Planning Attacks to Security Protocols: Case Studies in Logic Programming

Luigia Carlucci Aiello[1] and Fabio Massacci[2]

[1] Dip. di Informatica e Sistemistica - Univ. Roma "La Sapienza" - Italy
aiello@dis.uniroma1.it
[2] Dip. di Ingegneria dell'Informazione - Univ. Siena - Italy
massacci@dii.unisi.it

Abstract. Formal verification of security protocols has become a key issue in computer security. Yet, it has proven to be a hard task often error prone and discouraging for non-experts in formal methods.
In this paper we show how security protocols can be specified and verified efficiently and effectively by embedding reasoning about actions into a logic programming language.
In a nutshell, we view a protocol trace as a plan to achieve a goal, so that protocol attacks are plans achieving goals that correspond to security violations. Building on results from logic programming and planning, we map the existence of an attack to a protocol into the existence of a model for the protocol specification that satisfies the specification of an attack. To streamline such way of modeling security protocols, we use a description language $\mathcal{AL}_{SP}$ which makes it possible to describe protocols with declarative ease and to search for attacks by relying on efficient model finders (e.g. the smodels systems by Niemela and his group). This paper shows how to use $\mathcal{AL}_{SP}$ for modeling two significant case studies in protocol verification: the classical Needham-Schroeder public-key protocol, and Aziz-Diffie Key agreement protocol for mobile communication.

1 Introduction

The design of secure communication protocols over an insecure medium such as the internet is a daunting task. Notwithstanding the increasingly sophisticated cryptographic primitives for digitally signing messages, encrypting documents, getting notarized timestamps on files etc., most security protocols are often found seriously flawed, even after they make their way up to become a standard.

Interestingly, most of the errors encountered in security protocols are logical error, which do not depend on the strength of the underlying cryptographic algorithms. For instance, if we receive a document digitally signed by Alice, we may think that Alice actually signed this message and sent it to us. However, depending on how the protocol is designed, it might well be that Alice never intended to send that document to us, but rather to a certain Bob, who never asked for it. It is just a malicious hacker who, by intercepting and subtly cutting and pasting messages together, has made such an awkward situation possible.

A.C. Kakas, F. Sadri (Eds.): Computat. Logic (Kowalski Festschrift), LNAI 2407, pp. 533–560, 2002.

An interesting collection of examples can be found in the book by Schneider [36, Chap. 3] or the classical articles by Abadi, Needham et al. [7,2].

This phenomenon is somehow surprising because security protocols are not overly complex: academic protocols are seldom above 6 messages, whereas deployed and widely used protocols such as Kerberos or TLS/SSL (the internet secure payment protocol) hardly go beyond twenty ((see [36, Chap. 3] or [7,10] for some characteristic examples), and even a "monster protocol" such the Secure Electronic Transaction protocol (SET) by Visa and Mastercard is substantially composed by 6 suites of "normal" protocols [32]. Nothing even comparable to the intrinsic complexity of current CPU design with billion of gates.

The hardness of the design task can be explained by two different factors. First, security protocols try to achieve difficult and sometimes unclear goals such as entity authentication, confidentiality, proof of receipts etc.in a substantially untrusted medium. It is often not clear what authentication means (see for instance Gollmann [16] vs Lowe [21]).

Second, the medium itself allows for unsuspected interactions, parallelism of actions and events that are difficult to foresee. Consider electronic payment protocols. Even though we think in terms of Alice willing to buy something, and Bob wishing to sell it, Alice and Bob are processes and not persons. Alice (the person) cannot simultaneously go to the grocery and to the bakery. Bob, the grocer, will hardly serve more than one person at a time. Alice cannot really run away with few kilos of pasta to avoid paying Bob. In contrast `Alice` (the shopping softbot of Alice) can practically simultaneously open a connection with `Bob`, the web server of a DVD movies e-shop and `Charlie`, her CD supplier. `Bob`, on his own, can have thousands of these connections who may all be in parallel and ought to be served with minimum delay. He cannot run after `Alice` to grab back his DVD if she "forgets" to pay. Moreover, their orders are channeled through many intermediate untrusted nodes.

It is therefore not a surprise that formal methods have gained such a widespread use in the analysis of security protocols [26,25]. Unfortunately, it turned out that formal verification itself its quite an intensive task as to discourage the application of a formal method by anybody else than the developer of the method itself. As correctly pointed out by Brackin, Meadows and Millen in [6]:

> It became evident that it was difficult for analysts other than the developers of the various techniques to apply them. One reason for this difficulty is that the protocols had to be re-specified formally for each technique and it was not easy to transform the published description of the protocol into the required formal system. Some tool developers began work on translators or compilers that would perform the transformation automatically. The input to any of such translators still requires a formally-defined language, but it can be made similar to the message-oriented protocol description that are typically published in articles, books and protocol standard documents.

The research efforts resulted in languages such as CAPSL [6] and CASPER [22], that are "front-ends" to formal systems, intermediate between formal specifica-

tions and the language used in the published descriptions of protocols. Indeed, these languages allow the operational specification of the protocol in terms of messages sent and received, and in terms of the operations made.

Nevertheless, these languages tie the hand of the protocol analyst and bind him to adopt the interpretations of protocol properties made by the designers of the compiler, who usually coincide with the developers of the target formal method. The security analyst must still buy, lock, stock and barrel, the definition of authentication, secrecy, non-repudiation etc. which are hardwired in the tool.

Moreover, the intermediate language proved to be too weak for specifying more complex protocols. For instance, in a public key infrastructure, each agent may have a certificate for its public key, and certificates usually have expiration dates. In the design of a security protocol, a designer may want to specify that the validity period of a certificate must be appropriate: a server may reject a document supposed to be valid for 10 years which is signed with a private key expiring in a month. To overcome this modeling difficulty, front-ends allow the specifiers to hack directly such constraints into the target formal language [22].

So, one would like to combine the best of two worlds: an operational description of a protocol and a declarative specification of its properties.

1.1 Our Contribution

We proposed $\mathcal{AL}_{SP}$ (Action Language for Security Protocols, see [8,9]), an executable specification language for representing security protocols, and checking the possibility of attacks. The intuitions are the following:

- The operational description of a security protocol (what security designers would like) can be quite naturally cast into the general framework of an AI planning problem with simple actions such as sending and receiving. Checks on the protocol actions (such as verifying expiration dates on certificates) are naturally cast into action preconditions. Attacks are just plans to reach security violations.
- The preconditions for the executions of protocol actions and the properties of a protocol should be easily specifiable, as declaratively as possible.
- Modeling nonmonotonic behaviors is essential in this framework, as we may want to say that if something is not specified then it is false by default. Once we modeled the capability of an intruder and – with this capability – no attack is found, then we would like to conclude that no attack exists.
- The number of objects and agents would potentially be unbounded; therefore the language must allow for free variables and function symbols to describe properties of objects, compound objects (such as concatenation of messages) and agents, without forcing the analyst to hardwire each particular object into a particular message of particular protocol steps.
- As soon as we set a bound on the number of objects and agents that are around, it should be possible to check for attacks with fast state-of-the-art systems in a automatic way (that is, debugging should be mostly automatic).

– Decidability and expressiveness of the language matter more than complexity, because we do not verify a protocol on-line (whereas a robot must move in the real world), but we want to specify complex protocols without bit-oriented programming in logic, process algebras or other formal languages.

For all the above reasons, logic programming stands out as an extremely nice formalism upon which to build our specification language $\mathcal{AL}_{SP}$. However, for our formalization of security protocols not everything of logic programming can be bought; therefore $\mathcal{AL}_{SP}$ borrows selected features from logic programming .

$\mathcal{AL}_{SP}$ is based on logic programming with stable model semantics ($\mathcal{LP}_{SM}$) [3,14]. This choice is motivated by three properties guaranteed by $\mathcal{LP}_{SM}$:

– if a fact is true in a stable model, there is a justification for it and no circular justification is allowed;
– if something is not explicitly said, it is false by default;
– it is possible to say that some facts *must* be true in a stable model, and other facts *may* be true in it.

This is particularly appropriate to represent actions and changes, which is needed to model security. For example, consider modeling an intruder. If the intruder decrypted some messages, we want a well-founded justification for the intruder to know the key. Moreover, we want to say that the intruder may disrupt each step of the protocol, but he is not obliged to; he may disrupt some steps and let others remain unchanged.

Logic programming languages — hence $\mathcal{AL}_{SP}$ — allow for a declarative formalization of the operational behavior of the protocol and the possible attacks of an intruder. As mentioned, we borrow this formalization from robotic planning: out of a declarative specification of the world, the proof of the existence of a model for a goal state can be easily transformed into a plan (i.e. a sequence of actions) to achieve it. Conversely, the non-existence of a plan can be checked as an un-satisfiability problem. If no model for a goal state can be found, then we have proven that there is no plan that achieves it.

To achieve decidability for bounded model checking we impose some restriction on the form that free variables occurring in rules may have. Thus we obtain domain restricted logic programs. When a bound on the protocol resources, agents and time is set, we obtain a finite ground model of our specification.

