A roadmap to electronic payment transaction guarantees and a Colored Petri Net model checking approach

Panagiotis Katsaros

Department of Informatics

Aristotle University of Thessaloniki

54124 Thessaloniki, Greece

tel.: +30-2310-998532, fax: +30-2310-998419

katsaros@csd.auth.gr

Abstract

Electronic payment systems play a vital role in modern business-to-consumer and business-to-business e-commerce. Atomicity, fault tolerance and security concerns form a problem domain of interdependent issues that are taken into account to assure the transaction guarantees of interest. We focus on the most notable payment transaction guarantees: *money conservation*, *no double spending*, *goods atomicity*, *distributed payment atomicity*, *certified delivery* or *validated receipt* and the high-level guarantees of *fairness* and *protection of payment participants' interests*. Apart from a roadmap to the forenamed transaction guarantees, this work's contribution is basically a full-fledged methodology for building and validating high-level protocol models and for proving payment transaction guarantees by model checking them from different participants perspectives (*payer perspective*, as well as *payee perspective*). Our approach lies on the use of Colored Petri Nets and the CPN Tools environment (i) for editing and analyzing protocol models, (ii) for proving the required transaction guarantees by CTL-based (Computation Tree Temporal Logic) model checking and (iii) for evaluating the need of candidate security requirements.

KEYWORDS: electronic payments, atomicity, fault tolerance, e-commerce transactions, security, Colored Petri Nets, model checking

1. Introduction

Electronic payment systems are expected to ensure that payment transactions occur *atomically*. This means that each participating node must reach the same conclusion as to whether an ongoing payment is to be completed, even in the face of failures. Atomicity is one of the key properties (Atomicity, Consistency, Isolation and Durability) – known as ACID properties – of modern transactional information systems [20]. In these systems the mechanism used for achieving atomic commitment (e.g. the two-phase commit protocol) is

bundled together with a specific program-to-program communication protocol and that protocol lives on top of an appropriate infrastructure. In electronic payments, participants may use communication protocols for which there are no transactional variants (e.g. HTTP) and the programs may be deployed in very heterogeneous application environments. For these reasons, electronic payment systems cannot rely on traditional transaction mechanisms.

Another problem is that in addition to potential system crashes and accompanying message omission failures, we have to take into account the possibility of fraudulent behavior by the payment participants, as well as, the well-known security flaws of the Internet infrastructure. A payment protocol must provide an appropriate combination of transaction guarantees that depends on the application domain. Thus, we need means for proving the expected transaction guarantees and for studying the protection requirements against potential security flaws and intrusion attacks ([6]).

We focus on payment transaction guarantees like *money conservation*, *no double spending*, *goods atomicity*, *distributed payment atomicity* and *certified delivery* or *validated receipt*. Security concerns ([1] and [36]) are skimmed only to the degree needed to enable safe payments, in the presence of various transaction attack scenarios or potentially fraudulent behavior. Also, we refer to the high-level transaction guarantees of *fairness* ([2]) and *protection of participants' interests* ([47]).

The proposed model checking approach verifies the forenamed transaction guarantees from different participants' perspectives that are selected based on the adopted *trust model*. We suggest the construction and validation of a Colored Petri Net (CP-net) that reflects all protocol execution scenarios, including unilateral transaction aborts, potentially fraudulent behavior and all site failure and message loss possibilities. Valuable features of the CP-net modeling language that play an important role in our model checking approach are: (i) the

fact that the formalism builds upon true concurrency instead of an interleaving-based semantics, (ii) the fact that CP-nets provide a compact description of control, synchronization and data manipulation resulting in an explicit representation of both model states and events and (iii) the wide range of analysis alternatives, which allow to conveniently express and subsequently check the required model correctness criteria and the expected payment transaction guarantees.

The model is built in CPN Tools ([13]), an advanced toolset for editing, simulating and analyzing CP-nets ([22]). The expected guarantees are verified by CTL-based (Computation tree Temporal Logic) model checking. Our approach is described in terms of a CP-net developed for NetBill, a system for Internet-based micropayments for information goods and services.

Section 2 provides an overview of electronic payments and defines the transaction guarantees of interest. Section 3 describes the proposed model building and validation approach. Section 4 refers to the CTL-based model checking of the expected transaction guarantees in terms of the developed NetBill CP-net. Section 5 outlines related model checking works and other CP-net solutions to specific e-commerce problems. We conclude with a discussion on the usefulness of the proposed approach and its potential impact.

2. Electronic payments and payment transaction guarantees

2.1 Electronic payment models

The growing importance of e-commerce and the ever-increasing number of business transaction models has resulted in a plethora of payment systems. *Online payments* involve communication with a *trusted third party* (TTP) during payment and in general they are considered as more secure than *offline payments* that involve only the payer and the payee.

The vast majority of Internet payment systems are online systems that perform either:

- Credit-card payments (First Virtual, CyberCash, iKP, Anonymous Credit Cards)
- Micropayments (NetBill, Millicent, μ-iKP, MiniPay and NetCash)
- Or they are used as payment switches (OpenMarket).

Offline payment systems include

- The *electronic purses that use smart cards* (Danmont/Visa, CLIP, Mondex and EMV Electronic Purse)
- The *electronic checks* (FSTC Electronic Checks)
- A number of electronic cash systems (eCash and CAFE).

A detailed description of the forenamed types of payment systems is given in [14], [17] and [37]. In [1], the authors provide a thorough treatment of the most fundamental security requirements, as well as a complete source of references. In what is concerned with credit-card payments, iKP has been designed by IBM Research with the intention of serving as a starting point for new standards. A commercially successful standard that was based on iKP is the Secure Electronic Transactions (SET) specification, launched by Mastercard and VISA as an open non-proprietary, license-free standard for securing on-line transactions. In the field of micropayments, which mainly concern with payments of intangible products (digital goods or services), NetBill is probably the most widely known commercially successful payment system in use. Regarding the aforementioned offline payment options, we know that Mondex, FSTC Electronic Checks and eCash have been adopted by large financial organizations and banks as an alternative way of payments to be offered to their customers. An important development in the last few years is the widespread use of one-stop integrated payment processing services like the ones offered by PayPal, Amazon Payments and Google Checkout. The payment protection policy of the forenamed service providers applies only to tangible goods transactions (e.g. books, DVDs etc) and the arbitration process for the resolution of disputes is based on proofs of delivery that are

provided by the seller. In effect, the declared user agreements do not offer guarantees for the safety and the reliability of the application(s) used to access the payment service.

A number of recent contributions ([38], [30], [43], [29]) confirm an ongoing interest in the development of new payment systems. Also, the work published in [41] points out the need for custom-made payment systems, which provide payment services that are extended beyond the traditional bilateral transaction model.

The authors of [21] and [36] are probably the first who pinpoint the need for a systematic treatment of the correctness properties required in digital payment systems. The first work focuses on model checking three transaction guarantees from the ones mentioned in Section 1 and the second work proposes a framework of abstractions for the formal definition of security properties, like for example payment integrity and privacy.

In general, the majority of the published articles, as well as a relevant book on digital payment systems ([32]) and a well-known research project in e-commerce ([27]) focus on the security requirements of electronic payments. Our work refers to model checking the transaction guarantees mentioned in Section 1, in all protocol execution scenarios, including unilateral transaction aborts, potentially fraudulent behavior, all site failure and message loss possibilities and various protocol level transaction attacks.

2.2 Payment transaction guarantees

The *money conservation* guarantee ([21]) - also called *money atomicity* - is the basic level of atomicity in electronic payments. This guarantee ensures that there is no possibility of creation or destruction of money, while electronic money is being transferred. In a poorly designed payment system, money conservation can be compromised due to site failures, unilateral transaction aborts, fraudulent behavior and different forms of protocol-level attacks. More specifically, in account transfer systems we do not allow non-atomic

execution of pairs of debit - credit actions and also we do not allow redundant debits and credits within the same payment transaction.

We also require payment systems that *prevent double-spending* ([32]), that is, they prevent execution scenarios where a single payment order is performed more than once. In a *replay attack*, double spending takes place by replaying some messages from a previous legitimate run. A common mechanism to prevent this attack is to guarantee the *freshness* of messages exchanged between the participants. Freshness means that a message provably belongs to the current payment transaction and is not a replay of a previous message.

Goods atomicity ([21]) is a transaction guarantee that ensures money atomicity and also ensures that there is no possibility of paying without receiving goods or vice versa. In bilateral payment transactions, goods atomicity is checked from both participants' perspectives (payer and payee) in all cases of site failures, unilateral transaction aborts and potentially fraudulent behavior.

In the more general case of multi-party payments, which is the case of *distributed purchase transactions*, goods atomicity is required for all protocol participants ([25]). In a distributed purchase transaction a consumer interacts with multiple merchants. Consider for example a consumer who pays for an airline fare, if and only if, the accompanying accommodation payment transaction asked from another merchant is also successfully completed.

Distributed payment atomicity ([41]) guarantees the inclusion of interactions with independent participants into a single transaction. A way to provide this guarantee in heterogeneous environments, where applications use communication protocols with no transactional variants, is the Transaction Internet Protocol – TIP ([28]). TIP's two phase commit coordinates a system's transaction managers independently of the used application

communication protocol. TIP operates over TCP and optionally uses the Transport Layer Security protocol ([15]) to authenticate the senders and to encrypt the TIP commands.

However, TIP is amenable to different forms of *intrusion attacks* (two denial of service attacks, one transaction corruption attack, one packet-sniffing attack and one man-in-the-middle attack) and for this reason customized payment systems that use TIP have to be formally analyzed ([26]), in order to evaluate the need of candidate security characteristics.

Certified delivery ([21]) is a transaction guarantee that requires both money conservation and goods atomicity and also requires all payment participants to be able to prove the sensitive details of the transaction. In NetBill, certified delivery allows consumers and merchants to prove what happened on-line and to settle disputes off-line. This guarantee also has to be provided in all cases of site failures and message losses, unilateral transaction aborts and potentially fraudulent behavior. In bilateral payment transactions, certified delivery is checked from both participants' perspectives (payer and payee). Alternatively, a class of payment systems (e.g. [38]) used in digital goods transactions provides the *validated receipt* guarantee, which ensures that the payer is able to verify the contents of the product about to be received. This is achieved by enabling the payer to verify that the encrypted goods sent by the merchant are given as encryption of the same product, for which an escrowed copy has been previously encrypted by the TTP and has been subsequently placed at a publicly accessible place.

Goods atomicity satisfies the high-level *fair exchange* property ([2] and [18]): no protocol participant can gain any advantage over other participants by misbehaving. In payment transactions the payer gains advantage if he receives the goods (or payment receipt), but the payee does not receive the payment. On the other hand, a payee gains advantage if he receives the payment, but the payer does not receive the goods (or payment

receipt). We note that fairness guarantees only that money is exchanged for something and not necessarily for what the payer pays for.

Protection of participants' interests ([47]) is another high-level guarantee that ensures that participants get exactly what they are legitimate to get. This guarantee generalizes fairness in the sense that in some cases one participant's interests can be compromised even if no one else has gained any advantage.

To analyze a protocol with respect to fairness, we consider protocol execution scenarios where a participant misbehaves and we check whether the participant himself can obtain any advantage. When analyzing a protocol with respect to protection of a participant's interests, we consider protocol execution scenarios where everything (other participants and the network) except the participant's local execution can go wrong and we check whether the participant's interests can be hurt. Also, we take into account threats that are not considered when we analyze the protocol's fairness. As an example, the assumed failure model includes network failures, which are out of the control of the participants: in an electronic cash payment, if the payment sent by the payer gets lost while in transit and the "payee" does not send the goods, then no one has gained any advantage, but the customer's interests have been hurt. Also, we take into account the possibility of collusions among multiple participants, as well as a class of attacks known as *sabotage attacks*, where a participant misbehaves not aiming to obtain advantage, but to hurt someone else's interests.

When the studied payment system uses trusted parties, all the forenamed guarantees rely on these parties behaving as trusted. This means that trusted parties are assumed to perform protocol steps correctly and reliably and this behavior is decisive to guaranteeing satisfaction of the expected transaction guarantees. Thus, our high-level protocol models are required to satisfy a set of protocol-specific assumptions that we call *trust assumptions*, which are supposed to be part of the model's correctness criteria.

3. A Colored Petri Net modeling approach for payment systems

3.1 The Colored Petri Net modeling language

Apart from the proposed CP-net analysis, two alternatives have been used in model checking payment transaction guarantees. The work of [19] uses the SPIN model checker and the one reported in [21] employs a Communicating Sequential Processes approach and the Failure Divergence Refinement (FDR) tool ([39]). Both of them adopt a process-based representation of the system, where processes are described using events and operators. Events cause a process to change state, but *the representation of states is implicit*.

Significant restrictions of the forenamed approaches that motivate the alternative approach proposed by us are:

- their limited expressiveness in the modeling of concurrently executed events and
- their limited expressiveness in specifying the correctness properties of interest

The interleaving semantics of PROMELA (SPIN's specification language) and the trace semantics of the CSP/FDR approach imply that concurrent execution of two events can only be represented by the occurrence of the two events after each other, *in any order*. From our experience with PROMELA ([5] and [6]), we realized that this fact thrusts the modeler to express concurrency within a process by explicitly specifying all possible orders of occurrence for the concurrent events. However, this approach is inadequate for models with payment participants that have concurrent (and possibly synchronized) threads of control.

