Defining Authentication in a Trace Model

C.J.F. Cremers[1], S. Mauw[1], and E.P. de Vink[1,2]

[1] Eindhoven University of Technology
Department of Mathematics and Computer Science
P.O. Box 513, NL-5600 MB Eindhoven, the Netherlands
{ccremers,sjouke,evink}@win.tue.nl
[2] LIACS, Leiden University, the Netherlands

Abstract. In this paper we define a general trace model for security protocols which allows to reason about various formal definitions of authentication. In the model, we define a strong form of authentication which we call *synchronization*. We present both an injective and a non-injective version. We relate synchronization to a formulation of agreement in our trace model and contribute to the discussion on intensional vs. extensional specifications.

1 Introduction

Security protocols have become an established application area of formal methods today. Over the past years various modelling languages, logics and process algebras have been proposed for the systematic and tool supported analysis of security protocols. Exploitation of formal methods in this setting leads to a better understanding of implicit assumptions and gives feedback on what the protocol does or, in many cases, what it unexpectedly does not achieve.

In this paper we focus on the notion of authentication as a so-called intensional security property. We formulate a variant of authentication, called synchronization. In [10] Roscoe introduces the notion of canonical intensional specification, stating that all parties involved in a protocol, after completion of their role, are convinced that for their part the protocol has been executed according to its rules. Intensional properties are those induced by the form or structure of the protocol; extensional properties are related to the effect the protocol achieves. The notion of synchronization proposed here captures Roscoe's notion of canonical intensional specification as a general trace property. By casting intensionality in the same framework as extensional security properties, intensionality can be compared with and, as we shall see, related to so-called full injective agreement.

Important to this point of view is the observed behavior from the perspective of an individual agent. It will make no difference to an agent engaged in the protocol if a sequence of communications was the result of its interaction with an honest principal, with a malicious intruder controlling the network, or a mix of these. If the agent is entitled to believe at a point in time that the interaction thus

far is consistent with the agents role in the protocol, the agent simply proceeds under the assumption that all parties involved have obeyed the protocol rules.

In [8] Lowe studies, building on earlier work of [1] and [4], four different forms of authentication, viz. aliveness, weak agreement, full non-injective agreement and full injective agreement. On top of this, agreement on subsets of data items and recentness is considered (two topics we do not address here). By casting the notions in the setting of CSP Lowe shows that the notions constitute an ascending chain of authentication principles. Additionally, the relationship with Roscoe's definition of intensional specification is discussed at the conceptual level. The notion of injective synchronization that is proposed in this paper as intensional authentication can be shown to be strictly stronger than full injective agreement and thus, can be added at the top of Lowe's authentication hierarchy. This extension constitutes the body of the present paper.

Authentication and agreement is also studied in [3] by Focardi and Martinelli in the context of the so-called GNDC scheme. In a process algebra extended with inference systems reflecting cryptographic actions, one can reduce reasoning about security properties with respect to any arbitrary environment to the analysis of the behavior of the security protocol in the most general environment. It is argued that the GNDC scheme is valid for establishing various security properties, in particular agreement (as well as its weaker variants).

In Section 2 below we gather the machinery required for our description of security protocols. The notions of non-injective and injective synchronization are the main topic of Section 3. The thread of reasoning is continued for the case of agreement. In Section 4 agreement is reformulated in our setting and compared with its definition in [8]. We further present a taxonomy result for the introduced notions of injective and non-injective synchronization and agreement, and provide examples to show the strictness of the implied relationships. Section 5 then closes off with concluding remarks.

Acknowledgment We are grateful to Jerry den Hartog for his useful suggestions at various points.

2 Trace model

In this section we will define the notions which are essential for the definition of synchronisation as elaborated in Section 3. Synchronisation is based on the property that every successful execution of a protocol by an agent implies that its communication partners exactly follow their roles in the protocol and exchange the intended message in the intended order. Therefore, we first provide a formal definition of a security protocol and the partial order implied on its events.

The second step is the introduction of the trace model for asynchronously communicating agents. Synchronisation is verified as a property on the set of execution traces. It can be defined independently of the way in which the trace model is constructed for a given security protocol, a given set of agents and a given intruder model. Therefore, we will not find a need to introduce a formal semantics precisely defining the trace model. Rather, we expect that many of

the existing formal approaches towards security protocols can be molded in such a way as to produce the required trace model.

2.1 Security protocols

A security protocol defines the interacting behaviour of a number of agents. Agents can take part in one or more runs of a protocol by performing a role defined in the protocol.

In most approaches, a protocol description consists of a list of messages exchanged by some principals. In contrast to this approach we take as a starting point the projection of this behaviour onto the different roles. Thus, we split a communication into two separate events, viz. a send event and the corresponding read event. These two corresponding events are described in two (different) role definitions. We elaborate on the implications of this choice later.

We assume a finite set of role names, called *Role*. The behaviour of a role is defined by a role definition (*RoleDef*). A role definition is simply a list of role events (taken from the set *RoleEvent*). In the following, we will define send, read, and claim events.

Roles exchange message from the set *RoleMess* of role messages, by executing the events $send(r, r', m)$ and $read(r, r', m)$. The send event is executed by role r, which intends to send message m to role r'. The read event is executed by role r' and indicates the reception of message m, apparently sent by role r.

