Modelling a Java Ring based implementation of the N-Count payment system

John D Revill, Pieter H. Hartel
{jdr196,phh}@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-99-4
June 1 1999

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom
Modelling a Java Ring based implementation of the N-Count payment system

John D. Revill, Pieter H. Hartel *

June 1, 1999

Abstract

N-Count is a system for offline value transfer. A prototype of an N-Count payment system has been designed, and it has been implemented in Java. We have used the Java Ring with the Java Card API as a secure device. The system has also been modelled using the Spin model checker. The combined prototyping and model checking has made it possible to investigate safety properties of the prototype, both formally and intuitively. Because of this model building activity, problems have been identified and solved before an actual system has been built.

1 Introduction

N-Count [5] is an electronic value transfer system that can be used for fast payments with a secure purse. The payments are secure, off-line and inexpensive. Like in all electronic payment systems [14], there are three entities in an N-Count based payment system: the purse provider, the terminal and the purse. Value is transferred between these entities by the use of N-Counters. The role of the purse provider is to load the purse, and to clear N-Counters received from the terminal. The role of the terminal is to ask the purse to (re-)value N-Counters during payment.

An N-Count purse stores the balance of its user as well as secret keys. The N-Count protocol and an appropriate cryptographic one-way function ensure that the balance is increased (load) or decreased (pay) as appropriate for the transactions made by the user. An implementation that offers overall security therefore imposes three main requirements: (1) the device should be tamper resistant (to protect the balance and the keys) and it should be based on a correct design and implementation of (2) the one-way function and (3) the protocol.

Our ultimate aim is to design and build a system that satisfies all three requirements. As a first step towards achieving this goal we set out to build a prototype aimed at investigating the third aspect: the study of the protocols. The present paper reports on this first step.

To make a realistic prototype we used a Java Ring, with the Java Card API. The ring provides battery powered, active tamper resistance, which we would expect to offer better security than a typical smart card. The Java Card API offers an implementation of SHA-1 [15, Section 18.7] that is appropriate for use as a one way function. We believe it to be a reasonable assumption (at least for now) that the Java Ring and Java Card together satisfy our requirements (1) and (2) above. Therefore, we proceed to investigate requirement (3), the correctness of the protocols in the prototype on this basis.

Model checking [11] is a technique that checks whether a state machine satisfies certain properties. In the present context, the model describes the N-Count protocol, as well as the state of the three parties, upon which the protocols act. A model is then an assignment of values to the components of the state. Model checking verifies that only states arise that are correct in some sense. We use the gold standard model checker, SPIN [12], which uses a CSP like [10] notation for describing protocols, and a C-like description of the state components and operations on the state components.

- Using such a tool allows an appropriate formal model to be animated with relative ease. This helps designers and programmers to understand what the system should do.

- A formal model is unambiguous and it may therefore be used as a contract (at an appropriate level of detail) between designers and programmers. A SPIN model is relatively easy to understand by a good programmer.

*Dept. of Electronics and Computer Science, Univ. of Southampton, UK, Email: jdr96,phh@ecs.soton.ac.uk
Related work

Electronic payment systems have been proposed for various different purposes. The main driver in the design of current systems is the cost of value transfer, which should be considerably lower than the ‘real’ value transferred. At one end of the spectrum micro payment systems, such as BT’s Quickpay [9] avoid cryptographic calculations and support even off-line payments (but less securely so than on-line payments). At the other extreme, systems such as the Inter-sector Electronic Purse (IEP) [6] define complex transactions aimed at mainly large value transfers. It is probably fair to say that most attention has been paid to the security of the cryptography [15, pages 65-68]. Less attention has been paid to creating provably correct implementations of such systems. An exception is the work of Anderson and Bezuidenhoudt, who describe a design and implementation of an electronic pre-payment system for electricity meters [3]. The success of their approach is attributed to the use of a simple, specialised logic [2] (an extension of BAN [4] logic). Currently part of the executive card software for the Mondex electronic purse is in the process of being evaluated at ITSEC level 6 [13], which implies the use of some formal method during the evaluation.

