Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Formal Analysis of a Fair Payment Protocol

J.G. Cederquist, M.T. Dashti

REPORT SEN-R0410 JULY 2004

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Formal Analysis of a Fair Payment Protocol

ABSTRACT
We formally specify a payment protocol. This protocol is intended for fair exchange of time-sensitive data. Here the μCRL language is used to formalize the protocol. Fair exchange properties are expressed in the regular alternation-free μ-calculus. These properties are then verified using the finite state model checker from the CADP toolset. Proving fairness without resilient communication channels is impossible. We use the Dolev-Yao intruder, but since the conventional Dolev-Yao intruder violates this assumption, it is forced to comply to the resilient communication channel assumption.

2000 Mathematics Subject Classification: 68Q60; 68M12; 68N30
Keywords and Phrases: Fair exchange protocols; Model checking; Dolev-Yao intruder
Note: This work was carried out under project ACCOUNT: Accountability in Electronic Commerce Protocols. The first author is supported by an ERCIM fellowship.

Formal Analysis of a Fair Payment Protocol

Jan Cederquist and Muhammad Torabi Dashti

CWI, Amsterdam, The Netherlands
{Cederqui,Dashti}@cwi.nl

Abstract

We formally specify a payment protocol described in [14]. This protocol is intended for fair exchange of time-sensitive data. Here the μCRL language is used to formalize the protocol. Fair exchange properties are expressed in the regular alternation-free μ-calculus. These properties are then verified using the finite state model checker from the CADP toolset. Proving fairness without resilient communication channels is impossible. We use the Dolev-Yao intruder, but since the conventional Dolev-Yao intruder violates this assumption, it is forced to comply to the resilient communication channel assumption.

1 Introduction

A fair exchange protocol aims at exchanging items in a *fair* manner. Informally, fair means that all involved parties receive a desired item in exchange for their own, or neither of them does so. It has been shown that fair exchange is impossible without a trusted third party [10]. Vogt et al. describe a protocol for fair exchange of money for an item using customer's smart card as a trusted party [14]. This protocol considers time-sensitive items and is adapted for wireless and mobile applications which lack a reliable communication channel. Here a version of that protocol is considered. We describe, in contrast to [14], the exact contents of all messages. The protocol is formally specified and the fairness properties are verified using a finite-state model checker.

In comparison to other security issues, such as secrecy and authenticity, fairness has not been studied formally so intensively. There are however some notable exceptions. Shmatikov and Mitchell [13] use the finite state model checker Murφ to analyze fair exchange and contract signing protocols. They use an external intruder, based on the Dolev-Yao intruder, that collaborates with one of the participants to model the malicious participant. Liveness can in general not be expressed in the Murφ language. Most fairness properties can however be expressed as safety properties. But this is not the case with termination (of protocol). Termination thus relies on other arguments than a verification using Murφ. Kremer and Raskin [8] use a game based approach for verifying non-repudiation and fair exchange. They use *alternating transition systems* (ATS) to model protocols and *alternating temporal logic* (ATL) to express the requirements. The method is automated using the model checker

Mocha. They have no explicit intruder. Instead different versions of players are considered; honest and arbitrary. ATL then offers very neat ways of expressing all desired requirements, including liveness under fairness constraints. In ATS all players follow predetermined finite sequences of steps, including intruders (arbitrary versions of players). However due to complexity, we believe that, it would be impractical to describe an intruder (powerful enough) for the protocol investigated in our work in such a way. In [12] a non-repudiation protocol is modeled using CSP, and proofs are generated by hand. Belief logic is used to formalize a protocol in [15] and it is discussed what may be needed for the verification of non-repudiation protocols. In [2] the theorem prover Isabelle is used to model a non-repudiation protocol by an inductive definition and to prove some desired properties.

In our work we formally specify a payment protocol in the process algebraic language μCRL [7]. The idea of this protocol comes from [14], but there are some differences (see section 6). Fairness properties for this protocol are formulated in the regular alternation-free μ-calculus [9] and verified using the model checker EVALUATOR 3.0 [9] from the CADP tool set [6].

Our formalization in μCRL contains a Dolev-Yao intruder [5]. The intruder is not separated from malicious participants. Instead, we consider different versions of participants, honest and malicious, where a malicious participant is an intruder that has access to the participant's private key. Some fairness properties are liveness properties and to prove liveness properties resilient communication channels are needed. Since the Dolev-Yao intruder has complete control over network, some cooperation from the intruder is needed when verifying liveness properties (i.e. sent messages should eventually be delivered). This cooperation is obtained using fairness constraints on the labelled transition system generated from the protocol specification in μCRL. We have an intruder that can synthesize new messages from its knowledge, without being forced to do so. Moreover using fairness constraints, it is forced to comply to the resilient communication channel assumption.

One of the major problems during this work was state space explosion. Several techniques have been used to reduce the spaces and avoid generating the whole state spaces when possible (see section 4.8).

The rest of the paper is organized as follows. In section 2 we give an overview of properties for fair exchange protocols. The fair exchange protocol we investigate is described in section 3. In section 4 the formal analysis is described. Here the intruder model is presented and all properties verified using the model checker are given. Included in section 4 is also a brief description of some optimization techniques used to generate the state spaces. In section 5 the protocol is described in a practical context. The fairness properties for this "more practical" setting follow from fairness for the protocol described in section 3. Some concluding remarks are given in section 6. Finally in the appendices A and B, our formalization of the protocol in μCRL is presented.

2 Fair Exchange Protocols

We assume two parties A and B. When the protocol starts, both parties have an item m and a description h of what they would like to have in exchange for m. The notation m_A and m_B is used for A's and B's item, respectively, and $h(m_B)$ and $h(m_A)$ for the description of the item that A and B, respectively, would like to have in exchange for their own.

According to Asokan [1], a fair exchange protocol is a protocol satisfying *effectiveness*, *fairness*, *timeliness* and *non-repudiability*. Effectiveness means that if both parties behave according to the protocol and none of them want to abort during the protocol round (m_A satisfies $h(m_A)$ and m_B satisfies $h(m_B)$), then the protocol will terminate in a state where A has m_B and B has m_A. An exchange protocol is called fair if, when it has terminated, either A has received B's item and B has received A's item, or none of the parties have lost their items. Timeliness means that the protocol will terminate for all parties (that behave according to the protocol) and after the termination point the degree of achieved fairness will not change. Non-repudiability is, in general, not considered as a primary requirement for fair exchange protocols, and it is omitted here.

Asokan [1] distinguishes between *strong* and *weak* fairness. Strong fairness is the fairness described above. Weak fairness means that either strong fairness is achieved, or it is possible for a participant to prove to an outside party that an unfair situation has occurred. Pagnia et al. [11] extend Asokan's definitions by considering the parties' willingness to cooperate and compensation for suffered disadvantage. A protocol may thus guarantee different levels of fairness, with or without third party intervention, providing resolution procedures versus providing proofs to be used in external disputes.

3 Protocol Description

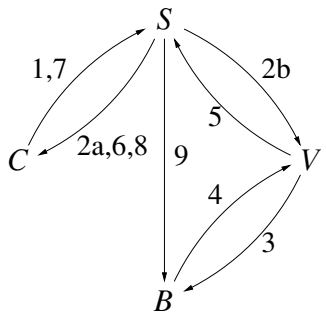
Here we describe the protocol which is to be analyzed in section 4. The protocol aims at fair exchange of time-sensitive data for some amount of money, between a customer (C) and a vendor (V). The exchange uses a bank (B) as a trusted online payment system and a trusted smartcard (S) attached to C. S is a tamper-proof hardware. The identity of S is however not necessarily known by V. Moreover, C is assumed to have a secure communication channel with S. When the protocol starts, V has an item m and a description $h(m)$ of m is known publicly. C wants to buy m for the amount a. Note that the item m is assumed to be confidential and should not be revealed for untrusted parties unless they pay for it. Below we describe the intended scenarios of the protocol, when all participants are honest.

In the protocol description, $pay(C, V, a)$ means that C shall pay the amount a to V, $(m)_X$ is the notation for the message m signed by X (using X's private key), and $\{m\}_X$ is the notation for m encrypted for X (using X's public key). It is assumed that m comes along with $(m)_X$ and can be extracted by anyone. For an encrypted message $\{m\}_X$ only X can extract m. A publicly known hash

function h is used for describing items and payments.

The main scenario (when none of the participants want to abort the protocol) is described as follows:

$$
\begin{array}{rrl}
1. & C \to S: & pay(C,V,a), h(m) \\
 & S: & initiate(n) \\
[2a. & S \to C: & (h(m),t,n,v,a)_S\ ] \\
2b. & S \to V: & h(m), (pay(C,V,a),n)_S \\
3. & V \to B: & (pay(C,V,a),n)_S \\
 & B: & block(n) \\
4. & B \to V: & (\mathsf{T}, h((pay(C,V,a),n)_S))_B \\
 & V: & commit(n) \\
5. & V \to S: & \{(m,n)_v\}_S \\
6. & S \to C: & \mathsf{T}, n \\
7. & C \to S: & \mathsf{T}, n \\
 & S: & receive(n) \\
8. & S \to C: & m \\
9. & S \to B: & (n,\mathsf{T})_S \\
 & B: & transfer(n),\ terminate(n).
\end{array}
$$



1. C sends a query to S for buying item m from V for amount a. On this request, S generates a fresh nonce. In this way, a protocol session possesses a unique nonce. Implicitly, S also notes the time t.

2a. Implicitly, S sends the nonce associated to the request and time to C. Later on, in step 6, when S asks C if the item is still interesting, it just needs to send the nonce. This simplifies the formalization. The time information is signed by S to prevent C from changing it.

2b. S signs and forwards the request together with the nonce to V. Since S is trusted, this message will be sent only upon a request from C.

3. If V wants to sell m to C for price a, it forwards the request to B. B notices the signature of S, checks whether the nonce n is fresh and that C has the amount a in its account. If this is the case, the money is blocked on C's account.

4. B notifies V that a transfer of amount a from C's account to V's account is possible. After this step V knows S is trusted.

5. V informs S that C can buy m for the amount a. Since this message could be received in parts and then assembled, the item is signed to protect integrity of transferred data.

6. S validates the received item by comparing it with $h(m)$ and asks if C is still interested in the item.

7. If C still wants the item, it answers T. The possibility to reject items is described in another scenario below.

8. S sends the item m to C.

9. S asks B to transfer the money, that was blocked on C's account, to V's account. On this request B performs the transaction.