Because it is allowed that the same message occurs more than once in a role definition (having the same sender and the same recipient), we will need to disambiguate these events by extending them with labels from the finite label set *Label*. Therefore, we have role events $send_\ell(r, r', m)$ and $read_\ell(r, r', m)$ for a label $\ell \in Label$. We require that all events in a security protocol have distinct labels, except for corresponding send and read events (from different roles), which share a common label.

These labels also serve a second purpose. Due to our decision to split communications into separate send and read events, we lost the intended correspondence between these events. We use the event labels to keep this correspondence information available. If a send event from one role definition shares a label with a read event from another role definition this expresses that these are corresponding events. We define the (partial) functions *sendrole* and *readrole* to determine for a given label the sending role and the receiving role, respectively.

Our treatment of security claims is somewhat different from other approaches. Rather than considering correctness of a security protocol as a property of the protocol as a whole, we consider a more refined approach where claims are local to each role. For this purpose, we introduce claim events.

A claim event has the form $claim(r, c)$, where r is the claiming role and c is the claim, taken from a set of claims *Claim*. An example of such an event is $claim(r, alive(r'))$. This means that if an agent executing role r has executed his part of the protocol up to the claim event, he can be sure that the agent executing role r' was alive. In Sections 3 and 4 we will give the definitions of four claims: *ni-synch*, *i-synch*, *ni-agree* and *i-agree*.

The fact that security claims are local to a role is a consequence of our approach to split communication events into separate send and read events and to consider role definitions as a basic concept.

Finally, we define a security protocol $p \in \mathit{SecProt}$ as a collection of role definitions, or rather as a function from Role to $\mathit{RoleDef}$. In summary, we have the following definitions.

$$\begin{aligned}
\mathit{SecProt} &= \mathit{Role} \to \mathit{RoleDef} \\
\mathit{RoleDef} &= \mathit{RoleEvent}^* \\
\mathit{RoleEvent} &\supseteq \{\mathit{send}_\ell(r, r', m), \mathit{read}_\ell(r, r', m), \mathit{claim}_\ell(r, c) \mid \\
&\qquad \ell \in \mathit{Label}, r, r' \in \mathit{Role}, m \in \mathit{RoleMess}, c \in \mathit{Claim}\} \\
\mathit{sendrole}(\ell) &= r \quad \text{if } \mathit{send}_\ell(r, r', m) \in p(r) \text{ or } \mathit{read}_\ell(r, r', m) \in p(r') \\
\mathit{readrole}(\ell) &= r' \quad \text{if } \mathit{send}_\ell(r, r', m) \in p(r) \text{ or } \mathit{read}_\ell(r, r', m) \in p(r')
\end{aligned}$$

Please notice that we did not provide an exhaustive definition of $\mathit{RoleEvent}$. We only require inclusion of the given events. The reason is that these events suffice for our definition of authentication. Nevertheless, a formal semantics could consider more role events, such as assignments to local variables.

The authentication property which will be defined in Section 3 expresses that the message exchanges between the agents take place exactly in the order as implied by the security protocol. In order to express this, we define the causality relation on the events in a security protocol as a partial order. Since security protocols are often visualised as Message Sequence Charts (MSCs, see [5]), it comes as no surprise that the partial order semantics for MSCs can be used to express the order of events in a security protocol. This partial order simply expresses that the events in a role definition are sequentially ordered, and that every read event is preceded by its corresponding send event. Adopting notation from [2], we start with defining the role order $\prec_r$ for $r \in \mathit{Role}$ as the total order on its events. Event e causally precedes event e' (notation $e \prec_r e'$) if e occurs before e' in the role definition of r. Next, we define the send-before-read order $\prec^{sr}$ which expresses that a send causally precedes the corresponding read (i.e. $\mathit{send}_\ell \prec^{sr} \mathit{read}_\ell$). Finally, we define the causality preorder $\prec_p$ of protocol p as the transitive closure (denoted by $^+$) of the union of all role orders and the send-before-read order, i.e.

$$\prec_p = (\bigcup_{r \in \mathit{Role}} \prec_r \ \cup \prec^{sr})^+$$

Example The well-known Needham-Schroeder-Lowe protocol (NSL) (cf. [9, 6]) can be depicted by the MSC in Figure 1. The added *i-synch* claims at the end of the roles will be defined in Section 3.

The NSL protocol has two roles, viz. one of the initiator I and one for the responder R. The role definitions for the initiator and responder are

$$\begin{aligned}
&\mathit{send}_1(I, R, \{I, n_I\}_{pk_R}) \cdot \mathit{read}_2(R, I, \{R, n_I, n_R\}_{pk_I}) \cdot \mathit{send}_3(I, R, \{n_R\}_{pk_R}) \cdot \mathit{claim}_4(I, \textit{i-synch}) \\
&\mathit{read}_1(I, R, \{I, n_I\}_{pk_R}) \cdot \mathit{send}_2(R, I, \{R, n_I, n_R\}_{pk_I}) \cdot \mathit{read}_3(I, R, \{n_R\}_{pk_R}) \cdot \mathit{claim}_5(R, \textit{i-synch})
\end{aligned}$$