Finally, we search for attacks on the finite ground representation using efficient model finders for the stable model semantics [30,31] which can handle hundreds of thousands of rules in few seconds. This makes $\mathcal{AL}_{SP}$ executable.

1.2 Plan of the Paper

In this paper we show how to model in $\mathcal{AL}_{SP}$ two important case case studies: the classical Needham-Schroeder public-key protocol [29,7] and the Aziz-Diffie key agreement protocol for mobile communication [4,38].

Thus, we first introduce some background on the logic approach to planning (Section 2). Then we shortly introduce the language $\mathcal{AL}_{SP}$ (Section 3) and sketch

how it can be used in practice (Section 4). Then we illustrate the formalization in $\mathcal{AL}_{SP}$ of the Needham-Schroeder protocol (Section 5) and the Aziz-Diffie key agreement protocol (Section 6). We conclude the paper with a brief comparison with related works (Section 7).

2 Logical Approach to Planning

Planning is a research area in AI aiming at the construction of algorithms — called planners — that enable an agent (a robot or a "softbot") to synthesize a course of actions that will achieve its goals (see Weld [39] for a recent survey).

A planner has to be provided with a *background theory*, i.e. a description of generally known properties about the world, and with a *planning problem*:

1. a description of the initial state of the world;
2. a description of the goal state the agent has to achieve;
3. a description of the possible actions that can be performed by the agent. This is often called *domain theory* or *action theory*.

The solution of the problem (if one exists) is a *plan*, i.e. a sequence of actions that, when executed in any world satisfying the initial state description, will achieve the goal.

Actions may have *preconditions*, i.e. requirements to be satisfied in order for the action to be executable. Actions modify the current status of the world; this is described by stating the "*causal laws*", i.e. how they affect the values of predicates and functions, in the form of the so called *effect axioms*. In addition, the "*laws of inertia*" for the domain are to be stated, i.e. which values are unaffected by each action, so they persist through its execution.

A planning problem in the context of security protocols, where agents exchange messages and are subject to attacks by intruders, is the following:

1. the initial state is described in terms of the keys known to agents and the messages already exchanged (typically none), at the time the protocol starts;
2. the goal state is an unwanted situation where some security violation has occurred (e.g. A receives a message allegedly from B who actually never sent it to A.);
3. actions are exchanges of messages among agents.

A solution of the planning problem, if any, is a sequence of actions leading to an unwanted situation, and thus a plan is an attack to the security of the protocol.

The background theory, in this case, includes the description of how messages are composed and decrypted by agents, the properties of keys, how knowledge is attained by the agents participating in the protocol, etc.

Causal laws and laws of inertia can be cast as constraints on the possible sets of predicates that are admissible for consecutive times t and $t+1$. For instance, if the action predicate $says(A, B, M, t)$ is true, then $\mathsf{said}(A, B, M, t+1)$ is true.

In this way, the planning problem becomes the problem of finding a time t such that $Goal(t)$ holds, where $Goal(t)$ is the conjunction of the (relevant) formulas true at time t such that all constraints are satisfied.

The relation between logic programming on the one side, and reasoning about action and planning on the other side, has been studied quite extensively, e.g. by Gelfond and Lifschitz [15], Denecker et al [11] in the context of the Event Calculus, or Subrahmanian and Zaniolo [37]. Kautz and Selman in [17] proposed to cast a planning problem as a model finding problem via an encoding of plans as propositional formulas. We do not adopt their encoding, but share with their proposal the idea that planning can be solved as model finding. Following ideas of Nebel and coworkers [12] and Niemelä [30], our basic intuition is to limit the size of plans (by considering plans whose length l is less than n for some fixed n) and then encode the planning problem as a satisfiability problem of logic programs, by encoding each causal and inertial constraint as a logic programming rule. If we find a stable model for the goal and all formulae, then we have a plan.

Plans can be generated using logic programs with the stable model semantics [14,3]. Stable models capture the two key properties of solution sets to logic programs: they are minimal and grounded, i.e. each atom in a stable model has a justification in terms of the program. Minimality and groundedness make logic programming with stable model semantics ($\mathcal{LP}_{SM}$) particularly suited to modeling actions and change, in particular in security problems, where we want to model exactly what happened (i.e. we do not want to leave room for unwanted models), and where everything has a justification in the model. For example, if an intruder has got a secret key, there is an explanation in the model in terms of actions that he has performed, it cannot have happened for other reasons not captured by the stable model itself.

Even though computing stable models has been proved NP-complete, the techniques for computing stable models for ground programs have advanced and there are systems that can cope with tens of thousands of rules. The system `smodels`, developed by Niemelä and his group [30,31], is one of them.

In order to introduce it, we present some more notions. Logic programs with variables can be given a semantics in terms of stable models. The stable models of a normal logic program P with variables are those of its ground instantiations P_H with respect to its Herbrand universe. If logic programs are function free, then an upper bound on the number of instantiations is rc^v, where r is the number of rules, c the number of the constants, and v the upper bound on the number of distinct variables in each rule. Hence, to keep the Herbrand Universe of a logic program finite, we need to restrict variables to range over finite domains.

Programs where variables are sorted are domain restricted to the domain of the sort predicates. This property holds for the logic programming language $\mathcal{AL}_{SP}$. Functions are allowed in $\mathcal{AL}_{SP}$ programs, but domain restrictedness is kept by imposing that arguments of functions range over finite domains.

Domain restrictedness is a limitation that still leaves logic programs with expressive power to deal with interesting applications. At the same time, with this limitation, the grounding problem and the search for stable models can be solved efficiently, in particular if the domain is nonrecursive, i.e. D does not contain predicates that are recursively defined in P. $\mathcal{AL}_{SP}$ enjoys this property.

`smodels` [31,30] is an implementation of $\mathcal{LP}_{SM}$, for range restricted function free normal programs. It consists of two modules: the proper `smodels`, which implements $\mathcal{LP}_{SM}$ for ground programs and `parse`, the grounding procedure, or better `lparse` a more efficient parsing module which works for domain restricted programs with nonrecursive domains. `lparse` automatically detects domain predicates and deals with them very efficiently. In addition, it has some built in arithmetic functions.

The stable model semantics for ground programs as implemented in `smodels` is a bottom-up backtracking search, where only the negative atoms in the program contribute to an increase of the search space, hence it is very efficient.

`smodels` offers the possibility of including a "choice" rule into logic programs:

$$\{c\} \longleftarrow a, b$$

It reads as: if a and b are both true, then c *may* be in the stable model, but this is not mandatory. Actually, a program containing the choice rule can in fact be translated into a normal program. The language $\mathcal{AL}_{SP}$ borrows the choice rule from `smodels`, as it is useful when representing security problems. For instance, it allows us to easily represent the fact that an agent may send a message, but he is not compelled to do it.

3 The Language $\mathcal{AL}_{SP}$

As already said, $\mathcal{AL}_{SP}$ is logic programming with negation as failure and stable model semantics. We here illustrate the primitives $\mathcal{AL}_{SP}$ offers, i.e. the logic programming rules common to the representation of (almost) all security protocols. $\mathcal{AL}_{SP}$ provides the user with *basic sort predicates* to characterize the basic components of protocols' specifications:

- $\mathsf{ag}(A)$ denotes that A is an *agent*
- $\mathsf{nonce}(N)$ denotes that N is a *nonce*[1]
- $\mathsf{key}(K)$ denotes that K is a *key*[2]
- $\mathsf{timestamp}(TS)$ denotes that TS is a *timestamp*.

$\mathcal{AL}_{SP}$ provides the user with *constructors for messages*. Some "classical" constructs are pairing, encryption, hashing, and exclusive-or, which we represent in BAN-like notation [7]:

- $\{M\}_K$ is the encryption of M with the key K;
- $M_1 \| M_2$ is the concatenation of M_1 with M_2;
- $h(M_1)$ is the hash of message M_1;
- $M_1 \oplus M_2$ is the bit-wise xor of M_1 and M_2.

[1] Nonce is a security jargon for "Number Used Once"; typically, an unguessable random number.

[2] We may have different keys such as shared, private or agreement keys. We distinguish them with additional predicates.

A *special sort predicate* is $\mathsf{msg}(M)$, which denotes that M is a valid (sub)*message* that may appear in a run of the protocol. The predicate $\mathsf{msg}(\cdot)$ specifies how messages are built with message constructors from basic components.

The direct approach would be using $\mathsf{msg}(\cdot)$ and defining messages inductively with constructors. For instance

$$\mathsf{msg}(M_1 \| M_2) \longleftarrow \mathsf{msg}(M_1), \mathsf{msg}(M_2)$$

could be a rule for inductively defining message concatenation. Unfortunately, inductively defined predicates with function symbols have infinitely many ground instances. In our application, we do not need inductive definitions for $\mathsf{msg}(\cdot)$: it is sufficient to use messages that may occur as submessages in a possible run of a protocol. For instance, the concatenation of thousands of nonces will never appear in the Needham-Schroeder public key protocol, and – if it does – it will be ignored by all honest agents. In most protocols, even complex ones, the format and number of valid messages is fixed[3] and can be expressed by few applications of the constructors to elements of the basic types (see [27]).

