Automated Protocol Verification in Linear Logic

Marco Bozzano
ITC - IRST
Via Sommarive 18
Povo, 38050 Trento, Italy
Tel. ++39-0461-314481
bozzano@irst.itc.it

Giorgio Delzanno
D.I.S.I., Università di Genova
Via Dodecaneso 35
16146 Genova - Italy
Tel. ++39-010-3536638
Fax: ++39-010-3536699
giorgio@disi.unige.it

ABSTRACT

In this paper we investigate the applicability of a *bottom-up evaluation strategy* for a first order fragment of linear logic [7] for the purposes of automated validation of *authentication protocols*. Following [11], we use *multi-conclusion* clauses to represent the behaviour of agents in a protocol session, and we adopt the *Dolev-Yao* intruder model and related message and cryptographic assumptions. Also, we use universal quantification to provide a logical and clean way to express creation of *nonces*. Our approach is well suited to verify properties which can be specified by means of *minimality conditions*. Unlike traditional approaches based on model-checking, we can reason about *parametric*, *infinite-state* systems, thus we do not pose any limitation on the number of parallel runs of a given protocol. Furthermore, our approach can be used both to find attacks and to prove correctness of protocols. We present some preliminary experiments which we have carried out using the above approach. In particular, we analyze the *ffgg* protocol introduced by Millen [30]. This protocol is a challenging case study in that it is free from *sequential attacks*, whereas it suffers from *parallel attacks* that occur only when at least two sessions are run in parallel.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.3.2 [**Software Engineering**]: Language Classifications—*Constraint and logic languages*; D.4.6 [**Software Engineering**]: Security and Protection—*Authentication*

General Terms

Security, Languages, Verification

Keywords

Linear Logic, Bottom-up Evaluation, Model Checking, Authentication Protocols

1. INTRODUCTION

Linear logic [20] provides a logical characterization of concepts and mechanisms peculiar to concurrency like *locality*, *recursion*, *non-determinism* in the definition of a process [3, 22], and *synchronization*. Following the paradigm of *proofs as computations* proposed in [2, 28], *provability* in fragments of linear logic can be used then as a formal tool to reason about behavioural aspects of *concurrent systems* (see e.g. [31]). In other paradigms for concurrency like the theory of Petri Nets there exist however a number of consolidated *algorithmic techniques* for the validation of system properties. In [5, 6], we made a first attempt of relating these techniques with propositional fragments of linear logic, and, more precisely, with the linear logic programming language called LO [3]. LO was originally introduced as a theoretical foundation for extensions of *logic programming* languages. The appealing feature of this fragment, however, is that it can also be viewed as a rich *specification* language for protocols and concurrent systems. In fact, specification languages like Petri Nets and multiset rewriting can be naturally embedded into *propositional* LO[12]. In [5], we established a connection between *provability* in LO and *reachability* of Petri Nets via the definition of an effective procedure to compute the set of linear logic *goals* (multisets of atomic formulas) that are consequences of a given propositional program. In other words we defined a *bottom-up*[1] evaluation procedure for *propositional* programs. Our construction is based on the *backward reachability* algorithm of [1] used to decide the so called *control state reachability problem* of Petri Nets. The algorithm presented in [5] is defined, however, for the more general case of propositional LO specifications (i.e. with nested conjunctive and disjunctive goals).

In our setting, a natural way of augmenting the expressivity of the specification language is to consider *first order fragments* of linear logic. First order formulas can be used, in fact, to *color* the internal state of processes with structured data [3, 28]. The combination between first order formulas and linear connectives provides a well-founded interpretation of the dynamics in the evolution of the internal state of a process [3, 27, 28]. First order quantification in goal formulas has several interesting interpretations here: it can be viewed either as a sort of *hiding* operator in the style of π-calculus [27], or as a mechanism to generate *fresh names* as in [11].

[1] According to the usual terminology in logic programming, *bottom-up* evaluation is intended to denote derivation of logical consequences of a program, starting from the axioms

In [10, 7] we defined a procedure for the *bottom-up* evaluation of first order LO *programs with universally quantified goals*. Via the connection between *provability* and *reachability* established in [5], we can view such an evaluation procedure as a validation technique for *colored* specifications. The bottom-up evaluation procedure is based on an *effective* fixpoint operator and on a *symbolic* and *finite* representation of a potentially *infinite collection* of first-order provable LO goals (multisets of atoms). The use of this symbolic representation is crucial when trying to prove properties of *parameterized* systems, i.e., systems in which the number of individual processes is left as a parameter of the specification like for *multi-agent protocols with multiple parallel sessions*.