This problem arises when we study the guarantees of interest in scenarios with concurrent payment transactions. The absence of double spending guarantee in replay attack scenarios is a typical case, where the TTP model is required to handle multiple payment transactions with concurrent threads that obey to the modeled protocol rules. In

SPIN, the analyst has to explicitly specify all possible orders of occurrence for the concurrent events within the TTP. Depending on the protocol size, the number of possible event orders can be large and this results in an error-prone analysis. The CP-net formalism is an attractive alternative, since concurrency is not expressed in an interleaving-based semantics. The replay attack scenario and the model checking of the absence of double spending are illustrated in more detail in section 4.5.

Regarding the second mentioned restriction of the published model checking approaches, we note that in SPIN correctness properties are specified in Linear Temporal Logic (LTL). In principle, LTL is poorly suited for reachability properties like the ones implied by the transaction guarantees of interest. It implicitly quantifies over all possible execution paths and therefore it can only express reachability negatively: something is not reachable. Moreover, with LTL there is still no way to choose an arbitrary set of starting states (nested reachability), for the model checking of the property of interest [7]. Figure 1 shows a typical case of two state space graphs, for which we cannot have an LTL formula that is true for some tree and false for the other tree.

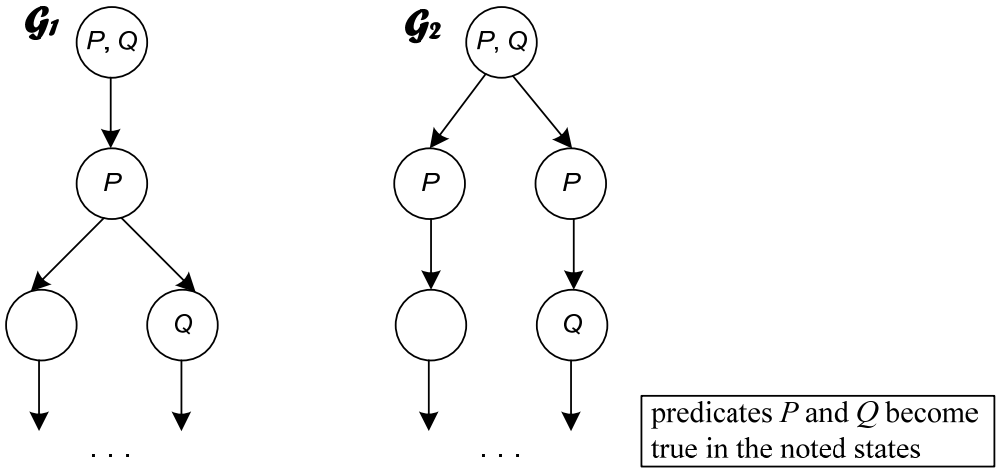

Figure 1. Two state space graphs, indistinguishable for Linear Temporal Logic (LTL)

In CSP/FDR the protocol and the property of interest are described as two different CSP processes. To determine whether the protocol satisfies the property the modeler tests whether the protocol's set of traces is a subset of the property's set of traces (trace refinement). In [21] the authors confess that in general the most obvious specification of a property is often incorrect or inadequately expressed. Indeed, a more precise property specification is usually obtained as a result of some experimentation.

In contrast to SPIN and CSP/FDR, Petri Net (PT-net) modeling languages provide an *explicit representation of both states and events* and an easy to understand and intuitively appealing graphical representation. They have well-defined formal semantics that instead of interleaving builds upon *true concurrency* and they also offer a wide range of formal analysis alternatives.

CP-nets ([22], [23]) constitute a compact and much more convenient modeling language when compared to ordinary PT-nets, in a similar way as high-level programming languages are more adequate for practical programming than assembly code. In CP-nets we attach a data value to each token and this results in much fewer places than would be needed in a low-level PT-net. Thus, the tokens of a CP-net are distinguishable from each other and hence *colored*. An important reason for using CP-nets is that they provide a *compact description of control and synchronization, integrated with the description of data manipulation*. This means that on a single workspace it can be seen what the environment, enabling conditions and effects of a state transition are. CP-nets also provide support for building large models, by relating smaller CP-nets to each other in a well-defined way. This results in *hierarchical descriptions* and makes it possible to model very large systems in a manageable and modular way.

CP-nets have been developed over the last 28 years and today constitute a mature modeling language supported by an advanced toolset ([13]) for editing, interactively

simulating and formally analyzing the model by a wide range of analysis alternatives. Reachability properties are specified in CTL by taking advantage of the offered explicit representation of the system's states. An informal introduction to the CP-net modeling language is provided in Appendix A.

3.2 The NetBill payment system

The NetBill transaction protocol ([12]) involves three participants: the consumer (*C*), the merchant (*M*) and the trusted third party (TTP). Transactions involve three phases: price negotiation, goods delivery and payment. We consider the selling of information goods or services, in which case NetBill links goods delivery and payment into a single atomic transaction. We use the notation "$X \Rightarrow Y$ *message*" to indicate that *X* sends the specified *message* to *Y*. The basic protocol consists of the following messages:

1. $C \Rightarrow M$ Price request
2. $M \Rightarrow C$ Price quote
3. $C \Rightarrow M$ Goods request
4. $M \Rightarrow C$ Requested goods, encrypted with a key *K*
5. $C \Rightarrow M$ Electronic Payment Order (*epo*)
6. $M \Rightarrow TTP$ Endorsed Electronic Payment Order (including the key *K*)
7. $TTP \Rightarrow M$ Transaction result (including *K* in a successful payment)
8. $M \Rightarrow C$ Transaction result (including *K* in a successful payment)

C and *M* interact with each other in the following way:

- *C* issues a price request for a particular product (1) and *M* replies with the requested price (2),
- *C* either aborts the transaction or issues a goods request to *M* (3),
- in the second case, *M* delivers the requested goods encrypted with a key *K* (4).

The goods are cryptographically checksummed in order to be able to confirm that received goods are not affected by potential transmission errors and that they have not been subsequently altered. The *TTP* is not involved until the payment phase:

- *C* sends to *M* (5) an electronic payment order (*epo*) including all necessary payment details and the received product checksum,
- *M* validates the received epo and checksum information and either aborts the transaction or endorses it by sending to the *TTP* the received payment order, together with *additional payment information* and the decryption key *K* (6),
- *TTP* responds to *M* (7) with the payment result and the decryption key *K* (successful payment), which are finally forwarded to *C* (8) to terminate the transaction.

NetBill protects *C* against fraud by *M*, in the following ways:

- the key *K*, which is needed to decrypt the goods is registered with the *TTP* and if *M* does not respond in a valid payment as expected, *C* asks the key from the *TTP*,
- if there is a discrepancy between what *C* ordered and what *M* delivered, *C* can easily demonstrate this discrepancy to the *TTP*, since the payment order received by *TTP* includes all details about what exactly was ordered, the amount charged, the key *K* sent by *M* and the checksum of the delivered encrypted goods. Thus, if the goods are faulty it is easy to demonstrate that the problem lies with the goods as sent and not with any subsequent alteration (that would produce different checksum information).

3.3 General model structure and assumptions

We propose the places of a CP-net payment model to belong to the following categories:

- places that represent *participants' states with respect to the ongoing purchase transaction* (e.g. `IDLE`, `WAIT`, `ABORTED`, `COMMITTED`, `FAILED` etc),

- places that represent participants' communication channels, like for example the channel used for the messages sent by the Consumer to the Merchant,
- places that represent sensitive information, like for example money or purchased goods (or payment receipt), which take one of usually two possible values depending on the ongoing *protocol execution scenario* (e.g. valid or invalid goods, enough or not enough account balance etc) and
- places used to represent transaction control flow, like for example places that trigger a query, due to an occurred transaction timeout.

```
colset E          = with e;
colset INT        = int;
colset BOOL       = bool;
colset STRING     = string;
colset validORnValid = with v | i;
colset accBalance= with gValue | lessMoney;
colset State      =  with IDLE   | WAIT         | W_FAILED
                       | ABORTED   | COMMITTED    | C_FAILED
                       | COMPLETED | DISPUTED_TR  | LISTEN       | NO_RECORD
                       | L_FAILED  | STARTED_TR   | ST_FAILED    | N_FAILED;
colset NetBillMSg=union gRequest:validORnValid  + eGoods:validORnValid
                        +pORequest:validORnValid + trResult:STRING
                        +dKey:validORnValid      + query:E;
colset NetBillMQ  =list NetBillMSg;
var p,q,r,s: NetBillMQ;
var mes,mes2: NetBillMSg;
var gReq:validORnValid;
var pOrder: validORnValid;
var enGoods: validORnValid;
var balance: accBalance;
var timer,timer2: BOOL;
var key: validORnValid;
var st: State;
```

Figure 2. Color sets and variables for the NetBill CP-net

Figure 2 introduces the color sets and variables used in the NetBill CP-net. The token values included in the enumerated color set `validORnValid` are used in symbolically representing sensitive information (goods request, payment order, encrypted goods and encryption key) that determines the ongoing protocol execution scenario. The token values included in the enumerated color set `accBalance` represent different cases of account balance, when compared to the ordered goods value. Color set `State` includes the token

values needed to represent all possible participants' states with respect to the ongoing purchase transaction. Initial participants' states are `IDLE` for *C*, `LISTEN` for *M* and `NO_RECORD` for the *TTP*. The union color set `NetBillMSg` specifies all possible types of messages exchanged through the model's communication channels. Each channel is represented as a list of `NetBillMSg` messages (`NetBillMQ`).

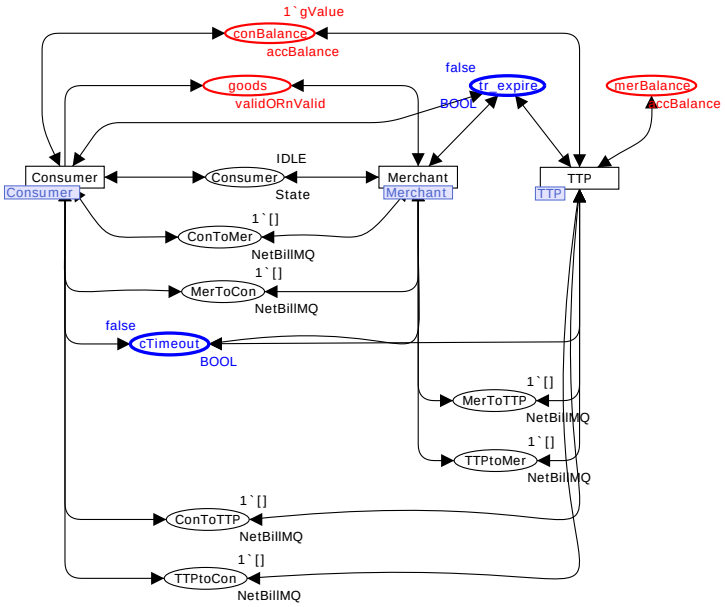

Figure 3. Top level CP-net for the NetBill payment system

Figure 3 presents the top level CP-net that includes places of all of the four forenamed categories. Place `Consumer` is the only one place of the first category shown in this page. Places `ConToMer`, `MerToCon`, `MerToTTP`, `TTPtoMer`, `ConToTTP` and `TTPtoCon` represent all participants' communication channels (initially empty). Places `conBalance`, `merBalance` and `goods` contain tokens that represent money and purchased goods respectively. Finally, places `cTimeout` and `tr_expire` are used to express the triggering of a query sent by *C* to the *TTP*, due to an occurred transaction timeout. Substitution transitions `Consumer`, `Merchant` and `TTP` include the corresponding participants' state transitions to be described in the forthcoming paragraphs.

In the shown NetBill CP-net, the adopted modeling assumptions are:

- Non-reliable FIFO message delivery by the participants' communication channels, with no eavesdropping and no message integrity violation (these possibilities may compromise privacy and payment integrity [6], but they are out of the scope of our concern, since we focus on the transaction guarantees of section 2.2[1]).
- Participants' sites fail by crashing, without emission of spurious messages (*fail-stop failure model*).
- While in a failed state, all protocol messages and data in participants' input communication channels are lost (*omission failures*).
- Message losses due to communication failures are modeled also as transitions to failure states for the recipients and this representation is consistent with the effects of a message loss to the ongoing purchase transaction.

The model's trust assumptions do not allow dishonest or unexpected behavior for the *TTP*. This means that irrespective of the occurred site failures and message losses the *TTP* either aborts or completes the transaction and delivers the transaction result that in all cases should be consistent with the occurrence or no occurrence of the requested payment.

3.4 Payment participants' state transitions

This section provides finite state automata that specify the participants' state transitions to be modeled. Consumer's finite state automaton (Figure 4 and Table 1) represents all protocol execution scenarios, including unilateral transaction aborts, merchant fraud, dishonest consumer behavior (low account balance) and all site failure and message loss cases. State transitions reflect the effects of the exchanged protocol messages on the state of the ongoing purchase transaction. Each transaction starts with the dispatch of a (valid or

[1] However, as we already noted, fake protocol messages in distributed purchase transactions are likely to break the assumed payment atomicity.

invalid) goods request (transition C1) and every goods request corresponds to the launch of a new purchase transaction.