Therefore we impose two constraints:

Definition 1. *A* basic sort predicate is admissible *for $\mathcal{AL}_{SP}$ if it is not recursively defined by logic programming rules.*

Definition 2. *A* logic programming rule with the special sort predicate $\mathsf{msg}(\cdot)$ in the head is admissible *for $\mathcal{AL}_{SP}$ only if basic sort predicates alone occur in the body of the rule.*

If we have finitely many basic objects (agents, nonces, etc.), then we have finitely many messages in $\mathcal{AL}_{SP}$ and therefore we have finite models. This is the only part of $\mathcal{AL}_{SP}$ specifications in which we forbid inductively defined predicates.

The trade off is that the rules defining $\mathsf{msg}(\cdot)$ depend on the particular protocol we are analyzing. We must define each submessage in terms of the atomic components. This tedious part of $\mathcal{AL}_{SP}$ specifications has been automated [18].

$\mathcal{AL}_{SP}$ has predicates for defining *properties of messages*:

- $\mathsf{part}(M_1, M)$ denotes that M_1 is a submessage of M;
- $\mathsf{invKey}(K, K_I)$ denotes that K_I is the inverse of K;
- $\mathsf{symKey}(K)$ denotes that K is a symmetric key;
- $\mathsf{sharedKey}(K, A, B)$ denotes that K is a (symmetric) key shared between A and B;
- $\mathsf{asymKeyPair}(Kpriv, Kpub)$ denotes that $Kpriv$ and $Kpub$ are an asymmetric key pair.

Other predicates may be introduced on demand.

Next, we have predicates for *knowledge and ability to compose messages.* From now on we must introduce *time* as an additional argument.

[3] The recursive protocol analyzed in [33,35] is an exception.

- $knows(A, M, T)$ denotes that agent A knows message M at time T;
- $\mathsf{synth}(A, M, T)$ denotes that A can construct message M at time T.

Then we have predicates for *actions*:

- $says(A, B, M, T)$ denotes the attempt[4] by A to send message M to B at time T;
- $gets(B, M, T)$ denotes the receipt[5] of message M by B at time T;
- $notes(A, M, T)$ denotes the storage of message M by A at time T.

These actions are present in the inductive theory of traces by Paulson and Bella [33,5]. Together with the predicate $knows(A, M, T)$, they are the only predicates typeset in italics, as they are the only ones whose truth value we need to know for extracting attacks from stable models.

We use the predicates $\mathsf{said}(A, B, M, T)$, $\mathsf{got}(B, M, T)$, and $\mathsf{noted}(A, M, T)$, with the obvious meaning that they are true when the corresponding action happened some time before T. We prefer this solution wrt the explicit temporal operators as for instance proposed by Syverson and Meadows [38] because it leads to simpler semantics and gives us the flexibility to explicitly axiomatize when and how information about past runs of the protocol carries on into the current run.

4 $\mathcal{AL}_{SP}$ at Work

In order to verify the security of protocols, building on the above primitives, we write specifications in $\mathcal{AL}_{SP}$, and then use the `smodels` systems, according to the following steps:

- we use the $\mathcal{AL}_{SP}$ specification of the general background and action theories;
- we write the $\mathcal{AL}_{SP}$ specification of the protocol dependent part, with choice rules for representing the correct execution of the protocol;
- we define a rule for the security property (attack) we want to check;
- we merge the three specifications, set the maximum execution time of the protocol to t_{max}, and a bound on the number of basic objects (agents, nonces, etc.);
- we use `lparse` to obtain the finite ground representation of $\mathcal{AL}_{SP}$ specifications;
- we use `smodels` to look for a stable model of the ground system.

If no stable model exists, then the attack does not exist for all (possibly parallel) interleaved runs of the protocol up to t_{max}.

If a stable model is found, then we look for the atoms representing actions ($says(A, B, M, T)$, $gets(B, M, T)$, $notes(A, M, T)$) that are true in the model: they give us the sequence of (parallel) actions that constitute the attack.

[4] Attempt because the spy might intercept the message and the intended recipient might never see it.

[5] We only specify the recipient in the "get" action as the sender is unreliable. See also [5,33] for a discussion of this modeling choice.

If we are looking for confidentiality attacks, then we must gather the atoms $knows(spy, M, t_{max})$ that are true in the model. They represent the knowledge of the intruder at the end of the protocol.

To speed up the search, we may add extra constraints on the rules that describe the protocol, for instance by limiting the possibility of agents to receive or send messages etc.

In a nutshell, search can be constrained by adding more determinism to the protocol description. Provided these constraints are reasonable and correspond to the "natural" implementation of the protocol, they do not preclude the possibility of finding attacks. Some of these optimizations are described in the subsequent case studies.

5 Needham-Schroeder Public-Key

The Needham-Schroeder Public-Key protocol is a classical workbench for formal analysis. It was introduced by Needham and Schroeder in the 70s and its aim is to allow two agents to exchange two independent secret numbers.

The basic idea is simple: Alice wants to talk to Bob, but doesn't know him directly. So she contacts a trusted server to provide her with the public key of Bob. Then, by using the protocol, Alice and Bob get hold of two shared secrets in the form of nonces which can then be used for subsequent communication[6].

The protocol is interesting because it has been formally analyzed using a belief logic [7], but a substantial weakness[7] has only been detected using model checking within process algebra [19].

The intuitive description of the protocol is the following:

1. Alice contacts Sam, a trusted server, who knows the public key of Bob;
2. Sam replies by sending Bob's public key signed with his private key;
3. Alice sends Bob a fresh nonce and her name encrypted with Bob's public key;
4. Bob reads the message and contacts Sam to get Alice's public key;
5. Sam replies by sending Alice's public key signed with his private key;
6. Then Bob creates a fresh nonce and sends it back to Alice together with her own nonce, all encrypted with Alice's public key;
7. Alice checks her nonce, and then sends back Bob's nonce encrypted with his public key, to show him that she has got hold of it.

[6] To be precise, this goal has been ascribed to the protocols by Needham, Abadi and Burrows in [7]. The original paper [29] uses the more vague term of authentication.

[7] Given the rather vague terms used in the original paper, it has been a subject of an intense debate whether Lowe's "attack" is indeed an attack (see [16]).

Formally, it corresponds to the following:

$$
\begin{array}{l}
A \longrightarrow S : A \| B \\
S \longrightarrow A : \{pK(B) \| B\}_{sK(S)} \\
A \longrightarrow B : \{N_a \| A\}_{pK(B)} \\
B \longrightarrow S : B \| A \\
S \longrightarrow B : \{pK(A) \| A\}_{sK(S)} \\
B \longrightarrow A : \{N_a \| N_b\}_{pK(A)} \\
A \longrightarrow B : \{N_b\}_{pK(B)}
\end{array}
$$

Leaving outside the steps involving Sam, which just distributed public keys, the security of the protocol rests upon the following reasoning (borrowed from [7]):

- if Alice sent Bob a number (the nonce N_a) that she has never used for that purpose before, and if she receives from Bob something that depends on knowing that number (the message $\{N_a \| N_b\}_{pK(A)}$), then she ought to believe that Bob's message originated recently, in fact after hers.
- if Alice believes that $pK(B)$ is Bob's public key, then she should believe that any message encrypted as $pK(B)$ can only be decrypted by Bob;
- if Alice believes that her private key $sK(A)$ has not been compromised then any message encrypted with $pK(A)$ can only be decrypted by her;
- thus, upon receiving $\{N_a \| N_b\}_{pK(A)}$, Alice can be assured that Bob is alive, and only her and Bob know N_a and N_b.

The same reasoning can be done for Bob, when he receives $\{N_b\}_{pK(B)}$.

Thus, "each principal knows the public key of the other, and has the knowledge of a shared secret which he believes the other will accept as being shared only by the two principals. [...] From this point, A and B can continue to exchange messages using N_a, N_b and public-key encryption. In this way they can transfer data or other keys securely" [7].

As Lowe has shown [19], this is not exactly the case. There are runs of the protocol where Bob believes that he has been running the protocol with Alice, whereas Alice has been running the protocol with Charlie and has never heard about Bob.

For simplicity sake, as in Lowe's analysis, we omit messages to and from S.

