Formally validated specification of a micro-payment protocol

P. Dargenton[1], D. Hirschkoff[1], P. Lescanne[1] and E. Pommateau[2]

[1] LIP, École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon Cedex 7 – France
[2] NTSys, 64 chemin des mouilles, F-69130 Ecully – France *and* CEDRIC, Conservatoire National des Arts et Métiers, 292 rue Saint Martin, F-75141 Paris Cedex 03 – France

Abstract: In this paper, we develop a formal specification for a micro-payment protocol, first on paper, then within the Coq proof assistant. Our approach in defining a notion of *execution traces* for protocol runs is inspired by previous works, notably by L. Paulson (in the Isabelle/HOL system). However, we show that the protocol we study entails some modifications to Paulson's framework, related to the modeling of the agents' internal state. We moreover take profit from Coq's expressive meta-language to mechanically derive proofs *about* the formalisation itself, by introducing a notion of well-formedness for protocol rules.

Keywords: electronic commerce protocol, protocol specification, theorem proving, Coq proof assistant

This work has been supported by *"Appel d'offres OPPIDUM – Offre de Procédés et de ProduIts De sécUrisation pour la Mise en œuvre des autoroutes de l'information"*, conventions N° 00.2.93.0466.

Formally validated specification of a micro-payment protocol

P. Dargenton[1], D. Hirschkoff[1], P. Lescanne[1], and É. Pommateau[2]

[1] École Normale Supérieure de Lyon – France
[2] NTSys and CNAM – France

Abstract. In this paper, we develop a formal specification for a micro-payment protocol, first on paper, then within the Coq proof assistant. Our approach in defining a notion of *execution traces* for protocol runs is inspired by previous works, notably by L. Paulson (in the Isabelle/HOL system). However, we show that the protocol we study entails some modifications to Paulson's framework, related to the modeling of the agents' internal state. We moreover take profit from Coq's expressive meta-language to mechanically derive proofs *about* the formalisation itself, by introducing a notion of well-formedness for protocol rules.

1 Introduction

The formal analysis of cryptographic protocols has grown into an important research thread. Various methods have been proposed to check mechanically various kinds of properties of various kinds of protocols. Recently, the development of e-commerce has further encouraged the study of formal methods for security, either through the adaptation of already existing approaches or through the design of new techniques.

The work at hand is the result of a collaboration between an academical laboratory and a startup, called NTSys, interested in developing e-commerce technology. We focus more precisely in this paper on a *micro-payment protocol*, referred to as the *"light signatures protocol"*. Micro-payment protocols are used in situations where a client is liable to buy a big amount of goods, each of these having a small cost. Typically, one can think of an internet service, the *content seller*, providing information in such a way that each mouse click in a certain set of web pages has a (small) price. Traditional, so-called "heavy" cryptography (e.g. using RSA or elliptic curves) cannot be used for each such a micro-contract, as it would dramatically slow down the transactions. Alternative solutions have then to be found, based on a trade-off between efficiency of the protocol and security matters, which are of course always crucial in the context of electronic commerce. The light signatures protocol[1] has been introduced to adress this question. The basic ideas of the protocol have been presented in [Opp01a,PBM01], together

[1] The use of light signatures in micro-payment systems is patended by NTSys; light signatures are based on an original idea by Jacky Montiel.

with a–rather informal–description of its behaviour. The general micro-payment system is structured in such a way that a *confidence party*, whose role is to arbitrate transactions between several content sellers and Internet end-users, aggregates the micro-transactions and monitors the contracts. In this infrastructure, the light signatures protocol is a client/server security protocol intented to be run between a content seller and the confidence party. Its goal is to provide authentication for the online control and registration of the transactions.

The formal definition of the light signatures protocol has been developed progressively, towards and increasing level of precision in the description of the protocol steps. What we present here is the result of this process: we describe the formal specification of the protocol, and establish results about this specification, but we do not address the security properties satisfied by the protocol itself (such as, for example, authentication or properties related to information flow during protocol execution). This is left for future work. It has actually turned out that, already in setting up a formal analysis of the original version of the protocol, many security issues could be raised, and small mistakes or imprecisions could be fixed.

In this paper, we describe the resulting protocol in several steps, from a rather informal account of the interactions between parties to a more mathematical presentation of the protocol, and finally to a mechanical definition. For this last step, we have used the Coq proof assistant [Bar01], which is an interactive theorem prover based on the Calculus of (Co)Inductive Constructions.

In moving towards mechanical verification as it is the case in this work, we isolate some difficult points in the protocol that we want to understand and validate, and leave aside some others (either by keeping them outside the representation framework or by "underspecifying" them). This is frequently the case when performing an analysis like ours. What is important, though, is to keep track of the simplifying assumptions that are made, so as to be aware of "what is actually being proved", and, in a further step, to characterise the attacks against which the protocol has been proved to be robust.