(The actions *initiate*(n), *block*(n), *receive*(n), *transfer*(n) and *terminate*(n) are explained with more details in section 4.4, and so are the actions *unblock*(n) and *cancel*(n) below.)

There are some alternative scenarios of the protocol. When B receives a payment request, the nonce n may not be fresh or C may not have the required amount of money on its account:

$$\begin{array}{lll} 4^1. & B \to V: & (\mathsf{F}, h((pay(C,V,a),n)_S))_B \\ 5^1. & V \to S: & (n)_V \\ 6^1. & S \to C: & \mathsf{F}, n \\ 7^1. & S \to B: & (n, \mathsf{F})_S \\ & B: & unblock(n),\ terminate(n). \end{array}$$

4^1. B notifies V that the transaction is not possible.

5^1. V informs S that C cannot buy m for the amount a.

6^1. S informs C that it cannot buy m for the amount a.

7^1. S asks B to unblock money at C's account. If the money was blocked earlier, with the same nonce, B unblocks it.

If V does not want to sell m to C for the amount a, step 5^1 follows immediately after step $2b$.

After step 2 and before step 6 (6^1), C has the possibility to cancel the payment. This prevents V from blocking C's money without sending the item to S:

$$\begin{array}{lll} & C: & cancel(n) \\ 6^2. & C \to S: & n \\ 7^2. & S \to B: & (n, \mathsf{F})_S \\ 8^2. & S \to V: & (\mathsf{F}, n)_S \\ & B: & unblock(n),\ terminate(n). \end{array}$$

S erases the session information after sending *unblock* (or *transfer*) commands to B, and does not consider any message with a nonce from completed sessions. In our model, messages do not contain the intended receiver's name, so different patterns are used to indicate different recipients (i.e.B or V above) and preventing type flaw attacks by intruder.

In exchange of items whose value may change during time, the protocol provides a possibility for C to reject items in case of (intentional) delay in delivery. So, C can answer F after step 6:

$$\begin{array}{lll} 7^3. & C \to S: & \mathsf{F}, n \\ 8^3. & S \to B: & (n, \mathsf{F})_S \\ & B: & unblock(n),\ terminate(n). \end{array}$$

After step $2b$, S can perform a *timeout*:

$$\begin{array}{rll} & S: & \mathit{timeout} \\ 3^4. & S \to C: & \mathsf{F}, n \\ 4^4. & S \to V: & (\mathsf{F}, n)_S \\ 5^4. & S \to B: & (n, \mathsf{F})_S \\ & B: & \mathit{unblock}(n),\ \mathit{terminate}(n). \end{array}$$

The timeout forces a time limit on the steps $2b$–7, it prevents in particular C from waiting arbitrarily before answering in step 7. Concerning timeout, our description is non-deterministic. But it can also be assumed that S reads the start time t, that was sent to C in step $2a$, and that it has a limit Δt either hard coded in S or provided by V. If the current time is greater than $t + \Delta t$, it generates a timeout.

4 Formal Analysis

The formalization of the protocol described in section 3 is carried out in μCRL [7]. The μCRL toolset includes an automatic state space (labelled transition systems) generator and symbolic state space reduction tools. The properties effectiveness, timeliness and fairness are expressed in the regular alternation-free μ-calculus [9]. The model checker EVALUATOR 3.0 [9] from the CADP tool set [6] is then used to verify these properties (the formulas 1 to 12, in the sections 4.4 to 4.7).

For fair exchange protocols, beside protection from external intruders, the participants need to be protected from each other. In our formal model(s), we have three cases: (i) both C and V behave according to the protocol, (ii) C is malicious (C is the attacker) and (iii) V is malicious (V is the attacker). In the cases (ii) and (iii) all messages go via the attacker, with exception of the messages between C and S, which are sent over a secure link. When verifying effectiveness, case (i) is considered. All other properties are verified for the cases (ii) and (iii). A formalization of the protocol, in case (ii), in the process algebraic language μCRL is given in appendix A.

4.1 The μCRL specification language

Here we briefly describe the symbols used in the formalization below. For a complete description of the syntax and semantics for μCRL we refer to [7].

The symbols . and + are used for the sequential and alternative composition operator, respectively. The sum and product operators $\sum_{d \in D} P(d)$ and $\prod_{d \in D} P(d)$ behave like $P(d_1) + P(d_2) + \cdots$ and $P(d_1).P(d_2).\cdots$, respectively. The process expression **if** b **then** p **else** q, where b is a term of sort **bool** and, p and q are processes, behaves like p if b is true, and like q if b is false. Finally, the constant δ expresses that, from now on, no action can be performed.

The notations $\mathit{send}(x, m, y)$ and $\mathit{recv}(x, m, y)$ are used for the actions "X sends message m to Y" and "Y receives message m from X", respectively. In our model, *send* and *recv* actions are synchronized, i.e. X can only perform $\mathit{send}(x, m, y)$ if Y at the same time performs $\mathit{recv}(x, m, y)$ and vice versa.

This synchronization point is denoted $com(x, m, y)$ (in section 3, the notation $X \to Y : m$ was used for that).

4.2 Regular Alternation-free μ-calculus

The regular alternation-free μ-calculus is used here to formulate properties of (states in) labelled transition systems (see the sections 4.4–4.7). It is a fragment of μ-calculus that can be efficiently checked. Here we just briefly describe what is needed for expressing the fairness properties of the protocol we investigate. For a complete description of the syntax and semantics we refer to [9]. The regular alternation-free μ-calculus is built up from three types of formulas: *action formulas*, *regular formulas* and *state formulas*. We use '.', '$\vee$' and '*' for concatenation, choice and transitive-reflexive closure, respectively, for regular formulas. F and T are used in both action formulas and state formulas. In action formulas they represent *no action* and *any action*, respectively. The meaning of F and T in state formulas are the empty set and the entire state space, respectively. The operators $\langle \cdots \rangle$ and $[\cdots]$ have their usual meaning ($\Diamond$ and $\Box$ in modal logics). Finally, μ is the minimal fixed point operator.

4.3 Intruder Models

We consider the Dolev-Yao intruder [5]. It can remember all messages that have been transmitted over network. It can decrypt and sign messages, if it knows the corresponding key. It can compose new messages from its knowledge. It can also remove or delay messages in favour of others being communicated. Moreover, in cases where an agent does not know the identity of another agent it communicates with (for instance, V does not know the identity of S), the intruder can play the role of the second agent.

Below we define two intruder models in μCRL, I and I'. Both of them are equivalent to the Dolev-Yao intruder, but they behave differently under fairness constraints. I is used when verifying safety properties and I' when verifying liveness properties. The reason for using both of them is that I is not suitable for liveness properties, and I' is expensive to use when generating state spaces (see section 4.8).

The intruder I acts as customer (or vendor), intruder and network. All messages (x) are sent to I explicitly. I decomposes (*decomp*) the messages and adds the pieces to its knowledge (X). I then uses its knowledge to synthesize (*synth*) new messages. In general, how well the decomposition and the synthesis work depend on what private keys the intruder knows (abilities to sign and decrypt messages), *decomp* and *synth* are thus parameterized over known keys.

For efficiency reasons the union $\bigcup$ also depend on known private keys.

$$
\begin{aligned}
I(X) = (\ & \textstyle\sum_{p \in Player, x \in Message} \\
& \quad recv(p, x, i).I(X \bigcup_i decomp_i(x)) + \\
& \quad \textbf{if } synth_i(x, X) \\
& \quad \textbf{then } send(i, x, p).I(X) \\
& \quad \textbf{else } \delta\ ) + \\
& (\ \textstyle\sum_{m \in Item} \\
& \quad \textbf{if } synth_i(m, X) \\
& \quad \textbf{then } got\text{–}hold\text{–}of(m).I(X) \\
& \quad \textbf{else } \delta\ )
\end{aligned}
$$

In order to prove liveness properties, resilient communication channels[1] are assumed. In fact, without this assumption fair exchange is not possible, because then the attacker can simply choose to never send the item to one of the participants. In the presence of an intruder, resilient communication channels are obtained by imposing fairness constraints[2] on the labelled transition system generated from the protocol specification. These fairness constraints are expressed directly in regular μ-calculus formulas (see property 7 in section 4.6). The use of fairness constraints makes the model checker "skip circuits" and, in particular, it eventually forces the intruder to try to synthesize and send messages whenever there is a recipient. Some amount of cooperation from the intruder is usually needed in order to prove liveness properties. But, the fact that the intruder I does not forget anything and its abilities to construct messages itself together with fairness constraints can make "too many" liveness properties true. In fact, an erroneous protocol that does not terminate without intruder, may terminate with the intruder I and fairness constraints.

The second intruder I' can synthesize new messages from its knowledge, without being forced to do so. Moreover using fairness constraints, it is forced to comply to the resilient communication channel assumption. It is parameterized over a set of "resilient links" and all messages sent over these links should eventually be delivered. In our case the resilient link is the link between S and B. The corresponding messages are represented by the set Z. As I, I' gathers a set X of knowledge by intercepting all communications. But, it can explicitly forget pieces from this knowledge. The intruder uses a separate buffer (sorted list Y) of messages transmitted over the resilient links. When fairness constraints are used, the intruder is forced to eventually send all messages from this buffer. The resilient channel assumption will thus be preserved. Since I' can forget, it is not forced to generate new messages. However, this does not restrict the intruder's power in general, as it has the choice of keeping its

[1] All sent messages will eventually be delivered.

[2] We are using two notions of fairness; fairness of a protocol and fairness constraints of a labelled transition system. The second one is used to describe "fair" execution traces. In our case, a trace is fair when no possibilities are excluded forever. Then only fair execution traces are considered when proving the desired (liveness) property. To avoid confusion, we refer to these two notions as "fairness" and "fairness constraints".

knowledge as well.

$$
\begin{aligned}
I'(X,Y) = (\ & \textstyle\sum_{p \in Player, x \in Message} \\
& \quad recv(p,x,i). \\
& \quad \textbf{if } x \in Z \\
& \quad \textbf{then } I'(decomp_i(x) \textstyle\bigcup_i X, insert(x,Y)) \\
& \quad \textbf{else } I'(decomp_i(x) \textstyle\bigcup_i X, Y)\ ) + \\
(\ & \textstyle\sum_{x \in Message} \\
& \quad \textbf{if } x \in X \\
& \quad \textbf{then } I'(X \setminus \{x\}, Y) \\
& \quad \textbf{else } \delta\ ) + \\
(\ & \textstyle\sum_{x \in Message, p \in Player} \\
& \quad \textbf{if } x \in Y \vee synth_i(x,X) \\
& \quad \textbf{then } send(i,x,p).I'(X, remove(x,Y)) \\
& \quad \textbf{else } \delta\ )
\end{aligned}
$$

Table 1 summarizes the discussion above.