The first step is the specification in $\mathcal{AL}_{SP}$ of the valid messages of the protocol, to guarantee that the $\mathcal{AL}_{SP}$ specification is admissible (see Definition 2 or [8,9] for further discussion). To this extent, we must define each sub-message in terms of the atomic components:

$$
\begin{array}{l}
\mathsf{msg}(\{N \| A\}_K) \longleftarrow \mathsf{key}(K), \mathsf{isPubKey}(K), \mathsf{nonce}(N), \mathsf{ag}(A) \\
\mathsf{msg}(\{N \| N'\}_K) \longleftarrow \mathsf{key}(K), \mathsf{isPubKey}(K), \mathsf{nonce}(N), \mathsf{nonce}(N') \\
\mathsf{msg}(\{N\}_K) \longleftarrow \mathsf{key}(K), \mathsf{isPubKey}(K), \mathsf{nonce}(N) \\
\mathsf{msg}(N \| A) \longleftarrow \mathsf{nonce}(N), \mathsf{ag}(A) \\
\mathsf{msg}(N \| N') \longleftarrow \mathsf{nonce}(N), \mathsf{nonce}(N') \\
\mathsf{msg}(N) \longleftarrow \mathsf{nonce}(N) \\
\mathsf{msg}(A) \longleftarrow \mathsf{ag}(A)
\end{array}
$$

This step is entirely mechanical and tedious. To avoid it, a translator from protocol descriptions in CASPER into $\mathcal{AL}_{SP}$ has been recently implemented at the Department of Informatica e Sistemistica [18] and a graphical interface is under way.

Next, we need rules to model the ability of agents to manipulate messages. We start by inductively defining the *parts of a message* on the basis of our constructors:

$$\begin{aligned}
&\mathsf{part}(M, M) \longleftarrow \mathsf{msg}(M)\\
&\mathsf{part}(M, M_1 \| M_2) \longleftarrow \mathsf{msg}(M), \mathsf{msg}(M_1), \mathsf{msg}(M_2),\\
&\qquad\qquad\qquad\qquad \mathsf{part}(M, M_1)\\
&\mathsf{part}(M, M_1 \| M_2) \longleftarrow \mathsf{msg}(M), \mathsf{msg}(M_1), \mathsf{msg}(M_2),\\
&\qquad\qquad\qquad\qquad \mathsf{part}(M, M_2)\\
&\mathsf{part}(M, \{M_1\}_K) \longleftarrow \mathsf{msg}(M), \mathsf{msg}(M_1), \mathsf{key}(K),\\
&\qquad\qquad\qquad\qquad \mathsf{part}(M, M_1)
\end{aligned}$$

In the sequel, for sake of readability, we omit all sort predicates and use the convention that A, B, C, etc. stand for agents, N stands for nonces, T stands for time, K stands for keys, and M stands for messages.

Keys have particular properties, which can be modeled provided the resulting rules are admissible according to Definition 1. For instance, we need to state that

1. public and private keys go in pairs,

$$\begin{aligned}
&\mathsf{isPubKey}(Kp) \longleftarrow \mathsf{asymKeyPair}(Ks, Kp)\\
&\mathsf{isPrivKey}(Ks) \longleftarrow \mathsf{asymKeyPair}(Ks, Kp)
\end{aligned}$$

2. each private key is the inverse of the corresponding public key, and vice versa,

$$\begin{aligned}
&\mathsf{invKey}(Ks, Kp) \longleftarrow \mathsf{asymKeyPair}(Ks, Kp)\\
&\mathsf{invKey}(Kp, Ks) \longleftarrow \mathsf{asymKeyPair}(Ks, Kp)
\end{aligned}$$

3. each agent has a public/private key pair.

$$\mathsf{asymKeyPair}(sK(A), pK(A)) \longleftarrow \mathsf{ag}(A)$$

Since Herbrand Equality (i.e. the unique name assumption) is implicit in our model, we obtain that each agent's public (private) key is different from all other asymmetric keys. We can explicitly impose these constraints:

$$\begin{aligned}
&\longleftarrow \mathsf{asymKeyPair}(Ks, Kp), \mathsf{asymKeyPair}(Ks, Kp'), Kp \neq Kp'\\
&\longleftarrow \mathsf{asymKeyPair}(Ks, Kp), \mathsf{asymKeyPair}(Ks', Kp), Ks \neq Ks'
\end{aligned}$$

If the above rules are the only rules about asymmetric keys, by stable model semantics we have that a public key cannot be another agent's private key. This constraint (which is not necessarily true for all crypto-systems, e.g. RSA [36]) can also be added:

$$\begin{aligned}
&\longleftarrow \mathsf{asymKeyPair}(Ks, Kp), \mathsf{asymKeyPair}(Kp, Kp')\\
&\longleftarrow \mathsf{asymKeyPair}(Ks, Kp), \mathsf{asymKeyPair}(Ks', Ks)
\end{aligned}$$

Shared keys can be modeled in a similar fashion (see [8,9]).

Next, we define what an agent can infer from other messages and how he can construct messages; that is we *model knowledge*. Most of these rules are protocol independent. The reader may find a comprehensive description in [8,9].

For instance, we may need to specify that if you get something then you obviously know it.

$$knows(A, M, T) \longleftarrow \mathsf{got}(A, M, T)$$

Beside sending and receiving messages, we need rules to peel constructors off. For the N-S protocol, we just need rules for concatenation and encryption:

$$\begin{aligned} knows(A, M_1, T) &\longleftarrow knows(A, M_1 \| M_2, T) \\ knows(A, M_2, T) &\longleftarrow knows(A, M_1 \| M_2, T) \\ knows(A, M, T) &\longleftarrow knows(A, \{M\}_K, T), \\ &\qquad knows(A, K_I, T), \mathsf{invKey}(K, K_I) \end{aligned}$$

In some cases concatenation is modeled as an associative operator. This can be captured by the following rule:

$$knows(A, (M_1 \| M_2) \| M_3, T) \longleftarrow knows(A, M_1 \| (M_2 \| M_3), T)$$

In first-order logic programs, this rule may lead to non termination. We would avoid this problem, as we use the ground representation for actual search.

However, we drop the rule altogether as it is not appropriate for modeling well-implemented protocols: ISO Distinguished Encoding Rules (DER) distinguishes precisely between the concatenation $A\|(B\|C)$ and the concatenation $(A\|B)\|C$ even from a bitwise point of view. Since the formal verification of badly implemented protocols have little sense we decided to leave it out.

Then we can model message composition as follows:

$$\begin{aligned} \mathsf{synth}(A, M, T) &\longleftarrow knows(A, M, T) \\ \mathsf{synth}(A, \{M\}_K, T) &\longleftarrow \mathsf{synth}(A, M, T), \\ &\qquad knows(A, K, T) \\ \mathsf{synth}(A, M_1 \| M_2, T) &\longleftarrow \mathsf{synth}(A, M_1, T), \\ &\qquad \mathsf{synth}(A, M_2, T) \end{aligned}$$

Now we can build the first part of the *protocol independent action theory* in $\mathcal{AL}_{SP}$. Again, some successor state axioms are identical for all protocols and we refer to [8,9] for further details. For instance, we have axioms to model what happens when a message is received:

$$\begin{aligned} \mathsf{got}(B, M, T+1) &\longleftarrow gets(B, M, T) \\ \mathsf{got}(B, M, T+1) &\longleftarrow \mathsf{got}(B, M, T) \end{aligned}$$

The first axiom models a causal law (getting something now causes it to be got afterwards) and the second one models the law of inertia (once you got something, you got it). We need identical axioms for the $notes(A, M, T)$, $says(A, B, M, T)$, etc.

We have not found the need for "forgetful" agents in the protocols we have seen so far [10], thought there might be protocols for which we may need to modify this law of inertia.

Next, we define the preconditions for getting and receiving messages that are independent of the protocol that we want to analyze. For instance, message reception:

$$\{gets(B, M, T)\} \longleftarrow says(A, B, M, T)$$

We use the choice rule (see Section 2) to specify that if A attempts to send a message M to B at time T then B *may* receive it. There are stable models where the message is delivered (the normal execution of the protocol) and stable models where B does not receive the message. A possible interpretation is that in these latter models, the intruder has intercepted the message, or that the communication lines went down. Thus, we do not need to explicitly model the action of message interception as done in [24,27,38].

Modeling the intruder according the classical Dolev-Yao model [13] is simple: he may get any message in transit and he may say any message (but in both cases he needs not to). We do not need to model the ability of intercepting messages as we have already modeled faulty channels by specifying that messages may not be delivered. Therefore, there will be stable models of the protocol where the intruder does nothing (the correct runs) and stable models where he is busy. Formally

$$\{gets(spy, M, T)\} \longleftarrow says(A, B, M, T)$$
$$\{says(spy, B, M, T)\} \longleftarrow \mathsf{synth}(spy, M, T)$$

As we mentioned, we may add more constraints on the action preconditions to cut meaningless attacks and cut the search in the verification stage. For instance, we may strengthen the action preconditions:

$$\{gets(spy, M, T)\} \longleftarrow says(A, B, M, T), A \neq spy, B \neq spy$$
$$\{says(spy, B, M, T)\} \longleftarrow \mathsf{synth}(spy, M, T), B \neq spy$$

In security protocols the notion of *freshness* plays a key role. The whole reasoning in the Needham-Schroeder protocol rests on the nonces being freshly generated. To model freshness, we introduce at first a fluent $\mathsf{used}(N, T)$ which is true when message M has been used by somebody before time T. We use the fluent $\mathsf{usedPar}(M, T)$ when two agents try to use the same message in parallel, or when an agent tries to send the same message to two different agents in parallel. Out of these two axioms we have rules to denote when something is fresh, i.e. when the fluent $\mathsf{fresh}(M, T)$ holds. Since the treatment of freshness is a bit subtle, we refer to [8,9] for further details.