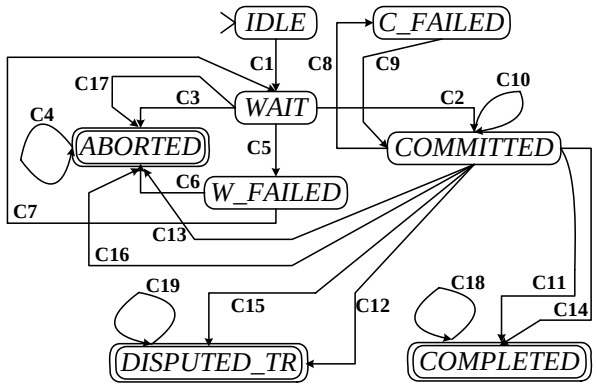

Figure 4. Consumer's finite state automaton

Table 1 Consumer's automaton transitions

Transitions from operational states:	
C1	Consumer sends to the Merchant a (valid or invalid) goods request.
C2	Consumer receives from the Merchant the requested encrypted goods and sends him an electronic payment order. Failure to perform these two actions atomically corresponds to executing transition C5.
C3	Consumer aborts the ongoing purchase transaction.
C4	Consumer receives a message sent by the Merchant or the TTP, while being in state `ABORTED`.
C5	This models the occurrence of a consumer site failure, while the consumer was in state `WAIT`. Protocol messages or data that lie in consumer's input communication channels are lost.
C8	This models the occurrence of a consumer site failure, while the consumer was in state `COMMITTED`. Protocol messages or data that lie in consumer's input communication channels are lost. The consumer remains committed to the already paid purchase transaction by means of permanent storage.
C10	Ongoing purchase transaction timeouts due to a merchant site failure or a fraudulent merchant abort. The consumer queries the TTP for the transaction result.
C11	Consumer receives a "Succeed" transaction result and the required decryption key by the merchant.
C12	Consumer receives a "Succeed" transaction result and a decryption key by the merchant, but discovers a merchant fraud.
C13	Consumer receives an "Aborted" transaction result by the merchant.
C14	Consumer receives a "Succeed" transaction result and the required decryption key by the TTP.
C15	Consumer receives a "Succeed" transaction result and a decryption key by the TTP, but discovers a merchant fraud.
C16	Consumer receives an "Aborted" transaction result by the TTP.
C17	Consumer's sent request timeouts, due to a merchant site failure or due to merchant's unilateral abort.
C18	Consumer receives a "Succeed" transaction result, while being in state `COMPLETED`.
C19	Consumer receives a "Succeed" transaction result, while being in state `DISPUTED_TR`.
Transitions from failure states:	
C6	Consumer's site recovers from a failure and the ongoing purchase transaction is aborted. Protocol messages or data that lie in consumer's input communication channels are lost (the consumer cannot receive messages while being in state `W_FAILED`).
C7	Consumer's site recovers from a failure and continues with the ongoing purchase transaction. Protocol messages or data that lie in consumer's input communication channels are lost.
C9	Consumer's site recovers from a failure and queries the TTP for the result of the ongoing purchase transaction. Protocol messages or data that lie in consumer's input communication channels are lost.

Site failures (including message loss cases) are not represented by terminating states: irrespective of the occurred site failures a consumer either aborts or completes a purchase transaction and the received goods are either the ordered ones or are not the ones expected. As a consequence, the shown finite state automaton includes three terminating states (`COMPLETED`, `ABORTED` and `DISPUTED_TR`) and two failure states (`W_FAILED` and `C_FAILED`) that correspond to two different recovery cases. While the consumer is in a failed state, all protocol messages and data received in its input communication channels are lost.

Merchant's finite state automaton (Figure 5 and Table 2) reflects all merchant behavior possibilities, including unilateral transaction aborts, merchant fraud and all site failures and message losses. We abstract from candidate recovery mechanisms (which result in loss of generality) by assuming that merchant's site does not provide recovery with respect to the ongoing transaction. As a consequence, the shown finite state automaton includes four terminating states (`COMPLETED`, `ABORTED`, `L_FAILED` and `ST_FAILED`) with two of them corresponding to site failure (and message loss) states. In all failed states, protocol messages and data received in merchant's input communication channels are lost.

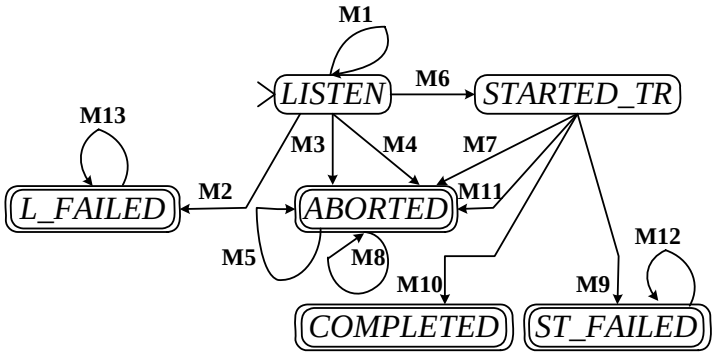

Figure 5. Merchant's finite state automaton

Table 2 Merchant's automaton transitions

Transitions from operational states:	
M1	The merchant thread receives a valid goods request and responds with an encrypted version of the requested goods. Failure to perform these two actions atomically corresponds to executing transition M2.
M2	This models the occurrence of a merchant site failure, while the merchant thread was in the `LISTEN` state. Protocol messages or data that lie in merchant's input communication channels are lost. We do not make assumptions regarding the merchant site recovery.
M3	The merchant thread receives an invalid goods request (e.g. wrong product) or an invalid payment order (e.g. invalid product checksum number) and aborts the ongoing purchase transaction.
M4	The merchant thread aborts the ongoing purchase transaction due to unilateral decision.
M5	The merchant thread receives a goods request or a payment order, while being in state `ABORTED`.
M6	The merchant thread endorses a valid electronic payment order and forwards it (including the required decryption key) to the TTP. Potential failure to perform these two actions atomically corresponds to executing transition M2 (site failure) or transition M4 (unilateral abort).
M7	The merchant thread aborts the ongoing purchase transaction.
M8	The merchant thread receives the transaction result and does not notify the consumer.
M9	This models the occurrence of a merchant site failure, while the merchant thread was in state `STARTED_TR`. Protocol messages or data that lie in merchant's input communication channels are lost. We do not make assumptions regarding the merchant's site recovery.
M10	The merchant thread receives a "Succeed" transaction result from the TTP and forwards it together with the required decryption key to the consumer. Failure to perform these two actions atomically corresponds to executing transition M9.
M11	The merchant thread receives an "Aborted" transaction result from the TTP and notifies the consumer. Failure to perform these two actions atomically corresponds to executing transition M9.
M12	Protocol messages sent to the merchant's input communication channels are lost.
M13	Protocol messages sent to the merchant's input communication channels are lost.

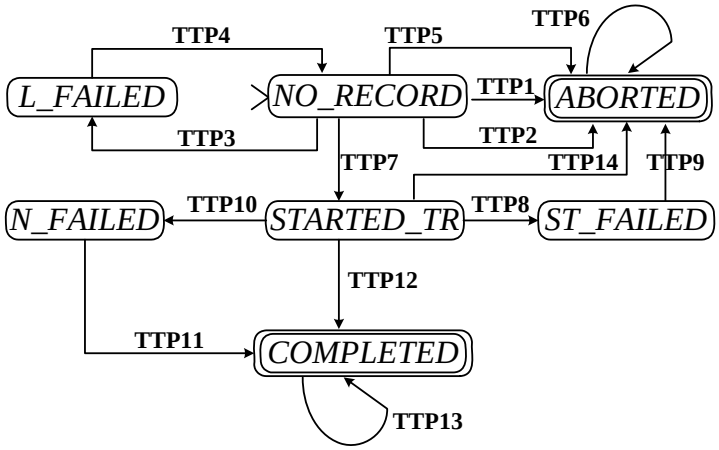

Figure 6. *TTP* finite state automaton

The *TTP* finite state automaton (Figure 6 and Table 3) reflects all protocol execution scenarios (valid or invalid payment order, low account balance, etc), as well as unilateral transaction aborts (debit or credit failures) and all site failure possibilities (including message loss cases). The adopted trust assumptions imply that irrespective of the occurred

site failures or message losses the *TTP* either aborts or completes the purchase transaction and delivers the transaction result as expected. As a consequence, the shown finite state automaton includes two terminating states (`COMPLETED` and `ABORTED`) and three failure states (`L_FAILED`, `ST_FAILED` and `N_FAILED`) that correspond to three different recovery cases. While the *TTP* is in a failed state, all protocol messages and data received in its input communication channels are lost.

Table 3 TTP automaton transitions

Transitions from operational states:	
TTP1	The TTP receives an invalid payment order (e.g. invalid merchant account) and notifies the merchant for the transaction abort. Failure to perform these two actions atomically corresponds to executing transition TTP3
TTP2	The TTP receives a valid payment order, but fails to debit consumer's account and notifies the merchant for the transaction abort. Failure to perform these actions atomically corresponds to executing transition TTP3.
TTP3	This models the occurrence of a TTP site failure, while the TTP was in the `NO_RECORD` state. Protocol messages or data that lie in TTP input communication channels are lost.
TTP5	TTP receives a consumer's query for the ongoing purchase transaction and responds with a "No Record" message. Failure to perform these two actions atomically corresponds to executing transition TTP3.
TTP6	The TTP receives a consumer's query for the ongoing purchase transaction and responds with an "Aborted" message.
TTP7	The TTP receives a valid payment order and debits consumer's account. Failure to perform these two actions atomically corresponds to executing transition TTP3.
TTP8	This models the occurrence of a TTP site failure, while the TTP was in the `STARTED_TR` state. Protocol messages or data that lie in TTP input communication channels are lost.
TTP10	The TTP credits merchant's account, but fails to deliver the transaction result due to a site failure. Protocol messages or data that lie in TTP input communication channels are lost.
TTP12	The TTP credits merchant's account and delivers to the merchant the transaction result. Failure to perform these two actions atomically corresponds to executing transition TTP10.
TTP13	The TTP receives a consumer's query for the ongoing purchase transaction and responds with a "Succeed" result notification accompanied by the required decryption key.
TTP14	The TTP fails to credit merchant's account, returns debited amount to the consumer's account and sends to the merchant an "Aborted" transaction result. Failure to perform these actions atomically corresponds to executing transition TTP8.
Transitions from failure states:	
TTP4	The TTP recovers from a site failure. Protocol messages or data that lie in TTP input communication channels are lost (the TTP cannot receive messages while being in state `L_FAILED`).
TTP9	The TTP recovers from a site failure, returns debited amount to the consumer's account (by means of permanent storage) and sends to the merchant an "Aborted" message. Protocol messages or data that lie in TTP input communication channels are lost (the TTP cannot receive messages while being in state `ST_FAILED`).
TTP11	The TTP site recovers from a failure and sends to the merchant a "Succeed" transaction result accompanied by the required decryption key (that is retrieved by means of permanent storage). Protocol messages or data that lie in TTP input communication channels are lost (the TTP cannot receive messages while being in state `N_FAILED`).

3.5 Payment participants' CP-nets

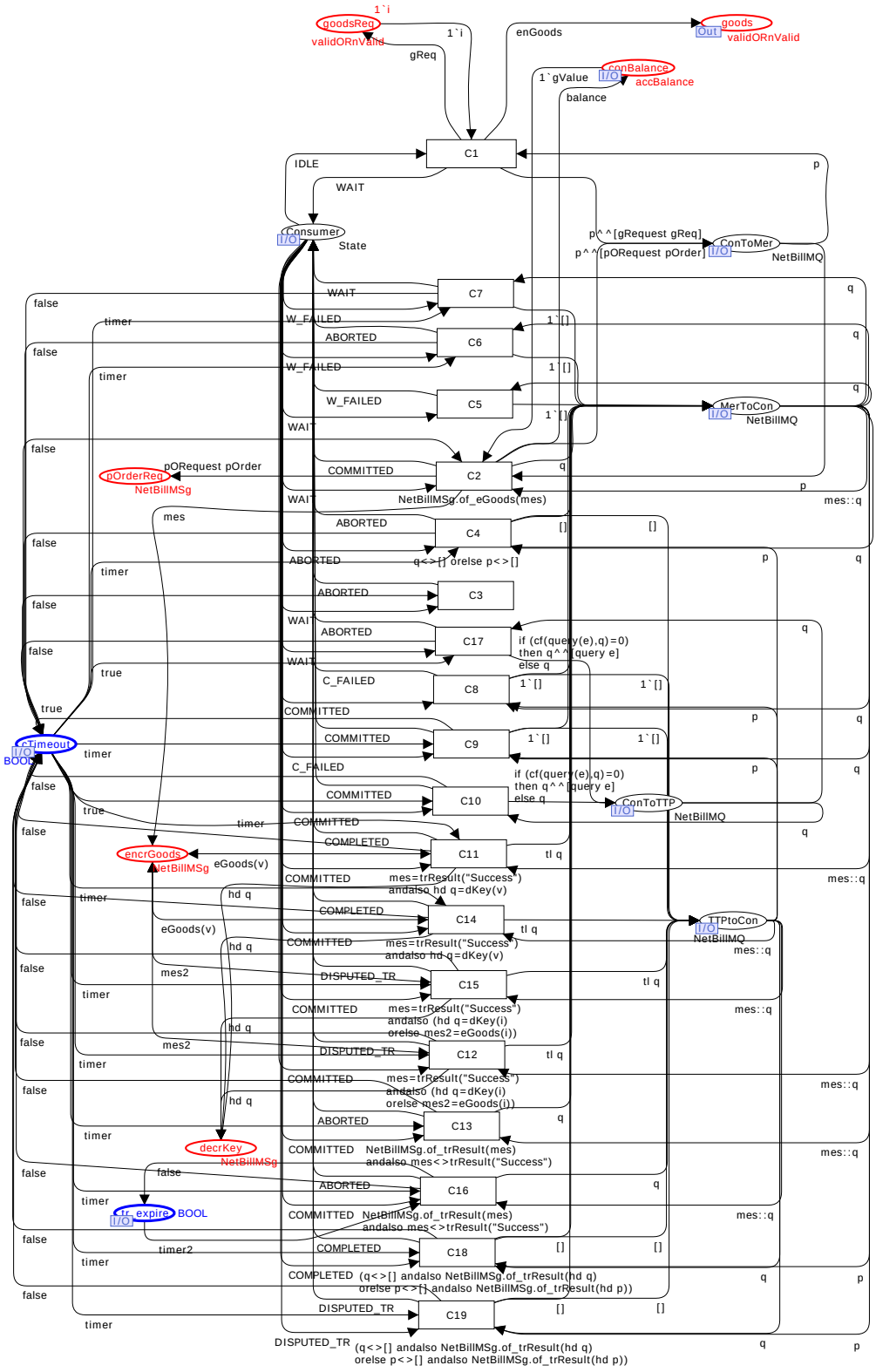

Figure 7. Consumer's CP-net for the NetBill payment system

The CP-net of Figure 7 implements *C*'s finite state automaton as it is specified in Section 3.4. C1 is the single transition that initially is enabled in this CP-net and this transition basically corresponds to the start of a purchase transaction. C1 changes *C*'s state from `IDLE` to `WAIT`, appends a goods request (unbound variable `gReq`) to the `ConToMer` channel and generates an encrypted goods token (unbound variable `enGoods`) for the ongoing purchase transaction. The symbolic value of `enGoods` determines the analyzed protocol execution scenario, in what is concerned with the possibility of *M* to respond with the requested goods (`v`) or not (`i`).