Intruder model	Violates resilient network assumption?	Regarding liveness properties	Regarding safety properties
Dolev-Yao (D-Y)	Yes	Sound, not complete	Equiv. to D-Y
I'	No	Sound and complete	Equiv. to D-Y
I	No	Complete, not sound	Equiv. to D-Y

Table 1: Intruders Under Fairness Constraints

4.4 Abstract Actions

For termination, the "abstract" action $terminate(n)$ in B (where n is a nonce) is used, instead of actual termination points for the users (C and V). This action is used because it is convenient to abstract away from messages to the users saying that a protocol round is terminated. This abstraction is safe since, if such messages had been used, the resilient communication channel assumption would have guaranteed their delivery. Thus $terminate(n)$ implies that the users terminate. Also note, the protocol may continue after $terminate(n)$ with a another protocol round, using another (fresh) nonce.

The CADP toolset [6] that we use to analyze the labelled transition system generated from a μCRL specification does not allow variables in action parameters, in regular μ-calculus formulas. So, properties containing variables should actually be checked for each combination of constants. To avoid this, the protocol is extended with abstract actions ($initiate(n)$, $block(n)$, $receive(n)$, $transfer(n)$,...) that can be used instead of actions containing more variables. In fact, each protocol session is associated to a nonce, so abstract actions (which only contain nonces) are enough for expressing most interesting properties of the protocol. Besides, they highlight implicit steps in the protocol and render more readable properties. However, the meaning of some of these abstract actions need to be uniquely defined. We start with $block(n)$. Without loss of

generality we can assume that $block(n)$ happens at the same time (or immediately after) B receives $(pay(C, V, a), n)_S$. So, $block(n)$ can be defined as *the amount a is blocked* (for nonce n). The fact that a indeed is the correct amount (the amount C is willing to pay) follows from

$$\begin{array}{l}[\mathsf{T}^*.com(s, (h(m), (pay(C, V, a_1), n)_S), i).\\ \ \mathsf{T}^*.com(i, (pay(C, V, a_2), n)_S, b)]\mathsf{F},\end{array} \tag{1}$$

where a_1 and a_2 are different amounts, and i is either c or v, depending on who is malicious. We define $transfer(n)$ and $unblock(n)$ to mean that the amount, which was blocked in $block(n)$, is transfered and unblocked, respectively. Now we turn to $receive(n)$. It can be assumed that $receive(n)$ happens at the same time as S sends an item m to C. That this item is the correct item (the item C ordered) follows from

$$\begin{array}{l}[\mathsf{T}^*.com(c, (pay(C, V, a), h(m_1)), s).initiate(n).\\ \ \mathsf{T}^*.receive(n).com(s, m_2, c)]\mathsf{F},\end{array} \tag{2}$$

where m_1 and m_2 are different items.

A malicious customer could possibly get hold of an item m by other means than from S in action $com(s, m, c)$. To show that this is not the case, we verify

$$[(\neg com(s, m, c))^*.got\text{–}hold\text{–}of(m)]\mathsf{F}, \tag{3}$$

where $got\text{–}hold\text{–}of(m)$ is an abstract action that occur if the malicious customer manages to synthesize the item m from gained knowledge.

4.5 Effectiveness

For effectiveness all participants are assumed to be honest and none of them want to abort the protocol. First, termination is inevitable

$$[\mathsf{T}^*.initiate(n)]\mu X(\langle\mathsf{T}\rangle\mathsf{T} \wedge [\neg terminate(n)]X), \tag{4}$$

for an arbitrary nonce n. Second, if S does not timeout, V does not say that C cannot buy the item, C does not answer F when S asks if the item is still valuable, and C does not cancel the payment, then the money will be transfered to V upon termination:

$$\begin{array}{l}[(\neg(timeout \vee com(v, (n)_v, s) \vee com(c, (\mathsf{F}, n), s) \vee cancel(n) \vee\\ \quad transfer(n)))^*.terminate(n)]\mathsf{F}.\end{array} \tag{5}$$

Under the same conditions, the item will also be received:

$$\begin{array}{l}[(\neg(timeout \vee com(v, (n)_v, s) \vee com(c, (\mathsf{F}, n), s) \vee cancel(n) \vee\\ \quad receive(n)))^*.terminate(n)]\mathsf{F}.\end{array} \tag{6}$$

4.6 Timeliness

For termination we verify that each fair trace eventually reaches $terminate(n)$[3]:

$$[\mathsf{T}^*.initiate(n).(\neg terminate(n))^*]\langle \mathsf{T}^*.terminate(n)\rangle \mathsf{T}. \tag{7}$$

Also, the degree of fairness does not change after termination:

$$\begin{array}{l}[\mathsf{T}^*.terminate(n).\mathsf{T}^*.\\ \ (receive(n) \vee block(n) \vee unblock(n) \vee transfer(n))]\mathsf{F}.\end{array} \tag{8}$$

4.7 Fairness[4]

For the properties that guarantee fairness it is important that the protocol terminates, which is part of timeliness, section 4.6.

Here we split up the notion of fairness (introduced in section 2) into fairness for C and V individually. We say that the protocol is fair for C if, whenever C pays for an item, C will receive it (V potentially being malicious). Fairness for V is defined correspondingly. Fairness for C is thus formalized as

$$[(\neg receive(n))^*.transfer(n).(\neg receive(n))^*.terminate(n)]\mathsf{F}. \tag{9}$$

From C's point of view it is also important that if money for an item is blocked and C does not receive the item, the block will be removed. The following property may thus also be considered as fairness for C:

$$[\mathsf{T}^*.block(n).(\neg(receive(n) \vee unblock(n)))^*.terminate(n)]\mathsf{F}. \tag{10}$$

Fairness for V means that if an item (corresponding to the nonce n) is received, money will be transfered:

$$[(\neg transfer(n))^*.receive(n).(\neg transfer(n))^*.terminate(n)]\mathsf{F}. \tag{11}$$

From V's point of view it is also important that the protocol terminates even if C does not cancel or respond when S asks if the item is still valuable:

$$\begin{array}{l}[\mathsf{T}^*.commit(n)]\\ \langle(\neg(com(c,(\mathsf{T},n),s) \vee com(c,(\mathsf{F},n),s) \vee cancel(n)))^*.\\ \ terminate(n)\rangle \mathsf{T}.\end{array} \tag{12}$$

4.8 Model Checking Details

One of the major obstacles during this work was state space explosion. The case when the customer is malicious turned out to be most difficult to generate. Our experiments show that this is mainly due to different knowledge of the intruder, gathered during different execution traces. A common abstraction technique in such situations is to make the intruder's knowledge more uniform.

[3]Whenever $terminate(n)$ has not occurred, there is a path leading to $terminate(n)$

[4]The notion of fairness we are proving corresponds to F_5 in the hierarchy of fairness guarantees described in [11].

We do that explicitly by, at the end of a protocol round, giving the intruder information about traces that were not taken (see appendix A, $reveal(n)$). More traces will now end up in same states, with a smaller state space as result. This extra information for the intruder should be chosen carefully though to avoid "false attacks". For safety properties this technique is sound, since the intruder just becomes more powerful. For liveness properties however, it is not sound. This technique may make "too many" liveness properties true. When assuming fairness constraints, an intruder with more knowledge provides more possibilities to reach a state. Instead, when generating the state space for proving liveness properties, we explicitly put a δ (deadlock) immediately after the action we want always to be reached. In this way, large parts of the state space will never be generated. In addition to these two methods, all actions except for the ones used in the properties are "hidden"[5] and symbolic reduction techniques from the μCRL toolset (see [3] for a description of these techniques) are applied to reduce the state spaces.

Using the techniques described above we could generate the state spaces and prove the safety properties, with 3 nonces (up to 3 concurrent protocol sessions) and 2 different items (also 1 nonce, 2 different items with 2 possible different prices), in the malicious customer[6] and malicious vendor cases. For liveness properties (termination in case of malicious customer and vendor), it was impossible to consider concurrent protocol sessions. The state spaces were generated for 2 items and 2 prices.

5 Practical Considerations

It is usually not possible for a smart card to receive large chunks of data, and store or process them. This limitation can cause practical problems when the item is some large software, and the smart card should store and validate it by comparing it with a preknown hash value. On the other hand, if the item validation phase is removed, it is not clear how the vendor is prevented from sending fake items. Here we suggest employing an offline Item Validation Party (IVP) that guarantees correspondence between an encrypted item and the description of the item. The protocol in section 3 is modified to

$$
\begin{array}{rll}
0a. & C \rightarrow V: & h(m) \\
0b. & V \rightarrow C: & \{m\}_{pk}, (h(m), h(\{m\}_{pk}), pk)_{IVP}. \\
1. & C \rightarrow S: & pay(C,V,a), pk \\
[2a. & S \rightarrow C: & (pk,t,n,v,a)_S\ ] \\
2b. & S \rightarrow V: & pk, (pay(C,V,a), n)_S \\
3. & V \rightarrow B: & (pay(C,V,a), n)_S
\end{array}
$$

[5] The properties 1, 2, 3 and 12 were verified using abstract actions instead of (but equivalent to) the concrete ones written in the formulas. This has to do with the hiding of actions before state space generation. In our μCRL specification all concrete actions have the same type, so either all of them are hidden or none. So by using only abstract actions in a property, all concrete actions can be hidden.

[6] For this case a distributed implementation of the μCRL toolset was used.

4. $B \rightarrow V: \quad (\mathsf{T}, h((pay(C,V,a),n)_S))_B$
5. $V \rightarrow S: \quad \{(sk,n)_V\}_S$
6. $S \rightarrow C: \quad \mathsf{T}, n$
7. $C \rightarrow S: \quad \mathsf{T}, n$
8. $S \rightarrow C: \quad sk$
9. $S \rightarrow B: \quad (n, \mathsf{T})_S.$

In this scenario it is assumed that V generates key pairs (pk,sk) for encryption and decryption of items. There is also an IVP which validate encryptions offline. When C asks for an item with description $h(m)$, it will receive m encrypted with pk along with a certificate from IVP. In this way, C can validate the item before decrypting it. Then C buys the decryption key sk from V using the protocol (in section 3), where the public key pk replaces the description of the item $h(m)$ (in message 1) and S checks that sk and pk match.