Finally, we are left with the rules specifying the *protocol's action*. We just need to "copy" them from the protocol description making just explicit all freshness checks:

$$\{says(A, B, \{N_a \| A\}_{pK(B)}, T)\} \longleftarrow \mathsf{fresh}(N_a, T)$$
$$\{says(B, A, \{N_a \| N_b\}_{pK(A)}, T)\} \longleftarrow \mathsf{got}(B, \{N_a \| A\}_{pK(B)}, T), \mathsf{fresh}(N_b, T)$$
$$\{says(A, B, \{N_b\}_{pK(B)}, T)\} \longleftarrow \mathsf{said}(A, B, \{N_a \| A\}_{K_b}, T), \mathsf{got}(A, \{N_a \| N_b\}_{pK(A)}, T)$$

As we have eliminated the exchanges with S, we have directly used the functions $pK(A)$ and $pK(B)$ to identify the corresponding public keys. In the full protocol, where agents do not know each other's public keys in advance, the check that the public key is appropriate must be made explicit:

$$\{says(B, A, \{N_a \| N_b\}_{K_a}, T)\} \longleftarrow \mathsf{got}(B, \{N_a \| A\}_{K_b}, T), \mathsf{isPubKey}(K_b), \mathsf{invKey}(K_b, sK(B)) \mathsf{isPubKey}(K_a), \mathsf{got}(B, \{K_a \| A\}_{sK(S)}, T), \mathsf{fresh}(N_b, T)$$

Once again, we can restrict the search by imposing further operation constraints on each action precondition. It is up to the security analyst to decide which checks are reasonable, depending on the way he thinks the protocol will be implemented. For instance, we can impose that an agent never knowingly sends a message to himself by setting:

$$\{says(A, B, \ldots, T)\} \longleftarrow \ldots A \neq B$$

for all the above rules.

This is a typical limitation common to all formal approaches to the verification of security protocols. Obviously, Alice might be fooled into running the protocol with herself (a classical "mirror attack"), but this typically happens because she is running two protocol instances in parallel, one instance as initiator and one instance as responder. So she sends her messages to Bob, but Bob never sees them: the intruder intercepts the messages and feeds them back to Alice, who might then believe that they come from Bob. These attacks are not prevented by this optimization.

These additional constraints substantially reduce the size of the ground program. Since each constraint eliminates some possible models from consideration, its introduction must be evaluated on a case by case basis, to be sure that we only eliminate models which do not correspond to meaningful attacks.

Last but not least, is the *goal of the protocol*. This depends on what the security analyst is interested in verifying. The procedure to specify an attack to a confidentiality or authentication goal is simple [8,9]:

1. we consider the view point of the agent for which the property must be verified;
2. we list all messages that he has sent or received up to the point of the protocol (typically the end) that we want to verify;
3. for authentication properties, we add the *negation* of the event(s) that we expected to have happened if the protocol was correct (e.g. Bob should have got some message but in reality has not);

4. for confidentiality properties, we say that the spy knows the messages that ought to have remained secret;
5. add additional checks that the security analyst may deem necessary (e.g. constraints on time or on nonces).

We obtain a rule of the form $attacks(T) \longleftarrow \ldots$ and we can finally ground the specification and look for stable models where $attack(t_{max})$ is true (see Section 4). The intuition behind this rule and indeed behind what an attack is can be also explained in the vernacular:

1. Look at the problem from the perspective of an agent A wishing to securily buy an item from a merchant B.
2. A has sent all appropriate messages to the network, allegedly to B or to another bunch of trusted guys and has received all appropriate answers (and thus we list all messages that he has sent or received up to now).
3. For sake of example, suppose that A wants to be sure that the message about B's bank coordinates did actually come from B, i.e. B's message is *authentic*. If the protocol is correct, there is no run (i.e. stable model) of the protocol in which A could have run for so long without apparent errors and without B actually issuing this message. So, to look for a an authentication bug we add the negation of the event whose authenticity we wish to verify. If there is a model for *attack*, then in this model B didn't actually send his bank coordinates, even though A received it, allegedly from B. Something fishy is going on...
4. Looking for a secrecy bug is similar: in all our intended model the intruder is not supposed to get A's credit card number. So we should add the negation of the event (not getting the credit card number) that we wish to verify. Then, loosely speaking, we cancel double negation and just ask for a model where the spy knows the secret.
5. Additional checks may be necessary to avoid attacks that the security analysist may deem uninteresting. For a secrecy attack we may want B to be trusted (lousy merchants may well lose credit card numbers without need of buggy protocols) whereas for authentication or non-repudiation attack we may want the security of the protocol guaranteed within a certain temporal interval (after which the low level connection may time-out or certificates be no longer relevant.

Let's exemplify this procedure in the Needham-Schroeder protocol. At first we may consider the authentication guarantee that the protocol offers to Alice, the initiator of the protocol: if Alice sent $\{N_a \| A\}_{pK(B)}$ to B, received $\{N_a \| N_b\}_{pK(A)}$ and sent $\{N_b\}_{pK(B)}$, she can be sure that Bob actually sent $\{N_a \| N_b\}_{pK(A)}$ to her.

$$
\begin{array}{l}
attack(T) \longleftarrow \\
\quad \left.\begin{array}{l} \mathsf{said}(A, B, \{N_a \| A\}_{pK(B)}, T), \\ \mathsf{got}(A, \{N_a \| N_b\}_{pK(A)}, T), \\ \mathsf{said}(A, B, \{N_b\}_{pK(B)}, T), \end{array}\right\} \ \%\text{The protocol is correct for } A \\
\quad \mathit{not}\ \mathsf{said}(B, A, \{N_a \| N_b\}_{pK(A)}, T) \%\text{yet } B \text{ didn't participate} \\
\quad A \neq spy, B \neq spy \qquad\qquad \%\text{and all agents are honest}
\end{array}
$$

The authentication guarantee from B's viewpoint is stated in dual form:

$$
\begin{array}{ll}
attack(T) \longleftarrow & \\
\left.\begin{array}{l} \mathsf{got}(B, \{N_a \| A\}_{pK(B)}, T), \\ \mathsf{said}(B, A, \{N_a \| N_b\}_{pK(A)}, T), \\ \mathsf{got}(B, \{N_b\}_{pK(B)}, T), \end{array}\right\} & \%\text{The protocol is correct for } B \\
\left.\begin{array}{l} not\ \mathsf{said}(A, B, \{N_a \| A\}_{pK(B)}, T), \\ not\ \mathsf{said}(A, B, \{N_b\}_{pK(B)}, T) \end{array}\right\} & \%\text{yet } A \text{ didn't participate at all} \\
A \neq spy, B \neq spy & \%\text{and all agents are honest}
\end{array}
$$

In some protocols we may also be worried about attacks in which only some steps are missing. In other words, we may consider attacks in which A partecipated only in a part of the protocol: e.g. in e-commerce protocol we want A to get the goods *and* to pay them. Obvioulsy, we have an attack if B completed the run successfully, apparently with A and A neither paid not got the goods; but we also have an attack if A got the goods but "forgot" to pay.

In this example, we can weaken the attack, by eliminating either the literal (i) $not\ \mathsf{said}(A, B, \{N_a \| A\}_{pK(B)}, T)$ or the literal (ii) $not\ \mathsf{said}(A, B, \{N_b\}_{pK(B)}, T)$ from the body of the rule. This means that we accept as valid attacks those in which A indeed participated in the protocol but only in part.

Of course the meaning of the attacks that is possibly found is different:

1. if a model where $attack(t)$ is found and both (i) and (ii) are true in the precondition, it means that we have found an attack where A never participated in the protocol at any stage. So A doesn't know at all that B even exists. This is indeed Lowe's attack [19].
2. If no model is found with both (i) and (ii), but a model is found with (i) true, it means that A actually never started the protocol run with B. However, for some unfatomable reasons she sent the last message. Therefore she knows N_b.
3. If no model is found with both (i) and (ii), but a model is found with (ii) true, it means that A actually started the protocol run with B but didn't complete it (at least she didn't completed it with B). Now we can only conclude that she knows N_a.

It is up to the security analyst to decide which attack is worth looking for. However, notice that the analyst does not need to specify *how* the attack is found by combining the protocol actions. He must only specify *what* should not happen. It is the task of the model finder to find the appropriate model that satisfies these declarative constraints.

Confidentiality properties can be equally well specified by imposing that the protocol completed and yet the spy happened to get the messages that ought to be secret. We can specify them either with respect to a particular agent (the run completed correctly for one agent and yet the spy knows the secret) or for all honest participants (the runs are correct for all participants, and yet the spy knows the secret). Whereas the first case is usually coupled with a lack of authentication (the spy grabbed some secret message because the protocol failed for the other agent), the last case is an example of a total break of the protocol.