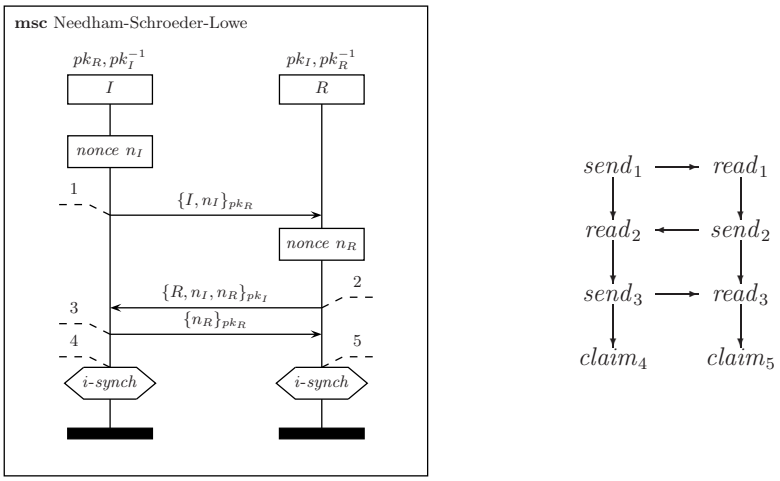

Fig. 1. The NSL protocol and the partial ordering on its events.

The indices of sends, reads and claims refer to the labels listed aside of the roles. On top of the roles, the initial knowledge of the agents is listed. It is assumed that both agents know the public key of the other and the private key of their own. The generation of nonces n_I and n_R is visualized in the *action boxes*. The hexagons contain the claims made by the agents. In fact, these are the security goals the protocol is supposed to achieve. The causality preorder $\prec_p$ for NSL is given by the lattice on the right in Figure 1.

2.2 Traces

Now that we have introduced security protocols, we can discuss the execution of a security protocol (or rather a role in a security protocol) by an agent. We start by defining the set of agents *Agent*. A role executed by an agent is called a run. Whereas we can consider a run as an instantiated role, we can consider run events (*RunEvent*) as instantiated role events. This instantiation simply amounts to replacing abstract role names by concrete agent names. Since the concrete messages sent in a run may be different from the abstract messages specified in a role (e.g. because of the instantiation of role names by agent names), we introduce the set of run messages *RunMess*. An agent can execute several roles (possibly the same) in an interleaved way. Although it is not necessary for the definitions below, a proper semantics will require that every (honest) agent executes the events exactly as prescribed by its role definition.

Because we need to be able to distinguish the events from different runs (possibly stemming from the same role), we need to add information to disambiguate run events. This is done by assigning a unique run identifier from the set *RunId* to every run and extending the events of every run with their run identifier. An event e extended with run identifier rid is denoted by $e\sharp rid$. Thus, we have the

following run events.

$$RunEvent \supseteq \{send_\ell(a, a', m)\sharp rid, read_\ell(a, a', m)\sharp rid, claim_\ell(a, c)\sharp rid \mid rid \in RunId, \ell \in Label, a, a' \in Agent, m \in RunMess, c \in Claim\}$$

Each of these events is executed by some agent, performing some role in the protocol. The role of the actor of event e is denoted by $role(e)$. It can be derived easily from the label of the event (which is unique in the protocol specification) taking into account if it is a send or read event, or a non-communication event.

The final step is to introduce the traces of a security protocol, induced by some semantic definition (e.g. assuming operational behaviour of agents and an intruder model). The semantics of security protocol p is denoted by $T(p)$. We assume that it is defined as a collection of sequences of run events.

$$T(p) \in \mathcal{P}(RunEvent^{\leq\omega})$$

We denote the i-th event of trace $\alpha \in T(p)$ by α_i (for $i \geq 0$). If we require a property of a security protocol p, we will, in our set-up, have to check this property for all of the traces in $T(p)$.

Thus, the semantics of a security protocol is expressed as a set of traces and each such trace is an interleaving of a number of runs. An agent can execute many such runs in an interleaved way and every run may be an instantiation of each of the roles in the protocol definition. We assume $T(p)$ to contain, for example, traces corresponding to the double instantiation of p where agent a is running both role r and r', and agents b and c are running roles r' and r, respectively. But $T(p)$ will also contain traces for the situation where role r is taken up by agents a and c and role r' by agents b and d with a talking to b and c to d. More concretely, in the case of NSL described above, $T(NSL)$ contains all the shuffles of the following four traces induced by the instantiation just given (and many more).

$$send_1(a, b, \{a, n_a^1\}_{pk_b})\sharp 1 \cdot read_2(b, a, \{b, n_a^1, n_b^2\}_{pk_a})\sharp 1 \cdot send_3(a, b, \{n_b^2\}_{pk_b})\sharp 1 \cdot claim_4(a, i\text{-}synch)\sharp 1$$
$$read_1(a, b, \{a, n_a^1\}_{pk_b})\sharp 2 \cdot send_2(b, a, \{b, n_a^1, n_b^2\}_{pk_a})\sharp 2 \cdot read_3(a, b, \{n_b^2\}_{pk_b})\sharp 2 \cdot claim_5(b, i\text{-}synch)\sharp 2$$
$$send_1(c, a, \{c, n_c^3\}_{pk_a})\sharp 3 \cdot read_2(a, c, \{a, n_c^3, n_a^4\}_{pk_c})\sharp 3 \cdot send_3(c, a, \{n_a^4\}_{pk_a})\sharp 3 \cdot claim_4(c, i\text{-}synch)\sharp 3$$
$$read_1(c, a, \{c, n_c^3\}_{pk_a})\sharp 4 \cdot send_2(a, c, \{a, n_c^3, n_a^4\}_{pk_c})\sharp 4 \cdot read_3(c, a, \{n_a^4\}_{pk_a})\sharp 4 \cdot claim_5(a, i\text{-}synch)\sharp 4$$