Assuming perfect cryptography and that the key pairs (pk,sk) are used only once, makes it safe to abstract away from the two initial messages $0a$ and $0b$. Consequently, correctness of this protocol follows from correctness of the protocol described in section 3, which was treated formally.

6 Conclusion

We have formally specified a payment protocol and verified its fairness properties. The idea of the protocol comes from [14], but there are some differences. A version of the protocol that has an online payment system (bank) is considered. We implement revocable payments using *block*, *unblock* and *transfer* (as described in section 3). To protect the vendor from the customer being passive, the smartcard can *timeout* (without *timeout* property 12 does not hold). We have also relaxed the assumptions on the communication links.

We have implemented an intruder that, for safety properties, is equivalent to the Dolev-Yao intruder (which is the most powerful intruder, see [4]). Our intruder is also suitable for verifying liveness properties, since it does not violate the resilient communication channel assumption under fairness constraints. It can be used in general purpose specification languages like μCRL.

One of the major obstacles during this work was state space explosion. Different reduction techniques were adopted when generating the state spaces for safety and liveness.

Model checking (for finite state analysis) requires a finite state space. So, only a finite number of concurrent protocol sessions can be verified. In general, after a finite analysis, nothing can be claimed about the security of a protocol for certain. However, although model checking can not be used to prove absolute security of a protocol in general, our case study confirms that it is helpful in finding flaws and can easily be utilized in a trial-and-error specification phase.

Acknowledgment

We are grateful to Wan Fokkink, Jaco van de Pol and Miguel Valero for discussions and comments on earlier versions of this report. We would also like to thank Stefan Blom for help with cluster machines.

References

[1] N. Asokan. *Fairness in electronic commerce*. PhD thesis, University of Waterloo, 1998.

[2] G. Bella and L. C. Paulson. Mechanical proofs about a non-repudiation protocol. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001*, volume 2152 of *LNCS*, pages 91–104. Springer-Verlag, September 2001.

[3] S. Blom, W. Fokkink, J. F. Groote, I van Langevelde, B. Lisser, and J. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *Proceedings of the 13th International Conference on Computer Aided Verification*, volume 2102 of *LNCS*, pages 250–254. Springer-Verlag, 2001.

[4] Iliano Cervesato. The Dolev-Yao Intruder is the Most Powerful Attacker. In J. Halpern, editor, *16th Annual Symposium on Logic in Computer Science — LICS'01*, Boston, MA, 16–19 June 2001. IEEE Computer Society Press.

[5] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, March 1983.

[6] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 437–440. Springer-Verlag, 1996.

[7] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. SV, 1995.

[8] S. Kremer and J. Raskin. A game-based verification of non-repudiation and fair exchange protocols. In *Proceedings of the 12th International Conference on Concurrency Theory*, volume 2154 of *LNCS*, pages 551–565. Springer-Verlag, 2001.

[9] R. Mateescu. Efficient diagnostic generation for boolean equation systems. In *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2000*, volume 1785 of *LNCS*, pages 251–265. Springer-Verlag, March 2000.

[10] H. Pagnia and F. C. Gärtner. On the impossibility of fair exchange without a trused third party. Technical Report TUD-BS-1999-02, Department of Computer Science, Darmstadt University of Technology, 1999.

[11] H. Pagnia, H. Vogt, and F. C. Gärtner. Fair exchange. *The Computer Journal*, 46(1):55–7, 2003.

[12] S. Schneider. Formal analysis of a non-repudiation protocol. In *Proceedings of The 11th Computer Security Foundations Workshop*, pages 54–65. IEEE Computer Society Press, 1998.

[13] V. Shmatikov and J. C. Mitchell. Finite-state analysis of two contract signing protocols. *Theor. Comput. Sci.*, 283(2):419–450, 2002.

[14] H. Vogt, H. Pagnia, and F. C. Gärtner. Using smart cards for fair exchange. In *Electronic Commerce – WELCOM 2001*, volume 2232 of *LNCS*, pages 101–113. Springer-Verlag, 2001.

[15] J. Zhou and D. Gollmann. Towards verification of non-repudiation protocols. In *International Refinement Workshop and Formal Methods Pacific '98: Proceedings of IRW/FMP '98, Discrete Mathematics and Theoretical Computer Science Series*, pages 370–380. Springer-Verlag, 1998.

A Formal Description of the Participants

Here we present the μCRL formalization of the case when C act as customer, network and intruder, and V is honest. The code for C is presented in section 4.3 as I and I'. Note that, in B and V, all messages are sent to and received from C. Also, $reveal(n)$ (in S) is only used when generating the state space for safety properties. When generating the state space for proving termination, a δ is put after each $terminate(n)$ in the corresponding *linear process operator* (see [3]).

$$
\begin{array}{l}
S(i) = \\
\quad \textbf{if } i = 3 \\
\quad \textbf{then } \delta \\
\quad \textbf{else } \sum_{m \in Item, d \in Price, p \in \{c,v\}} \\
\qquad (\ recv(c, (pay(c,p,d), h(m)), s). \\
\qquad\quad initiate(n_i). \\
\qquad\quad send(s, (h(m), (pay(c,p,d), n_i)_s), c). \\
\qquad\quad (\ recv(c, \{(m, n_i)_p\}_s, s). \\
\qquad\qquad send(s, (\mathsf{T}, n_i), c). \\
\qquad\qquad (\ recv(c, (\mathsf{T}, n_i), s).receive(n_i).send(s, m, c).send(s, (n_i, \mathsf{T})_s, c) + \\
\qquad\qquad\quad recv(c, (\mathsf{F}, n_i), s).send(s, (n_i, \mathsf{F})_s, c) + \\
\qquad\qquad\quad timeout.send(s, (\mathsf{F}, n_i), c).send(s, (n_i, \mathsf{F})_s, c)\ ) + \\
\qquad\qquad recv(c, (n_i)_p, s).send(s, (\mathsf{F}, n_i), c).send(s, (n_i, \mathsf{F})_s, c) + \\
\qquad\qquad timeout.send(s, (\mathsf{F}, n_i), c).send(s, (n_i, \mathsf{F})_s, c) + \\
\qquad\qquad recv(c, n_i, s).send(s, (n_i, \mathsf{F})_s, c).send(s, (n_i, \mathsf{F})_s, c)\ ).reveal(n)\ ).S(i+1)
\end{array}
$$

where

$$
\begin{array}{l}
reveal(n) = send(s, (n, \mathsf{T})_s, c). \\
\qquad\qquad \prod_{m \in Item} send(s, \{(m, n)_v\}_s, c). \\
\qquad\qquad send(s, (n)_v, c). \\
\qquad\qquad \prod_{d \in Price} send(s, (\mathsf{F}, h((pay(c, v, d), n)_s))_b, c). \\
\qquad\qquad \prod_{d \in Price} send(s, (\mathsf{T}, h((pay(c, v, d), n)_s))_b, c)
\end{array}
$$

$$
\begin{aligned}
&B(X,Y) = \\
&\quad (\ \textstyle\sum_{d\in Price, n\in Nonce, p_c, p_v \in Player} \\
&\qquad recv(c,(pay(p_c,p_v,d),n)_s,b). \\
&\qquad \textbf{if } n \in X \\
&\qquad \textbf{then } send(b,(\mathsf{F},h((pay(p_c,p_v,d),n)_s))_b,c).B(X,Y) \\
&\qquad \textbf{else } block(n).send(b,(\mathsf{T},h((pay(p_c,p_v,d),n)_s))_b,c).B(X,Y\cup\{n\})\ ) + \\
&\quad (\ \textstyle\sum_{n\in Nonce} \\
&\qquad recv(c,(n,\mathsf{F})_s,b). \\
&\qquad \textbf{if } n \in Y \\
&\qquad \textbf{then } unblock(n).terminate(n).B(X\cup\{n\},Y\setminus\{n\}) \\
&\qquad \textbf{else } terminate(n).B(X\cup\{n\},Y) + \\
&\qquad recv(c,(n,\mathsf{T})_s,b). \\
&\qquad \textbf{if } n \in Y \\
&\qquad \textbf{then } transfer(n).terminate(n).B(X\cup\{n\},Y\setminus\{n\}) \\
&\qquad \textbf{else } terminate(n).B(X\cup\{n\},Y)\ )
\end{aligned}
$$

$$
\begin{aligned}
&V = (\ \textstyle\sum_{m\in Item, d\in Price, n\in Nonce, p_s, p_c \in Player} \\
&\qquad recv(c,h(m,(pay(p_c,v,d),n)_{p_s}),v). \\
&\qquad (\ send(v,(n)_v,c) + \\
&\qquad\quad (\ send(v,(pay(p_c,v,d),n)_{p_s},c). \\
&\qquad\qquad (\ recv(c,(\mathsf{T},h(pay(p_c,v,d),n)_{p_s})_b,v). \\
&\qquad\qquad\quad (\ commit(n).send(v,\{(m,n)_v\}_{p_s},c) + \\
&\qquad\qquad\qquad recv(c,(\mathsf{F},n)_{p_s},v)\ ) + \\
&\qquad\qquad\quad recv(c,(\mathsf{F},h(pay(p_c,v,d),n)_{p_s})_b,v). \\
&\qquad\qquad\quad (\ send(v,(n)_v,c)\ +\ recv(c,(\mathsf{F},n)_{p_s},v)\ )\ )\ )\ )\ ).V
\end{aligned}
$$

B Formalization of the Protocol in μCRL

B.1 Data Type Definitions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Bool Data Type
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    Bool
func    T,F: -> Bool
map     and,or: Bool # Bool -> Bool
        not: Bool -> Bool
        eq: Bool # Bool -> Bool
        if: Bool # Bool # Bool -> Bool
var     x,x': Bool
rew     and(T,T)=T
        and(F,x)=F
        and(x,F)=F
        or(T,x)=T
        or(x,T)=T
        or(F,F)=F
        not(F)=T
        not(T)=F
        eq(x,x)=T
        eq(T,F)=F
        eq(F,T)=F
```