It may also be the case that the specification process itself introduces some unexpected behaviour of the protocol: for example, a given simplification in the protocol can have the effect of allowing more interactions between agents than were actually possible in the original, more intricate, version. One is therefore interested in reasoning *about* the specification under construction, to establish properties that are more related to the specification itself than to the system being specified. Another motivation for this kind of reasoning comes from our experience in designing the description of the protocol: as the presentation moved towards a very formal account, some of the people involved in this work, who were less skilled in formal methods, wanted to be sure that we were "still talking about the same thing". If we have a way to prove some properties of the specification saying that the formal entities that we manipulate actually do what they are supposed to do, we can increase the confidence in the adequacy of the specification under constuction. We have developed this kind of reasoning within our Coq specification of the light signatures protocol, in order to prove mechanically that the rules of the protocol as they are stated in Coq satisfy a kind of *well-formedness* (a notion that we define below).

Contributions The contributions of this work are twofold. First, we present the formal specification of a new micro-payment protocol, described as a set of traces. Second, we propose a novel approach for the mechanical reasoning *about the specification itself*, that can be combined (in the Coq system) with Paulson's inductive approach for the study of traces and of possible interferences with evil agents.

Related works Paulson's work [Pau98] has been a major influence in introducing a notion of traces for our protocol. However, we use Coq instead of Isabelle/HOL [Pau94b,Pau93], which will have some consequences on our formalisation, as will be seen in Sec. 4. It has to be noted that the work of Bolignano [Bol96] has led to the definition of a theorem proving framework for the verification of cryptographic protocols in Coq. Our approach however is closer to Paulson's in the way we formalise traces.

Outside the theorem proving community, different techniques have been proposed for the formal study of cryptographic protocols. [Bas99] presents an automated method to handle inductively defined traces which are reminiscent of Paulson's work. We can also mention approaches based on model checking [Low97,Mea96], on term rewriting techniques [JRV00], or, even somewhat closer to our method, on typed multiset rewriting [CDL+99], and also on process algebra descriptions of protocols [AL00,AF01,AG99,Bor01].

Outline of the paper The paper is structured towards an increasingly formal understanding of the micro-payment protocol we study. We first introduce its principles in Section 2. In Section 3, we turn to a more mathematical presentation, by defining a notion of traces generated by protocol runs. We then turn to the Coq mechanisation, by describing the specification of the protocol in Section 4 and the proofs about our specification in Section 5. In Section 6, we conclude and comment on some important originalities of the approach we have followed, especially in the methodology we use for the formalisation of the micro-payment protocol.

Parts of this work have been presented in [Opp01b]. The Coq development corresponding to the formalisation discussed in Sections 4 and 5 is available at `http://www.ens-lyon.fr/~hirschko/oppidum`

2 The Micro-Payment Protocol: Informal Description

In this section, we present the light signatures protocol, as introduced in [PBM01], and discuss the simplifying assumptions that we make for the sake of our formal study.

2.1 Preliminaries

The light signatures protocol specifies the transactions between a *client* and a confidence party that plays the role of a *server*. Its general principle is as follows: first the client generates a seed α, and sends α to the server. Heavy cryptography (typically, RSA) is used for this transaction. Once both agents know the seed, they can prepare for the protocol by computing a sequence of $2N$ nonces, where the integer N is a bound on the number of transactions between client and server in the current session. The nonces are computed by applying a one-way hash function H to α, generating the sequence

$$\alpha, H(\alpha), H^2(\alpha), H^3(\alpha), \ldots$$

Thus the ith nonce N_i is equal to $H^{i-1}(\alpha)$.

Remark 1. H being a (one-way) hash function, it is easy to compute N_{i+1} from N_i, and we suppose it is much more difficult the other way around. This is the base idea which is used in the transactions that take place once initialisation is done and both agents know the N_is.

Once the generation of the nonces sequence is completed on both sides, the agents start exchanging messages of the following form:

$$\langle Ag, k, \mathsf{Sign}(C, k), C \rangle \,,$$

where Ag is the identifier of the agent (client or server), C is the content being transmitted (a query of the client or an ackowledgment of the server), k is an integer computed from the agent's current *session index* (to be described below), and $\mathsf{Sign}(\_,\_)$ is a *signature function* that is used to authenticate the message. We set

$$\mathsf{Sign}(C, k) \stackrel{def}{=} H'(C, N_{2N-1-k}) \,,$$

where H' is a hash function (possibly taken equal to H).

Along the execution of the protocol, each agent maintains an internal index that is incremented after each query/answer; this index allows both parties to synchronise and to detect strange behaviours in transactions. By definition, the client sends messages of the form $\langle Clt, 2 * Ind_{Clt}, \mathsf{Sign}(Q_{Ind_{Clt}}, Ind_{Clt}), Q_{Ind_{Clt}} \rangle$, where Ind_{Clt} is the current value of the client's index. Symmetrically, the server answers using nonces corresponding to odd integers computed from its current internal index Ind_{Srv}, and sending acknowledgments of the form $A_{Ind_{Clt}}$. It can be noted that by definition of the signature function, the nonces will be used in reverse order, according to Remark 1 above. Note as well that the usage we make of hash chains is reminiscent of Lamport's one-time passwords.

2.2 Description of the Protocol

Let us now describe the protocol in a more formal way. We adopt the usual notation of the form $A \to B : M$ to say that at a given step of the protocol, agent A sends message M to agent B.