Please note that the send and read events that make up these traces are really executed by the running agents. So, the occurrence of an event $read_\ell(r, r', m)$ in trace α denotes that in this scenario α agent r' really receives message m, which might or might not have been sent by agent r. This means that the read and send events are not intruder events. Nevertheless, the order and contents of these events may be under control of an intruder, as specified in a formal semantic definition. Due to the fact that the read and send events are executed by the agents (performing some role in the protocol) it is clear that the extension of these events with labels and run identifiers is by no means under control of the intruder. The label (like a program counter) simply indicates the state of the agent when executing his part of the protocol. The run identifier expresses in which run of the agent activity takes place.

Although it is not common to include labels and run identifiers in the semantics of a security protocol, they are often left implicit or occur when developing tool support for a particular semantics. For an example of such use of labels and run identifiers see [7]. Error traces of a protocol are displayed as follows: "$Msg\ \alpha.1.\ I_A \rightarrow B : A, B;\ Msg\ \alpha.2.\ B \rightarrow I_A : B, N_b;\ Msg\ \beta.1.\ I_A \rightarrow B : A, N_b$". Clearly, α and β represent run identifiers and the numbers $1, 2$, etc. (the program counters in the role) represent labels. Making this information explicit allows us to formally reason about these notions.

3 Synchronisation

In this section we formally define the notion of synchronization based on the trace model introduced in Section 2. As is the case with other forms of authentication, we will make a distinction between non-injective synchronisation, and the somewhat stronger notion of injective synchronisation, which rules out a class of replay attacks.

3.1 Non-injective Synchronisation

For the purpose of a modular presentation, we present the definition of non-injective synchronisation in three steps. The first step is the definition of authentication for a single message, or rather, for a single label shared by two corresponding send and read events. For this we define the predicate *1L-SYNCH* (for 1 label synchronisation). It has five arguments: the trace α being validated up to index k, the label ℓ for which we check the authentication property, and the two runs rid_1 and rid_2 containing the read and send events that must be validated. This basic authentication property simply amounts to checking whether the send labeled with ℓ from the first run is followed by the corresponding read from the second run. These corresponding send and read events should exactly agree upon the the sender, the recipient, and the contents of the message.

Definition 3.1 For all traces α, $k \in N$, labels ℓ and run identifiers rid_1, rid_2, the single-label synchronization predicate *1L-SYNCH* is given by

$$\begin{array}{l} \textit{1L-SYNCH}(\alpha, k, \ell, rid_1, rid_2) \iff \\ \quad \exists_{i,j \in N, a,b \in Agent, m \in RunMess} \\ \qquad i < j < k \wedge \alpha_i = send_\ell(a, b, m) \sharp rid_1 \wedge \alpha_j = read_\ell(a, b, m) \sharp rid_2 \end{array}$$

If $\textit{1L-SYNCH}(\alpha, k, \ell, rid_1, rid_2)$ holds, we say that the communication labeled with ℓ, sent by run rid_1 and read by run rid_2 has occurred correctly in a trace α before position k.

The second step towards the definition of synchronisation is to extend the *1L-SYNCH* predicate to range over multiple communications, given by a set of labels. This predicate is denoted by *ML-SYNCH* (for multi-label synchronisation).

When dealing with multiple communications, we require consistency over roles. For example, if an initiator receives two messages from a responder during a run, we require that they are sent from the same responder in the same run. The requirement that roles from a protocol are consistently mapped to runs leads to the introduction of an instantiation function $inst : Role \rightarrow RunId$. This explains the four parameters of the *ML-SYNCH* predicate: α and k give the part of the trace that is to be checked, L is the set of labels for each of which we check the *1L-SYNCH* property, and *inst* is the instantiation function which determines a run for each role definition from the protocol. The definition of *ML-SYNCH* consists of the requirement that for every label from L the predicate *1L-SYNCH* holds. In order to determine for a given label the sending role and the receiving role, we use the auxiliary functions *sendrole* and *readrole*, respectively, which have been introduced in Section 2.

Definition 3.2 For all traces α, $k \in N$, label set L and instantiation *inst*, the multi-label synchronization predicate *ML-SYNCH* is given by

$$\begin{aligned} &\mathit{ML\text{-}SYNCH}(\alpha, k, L, \mathit{inst}) \iff \\ &\quad \forall_{\ell \in L}\ \mathit{1L\text{-}SYNCH}(\alpha, k, \ell, \mathit{inst}(\mathit{sendrole}(\ell)), \mathit{inst}(\mathit{readrole}(\ell)))\ ) \end{aligned}$$

If $\mathit{ML\text{-}SYNCH}(\alpha, k, L, \mathit{inst})$ holds, we say that the set of labels L has correctly occurred in a trace α before position k with respect to the instantiation *inst*.