```
        if(T,x,x')=x
        if(F,x,x')=x'
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Natural Number Data Type
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    Nat
func    zero: -> Nat
        succ: Nat -> Nat
map     eq: Nat # Nat -> Bool
        leq:  Nat # Nat -> Bool
var     m,n: Nat
rew     eq(zero,zero) = T
        eq(zero,succ(n)) = F
        eq(succ(n),zero) = F
        eq(succ(n),succ(m)) = eq(n,m)
        leq(zero,m)=T
        leq(succ(m),zero)=F
        leq(succ(m),succ(n))=leq(m,n)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Item Data Type.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    Item
func    d1,d2: -> Item
map     eq: Item # Item -> Bool
        leq: Item # Item -> Bool
var     d: Item
rew     eq(d,d)=T
        eq(d2,d1)=F
        eq(d1,d2)=F
        leq(d,d)=T
        leq(d1,d)=T
        leq(d2,d1)=F
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Price Data Type
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    Price
func    c1,c2 :-> Price
map     eq: Price # Price -> Bool
        leq: Price # Price -> Bool
var     pd: Price
rew     eq(pd,pd)=T
        eq(c1,c2)=F
        eq(c2,c1)=F
        leq(pd,pd)=T
        leq(c1,pd)=T
        leq(c2,c1)=F
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Player Data Type
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    Player
func    s,v,c,b :-> Player
map     eq : Player # Player->Bool
        leq: Player # Player->Bool
var     p: Player
rew     eq(p,p)=T
        eq(s,v)=F       eq(s,c)=F       eq(s,b)=F
        eq(v,s)=F       eq(v,c)=F       eq(v,b)=F
```

```
        eq(c,s)=F       eq(c,v)=F       eq(c,b)=F
        eq(b,s)=F       eq(b,v)=F       eq(b,c)=F
        leq(p,p)=T
        leq(b,c)=T      leq(b,s)=T      leq(b,v)=T
        leq(c,b)=F      leq(c,s)=T      leq(c,v)=T
        leq(s,b)=F      leq(s,c)=F      leq(s,v)=T
        leq(v,b)=F      leq(v,c)=F      leq(v,s)=F
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Key Data Type
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    Key
func    key:Player->Key
map     eq:Key#Key->Bool
        leq:Key#Key->Bool
        known:Player # Key->Bool
var     x,y:Player
rew     eq(key(x),key(y))=eq(x,y)
        leq(key(x),key(y))=leq(x,y)
        known(x,key(y))=eq(x,y)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Nonce Data Type
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    Nonce
func    nnc : Nat -> Nonce
map     eq: Nonce # Nonce -> Bool
        NonceGen : Nonce -> Nonce
        if: Bool # Nonce # Nonce -> Nonce
        leq: Nonce # Nonce -> Bool
var     i,j : Nat
        x,y: Nonce
rew     eq(nnc(i),nnc(j))=eq(i,j)
        NonceGen(nnc(i))=if(eq(i,succ(succ(succ(zero)))),
                            nnc(zero),
                            nnc(succ(i)))
        if(T,x,y)=x
        if(F,x,y)=y
        leq(nnc(i),nnc(j))=leq(i,j)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Message Data Type
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    Message
func    bol:Bool-> Message
        it:Item->Message
        nonce:Nonce->Message
        pay:Player#Player#Price->Message
        pair:Message#Message->Message
        hash:Message->Message
        sign:Key#Message->Message
        enc:Key#Message->Message
map     eq:Message#Message->Bool
        leq:Message#Message->Bool
var     k1,k2:Key
        m1,m2,m3,m4:Message
        a,a' : Bool
        d,d':Item
        n,n':Nonce
        a1,a2,b1,b2:Player
```

```
        pd1, pd2: Price
rew     eq(m1,m1)=T
        eq(bol(a), bol(a'))=eq(a,a')
        eq(it(d),it(d'))=eq(d,d')
        eq(nonce(n),nonce(n'))=eq(n,n')
        eq(pay(a1,b1,pd1),pay(a2,b2,pd2))=and(and(eq(a1,a2),eq(b1,b2)),eq(pd1,pd2))
        eq(pair(m1,m2),pair(m3,m4))=and(eq(m1,m3),eq(m2,m4))
        eq(hash(m1),hash(m2))=eq(m1,m2)
        eq(sign(k1,m1),sign(k2,m2))=and(eq(k1,k2),eq(m1,m2))
        eq(enc(k1,m1),enc(k2,m2))=and(eq(k1,k2),eq(m1,m2))
        eq(bol(a), it(d))=F
        eq(bol(a), nonce(n))=F
        eq(bol(a), pay(a2,b2,pd2))=F
        eq(bol(a), pair(m3,m4))=F
        eq(bol(a), hash(m2))=F
        eq(bol(a), sign(k2,m2))=F
        eq(bol(a), enc(k2,m2))=F
        eq(it(d), bol(a))=F
        eq(it(d),nonce(n))=F
        eq(it(d),pay(a2,b2,pd2))=F
        eq(it(d),pair(m3,m4))=F
        eq(it(d),hash(m2))=F
        eq(it(d),sign(k2,m2))=F
        eq(it(d),enc(k2,m2))=F
        eq(nonce(n),bol(a))=F
        eq(nonce(n),it(d'))=F
        eq(nonce(n),pay(a2,b2,pd2))=F
        eq(nonce(n),pair(m3,m4))=F
        eq(nonce(n),hash(m2))=F
        eq(nonce(n),sign(k2,m2))=F
        eq(nonce(n),enc(k2,m2))=F
        eq(pay(a1,b1,pd1),bol(a))=F
        eq(pay(a1,b1,pd1),it(d'))=F
        eq(pay(a1,b1,pd1),nonce(n))=F
        eq(pay(a1,b1,pd1),pair(m3,m4))=F
        eq(pay(a1,b1,pd1),hash(m2))=F
        eq(pay(a1,b1,pd1),sign(k2,m2))=F
        eq(pay(a1,b1,pd1),enc(k2,m2))=F
        eq(pair(m1,m2),bol(a))=F
        eq(pair(m1,m2),it(d'))=F
        eq(pair(m1,m2),nonce(n))=F
        eq(pair(m1,m2),pay(a2,b2,pd2))=F
        eq(pair(m1,m2),hash(m3))=F
        eq(pair(m1,m2),sign(k2,m3))=F
        eq(pair(m1,m2),enc(k2,m3))=F
        eq(hash(m1),bol(a))=F
        eq(hash(m1),it(d'))=F
        eq(hash(m1),nonce(n))=F
        eq(hash(m1),pay(a2,b2,pd2))=F
        eq(hash(m1),pair(m3,m4))=F
        eq(hash(m1),sign(k2,m2))=F
        eq(hash(m1),enc(k2,m2))=F
        eq(sign(k1,m1),bol(a))=F
        eq(sign(k1,m1),it(d'))=F
        eq(sign(k1,m1),nonce(n))=F
        eq(sign(k1,m1),pay(a2,b2,pd2))=F
        eq(sign(k1,m1),pair(m3,m4))=F
```

```
        eq(sign(k1,m1),hash(m2))=F
        eq(sign(k1,m1),enc(k2,m2))=F
        eq(enc(k1,m1),bol(a))=F
        eq(enc(k1,m1),it(d'))=F
        eq(enc(k1,m1),nonce(n))=F
        eq(enc(k1,m1),pay(a2,b2,pd2))=F
        eq(enc(k1,m1),pair(m3,m4))=F
        eq(enc(k1,m1),hash(m2))=F
        eq(enc(k1,m1),sign(k2,m2))=F
%definition of leq
        leq(m1,m1)=T
        leq(bol(a),bol(a'))=or(not(a),a')
        leq(it(d),it(d'))=leq(d,d')
        leq(nonce(n),nonce(n'))=leq(n,n')
        leq(pay(a1,b1,pd1),pay(a2,b2,pd2))=
          if(eq(a1,a2),if(eq(b1,b2),leq(pd1,pd2),leq(b1,b2)),leq(a1,a2))
        leq(pair(m1,m2),pair(m3,m4))=if(eq(m1,m3),leq(m2,m4),leq(m1,m3))
        leq(hash(m1),hash(m2))=leq(m1,m2)
        leq(enc(k1,m1),enc(k2,m2))=if(eq(k1,k2),leq(m1,m2),leq(k1,k2))
        leq(sign(k1,m1),sign(k2,m2))=if(eq(k1,k2),leq(m1,m2),leq(k1,k2))
        leq(bol(a), it(d))=T
        leq(bol(a), nonce(n))=T
        leq(bol(a), pay(a2,b2,pd2))=T
        leq(bol(a), pair(m3,m4))=T
        leq(bol(a), hash(m2))=T
        leq(bol(a), sign(k2,m2))=T
        leq(bol(a), enc(k2,m2))=T
        leq(it(d), bol(a))=F
        leq(it(d),nonce(n))=T
        leq(it(d),pay(a2,b2,pd2))=T
        leq(it(d),pair(m3,m4))=T
        leq(it(d),hash(m2))=T
        leq(it(d),sign(k2,m2))=T
        leq(it(d),enc(k2,m2))=T
        leq(nonce(n),bol(a))=F
        leq(nonce(n),it(d'))=F
        leq(nonce(n),pay(a2,b2,pd2))=T
        leq(nonce(n),pair(m3,m4))=T
        leq(nonce(n),hash(m2))=T
        leq(nonce(n),sign(k2,m2))=T
        leq(nonce(n),enc(k2,m2))=T
        leq(pay(a1,b1,pd1),bol(a))=F
        leq(pay(a1,b1,pd1),it(d'))=F
        leq(pay(a1,b1,pd1),nonce(n))=F
        leq(pay(a1,b1,pd1),pair(m3,m4))=T
        leq(pay(a1,b1,pd1),hash(m2))=T
        leq(pay(a1,b1,pd1),sign(k2,m2))=T
        leq(pay(a1,b1,pd1),enc(k2,m2))=T
        leq(pair(m1,m2),bol(a))=F
        leq(pair(m1,m2),it(d'))=F
        leq(pair(m1,m2),nonce(n))=F
        leq(pair(m1,m2),pay(a2,b2,pd2))=F
        leq(pair(m1,m2),hash(m3))=T
        leq(pair(m1,m2),sign(k2,m3))=T
        leq(pair(m1,m2),enc(k2,m3))=T
        leq(hash(m1),bol(a))=F
        leq(hash(m1),it(d'))=F
```