• ***Initialisation and first message*** the client computes a fresh seed α and sends it using public key cryptography to the server:

$$(Clt_0)\;\; Clt \to Srv \,:\, \langle Clt, 0, \mathsf{Sign}(\{\{\alpha, Q_0\}_{S_{Clt}}\}_{P_{Srv}}, 0), Q_0\rangle\,.$$

S_{Clt} and P_{Srv} are respectively the client's private key and the server's public key. If K is a key, $\{M\}_K$ denotes the result of encrypting information M with K. Private/public keys of every agent are supposed to be the inverse of each other in the encryption/decryption process. Once both agents know the seed α, they compute the sequence of nonces N_is and store it locally (in a protected area). Note that the very first message of the protocol has a somehow special form, because, together with the first request Q_0, the client also has to send the seed α. Therefore, public key cryptography is used to encrypt the pair $\{\alpha, Q_0\}$, using the client's shared key S_{Clt} and the server's private key P_{Srv}.

A different initialisation step is actually described in [PBM01], where the first message is $\langle Clt, 0, \{\{\alpha\}_{S_{Clt}}\}_{P_{Srv}}, \mathsf{Sign}(Q_0, 0), Q_0\rangle$. The modification we have brought here is mainly due to technical reasons, in order for all messages to be 4-uples, which will simplify the formal study. However, there does not seem to be any reason to fear a security loss in encrypting the seed together with Q_0 instead of treating both contents separately as is done in [PBM01].

• ***Client's requests*** the requests of the client–represented as the Q_is–are sent in messages corresponding to even indices of the nonces:

$$(Clt_1)\;\; Clt \to Srv \,:\, \langle Clt, 2 * Ind_{Clt}, \mathsf{Sign}(Q_{Ind_{Clt}}, 2 * Ind_{Clt}), Q_{Ind_{Clt}}\rangle\,.$$

• ***Server's responses*** symmetrically, answers of the server–the A_is–are sent with odd nonce indices:

$$(Srv_1)\; Srv \to Clt :\, \langle Srv, 2 * Ind_{Srv} + 1, \mathsf{Sign}(A_{Ind_{Srv}}, 2 * Ind_{Srv} + 1), A_{Ind_{Srv}}\rangle.$$

The last query/answer exchange in a normal run of the protocol happens when rules (Clt_1) and (Srv_1) are used with $Ind = N - 1$, and has the effect of concluding the session.

• ***Desynchronisation on the client's side*** rule (Clt_1) above specifies the emission of client's requests during a normal execution of the protocol. In particular, before sending the $(i + 1)$th request, the client makes sure that the last message received from the server has been generated with the right index (i.e. i). Let us now examine the case where the index Ind_{Srv} in the last message received by the client does not correspond to its "view" of the protocol, as defined by index Ind_{Clt}.

There are two cases: either the last message from the client is such that $Ind_{Srv} < Ind_{Clt}$: this probably corresponds to an old message from the server, that has been accidentally resent to the client. In that case, the client simply ignores it (or may decide to send a warning message to the server, in an enriched version of this protocol). If $Ind_{Srv} > Ind_{Clt}$ (and if the message from the server has the right shape), the situation is much more suspect: in some way, either an evil agent has managed to construct messages to be sent "in the future", or for some reason the server erroneously believes that more transactions have taken place than is actually the case. The best thing to do here is to abort the protocol session: this is done by sending a message built with the last client integer (namely $2N$) and pointing the fact that an error has occured:

$$(Clt_2) \;\; Clt \rightarrow Srv \; : \; \langle Clt, 2N-2, \mathsf{Sign}(Error, 2N-2), Error \rangle \, .$$

$Error$ is a special message to indicate a misbehaviour.

• ***Desynchronisation on the server's side*** A similar situation may occur on the server's side, with a received client index Ind_{Clt} that does not correspond to the server's internal index Ind_{Srv}. If $Ind_{Clt} < Ind_{Srv}$, this is probably an old message, ignore it. If $Ind_{Clt} > Ind_{Srv}$, then it may be the case that some messages from the client were lost; the server resynchronises with the client by setting its current session index to Ind_{Clt} (and may emit a warning message).

$$(Srv_2) \;\; Srv \rightarrow Clt \; : \; \langle Srv, 2*Ind_{Clt}+1, \mathsf{Sign}(R, 2*Ind_{Clt}+1), A_{Ind_{Srv}} \rangle \, .$$

Note that at every step, a transaction has to be initiated by the client and acknowledged by the server: this is the reason why the reaction of the agents differs whenever they discover that the other agent seems to be "ahead of time" in the protocol.

• ***Time outs*** If the client fails to receive an acknowledgment from the server for a given amount of time (to be fixed in practice), he generates a time out, and sends his last message to the server again. After a given amount of time outs, the client decides to abort the protocol session, by sending the same message as in rule (Clt_2) above. Therefore, in addition to his current session index, the client also has to keep track of the number of time outs that have been occurred in order to be able to decide to abort the transactions.