In this paper we investigate the applicability of the bottom-up evaluation strategy of [7] for the purposes of automated validation of *authentication protocols*. The design and implementation of cryptographic protocols are difficult and error-prone. Authentication protocols should be reliable enough to be used in a potentially compromised environment. While cryptographic primitives are a common means to achieve these goals, they are not sufficient to ensure authentication. Exchanging *nonces*, i.e. fresh values, is a commonly used technique which is exploited in combination with cryptography to achieve authentication. Different approaches have been followed to specify and analyze protocols. An incomplete list include for instance using belief logics [4], rewriting techniques [17, 11, 14], theorem proving [33], logic programming [26, 16, 8], and model-checking [23, 25, 34].

Following [11], as specification language we will use *multi-conclusion* clauses to represent a given set of agents (called *principals*) executing *parallel* protocol sessions by exchanging messages over a network. We will use the *Dolev-Yao* intruder model and related message and cryptographic assumptions. Also, enriching linear logic specifications with universal quantification in goal formulas will provide a logical and clean way to express creation of new values like *nonces*.

In order to reason about security properties, we will apply our *general purpose* bottom-up evaluation scheme for first order linear logic. Our approach is well suited to verify properties which can be specified by means of *minimality conditions* (e.g., a given state is unsafe if there are *at least* two principals which have completed the execution of a protocol and a given shared secret has been unintentionally disclosed to a third malicious agent). The resulting verification method has connections both with (symbolic) model checking [1] and with theorem proving [2]. Unlike traditional approaches based on model-checking, we can reason about *parametric*, *infinite-state* systems, thus we do not pose any limitation on the number of parallel runs of a given protocol (we also allow a principal to take part into different sessions at the same time, possibly with different roles).

We have built a prototype, written in standard ML, to implement the bottom up evaluation of LO programs. We present some preliminary experiments which we have carried out using the above approach. In particular, we will focus our attention on the validation of the *ffgg* protocol introduced by Millen [30]. This protocol is a challenging case study for the following reasons. First, the protocol is free from *sequential attacks*, whereas it suffers from *parallel attacks* that occur only when at least two sessions are run in parallel. Secondly, the scheme underlying *ffgg* can be generalized so as to obtain *higher order* attacks (i.e. attacks that need at least k sessions in parallel). Since with our bottom-up evaluation scheme we do not need to put a bound on the number of parallel sessions, the application of our method is sound for any instance of the protocol. In this paper we will discuss the experiments obtained on the original formulation presented in [30]. Furthermore, we will present experimental results obtained for a *corrected* version of Needham-Schroeder, and for the Otway-Rees protocol [15]. Although much work remains to be done, experiments show that our methodology can be effective to analyze interesting aspects of authentication such as secrecy or confidentiality.

Structure of the paper. The rest of this paper is structured as follows. In Section 2 we introduce the language LO with universally quantified goals, and in Section 3 we briefly discuss the bottom-up evaluation scheme for this language. In Section 4 we explain how authentication protocols can be encoded in linear logic and we present our case-study, namely Millen's *ffgg* protocol. In Section 5 we discuss the application of our bottom-up evaluation algorithm for the verification of security properties of authentication protocols, and we show some experimental results. Finally, in Section 6 we discuss related work and draw some conclusions.

2. THE FRAGMENT LO$_\forall$

LO [3] is a logic programming language based on a fragment of LinLog [2]. Its mathematical foundations lie on a proof-theoretical presentation of a fragment of linear logic defined over the linear connectives $\multimap$ (*linear implication*, we use the reversed notation $H \multimapinv G$ for $G \multimap H$), & (*additive conjunction*), $\parr$ (*multiplicative disjunction*), and the constant $\top$ (*additive identity*). In this section we present the proof-theoretical semantics, corresponding to the usual *top-down* operational semantics for traditional logic programming languages, for an extension of LO. First of all, we consider a slight extension of LO which admits the constant $\bot$ in goals and clause heads. More importantly, we allow the universal quantifier to appear, possibly nested, in goals. This extension is inspired by *multiset rewriting with universal quantification* [11]. The resulting language will be called LO$_\forall$ hereafter.

Following [3], we give the following definitions. Let Σ be a signature with predicates including a set of constant and function symbols $\mathcal{L}$ and a set of predicate symbols $\mathcal{P}$, and let $\mathcal{V}$ be a denumerable set of variables. An atomic formula over Σ and $\mathcal{V}$ has the form $p(t_1, \ldots, t_n)$ (with $n \geq 0$), where $p \in \mathcal{P}$ and $t_1, \ldots, t_n$ are (non-ground) terms in $T_\Sigma^\mathcal{V}$. We denote the set of such atomic formulas as $A_\Sigma^\mathcal{V}$, and the set of *ground* (i.e., without variables) atomic formulas as A_Σ. Finally, given a formula F, we denote by $FV(F)$ the set of free variables of F.