In case of the Needham-Schroeder protocol, a confidentiality attack from the viewpoint of B is the following:

$$
\begin{array}{ll}
attack \longleftarrow & \\
\left.\begin{array}{l} \mathsf{got}(B, \{N_a \| A\}_{pK(B)}, T), \\ \mathsf{said}(B, A, \{N_a \| N_b\}_{pK(A)}, T), \\ \mathsf{got}(B, \{N_b\}_{pK(B)}, T), \end{array}\right\} & \%\text{The protocol is correct for } B \\
knows(spy, N_b, T) & \%\text{yet } spy \text{ knows } N_b \\
A \neq spy, B \neq spy & \%\text{and all agents are honest}
\end{array}
$$

We can formalize the protocol and this attack in $\mathcal{AL}_{SP}$ and run `smodels` to see what happens. Indeed, we have used the Casper2ALsp translator by Lorenzon [18] to generate the $\mathcal{AL}_{SP}$ specification of the protocol from the Casper specification used by Lowe [20]. We have added some general rules for trimming down useless steps (e.g. there is no sense for the intruder to send a message to somebody if the intruder itself intercepts this very message, etc.), put some restriction on freshness similar to those imposed by Lowe on its CSP encoding and run `smodels` by setting a bound on time to 4, 5, and 6.

The result is shown in Figure 1. Each $says(A, B, M, T)$ action in the final stable model corresponding to the attack is indicated by `T. A --->B : M`, $gets(A, M, T)$ actions are indicated by `T. -> A:M` and the $notes(A, M, T)$ is indicated by `T. # A:M`. The ellipsis indicates that we have eliminated some obviously spurious messages[8] that have been also sent by the intruder.

Since we have no control on `smodels` search heuristics, it is often the case that the attack (i.e. the stable model) is not minimal and that there are some spurious actions. In a nutshell, the attack found by `smodels` is still an attack but the intruder might have wasted some time (i.e. the plan is not optimal).

A total break of the confidentiality of the protocol would be represented by

$$
\begin{array}{ll}
attack \longleftarrow & \\
\left.\begin{array}{l} \mathsf{said}(A, B, \{N_a \| A\}_{pK(B)}, T), \\ \mathsf{got}(B, \{N_a \| A\}_{pK(B)}, T), \\ \mathsf{said}(B, A, \{N_a \| N_b\}_{pK(A)}, T), \\ \mathsf{got}(A, \{N_a \| N_b\}_{pK(A)}, T), \\ \mathsf{said}(A, B, \{N_b\}_{pK(B)}, T), \\ \mathsf{got}(B, \{N_b\}_{pK(B)}, T), \end{array}\right\} & \%\text{ The protocol is correct for both } A \text{ and } B \\
knows(spy, A, N_b)T & \%\text{yet } spy \text{ knows } N_b \\
A \neq spy, B \neq spy & \%\text{and all agents are honest}
\end{array}
$$

6 Aziz-Diffie Key Agreement

The Aziz-Diffie key agreement protocol for mobile communication [4] as simplified by Meadows and Syverson [38] aims at establishing a shared key between a mobile unit A and a base station B.

[8] For instance, when the intruder sends to B a message encrypted with A's public key that B can't obviously read.

```
massacci{goldrake}: nice-filter.sh ns-pk-trial.lp domain.lp generic.lp 4
****** Model Checking ns-pk-trial.lp up to 4 steps **********
Pre-processing Domain
    Original program has 41 rules
    Ground program has 20924 rules
 - Searching for attacks with smodels
******* NO attack found in  1.250 second (after 0 choices)*******
massacci{goldrake}: nice-filter.sh ns-pk-trial.lp domain.lp generic.lp 5
****** Model Checking ns-pk-trial.lp up to 5 steps **********
Pre-processing Domain
    Original program has 41 rules
    Ground program has 26129 rules
 - Searching for attacks with smodels
******* NO attack found in  1.700 second (after 0 choices)*******
massacci{goldrake}: nice-filter.sh ns-pk-trial.lp domain.lp generic.lp 6
****** Model Checking ns-pk-trial.lp up to 6 steps **********
Pre-processing Domain
    Original program has 41 rules
    Ground program has 31334 rules
 - Searching for attacks with smodels
******* ATTACK found in  4.140 second ******
with 108 choices of which 0 are wrong ones *******
1.  A ---> I : {na,A}pk_I
1.        -> I : {na,A}pk_I
...
2.  I ---> B : {na,A}pk_B
2.        -> B : {na,A}pk_I
...
3.  B ---> A : {na,nb}pk_A
3.        -> A : {na,nb}pk_A
...
4.  A ---> I : {nb}pk_I
4.        -> I : {nb}pk_I
...
5.  I ---> B : {nb}pk_B
5.        -> B : {nb}pk_B
...
6.        # B : {nb}pk_B
...
******* The SPY Learned *******
na
nb
nm
```

Fig. 1. `smodels` running on Needham-Schröder

This protocol is called *key agreement protocol* because both A and B "contribute" to the generation of the key, and thus they have to agree on its value (and hence the name of the protocol). The agreement is typically done by having A and B each proposing a share of the key and the final key composed by applying some function to the two shares. In this case, the function is simple an exclusive-or of the two shares, but more complicated forms of key agreement can be found in the literature [36, Cap.22].

The informal description of the protocol is the following:

1. The mobile unit Alice sends her certificate and a fresh nonce to the base unit Bob.
2. Bob checks the certificate and replies with his certificate, a fresh share of the agreement key K_B (encrypted with Alice's public key) and binds the encrypted share and the nonce, by signing them with his private key.
3. Alice checks that everything is correct and generates her fresh share of the agreement key K_A, binds K_A and K_B together by signing the pair and sends it to Bob.

If the protocol successfully completes, then Alice and Bob agree on the key $K_A \oplus K_B$ for further communication. The first nonce is used by Alice as a guarantee that Bob's share of the key is fresh, under the obvious assumption that Bob's signature key has not been compromised. Bob's share of the key plays also the role of a nonce, guaranteeing that Alice's share is fresh.

With respect to the original protocol, we have omitted the possibility of choosing the encryption algorithm. From the viewpoint of the formal analysis all algorithms are equivalent (as we abstract most of their details away), so this is usually modeled with an extra message field which would just make the present description more complex.

Formally, it boils down to three messages:

$$\begin{array}{l} A \longrightarrow B : Cert_A \| N \\ B \longrightarrow A : Cert_B \| \{K_B\}_{pK(A)} \| Sign_{BforA}\{K_B, N\} \\ A \longrightarrow B : \{K_A\}_{pK(B)} \| Sign_{AforB}\{K_A, K_B\} \end{array}$$

where A is the mobile unit, B is the base unit, N a fresh nonce. The message $Cert_X$ is an abbreviation for $\{X, pK(X), T_{not-before}, T_{not-after}, \ldots\}_{sK(CA)}$, a certificate issued by a trusted certification authority CA. We also use the abbreviations

$$\begin{array}{rl} Sign_{BforA}\{K_B, N\} & \doteq \{h(\{K_B\}_{pK(A)} \| N)\}_{sK(B)} \\ Sign_{AforB}\{K_A, K_B\} & \doteq \{h(\{K_A\}_{pK(B)} \| \{K_B\}_{pK(A)})\}_{sK(A)} \ . \end{array}$$

The first step is always the modeling of the cryptographic primitives and the theory of knowledge and messages. To this extent we borrow from Section 5 all the corresponding rules and add more rules for modeling exclusive-or and the hash function.

The first rules about message composition are obvious:

$$\mathsf{part}(M, M_1 \oplus M_2) \longleftarrow \mathsf{msg}(M), \mathsf{msg}(M_1), \mathsf{msg}(M_2), \mathsf{part}(M, M_i)$$
$$\mathsf{part}(M, h(M_1)) \longleftarrow \mathsf{msg}(M), \mathsf{msg}(M_1), \mathsf{part}(M, M_1)$$

Reasoning about knowledge is subtler, as it heavily exploits the stable model semantics of logic programs:

$$knows(A, M_1, T) \longleftarrow knows(A, M_1 \oplus M_2, T), knows(A, M_2, T)$$
$$knows(A, M_2, T) \longleftarrow knows(A, M_1 \oplus M_2, T), knows(A, M_1, T)$$

First, we should notice that we only mention xor, and not the hash function. Infact, we have no rule for knowing the content of a message out of its hash. Thus, there is no way to derive $knows(A, M, T)$ from the sole knowledge of $knows(A, h(M), T)$, as it should be.

Second, the stable model semantics rules out unwanted models of the xor-rules that are very difficult to cope with when using monotonic logic formalisms. Suppose that we asked for a model with the additional fact that $knows(A, M_1 \oplus M_2, T)$. The correct interpretation is that A doesn't know anything else. In any monotonic logic we would have the model in which A knows also M_1 and M_2. This knowledge would be self sustained: intuitively we will use the first rule to derive that M_1 is there because M_2 is there and the second to rule to conclude that M_2 is there because M_1 is there. Here, $knows(A, M_1, T)$ and $knows(A, M_2, T)$ are not grounded in the premise $knows(A, M_1 \oplus M_2, T)$.