On the server side, the time out mechanism is somehow simpler: after receiving no query from the client for a certain amount of time, the last nonce is sent along with an error message to inform the client that the protocol is aborted. This corresponds to the following step:

$$Srv \rightarrow Clt \; : \; \langle Srv, 2N-1, \mathsf{Sign}(Err, 2N-1), Err \rangle \, .$$

2.3 Discussion – Simplifications Made for the Formal Study

We have given a first description of the light signatures protocol, which is rather close to the presentation of [PBM01]. With respect to that work, we have already introduced a few small refinements in some protocol steps, but many aspects are still unprecise. For example, it is not clear when agents index modifications have to occur, or how the session starts on the server's side. One of the contributions of the formal description we shall present in the next section is to give a more detailed treatment of these issues.

We first discuss here some of the aspets of the light signatures protocol that are not handled in our formal study. As said before, it is important to be aware of the simplifications we make for the sake of our analysis, in order to understand the limitations of the specification (this issue has been adressed for example when applying methods based on formalisms such as the BAN logic [BAN90]).

The first issue that we do not address here is *time*. The overall duration of a session of the protocol, as well as the value of time outs on the client's and the server's sides, are to be fixed by practical experiments on an actual implementation of the protocol. Note however that the shape of the transition rules we shall present in the next section determines (temporal) causality relations between message emissions.

We do not describe either the mechanisms that are used to take into account several contemporaneous sessions, though the handling of *freshness* of the seed α. Such issues are rather common in cryptographic protocols, and we believe that traditional techniques can be used for this purpose.

Finally, as said above, we do not handle warning messages in the current description of the protocol. These could be included in a future, more detailed, specification of the light signatures.

3 A Notion of Trace

In this section, following Paulson's approach, we inductively define a set of traces generated by possible protocol runs. Informally, the idea is to represent the traffic on the network due to the protocol, and describe the messages that can be sent by the various agents, these messages being possibly intercepted by an evil agent. Agents which obey the protocol react to the presence of messages of a certain form on the network by emitting new messages, and so on until the protocol completes. Of course, at any point in the execution, some agent watching the traffic on the network can choose to pick a message and try to decode it, or to resend either previously emitted messages or newly generated ones. This is captured by the *Spy*'s behaviour in Paulson's framework [Pau98]. In the present study, we do not represent the spy and its possible attacks yet, but rather focus on the construction of protocol traces. The formalisation we obtain can be enriched following the lines in [Pau98] to handle attacks.

According to this approach, a typical rule for the inductive construction of protocol traces would state something like

> *"if a message of shape M is present on the network, then add a message M' to the current traffic"*,

this behaviour corresponding to some step in the protocol where an agent B replies to A's message M by sending message M'. We shall see below how this kind of construction can be defined in a formal way.

3.1 Internal State of Agents

An important specificity of the protocol we study is that both the client's and the server's behaviours depend on an internal representation of the current status of the protocol session (session index and number of time outs for the client, session index for the server). This seems to be in contrast with the methodology described above, where the traces are generated by adopting a *global* point of view on message traffic. Indeed, we have modified the framework to take into account a global notion of *state*, to represent the information mantained by each agent. The informal description of each protocol step given above is then refined into something like

> *"if a message of shape M is present on the network and the current state is E, then set state to be E' and emit message M'".*

Of course, this way of presenting the protocol rules is much too general, for at least two reasons. First, the message M' that will be emitted will clearly depend most of the time on the shape of M. Second, the possible modification to the global state will heavily rely on the agent which is supposed to react at this step of the protocol, and in particular, we wish to be sure that the agent may modify the values of *"its"* components of state, leaving e.g. the index of the other agent unchanged.

It turns out indeed that adopting a very general approach like we do here simplifies the task of the formalisation: trying to separate the global structure representing the state of each agent into several pieces, to take into account the private character of state, would probably lead to more complex notations and representation mechanisms. On the other hand, one has to make sure that the flexibility we gain is not misused, for example by allowing an agent to modify an other agent's private values: this is the issue we shall address in Section 5.

3.2 Generating Traces

Let us now move to the formal definition of traces. Every step of the protocol (except initialisation, see below) is described by a rule of the form

$$\frac{E \;\;\&\;\; M}{\boxed{\leadsto}\, M' \;\;;\;\; E'}$$

This judgment means that whenever the system is in state E and M belongs to the current trace, *first* message M' is emitted (added to the trace), *then* the system evolves to state E'. It is important to notice that state modification takes place after emission of the message, so that one can tell the value of indices possibly used in the content of the message.

The rules that define the traces generated by protocol runs are given on Figure 1. They correspond to a precise formalisation of the description given in Sec. 2. The correspondence is rather easy for rules c_1, c_2, s_1 and s_2, which embed rules Clt_1, Clt_2, Srv_1 and Srv_2 respectively. Note however that in this presentation the update of internal indices has been made explicit. As far as state representation is concerned, we use an underscore (_) to denote those state components that are not useful for the statement of a rule (and hence remain unchanged). The rules for the initialisation of the protocol on both sides, namely c_0 and s_0, specify how the agents start a new session and set their indices to 0. In particular, rule c_0 has no premiss, as the client can decide at any time to start a protocol session.