```
        leq(hash(m1),nonce(n))=F
        leq(hash(m1),pay(a2,b2,pd2))=F
        leq(hash(m1),pair(m3,m4))=F
        leq(hash(m1),sign(k2,m2))=T
        leq(hash(m1),enc(k2,m2))=T
        leq(sign(k1,m1),bol(a))=F
        leq(sign(k1,m1),it(d'))=F
        leq(sign(k1,m1),nonce(n))=F
        leq(sign(k1,m1),pay(a2,b2,pd2))=F
        leq(sign(k1,m1),pair(m3,m4))=F
        leq(sign(k1,m1),hash(m2))=F
        leq(sign(k1,m1),enc(k2,m2))=T
        leq(enc(k1,m1),bol(a))=F
        leq(enc(k1,m1),it(d'))=F
        leq(enc(k1,m1),nonce(n))=F
        leq(enc(k1,m1),pay(a2,b2,pd2))=F
        leq(enc(k1,m1),pair(m3,m4))=F
        leq(enc(k1,m1),hash(m2))=F
        leq(enc(k1,m1),sign(k2,m2))=F
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Data Type for Knowledge of Intruder
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    PM
func    emptyM:->PM
        insert:Message#PM->PM
map     in:Message#PM->Bool
        synthadd:Player#Message#PM->PM
        add:Message#PM->PM
        rem:Message#PM->PM
        union:Player#PM#PM->PM
        if:Bool#PM#PM->PM
        eq:PM#PM->Bool
var     x,y:Message
        ms,ms1,ms2:PM
        U,V:PM
        p: Player
rew     eq(U,U)=T
        eq(U,insert(x,U))=F
        eq(emptyM,insert(x,U))=F
        eq(insert(x,U),emptyM)=F
        eq(insert(y,V),insert(x,U))=and(eq(x,y),eq(U,V))
        if(T,U,V)=U
        if(F,U,V)=V
        in(x,emptyM)=F
        in(x,insert(y,U))=or(eq(x,y),in(x,U))
% Messages are ordered in knowledge set.
        add(x,emptyM)=insert(x,emptyM)
        add(x,insert(y,U))=if(leq(x,y),
                              if(eq(x,y),insert(y,U),insert(x,insert(y,U))),
                              insert(y,add(x,U)))
        rem(x,emptyM)=emptyM
        rem(x,insert(y,U))=if(leq(x,y),if(eq(x,y),U,insert(y,U)),insert(y,rem(x,U)))
        union(p,emptyM,V)=V
        union(p,V,emptyM)=V
        union(p,insert(x,U),V)=synthadd(p,x,union(p,U,V))
% A Message is added to knowledge set only if can not be synthesized.
        synthadd(p,x,U)=if(synth(p,x,U),U,add(x,U))
```

```
% A Message is decomposed before being added to knowledge set.
map     decomp:Player#Message->PM
var     p,a,b':Player
        x:Bool
        d:Item
        n:Nonce
        m,m':Message
        k:Key
        pd1: Price
rew     decomp(p,bol(x))=emptyM
        decomp(p,it(d))=insert(it(d),emptyM)
        decomp(p,nonce(n))=emptyM
        decomp(p,pay(a,b',pd1))=emptyM
        decomp(p,pair(m,m'))=union(p,decomp(p,m),decomp(p,m'))
        decomp(p,hash(m))=emptyM
        decomp(p,sign(k,m))=synthadd(p,sign(k,m),decomp(p,m))
        decomp(p,enc(k,m))=if(known(p,k),decomp(p,m),insert(enc(k,m),emptyM))
% Intruder can synthesize new messages from its knowledge set according
% to the following rules.
map     synth:Player#Message#PM->Bool
var     ms:PM
        p,a,b':Player
        x:Bool
        d:Item
        n:Nonce
        m,m':Message
        k:Key
        pd1: Price
rew     synth(p,bol(x),ms)=T
        synth(p,it(d),ms)=or(in(it(d),ms),eq(p,v))
        synth(p,nonce(n),ms)=leq(n,nnc(succ(succ(zero))))
        synth(p,pay(a,b',pd1),ms)=T
        synth(p,pair(m,m'),ms)=and(synth(p,m,ms),synth(p,m',ms))
        synth(p,hash(m),ms)=if(or(eq(m,it(d2)),eq(m,it(d1))),
                               T,
                               synth(p,m,ms))
        synth(p,sign(k,m),ms)=or(and(known(p,k),synth(p,m,ms)),in(sign(k,m),ms))
        synth(p,enc(k,m),ms)=or(synth(p,m,ms),in(enc(k,m),ms))
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Data Type for Knowledge of Bank
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    Knowledge
func    emptset :-> Knowledge
        set: Nonce # Knowledge -> Knowledge
map     add: Nonce # Knowledge -> Knowledge
        rem: Nonce # Knowledge -> Knowledge
        test: Nonce # Knowledge -> Bool
        if: Bool # Knowledge # Knowledge -> Knowledge
        eq: Knowledge # Knowledge -> Bool
var     k, k': Knowledge
        a,a': Nonce
rew     add(a,set(a',k))=if(leq(a,a'),
                            if(eq(a,a'),set(a,k),set(a,set(a',k))),
                            set(a',add(a,k)))
        add(a,emptset)=set(a,emptset)
        test(a, emptset)=F
        test(a, set(a',k))=if(leq(a,a'),eq(a,a'),test(a,k))
```

```
        if(T, k, k')=k
        if(F, k, k')=k'
        eq(emptset,set(a,k))=F
        eq(set(a,k),emptset)=F
        eq(set(a,set(a',k)),set(a',set(a,k)))=T
        eq(set(a,k),set(a',k'))=and(eq(a,a'),eq(k,k'))
        eq(k,k)=T
        rem(a,set(a',k))=if(eq(a,a'),k,set(a',rem(a,k)))
        rem(a,emptset)=emptset
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Declaration of Actions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act     send,recv,com:Player#Message#Player
        timeout
% Abstract actions
        initiate: Nonce
        block: Nonce
        commit: Nonce
        cancel: Nonce
        receive: Nonce
        transfer: Nonce
        unblock: Nonce
        terminate: Nonce
        got-hold-of: Item
% Synchronization point
comm    send|recv=com
```

B.2 Honest Participants

In this section only μCRL code for players is presented. The data types are the same as in section B.1.

```
proc

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Customer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Customer=sum(d:Item,sum(pd:Price,
send(c,pair(pay(c,v,pd),hash(it(d))),s).
(
  sum(n:Nonce,
    (recv(s,pair(bol(T),nonce(n)),c).
      (
        send(c,pair(bol(T),nonce(n)),s).recv(s,it(d),c)
        +
        send(c,pair(bol(F),nonce(n)),s)
        +
        recv(s,pair(bol(F),nonce(n)),c)
      )
    )
    +
    recv(s,pair(bol(F),nonce(n)),c)
    +
    cancel(n).send(c,nonce(n),s)
  )
))).Customer

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Smart Card
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Scard(n:Nonce)=delta<|eq(n,nnc(succ(succ(succ(zero)))))|>
(
 sum(d:Item,sum(pd:Price,sum(p:Player,
 (
  recv(c,pair(pay(c,p,pd),hash(it(d))),s).
  initiate(n).
  send(s,pair(hash(it(d)),sign(key(s),pair(pay(c,p,pd),nonce(n)))),p).
  (
   (
    recv(p,enc(key(s),sign(key(p),pair(it(d),nonce(n)))),s).
    send(s,pair(bol(T),nonce(n)),c).
    (
     (
      recv(c,pair(bol(T),nonce(n)),s).
      receive(n).
      send(s,it(d),c).
      send(s,sign(key(s),pair(nonce(n),bol(T))),b)
     )
     +
     (
      recv(c,pair(bol(F),nonce(n)),s).
      send(s,sign(key(s),pair(nonce(n),bol(F))),b)
     )
     +
     timeout.
     send(s,pair(bol(F),nonce(n)),c).
     send(s,sign(key(s),pair(nonce(n),bol(F))),b)
    )
   )
   +
   (
    recv(p,sign(key(p),nonce(n)),s).
    send(s,pair(bol(F),nonce(n)),c).
    send(s,sign(key(s),pair(nonce(n),bol(F))),b)
   )
   +
   (
    recv(c,nonce(n),s).
    send(s,sign(key(s),pair(nonce(n),bol(F))),b).
    send(s,sign(key(s),pair(bol(F),nonce(n))),p)
   )
   +
   (
    timeout.
    send(s,pair(bol(F),nonce(n)),c).
    send(s,sign(key(s),pair(nonce(n),bol(F))),b).
    send(s,sign(key(s),pair(bol(F),nonce(n))),p)
   )
  )
 )
))).Scard(NonceGen(n))
)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Bank
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
Bank(U:Knowledge,G:Knowledge)=
(sum(n:Nonce,sum(pc:Player,sum(pv:Player,sum(pd:Price,
 (
  recv(pv,sign(key(s),pair(pay(pc,pv,pd),nonce(n))),b).
  (
   (
    send(b,sign(key(b),pair(bol(F),hash(sign(key(s),pair(pay(pc,pv,pd),nonce(n))))),pv).
    Bank(U,G)
   )
   <|test(n,G)|>
   (
    send(b,sign(key(b),pair(bol(T),hash(sign(key(s),pair(pay(pc,pv,pd),nonce(n))))),pv).
    block(n).
    Bank(add(n,U),G)
   )
  )
 )
 ))))
)
+
sum(n:Nonce,
 recv(s,sign(key(s),pair(nonce(n),bol(F))),b).
 (
  unblock(n).terminate(n).Bank(rem(n,U),add(n,G))
  <|test(n,U)|>
  terminate(n).Bank(U,add(n,G))
 )
 +
 recv(s,sign(key(s),pair(nonce(n),bol(T))),b).
 (
  transfer(n).terminate(n).Bank(rem(n,U),add(n,G))
  <|test(n,U)|>
  terminate(n).Bank(U,add(n,G))
 )
)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Vendor
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Vendor=
sum(n:Nonce,sum(pd:Price,sum(d:Item,sum(ps:Player,sum(pc:Player,
(
 recv(ps,pair(hash(it(d)),sign(key(ps),pair(pay(pc,v,pd),nonce(n)))),v).
 (
  send(v,sign(key(v),nonce(n)),ps)
  +
  (
   send(v,sign(key(ps),pair(pay(pc,v,pd),nonce(n))),b).
   (
    recv(b,sign(key(b),pair(bol(T),hash(sign(key(ps),pair(pay(pc,v,pd),nonce(n))))),v).
    (
     send(v,enc(key(ps),sign(key(v),pair(it(d),nonce(n)))),ps).commit(n)
     +
     recv(ps,sign(key(ps),pair(bol(F),nonce(n))),v)
    )
    +
    recv(b,sign(key(b),pair(bol(F),hash(sign(key(ps),pair(pay(pc,v,pd),nonce(n))))),v).
    (
```

```
     send(v,sign(key(v),nonce(n)),ps)
     +
     recv(ps,sign(key(ps),pair(bol(F),nonce(n))),v)
    )
   )
  )
 )
))))))).Vendor

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Instantiation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init  encap({send,recv},Customer||Scard(nnc(zero))||Bank(emptset,emptset)||Vendor)
```

B.3 Vendor Combined with Intruder, Honest Customer

In this section only μCRL code for players is presented. The data types are the same as in section B.1.