Unbound variables allow us to interactively choose and simulate the protocol execution scenario of interest, but the model's state space analysis includes all possible protocol execution scenarios. Variable `balance` is another unbound variable with values that symbolically represent different cases of account balance. Places `conBalance`, `encrGoods` and `decrKey` store token values for sensitive data that respectively refer to:

- the amount available in *C*'s account, when compared to the ordered goods value,
- the received encrypted goods and more precisely, whether they are the requested ones (`v`) or not (`i`) and
- the received decryption key and more precisely, whether it is the required one (`v`) or not (`i`).

Places `goodsReq` and `pOrderReq` store token values that determine the ongoing purchase transaction, with respect to the validity of (a) the dispatched goods request and (b) the dispatched payment order request.

A `true` token value in `cTimeout` triggers transition C10 that essentially represents the dispatch of a query to the *TTP*, regarding the result of the ongoing payment transaction. Finally, place `tr_expire` represents a timer used by the *TTP* CP-net.

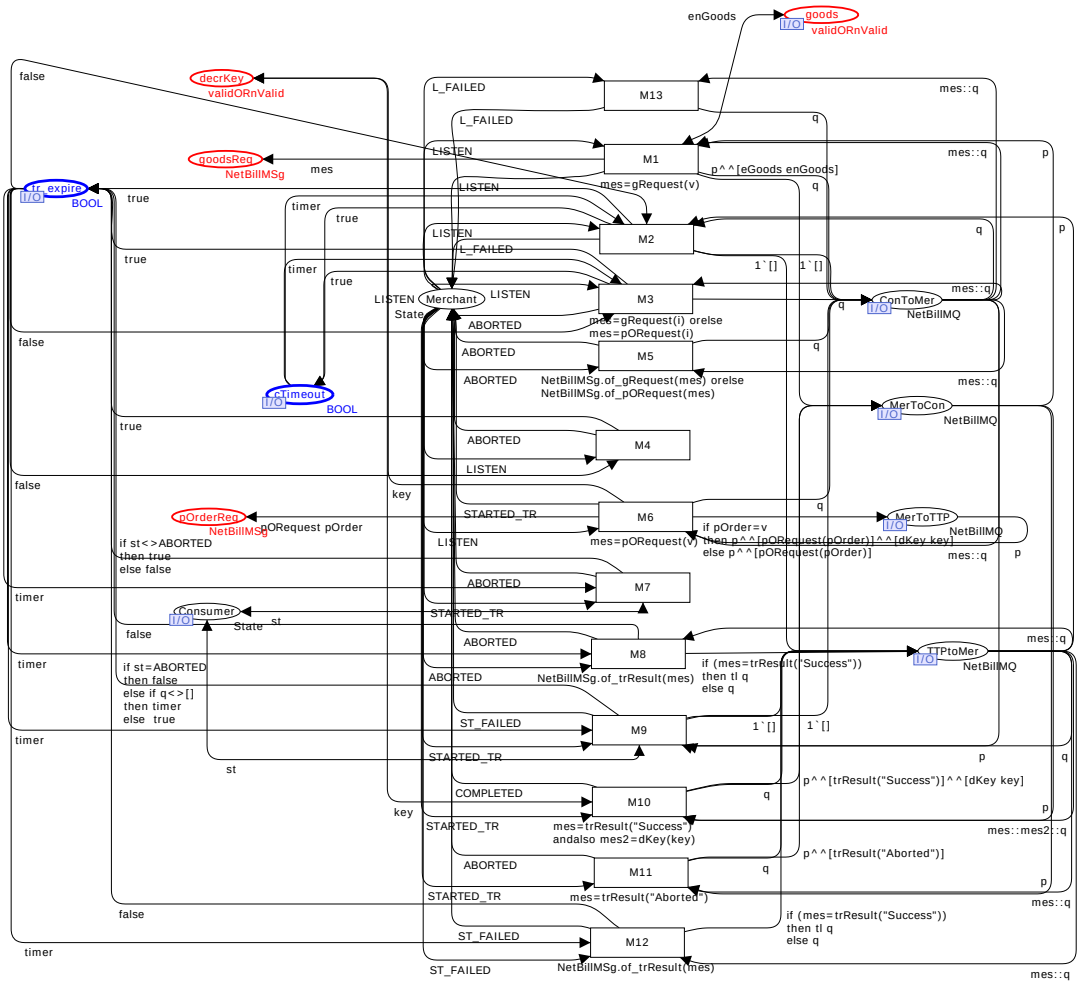

Figure 8. Merchant's CP-net for the NetBill payment system

The CP-net in Figure 8 implements *M*'s finite state automaton as it is specified in Section 3.4. Transition M1 is enabled when *M*'s state is `LISTEN` and if there is a valid goods request in the `ConToMer` channel. Occurrence of M1 places the encrypted goods token value generated by C1 (found in place `goods`) in the `MerToCon` channel. Variables `pOrder` and `key` are unbound variables that respectively represent

- whether the payment order including the data filled by *M* is valid (`v`) or not valid (`i`) and

- the sent decryption key and more precisely, whether it is the required one (`v`) or not (`i`).

The token values that determine the ongoing protocol execution scenario are stored in places `pOrderReq` and `decrKey`. *M*'s CP-net also uses the places `cTimeout` and `tr_expire` that have been already described when introducing *C*'s CP-net.

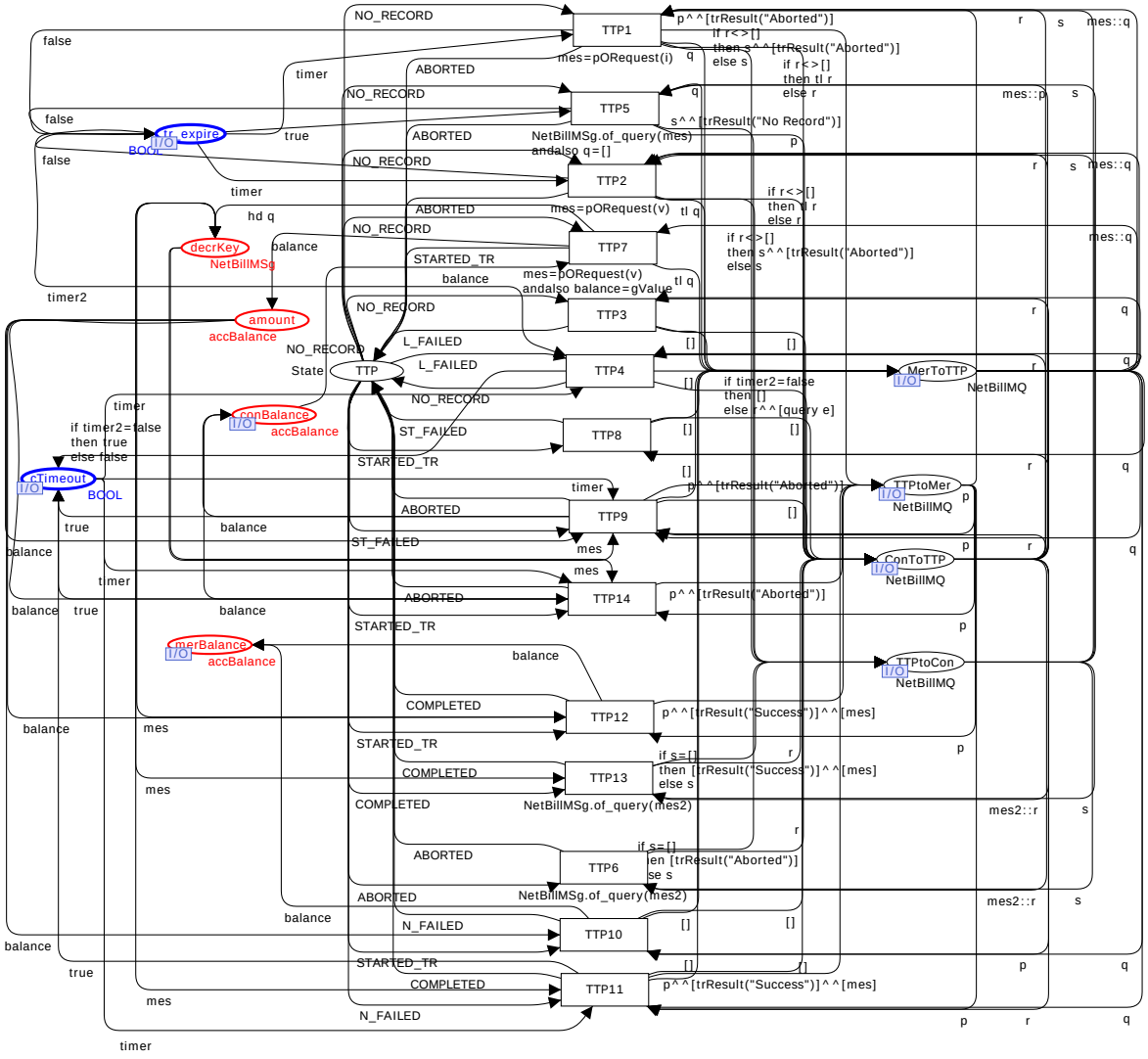

Figure 9. The *TTP* CP-net for the NetBill payment system

Figure 9 shows the CP-net that implements TTP's finite state automaton. Place `merBalance` represents the amount transferred to M's account, in case of a succeed payment. Place `amount` stores the amount to be transferred and place `decrKey` stores the decryption key to be transmitted. In both places, sensitive data are stored temporarily and they are used as prescribed by the adopted trust assumptions.

3.6 State space analysis and model validation

State space analysis is used to explore a standard set of dynamic properties for the developed high-level protocol model and to validate (or correct) the model with respect to a set of model correctness criteria that include:

(a) the absence of self-loop terminal markings,

(b) correct protocol termination and absence of deadlocks,

(c) the absence of livelocks and

(d) the validity of the adopted trust assumptions.

Figure 10 shows the standard state space analysis report for the NetBill CP-net. The first part of the shown standard report provides statistical information regarding the automatically generated *state space graph* (also called occurrence graph). The NetBill state space includes 6439 markings and 18915 arcs that represent the occurrence of different transition instances. The corresponding *graph of strongly connected components* (Scc graph) includes 2678 nodes and 11257 arcs.

The checked *bounds-related properties* characterize the CP-net in terms of the tokens we may have at the places of interest. The shown integer bounds refer to the upper bounds and lower bounds of the number of tokens we may have and essentially provide a mean to explore the places that represent sensitive information.

```
Statistics
----------------------------------------------------------------------
  State Space
    Nodes:  6439
    Arcs:   18915
    Secs:   30
    Status: Full
  Scc Graph
    Nodes:  2678
    Arcs:   11257
    Secs:   2

Boundedness Properties
----------------------------------------------------------------------
  Best Integers Bounds    Upper      Lower
  Consumer'decrKey 1      1          0
  Consumer'encrGoods 1    1          0
  Consumer'goodsReq 1     1          1
  Consumer'pOrderReq 1    1          0
  Merchant'Merchant 1     1          1
  Merchant'decrKey 1      1          0
  Merchant'pOrderReq 1    1          0
  TTP'TTP 1               1          1
  TTP'amount 1            1          0
  TTP'decrKey 1           1          0
  TopLevel'ConToMer 1     1          1
  TopLevel'ConToTTP 1     1          1
  TopLevel'Consumer 1     1          1
  TopLevel'MerToCon 1     1          1
  TopLevel'MerToTTP 1     1          1
  TopLevel'TTPtoCon 1     1          1
  TopLevel'TTPtoMer 1     1          1
  TopLevel'cTimeout 1     1          1
  TopLevel'conBalance 1   1          0
  TopLevel'goods 1        1          0
  TopLevel'merBalance 1   1          0
  TopLevel'tr_expire 1    1          1

Home Properties
----------------------------------------------------------------------
  Home Markings:  None

Liveness Properties
----------------------------------------------------------------------
  Dead Markings:  72 [963,665,592,5905,5890,...]
  Dead Transitions Instances: None
  Live Transitions Instances: None

Fairness Properties
----------------------------------------------------------------------
   Consumer'C1 1          No Fairness            Merchant'M13 1         No Fairness
   Consumer'C10 1         No Fairness            Merchant'M2 1          No Fairness
   Consumer'C11 1         Fair                   Merchant'M3 1          No Fairness
   Consumer'C12 1         Fair                   Merchant'M4 1          No Fairness
   Consumer'C13 1         Fair                   Merchant'M5 1          No Fairness
   Consumer'C14 1         Just                   Merchant'M6 1          No Fairness
   Consumer'C15 1         Just                   Merchant'M7 1          No Fairness
   Consumer'C16 1         Just                   Merchant'M8 1          No Fairness
   Consumer'C17 1         No Fairness            Merchant'M9 1          No Fairness
   Consumer'C18 1         Fair                   TTP'TTP1 1             No Fairness
   Consumer'C19 1         Fair                   TTP'TTP10 1            No Fairness
   Consumer'C2 1          No Fairness            TTP'TTP11 1            No Fairness
   Consumer'C3 1          No Fairness            TTP'TTP12 1            No Fairness
   Consumer'C4 1          No Fairness            TTP'TTP13 1            No Fairness
   Consumer'C5 1          No Fairness            TTP'TTP14 1            No Fairness
   Consumer'C6 1          No Fairness            TTP'TTP2 1             No Fairness
   Consumer'C7 1          No Fairness            TTP'TTP3 1             No Fairness
   Consumer'C8 1          No Fairness            TTP'TTP4 1             No Fairness
   Consumer'C9 1          No Fairness            TTP'TTP5 1             No Fairness
   Merchant'M1 1          No Fairness            TTP'TTP6 1             No Fairness
   Merchant'M10 1         No Fairness            TTP'TTP7 1             No Fairness
   Merchant'M11 1         No Fairness            TTP'TTP8 1             No Fairness
   Merchant'M12 1         No Fairness            TTP'TTP9 1             No Fairness
```

