Typed MSR: Syntax and Examples ⋆

Iliano Cervesato

Advanced Engineering and Sciences Division, ITT Industries, Inc.,
2560 Huntington Avenue, Alexandria, VA 22303 — USA
iliano@itd.nrl.navy.mil

Abstract. Many design flaws and incorrect analyses of cryptographic protocols can be traced to inadequate specification languages for message components, environment assumptions, and goals. In this paper, we present *MSR*, a strongly typed specification language for security protocols, which is intended to address the first two issues. Its typing infrastructure, based on the theory of dependent types with subsorting, yields elegant and precise formalizations, and supports a useful array of static check that include type-checking and access control validation. It uses multiset rewriting rules to express the actions of the protocol. The availability of memory predicates enable it to faithfully encode systems consisting of a collection of coordinated subprotocols, and constraints allow tackling objects belonging to complex interpretation domains, *e.g.* time stamps, in an abstract and modular way. We apply *MSR* to the specification of several examples.

1 Introduction

The design and analysis of cryptographic protocols are notoriously complex and error-prone activities. Part of the difficulty derives from subtleties of the cryptographic primitives. Another portion is due to their deployment in distributed environments plagued by powerful and opportunistic attackers. We claim that a third major source of problems arises from the use of ambiguous, complex or inexpressive languages for the specification of protocols, of the assumptions on their operating environment, and of their goals.

The Dolev-Yao model of security [19, 14] tackles the first problem by promoting an abstraction that has the effect of separating the analysis of the message flow from the validation of the underlying cryptographic operations. It assumes that elementary data such as principal names, keys and nonces are atomic rather than bit strings, and views the message formation operations (*e.g.* concatenation and encryption) as symbolic combinators. The cryptographic operations are therefore assumed to be flawless. This model is generally reasonable for authentication protocols and underlies most systems designed for protocol analysis, *e.g.* [5, 18, 16, 1, 13]. Within the Dolev-Yao model, the capabilities of the intruder are circumscribed. They can be in many respects neutralized by relying on appropriate message formats [2, 22]. However, practical reasons, such as limited bandwidth, sometimes make such architectures inviable.

We claim that a significant source of faulty designs and contradictory analyses can be traced to shortcomings in the languages used to specify protocols. The popular "usual

⋆ Partially supported by NRL under contract N00173-00-C-2086.

Appeared in the *Proceedings of the First International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security — MMM'01* (V.I. Gorodetski, V.A. Skormin and L.J. Popyack, editors), pp. 159–176, ©Springer-Verlag LNCS 2052, St. Petersburg, Russia, 21–23 May 2001

notation" relies on the Dolev-Yao model and describes a protocol as the sequence of the messages transmitted during an expected run. Besides distracting the attention from the more dangerous unexpected runs, this description expresses fundamental assumptions and requirements about message components, the operating environment and the protocol's goals as side remarks in natural language. This is clearly ambiguous and error-prone. Strand formalizations [16], like most modern languages, represent protocols as a collection of independent roles that communicate by exchanging message. Their reliance on a fair amount of natural language still makes it potentially ambiguous.

In [9, 15], we proposed *MSR*, a language based on multiset rewriting, as a formalism for unambiguously representing authentication protocols, with the aim of studying properties such as the decidability of attack detection. The actions within a role were formulated as multiset rewrite rules, threaded together by dedicated *role state predicates*. The nature and properties of message components was expressed in a relational manner by means of *persistent information predicates* and to a minor extent by typing declarations. In particular, variables that ought to be instantiated to "fresh" objects during execution were marked with an existential quantifier. In [11, 10], we proved the substantial equivalence between *MSR* and extensions of popular formalisms such as strand spaces. Nonetheless, the resulting specifications were not completely satisfactory for two reasons: persistent information proved difficult to reason about, and the rigid structure of *MSR* rules limited its applicability to basic authentication protocols.

This paper proposes a thorough redesign of *MSR* and establishes this formalism as a usable specification language for security protocols. The major innovations include the adoption of a flexible yet powerful typing methodology that subsumes persistent information predicates, and the introduction of memory predicates and of constraints on interpreted domains that significantly widen the range of applicability of this language.

The type annotations of our new language, drawn from the theory of dependent types with subsorting, enable precise object classifications for example by distinguishing keys on the basis of the principals they belong to, or in function of their intended use. Therefore, the public key of any two principals can be assigned a different type, in turn distinct from their digital signature keys. Protocol specifications, called protocol theories in *MSR*, are strongly typed, and we have devised algorithms for statically catching type violations, *e.g.* the use of a shared key to perform public-key encryption [7]. Our typing infrastructure can point to more subtle access control errors, such as a principal trying to encrypt a message with a key that does not belong to him [6].

Memory predicates allow a principal to remember information across role executions. Their presence opens the doors to the specification of protocols structured as a collection of coordinated subprotocols. In this paper, we exemplify this possibility by formalizing the Neuman-Stubblebine repeated authentication protocol [20], which lies outside the reaches of our previous version of *MSR*. In [8], we use this technique to give a specification of the Dolev-Yao intruder that lies fully within the syntax of *MSR* roles.

Constraints are another novelty of the language presented in this paper. They permit referring to objects belonging to complex interpretation domains in an abstract and modular way. Our specification of the Neuman-Stubblebine protocol [20] relies on constraints to verify the validity of timestamps: how these objects and their operations are implemented is invisible (and irrelevant) to the resulting protocol theory.

This presentation is organized as follow. In Section 2, we introduce the syntax of *MSR*. The next three sections formalize as many popular case studies: Section 3 implements the abridged version of the Needham-Schroeder public-key authentication protocol; Section 4 extends this specification to the full protocol, inclusive of the server activity; and Section 5 formalizes the Neuman-Stubblebine protocol. Section 6 summarizes the ideas discussed in this paper and hints at directions of future work.

2 Typed MSR

In the past, cryptoprotocols have often been presented as the temporal sequence of messages being transmitted during a "normal" run. Recent proposals champion a view that places the involved parties in the foreground. A protocol is then a collection of independent *roles* that communicate by exchanging messages, without any reference to runs of any kind. A role has an owner, the principal that executes it, and specifies the sequence of messages that he/she will send, possibly in response to receiving messages of some expected form. *MSR* adopts and formalizes this perspective. A role is given as a parameterized collection of multiset rewrite rules that encode the expected message receptions and the corresponding transmissions. Rule firing emulates receiving (and accepting) a message and/or sending a message, the smallest execution steps. The messages in transit, the actions and information available to the roles, and other data constitute the state of execution of a protocol. Rules implement partial transformations between states. Their applicability is constrained by the contents of the current state and by the satisfaction of guards. Execution is preceded by static type-checking [7] and access control validation [6] which limits the number of run-time checks and allows catching common specification errors early.

This section describes the form of an *MSR* specification. More specifically, in Section 2.1, we define our notion of messages. In Section 2.2, we present the predicates that appear in a state, in turn defined in Section 2.3. In Section 2.4, we introduce the typing infrastructure that allows us to make sense of these objects. In Section 2.5, we discuss rules and their constituents. Roles and protocol theories are defined in Section 2.6.