For the third and final step towards the definition of synchronisation we need to fix the set of labels that should be checked and we must quantify over the α, k and *inst* parameters of *ML-SYNCH*.

It is tempting to take for L the set of all labels occurring in the protocol. However, as a consequence of our choice to have security claims attached to individual roles, rather than to a protocol as a whole, this would result in too strong a predicate.

For any (local) synchronisation claim, we can at most require that all communications that causally precede the claim, have occurred correctly. This is because the claiming role may only expect these events to have been executed. Since a communication consists of two events, we must make it more precise when a communication precedes a claim. Clearly, it must be the case that if a read event precedes the claim, the corresponding send event precedes as well. However, in general we cannot expect the opposite. It may be the case that a send (causally) occurs before a claim, while the corresponding read does not. This observation leads to the definition of the set of labels that causally precede a synchronisation claim.

Definition 3.3 The set $\mathit{prec}(p, cl)$ of causally preceding communications of a claim role event labeled with cl, for a security protocol p is given by

$$\mathit{prec}(p, cl) = \{\ell \mid \mathit{read}_\ell(\_, \_, \_) \prec_p \mathit{claim}_{cl}(\_, \_)\}$$

Section 2, and Figure 1. Note that for the NSL protocol, in agreement with the discussion in Section 2, we have that $prec(NSL, 4) \neq prec(NSL, 5)$. In particular, $read_3(I, R, \{n_R\}_{pk_R}) \prec_p claim_5(R, ni\text{-}synch)$, but $read_3(I, R, \{n_R\}_{pk_R}) \not\prec_p claim_4(R, ni\text{-}synch)$, as can be seen in Figure 1.

Finally, we define the synchronisation predicate $NI\text{-}SYNCH$. It states that if we encounter a synchronisation claim $claim_\ell(a, ni\text{-}synch)\sharp rid$ as an event in any of the traces α of a security protocol, the communications causally preceding this claim must be well behaved. With respect to the instantiation function $inst$, we can simply require the existence of any such function, mapping roles to run identifiers, with the restriction that it maps the role of the claiming agent to the run containing this claim (i.e. $inst(role(\alpha_k)) = rid$, where α_k is the claim event).

Definition 3.4 For all security protocols p, the synchronisation predicate $NI\text{-}SYNCH$ for claims labeled with ℓ, is given by

$$
\begin{array}{l}
NI\text{-}SYNCH(p, \ell) \iff \\
\quad \forall_{\alpha \in T(p), k \in N, a \in Agent, rid \in RunId}\ \alpha_k = claim_\ell(a, ni\text{-}synch)\sharp rid \Rightarrow \\
\qquad \exists_{inst:Role \to RunId}\ inst(role(\alpha_k)) = rid \wedge ML\text{-}SYNCH(\alpha, k, prec(p, \ell), inst)
\end{array}
$$

It expresses that for all instantiated claims in any trace of a given security protocol, there exist runs for the other roles in the protocol, such that all communications preceding the claim must have occurred correctly within these runs.

3.2 Injective synchronisation

Synchronisation guarantees that a single protocol role run has executed as expected. For each other role in the protocol, there exists a run that has sent and read messages according to the protocol. However, this property does not rule out a particular type of replay attacks. If we consider, for instance, a simple two-party protocol, it can be the case that every run of the initiator neatly matches the same run of the responder. Such a protocol could be abused by an intruder replaying an old responder run for every new session. (See Figure 3 in Section 4.2 for a more detailed discussion.) This weakness of the protocol could easily be detected by requiring that there is an injective relation between the claiming run and the other runs. This notion of injectivity has been defined for several authentication properties (see e.g. [8]). Here we will define it for synchronisation.

The definition of injective synchronisation proceeds in two steps. The first step is to bring the definition of non-injective synchronisation into a form that allows us to easily add the injectivity criterion. Note that in Definition 3.4 the relation between the claiming run (with run identifier rid) and the runs playing the other roles (as expressed by the instantiation function $inst$) is not made explicit. If we want to require that this relation is injective, we must first formulate an explicit version of this definition. The functional dependence of the instantiations $inst$ on the claiming run rid will be expressed by a function

$Inst : RunId \times Role \rightarrow RunId$. We can easily extract the instantiation functions from $Inst$ by taking $inst = \lambda r.Inst(rid, r)$, i.e. the bindings that are relevant for the run identifier rid. These observations yield the following rewrite of Definition 3.4.

$$\begin{array}{l} \mathit{NI\text{-}SYNCH}(p,\ell) \iff \\ \quad \forall_{\alpha \in T(p)} \exists_{Inst:RunId \times Role \rightarrow RunId} \forall_{k \in N, a \in Agent, rid \in RunId} \\ \qquad \alpha_k = claim_\ell(a, ni\text{-}synch) \sharp rid \Rightarrow Inst(rid, role(\alpha_k)) = rid \\ \qquad\quad \wedge\ \mathit{ML\text{-}SYNCH}(\alpha, k, prec(p,\ell), \lambda r.Inst(rid, r)) \end{array}$$

We can now define injective synchronisation in almost the same way as the second formulation of *NI-SYNCH*, by requiring that the function *Inst* in the definition is injective.