In previous work we have built a formal model of Quickpay [8], which enabled us to prove properties of the protocol. Our model of the IEP [7] was geared towards animating and studying the protocol. Both efforts helped to gain a better understanding of implementation choices for the security of the system. The present paper complements our research portfolio in that it uses model checking, which in a sense combines animation and automatic theorem proving.

2 N-Count protocol

An N-Counter is made up of a sequence of values $X_0, X_1, \ldots, X_n$. Here $X_0$ represents the maximum value possible: $X_0$ is worth $n \times u$ units. The actual value of a unit $u$ can be decided as appropriate, and it may differ between N-Counters. $X_0$ represents the minimum value possible (i.e., 0 units).

The purse contains a secret key $k$, and two functions $f$ and $g$. The most important is the one-way function $f$, which is used to de-value an N-Counter. A block cipher or hash function like SHA-1 [15, Section 18.7] can be used for $f$. The other function $g$, is used to create an initial N-Counter. It is quite possible to implement $g$ and $f$ with same cryptographic algorithm. Table 1 shows the values in an N-Counter, initially created by an application of $g$ at full value, and subsequently de-valued by repeated applications of $f$.

Associated with every N-Counter is a unique random number $r$. To support more than one terminal in the system, each N-Counter is related to a terminal identifier $t$. In practice $t$ and $r$ may be combined as an optimisation, but this does not concern us here.

$X_0$ is generated by $g(u, r, t, n, k)$ as can be seen at the top of Table 1. $f^n(g(u, r, t, n, k), k)$ is evaluated to get $X_n$, where the notation $h^n(x) = h(h^{n-1}(x))$. The essential idea of the N-Count protocol is thus that a counter is de-valued simply by applying the one-way function $f$. To re-value an N-Counter one has to start from the beginning, (using $g$ and the secret key $k$, etc.) and then to de-value as required.

The purse provider knows everything $(f, g, u, r, n, t$ and $k)$ for all N-Counters, terminals and purses. The purses are trusted, but the terminals need not be trusted as they can not steal, reuse or generate any values or N-Counters.

The purpose of the N-Count protocol is to pass N-Counters around and to authenticate them. Figure 1 shows some of the messages as they might be exchanged in the N-Count protocol. The first message,
INITIALISE, prepares the purse for use. The initial stored value and the secret key, k, is transmitted from
the purse provider to the purse and this happens in a secure environment.

The CREATE message is used by a terminal to ask the purse provider for a new N-Counter with which it
can contain values. The unit value u, terminal id t, and the desired capacity n are sent with the message
and the purse provider responds by creating a zero valued N-Counter $X_0$ (a valuegram). This new counter
and its random number, $r_1$, is returned to the terminal by a NEWCOUNTER message. In the prototype, the
capacity of all N-Counters, n, is fixed at 10.

A payment is initiated by a terminal with the GENERATE message. This message communicates the
terminal’s current initially zero valued N-Counter, $X_n$, to the purse (along with the random number $r_1$, the
terminal id t and the unit value u of the N-Counter). The purse recreates $X_0$ using the function $g$ on
the properties of the terminal’s N-Counter. It then steps through the whole range of values from $X_0$ to
$X_n$, using the function $f$, and remembers the value $X_{n-1}$. The purse then compares its version of $X_n$
to the value received from the terminal. If they agree then $X_{n-1}$ is returned (the RETURNGENERATECOUNTER
message) representing a unit payment.

At the end of the day the N-Counter $X_{n-1}$ is sent to the purse provider for clearance, and a new
N-Counter is returned which uses a fresh random number $r_2$. This example involves just one value, but
N-Count is not restricted to just 1 credit per transaction.

<table>
<thead>
<tr>
<th>N-Counter</th>
<th>Worth</th>
<th>Generated by</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_0$</td>
<td>$n \times u$</td>
<td>$g(u, r, t, n, k)$</td>
</tr>
<tr>
<td>$X_1$</td>
<td>$(n-1) \times u$</td>
<td>$f(X_0, k)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$X_{n-2}$</td>
<td>$2 \times u$</td>
<td>$f(X_{n-3}, k)$</td>
</tr>
<tr>
<td>$X_{n-1}$</td>
<td>$u$</td>
<td>$f(X_{n-2}, k)$</td>
</tr>
<tr>
<td>$X_n$</td>
<td>$0$</td>
<td>$f(X_{n-1}, k)$</td>
</tr>
</tbody>
</table>

Table 1: Progression of values in an N-Counter.