2.1 Messages

Messages are obtained by applying a number of message forming constructs, discussed below, to a variety of *atomic messages*. The atomic messages we will consider in this paper are principal identifiers A, keys k, nonces n, timestamps T, and raw data m (*i.e.* pieces of data that have no other function in a protocol than to be transmitted). We formalize our notion of atomic message in the following grammatical productions:

$$a ::= \mathsf{A} \mid \mathsf{k} \mid \mathsf{n} \mid \mathsf{T} \mid \mathsf{m}$$

We will also use B to denote a principal while we reserve the letter S for servers. Although we limit the discussion in this paper to these kinds of atomic messages, it should be noted that others can be accommodated by extending the appropriate definitions.

The *message constructors* we will consider consist of concatenation $(t_1\, t_2)$, shared-key encryption $\{t\}_k$, public-key encryption $\{\!|t|\!\}_k$, and digital signature $[t]_k$. Altogether, they give rise to the following definition of a *message*, or more properly a *term*.

$$t \ ::= \ a \ \mid \ x \ \mid \ t_1 \, t_2 \ \mid \ \{t\}_k \ \mid \ \{\!|t|\!\}_k \ \mid \ [t]_k$$

Observe that we use a different syntax for shared-key and public-key encryption. We could have identified them, as done in many approaches. We choose instead to distinguish them to show the flexibility and precision of our technique. Similarly, we define digital signatures as an independent primitive operation rather than as asymmetric key encryption with a private key. As usual, $[t]_k$ denotes the term t being signed, together with the signer's certificate cryptographically constructed from t using the key k.

Again, other constructors, for example hash functions, can easily be accommodated by extending the appropriate definitions. We refrain from doing so since their inclusion would lengthen the discussion without introducing substantially new concepts.

A *parametric message* allows variables x wherever terms could appear. We use a sans-seriffed font to denote possibly parametric principals A (or B), keys k, nonces n, timestamps T and raw data m. Constants and variables constituted the class of *elementary terms*, denoted with the letter e.

2.2 Message Predicates

Message predicates are the basic ingredient of states, defined in Section 2.3. They are atomic first-order formulas with zero or more terms as their arguments. More precisely, they are applied to ordered sequences of terms called *message tuples* and denoted $\bar{t}$.

The predicates that can enter a state or a rewrite rule are of three kinds:

- First, the predicate $\mathsf{N}(\_)$ implements the contents of the *public network* in a distributed fashion: for each (ground) message t currently in transit, the state will contain a component of the form $\mathsf{N}(t)$.
- Second, active roles rely on a number of *role state predicates*, generally one for each rule in them, of the form $\mathsf{L}_l(\_, \ldots, \_)$, where l is a unique identifying label. The arguments of this predicate record the value of known parameters of the execution of the role up to the current point.
- Third, a principal A can store data in private memory predicates of the form $\mathsf{M}_A(\_, \ldots, \_)$ that survives role termination and can be used across the execution of different roles, as long as the principal stays the same.

The reader familiar with our previous work on *MSR* will have noticed a number of differences with respect to the definitions given in [9, 10]. Memory predicates are indeed new. They are intended to model situations that need to maintain data private across role executions: for example, this allows a principal to remember his Kerberos ticket, or the trusted-third-party of a fair exchange protocol to avoid fraudulent recoveries from aborted transactions. Memory predicates can further be used to represent such entities as local clocks, as we will see in Section 5. Another difference with respect to our earlier work is the absence of a dedicated predicate retaining the intruder's knowledge. This can however be easily implemented using memory predicates, as described in [8].

2.3 States

States are a fundamental concept in *MSR*. They are the objects transformed by rewrite rules to simulate message exchange and information update and, together with execution traces, they are the hypothetical scenarios on which protocol analysis is based. A

state S is a finite collection of ground state predicates:

$$S ::= \cdot \mid S,\ \mathsf{N}(t) \mid S,\ \mathsf{L}_l(\bar{t}) \mid S,\ \mathsf{M}_\mathsf{A}(\bar{t})$$

Protocol rules transform states. They do so by identifying a number of predicates, removing them from the state, and adding other, usually related, state elements. The antecedent and consequent of a rewrite rule embed therefore substates. However, in order to be applicable to a wide array of states, rules usually contain variables that are instantiated at application time. This calls for a parametric notion of states. For the most part, this reduces to admitting variables in embedded terms. However, role state predicates need to be created on the spot in order to avoid interferences between concurrently executing role instances. We achieve this by introducing variables, denoted L, that are instantiated to actual role state predicates during execution.

2.4 Types

While types played a very modest role in the original definition of *MSR* [9, 10], they stand at the core of the extension presented in this paper. Through typing, we can enforce basic well-formedness conditions (*e.g.* that only keys be used for encrypting a message), as described in detail in [7]. Types also provide a statically checkable way to ascertain complex desiderata such as, for example, that no principal may grab a key he/she is not entitled to access. This aspect is thoroughly analyzed in [6]. The central role of types in our present approach is witnessed by the fact that they subsume and integrally replace the "persistent information" of the original *MSR* [10].

The typing machinery that best fits our goals is based on the type-theoretic notion of *dependent product types with subsorting* [3, 21]. Rather than delving into the depth of the definitions and properties of this formalism, we introduce only the facets that we will use, and only to the extent we will need them.

Types are syntactic constructions that are used to classify other syntactic expression, such as terms. By doing so, they give them a *meaning*, saying for example that an object we interpret as a key is not a nonce. Whenever a key is used where a nonce is expected, something has gone wrong since the meaning of this term has been violated. The types we will use in this paper are summarized in the following grammar:

$$\tau ::= \mathsf{principal} \mid \mathsf{nonce} \mid \mathsf{shK}\,A\,B \mid \mathsf{pubK}\,A \mid \mathsf{privK}\,k \mid \mathsf{sigK}\,A \mid \mathsf{verK}\,k \mid \mathsf{time} \mid \mathsf{msg}$$

Needless to say, the types "principal" and "nonce" are used to classify principals and nonces respectively. The next three productions allow distinguishing between shared keys, public keys and private keys. Dependent types offer a simple and flexible way to express the relations that hold between keys and their owner or other keys. Given principals "A" and "B", a shared key "k" between "A" and "B" will have type "shKAB". Here, the type of the key *depends* on the specific principals "A" and "B". Similarly, a constant "k" is given type "pubK A" to indicate that it is a public key belonging to "A". We use dependent types again to express the relation between a public key and its inverse. Continuing with the last example, the inverse of "k" will have type "privK k", from which it is easy to establish that it belongs to principal "A". A similar design principle applies in the case of digital signatures: the signature key "k" of principal

"A" has type "sigK A" while its inverse, the verification key "k′", has type "verK k". Timestamps are assigned type "time".