When using exclusive-or, it is useful to add some of its simplest algebraic properties, as many attacks exploit them [35]. Commutativity is one of them and the simplest way to cope with it is to add the axiom:

$$knows(A, M_1 \oplus M_2, T) \longleftarrow knows(A, M_2 \oplus M_1, T)$$

It is convenient to use abbreviations in the actual $\mathcal{AL}_{SP}$ code. To this extent we can use a relational translation: in every rule where an abbreviation $f(m_1, \ldots, m_n)$ occurs as symbol (or where its use can make the rule more readable), replace the abbreviation with a fresh variable M, add a new atom $is\_f(M, m_1, \ldots, m_n)$, and then define is_f appropriately. For instance for $Sign_{AforB}\{K_B, N\}$ we can use the following:

$$is\_sign(\{h(\{K_B\}_{pK(A)} \| N)\}_{sK(B)}, B, A, K_B \| N).$$
$$is\_sign(\{h(\{K_A\}_{pK(B)} \| \{K_B\}_{pK(A)})\}_{sK(A)}, A, B, K_A \| K_B).$$

The rules for sending and receiving actions are identical to the general case described in [8,9] and sketched in Section 5. So we are only left with the axioms for the protocol dependent parts.

Since we have an explicit notion of time, we can verify that certificates have not expired when writing down action preconditions for the choice rules.

To this extent, we introduce a defined fluent $\mathsf{validCert}(A, B, K_B, Cert, T)$ which specifies whether at time T, the agent A considers $Cert$ a valid certificate for the public key K_B of B.

$\mathcal{AL}_{SP}$ gives the security analyst the flexibility to specify the validity conditions. For instance, certificates are emitted by a suitably trusted certification authority, they must refer to a public key, and the current time should be within the validity period of the certificate.

$$\mathsf{validCert}(A, B, K_B, \{B \| K_B \| T_{nb} \| T_{na}\}_{sK(CA)}, T) \longleftarrow \\ \mathsf{trusts}(A, CA), \mathsf{isPubKey}(K_B), T_{nb} \leq T, T \leq T_{na}$$

During the model-checking phase, the grounder `lparse` will directly compile away the cases where the certificate is expired.

To start the protocol, A picks up a valid certificate for her public key and generates a fresh nonce:

$$\{says(A, B, Cert_A \| N, T)\} \longleftarrow mobile(A), base(B), \\ \mathsf{validCert}(A, A, pK(A), Cert_A, T), \\ \mathsf{fresh}(N, T)$$

Notice that the certificate must be valid for A, as in principle A might trust different certification authorities than B.

The agent B responds when the message he receives is valid, and has appropriately generated his fresh share of the key. He also attaches a valid certificate:

$$\{says(B, A, Cert_B \| \{K_B\}_{pK(A)} \| \{h(\{K_B\}_{pK(A)})\}_{sK(B)}, T)\} \longleftarrow \\ mobile(A), base(B), \\ \mathsf{got}(B, Cert_A \| N, T) \\ \mathsf{validCert}(B, A, pK(A), Cert_A, T) \\ \mathsf{validCert}(B, B, pK(B), Cert_B, T) \\ \mathsf{fresh}(K_B, T)$$

The last step of the protocol is carried forward by A:

$$\{says(A, B, \{K_A\}_{pK(B)} \| \{h(\{K_A\}_{pK(B)} \| \{K_B\}_{pK(A)})\}_{sK(A)}, \}) \longleftarrow \\ mobile(A), base(B), \\ \mathsf{said}(A, B, Cert_A \| N, T) \\ \mathsf{got}(A, Cert_B \| \{K_B\}_{pK(A)} \| \{h(\{K_B\}_{pK(A)})\}_{sK(B)}, T) \\ \mathsf{validCert}(A, A, pK(A), Cert_A, T) \\ \mathsf{validCert}(A, B, pK(B), Cert_B, T) \\ \mathsf{fresh}(K_A, T)$$

Notice that by adding the fluent $\mathsf{validCert}(A, A, pK(A), Cert_A, T)$ we impose that A replies only if the certificate she sent to B is still valid at the time in which the third message is issued.

Other checks can be encoded in different ways. For instance, a security analyst may impose that a certificate is valid only if the timespan $[T_{nb}, T_{na}]$ is not larger than a predefined constant. It is rather straightforward to incorporate this check into the definition of the $\mathsf{validCert}(A, B, K, C, T)$ fluent.

Another analyst may impose tougher constraints on the timeliness of messages: A only replies to B if B's message comes back within a certain time limit

t_{lim} from her initial request. In such a way the protocol implementation may avoid checking the validity of A's certificate a second time, by imposing that the timespan must exceed t_l.

$$\begin{aligned}&\mathsf{validCert}(A, B, K_B, \{B\|K_B\|T_{nb}\|T_{na}\}_{sK(CA)}, T) \longleftarrow \\ &\qquad \mathsf{trusts}(A, CA), \mathsf{isPubKey}(K_B), T_{nb} \leq T, T + t_{lim} \leq T_{na}\end{aligned}$$

Forcing these checks as preconditions on protocol actions is particularly tricky in process algebras approaches which have only an indirect notion of time. In our case it is rather simple to incorporate this check. We revise the action preconditions by replacing "Said" with "Says" and adding the time constraints.

$$\begin{aligned}&\{says(A, B, \{K_A\}_{pK(B)}\|\{h(\{K_A\}_{pK(B)}\|\{K_B\}_{pK(A)})\}_{sK(A)},\}) \longleftarrow \\ &\qquad mobile(A), base(B), \\ &\qquad says(A, B, Cert_A\|N, T_i) \\ &\qquad \mathsf{got}(A, Cert_B\|\{K_B\}_{pK(A)}\|\{h(\{K_B\}_{pK(A)})\}_{sK(B)}, T) \\ &\qquad \mathsf{validCert}(A, B, pK(B), Cert_B, T) \\ &\qquad T_i + t_{lim} \leq T, \\ &\qquad \mathsf{fresh}(K_A, T)\end{aligned}$$

These three rules are not sufficient to completely model the protocol. Indeed, the protocol description specifies that B accepts the key only after having made a number of additional checks. Thus, from the viewpoint of B the protocol can be considered completed only after these extra checks have been made.

The final "agreement step" is formalized with an action $notes(X, K_{AB}, T)$ that takes place after all messages are sent and checks made, to mark the event that X noted the final agreement key for future use.

$$\begin{aligned}&\{notes(B, K_A \oplus K_B, T)\} \longleftarrow mobile(A), base(B), \\ &\qquad \mathsf{got}(B, Cert_A\|N, T), \\ &\qquad \mathsf{said}(B, A, Cert_B\|\{K_B\}_{pK(A)}\|\{h(\{K_B\}_{pK(A)})\}_{sK(B)}, T), \\ &\qquad \mathsf{got}(B, \{K_A\}_{pK(B)}\|\{h(\{K_A\}_{pK(B)}\|\{K_B\}_{pK(A)})\}_{sK(A)}, T).\end{aligned}$$

We have used the messages exactly as they appear in the protocol description. We could use a similar rule for A, which would however be redundant.

For this complex protocol, it makes sense to define events which compromise the current value of the key agreement pair to see whether future runs of the protocol can be compromised. This is done with an oops-rule following the technique introduced by Paulson [33]: we take all short term secrets, all nonces and key which appear in the messages exchanged during a successful protocol run and let the spy note their value.

$$\begin{aligned}&\{notes(spy, N\|K_A\|K_B, T)\} \longleftarrow \\ &\qquad \mathsf{said}(A, B, Cert_A\|N, T), \\ &\qquad \mathsf{said}(B, A, Cert_B\|\{K_B\}_{pK(A)}\|\{h(\{K_B\}_{pK(A)})\}_{sK(B)}, T), \\ &\qquad \mathsf{said}(A, B, \{K_A\}_{pK(B)}\|\{h(\{K_A\}_{pK(B)}\|\{K_B\}_{pK(A)})\}_{sK(A)}, T), \\ &\qquad \mathsf{noted}(B, K_A \oplus K_B, T)\end{aligned}$$

The loss of old agreement keys is an additional event wrt the "normal" attacks that the *s*py can perform on the protocol by just intercepting and manipulating messages. When adding this rule, we want to test the robustness of the protocol if past keys can be lost to the spy.

If we add the oops-rule we must slightly change the definition of attack, otherwise trivial attacks will always be found during the verification phase: complete a run of the protocol and then pipe all secret values to the spy with an oops rule. In contrast, what really matters when checking an attack is the following: suppose that the protocol run completed successfully, and the spy didn't get the secret value by means of an oops rule, were the value compromised nonetheless?