Figure 10. State space analysis report for the NetBill CP-net

A more detailed version of the shown report provides *upper and lower multi-set bounds* that include all token values that is possible to appear in the places of interest (by definition, the upper multi-set bound of a place is the smallest multi-set which is larger than all reachable markings of the place). Apart from the places that represent sensitive information, this allows us to explore the contents of the lists used as participants' communication channels, as well as the participants' reachable states. For the NetBill CP-net we did not find unreachable participants' states and unexpected (combinations of) protocol messages in participants' communication channels. Also, we verified that the token values that appear in places that represent sensitive information reflect all possible protocol participants' inputs.

Home properties provide information about markings or sets of markings to which it is always possible to return. However, the protocol termination correctness assumption requires *the model to not include home markings* and this is successfully verified for the developed NetBill CP-net.

Liveness properties provide information regarding:

- The number of *dead markings* that is, markings with no enabled transitions. Dead markings are protocol termination or deadlock states and CP-net correctness in terms of them requires further analysis.
- The number of *dead transitions* that is, transitions that are not enabled in at least one reachable marking.
- The number of *live transitions* that is, transitions that always can become enabled once more.

As expected, the developed NetBill CP-net does not include dead and live transitions, but correct protocol termination and absence of deadlocks requires further analysis. An

enumeration of all dead markings is easily obtained by the non-standard query shown in Figure 11.

```
let
  val fid = TextIO.openOut "ListOfDeadMarkings.txt"
  val _ = TextIO.output(fid, "List of dead markings: \n")
  val _ = EvalNodes(ListDeadMarkings(),
          fn n => INT.output(fid,n) )
  val _ = TextIO.output(fid, "\nNumber of dead markings: ")
  val _ = INT.output(fid,length (ListDeadMarkings()))
in
  TextIO.closeOut(fid)
end
```

```
type ListOfDeadMarkings.txt

List of dead markings:
963 665 592 5905 5890 5875 5860 5269 5249
5229 5209 519 518 516 515 3555 3549 3543
3537 3380 3377 3373 3370 3366 3360 3169
3166 3144 3138 3132 3126 2969 2966 2962
2959 2955 2949 2758 2755 2716 2708 2687
2679 2648 2640 2619 2611 1925 1922 1921
1920 1851 1841 1833 1815 1805 1797 1793
1790 1786 1783 1725 1715 1707 1689 1679
1671 1667 1664 1660 1657 1084
Number of dead markings: 72
```

Figure 11. Dead markings in the NetBill CP-net

The last part of the produced standard analysis report (Figure 10) refers to the model's *fairness properties* (these properties are not related to the fair exchange transaction guarantee defined in Section 2.2) and provides information about how often the individual transitions occur. An *impartial transition* occurs infinitely often in any infinite occurrence sequence. If this transition ceases to occur, then the protocol must terminate after some number of additional steps. The same is true if a transition is found to be *fair* (like the transitions C11, C12, C13, C18 and C19), which is a weaker fairness property stating that the transition occurs infinitely often in all infinite occurrence sequences where it is infinitely often enabled. Finally, a transition is *just* (like the transitions C14, C15 and C16), if the transition occurs in all cases where this transition is persistently enabled. We easily verify that the above fairness interpretations are compatible with the descriptions of the forenamed transitions that are given in Table 1.

Regarding the analysis needed to prove correct protocol termination, Figure 12 shows the non-standard state space query verifying that the NetBill CP-net does not include self-loop terminal markings. This means that all protocol termination cases are included in the

list of dead markings shown in Figure 11. This finding is crucial for correctly expressing the CTL-based formulae used to verify the transaction guarantees of interest.

```
fun SelfLoopTerminal n=(OutNodes(n)=[n])
fun InValidTerminal()=PredNodes(EntireGraph,
                                fn n => (SelfLoopTerminal n),
                                NoLimit);
let
  val fid = TextIO.openOut "ListOfSelfLoops.txt"
  val _ = TextIO.output(fid, "List of self loop terminals: \n")
  val _ = EvalNodes(InValidTerminal(),
                    fn n => INT.output(fid,n) )
in
  TextIO.closeOut(fid)
end
```

```
type ListOfSelfLoops.txt

List of self loop terminals:
```

Figure 12. Absence of self-loop terminal nodes

```
fun ValidTerminal n=(length (hd (Mark.TopLevel'ConToMer 1 n))=0 andalso
            length (hd (Mark.TopLevel'MerToCon 1 n))=0 andalso
            length (hd (Mark.TopLevel'ConToTTP 1 n))=0 andalso
            length (hd (Mark.TopLevel'TTPtoCon 1 n))=0 andalso
            length (hd (Mark.TopLevel'MerToTTP 1 n))=0 andalso
            length (hd (Mark.TopLevel'TTPtoMer 1 n))=0 andalso

            (Mark.Consumer'Consumer 1 n=[ABORTED] andalso
                Mark.TTP'TTP 1 n=[ABORTED] andalso
                (Mark.Merchant'Merchant 1n=[ABORTED]
                    orelse Mark.Merchant'Merchant 1n=[L_FAILED]
                    orelse Mark.Merchant'Merchant 1n=[ST_FAILED])
             orelse Mark.Consumer'Consumer 1 n=[DISPUTED_TR] andalso
                Mark.TTP'TTP 1 n=[COMPLETED] andalso
                (Mark.Merchant'Merchant 1 n=[COMPLETED]
                    orelse Mark.Merchant'Merchant 1 n=[ST_FAILED]
                    orelse Mark.Merchant'Merchant 1 n=[ABORTED])
             orelse Mark.Consumer'Consumer 1 n=[COMPLETED] andalso
                Mark.TTP'TTP 1 n=[COMPLETED] andalso
                (Mark.Merchant'Merchant 1 n=[COMPLETED]
                    orelse Mark.Merchant'Merchant 1 n=[ST_FAILED]
                    orelse Mark.Merchant'Merchant 1 n=[ABORTED])))

fun InValidTerminal()=PredNodes(ListDeadMarkings(),
                                fn n => not (ValidTerminal n),
                                NoLimit);
let
  val fid = TextIO.openOut "DeadlockMarkings.txt"
  val _ = TextIO.output(fid, "List of deadlock markings: \n")
  val _ = EvalNodes(InValidTerminal(),
          fn n => INT.output(fid,n) )
in
  TextIO.closeOut(fid)
end
```

```
type DeadlockMarkings.txt

List of deadlock markings:
```

Figure 13. Absence of deadlock markings

Figure 13 shows the non-standard state space query verifying that the NetBill CP-net does not include deadlock markings. We prove that all protocol termination cases (dead

markings) are correct that is, communication channels are empty and we only have valid participants' state combinations. More precisely, we verify that when *C* ends in state `COMPLETED`, this is also true for the *TTP*, but *M* can be in state `COMPLETED`, `ST_FAILED` or `ABORTED` depending on the merchant's honesty or the occurred site failures or message losses. When *C* ends in state `DISPUTED_TR` we require the *TTP* to end in state `COMPLETED` (the *TTP* includes only two final states that is, states `COMPLETED` and `ABORTED`; merchant fraud regarding the delivered goods - disputed transaction - is not tangible by the *TTP*) and *M* to end as in the previous termination case. Finally, when *C* ends in state `ABORTED` we require the *TTP* to end in state `ABORTED` and *M* to end in state `ABORTED`, `L_FAILED` or `ST_FAILED` depending on the merchant's unilateral decision or the occurred site failures and message losses. Function `InValidTerminal()` checks for invalid dead markings and the empty list obtained in query's output confirms the absence of deadlocks.

An important model correctness criterion is the absence of livelocks. A livelock is detected, when the state space contains a cycle that leads to no markings outside the cycle. In this case, once the cycle is entered it will repeat forever. A protocol model, which terminates properly, should be free from livelocks.

A convenient way to check the absence of livelocks is to study the automatically generated graph of strongly connected components (Scc graph). A strongly connected component of the state space is a maximal sub-graph whose nodes are mutually reachable from each other. The generated Scc graph has a node for each strongly connected component and includes arcs that connect two different components, if there is an arc in the original state space graph that connects a node of the first component to a node that belongs to the second one. As initial strongly connected component is characterized a component without incoming arcs and as terminals are characterized all components without outgoing

arcs. Since each node in the state space belongs to only one strongly connected component, the Scc graph is always smaller than or equal to the corresponding state space.

Depending on the structure of the generated state space, model checking the absence of livelocks takes one of the following two forms:

- If the state space and its Scc graph are isomorphic and also there are no self-loops, then the protocol model is free of livelocks.
- If the state space contains self-loops or if there is at least one strongly connected component that consists of more than one node, then we need to examine if all terminal components are trivial that is, they consist of a single node and no arcs. A non-trivial terminal component represents a livelock in the protocol model.

The non-standard state space query shown in Figure 14 verifies that the NetBill Scc graph does not include non-trivial terminal strongly connected components and so the developed protocol model is free of livelocks.

```
fun ListTerminalSCCs()=PredAllSccs(SccTerminal);
fun InValidTermSCC()=PredSccs(ListTerminalSCCs(),
                        fn n => not (SccTrivial n),
                        NoLimit);
let
  val fid = TextIO.openOut "AbsenceOfLivelocks.txt"
  val _ = if InValidTermSCC()=[]
          then TextIO.output(fid, "No Livelocks!")
          else TextIO.output(fid, "Livelocks detected!")
in
  TextIO.closeOut(fid)
end
```

```
type AbsenceOfLivelocks.txt

No Livelocks!
```

Figure 14. Absence of livelocks

In fact, the state space queries shown in Figures 12 and 13, as well as other auxiliary queries, helped us to debug the NetBill CP-net with respect to the described model correctness criteria. Also, we minimized the number of correct dead markings to only 72, by having managed to exclude similar markings that differ only in terms of the token values

of cTimeout and tr_expire. We remind that these places are used to express the triggering of queries and such a difference is irrelevant for all protocol termination states.

```
fun TrCompleted n = (Mark.TTP'TTP 1 n=[COMPLETED]);
fun reportSucceed n = (List.nth(Mark.TopLevel'TTPtoMer 1 n,0)<>[] andalso
        List.nth(List.nth(Mark.TopLevel'TTPtoMer 1 n,0),0) = trResult"Success");
fun reportNotSucceed n = (List.nth(Mark.TopLevel'TTPtoMer 1 n,0)<>[] andalso
       List.nth(List.nth(Mark.TopLevel'TTPtoMer 1 n,0),0) <> trResult"Success");
fun creditDone n = (Mark.TopLevel'merBalance 1 n = [gValue]);
val trustViolation1States = PredNodes(EntireGraph,
                                fn n => (TrCompleted n andalso reportNotSucceed n),
                                NoLimit);
val trustViolation2States = PredNodes(EntireGraph,
                                fn n => (reportSucceed n andalso not (creditDone n)),
                                NoLimit);
let
  val fid = TextIO.openOut "TrustAssumptionsA.txt"
  val _ = if (trustViolation1States=[] andalso trustViolation2States=[])
          then TextIO.output(fid,"No trust assumptions violation!")
          else TextIO.output(fid,"Trust assumptions violation detected!")
in
  TextIO.closeOut(fid)
end
```

```
type TrustAssumptionsA.txt

No trust assumptions
violation!
```

Figure 15. Model checking trust assumptions violation (A)

The query of Figure 15 model checks the trust assumptions for the successful completion of the ongoing payment transaction. Set trustViolation1States detects if there are markings, where *TTP*'s state is COMPLETED and the *TTP* does not report the correct transaction result through the TTPtoMer communication channel. Set trustViolation2States detects if there are markings, where *TTP* reports a success transaction result, but the payment (gValue) has not been credited to *M*'s account (merBalance). No trust assumption violation is detected.