We use the type msg to classify generic messages. Clearly raw data have type msg. This is however not sufficient since nonce, keys, timestamps, and principal identifiers are routinely part of messages. We solve this problem by imposing a *subsorting* relation between types. In this paper, each of the types discussed above, with the exception of signature keys and their inverses, will be a subtype of msg. With the appropriate array of typing rules (see [7]), not defining signature and signature verification keys as subsorts of "msg" has the effect of banning these keys from well-typed messages, except as the unrecoverable indices of signed messages: any attempt at transmitting a signature key will be statically marked as violating the typing policy.

Again, the types and the subsorting rules above should be thought of as a reasonable instance of our approach rather than the approach itself. Other schemas can be specified by defining appropriate types and how they relate to each other. For example, an application may find it convenient to see each of the above types related to encryption or decryption as a subtype of a universal key type, say, key, in turn a subsort of msg. Alternatively, we may want to define distinct types for long-term keys and have them not be a subsort of msg, prohibiting in this way the transmission of long-term secrets as parts of messages. We are already handling signature keys in this way.

Predicate symbols are assigned a type by listing the type of their arguments. It is tempting to define the type of a tuple as the sequence of the types of its components. Therefore, if A is a principal name and $\mathsf{k_A}$ is a public key for A, the tuple $(\mathsf{A}, \mathsf{k_A})$ would have type "principal $\times$ pubK A" (the *Cartesian product* symbol "$\times$" is the standard constructor for type tuples). This construction allows us to associate a generic principal with A's public key: if B is another principal, then $(\mathsf{B}, \mathsf{k_A})$ will have this type as well. We will often need stricter associations, such as between a principal and his *own* public key. In order to achieve this, we will rely on the notion of *dependent type tuple*. In this example, the tuple $(\mathsf{A}, \mathsf{k_A})$ will be attributed type "$\mathsf{principal}^{(A)} \times \mathsf{pubK}\ A$", where the variable A in "$\mathsf{principal}^{(A)}$" records the name of the principal at hands and forces the type of the key to be "pubK A" for this particular A: therefore $(\mathsf{A}, \mathsf{k_A})$ is a valid object of this type, but $(\mathsf{B}, \mathsf{k_A})$ is now ill-typed since $\mathsf{k_A}$ has type "pubK A" rather than the expected "pubK B".[1]

We attribute a type to a term tuple by collecting the type of each constituent message, but we label these objects with variables to be used in later types that may depend on them. Thus, a *dependent type tuple* is an ordered sequence of parameterized types:

$$\bar{\tau} \ ::= \ \cdot \ \mid \ \tau^{(x)} \times \bar{\tau}$$

Given a dependent tuple type $\tau^{(x)} \times \bar{\tau}$, we will drop the label $^{(x)}$ whenever the variable x does not occur (free) in $\bar{\tau}$. The resulting simplified notation, $\tau \times \bar{\tau}$, will help writing more legible specifications when possible. As for term tuples, we will omit the leading "$\cdot$" whenever convenient.

[1] Our dependent type tuples are usually called strong dependent sums in the type theoretic community, and the standard notation for the dependent type tuple we have written as "$\mathsf{principal}^{(A)} \times \mathsf{pubK}\ A$" is "$\Sigma A : \mathsf{principal}.\ \mathsf{pubK}\ A$". We believe that our syntax is likely to be more clear to the target audience of this paper.

2.5 Rules

The core of a *rule* has the form "$lhs \rightarrow rhs$". Rules are the basic mechanism that enables the transformation of a state into another, and therefore the simulation of protocol execution: whenever the antecedent "lhs" matches part of the current state, this portion may be substituted with the consequent "rhs" (after some processing).

It is convenient to make protocol rules parametric so that the same rule can be used in a number of slightly different scenarios (*e.g.* without fixing interlocutors or nonces). A typical rule will therefore mention variables that will be instantiated to actual terms during execution. Typed universal quantifiers can conveniently express this fact. This idea is captured by the following grammar:

$$r \ ::= \ lhs \rightarrow rhs \ \mid \ \forall x : \tau . \, r$$

Both the right-hand side and the left-hand side of a rule embed a finite collection of *parametric message predicates*, some ground instance of which execution will respectively add to and retract from the current state when the rule is applied:

$$\bar{P} \ ::= \ \cdot \ \mid \ \bar{P}, \ \mathsf{N}(t) \ \mid \ \bar{P}, \ L(\bar{e}) \ \mid \ \bar{P}, \ \mathsf{M}_A(\bar{t})$$

Observe that predicate sequences differ from states (see Section 2.3) mainly by the limited instantiation of role state predicates: in a rule, these objects consist of a role state predicate variable applied to as many elementary terms as dictated by its type (this is enforced by the typing rules in [7]). Recall that elementary terms are either variables or atomic message constants. Network and memory predicates will in general contain parametric terms, although not necessarily raw variables as arguments.

The Dolev-Yao model [14] champions a symbolic interpretation of cryptographic primitives that reduces messages to expressions in an initial algebra. Some of the components of a message, such as timestamps, are however subject to operations or tests that are not conveniently expressed in this way. We reconcile the simplicity of the Dolev-Yao model and the necessity to accommodate objects drawn from complex interpretation domains by treating the latter as atomic constants when embedded in messages, but by relying on dedicated *constraint handlers* to perform operations and resolve tests. These invocations enter the syntax of *MSR* as *constraints* in the left-hand side of a rule. We use the letter χ to denote them. Constraints are not part of the state, but should rather be thought of as guards to the applicability of a rule.

In this paper, the only message constituents that require a constraint handler are timestamps. We need to check their validity against the current time, which is modeled by arithmetic constraints involving the usual ordering relations; a possible such constraint could be $(T < T_{now})$ where T and T_{now} are respectively the timestamp and the current time. We will also need to set alarms T_{alarm} by adding predetermined temporal values, say T_{val}, to the current time T_{now}. This operation is expressed as the constraint $(T_{alarm} = T_{now} + T_{val})$. The domain of timestamps will correspond to the real numbers or any sufficiently precise approximation supported by the implementation at hand.

The use of constraints allows for an abstract architecture since it isolates the specification of interpretation domains away from the formalization of the security protocols that use them. The interface is limited to a few type declarations and a syntax for the

operations and tests that can enter a constraint. Constraint handlers are then external and interchangeable modules that can be plugged to the protocol specification on demand.

The *left-hand side*, or *antecedent*, of a rule is a finite collection of parametric message predicates guarded by finitely many constraints on interpreted data:

$$lhs \ ::= \ \bar{P} \ \mid \ lhs, \ \chi$$

The *right-hand side*, or *consequent*, of a rule consists of a predicate sequence possibly prefixed by a finite string of fresh data declarations such as nonces or short-term keys. We rely on the existential quantification symbol to express data generation:

$$rhs \ ::= \ \bar{P} \ \mid \ \exists x : \tau. \, rhs$$

2.6 Roles and Protocol Theories

Role state predicates record the information accessed by a rule. They are also the mechanism by which a rule can enable the execution of another rule in the same role. Relying on a fixed protocol-wide set of role state predicates is dangerous since it could cause unexpected interferences between different instances of a role executing at the same time. Instead, we make role state predicates local to a role by requiring that fresh names be used each time a new instance of a role is executed. As in the case of rule consequents, we achieve this effect by using existential quantifiers: we prefix a collection of rules ρ that should share the same role state predicate L by a declaration of the form "$\exists L : \bar{\tau}$", where the typed existential quantifier indicates that L should be instantiated with a fresh role state predicate name of type $\bar{\tau}$.