Definition 3.5 For all security protocols p, the injective synchronisation predicate
I-SYNCH for claims labeled with ℓ, is given by

$$\begin{array}{l} \mathit{I\text{-}SYNCH}(p,\ell) \iff \\ \quad \forall_{\alpha \in T(p)} \exists_{Inst:RunId \times Role \rightarrow RunId \text{ injective}} \forall_{k \in N, a \in Agent, rid \in RunId} \\ \qquad \alpha_k = claim_\ell(a, i\text{-}synch) \sharp rid \Rightarrow Inst(rid, role(\alpha_k)) = rid \\ \qquad\quad \wedge\ \mathit{ML\text{-}SYNCH}(\alpha, k, prec(p,\ell), \lambda r.Inst(rid, r)) \end{array}$$

It expresses that for all instantiated claims, there exist runs for the other roles in the protocol. All communications preceding the claim must have occurred correctly within these runs. Furthermore, for each such claim there are unique runs executing the roles in the protocol.

4 Agreement

In this section we formalize a form of authentication called agreement. This concept is weaker than synchronisation, as it places less restrictions on the ordering of the occurring events. We first give formal definitions. We will argue that these definitions correspond to a form of *agreement* as described by Lowe in [8]. Next, we present an extension of the authentication hierarchy and provide examples showing the strictness of the various relationships.

4.1 Non-injective and injective agreement

We construct the definition of agreement analogously to synchronisation. Synchronisation required that all sends occur before their corresponding reads. The definition of *1L-AGREE* does not require that the send occurs before the corresponding read.

Definition 4.1 For all traces α, $k \in N$, labels ℓ and run identifiers rid_1, rid_2, the single-label agreement predicate *1L-AGREE* is given by

$$\begin{array}{l} \mathit{1L\text{-}AGREE}(\alpha, k, \ell, rid_1, rid_2) \iff \\ \quad \exists_{i,j \in N, a,b \in Agent, m \in RunMess} \\ \qquad i < k \wedge j < k \wedge \alpha_i{=}send_\ell(a,b,m) \sharp rid_1 \wedge \alpha_j{=}read_\ell(a,b,m) \sharp rid_2 \end{array}$$

If $\mathit{1L\text{-}AGREE}(\alpha, k, \ell, rid_1, rid_2)$ holds we say that the communication labeled with ℓ, sent by run rid_1 and read by run rid_2, is agreed upon in trace α before position k.

There are no requirements on the order of the read and send. We observe that if *1L-AGREE* holds for some communication, all variables sent as part of the message m are received exactly as expected. Therefore, both parties will agree over the values of the variables that are sent. Further definitions are constructed as with synchronisation. First we define a notion of multi-label agreement similar to *ML-SYNCH*.

Definition 4.2 For all traces α, $k \in N$, set of labels L and run identifiers rid_1, rid_2, the multi-label agreement predicate *ML-AGREE* is given by

$$\begin{array}{l}\mathit{ML\text{-}AGREE}(\alpha, k, L, \mathit{inst}) \iff \\ \quad \forall_{\ell \in L}\ \mathit{1L\text{-}AGREE}(\alpha, k, \ell, \mathit{inst}(\mathit{sendrole}(\ell)), \mathit{inst}(\mathit{readrole}(\ell)))\end{array}$$

If $\mathit{ML\text{-}AGREE}(\alpha, k, L, \mathit{inst})$ holds we say that the set of labels L is agreed upon in trace α before position k, for instantiation function *inst*.

Next we define non-injective agreement.

Definition 4.3 For all security protocols p, the agreement predicate *NI-AGREE* for claims labeled with ℓ, is given by

$$\begin{array}{l}\mathit{NI\text{-}AGREE}(p, \ell) \iff \\ \quad \forall_{\alpha \in T(p), k \in N, a \in \mathit{Agent}, \mathit{rid} \in \mathit{RunId}} \\ \qquad \alpha_k = \mathit{claim}_\ell(a, \mathit{ni\text{-}agree}) \sharp \mathit{rid} \Rightarrow \exists_{\mathit{inst}:\mathit{Role} \to \mathit{RunId}}\ \mathit{inst}(\mathit{role}(\alpha_k)) = \mathit{rid} \\ \qquad\quad \wedge\ \mathit{ML\text{-}AGREE}(\alpha, k, \mathit{prec}(p, \ell), \mathit{inst})\end{array}$$

The agreement predicate expresses that for all instantiated claims in any trace of a given security protocol, there exist runs for the other roles in the protocol, such that all communication events causally preceding the claim must have occurred before the claim.

Injective agreement is defined in the same way as injective synchronisation.

Definition 4.4 [*I-AGREE*] For all security protocols p, the injective agreement predicate *I-AGREE* for claims labeled with ℓ, is given by

$$\begin{array}{l}\mathit{I\text{-}AGREE}(p, \ell) \iff \\ \quad \forall_{\alpha \in T(p)}\ \exists_{\mathit{Inst}:\mathit{RunId} \times \mathit{Role} \to \mathit{RunId}\ \text{injective}}\ \forall_{k \in N, a \in \mathit{Agent}, \mathit{rid} \in \mathit{RunId}} \\ \qquad \alpha_k = \mathit{claim}_\ell(a, \mathit{i\text{-}agree}) \sharp \mathit{rid} \Rightarrow \mathit{Inst}(\mathit{rid}, \mathit{role}(\alpha_k)) = \mathit{rid} \\ \qquad\quad \wedge\ \mathit{ML\text{-}AGREE}(\alpha, k, \mathit{prec}(p, \ell), \lambda r.\mathit{Inst}(\mathit{rid}, r))\end{array}$$

It expresses that for any trace and for any run of any role in the protocol there exist unique runs for the other roles of the protocol such that for all claims occurring in the trace all communications preceding the claim must have occurred correctly within these runs.