The query of Figure 16 model checks the trust assumptions regarding the aborted payment transactions. In this case, the transaction result is reported either through the TTPtoMer or through the TTPtoCon communication channel and model checking attempts to detect two different sets of trust violation markings. Set trustViolation1States detects if there are markings, where *TTP*'s state is ABORTED and the *TTP* does not report the correct transaction result. Set trustViolation2States detects if there are markings, where *TTP* reports an aborted

transaction result, but the *TTP* has debited *C*'s account (`conBalance`) as opposed to what is expected. No trust assumption violation is detected.

```
fun TrAborted n = (Mark.TTP'TTP 1 n=[ABORTED]);
fun reportSucceed n = ((List.nth(Mark.TopLevel'TTPtoMer 1 n,0)<>[] andalso
   List.nth(List.nth(Mark.TopLevel'TTPtoMer 1 n,0),0) = trResult"Success") orelse
                        (List.nth(Mark.TopLevel'TTPtoCon 1 n,0)<>[] andalso
       List.nth(List.nth(Mark.TopLevel'TTPtoCon 1 n,0),0) = trResult"Success"));
fun reportNotSucceed n = ((List.nth(Mark.TopLevel'TTPtoMer 1 n,0)<>[] andalso
 List.nth(List.nth(Mark.TopLevel'TTPtoMer 1 n,0),0) <> trResult"Success") orelse
                          (List.nth(Mark.TopLevel'TTPtoCon 1 n,0)<>[] andalso
      List.nth(List.nth(Mark.TopLevel'TTPtoCon 1 n,0),0) <> trResult"Success"));
fun debitDone n = (Mark.TopLevel'conBalance 1 n = []);
val trustViolation1States = PredNodes(EntireGraph,
                                      fn n => (TrAborted n andalso reportSucceed n),
                                      NoLimit);
val trustViolation2States = PredNodes(EntireGraph,
                                    fn n => (reportNotSucceed n andalso debitDone n),
                                    NoLimit);
let
  val fid = TextIO.openOut "TrustAssumptionsB.txt"
  val _ = if (trustViolation1States=[] andalso trustViolation2States=[])
          then TextIO.output(fid,"No trust assumptions violation!")
          else TextIO.output(fid,"Trust assumptions violation detected!")
in
  TextIO.closeOut(fid)
end
```

```
type TrustAssumptionsB.txt

No trust assumptions
violation!
```

Figure 16. Model checking trust assumptions violation (B)

In the NetBill CP-net, money conservation is a *TTP* responsibility and for this reason, it is also one of the model's correctness criteria (trust assumption) that have to be checked. However, in the general case, payment systems are not necessary to involve trusted parties and therefore money conservation is not always one of the model's trust assumptions ([48]). Moreover, since money conservation is conveniently expressed as a CTL property, model checking this guarantee is presented together with other protocol correctness criteria in the next section.

4. Model checking payment transaction guarantees

The NetBill system aims to provide a wide range of payment transaction guarantees, from those mentioned in section 2.2. Protocol's design adopts an *encryption-based atomicity approach* ([44]), where the goods are initially sent to *C* in an encrypted form and therefore

cannot be used, without the required decryption key. The key is dispatched only on receipt of the corresponding payment. On the other hand, payment systems that adopt an *authority-based atomicity approach* ([38]) require the *TTP* to retain an escrowed copy of the purchased goods.

There is an increasing interest for payment systems, which will be based on the already known "non-blocking" commit protocols, as well as, for distributed payment systems ([41]) and systems for offline payments. Also, an active field of research is the design of systems with a semi-trusted third party ([18]) that is possible to misbehave on its own, but will not collude with any of the payment participants.

The described CP-net modeling approach is a formal analysis alternative for proving the transaction guarantees of section 2.2. The current section introduces the use of CTL-based model checking in terms of the developed NetBill CP-net. Preliminary results regarding some of the transaction guarantees of section 2.2 were first shown in [24].

We propose the use of the ASK-CTL library ([11]) of the CPN Tools ([13]), in order to express the expected transaction guarantees as properties of *paths* in the state space. A path is a sequence of states and transition occurrences of the state space, constrained by the direction of arcs. Paths may be infinite and this implies some difficulties in selecting the proper path quantification operators, based on their semantics. An ASK-CTL formula is interpreted either over the domain of states or over the domain of transition occurrences in a path. However, the domain switch operator `MODAL` allows one to jump from one domain to the other.

4.1 Model checking money conservation

Money conservation is a requisite in all payment systems. We do not accept the protocol to leave one or more transactions in a partial or ambiguous state, where the system has created

or destroyed money. This guarantee should be assured in all cases of site failures, message losses and fraudulent behavior.

In an account transfer system with a *TTP* like NetBill, money conservation is included in the model's trust assumptions and it guarantees that either the funds transfer should complete (*M* has the money and *C* does not) or it should not occur at all (*C* has the money and *M* does not).

In Figure 6 we observe that funds transfer commences on the occurrence of TTP7. Thus, model checking money conservation includes all paths starting with an arc that corresponds to the occurrence of TTP7 (96 paths). In Figure 17, this list of arcs (transition occurrences) is given by the value `debitTIs`. Having moved to a non-legitimate state, we require the system to guarantee that *eventually* either, *M* has the money and *C* does not, or *C* has the money and *M* does not.

```
fun debitC a = (ArcToTI a = TI.TTP'TTP7 1);
val debitTIs = PredArcs(EntireGraph,
                        fn a => (debitC a),
                        NoLimit);
fun moneyLoss n = ((Mark.TopLevel'conBalance 1 n=[]
                        andalso Mark.TopLevel'merBalance 1 n=[])
                   orelse (Mark.TopLevel'conBalance 1 n<>[]
                        andalso Mark.TopLevel'merBalance 1 n<>[]));
val debitAction = AF("No debit!",debitC);
val mLossForm = MODAL(NF("",moneyLoss));
val noMoneyConservation = INV(ALONG(mLossForm));
fun verify a = eval_arc noMoneyConservation a;
val results = map verify debitTIs;
let
  val fid = TextIO.openOut "MoneyConservation.txt"
  val _ = if (cf(true,results)>0)
          then TextIO.output(fid,"No money conservation!")
          else TextIO.output(fid,"Protocol does not create or destroy money!")
in
  TextIO.closeOut(fid)
end
```

```
type MoneyConservation.txt

Protocol does not create or
destroy money!
```

Figure 17. Money conservation in the NetBill CP-net

To avoid using the `EV` operator, where for some argument say A, `EV(A)`≡ `FORALL_UNITL(TT,A)` holds, if A becomes true in a finite - but not infinite - number of steps, we model check the converse: there is no reachable path, where for every state

neither *M* nor *C* has the money and there is also no reachable path, where for every state both *M* and *C* have the money. In Figure 17, this is expressed by the value `mLossForm`.

Operator `INV` (where for some argument A, `INV(A)≡NOT(POS(NOT(A)))`) returns true, if the argument is true for all reachable states (or arcs) from the state (or arc) we are now. Finally, operator `ALONG` (where `ALONG(A)≡NOT(EV(NOT(A)))`) can also be applied in infinite paths and returns true, if there is a path for which the argument holds for every state (or arc). Model checking is performed by `eval_arc`.

4.2 Model checking goods atomicity (fair exchange)

Having verified money conservation, goods atomicity (fair exchange) also requires that goods (or payment receipt) will be obtained, if and only if, the payment is transferred to the merchant or the merchant's account. Goods atomicity is an important guarantee that in bilateral payment transactions has to be ensured from both participants' perspectives, in all cases of site failures, message losses, unilateral aborts and fraudulent behavior.

```
fun validEPObyC n = (Mark.TopLevel'ConToMer 1 n = [[pORequest(v)]]);
val validEPOstates = PredNodes(EntireGraph,
                               fn n => (validEPObyC n),
                               NoLimit);
fun noMoney n = (Mark.TopLevel'conBalance 1 n = []);
fun noDecrKey n = (Mark.Consumer'decrKey 1 n = []);
val paidTrans = NF("No payment!", noMoney);
val noGoods = NF("Found decr key!", noDecrKey);
val noGAtomicityForC = AND(POS(INV(paidTrans)), INV(noGoods));
fun verify n = eval_node noGAtomicityForC n;
val results = map verify validEPOstates;
let
  val fid = TextIO.openOut "GoodsAtomicityForC.txt"
  val _ = if (cf(true,results)>0)
          then TextIO.output(fid,"Non atomic goods delivery for C!")
          else TextIO.output(fid,"Atomic goods delivery for C!")
in
  TextIO.closeOut(fid)
end
```

```
type GoodsAtomicityForC.txt

Atomic goods delivery for C!
```

Figure 18. Goods atomicity from the consumer's perspective

In Figure 18, value `noGAtomicityForC` expresses the possibility of some state, where for each reachable state we have at the same time no money in *C*'s account

(Consumer has paid) and *C* does not own the decryption key for the paid goods. Function `eval_node` model checks the absence of goods atomicity for all states, in which *C* has dispatched a valid payment order (`pORequest(v)`). The obtained result proves goods atomicity from the consumer's perspective.

Figure 19, proves goods atomicity from the merchant's perspective: there is no state, in which for each reachable state *M* does not own the money for the payment and at the same time, it is possible for *C* to get the decryption key and to keep it forever.

```
fun sentEPObyC n = (Mark.TopLevel'ConToMer 1 n = [[pORequest(v)]]
                    orelse Mark.TopLevel'ConToMer 1 n = [[pORequest(i)]]);
val sentEPOstates = PredNodes(EntireGraph,
                              fn n => (sentEPObyC n),
                              NoLimit);
fun noMoney n = (Mark.TopLevel'merBalance 1 n = []);
fun foundDecrKey n = (Mark.Consumer'decrKey 1 n <> []);
val notPaidTrans = NF("Payment found!", noMoney);
val goodsDelivered = NF("No goods delivered!", foundDecrKey);
val noGAtomicityForM = AND(INV(notPaidTrans), POS(INV(goodsDelivered)));
fun verify n = eval_node noGAtomicityForM n;
val results = map verify sentEPOstates;
let
  val fid = TextIO.openOut "GoodsAtomicityForM.txt"
  val _ = if (cf(true,results)>0)
          then TextIO.output(fid, "Non atomic goods delivery for M!")
          else TextIO.output(fid, "Atomic goods delivery for M!")
in
  TextIO.closeOut(fid)
end
```

```
type GoodsAtomicityForM.txt

Atomic goods delivery for M!
```

Figure 19. Goods atomicity from the merchant's perspective

4.3 Model checking certified delivery

Having proved money conservation and goods atomicity, certified delivery also requires that all payment participants can prove the sensitive details of the performed transaction. Certified delivery in bilateral payment transactions should be ensured from both participants' perspectives, in all cases of site failures, message losses, unilateral aborts and fraudulent behavior.

In the NetBill payment system, *C* proves that the received encrypted goods are intact, if he owns the checksum number of the received goods and if this number is the same as the one gathered by the *TTP* through *C*'s payment order. The code of Figure 20 proves that it is

not possible to eventually deliver any goods decryption key when *C* has not obtained the corresponding checksum number.

```
fun noChecksumOwnedByC n = (Mark.Consumer'encrGoods 1 n = []);
fun foundDecrKey n = (Mark.Consumer'decrKey 1 n <> []);
val noChecksum = NF("Checksum owned by C!", noChecksumOwnedByC);
val goodsDelivered = NF("No goods delivered!", foundDecrKey);
val noCertifiedDeliveryForC = POS(AND(ALONG(noChecksum),EV(goodsDelivered)));
val result = eval_node noCertifiedDeliveryForC InitNode;
let
  val fid = TextIO.openOut "CertifiedDelForC.txt"
  val _ = if (result=true)
          then TextIO.output(fid, "Non certified delivery for C!")
          else TextIO.output(fid, "Certified delivery for C!")
in
  TextIO.closeOut(fid)
end
```

```
type CertifiedDelForC.txt

Certified delivery for C!
```

Figure 20. Certified delivery from the consumer's perspective

On the other hand, the code of Figure 21 proves that it is not possible for *M* to have endorsed *C*'s payment order when he has not have received the corresponding *C*'s goods request. Goods request's checksum number is in fact included in an encrypted part of *C*'s endorsed payment order, which can be read only by the *TTP*. The obtained result proves the certified delivery guarantee from the merchant's perspective.

```
fun noGRequestOwnedByM n = (Mark.Merchant'goodsReq 1 n = []);
fun foundEndorsedEPO n = (Mark.Merchant'pOrderReq 1 n <> []);
val noGRequest = NF("", noGRequestOwnedByM);
val endorsedEPO = NF("No goods delivered!", foundEndorsedEPO);
val noCertifiedDeliveryForM = POS(AND(ALONG(noGRequest), EV(endorsedEPO)));
val result = eval_node noCertifiedDeliveryForM InitNode;
let
  val fid = TextIO.openOut "CertifiedDeliveryForM.txt"
  val _ = if (result=true)
          then TextIO.output(fid,"Non certified delivery for M!")
          else TextIO.output(fid,"Certified delivery for M!")
in
  TextIO.closeOut(fid)
end
```

```
type CertifiedDelForM.txt

Certified delivery for M!
```

Figure 21. Certified delivery from the merchant's perspective

4.4 Model checking protection of participants' interests

We remind that the high-level guarantee of protection of participants' interests ensures that participants get exactly what they are legitimate to get, in all cases of site failures, message losses, unilateral aborts and fraudulent behavior.

For the NetBill payment system, collusions between the *TTP* and *C* or between the *TTP* and *M* are not studied, as a consequence of the adopted trust assumptions. Thus, protection of participants' interests takes the following context:

C's protection guarantee - "If *M* is entitled to a payment, then *C* actually receives the goods, or *C* can claim them in an offline dispute handling."