With this insight, the following grammar defines the notion of *rule collection*:

$$\rho \ ::= \ \cdot \ \mid \ \exists L : \bar{\tau}. \, \rho \ \mid \ r, \ \rho$$

It should be observed that this definition allows for role state predicate parameters declarations and rules to be interleaved in a rule collection. We will however generally divide a collection in a *preamble* where all roles state parameters are declared, and a *body* that lists the rules that constitute a role.

A *role* is given as the association between a *role owner* A and a collection of rules ρ. Some roles, such as those implementing a server or an intruder, are intrinsically bound to a few specific principals, often just one. We call them *anchored roles* and denote them as ρ^{A}. Here, the role owner A is an actual principal name, a constant. Other roles can be executed by any principal. In these cases A must be kept as a parameter bound to the role. These *generic roles* are denoted $\rho^{\forall A}$, where the implicitly typed universal quantification symbol implies that A should be instantiated to a principal before any rule in ρ is executed, and sets the scope of the binding to ρ. Observe that in this case A is a variable.

We require that the owner of a role ρ be the first argument of all the role state predicates in the rules that constitute it. This object shall also be the subscript of every memory predicate in ρ. These constraints are formally expressed in the typing and access control policy of *MSR* [7, 6].

A *protocol theory*, written $\mathcal{P}$, is a finite collection of roles:

$$\mathcal{P} \ ::= \ \cdot \ \mid \ \mathcal{P},\ \rho^{\forall A} \ \mid \ \mathcal{P},\ \rho^{\mathsf{A}}$$

It should be observed that we do not make any special provision for the intruder. The adversary is expressed as one or more roles in the same way as proper protocols. We have illustrated in [8, 6] how this is achieved for the standard Dolev-Yao intruder.

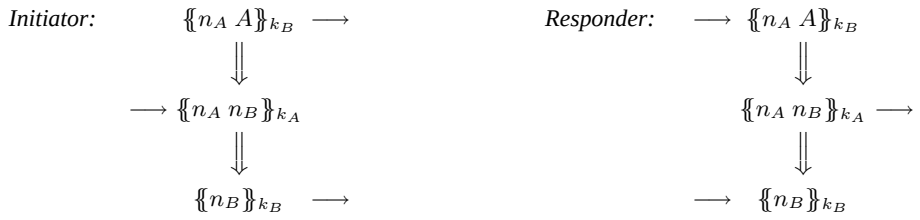
3 Simplified Needham-Schroeder Authentication Protocol

As our first example using *MSR* as a specification language, we will formalize the Needham-Schroeder public-key authentication protocol [19]. We familiarize the reader with *MSR* by first considering the two-party nucleus of this protocol. We will tackle the full protocol, which relies on a server to generate session keys in Section 4.

1. $A \rightarrow B$: $\{\!|n_A\ A|\!\}_{k_B}$
2. $B \rightarrow A$: $\{\!|n_A\ n_B|\!\}_{k_A}$
3. $A \rightarrow B$: $\{\!|n_B|\!\}_{k_B}$

The server-less variant of the Needham-Schroeder public-key protocol [19] is a two-party crypto-protocol aimed at authenticating the initiator A to the responder B (but not necessarily vice versa). It is expressed as the expected run on the right in the "usual notation" (where we have used our syntax for messages). In the first line, the initiator A encrypts a message consisting of a nonce n_A and her own identity with the public key k_B of the responder B, and sends it (ideally to B). The second line describes the action that B undertakes upon receiving and interpreting this message: he creates a nonce n_B, combines it with A's nonce n_A, encrypts the outcome with A's public key k_A, and sends the resulting message out. Upon receiving this message in the third line, A accesses n_B and sends it back encrypted with k_B. The run is completed when B receives this message.

MSR and most modern security protocol specification languages focus on the sequence of actions that each principal involved in a protocol executes. We called such sequences roles. Strand spaces [16] are a simple and intuitive notation that emphasize this notion. The strand representation of this protocol is given by the following picture:



Here incoming and outgoing single arrows respectively denote the reception and transmission of a message. The double arrows assign a temporal ordering on these actions.

We will now express each role in turn in the syntax of *MSR*. For space reasons, we will typeset homogeneous constituents, namely the universal variable declarations and the predicate sequences in the antecedent and consequent, in columns within each rule; we will also rely on some minor abbreviation. We mark types that can be reconstructed from the other information present in a rule by denoting them in a shaded font.

The initiator's actions are represented by the following two-rule role:

$$\left(\begin{array}{llll} \exists L : \mathsf{principal} \times \mathsf{principal}^{(B)} \times \mathsf{pubK}\ B \times \mathsf{nonce}. & & & \\ \forall B : \mathsf{principal}. \\ \forall k_B : \mathsf{pubK}\ B. & . & \rightarrow \quad \exists n_A \colon \mathsf{nonce}. & \mathsf{N}(\{\!|n_A\ A|\!\}_{k_B}) \\ & & & L(A, B, k_B, n_A) \\ \forall \ldots \\ \forall k_A : \mathsf{pubK}\ A. & \mathsf{N}(\{\!|n_A\ n_B|\!\}_{k_A}) & \rightarrow & \mathsf{N}(\{\!|n_B|\!\}_{k_B}) \\ \forall k'_A : \mathsf{privK}\ k_A. & L(A, B, k_B, n_A) & & \\ \forall n_A, n_B : \mathsf{nonce}. & & & \end{array}\right)^{\forall A}$$

Clearly, any principal can engage in this protocol as an initiator (or a responder). Our encoding is therefore structured as a generic role. Let A be its postulated owner. The first rule formalizes of the first line of the "usual notation" description of this protocol from A's point of view. It has an empty antecedent since initiation is unconditional in this protocol fragment. Its right-hand side uses an existential quantifier to mark the nonce n_A as fresh. The consequent contains the transmitted message and the role state predicate $L(A, B, k_B, n_A)$, necessary to enable the second rule of this protocol: it corresponds to the topmost double arrow in the strand specification on the left. The arguments of this predicate record variables used in the second rule.

The second rule encodes the last two lines of the "usual notation" and strand description. It is applicable only if the initiator has executed the first rule (enforced by the presence of the role state predicate) and she receives a message of the appropriate form. Its consequent sends the last message of the protocol. The presence of both a message receptions and transmission in the same rule corresponds to the second double arrow in the strand specification of the initiator of this role.

Our notation provides a specific type for each variable appearing in these rules. The equivalent "usual notation" specification relies instead on natural language and conventions to convey this same information, with clear potential for ambiguity. Observe that most declarations are grayed out, meaning that they can be reconstructed automatically: this simplifies the task of the author of the specification by enabling him or her to concentrate on the message flow rather than on typing details, and of course it limits the size of the specification. Algorithmic rules for this form of type reconstruction are the subject of a forthcoming paper.