In this way, we only block the oops rule for the current run of the protocol, but we do not forbid older protocol runs to be compromised and that compromised runs might be used by the spy to compromise the current run.

$$
\begin{aligned}
attack \longleftarrow\ & mobile(A), base(B), A \neq spy, B \neq spy,\\
& \mathsf{said}(A, B, Cert_A \| N, T),\\
& \mathsf{got}(A, Cert_B \| \{K_B\}_{pK(A)} \| Sign_{B for K_B, N}\{,\}T),\\
& \mathsf{said}(A, B, \{K_A\}_{pK(B)} \| Sign_{A for K_A, K_B}\{,\}T),\\
& not\ \mathsf{noted}(spy, N \| K_A \| K_B, T),\\
& knows(spy, K_A \oplus K_B, T).
\end{aligned}
$$

The intuition is the following: we have an attack if we have completed a run of the protocol, the current agreement keys have not been compromised by some unfortunate oops-action and yet the *s*py knows the agreement key.

7 Discussion

Throughout the paper we have referred to the differences with some of the state-of-the-art approaches for protocol verification which have been automated. Here we just summarize the main differences.

We have already pointed out that there are many connections between our proposal and Paulson's inductive method [33,34,5]. Indeed, we have in common the operational semantics for the specification of protocols. In the inductive method one models a protocol as a set of traces and then uses interactive theorem proving to prove that the protocol is secure, i.e. prove that *all* traces satisfy a desired guarantee. The price to pay is that inductive theorem proving is interactive and requires expert knowledge, even if current tools substantially help in shortening the verification efforts. Our approach is based on model finding and thus we look for *one* trace that satisfies a given property, i.e. a security violation. Thus, we can substantially automate the search for attacks.

The NRL Protocol Analyzer (NPA) shares with us the choice of the programming paradigm, as we both use logic programs. A key difference is that we use the logic programming language $\mathcal{AL}_{SP}$ as specification language whereas, Prolog is used as implementation language for the NPA [24,27]. The protocol description and the specifications for the NPA are based on state variables and rules for changing state variables with an explicit modeling of the words learned

by the intruder. This aspect of NPA is closer to state exploration tools such as Murphi [28]. Security specifications, whose violation may lead to an attack, must be written in a different language either with temporal operators as done by Syverson and Meadows [38] or by using the CAPSL intermediate language [6]. Such specifications are declarative but not executable [38].

Our current formalization does not cope with an infinite search space, which can be treated by NAP at the price of becoming interactive rather than fully automatic. Infinite state space (such as an infinite number of agents or nonces) can be modeled in our approach by minor modifications, but the price to pay is that we would also lose decidability: we could use iterative deepening on t_{max} and the number of basic objects, as this allows us to retain the benefits of the bounded model checking completeness.

We believe that, wrt other model checking approaches, the use of a declarative specification language greatly simplifies the presentation of actions and events [20,21,27,28,38]. Indeed, $\mathcal{AL}_{SP}$ is a good compromise between three contrasting needs: being close to the description of protocols as specified in the security literature, specifying security properties at a high level of abstraction, automating the analysis of the protocols and the search for bugs (i.e. security attacks). Gollmann in [16, pag. 53] writes:

> High level definitions of entity authentication may obscure the precise goals an authentication protocol should achieve. On the other hand, a low level description of the cryptographic mechanisms employed in the protocol may obscure their intended purpose.

Our specification language $\mathcal{AL}_{SP}$ is a step towards making these ends meet.

We plan to apply our verification methodology to more complex protocols such as SET [23] and test to what extent, in terms of the size of specifications, can we use only general purpose tools such as `smodels` for verifying $\mathcal{AL}_{SP}$ specifications. To ease comparison and integration with other approaches, a translator from CASPER specifications [22] to $\mathcal{AL}_{SP}$ specifications has been built [18].

Acknowledgements

We thank P. Baumgartner and U. Furbach for many useful comments on an earlier formalization of this work, I. Niemela and T. Syrianen for support in using `smodels`. This work is partly supported by ASI, CNR, and MURST grants. F. Massacci acknowledges the support of a CNR Short Term Mobility fellowship and the CNR-201-15-9 fellowship.

We dedicate this paper to Bob Kowalski on his 60th birthday, for his seminal contributions to the applications of computational logic to practical problems.

References

1. M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. Research Report SRC-125, Digital System Research Center, 1994.

2. M. Abadi and R. M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996. Preliminary version in [1].
3. K. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.
4. A. Aziz and W. Diffie. Privacy and authentication for wireless local area networks. *IEEE Personal Communications*, 1(1):25–31, 1994.
5. G. Bella and L. C. Paulson. Kerberos version IV: Inductive analysis of the secrecy goals. In *Proceedings of the Fifth European Symposium on Research in Computer Security (ESORICS'98)*, volume 1485 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, 1998.
6. S. Brackin, C. Meadows, and J. Millen. CAPSL interface for the NRL Protocol Analyzer. In *IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET-99)*. IEEE Computer Society Press, 1999. A complete specification of the Clark-Jacob Library [10] is available at `http://www.cs.sri.com/~millen/capsl/`.
7. M. Burrows, M. Abadi, and R. Needham. A logic for authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.
8. L. Carlucci Aiello and F. Massacci. An executable specification language for planning attacks to security protocols. In P. Syverson, editor, *IEEE Computer Security Foundation Workshop*, pages 88–103. IEEE Computer Society Press, 2000.
9. L. Carlucci Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, Vol. 2, No. 4, pages 542–580, 2001.
10. J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. Technical report, University of York, Department of Computer Science, November 1997. Available on the web at `http://www-users.cs.york.ac.uk/~jac/`.
11. M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal Reasoning with Abductive Event Calculus. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, 1992.
12. Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In S. Steel and R. Alami, editors, *Proceedings of the Fourth European Conference on Planning (ECP'97)*, volume 1348 of *Lecture Notes in Artificial Intelligence*, pages 169–181. Springer-Verlag, 1997.
13. D. Dolev and A. Yao. On security of public key protocols. *IEEE Transactions on Information Theory*, IT-30:198–208, 1983.
14. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming (ICLP'88)*, pages 1070–1080. MIT-Press, 1988.
15. M. Gelfond and V. Lifschitz. Representing Actions and Change as Logic Programs. *Journal of Logic Programming*, 17:301–322, 1993.
16. D. Gollmann. What do we mean by entity authentication? In *Proceedings of the Fifteenth IEEE Symposium on Security and Privacy (SSP'96)*, pages 46–54. IEEE Computer Society Press, 1996.
17. H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363. John Wiley & Sons, 1992.
18. L. Lorenzon. Un traduttore da CASPER ad $\mathcal{AL}_{SP}$. Master's thesis, Facoltà di Ingegneria, Univ. di Roma I "La Sapienza", March 2000. In Italian.

19. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In T. Margaria and S. B., editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.
20. G. Lowe. Some new attacks upon security protocols. In *Proceedings of the Ninth IEEE Computer Security Foundations Workshop (CSFW'96)*, pages 162–169. IEEE Computer Society Press, 1996.
21. G. Lowe. A hierarchy of authentication specifications. In *Proceedings of the Tenth IEEE Computer Security Foundations Workshop (CSFW'96)*, pages 31–43. IEEE Computer Society Press, 1997.
22. G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(18-30):53–84, 1998.
23. Mastercard & VISA. *SET Secure Electronic Transaction Specification: Business Description*, May 1997. Available electronically at `http://www.setco.org/set_specifications.html`.
24. C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1994.
25. C. Meadows. Open issues in formal methods for cryptographic protocol analy sis. In *Proceedings of DISCEX 2000*, pages 237–250. IEEE Computer Society Press, 2000.
26. C. A. Meadows. Formal verification of cryptographic protocols: A survey. In *Advances in Cryptology - Asiacrypt'94*, volume 917 of *Lecture Notes in Computer Science*, pages 133–150. Springer-Verlag, 1995.
27. C. A. Meadows. Analyzing the needham-schroeder publik key protocol: A comparison of two approaches. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Proceedings of the Fourth European Symposium on Research in Computer Security (ESORICS'96)*, volume 1146 of *Lecture Notes in Computer Science*, pages 351–364. Springer-Verlag, 1996.
28. J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murphi. In *Proceedings of the Sixteenth IEEE Symposium on Security and Privacy (SSP'97)*, pages 141–151. IEEE Computer Society Press, 1997.
29. R. M. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
30. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
31. I. Niemelä and P. Simmons. Smodels – an implementation of Stable Model and Well-founded Semantics for Normal Logic Programs. In *Proceedings of the Fourth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 420–429. Springer-Verlag, 1997.
32. D. O'Mahony, M. Peirce, and H. Tewari. *Electronic payment systems*. The Artech House computer science library. Artech House, 1997.
33. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
34. L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.
35. P. Ryan and S. Schneider. An attack on a recursive authentication protocol. a cautionary tale. *Information Processing Letters*, 65(15):7–16, 1998.
36. B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 1994.

37. V. S. Subrahmanian and C. Zaniolo. Relating Stable Models and AI Planning Domains. In *Proceedings of the International Conference on Logic Programming (ICLP-95)*, 1995.
38. P. Syverson and C. Meadows. A formal language for cryptographic protocol requirement. *Designs, Codes and Cryptography*, 7:27–59, 1996.
39. D. S. Weld. Recent Advances in AI Planning. *Artificial Intelligence Magazine*, (Summer 1999):93–123, 1999.