4 Prototype

We have designed a prototype payment system based on the N-Count protocol. The system has been
implemented in Java, using a Java Ring with the Java Card 2.0 API as the secure device. The terminal and
purse provider use the OpenCard framework to communicate with the Java Ring. We have implemented
TEA [16] as the one way function, since there is no block cipher algorithm available on the export version
of the Java Ring. TEA is also small and fast. The secret keys transmitted from the purse provider to
the purse are not encrypted, as the initialisation of the purse happens in a secure context. The prototype
offers the following functionality:

1. The code on the ring, or applet, accepts incoming communication packets (APDU packets as specified
   in ISO 7816 [1]) and can execute any of the following instructions:
   - INITIALISE, which expects a starting value and a secret key;
   - CREDIT, which expects a secret key and value;
   - GETBALANCE, which returns the amount that is currently stored in the applet;
   - GENERATECOUNTER, which expects counter information, and returns the next higher value
     within it. The balance is lowered appropriately.

2. A terminal may either ask the purse provider to create a new counter, or it may cash in a full
   counter. The terminal may also pass an N-Counter to the ring and ask for an extra value to be
   created in the counter.

3. The purse provider can credit the ring with more money and initialise it with a secret key. The
   purse provider keeps track of all the counters that each terminal holds and will redeem them upon
   request, crediting their bank account.
Figure 1: The life cycle of an N-Counter.
Figure 2: Interactions between the three entities, showing the messages of the N-Count protocol, as well as those added for the bookkeeping in the prototype.
Figure 2 shows how the protocol has been implemented. The messages described in the previous section have been represented as solid arrows. Some extra messages have been added to perform book keeping tasks (dashed arrows). These include operations to credit the ring, and to retrieve the balance of the ring.

There are some significant simplifying assumptions made in the prototype:

1. There is only one purse provider;
2. The purse provider is active and running before anything else;
3. The applet on the Java ring is installed and the secret key is initialised by the purse provider before any terminals can have access to it;
4. One of either the purse provider or a terminal can communicate with the ring at any one time;
5. There is only one Java Ring and one applet;
6. The capacity of all N-Counters, $n$, is fixed at 10.

We believe these assumptions to be a reasonable starting point. In particular 1, 2 and 3 are common assumption in many systems. Assumption 4 is justified because our system has been implemented on top of an API that offers mutual exclusion. Actually, the complete N-Count protocol suite guarantees atomicity of the payment operation such that simultaneous load and multiple payment operations can take place if supported by the communication interface. Having just one ring is not common, but we are confident that the prototype will scale up. This is a topic for further research.

5 Verification

SPIN [11], is a model verification tool used to prove or disprove certain properties of specifications. SPIN constructs a finite state machine for the specification, which it explores exhaustively during verification. The tool is particularly suited to verifying the logical consistency of concurrent processes, especially communication protocols. The tool allows for random or interactive simulations of the model with a trace of the execution.

The N-Count prototype is modelled in SPIN’s language PROMELA (Process Meta Language). This is a non-deterministic language with support for synchronous or asynchronous communication channels. The PurseProvider, Terminal and RingApplet (the purse) are represented as processes, communicating via synchronous channels.

In the model there are several assumptions made in addition to those made for the prototype:

7. Only one counter is used, and the counter is of one unit value;
8. Every transaction involves just one credit;
9. Counters are only redeemed when full;
10. The actual data are represented as a byte to reduce the size of the state space.
11. Cryptographic operations are modelled by simple arithmetic operations.
12. All the instructions on the ring are atomic.

Assumptions 7–10 serve to reduce the number of states and possible interactions in the model to the essential. No relevant information is gained by increasing the number of counters and/or credits. Assumption 11 is consistent with our aim of studying the protocol, rather than the cryptographic aspects of the protocol. This assumption is justified because we are looking for problems in the protocols regardless of the particular cryptographic algorithms used. Assumption 12 is justified because there is provision in the Java Card framework for atomic transactions. If the ring is removed from the terminal halfway through an instruction then the state of the applet is not changed.