The rationale behind the reconstructible types in this rule are as follows. The universal declarations for B, k_B, and n_A and the type of the existential declaration for n_A in the first rule can be deduced from the declaration of the role state predicate L. The declarations for k_A and k'_A can be omitted since k_A must be the public key of A, and k'_A be the corresponding private key. The possibility of reconstructing this information is intimately linked to the access control policy of *MSR*, formally defined in [6]. The only universal declaration that cannot be reconstructed is "$\forall n_B$: nonce": n_B is clearly a universally quantified variable in this rule, but there is no hint that it should be a nonce. Let us now examine the declaration for L: the first argument is always the rule owner, which is a principal. The third argument must be the public key of some principal B because of the way k_B is used. Therefore, we only need to indicate that B is bound in the second argument of L.

The responder is encoded as the generic role below, whose owner we have mnemonically called B. The first rule of this role collapses the two topmost lines of the "usual notation" specification of this protocol fragment from the receiver's point of view. The second rule captures the reception and successful interpretation of the last message in the protocol by B: this step is often overlooked. This rule has no consequent.

$$\left(\begin{array}{lll} \exists L : \mathsf{principal}^{(B)} \times \mathsf{principal} \times \mathsf{pubK}\ B^{(k_B)} \times \mathsf{privK}\ k_B \times \mathsf{nonce}. & & \\ \forall k_B : \mathsf{pubK}\ B. & & \\ \forall k'_B : \mathsf{privK}\ k_B. & & \\ \forall A : \mathsf{principal}. \quad \mathsf{N}(\{\!|n_A\ A|\!\}_{k_B}) & \rightarrow & \exists n_B\colon \mathsf{nonce}.\ \mathsf{N}(\{\!|n_A\ n_B|\!\}_{k_A}) \\ \forall n_A : \mathsf{nonce}. & & L(B, A, k_B, k'_B, n_B) \\ \forall k_A : \mathsf{pubK}\ A & & \\ & & \\ \forall \ldots & \mathsf{N}(\{\!|n_B|\!\}_{k_B}) & \\ \forall n_B : \mathsf{nonce}. & L(B, A, k_B, k'_B, n_B) \quad \rightarrow & . \end{array}\right)^{\forall B}$$

Again, observe that most typing information has been grayed out since it can be reconstructed from the way variables are used and the few types left.

4 Full Needham-Schroeder Authentication Protocol

We will now specify the full version of the Needham-Schroeder public-key authentication protocol [19], which relies on a server S to generate the keys k_A and k_B used in the fragment discussed in the previous section. This protocol is written in the "usual notation" to the right of this text. The simplified version discussed in Section 3 corresponds to lines (3), (6) and (7) of this protocol. In line (1), A asks the server for a key to communicate with B, which is obtained in the signed message on line (2). The responder issues and is granted a similar request in lines (4) and (5), respectively.

1. $A \rightarrow \mathsf{S}$: $A\ B$
2. $\mathsf{S} \rightarrow A$: $[k_B\ B]_{k_\mathsf{S}}$
3. $A \rightarrow B$: $\{\!|n_A\ A|\!\}_{k_B}$
4. $B \rightarrow \mathsf{S}$: $B\ A$
5. $\mathsf{S} \rightarrow B$: $[k_A\ A]_{k_\mathsf{S}}$
6. $B \rightarrow A$: $\{\!|n_A\ n_B|\!\}_{k_A}$
7. $A \rightarrow B$: $\{\!|n_B|\!\}_{k_B}$

The actions of the initiator are expressed in *MSR* by the following generic role, which consists of three rules that have to fire in sequence, and consequently mentions two role state predicate declarations. The first rule corresponds to line (1) in the "usual notation", the second to lines (2) and (3), and the third to lines (6) and (7).

$$\left(\begin{array}{lll} \exists L : \mathsf{principal} \times \mathsf{principal}. & & \\ \exists L' : \mathsf{principal} \times \mathsf{principal}^{(B)} \times \mathsf{pubK}\ B \times \mathsf{nonce}. & & \\ \forall B : \mathsf{principal}. \quad \cdot & \rightarrow & \mathsf{N}(A\ B) \\ & & L(A, B) \\ \forall \ldots & & \\ \forall k_\mathsf{S} : \mathsf{sigK}\ \mathsf{S}. \quad \mathsf{N}([k_B\ B]_{k_\mathsf{S}}) & \rightarrow & \exists n_A\colon \mathsf{nonce}.\ \mathsf{N}(\{\!|n_A\ A|\!\}_{k_B}) \\ \forall k'_\mathsf{S} : \mathsf{verK}\ k_\mathsf{S}. \quad L(A, B) & & L'(A, B, k_B, n_A) \\ \forall k_B : \mathsf{pubK}\ B. & & \\ \forall \ldots & & \\ \forall k_A : \mathsf{pubK}\ A. \quad \mathsf{N}(\{\!|n_A\ n_B|\!\}_{k_A}) & \rightarrow & \mathsf{N}(\{\!|n_B|\!\}_{k_B}) \\ \forall k'_A : \mathsf{privK}\ k_A. \quad L'(A, B, k_B, n_A) & & \\ \forall n_A, n_B : \mathsf{nonce}. & & \end{array}\right)^{\forall A}$$

Observe again that most declarations and types can be reconstructed. Notice in particular that, since in the second rule the arguments of L form a prefix of the arguments of L', the entire declaration for L can be synthesized from the type of L'.

The responder's actions are expressed in the following generic role. The first rule corresponds to lines (3) and (4) in the "usual notation", the second to lines (5) and (6), and the third to line (7). Observe again that most declarations can be automatically reconstructed.

$$\left(\begin{array}{lll}
\exists L : \mathsf{principal}^{(B)} \times \mathsf{principal} \times \mathsf{pubK}\ B^{(k_B)} \times \mathsf{privK}\ k_B \times \mathsf{nonce}. & & \\
\exists L' : \mathsf{principal}^{(B)} \times \mathsf{principal} \times \mathsf{pubK}\ B^{(k_B)} \times \mathsf{privK}\ k_B \times \mathsf{nonce}. & & \\
\forall k_B : \mathsf{pubK}\ B. & & \\
\forall k'_B : \mathsf{privK}\ k_B. & \mathsf{N}(\{\!|n_A\ A|\!\}_{k_B}) & \rightarrow \quad \mathsf{N}(B\ A) \\
\forall A : \mathsf{principal}. & & \quad\ L(B, A, k_B, k'_B, n_A) \\
\forall n_A : \mathsf{nonce}. & & \\
\forall \ldots & & \\
\forall k_{\mathsf{S}} : \mathsf{sigK}\ \mathsf{S}. & \mathsf{N}([k_A\ A]_{k_{\mathsf{S}}}) & \rightarrow \exists n_B\colon \mathsf{nonce}.\ \mathsf{N}(\{\!|n_A\ n_B|\!\}_{k_A}) \\
\forall k'_{\mathsf{S}} : \mathsf{verK}\ k_{\mathsf{S}}. & L(B, A, k_B, k'_B, n_A) & \quad\ L'(B, A, k_B, k'_B, n_B) \\
\forall k_A : \mathsf{pubK}\ A. & & \\
\forall \ldots & & \\
\forall k'_A : \mathsf{privK}\ k_A. & \mathsf{N}(\{\!|n_B|\!\}_{k_B}) & \rightarrow \quad . \\
\forall n_B : \mathsf{nonce}. & L'(B, A, k_B, k'_B, n_B) &
\end{array}\right)^{\forall B}$$

