mGetsRefundSlip,
             mGetsToken,tick|}|}
has behaviour
After <tau,tau,tau,tau,tau,tau,tau,tau,tau,
       tau,tau,tau,tau,mGetsToken,tau,tau,
       tau,depositM,tau,tau,tau,tau,tau,
       tau,tau,tau,tau,tau,tau,mGetsToken,
       tau,tau,tau,tau,tau,tau,tau,tau,tau>
refuses {|{|depositM,mGetsRefundSlip,
          mGetsToken,tick|}|}
Accepts only {{||}|}
Contributions:
Component 1
((CONSUMER[|{||}|]MERCHANT)
 [|{||}|]BANK)
[|{|coutb,coutm,moutc,moutb,boutc, boutm,
    cinb,cinm,minc,minb,binc,binm|}|]COMM
Interface={|{|coutb,coutm,moutc,moutb,boutc,
             boutm,cinb,cinm,minc,minb,
             binc,binm,debitC,depositC,
             depositM,mFraud,cFraud,
             mGetsRefundSlip,tokenSpent,
             mGetsToken,cKeepsToken,
             timeoutEvent,arbitration,
             tick|}|}
has behaviour
After <tau,tau,coutb.tokenReq,binc.tokenReq,
       debitC,boutc.token,cinb.token,
       coutm.goodsReq,minc.goodsReq,
       moutc.challengeA,cinm.challengeA,
       coutm.responseA,minc.responseA,
       mGetsToken,moutc.goods,
       moutb.responseA,binm.responseA,
       depositM,boutm.depositSlip,
       minb.depositSlip,tau,cinm.goods,tau,
       coutm.goodsReq,minc.goodsReq,
       moutc.challengeB,cinm.challengeB,
       coutm.responseB,minc.responseB,
       mGetsToken,moutc.goods,
       moutb.responseB,binm.responseB,
       boutm.alreadyDeposited,cFraud,
       minb.alreadyDeposited,cinm.goods,tau,
       tokenSpent>
refuses {|{|coutb,coutm,moutc,moutb,boutc,
                      boutm,cinb,cinm,minc,minb,binc,
                      binm,debitC,depositC,depositM,
                      mFraud,cFraud,mGetsRefundSlip,
                      tokenSpent,mGetsToken,
                      cKeepsToken,timeoutEvent,
                      arbitration,tick|}|}
Accepts only {{||}|}
it = - : (label,selector,cause) Context
fdr>
```