Having discussed our modelling assumptions, will now present the model itself. Given that the prototype implementation comprises 15 classes and roughly 4000 lines of code, the reader will appreciate the succinctness of the model represented in full on the next four pages.

Figure 3 shows the constants of the model. These include the number of terminals, the number of values in a counter, the number of credits to initialise an applet with, the number of credits to give to the
#define NUMOFTERMINALS 2
#define COUNTERSIZE 5
#define STARTCREDITS 10
#define CREDITAMOUNT 5
#define CREDITMAX 20
#define NUMOFRANDOMNUMS 200

Figure 3: Constants of the N-Count model.

mtype = { CREATE, REDEEM, NEWCOUNTER, INITIALISE, GENERATECOUNTER,
         RETURNINGGENERATECOUNTER, CREDIT, GETBALANCE, RETURNBALANCE };  

Figure 4: Message types declaration.

applet every time it is credited, the maximum number of credits an applet can contain, and the number of random numbers available for new counters.

Figure 4 gives the different types of messages in the N-Count model. CREATE and REDEEM are for a message from the terminal to the purse provider asking for counters either to be created or redeemed respectively. NEWCOUNTER is for sending new counters from the purse provider to a terminal. INITIALISE and CREDIT are messages sent from the purse provider to the ring, either for initialising or crediting the applet. GENERATECOUNTER and RETURNINGGENERATECOUNTER are messages sent between a terminal and the ring for generating a counter. GETBALANCE and RETURNBALANCE both are messages for getting the balance from the ring.

The state of the N-Counter is defined in Figure 5. This is essentially just a stack of values implemented by an array, xValue, and indexing variable counterSize. Here randomNumbersUsed keeps track of the number of unique random numbers ever created for counters used by this terminal. In fact, randomNumbersUsed also represents the total number of counters created for the terminal because of the assumption that each terminal holds only one counter.

Figure 6 shows the global channels used by the processes of the model. A channel has a type based on the contents of the data it transmits. For example the channel used by the terminal to obtain a fresh N-Counter from the purse provider (channel toPurseProvider_Create) transmits a pair of data consisting of a message type (mtype) and a return channel (chan). There may be more than one channel connecting two processes, each with their own message type. Multiplexing these parameters onto one channel would be an optimisation. The annotation [0] indicates that the channels are unbuffered, thus giving rise to synchronous communication.

Figure 7 shows the PurseProvider process. This process first sends an INITIALISE message to the ring (the send operator on a channel is !). The process then goes into a loop (bracketed by do and od). In the loop the process is ready to accept either a CREATE or a REDEEM message, or it may of its own accord send a GETBALANCE message. For example a CREATE message would arrive (the receive operation on a channel is ?) with further data specifying which return channel to use. This information is bound to the local process variable toTerminal. Once the N-Counter c has been created, this return channel is used for sending the NEWCOUNTER(c) message. The N-Counter c is created in the model without the use of any one way functions, c.f. assumption 10.

The do-statement loop has been labelled by end to indicate that the beginning of the loop is a valid end state for the process. If the process ends in any other state, a deadlock has occurred.

The assertions throughout the specification serve to uphold the safety properties.

Figure 8 shows the Terminal process, of which there are NUMOFTERMINALS active at the start. Counter c is the N-Counter for the terminal. The terminal first asks the purse provider to create a new counter. It then goes into a loop checking the ring has enough credits (GETBALANCE message) and asking for new values (GENERATECOUNTER message). In a practical implementation the payment request is made without first checking the balance. The purse will return an "insufficient balance error" if appropriate. If the

typedef Counter { byte xValue[COUNTERSIZE];
                 byte counterSize;
                 byte randomNumbersUsed; }  

Figure 5: Counter structure definition.
chan toPurseProvider_Redeem = [0] of { mtype, chan, Counter };
chan toPurseProvider_Create = [0] of { mtype, chan };
chan toPurseProvider_Balance = [0] of { mtype, byte };
chan toRing_Generate = [0] of { mtype, chan, Counter };
chan toRing_Balance = [0] of { mtype, chan };
chan toRing_ProviderCommand = [0] of { mtype };