In [8], Lowe defines several forms of authentication. The strongest form of authentication, not involving time, is called *full (injective) agreement*:

> Initiator I is in agreement with responder R, whenever I as initiator completes a run of the protocol with R, then R as responder has been running the protocol with I. Moreover, I and R agree on all data variables, and each run of I corresponds to a unique run of R.

We will argue that for any protocol for which an *I-AGREE* claim at the end of a role holds, full (injective) agreement holds with respect to all other roles and data items, and vice versa. Note that we will only consider the case where agreement is claimed over all roles involved in the protocol.

The main difference between our approach and that of Lowe, is that we consider abstract messages only. The definition of full injective agreement does not depend on the actual messages that are passed through. Instead, Lowe refers to the data items contained in messages. The relation between the messages and the data items does imply however, that when a message is read exactly as it is sent, both agents will agree over the variables sent in the message, and the other way around. Thus, if all messages in a protocol are read as they were sent, there must be agreement over all variables in the protocol.

The definition of *I-AGREE* does not involve *all* communications, but only the set $prec(p, \ell)$ of communications that precede a claim. However, it turns out that the way in which full agreement is made precise in terms of CSP, as can be checked by compiling Casper-code into CSP, it also takes only preceding communications into account. For this, running-commit signals (see [11]) are introduced in the protocol. For each role, a running signal is added to the last communication preceding the agreement claim. In the role of the claim, a commit signal is added to the last communication. Full injective agreement over all roles requires that the running signals of each role precede the commit signal. This corresponds to the order requirements of *I-AGREE*.

It follows that the notions of *I-AGREE* and full injective agreement over all roles coincide.

4.2 Hierarchy

The formulations of the four security properties in the previous sections clearly reveal their relative strengths in preventing attacks. Every injective protocol is also non-injective and if a protocol satisfies agreement then it satisfies synchronisation too. Figure 2 shows this hierarchy. An arrow from property X to property Y means that every protocol satisfying X also satisfies Y. Phrased differently, the class of protocols satisfying X is included in the class satisfying Y.

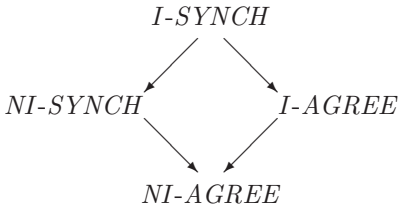

Fig. 2. Hierarchy of security properties.

The proof of the following theorem simply follows from the definitions above.

Theorem 1. *The security properties I-SYNCH, NI-SYNCH, I-AGREE, and NI-AGREE satisfy the inclusion relation as depicted in Figure 2.*

The question whether the inclusions in Figure 2 are strict is harder to answer. This is due to the abstractness of our trace model. Since our approach is parameterised over the actual semantics, and thus over the intruder model, we cannot determine for a given protocol to which class it belongs. Therefore, strictness of the inclusions can only be answered relative to a given semantics. Consequently, the following reasoning will be at a conceptual level only.

If we take e.g. a model where all agents simply follow their roles and the intruder has no capabilities at all, then the diamond in Figure 2 collapses into a single class. The same holds if the intruder can only eavesdrop on the communications. However, in the Dolev-Yao model, all inclusions are strict. The case of injectivity vs. non-injectivity has been studied extensively before. The MSC on the left in Figure 3 shows a protocol that satisfies *NI-SYNCH* and *NI-AGREE*, but neither *I-SYNCH*, nor *I-AGREE*.

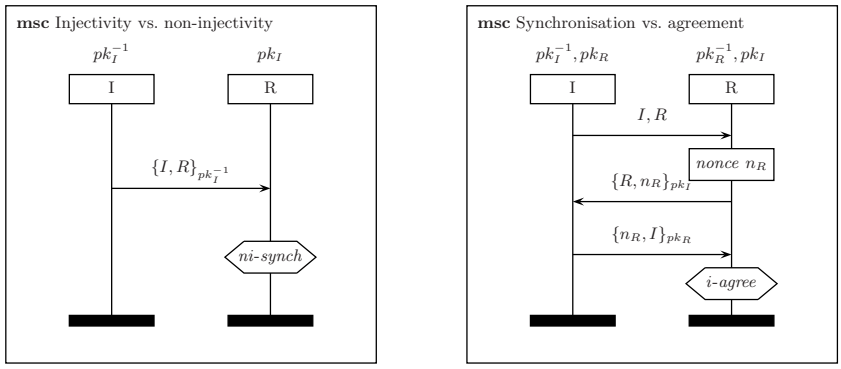

Fig. 3. Distinguishing Protocols

The intruder will only be able to construct message $\{I, R\}_{pk_I^{-1}}$ after having eavesdropped this message from a previous run. Therefore every read event of this message is preceded by a corresponding send event, so the protocol is both *NI-SYNCH* and *NI-AGREE*. However, once the intruder has learnt this message, he can replay it as often as desired, so the protocol is not injective.