M's protection guarantee - "If *C* actually receives the goods, or *C* can claim them in an offline dispute handling, then *M* is entitled to a payment."

```
fun paymentEntitledForM n = (Mark.Merchant'pOrderReq 1 n = [pORequest(v)]
                                andalso Mark.TTP'TTP 1 n = [COMPLETED]);
fun noGoodsReceivedByC n = (Mark.Consumer'encrGoods 1 n = []
                                orelse Mark.Consumer'decrKey 1 n = []);
fun noGoodsClaimedOfflineByC n = (Mark.Consumer'Consumer 1 n <> [DISPUTED_TR]);
val paymentEntitled = NF("No payment for M!", paymentEntitledForM);
val noGoodsReceived = NF("Goods received by C!", noGoodsReceivedByC);
val noGoodsClaimedOffline = NF("Goods claimed offline!", noGoodsClaimedOfflineByC);
val noProtectionForC = POS(INV(AND(paymentEntitled,
                                    AND(noGoodsReceived,noGoodsClaimedOffline))));
val result = eval_node noProtectionForC InitNode;
let
  val fid = TextIO.openOut "ProtectionForC.txt"
  val _ = if (result=true)
          then TextIO.output(fid, "No protection of interests for C!")
          else TextIO.output(fid, "The protocol protects C's interests!")
in
  TextIO.closeOut(fid)
end
```

```
type ProtectionForC.txt

The protocol protects C's
interests!
```

Figure 22. Protection of consumer's interests

The code shown in Figure 22 proves C's protection guarantee as follows. *M* is entitled to a payment, if and only if he has proof that the *TTP* committed the transaction a) with a successful result and b) it did so, in response to a valid electronic payment order.

On the other hand, *C* actually receives the goods during the protocol execution, if and only if *C* receives an encrypted version of the ordered goods alongside a key, and the goods can be obtained by decrypting the encrypted goods with the owned key. Also, *C* can claim

ordered goods in an offline dispute handling, if *C* can show that the transaction was successfully processed by the *TTP*, in response to a valid request, but the goods cannot be retrieved from the decryption key released with the transaction result and the encrypted goods given by *M*.

C's protection guarantee is violated, when in some state the value `noProtectionForC` becomes true. The result ensures that NetBill protects C's interests.

4.5 A replay attack counterexample

As in most well known payment systems, NetBill transactions are amenable to protocol level replay attacks that aim in replaying some protocol messages from a previous legitimate run. Figure 23 introduces a modified NetBill CP-net that compared to the first one shown in Figure 3 provides a basis for proving the double spending possibility and for studying potential countermeasures.

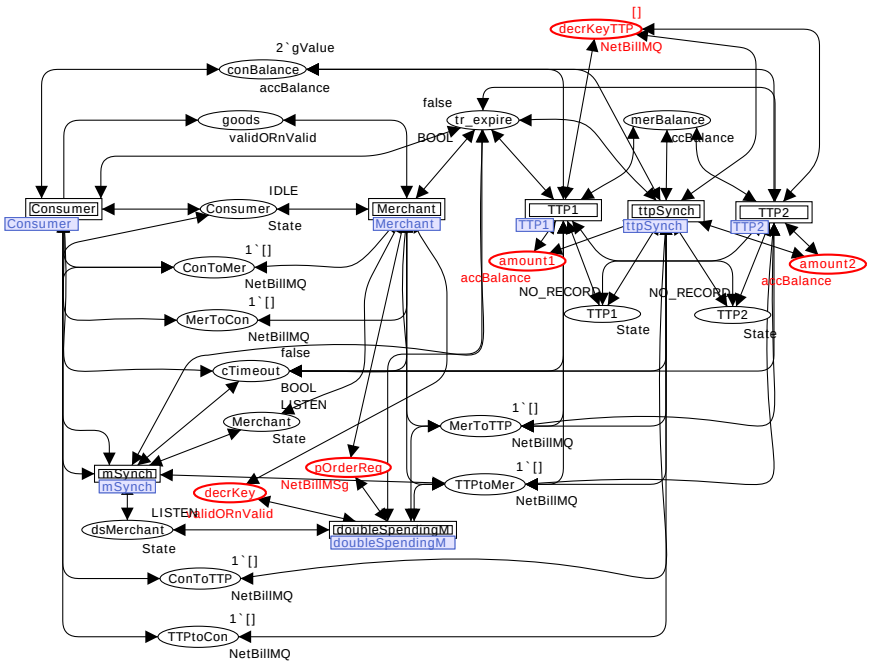

Figure 23. Top level CP-net with a double spending merchant and two TTP threads

We point out the following modifications:

- An additional substitution transition, named `doubleSpendingM`, includes the state transitions for the new place `dsMerchant`, which is used to represent a protocol replay merchant thread. The `mSynch` substitution transition synchronizes the state transitions of the `Merchant` and `dsMerchant` places.
- Two similar *TTP* transitions are used instead of the single *TTP* transition shown in Figure 3. Finally, the new `ttpSynch` transition synchronizes state transitions of the `TTP1` and `TTP2` places, as well as token exchanges between *TTP*'s communication channels and the temporary store places `amount1`, `amount2` and `decrKeyTTP`.

We omit the details of the new substitution transitions, but we note the addition of double charging possibilities by having sent in the `conBalance` place, two `accBalance` tokens instead of one.

```
fun ttp1Charged n=(Mark.TopLevel'TTP1 1 n=[COMPLETED]);
fun ttp2Charged n=(Mark.TopLevel'TTP2 1 n=[COMPLETED]);
fun consumerTrCompleted n=(Mark.TopLevel'Consumer 1 n
                                                =[COMPLETED]);
fun doubleSpending()=PredNodes(ListDeadMarkings(),
                               fn n => (ttp1Charged n
                                        andalso ttp2Charged n
                                       andalso consumerTrCompleted n),
                               NoLimit);
let
val fid=TextIO.openOut "DSpendingCounterexample.txt"
val _ =TextIO.output(fid,"Path to a double spending dead marking: \n")
val _ = EvalArcs(ArcsInPath (1,hd (doubleSpending())),
                 fn a => STRING.output(fid,st_BE(ArcToBE a)))
in
TextIO.closeOut(fid)
end
```

```
type DSpendingCounterexample.txt

Path to a double spending dead marking:
"Consumer'C1 1: {p=[],gReq=v,enGoods=v}"
"Merchant'M1 1: {p=[],enGoods=v,mes=gRequest(v),q=[]}"
"Consumer'C2 1: {p=[],mes=eGoods(v),q=[],pOrder=v,
        balance=gValue,balance2=gValue}"
"Merchant'M6 1: {p=[],mes=pORequest(v),q=[],pOrder=v,
        key=v}"
"TTP1'TTP7 1: {p=[],mes=pORequest(v),q=[dKey(v)],
        balance=gValue}"
"ttpSynch'TTPF7 1: {balance=gValue,r=[],q=[]}"
"ttpSynch'TTPF8 1: {timer=false,p=[],q=[],r=[],
        timer2=false,s=[dKey(v)]}"
"doubleSpendingM'M6 1: {mes=pORequest(v),p=[],key=v}"
"TTP2'TTP7 1: {p=[dKey(v)],mes=pORequest(v),q=[dKey(v)],
        balance=gValue}"
"TTP2'TTP12 1: {q=[dKey(v)],p=[trResult("Success"),dKey(v)],
        balance=gValue}"
"Consumer'C10 1: {q=[]}"
"ttpSynch'TTPF20 1: {s=[],q=[dKey(v)],mes=query(e),r=[],
        st=COMPLETED,st2=COMPLETED}"
"doubleSpendingM'M7 1: {}"
"Consumer'C14 1: {timer=false,mes=trResult("Success"),
        q=[dKey(v)]}"
"mSynch'M9 1: {st2=ABORTED,timer=false,
        q=[trResult("Success"),dKey(v), trResult("Success"),
        dKey(v)],p=[],st=COMPLETED}"
```

Figure 24. Counterexample that represents a protocol replay attack

Figure 24 shows the generation of a counterexample that represents a protocol replay attack. Under the assumptions of no self-loop terminal markings and no livelocks, we examine all dead markings, such that the states of both *TTP* places are `COMPLETED` and the state of the `Consumer` place does not allow *C* to set a transaction dispute. Since the resulted list is not empty, we use function `ArcsInPath` to generate a path from marking 1 to one of the markings included in the list.

Protocol level replay scenarios can be used to simulate undesirable fraudulent behavior, in order to design appropriate double spending prevention schemes. In NetBill ([12]), double spending prevention is based on a globally unique *epo* (electronic payment order) identifier that is first sent by *M* to *C*, together with the encrypted version of the ordered goods. The received *epo* identifier is then included in the payment order signed by *C* and eventually received by the *TTP*. A message freshness indication (a timestamp marking the time at the end of goods delivery) allows the *TTP* to discard stale *epo* identifiers from time to time.

Other common freshness mechanisms are based on the use of *nonces*. Because nonces are unpredictable and are used in only one context, they ensure that a message cannot be reused in later transactions.

4.6 Model checking concurrent payment transactions and protocol-level intruder attacks

Model checking the described transaction guarantees in concurrent payment systems has the following difficulties:

- The time needed to develop a model increases with respect to the number of concurrent payment transactions, due to the complexity of the thread synchronizing substitution transitions (see for example transitions `ttpSynch` and `mSynch` in Figure 23).

- Concurrent payments are likely to result in computationally expensive state spaces.

If we are interested in deriving a transaction guarantee counterexample, we can exploit the CPN Tools state space branching options for the partial generation of the model's state space. We stop after having derived a representative counterexample.

If we are interested in generating the whole state space, we observe that the different orders of firing transitions of participants' threads result in different markings and paths in the model's state space. In order to prevent unnecessary interleaving, we have to define an equivalence relation for any two occurrence sequences, with respect to the checked transaction guarantee. Then, there is no need to include all of the unnecessary interleaving of participants' threads in the generated state space graph. One possibility is to employ an appropriate token-passing scheme, in order to allow only the transitions of one participant's thread to be enabled at a given time.

In [3], the authors use a token-passing scheme to model check a key exchange cryptographic protocol (the TMN protocol), against possible intruder attacks. In a payment system, an intruder attack [6] may also target the system's security guarantees (integrity, authorization, non-repudiation, confidentiality etc), but our modeling approach focuses in model checking the described transaction atomicity guarantees. A payment transaction guarantee that is amenable to different forms of intruder attacks is the distributed payment atomicity.

4.7 Colored Petri Net model checking alternatives

Payment transaction guarantees may also be checked by detailed examination of the protocol's dead markings, but this is not true for all of them. If we consider the money conservation guarantee, we model check that having moved to a non-legitimate state, the system guarantees that eventually either, *M* has the money and *C* does not, or *C* has the

money and *M* does not. By examination of the protocol's dead markings we cannot guarantee that all transitions to non-legitimate states eventually result in legitimate ones.

An alternative CP-net analysis is the invariant analysis. Place invariant analysis aims to formulate some equations, which we postulate to be satisfied independently of the steps that occur. Each invariant property is then transformed to an equivalent place flow that is checked by considering individual transitions one at a time. This implies that for each transition, we need only look at its immediate surroundings. Transition invariants are similar to place invariants, but they are used to determine occurrence sequences that have no total effect, i.e., they have the same start and end markings.

5. Related work

Part of our work refers to model checking fault tolerance with respect to the described payment transaction guarantees. In related work, we refer to the analysis reported in [8] for verifying the redundancy mechanisms employed in fault-tolerant control systems. Two different formalisms are interchangeably used to specify a system: the Calculus of Communicating Systems (CCS)/Meije process algebra [4] and a Labeled Transition System (LTS) representation developed with the ATG tool [40]. The JACK verification environment [10] is then used to generate the whole system's LTS from the provided network of subsystems. Fault tolerance properties like for example the fail-stop, the fail-silence, the fail-safety and other properties are then expressed in Action-based Computation Tree Logic (ACTL) [31] and are checked by the AMC model checker, which is available in JACK.

The most important similarity between the work reported in [8] and the approach of the present article is the fact that both works model explicitly the occurrence of faults as opposed to other works in the literature of fault tolerant systems, which in fact model only the failure behavior itself. However, [8] does not address the possibility of message losses

due to communication failures and also does not meet the problem of verifying trust assumptions and fraud scenarios that are prominent in electronic payment systems. Regarding the adopted approach, we underline the obvious advantages of the CPN Tools alternative, which offers a single integrated environment for model building and simulation, as well as, for expressing and verification of the target system properties. In contrast to the process algebraic approach of [8], the CP-net formalism provides an explicit representation of both states and events and a formal semantics that builds upon true concurrency and sets the ground for a compact description of control and synchronization that is integrated with the description of data manipulation.

Another interesting model checking approach is the one described in [42]. In that work, the authors attempt a finite-state analysis of two contract signing protocols. The correctness properties of interest do not include the atomicity guarantees addressed in the present article, but are related to the fair exchange, the accountability and the abuse-freeness of the examined protocols.

Model checking is based on Murφ [16], a tool that employs its own high-level language for the description of nondeterministic finite-state machines. While there is no explicit notion of process, it is implicitly modeled by a set of related rules and communication between processes is modeled by shared variables. The Murφ system can then check, by explicit state enumeration, if every reachable state of the model satisfies a given set of invariants. This approach seems adequate for correctness properties that are characterized as "monotonic", i.e. properties that if they cease to hold at some point, this does not change in the remainder of the run. However, Murφ and the model checking by state invariants underlie the restrictions discussed in section 3.1, when trying to express nested reachability. This fact is also recognized by the authors of [42], who call "non-monotonic" all properties, which do not hold on intermediate states (as the money conservation guarantee that we

proved in the NetBill case). In effect, they recognize that with tools like Murφ it is only possible "to formalize and check an approximation to a non-monotonic property". In their proposal they conjectured the states that invalidate the checked property and eventually verified this conjecture by analyzing in Murφ a modified protocol environment. The CP-net approach discussed in present article does not underlie the described restrictions in expressing complex correctness properties. Invariant analysis is not excluded, but in CP-nets this analysis concerns the place or transition invariants and it is not based on explicit state enumeration, with the associated state space explosion risks.