With respect to the presentation of [PBM01], we have added rule s_0 to model the protocol initialisation on the server's side: this allows us to state explicitely client

$$c_0 \quad \boxed{\leadsto} \quad \{\!\{ Clt,\ 0,\ \mathsf{Sign}(\aleph, 0),\ \aleph \}\!\} \ \ ; \ \ \langle\!\langle 0\,,\ 0 \parallel \_ \rangle\!\rangle$$

$$\text{where } \aleph \ = \ \{\{\alpha, Q_0\}_{SClt}\}_{PSrv}$$

$$c_1 \quad \frac{\langle\!\langle c\,,\ \textsc{to}_C \parallel \_ \rangle\!\rangle \ \ \&\ \ \{\!\{ Srv,\ 2c+1,\ \mathsf{Sign}(A_c, 2c+1),\ A_c \}\!\}}{\boxed{\leadsto}\ \{\!\{ Clt,\ 2(c+1),\ \mathsf{Sign}(Q_{c+1}, 2(c+1)),\ Q_{c+1} \}\!\} \ \ ; \ \ \langle\!\langle c+1\,,\ 0 \parallel \_ \rangle\!\rangle} \quad c < N$$

$$c_2 \quad \frac{\langle\!\langle c\,,\ \textsc{to}_C \parallel \_ \rangle\!\rangle \ \ \&\ \ \{\!\{ Srv,\ 2s+1,\ \mathsf{Sign}(A_s, 2s+1),\ A_s \}\!\}}{\boxed{\leadsto}\ \{\!\{ Clt,\ 2N-2,\ \mathsf{Sign}(Err, 2N-2),\ Err \}\!\} \ \ ; \ \ \langle\!\langle 0\,,\ 0 \parallel \_ \rangle\!\rangle} \quad c < s-1$$

server

$$s_0 \quad \frac{\begin{array}{c}\langle\!\langle \_ \parallel s \rangle\!\rangle \ \ \&\ \ \{\!\{ Clt,\ 0,\ \aleph,\ \mathsf{Sign}(\aleph, 0) \}\!\} \\ \text{where } \aleph \ = \ \{\{\alpha, Q_0\}_{SClt}\}_{PSrv}\end{array}}{\boxed{\leadsto}\ \{\!\{ Srv,\ 1,\ \mathsf{Sign}(A_0, 1),\ A_0 \}\!\} \ \ ; \ \ \langle\!\langle \_,\ \_ \parallel 0 \rangle\!\rangle} \quad \alpha \ \ \textbf{fresh}$$

$$s_1 \quad \frac{\langle\!\langle \_ \parallel s \rangle\!\rangle \ \ \&\ \ \{\!\{ Clt,\ 2s,\ \mathsf{Sign}(Q_s, 2s),\ Q_s \}\!\}}{\boxed{\leadsto}\ \{\!\{ Srv,\ 2s+1,\ \mathsf{Sign}(A_s, 2s+1),\ A_s \}\!\} \ \ ; \ \ \langle\!\langle \_,\ \_ \parallel s+1 \rangle\!\rangle} \quad c = s$$

$$s_2 \quad \frac{\langle\!\langle \_ \parallel s \rangle\!\rangle \ \ \&\ \ \{\!\{ Clt,\ 2c,\ \mathsf{Sign}(Q_c, 2c),\ Q_c \}\!\}}{\boxed{\leadsto}\ \{\!\{ Srv,\ 2c+1,\ \mathsf{Sign}(A_c, 2c+1),\ A_c \}\!\} \ \ ; \ \ \langle\!\langle \_,\ \_ \parallel c+1 \rangle\!\rangle} \quad c > s$$

handling time outs

$$\textsc{to}_{c1} \quad \frac{\langle\!\langle c\,,\ \textsc{to}_C \parallel \_ \rangle\!\rangle \ \ \&\ \ \_\_}{\boxed{\leadsto}\ \{\!\{ Clt,\ 2c,\ \mathsf{Sign}(Q_c, 2c),\ Q_c \}\!\} \ \ ; \ \ \langle\!\langle c\,,\ \textsc{to}_C + 1 \parallel \_ \rangle\!\rangle} \quad \textsc{to}_C < \textsc{to}_{C\,max}$$

$$\textsc{to}_{c2} \quad \frac{\langle\!\langle c\,,\ \textsc{to}_C \parallel \_ \rangle\!\rangle \ \ \&\ \ \_\_}{\boxed{\leadsto}\ \{\!\{ Clt,\ 2N-2,\ \mathsf{Sign}(Err, 2N-2),\ Err \}\!\} \ \ ; \ \ \langle\!\langle 0\,,\ 0 \parallel \_ \rangle\!\rangle} \quad \textsc{to}_C \geq \textsc{to}_{C\,max}$$

$$\textsc{to}_{s} \quad \frac{\langle\!\langle \_ \parallel s \rangle\!\rangle \ \ \&\ \ \_\_}{\boxed{\leadsto}\ \{\!\{ Srv,\ 2N-1,\ \mathsf{Sign}(Err, 2N-1),\ Err \}\!\} \ \ ; \ \ \langle\!\langle \_,\ \_ \parallel 0 \rangle\!\rangle}$$

Fig. 1. Definition of traces

at which point the session index of the server is set to 0, and to make precise the freshness condition on the seed α. Indeed, without this freshness condition, any evil agent could keep resending the client's first message, which would have the effect of restarting the protocol on the server's side, thus bringing a form of denial of service.