```
proc

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Customer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Customer=sum(pd:Price,sum(d:Item,
send(c,pair(pay(c,v,pd),hash(it(d))),s).
(
  sum(n:Nonce,
    (recv(s,pair(bol(T),nonce(n)),c).
      (
        send(c,pair(bol(T),nonce(n)),s).recv(s,it(d),c)
        +
        send(c,pair(bol(F),nonce(n)),s)
        +
        recv(s,pair(bol(F),nonce(n)),c)
      )
    )
    +
    recv(s,pair(bol(F),nonce(n)),c)
    +
    cancel(n).send(c,nonce(n),s)
  )
))).Customer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Smart Card
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Scard(n:Nonce)=delta<|eq(n,nnc(succ(succ(succ(zero)))))|>
(
 sum(d:Item,sum(p:Player,sum(pd:Price,
 (
  recv(c,pair(pay(c,p,pd),hash(it(d))),s).
  initiate(n).
  send(s,pair(hash(it(d)),sign(key(s),pair(pay(c,p,pd),nonce(n)))),v).
  (
   (
    recv(v,enc(key(s),sign(key(p),pair(it(d),nonce(n)))),s).
    send(s,pair(bol(T),nonce(n)),c).
    (
```

```
     (
      recv(c,pair(bol(T),nonce(n)),s).
      receive(n).
      send(s,it(d),c).
      send(s,sign(key(s),pair(nonce(n),bol(T))),v)
     )
     +
     (
      recv(c,pair(bol(F),nonce(n)),s).
      send(s,sign(key(s),pair(nonce(n),bol(F))),v)
     )
     +
     timeout.
     send(s,pair(bol(F),nonce(n)),c).
     send(s,sign(key(s),pair(nonce(n),bol(F))),v)
    )
   )
   +
   (
    recv(v,sign(key(p),nonce(n)),s).
    send(s,pair(bol(F),nonce(n)),c).
    send(s,sign(key(s),pair(nonce(n),bol(F))),v)
   )
   +
   (
    recv(c,nonce(n),s).
    send(s,sign(key(s),pair(nonce(n),bol(F))),v).
    send(s,sign(key(s),pair(bol(F),nonce(n))),v)
   )
   +
   (
    timeout.
    send(s,pair(bol(F),nonce(n)),c).
    send(s,sign(key(s),pair(nonce(n),bol(F))),v).
    send(s,sign(key(s),pair(bol(F),nonce(n))),v)
   )
  )
 )
))).Scard(NonceGen(n))
)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Bank
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Bank(U:Knowledge,G:Knowledge)=
(sum(n:Nonce,sum(pc:Player,sum(pv:Player,sum(pd:Price,
 (
  recv(v,sign(key(s),pair(pay(pc,pv,pd),nonce(n))),b).
  (
   (
    send(b,sign(key(b),pair(bol(F),hash(sign(key(s),pair(pay(pc,pv,pd),nonce(n))))),v).
    Bank(U,G)
   )
   <|test(n,G)|>
   (
    send(b,sign(key(b),pair(bol(T),hash(sign(key(s),pair(pay(pc,pv,pd),nonce(n))))),v).
    block(n).
    Bank(add(n,U),G)
```

```
   )
  )
 )
 ))))
)
+
sum(n:Nonce,
 recv(v,sign(key(s),pair(nonce(n),bol(F))),b).
 (
  unblock(n).terminate(n).Bank(rem(n,U),add(n,G))
  <|test(n,U)|>
  terminate(n).Bank(U,add(n,G))
 )
 +
 recv(v,sign(key(s),pair(nonce(n),bol(T))),b).
 (
  transfer(n).terminate(n).Bank(rem(n,U),add(n,G))
  <|test(n,U)|>
  terminate(n).Bank(U,add(n,G))
 )
)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Intruder (and Vendor)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Intruder(X:PM)=
sum(p:Player,sum(m:Message,
 recv(p,m,v).Intruder(union(v,decomp(v,m),X))
 +
 (send(v,m,p).Intruder(X)<|synth(v,m,X)|>delta)
))
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Instantiation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init  encap({send,recv},Customer||Scard(nnc(zero))||
                        Bank(emptset,emptset)||Intruder(emptyM))
```

B.4 Customer Combined with Intruder, Honest Vendor - Safety Properties

In this section only μCRL code for players is presented. The data types are the same as in section B.1.

```
proc

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Intruder (and Customer)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Intruder(X:PM)=
sum(p:Player,sum(m:Message,
    recv(p,m,c).Intruder(union(c,decomp(c,m),X))
    +
    (send(c,m,p).Intruder(X)<|synth(c,m,X)|>delta)
))
+
sum(m:Item,
    got-hold-of(m).Intruder(X)<|in(it(m),X)|>delta)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Smart Card
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Scard(n:Nonce)=delta<|eq(n,nnc(succ(succ(succ(zero)))))|>
(
 sum(d:Item,sum(p:Player,sum(pd:Price,
 (
  recv(c,pair(pay(c,p,pd),hash(it(d))),s).
  initiate(n).
  send(s,pair(hash(it(d)),sign(key(s),pair(pay(c,p,pd),nonce(n)))),c).
  (
   (
    recv(c,enc(key(s),sign(key(p),pair(it(d),nonce(n)))),s).
    send(s,pair(bol(T),nonce(n)),c).
    (
     (
      recv(c,pair(bol(T),nonce(n)),s).
      receive(n).
      send(s,it(d),c).
      send(s,sign(key(s),pair(nonce(n),bol(T))),c)
     )
     +
     (
      recv(c,pair(bol(F),nonce(n)),s).
      send(s,sign(key(s),pair(nonce(n),bol(F))),c)
     )
     +
     timeout.
     send(s,pair(bol(F),nonce(n)),c).
     send(s,sign(key(s),pair(nonce(n),bol(F))),c)
    )
   )
   +
   (
    recv(c,sign(key(p),nonce(n)),s).
    send(s,pair(bol(F),nonce(n)),c).
    send(s,sign(key(s),pair(nonce(n),bol(F))),c)
   )
   +
   (
    recv(c,nonce(n),s).
% The position of the abstract action cancel is changed.
    cancel(n).
    send(s,sign(key(s),pair(nonce(n),bol(F))),c).
    send(s,sign(key(s),pair(bol(F),nonce(n))),c)
   )
   +
   (
    timeout.
    send(s,pair(bol(F),nonce(n)),c).
    send(s,sign(key(s),pair(nonce(n),bol(F))),c).
    send(s,sign(key(s),pair(bol(F),nonce(n))),c)
   )
  )
 )
.(
%Reveal some information about completed session to Intruder.
 send(s,sign(key(s),pair(nonce(n),bol(T))),c).
```

```
 send(s,enc(key(s),sign(key(v),pair(it(d1),nonce(n)))),c).
 send(s,enc(key(s),sign(key(v),pair(it(d2),nonce(n)))),c).
 send(s,sign(key(v),nonce(n)),c).
 send(s,sign(key(b),pair(bol(F),hash(sign(key(s),pair(pay(c,v,pd),nonce(n))))),c).
 send(s,sign(key(b),pair(bol(T),hash(sign(key(s),pair(pay(c,v,pd),nonce(n))))),c)
)
))).Scard(NonceGen(n))
)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Bank
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Bank(U:Knowledge,G:Knowledge)=
(sum(n:Nonce,sum(pc:Player,sum(pv:Player,sum(pd:Price,
 (
  recv(c,sign(key(s),pair(pay(pc,pv,pd),nonce(n))),b).
  (
   (
    send(b,sign(key(b),pair(bol(F),hash(sign(key(s),pair(pay(pc,pv,pd),nonce(n))))),c).
    Bank(U,G)
   )
   <|test(n,G)|>
   (
    send(b,sign(key(b),pair(bol(T),hash(sign(key(s),pair(pay(pc,pv,pd),nonce(n))))),c).
    block(n).
    Bank(add(n,U),G)
   )
  )
 )
 )
))))
+
sum(n:Nonce,
 recv(c,sign(key(s),pair(nonce(n),bol(F))),b).
 (
  unblock(n).terminate(n).Bank(rem(n,U),add(n,G))
  <|test(n,U)|>
  terminate(n).Bank(U,add(n,G))
 )
 +
 recv(c,sign(key(s),pair(nonce(n),bol(T))),b).
 (
  transfer(n).terminate(n).Bank(rem(n,U),add(n,G))
  <|test(n,U)|>
  terminate(n).Bank(U,add(n,G))
 )
)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Vendor
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Vendor=
sum(n:Nonce,sum(pd:Price,sum(d:Item,sum(ps:Player,sum(pc:Player,
(
 recv(c,pair(hash(it(d)),sign(key(ps),pair(pay(pc,v,pd),nonce(n)))),v).
 (
  send(v,sign(key(v),nonce(n)),c)
  +
  (
```

```
   send(v,sign(key(ps),pair(pay(pc,v,pd),nonce(n))),c).
   (
    recv(c,sign(key(b),pair(bol(T),hash(sign(key(ps),pair(pay(pc,v,pd),nonce(n))))),v).
    (
% The position of the abstract action commit is changed.
     commit(n).send(v,enc(key(ps),sign(key(v),pair(it(d),nonce(n)))),c)
     +
     recv(c,sign(key(ps),pair(bol(F),nonce(n))),v)
    )
    +
    recv(c,sign(key(b),pair(bol(F),hash(sign(key(ps),pair(pay(pc,v,pd),nonce(n))))),v).
    (
     send(v,sign(key(v),nonce(n)),c)
     +
     recv(c,sign(key(ps),pair(bol(F),nonce(n))),v)
    )
   )
  )
 )
)
% The following condition is used to reduce the state space.
<|and(eq(pc,c),or(eq(ps,s),eq(ps,c)))|>delta
))))).Vendor
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Instantiation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init  encap({send,recv},Intruder(emptyM)||Scard(nnc(zero))||
                        Bank(emptset,emptset)||Vendor)
```

B.5 Customer Combined with Intruder, Honest Vendor - Liveness Properties

In this section the μCRL code for generating the state space when verifying liveness in the case of malicious Customer is presented. The data types which are not defined here are the same as in section B.1.