The last role in this protocol encompasses the actions of the server. Assuming that there is a single server, S, they can conveniently be expressed by the following anchored role, which consists of a single rule. The "usual notation" specification of this protocol makes use of this role twice: in lines (1) and (2) to create B's keys for A, and then in lines (4) and (5) for the dual operation.

$$\left(\begin{array}{lll}
\forall A, B : \mathsf{principal}. & \mathsf{N}(A\ B) & \rightarrow \quad \exists k_B : \mathsf{pubK}\ B. \quad \mathsf{N}([k_B\ B]_{k_{\mathsf{S}}}) \\
\forall k_{\mathsf{S}} : \mathsf{sigK}\ \mathsf{S}. & & \quad\ \exists k'_B : \mathsf{privK}\ k_B.
\end{array}\right)^{\mathsf{S}}$$

Upon receiving a message of the form $\mathsf{N}(A\ B)$, the server constructs a public/private key pair for principal B and notifies A by sending the signed message $\mathsf{N}([k_B\ B]_{k_{\mathsf{S}}})$. It should be observed how key generation is specified as existential quantification. The use of dependent types makes this process particularly elegant.

5 Neuman-Stubblebine Repeated Authentication Protocol

In this section, we devise an *MSR* specification of the Neuman-Stubblebine repeated authentication protocol [20]. Similarly to Kerberos, this protocol consists of two phases. In a first phases, a principal A negotiates a "ticket" with a server in order to use services provided by another principal B. In the second phases, A can reuse the ticket over and over to request this same service from B until the ticket expires.

5.1 Initialization Subprotocol

The Neuman-Stubblebine protocol [20] is intended to enable a principal A to repeatedly authenticate herself to another principal B. Typically, B provides a service that A is

interested in using repeatedly. Each time A intends to use this service, she authenticates herself to B by presenting a ticket he is expected to honor. The responder B marks the ticket with a timestamp and will accept it within some expiration period.

The initialization phase of this protocol involves an interaction with a server S to obtain the ticket, as well as its first use to request the service provided by B. The expected trace in the "usual notation" is given to the right of this text. In the first line, A manifests her intention to use B's service by sending him her identity and a nonce n_A. In the second line, B forwards this information to the server S together with his identity, a nonce of his own n_B, and a timestamp T_B. In the third line, the server constructs the ticket $\{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}}$ by combining A's name, a freshly generated key for communication between A and B, and B's time stamp. It is encrypted with the key $k_{B\mathsf{S}}$ that B shares with S so that A cannot modify it. The server also informs A of the extremes of the ticket in the message $\{B\ n_A\ k_{AB}\ T_B\}_{k_{A\mathsf{S}}}$ and forwards B's nonce to her. In the last line, A identifies herself to B by sending him the ticket and his nonce n_B encrypted with the newly created k_{AB}. Although part of no messages, this protocol assumes that B assigns a validity period to the ticket and will honor it until it expires. Therefore, upon receiving any message from A, B will verify if the timestamp is still valid.

1. $A \to B$: $A\ n_A$
2. $B \to \mathsf{S}$: $B\ \{A\ n_A\ T_B\}_{k_{B\mathsf{S}}}\ n_B$
3. $\mathsf{S} \to A$: $\{B\ n_A\ k_{AB}\ T_B\}_{k_{A\mathsf{S}}}\ \{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}}\ n_B$
4. $A \to B$: $\{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}}\ \{n_B\}_{k_{AB}}$

This initialization subprotocol is encoded in *MSR* by means of three roles, one for A, one for B, and one for S. We start by giving a specification of A's actions, reported in the following role:

$$\left(\begin{array}{lll} \exists L : \mathsf{principal} \times \mathsf{nonce}. & & \\ \qquad . & \to & \exists n_A\colon \mathsf{nonce}.\ \begin{array}{l}\mathsf{N}(A\ n_A)\\ L(A, n_A)\end{array} \\ \forall B\colon \mathsf{principal}. & & \\ \forall n_A, n_B : \mathsf{nonce}. & & \\ \forall k_{AB} : \mathsf{shK}\ A\ B. & \begin{array}{l}\mathsf{N}(\{B\ n_A\ k_{AB}\}_{k_{A\mathsf{S}}}\ X\ n_B)\\ L(A, n_A)\end{array} & \to \quad \begin{array}{l}\mathsf{N}(X\ \{n_B\}_{k_{AB}})\\ \mathsf{Ticket}_A(B, k_{AB}, X)\end{array} \\ \forall k_{A\mathsf{S}} : \mathsf{shK}\ A\ \mathsf{S}. & & \\ \forall X : \mathsf{msg}. & & \end{array}\right)^{\forall A}$$

The first rule is a straightforward encoding of line (1) of the "usual notation" description of this subprotocol. The more interesting second rule corresponds to lines (3) and (4). Notice that A is not entitled to observe the inner structure of the ticket. We express this fact by placing the variable X in the second component of the received message. Expanding this object as $\{Ak_{AB}T_B\}_{k_{B\mathsf{S}}}$ to expose its structure would violate the access control policy [6]. In the consequent of this same rule, A sends the message on line (4) of the informal presentation to B. She also needs to memorize some information to be able to reuse the ticket in the future, namely the ticket itself, the associated key k_{AB}, and B's identity. This is achieved by means of the memory predicate $\mathsf{Ticket}_A(\_, \_, \_)$. The type of this predicate is "$\mathsf{principal}^{(A)} \times \mathsf{principal}^{(B)} \times \mathsf{shK}\ A\ B \times \mathsf{msg}$" where the last argument corresponds to the ticket.