Figure 6: Global communication channels.

active proctype PurseProvider() {
    byte balance;
    chan toTerminal;
    Counter c;

    toRing_ProviderCommand !INITIALISE;

end:do
:: toPurseProvider_Create?CREATE(toTerminal);
    c.xValue[0] = 0;
    c.counterSize = 1;
    c.randomNumbersUsed = 0;
    toTerminal!NEWCOUNTER(c)

:: toPurseProvider_Redeem?REDEEM(toTerminal, c);
    c.xValue[0] = 0;
    c.counterSize = 1;
    c.randomNumbersUsed++;
    assert(c.randomNumbersUsed <= NUMOFRANDOMUMS);
    toTerminal!NEWCOUNTER(c)

:: toRing_Balance!GETBALANCE(toPurseProvider_Balance);
    toPurseProvider_Balance?RETURNBALANCE(balance);
    if :: ((balance + CREDITAMOUNT) <= CREDITMAX) ->
        toRing_ProviderCommand!CREDIT
    :: ((balance + CREDITAMOUNT) > CREDITMAX) ->
        skip
    fi;
od

Figure 7: PurseProvider process.
counter becomes full, the terminal asks the purse provider to clear the N-Counter (REDEEM message) and to create a new one (CREATE message). The two local channels are used for return communication from the PurseProvider or RingApplet processes. Whenever a terminal sends a message, one of these channels is attached to it.

Figure 9 shows the RingApplet process models the applet on the Java Ring. When a message is received on a channel (an incoming APDU), the processing (executed instruction) is done atomically.

When a value is to be generated for a terminal (a GENERATECOUNTER message), the new value is added to the counters stack, its size is incremented and currentCredits, the stored balance on the applet, is decremented.

If a GETBALANCE message is received, currentCredits is returned on the channel supplied (reply) by the sender.

The CREDIT message is sent by the PurseProvider process. There is no reply but as a side effect the stored balance is increased by the constant CREDITAMOUNT.

6 Verification results

The model was created in parallel with the prototype. This gave two different views on the system. The presentation of the results is therefore split into two: results obtained through the modelling activity and results from the exploration of the state space by model checking. Following are the flaws detected by Spin when checking for violations of safety properties and changes of system invariants:

- It is possible for the variable used to store the number of credits on the applet to overflow. For example, if a byte were used to store the number of credits and the purse provider credited the applet with 256 credits, the purse holder would be disappointed. As well as a minimum number of credits, zero in this case, and there must be a maximum to ensure that overflow does not occur. An assertion failed when verifying an earlier model with CREDITMAX set to 1000. The current implementation now includes a guard in the purse provider. This change is reflected in the PurseProvider process by getting the balance from the ring and checking that a credit would not overflow.
active proctype RingApplet() {
    byte currentCredits;
    Counter c;
    chan reply;

    toRing_ProviderCommand?INITIALISE;
    currentCredits = STARTCREDITS;

    end:do
        :: atomic {
            toRing_Generate?GENERATECOUNTER(reply, c);
            assert (c.counterSize < COUNTERSIZE);
            assert (currentCredits > 0);
            c.xValue[c.counterSize] = c.xValue[c.counterSize - 1] + 1;
            c.counterSize++;
            currentCredits--;
            reply!RETURNGENERATECOUNTER(c)
        }

        :: atomic {
            toRing_Balance?GETBALANCE(reply);
            reply!RETURNBALANCE(currentCredits)
        }

        :: atomic {
            toRing_ProviderCommand?CREDIT;
            assert ((currentCredits + CREDITAMOUNT) <= CREDITMAX);
            currentCredits = currentCredits + CREDITAMOUNT;
        }
    od
}

Figure 9: Ring Applet process.
There may be too few random numbers available for new counters in the prototype. Typically in a practical implementation the random numbers are 8 byte quantities, so there should be an ample supply for the rest of the lifetime of the universe. Actually, the number of random numbers is less than $2^{64}$ as the hash function used, typically only has a subset of the full number range in its image, of which it produces the elements in arbitrary cyclic order. Most hash functions have a number of distinct and potentially relatively small image sets. The random number, $r$, to be used in an N-Counter must be selected to produce under repeated application of the one-way function a cycle of at least the number of valuegrams used in an N-Counter.