A distinguishing example between synchronisation and agreement is depicted on the right in Figure 3. As confirmed by the Casper/FDR tool set, this protocol satisfies unilateral authentication in the sense of agreement (both injective and non-injective). However, the protocol does not satisfy synchronisation (both variants): the intruder can send message I, R long before I actually initiates the protocol, making R to believe that I has requested the start of a session before he actually did.

The two examples show that the diamond in Figure 2 is strict if the intruder has the capabilities to eavesdrop, deflect and inject messages. Both examples also imply that there are no arrows between *NI-SYNCH* and *I-AGREE*.

In Figure 4 we show how the difference between *NI-SYNCH* and *NI-AGREE* might be exploited. Here, R is an Internet Service Provider, used by I. I pays R for the time he is connected. When I wants to connect, R retrieves the certificate of I from the trusted server S and uses this to authenticate I. After a successful session, I is billed from the moment the first message was received by R.

This protocol is a slightly modified version of the Needham-Schroeder-Lowe protocol. It can be exploited as follows. An intruder can send the first message preemptively, causing R to initiate a session with what it believes is I. If at some later time I decides to initiate a session with R and finishes it successfully, I will receive a bill that is too high. In fact, although, this protocol satisfies agreement for R, the first message is not authenticated at all.

This protocol does not satisfy synchronisation. The protocol can be easily modified to satisfy *NI-SYNCH* and thus to be resilient against the sketched type of timing attacks.

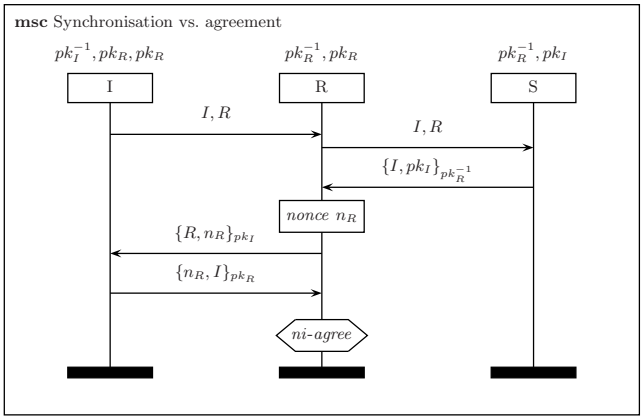

Fig. 4. A protocol that satisfies *NI-AGREE* but not *NI-SYNCH*.

5 Conclusions

In this section we summarize the main contributions of our research and discuss some directions for future work.

First of all, we have defined a general trace model for security protocols which allows for the definition of security properties in an intensional style. This trace model is not tied to a particular semantics, making it independent of e.g. the execution model of an agent and the intruder model. The starting point of the trace model is a role-based protocol description, where security claims are local to

the protocol roles. Such a subjective security claim expresses what an agent may safely assume after having executed his part of the protocol. The events in our model are extended with event labels and run identifiers to unambiguously define the origin of an event. These attributes are not under control of an intruder, but serve to identify the events when reasoning about protocols. The main motivation for developing the trace model was to study intensionality of specifications (which we call synchronisation) and agreement in an abstract framework, allowing us to pinpoint what the exact differences are. Due to the uniform phrasing, the two notions of authentication can be distinguished easily: agreement allows that an intruder injects a (correct and expected) message before it is sent by the originator of the message. As for agreement, we provide both an injective and a non-injective variant of synchronisation.

Since we only assumed an abstract trace model, the theory presented in this paper does not suffice to prove protocols (in)correct. For this purpose we are currently defining a canonical operational semantics of security protocols which satisfies the requirements for the trace model put forward here. Experiments with this operational definition of synchronisation indicate that formal proofs that security protocols satisfy synchronisation are feasible.

Finally, we think that it would be interesting to study extensions of our model, such as timing and recentness as has been done in [8].

References

1. W. Diffie, P.C. van Oorschot, and M.J. Wiener. Authentication and authenticated key-exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.
2. A.G. Engels, S. Mauw, and M.A. Reniers. A hierarchy of communication models for Message Sequence Charts. *SoCP*, 44:253–292, 2002.
3. R. Focardi and F. Martinelli. A uniform approach for the definition of security properties. In *World Congress on Formal Methods (1)*, pages 794–813, 1999.
4. D. Gollmann. What do we mean by entity authentication. In *Proc. Symposium on Research in Security and Privacy*, pages 46–54. IEEE, 1996.
5. ITU-TS. *Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1999.
6. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 147–166. LNCS 1055, 1996.
7. G. Lowe. Some new attacks upon security protocols. In *Proc. 9th Computer Security Foundations Workshop*, pages 162–169. IEEE, 1996.
8. G. Lowe. A hierarchy of authentication specifications. In *Proc. 10th Computer Security Foundations Workshop*, pages 31–44. IEEE, 1997.
9. R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21:120–126, 1978.
10. A. W. Roscoe. Intensional Specifications of Security Protocols. In *Proc. 9th Computer Security Foundations Workshop*, pages 28–38. IEEE, 1996.
11. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and Bill Roscoe. *Modelling and Analysis of Security Protocols*. Addison Wesley, 2000.