In the field of e-commerce transactions, CP-nets have also been used in [33] and [34], which describe a formal analysis of the Internet Open Trading Protocol (IOTP). Protocol verification is based on the methodology of [9] and aims to prove that the protocol specification satisfies the requirements of its users, as they are described in the so-called protocol's service specification. However, the forenamed protocol verification approach does not aim to prove potential transaction atomicity guarantees as we do, by taking into account site failure, message loss and participants' fraud possibilities.

6. Conclusion

This work's contribution is a systematic approach in the development and validation of high-level CP-net models of electronic payment systems. We proposed the use of four different types of places and an automata-driven model building technique. The developed models are appropriate for model checking all levels of transaction atomicity guarantees, as well as potential protocol-level intrusion attacks. In the obtained model checking results we take into account all cases of site failures, message losses, unilateral transaction aborts and fraudulent participant behavior.

Our experience suggests that when systematic model development is focused in the verification of the considered payment transaction guarantees, it exploits the strengths of Colored Petri Nets and avoids their pitfalls. For a real-scale protocol like NetBill, our approach resulted in a model of manageable size in terms of both its usability within the advanced graphical environment of CPN Tools and its generated state space. Although we were aware that in CPN Tools current representation and storage of the generated states is far from being optimal - compared to other mature tools like SPIN - we noted that a computer with a Pentium IV processor (2.4 GHz) and 500 MB RAM generated the model's state space in only 30 seconds. Also, the graph of the strongly connected components, which is utilized in CTL model checking, was generated in only 2 seconds. In effect, this allowed us to exploit the strengths of CTL in expressing the considered transaction guarantees in the form of state space queries, each of which was answered in a couple of seconds.

The overall approach can be used in studying an important class of open problems in electronic commerce ([45]), like for example the atomicity mechanisms needed in distributed purchase transactions ([46]), the development of "non-blocking" payment systems or systems with semi-trusted or no trusted parties and so on. We referred to some of these challenges, but we also underline the potentiality of our approach for model checking transaction guarantees of different types of systems (contract signing systems, electronic auction systems, orchestrated web services for transactional workflows [35] etc).

Acknowledgments

We acknowledge the CPN Tools team at Aarhus University, Denmark for kindly providing us the license of use of the valuable CP-net toolset. Also, we acknowledge the anonymous referees for their thorough contribution in improving the quality of the present article.

References

[1] Asokan N., Janson, P., Steiner M., Waidner, M. 1997. State of the art in electronic payment systems. *IEEE Computer*, **30** (9): 28-35

[2] Asokan, N. Fairness in Electronic Commerce, PhD Thesis, University of Waterloo, Ontario, Canada, 1998

[3] Al-Azzoni, I., Down, D. G., Khedri, R. 2005. Modeling and verification of cryptographic protocols using Coloured Petri Nets and Design/CPN. *Nordic Journal of Computing*, **12** (3): 200-228

[4] Austry, D., Boudol, G. 1989. Algebre de proessus at synchronization. *Theoretical Computer Science*, **1** (30): 91-131

[5] Basagiannis, S., Katsaros, P., Pombortsis, A. Interlocking control by Distributed Signal Boxes: design and verification with the SPIN model checker. Proceedings of the International Symposium on Parallel and Distributed Processing and Applications (ISPA 2006), Lecture Notes in Computer Science 4330, Springer-Verlag, 2006; 317-328

[6] Basagiannis, S., Katsaros, P., Pombortsis, A. Intrusion attack tactics for the model checking of e-commerce security guarantees. Proceedings of the 26th International Conference on Computer Safety, Reliability and Security (SAFECOMP), Lecture Notes in Computer Science 4680, Springer-Verlag, 2007; 238-251

[7] Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., McKenzie, P. *Systems and Software Verification – Model-Checking Techniques and Tools*, Springer, 2001

[8] Bernardeschi, C., Fantechi, A., Gnesi, S. 2002. Model checking fault tolerant systems. *Software testing, verification and reliability*, Wiley, 12: 251-275

[9] Billington, J., Gallasch, G. E., Han. A Coloured Petri Net approach to protocol verification. In: Lectures on Concurrency and Petri Nets - Advances in Petri Nets, Lecture Notes in Computer Science 3098, Springer-Verlag, 2004; 210-290

[10] Bouali, A., Gnesi, S., Larosa, S. 1994. The integration project for the JACK environment. *Bullentin of the EATCS*, **54**: 207-223.

[11] Cheng, A., Christensen, S., Mortensen, K. H. Model checking Coloured Petri Nets exploiting strongly connected components, Proceedings of the International Workshop on Discrete Event Systems, Edinburgh, Scotland, UK, 1996; 169-177.

[12] Cox, B., Tygar, J. D., Sirbu, M. NetBill security and transaction protocol. Proceedings of the 1st USENIX Workshop in Electronic Commerce, USENIX Association, 1995; 77-88

[13] CPN Tools, http://wiki.daimi.au.dk/cpntools/cpntools.wiki

[14] Deitel, H. M., Deitel, P. J., Nieto, T. R. *e-Business & e-Commerce: How to program*. Prentice Hall, 2001.

[15] Dierks, T., Allen, C. The TLS Protocol, Version 1.0, Network Working Group, IETF 2246, January 1999. On-line: http://www.ietf.org/rfc/rfc2246.txt

[16] Dill, D. The Murφ verification system, Proceedings of the 8th International Conference on Computer Aided Verification, 1996; 390-393.

[17] Ferreira, L. de C., Dahab, R. A scheme for analyzing electronic payment systems, Proceedings of the 14th Annual Computer Security Applications Conference, IEEE Computer Society, 1998; 137-146.

[18] Franklin, M., Reiter, M., Fair exchange with a semi-trusted third party, Proceedings of the 4th ACM Conference on Computer and Communication Security, 1997; 1-6.

[19] Garcia-Fanjul, J., Tuya, J. and Corrales, J. A. Formal verification and simulation of the NetBill protocol using SPIN, Proceedings of the 4th International Workshop on Automata Theoretic Verification with the SPIN Model Checker, ENST, Paris, France, 1998; 195-210

[20] Georgiadis, C. K., Pimenidis, E., Web services enabling virtual enterprise transactions, Proceedings of the IADIS International Conference on e-Commerce, Barcelona, Spain, 2006; 297-302

[21] Heintze, N., Tygar, J., Wing, J., Wong, H., Model checking electronic commerce protocols, Proceedings of the 2nd USENIX Workshop in Electronic Commerce, Oakland, CA, USENIX Association, California, 1996; 146-164

[22] Jensen, K. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Volumes 1-3 (second corrected printing), Monographs in Theoretical Computer Science, Springer-Verlag, 1997

[23] Jensen, K. An introduction to the practical use of Coloured Petri Nets, In: Lectures on Petri Nets II: Applications, ed.: W. Reisig, G. Rozenberg, LNCS 1492, Springer, 1996; 237-292

[24] Katsaros, P., Odontidis, V., Gousidou-Koutita, M. Colored Petri Net based model checking and failure analysis for E-commerce protocols, Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'05), DAIMI PB-576, Dept. of Computer Science, University of Aarhus, Denmark, 2005; 267-283

[25] Ketchpel S., Garcia-Molina, H. Making trust explicit in distributed commerce transactions. Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-96), IEEE Computer Society Press, 1996; 270–281

[26] Kempster, T., Stirling, C., Thanisch, P. A critical analysis of the Transaction Internet Protocol, Proceedings of the 2nd International Conference on Telecommunications and Electronic Commerce (ICTEC), Nashville, TN, USA, 1999

[27] Lacoste, G., Pfitzmann, B., Steiner, M., Waidner, M. *SEMPER: Secure Electronic Marketplace for Europe*. Lecture Notes in Computer Science 1854, Springer-Verlag, 2000

[28] Lyon, J., Evans, K., Klein, J. Transaction Internet Protocol, Version 3.0, Network Working Group, IETF 2371, July 1998. On-line: http://www.ietf.org/rfc/rfc2371.txt

[29] Mu, Y., Nguyen, K. Q., Varadharajan, V. A fair electronic cash scheme. Proceedings of ISEC 2001, Lecture Notes in Computer Science 2040, Springer-Verlag, 2001; 20-32

[30] Nenadic, A., Zhang, N., Barton, S. A security protocol for certified e-goods delivery. Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 04), IEEE Computer Society, 2004.

[31] De Nicola, R., Vaandrager, FW. Actions versus state based logics for transition systems, Proceedings de l' Ecole de Printemps on Semantics of Concurrency, Lecture Notes in Computer Science 469, Springer-Verlag, 1990; 407-419.

[32] O' Mahony, D., Peirce, M., Tewari, H. *Electronic payment systems for e-commerce* (Second Edition). Artech House, 2001.

[33] Ouyang, C., Kristensen, L. M., Billington, J. A formal and executable specification of the Internet Open Trading Protocol, Proceedings of EC-Web 2002, Lecture Notes in Computer Science 2455, Springer-Verlag, 2002; 377-387

[34] Ouyang, C., Billington, J. An improved formal specification of the Internet Open Trading Protocol, Proceedings of the 2004 ACM Symposium on Applied Computing, Nicosia, Cyprus, 2004; 779-783

[35] Papazoglou, M. P. 2003. Web services and Business Transactions. *World Wide Web: Internet and Web Information Systems*, 6: 49-91

[36] Pfitzmann B., Waidner M. Properties of payment systems – General definition sketch and classification. *Research Report RZ 2823 (#90126), IBM Research Division,* May 1996

[37] Pombortsis, A., Tsoulfas, A. *An introduction to Electronic Commerce* (In Greek). Tziolas Publishers. 2002.

[38] Ray, I., Ray, I., Natarajan, N. 2005. An anonymous and failure resilient fair-exchange e-commerce protocol. *Decision Support Systems*, 39: 267-292

[39] Roscoe, A. W. *The theory and practice of concurrency*. Prentice-Hall (Pearson). 2005.

[40] Roy, V., De Simone, R. AUTO and Autograph. Proceedings of the Workshop on Computer Aided Verification, Lecture Notes in Computer Science 531, Springer-Verlag, 1990; 65-75.

[41] Schuldt, H., Popovici, A., Schek, H.-J. Automatic generation of reliable e-commerce payment processes. Proceedings of the First International Conference on Web Information Systems Engineering (WISE 00), Vol. 1, IEEE Computer Society, 2000; 434-441

[42] Shmatikov, V., Mitchell, J. C. 2002. Finite-state analysis of two contract signing protocols. *Theoretical Computer Science*, 283 (2): 419-450

[43] Shyamasundar, R. K., Deshmukh, B. MicroBill: An efficient secure system for subscription based services. Proceedings of ASIAN 2002, Lecture Notes in Computer Science 2550, Springer-Verlag, 2002; 220-232

[44] Su, J., Tygar, J. D. Building blocks for atomicity in electronic commerece, Proceedings of the Sixth USENIX UNIX Security Symposium, USENIX Association, San Jose, California, 1996;

[45] Tygar, J. D. Atomicity in electronic commerce. 1998. Atomicity in electronic commerce. *netWorker*, ACM Press, **2** (2): 32-43

[46] Wang, G., Das, A. Models and protocol structures for software agent based complex e-commerce transactions, Proceedings of EC-Web 2001, Lecture Notes in Computer Science 2115, Springer-Verlag, 2001; 121-131

[47] Wong, H. L. Protecting individuals' interests in Electronic Commerce Protocols, PhD Thesis, CMU-CS-00-160, School of Computer Science. Carnegie Mellon University, Pittsburgh, 2000

[48] Xu, S., Yung, M., Zhang, G., Zhu, H. Money conservation via atomicity in fair off-line e-cash, Proceedings of the 2nd International Workshop of Information Security, Lecture Notes in Computer Science 1729, Springer-Verlag, 1999; 14-31

Appendix A

In CP-nets the states are represented by means of *places* (which are drawn as ellipses). By convention we write the names of the places inside the ellipses. Each place has an associated *data type* determining the kind of data that the place may contain (by convention the type information is written in italics, next to the place). The type declarations implicitly specify the operations that can be performed on the values of the types. A state of a CP-net is called a *marking* and consists of a number of *tokens* positioned on the individual places. Each token carries a data value, which belongs to the type of the corresponding place.

A marking of a CP-net is a function, which maps each place into a *multi-set of tokens* of the correct type. We refer to the token values as *token colors* and to their data types as *color sets*. The types of a CP-net can be arbitrarily complex, e.g., a record where one field is a real, another a text string and a third a list of integers.

The actions of a CP-net are represented by means of *transitions*, which are drawn as rectangles. An incoming arc indicates that the transition may remove tokens from the corresponding place while an outgoing arc indicates that the transition may add tokens. The exact number of tokens and their data values are determined by the *arc expressions*, which are positioned next to the arcs. Arc expressions may contain variables as well as constants. To talk about the *occurrence of a transition*, we need to bind incoming expressions to values from their corresponding types. Let us assume that we bind the incoming variable v of some transition T to the value d. The pair $(T, <v = d>)$ is called *binding element* and this binding element is *enabled* in a marking M, when there are enough tokens in its input places. In a marking M, it is possible to enable more than one binding elements of T. If the binding element $(T, <v = d>)$ occurs, it removes tokens from its input places and adds tokens to its output places. In addition to the arc expressions, it is possible to attach a boolean expression with variables to each transition. The boolean expression is called a

guard and specifies that we only accept binding elements, for which the boolean expression evaluates to true.