```
% The data types Bool, Item, Price, Player, Key, Message and Knowledge
% are the same as in previous codes.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Nonce Data Type
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    Nonce
func    n0,n1 :-> Nonce
map     eq: Nonce # Nonce -> Bool
        NonceGen : Nonce -> Nonce
        if: Bool # Nonce # Nonce -> Nonce
        leq: Nonce # Nonce -> Bool
var     x,y: Nonce
rew     eq(x,x)=T
        eq(n0,n1)=F
        eq(n1,n0)=F
        leq(x,x)=T
        leq(n0,n1)=T
        leq(n1,n0)=F
        NonceGen(n0)=n1
        NonceGen(n1)=n0
        if(T,x,y)=x
```

```
        if(F,x,y)=y
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Data Type for Knowledge of Intruder
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort    PM
func    emptyM:->PM
        insert:Message#PM->PM
map     in:Message#PM->Bool
        synthadd:Player#Message#PM->PM
        add:Message#PM->PM
        addlist:Message#PM->PM
        rem:Message#PM->PM
        union:Player#PM#PM->PM
        if:Bool#PM#PM->PM
        eq:PM#PM->Bool
var     x,y:Message
        ms,ms1,ms2:PM
        U,V:PM
        p: Player
rew     eq(U,U)=T
        eq(U,insert(x,U))=F
        eq(emptyM,insert(x,U))=F
        eq(insert(x,U),emptyM)=F
        eq(insert(y,V),insert(x,U))=and(eq(x,y),eq(U,V))
        if(T,U,V)=U
        if(F,U,V)=V
        in(x,emptyM)=F
        in(x,insert(y,U))=or(eq(x,y),in(x,U))
% Messages are ordered in knowledge set.
        add(x,emptyM)=insert(x,emptyM)
        add(x,insert(y,U))=
          if(leq(x,y),if(eq(x,y),insert(y,U),insert(x,insert(y,U))),
                      insert(y,add(x,U)))
% Sorted list to keep messages which belong to resilient links
        addlist(x,emptyM)=insert(x,emptyM)
        addlist(x,insert(y,U))=
          if(leq(x,y),insert(x,insert(y,U)),insert(y,addlist(x,U)))
        rem(x,emptyM)=emptyM
        rem(x,insert(y,U))=
          if(leq(x,y),if(eq(x,y),U,insert(y,U)),insert(y,rem(x,U)))
        union(p,emptyM,V)=V
        union(p,V,emptyM)=V
        union(p,insert(x,U),V)=synthadd(p,x,union(p,U,V))
% A Message is added to knowledge set only if can not be synthesized.
        synthadd(p,x,U)=if(synth(p,x,U),U,add(x,U))
% A Message is decomposed before being added to knowledge base.
map     decomp:Player#Message->PM
var     p,a,b':Player
        x:Bool
        d:Item
        n:Nonce
        m,m':Message
        k:Key
        pd1: Price
rew     decomp(p,bol(x))=emptyM
        decomp(p,it(d))=insert(it(d),emptyM)
% Nonces are kept explicitly. Intruder should have a fresh nonce, like n1,
```

```
% to consider attacks which require a fresh nonce. If we consider all nonces
% known to the intruder, then it can not stop composing messages when fairness
% constraint are applied. But by having explicit nonces in its
% knowledge, intruder can delete them and then stop cooperation.
        decomp(p,nonce(n))=insert(nonce(n),emptyM)
        decomp(p,pay(a,b',pd1))=emptyM
        decomp(p,pair(m,m'))=union(p,decomp(p,m),decomp(p,m'))
        decomp(p,hash(m))=emptyM
        decomp(p,sign(k,m))=synthadd(p,sign(k,m),decomp(p,m))
        decomp(p,enc(k,m))=if(known(p,k),decomp(p,m),insert(enc(k,m),emptyM))
% Intruder can synthesize new messages from its knowledge set according
% to the following rules.
map     synth:Player#Message#PM->Bool
var     ms:PM
        p,a,b':Player
        x:Bool
        d:Item
        n:Nonce
        m,m':Message
        k:Key
        pd1: Price
rew     synth(p,bol(x),ms)=T
        synth(p,it(d),ms)=or(in(it(d),ms),eq(p,v))
        synth(p,nonce(n),ms)=or(in(nonce(n),ms),eq(n,n1))
        synth(p,pay(a,b',pd1),ms)=T
        synth(p,pair(m,m'),ms)=and(synth(p,m,ms),synth(p,m',ms))
        synth(p,hash(m),ms)=if(or(eq(m,it(d1)),eq(m,it(d2))),
                               T,
                               synth(p,m,ms))
        synth(p,sign(k,m),ms)=or(and(known(p,k),synth(p,m,ms)),in(sign(k,m),ms))
        synth(p,enc(k,m),ms)=or(synth(p,m,ms),in(enc(k,m),ms))
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Declaration of Actions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act     send,recv,com:Player#Message#Player
        timeout
% Abstract actions
        forget
        initiate: Nonce
        block: Nonce
        commit: Nonce
        cancel: Nonce
        receive: Nonce
        transfer: Nonce
        unblock: Nonce
        terminate: Nonce
% Synchronization point
comm    send|recv=com
proc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Intruder
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Intruder(X:PM,Y:PM,Z:PM)=
%learn
sum(p:Player,sum(m:Message,
 recv(p,m,c).(Intruder(addlist(m,X),union(c,decomp(c,m),Y),Z)
                <|in(m,Z)|>
```

```
              Intruder(X,union(c,decomp(c,m),Y),Z))
))
+
%forget
sum(m:Message,
 forget.Intruder(X,rem(m,Y),Z)<|in(m,Y)|>delta
)
+
%resilient network
sum(m:Message,
 (sum(p:Player,send(c,m,p)).Intruder(rem(m,X),Y,Z))<|in(m,X)|>delta
)
+
%compose
sum(m:Message,
 (sum(p:Player,send(c,m,p)).Intruder(rem(m,X),Y,Z))<|synth(c,m,Y)|>delta
)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Smart Card
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Scard(n:Nonce)=delta<|eq(n,n1)|>
(
 sum(d:Item,sum(p:Player,sum(pd:Price,
 (
  recv(c,pair(pay(c,p,pd),hash(it(d))),s).
  initiate(n).
  send(s,pair(hash(it(d)),sign(key(s),pair(pay(c,p,pd),nonce(n)))),c).
  (
   (
    recv(c,enc(key(s),sign(key(p),pair(it(d),nonce(n)))),s).
    send(s,pair(bol(T),nonce(n)),c).
    (
     (
      recv(c,pair(bol(T),nonce(n)),s).
      send(s,it(d),c).
      send(s,sign(key(s),pair(nonce(n),bol(T))),c)
     )
     +
     (
      recv(c,pair(bol(F),nonce(n)),s).
      send(s,sign(key(s),pair(nonce(n),bol(F))),c)
     )
     +
     timeout.
     send(s,pair(bol(F),nonce(n)),c).
     send(s,sign(key(s),pair(nonce(n),bol(F))),c)
    )
   )
   +
   (
    recv(c,sign(key(p),nonce(n)),s).
    send(s,pair(bol(F),nonce(n)),c).
    send(s,sign(key(s),pair(nonce(n),bol(F))),c)
   )
   +
   (
    recv(c,nonce(n),s).
```

```
    cancel(n).
    send(s,sign(key(s),pair(nonce(n),bol(F))),c).
    send(s,sign(key(s),pair(bol(F),nonce(n))),c)
   )
   +
   (
    timeout.
    send(s,pair(bol(F),nonce(n)),c).
    send(s,sign(key(s),pair(nonce(n),bol(F))),c).
    send(s,sign(key(s),pair(bol(F),nonce(n))),c)
   )
  )
 )
))).Scard(NonceGen(n))
)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Bank
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Bank(U:Knowledge,G:Knowledge)=
(
(sum(n:Nonce,sum(pc:Player,sum(pv:Player,sum(pd:Price,
 (
  recv(c,sign(key(s),pair(pay(pc,pv,pd),nonce(n))),b).
  (
   (
    send(b,sign(key(b),pair(bol(F),hash(sign(key(s),pair(pay(pc,pv,pd),nonce(n))))),c).
    Bank(U,G)
   )
   <|test(n,G)|>
   (
    send(b,sign(key(b),pair(bol(T),hash(sign(key(s),pair(pay(pc,pv,pd),nonce(n))))),c).
    block(n).
    Bank(add(n,U),G)
   )
  )
 )
 )
))))
+
sum(n:Nonce,
 recv(c,sign(key(s),pair(nonce(n),bol(F))),b).
 (
  unblock(n).terminate(n).Bank(rem(n,U),add(n,G))
  <|test(n,U)|>
  terminate(n).Bank(U,add(n,G))
 )
 +
 recv(c,sign(key(s),pair(nonce(n),bol(T))),b).
 (
  transfer(n).terminate(n).Bank(rem(n,U),add(n,G))
  <|test(n,U)|>
  terminate(n).Bank(U,add(n,G))
 )
)
)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Vendor
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Vendor=
sum(n:Nonce,sum(pd:Price,sum(d:Item,sum(ps:Player,sum(pc:Player,
(
 recv(c,pair(hash(it(d)),sign(key(ps),pair(pay(pc,v,pd),nonce(n)))),v).
 (
  send(v,sign(key(v),nonce(n)),c)
  +
  (
   send(v,sign(key(ps),pair(pay(pc,v,pd),nonce(n))),c).
   (
    recv(c,sign(key(b),pair(bol(T),hash(sign(key(ps),pair(pay(pc,v,pd),nonce(n)))))),v).
    (
% The position of the abstract action commit is changed.
     commit(n).send(v,enc(key(ps),sign(key(v),pair(it(d),nonce(n)))),c)
     +
     recv(c,sign(key(ps),pair(bol(F),nonce(n))),v)
    )
    +
    recv(c,sign(key(b),pair(bol(F),hash(sign(key(ps),pair(pay(pc,v,pd),nonce(n)))))),v).
    (
     send(v,sign(key(v),nonce(n)),c)
     +
     recv(c,sign(key(ps),pair(bol(F),nonce(n))),v)
    )
   )
  )
 )
)
% The following condition is used to reduce the state space.
<|and(eq(pc,c),or(eq(ps,s),eq(ps,c)))|>delta
))))).Vendor

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Instantiation
% Set of messages over resilient links is defined explicitly here.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init  hide({com},encap({send,recv},
Intruder(emptyM,emptyM,insert(sign(key(s),pair(nonce(n0),bol(F))),
                              insert(sign(key(s),pair(nonce(n0),bol(T))),emptyM)))||
                              Scard(n0)||Bank(emptset,emptset)||Vendor))
```