The rules for the description of time outs are also new with respect to [PBM01]: they allow us to take time outs into account in our description of the protocol without explicitely giving a notion of *time* along protocol runs. Indeed, we instead introduce a form of non-determinism in the description of the protocol, which makes it possible to model any possible run of the protocol (possibly interleaved with server or client time-outs), thus freeing us from the necessity of handling a notion of time. Rules TO_{c1} and TO_{c2} model time outs on the client's side, while rule TO_{s} defines the server's time outs.

4 The Protocol in Coq

4.1 The Coq Proof Assistant

For the implementation of our specification, we have adopted the Coq proof assistant [Bar01]. This system is an theorem prover based on the Calculus of (Co)Inductive Constructions, that allows the user to define theories and interactively build proofs about the objects that have been introduced. One of the main original features of Coq with respect to Isabelle is the presence of *dependent types* [CH88], which we exploit in the present work. Indeed, while on one hand one could that Isabelle/HOL provides a more powerful language of tactics, Coq's metalanguage comes with greater expressiveness, which shall be exploited below to reason on our formalisation within the prover.

Coq's syntax is somehow reminiscent of a programming language à la ML, with the following notations and definitions:

- We shall work with Coq's `Set` and `Prop` kinds. While `Prop` is the kind for (non-informative) proof objects, `Set` is used to build constructions whose structure can be analysed. This will be useful below.
- Product types (resp. abstractions) are written `(x:T)T'` (resp (`[x:T]T'`). When `T'` expresses a property or a logical statement, `(x:T)T'` may be read *"for all x of type T, T' holds"*.
- Inductive types are declared using the `Inductive` keyword, by providing the type of each constructor. After such a definition, the corresponding case-analysis and elimination principles are automatically computed by Coq.

These informal explanations should be enough to follow the code excerpts that are provided in the following paragraphs. The reader interested in more details should refer to the documentation of the system [Bar01].

4.2 Representing the Entities Involved in the Protocol

Figure 2 presents the Coq code for the specification of the data structures we need to specify the micro-payment protocol. Let us paraphrase these definitions:

```
Inductive agent : Set := Clt : agent | Srv : agent.

Parameter seed : Set.

Inductive content : Set :=
   A : nat -> content | Q : nat -> content | Err : content
 | Seed : seed -> content -> content.

Inductive signed : Set := Sign : content -> nat -> signed.

Inductive message : Set :=
  msg : agent -> nat -> signed -> content -> message.

Inductive state : Set :=
  st : nat -> nat -> nat -> state.

(* an axiomatisation of a set constructor *)

Parameter set : Set -> Set.
Parameter in_set : (S:Set)(set S) -> S -> Prop.
Parameter add_set : (S:Set)S -> (set S) -> (set S).
Parameter empty_set : (S:Set)(set S).
Definition mk_set := [S:Set][x:S](add_set S x (empty_set S)).
Definition incl_set := [S:Set][s1,s2:(set S)]
   (x:S)(in_set S s1 x) -> (in_set S s2 x).
(* a very reasonable property relating in_set and add_set *)
Parameter in_add :
   (S:Set)(s:(set S))(x:S)(in_set S (add_set S x s) x).

Definition trace := (set message).
```

Fig. 2. Coq definitions for the entities involved in the protocol

- We represent two agents, the client and the server (according to Paulson's methodology, the spy shall come into the play afterwards).
- Message bodies can either consist in client's queries $(Q_1, Q_2, \dots)$, server's answers $(A_1, A_2, \dots)$, or error messages.
- The specification of the signature function boils down to a tuple definition. Its properties can be postulated independently.
- Current state is given by a quadruple of integers.
- The definition of the trace relies on a straightforward axiomatisation of a set constructor enjoying elementary properties. This part of the specification is left unspecified for the moment, as it does not influence the results we are interested in here.

4.3 Implementing Traces

The rules given at Figure 1 can be translated rather directly into a Coq definition of an inductive object `micro` of type `state -> t:trace -> Set`. Intuitively, a term of type `(micro e t)` means that there exists an execution of the protocol leading to state `e` and generating trace `t`.

Here is an example, giving the type associated to the constructor `c1` of the inductive type `micro`:

```
c1 : (e:state)(t:trace)(micro e t) ->
  (c:nat)
  (in_set message t (msg Srv (S (mult (2) c))
                  (Sign (A c) (S (mult (2) c))) (A c)))
  /\ (state_c e)=c
  -> (micro (inc_c e) (add_set message
          (msg Clt (S (mult (2) c))
                   (Sign (Q (S c)) (S (mult (2) c))) (Q (S c)))   t))
```

As we can see, modulo some syntactical conversions (`mult` is Coq's multiplication on natural numbers, `S` is the successor function, `/\` is conjunction, and `state_c` and `inc_c` are obvious functions to manipulate state), we basically recognise rule c_1 from Figure 1. Keeping rules close to their formulation on paper is good practice because it gives confidence that no hidden anomaly or extra hypothesis is added in the process of porting the specification into the machine. Of course, this is possible in our case because the rules we use to describe traces are already very formal. Such rules were obviously not the kind of tools we have been using at early stages of the protocol formalisation. It is useful to go through a step where we have a definition as precise as possible on paper before going on the machine, in order for the person(s) in charge of the mechanical specification to take as few *unnoticed* design decisions as possible as far as the protocol is concerned (we indeed proceeded this way in the present work).