The prototype allows the number of credits on the ring to go below zero, or whatever minimum value is used. It was possible for the terminal 1 to get the balance on the applet, terminal 2 to debit the applet such that the balance is 0 and then terminal 1 to debit the applet (or attempt to) so the balance is below 0. This arises because the implementation of the terminal assumes that the ring remains inserted between getting the applet balance and debiting the applet. This may be a trivial bug easily discovered when testing the implementation, however, if communication to each terminal were via a proxy (i.e., a normal Java applet) over the Internet, this could have been a more serious bug. In this situation the terminal may not know when the ring has been inserted or removed. Figure 10 shows this scenario produced by a guided SPIN simulation. Firstly the ring is initialised with a balance of 2. Secondly, the two terminals get new counters from the purse provider. Terminal B asks the ring for a new value after checking there are enough credits. Another transaction session starts and Terminal B gets the balance of the ring (1 credit). The ring is removed and inserted into Terminal A which gets the balance (1) and asks for a new value. The balance of the ring applet is now 0. The ring is removed and re-inserted into Terminal B which now asks for a new value. The assertion $(\text{currentCredits} > 0)$ is violated in the RingApplet and execution can not continue.

Whilst not detected by Spin, the problems identified through the modelling activity are shown below:

- The applet must be installed and initialised at the purse provider before anything else. If the applet were not installed at the purse provider, a rogue applet could be installed elsewhere. This rogue applet would cause some annoyance to terminals by creating bogus values for their counters. The purse provider can trace these bogus values back to the ring, when the terminal comes to redeem the counter (which could be immediate). The purse provider will know that the ring applet has not been installed at the purse provider. If the applet is installed but not initialised, then it is possible for the applet to be initialised elsewhere with a bogus secret key. Again, the purse provider can detect the rogue applet. In both cases, the annoyance could cause loss of business, but the problem would be detected.

- By passing some extra information to the ring applet, namely a number, it becomes possible to ask for several values in a counter to be generated at one time, thus reducing the number of messages involved significantly. Communication to and from a Java Ring applet is at least 1.8 seconds (with the ring used), so at least $1.8 \times (r - 1)$ seconds are saved if $r$ values in a counter are asked for in a transaction. This thus lowers the cost of the transaction.

- A guided simulation detected that starvation is possible amongst terminal processes. One or more terminals could hog the communication channel to the purse provider, making it difficult for some terminal to redeem counters. This indicates the possibility for a denial of service attack.

## 7 Conclusion

A prototype payment system based on the N-Count protocol has been designed, implemented in Java, and it has been modelled using the SPIN model checker. The modelling as well as the automatic verification with the SPIN model checker has revealed a number of problems with the prototype. For example, bogus purses could collude to damage the business of a merchant, and merchants could subject one of their number to a denial of service attack.

Using a model checker requires the designer to separate the main issues from the less important ones. The core of the design can then be subjected to exhaustive state space exploration. This process delivers useful and authoritative information about the system, and it does so more or less automatically. Separating the design issues is hard, but in the end, it is beneficial, as new insights will emerge, enabling better design decisions.
Figure 10: Simulation output from xspin of an interactive session showing how the ring balance is brought below zero (and consequently deadlock).
As an example, we concluded that it would be advantageous for another instruction to be included on the ring that gives some information about itself such as the maximum number of credits that can be held on it. The information that is given should be flexible and extensible so currently unknown information can be given in the future without any change to the protocol.

After ironing out the flaws found by the model checking, more confidence in the implementation of the N-Count protocol has been gained together with detailed knowledge and understanding of its behaviour.

We have shown that design and implementation of a Java Card application benefits from the modelling procedure. This is routinely practised in some other domains, but there are signs that the situation is changing rapidly in the smart card arena.

8 Acknowledgements

We are grateful to QC Technology for providing us with the N-Count demonstration software, and to SUN Microsystems for providing us with a Java Ring development kit. Eduard de Jong gave us helpful comments on a draft version of the paper.

References