The responder's actions in this subprotocol are specified by the following role. Its two rules correspond to lines (1) and (2), and line (4) of the "usual notation" specifica-

tion above.

$$\left(\begin{array}{lll} \exists L : \mathsf{principal}^{(B)} \times \mathsf{principal} \times \mathsf{nonce} \times \mathsf{shK}\ B\ \mathsf{S} \times \mathsf{nonce} \times \mathsf{time}. & & \\ \forall A : \mathsf{principal}. & & \exists n_B \colon \mathsf{nonce}. \\ \forall n_A : \mathsf{nonce}. & \mathsf{N}(A\ n_A) & \rightarrow \quad \mathsf{N}(B\ \{A\ n_A\ T_B\}_{k_{B\mathsf{S}}}\ n_B) \\ \forall k_{B\mathsf{S}} : \mathsf{shK}\ B\ \mathsf{S}. & \mathsf{Clock}_B(T_B) & \quad\ \ \mathsf{Clock}_B(T_B) \\ \forall T_B : \mathsf{time}. & & \quad\ \ L(B, A, n_A, k_{B\mathsf{S}}, n_B, T_B) \\ \\ \forall \ldots & \mathsf{N}(\{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}}\ \{n_B\}_{k_{AB}}) & \\ \forall k_{AB} : \mathsf{shK}\ A\ B & L(B, A, n_A, k_{B\mathsf{S}}, n_B, T_B) & \\ \forall n_B : \mathsf{nonce}. & \mathsf{Valid}_B(A, T_B, T_V) & \rightarrow \quad \mathsf{Auth}_B(A, k_{AB}, T_B, T_{exp}) \\ \forall T_B, T_{now} : \mathsf{time}. & (T_{exp} = T_B + T_V) & \quad\ \ \mathsf{Valid}_B(A, T_B, T_V) \\ \forall T_V, T_{exp} : \mathsf{time}. & & \end{array}\right)^{\forall B}$$

Upon receiving the message $\mathsf{N}(A\ n_A)$, the responder B reads the timestamp T_B off his local clock, which we model by means of the memory predicate $\mathsf{Clock}_B(\_)$, of type "$\mathsf{principal} \times \mathsf{time}$". A specification of how local clocks are updated is outside of the scope of this paper. A technique akin to the handling of time in [17] is particularly appealing for the elegant form of automated reasoning about temporal entities it supports.

In the second rule of this role, B receives the ticket $\{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}}$ and the response $\{n_B\}_{k_{AB}}$ to the challenge he issued in the first rule by creating a nonce. He can clearly access the contents of the ticket and therefore verify this latter message. The timestamp must have the same value T_B memorized in the last argument of the role state predicate L in the first rule. The responder now assigns an expiration date to the ticket: he consults the memory predicate $\mathsf{Valid}_B(A, T_B, T_V)$ to decide on the length of time T_V it should be valid for (possibly on the basis of the initiator's identity A and the time of the day T_B it was requested), and then uses the arithmetic constraint $(T_{exp} = T_B + T_V)$ to compute its expiration date T_{exp}. The components of the ticket together with its expiration date are stored in the memory predicate $\mathsf{Auth}_B(\_, \_, \_, \_)$, of type "$\mathsf{principal}^{(B)} \times \mathsf{principal}^{(A)} \times \mathsf{shK}\ A\ B \times \mathsf{time} \times \mathsf{time}$".

Finally, we have a single rule that formalizes the actions of the server. Upon receiving a request from B, the server generates the shared key k_{AB}, constructs the ticket and the notification message for A, and transmits this information.

$$\left(\begin{array}{lll} \forall A, B : \mathsf{principal}. & & \exists k_{AB} : \mathsf{shK}\ A\ B. \\ \forall k_{A\mathsf{S}} : \mathsf{shK}\ A\ \mathsf{S}. & & \\ \forall k_{B\mathsf{S}} : \mathsf{shK}\ B\ \mathsf{S}. & \mathsf{N}(B\ \{A\ n_A\ T_B\}_{k_{B\mathsf{S}}}\ n_B) & \rightarrow \quad \mathsf{N}(\{B\ n_A\ k_{AB}\ T_B\}_{k_{A\mathsf{S}}} \\ \forall n_A, n_B : \mathsf{nonce}. & & \qquad \{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}} \\ \forall T_B : \mathsf{time}. & & \qquad n_B) \end{array}\right)^{\mathsf{S}}$$

5.2 Repeated Authentication Subprotocol

The second phase of the Neuman-Stubblebine protocol allows the initiator A to repeatedly use the ticket she has acquired in the first phase to access the service provided by B, as long as the ticket has not expired. It is expressed in the "usual notation" by the three-step subprotocol displayed to to the right of this text. In the first line, A generates a new nonce

1. $A \rightarrow B$: $n'_A\ \{A\ k_{AB}\ T_B\}_{k_{B\mathsf{S}}}$
2. $B \rightarrow A$: $n'_B\ \{n'_A\}_{k_{AB}}$
3. $A \rightarrow B$: $\{n'_B\}_{k_{AB}}$

n'_A and sends it to B together with the ticket $\{A\ k_{AB}\ T_B\}_{k_{BS}}$ she has acquired in the initial phase of the protocol. Upon receiving this message, B checks that the ticket is still valid, creates a nonce of his own n'_B, and transmits it to A in line (2) together with the encryption of n'_A with the key k_{AB} embedded in the ticket. In the last line, A sends B's nonce back after encrypting it with their shared key k_{AB}.

This subprotocol is formalized in *MSR* by means of the three roles below. The initiator's actions are expressed by the following generic role. In its first rule, corresponding to line (1) of the informal specification, A accesses the ticket she has stored in the memory predicate Ticket_ during the initialization phase. The second rule corresponds to the remaining lines of the "usual notation" specification.

$$\left(\begin{array}{llcl} \exists L : \mathsf{principal}^{(A)} \times \mathsf{principal}^{(B)} \times \mathsf{shK}\ A\ B \times \mathsf{nonce}. & & & \\ \forall B : \mathsf{principal}. & & & \mathsf{N}(n'_A\ X) \\ \forall X : \mathsf{msg}. & \mathsf{Ticket}_A(B, k_{AB}, X) & \rightarrow & \exists n'_A\colon \mathsf{nonce}.\ \mathsf{Ticket}_A(B, k_{AB}, X) \\ \forall k_{AB} : \mathsf{shK}\ A\ B. & & & L(A, B, k_{AB}, n'_A) \\ & & & \\ \forall \ldots & \mathsf{N}(n'_B\ \{n'_A\}_{k_{AB}}) & & \\ \forall n'_A, n'_B : \mathsf{nonce}. & L(A, B, k_{AB}, n'_A) & \rightarrow & \mathsf{N}(\{n'_B\}_{k_{AB}}) \end{array}\right)^{\forall A}$$

The actions of the service provider B are given by the following two generic roles. The first rule of the first of them captures lines (1) and (2) of the informal specification, while the second rule formalizes the remaining line.

$$\left(\begin{array}{llcl} \exists L : \mathsf{principal}^{(B)} \times \mathsf{principal}^{(A)} \times \mathsf{shK}\ A\ B \times \mathsf{nonce}. & & & \\ \forall n'_A : \mathsf{nonce}. & \mathsf{N}(n'_A\ \{A\ k_{AB}\ T_B\}_{k_{BS}}) & & \exists n'_B\colon \mathsf{nonce}. \\ \forall k_{BS} : \mathsf{shK}\ B\ \mathsf{S}. & \mathsf{Auth}_B(A, k_{AB}, T_B, T_{exp}) & & \mathsf{N}(n'_B\ \{n'_A\}_{k_{AB}}) \\ \forall A : \mathsf{principal}. & \mathsf{Clock}_B(T_{now}) & \rightarrow & \mathsf{Auth}_B(A, k_{AB}, T_B, T_{exp}) \\ \forall k_{AB} : \mathsf{shK}\ A\ B. & (T_{now} < T_{exp}) & & \mathsf{Clock}_B(T_{now}) \\ \forall T_B, T_{exp}, T_{now} : \mathsf{time}. & & & L(B, A, k_{AB}, n'_B) \\ & & & \\ \forall \ldots & \mathsf{N}(\{n'_B\}_{k_{AB}}) & & \\ \forall n'_B : \mathsf{nonce}. & L(B, A, k_{AB}, n'_B) & \rightarrow & . \end{array}\right)^{\forall B}$$

Upon receiving each new request, B checks that the ticket has not expired yet. This is achieved by means of the constraint $(T_{now} < T_{exp})$ that verifies whether the current time T_{now} (read from his "Clock_" memory predicate) is less than the ticket's expiration time T_{exp}.