Limitations of the specification For the moment, we have not taken into account the freshness condition on the seed α, neither have we formalised rules $\textsc{to}_{c1}$, $\textsc{to}_{c2}$ and $\textsc{to}_s$ in Coq. Defining the properties of α should be done at some point when verifying the protocol, exactly like specifying the behaviour of the signature function `Sign`. As we have said above, this is not our purpose yet, as we are rather interested in examining the kind of traces that are generated when executing our micro-payment protocol. We believe though that providing a notion of freshness for seeds within our specification should not be a problematic issue. As far as leaving rules for the managment of time outs outside our specification, we think that it can be good practice to first study the protocol without time outs, to check its behaviour in "normal conditions", and *then* add time outs into the game. Moreover, the specification of the protocol we work with is smaller without time outs, which is helpful in terms of clarity of our study. Here again, we do not think that formalising the extra rules to handle time outs would cause any difficulty in the framework we are presenting.

5 Reasoning about the Encoding of the Protocol

5.1 State Integrity

As has been noted in paragraph 3.1, the framework we use to formalise the protocol rules and their effect on internal states of agents is rather general, and may a priori make it possible to represent quite weird behaviours. One problem is due to the notion of global state where every information about every agent's state can be represented in formulating a rule.

Well-behaved protocols nevertheless should be designed in such a way that every agent only has the possibility to modify its own component of the global state (at least–one could also think of situations where the internal state of other agents should not be accessible, not only in writing, but also in reading). We refer to this property as *state integrity*, to express the fact that the protocol rules indeed respect this assumption.

One could imagine adressing this kind of property by using an adequate formalism. For example, Alur and Henzinger's reactive modules [AH96] introduce a partition of a module's parameters into *controlled* and *external* variables: in our case, this would mean showing that all state variables are controlled. However, when doing a machine-assisted verification, one may want to "bring into the prover" as much information about the system being analysed as possible. In the following, we define a framework to capture within Coq a notion of *well-formedness* for protocol rules, which basically says that the interaction described by a given rule respects state integrity.

5.2 Well-Formedness of the Protocol Specification

In order to reason on the evolutions of global state, we define a function extracting from a trace object a pair of successive states during the execution of the protocol. This function, called `next_state`, is of type

```
state -> (e:state) (t:trace)(micro e t) -> Prop.
```

It is defined by case analysis on the structure of its fourth argument, namely an hypothesis `H` of type `(micro e t)`. Using implicit arguments (a mechanism available in Coq to simplify notations by allowing the user to omit redundant parameters in function calls), we can reason on terms of type `(next_state e0 H)` (`H` being of type `(micro e t)`); the existence of such a term represents the fact that the protocol makes state `e0` evolve into `e`.

Analogously, we can introduce a function `added_message`, extracting from an hypothesis `H` of type `(micro e t)` the last message added in the execution of the protocol so far. Functions `next_state` and `added_message` are then used to state and prove the following theorem, establishing well-formedness of the constructors of the inductive type `micro`:

```
Theorem wf_rules : (e,e':state)(t:trace)
  (H:(micro e' t)) (next_state e H)
   ->
     (* either the client is sending, and s is invariant ... *)
   ( (state_s e)=(state_s e') /\ (msg_sender (added_message H))=Clt ) \/
     (* ... or the server is sending, and c is invariant.    *)
   ( (state_c e)=(state_c e') /\ (msg_sender (added_message H))=Srv ).
```

Theorem `wf_rules` expresses the fact that in any step in the execution of the protocol, only the internal state of the sender of the last message being emitted may have changed in the last protocol step. In other words, this means that no agent can modify other agents' state. Once the functions for the analysis of protocol traces have been defined, this result is easily proved by case analysis on the shape of the object of type `(micro e' t)`.

5.3 Exploiting Dependent Types to Derive Proofs about the Specification

In the paragraphs above, we have been able to state and prove some properties about objects of type `micro` *within* the Coq system. To achieve this, we have defined ways to analyse the structure of an hypothesis `H:(micro e t)` in order to extract the information we needed for our proofs. This has been possible in a rather natural way by exploiting Coq's setting where proofs are terms (and propositions are types), these terms possibly having dependent types (like `(micro e t)`). These features are specific to Coq w.r.t. Isabelle/HOL, so we believe that the proofs we have derived could not be directly adapted to the setting of Paulson.