Although there is no arm in keeping stale tickets, it is easy to write an *MSR* rule that removes expired tickets: whenever B notices that a ticket has expired (by means of the constraint $(T_{now} \geq T_{exp})$), he simply retracts the corresponding "Auth_" predicate.

$$\left(\begin{array}{lll} \forall A : \mathsf{principal}. & \mathsf{Auth}_B(A, k_{AB}, T_B, T_{exp}) & \\ \forall k_{AB} : \mathsf{shK}\ A\ B. & \mathsf{Clock}_B(T_{now}) & \rightarrow\ \ \mathsf{Clock}_B(T_{now}) \\ \forall T_B, T_{exp}, T_{now} : \mathsf{time}. & (T_{now} \geq T_{exp}) & \end{array}\right)^{\forall B}$$

This concludes our *MSR* specification of the Neuman-Stubblebine repeated authentication protocol. The two phases that constitute it have been modeled by providing two sets of roles. The connection between them is given by a number of memory predicates used by both the client A and the service provider B. It should be noted that this protocol lies outside of the scope of the previous version of *MSR* [9, 10], which did not provide any secure means to share data across different roles.

6 Conclusions and Future Work

In this paper, we have presented the syntax of *MSR*, a strongly typed specification language for security protocol. The typing infrastructure, based on the theory of dependent types with subsorting, yields elegant and precise formalizations. The underlying methodology does not prescribe a fixed set of types to be used for every protocol, but rather allows defining the objects (both types and term constructors) needed in each individual circumstance. This typing information is mostly used statically to discover simple but potentially harmful mistakes in a specification: for example, assuming appropriate declarations, type-checking would catch the unduly transmission of a long-term key in a network message. On the other hand, access control verification will point at attempts to use keys that do not belong to a principal. These two applications are presented in detail in [7] and [6], respectively. Static checks of this kind are particularly useful when modeling complex crypto-protocols.

Previous versions of *MSR* were mostly aimed at investigating decidability problems for crypto-protocols [9, 15] and at establishing the relative expressive power of different formalisms [11, 10]. The present work makes *MSR* usable as a specification language for a large class of security protocols thanks to the introduction of a few key constructs and a flexible typing infrastructure. Memory predicates, in particular, allow a principal to share data and control among different role instances. This makes our formalism applicable to protocols structured as a collection of subprotocols. Constraints allow instead factoring recurrent operations on complex domain as external modules, keeping in this way protocol specifications simple.

We have undertaken a formal study of various aspects of *MSR*. Besides the general discussion and case studies presented in this paper, its type-checking rules and their properties are analyzed in [7], access control is the subject of [6], while [8] starts examining parallel executions and implements different formulations of the Dolev-Yao intruder. A number of problems are however still open and subject of current investigation. First, a number of issues need to be solved in order to make *MSR* practical. In particular, a reliable type reconstruction algorithm is necessary to shelter users from the often tedious process of providing all the type declarations, and also to make formalizations reasonably sized. We are also extending our current collection of case studies to encompass not only the most common authentication protocols [12], but also complex schemes such as key management protocols for group multicast [4] and fair exchange protocols. Among other results, the formalization of these examples will allow us to experiment with numerous constructs and type layouts. We hope that this activity will enable us to extract useful specification techniques for the constructions needed in the formalization of a protocol. For example, we would like to be able to give a specification of hash functions from which the appropriate typing and access control rules can be automatically generated together with arguments that extend the validity of their various properties to these objects.

References

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

[2] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. Research Report 125, Digital Equipment Corp., System Research Center, 1994.

[3] D. Aspinall and A. Compagnoni. Subtyping dependent types. In E. Clarke, editor, *Proc. LICS'96*, pages 86–97, New Brunswick, NJ, 1996. IEEE Computer Society Press.

[4] D. Balenson, D. McGrew, and A. Sherman. Key management for large dynamic groups: One-way function trees and amortized initialization. Internet Draft (work in progres), draft-irtf-smug-groupkeymgmt-oft-00.txt, Internet Engineering Task Force (August 25, 2000).

[5] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society, Series A*, 426(1871):233–271, 1989.

[6] I. Cervesato. *MSR*, access control, and the most powerful attacker. Submitted to LICS'01, Boston, MA, 2001. `http://www.cs.stanford.edu/~iliano`.

[7] I. Cervesato. A specification language for crypto-protocol based on multiset rewriting, dependent types and subsorting. `http://www.cs.stanford.edu/~iliano`.

[8] I. Cervesato. Typed multiset rewriting specifications of security protocols. Submitted to *Proc. MFCSIT'00*, ENTCS. `http://www.cs.stanford.edu/~iliano`.

[9] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proc. CSFW'99*, pages 55–69, Mordano, Italy, 1999. IEEE/CS Press.

[10] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *Proc. CSFW'00*, pages 35–51, 2000.

[11] I. Cervesato, N. A. Durgin, M. Kanovich, and A. Scedrov. Interpreting strands in linear logic. In *Proc. FMCS'00*, Chigaco, IL, 2000.

[12] J. Clark and J. Jacob. A survey of authentication protocol literature. Technical report, Department of Computer Science, University of York, 1997. Web Draft Version 1.0 available from `http://www.cs.york.ac.uk/~jac/`.

[13] G. Denker and J. K. Millen. CAPSL Intermediate Language. In N. Heintze and E. Clarke, editors, *Proc. FMSP'99*, Trento, Italy, 1999.

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

[15] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In N. Heintze and E. Clarke, editors, *Proc. FMSP'99*, Trento, Italy, 1999.

[16] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proc. SSP'98*, pages 160–171, Oakland, CA, 1998. IEEE/CS Press.

[17] M. I. Kanovich, M. Okada, and A. Scedrov. Specifying real-time finite-state systems in linear logic. In *Proc. COTIC'98*, Nice, France, 1998. ENTCS 16(1).

[18] C. Meadows. The NRL protocol analyzer: an overview. *J. Logic Programming*, 26(2):113–131, 1996.

[19] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[20] B. C. Neuman and S. G. Stubblebine. A note on the use of timestamps as nonces. *Operating Systems Review*, 27(2):10–14, 1993.

[21] F. Pfenning. Refinement types for logical frameworks. In H. Geuvers, editor, *Proc. TYPES'93*, pages 285–299, Nijmegen, The Netherlands, 1993.

[22] P. F. Syverson. A different look at secure distributed computation. In *Proc. CSFW-10*, pages 109–115. IEEE Computer Society Press, 1997.