Remark 2 (About `Set` and `Prop`). We have defined `micro` as a predicate having values in `Set` because we were interested in analysing the structure of a trace. Indeed, Coq's distinction between kinds `Prop` and `Set` introduces a form of asymmetry between types carrying a constructive information, which are in `Set`, and types in `Prop`, that are seen as "comments", to be erased when performing program extraction. As a consequence, it is not possible to eliminate an object in `Prop` to construct something in `Set`, which is exactly what we wanted to do when defining function `next_state` above. Therefore, we have been led to define `micro` in `Set`, so as to have all elimination principles we need to mechanically derive proofs about the shape of protocol execution traces.

6 Conclusion

In this paper, we have presented the formal specification of an original micro-payment protocol, and implemented its specification in the Coq proof assistant. In doing this, we have modified Paulson's approach by introducing a notion of internal state of the agents, and we have established some well-formedness properties of the encoding w.r.t. the novelty of state managment.

The fact that we have to manipulate internal knowledge of the agents along the execution of the protocol is not due to the kind of protocol we have examined, but rather to the design choices that have been made in defining the protocol. However, this specificity has been smoothly integrated into the framework used by Paulson, which demonstrates the flexibility of the inductive approach for the verification of cryptographic protocols.

Using Coq instead of Isabelle/HOL has made it possible to reason about the formalisation within the prover rather easily. The theorem we have proved in paragraph 5.2 above captures a property of the constructors of the inductive type we use to represent traces. As we have seen, this comes quite naturally in Coq, using dependent types. It would be interesting to study how this kind of proofs could be formalised in Isabelle/HOL. It could be the case that to achieve this, the user would in some way have to deal with the technicalities involved in the object-level development of inductive types in Isabelle/HOL [Pau94a]. It is no surprise, anyway, that proofs about the object-level formalisation bring to light the rather strong differences between Coq and Isabelle's underlying frameworks. Still, the general approach defined by Paulson can be adopted without much differences in both systems.

We have found it important to define a mechanism to guarantee that the managment of state is done in a safe way in the rules of our protocol, mostly because this feature is original w.r.t. previous works. It would be interesting, more generally, to find out whether other properties about the representation of the protocol could be formalised and verified within the prover, so as to be able not only to reason about the protocol, but also about its representation. It is indeed common knowledge that many important decisions in the formal proof activity are taken when specifying the entities one shall reason about: on one hand, oversimplifying the representation of a problem can make its verification easy (but questionable as far as the meaning of the resulting proof is concerned), and on the other hand, specifying too many aspects of a problem may prevent any formal proof to be completed. Tools to study the specification itself can help in finding a compromise in this context.

Acknowledgments Comments from the referees have been useful both to improve the presentation and to relate the present work with other existing approaches.

References

[AF01] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. of POPL'01*. ACM Press, 2001.

[AG99] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[AH96] R. Alur and T. Henzinger. Reactive modules. In *proc. of 11th LICS*, pages 207–218, 1996.

[AL00] R. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In *International Conference on Concurrency Theory*, pages 380–394, 2000.

[BAN90] M. Burrows, M. Abadi, and R. Needham. A Logic for Authentication. *ACM transactions on Computer Systems*, 8(1):18–36, 1990.

[Bar01] B. Barras et al. The Coq proof assistant, version 7.0. INRIA, 2001. available at `http://coq.inria.fr/`.

[Bas99] D. Basin. Lazy infinite-state analysis of security protocols. In *proc. of CQRE'99*, volume 1740 of *LNCS*, pages 30–42. Springer Verlag, 1999.

[Bol96] D. Bolignano. An approach to the formal verification of cryptographic protocols. In *Third ACM Conference on Computer and Communications Security*, pages 106–118. ACM Press, 1996.

[Bor01] M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proc. of ICALP'01*. Springer Verlag, 2001.

[CDL+99] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *proc. of CSFW'99*, pages 55–69. IEEE Computer Society Press, 1999.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

[Jei00] M. Jeilton. The cost of carambars. *Prospects in Net-Economy*, 2:13–27, 2000.

[JRV00] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Verifying Security Protocols. In *Proc. of LPAR'2000*, volume 1955 of *LNCS*. Springer Verlag, 2000.

[Low97] G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. of the 10th CSFW*, pages 18–30. IEEE Computer Society Press, 1997.

[Mea96] C. Meadows. Analyzing the Needham-Schroeder public-key protocol: A comparison of two approaches. In *Proc. of Computer Security – ESORICS'96*, volume 1146 of *LNCS*, pages 351–364. Springer Verlag, 1996.

[Opp01a] "Spécification du protocole des signatures légères", 2001. Appel d'offres OPPIDUM - délivrable E1-2.

[Opp01b] "Validation formelle", 2001. Appel d'offres OPPIDUM - délivrable E1-3.

[Pau93] L. C. Paulson. Isabelle's object-logics. Technical Report 286, University of Cambridge, Computer Laboratory, 1993.

[Pau94a] L. C. Paulson. A fixed point approach to implementing (co)inductive definitions. In *Proc. of CADE'94*, number 814 in LNAI, pages 148–161. Springer Verlag, 1994.

[Pau94b] L. C. Paulson, editor. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer Verlag, 1994.

[Pau98] L. C. Paulson. The inductive approach to verifying crypotgraphic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[PBM01] E. Pommateau, F. Bellaiche, and J. Montiel. The Light Signatures Protocol. In *Proc. of IC'01*, 2001.