The classes of **G**-formulas (goal formulas), and **D**-formulas (multi-headed clauses) over Σ and $\mathcal{V}$ are defined by the following grammar:

$$
\begin{aligned}
\mathbf{G} &::= \mathbf{G} \parr \mathbf{G} \mid \mathbf{G} \ \& \ \mathbf{G} \mid \forall x.\mathbf{G} \mid \mathbf{A} \mid \top \mid \bot \\
\mathbf{H} &::= \mathbf{A} \parr \ldots \parr \mathbf{A} \mid \bot \\
\mathbf{D} &::= \forall (\mathbf{H} \multimapinv \mathbf{G})
\end{aligned}
$$

where $\mathbf{A}$ stands for an atomic formula over Σ and $\mathcal{V}$, and $\forall (H \multimapinv G)$ stands for $\forall x_1 \ldots x_k. (H \multimapinv G)$, with $\{x_1, \ldots, x_k\} = FV(H \multimapinv G)$.

$$\frac{}{P \vdash_\Sigma \top, \Delta}\ \top_r \qquad \frac{P \vdash_\Sigma G_1, G_2, \Delta}{P \vdash_\Sigma G_1 \mathbin{⅋} G_2, \Delta}\ ⅋_r \qquad \frac{P \vdash_\Sigma G_1, \Delta \quad P \vdash_\Sigma G_2, \Delta}{P \vdash_\Sigma G_1 \,\&\, G_2, \Delta}\ \&_r$$

$$\frac{P \vdash_\Sigma \Delta}{P \vdash_\Sigma \bot, \Delta}\ \bot_r \qquad \frac{P \vdash_{\Sigma,c} G[c/x], \Delta}{P \vdash_\Sigma \forall x.G, \Delta}\ \forall_r\ \ (c \not\in \Sigma) \qquad \frac{P \vdash_\Sigma G, \mathcal{A}}{P \vdash_\Sigma \widehat{H}, \mathcal{A}}\ bc\ \ (H \multimapinv G\ \in Gnd(P))$$

Figure 1: A proof system for LO$_\forall$

An LO$_\forall$ program over Σ and $\mathcal{V}$ is a set of **D**-formulas over Σ and $\mathcal{V}$. A multiset of goal formulas will be called a *context* hereafter. In the following we usually omit the universal quantifier in **D**-formulas, i.e., we consider free variables as being *implicitly* universally quantified.

Let Σ_P be a signature with predicates and $\mathcal{V}$ a denumerable set of variables. An LO$_\forall$ sequent has the form

$$P \vdash_\Sigma G_1, \ldots, G_k,$$

where P is an LO$_\forall$ program over Σ_P and $\mathcal{V}$, $G_1, \ldots, G_k$ is a context (i.e., a multiset of goals) over Σ and $\mathcal{V}$, and Σ is a signature such that $\Sigma_P \subseteq \Sigma$. In the following we will use Sig_P to denote the set of all possible extensions of Σ_P.

2.1 Top-down Provability

We now define provability in LO$_\forall$. Let Σ be a signature with predicates and $\mathcal{V}$ a denumerable set of variables. Given an LO$_\forall$ program P over Σ and $\mathcal{V}$, the set of ground instances of P, denoted $Gnd(P)$, is defined as follows:

$$Gnd(P) = \{(H \multimapinv G)\,\theta \mid \forall\,(H \multimapinv G) \in P\},$$

where θ is a grounding substitution for $H \multimapinv G$ (i.e., it maps variables in $FV(H \multimapinv G)$ to ground terms in T_Σ). The execution of a multiset of **G**-formulas $G_1, \ldots, G_k$ in P corresponds to a *goal-driven* proof for the LO$_\forall$ sequent $P \vdash_\Sigma G_1, \ldots, G_k$. According to this view, the operational semantics of LO$_\forall$ is given via the *uniform* (*focusing*) [2] proof system presented in Figure 1, where P is a set of clauses, $\mathcal{A}$ is a multiset of atomic formulas, and Δ is a multiset of **G**-formulas. We have used the notation $\widehat{H}$, where H is a linear disjunction of atomic formulas $a_1 ⅋ \ldots ⅋ a_n$, to denote the multiset $a_1, \ldots, a_n$ (by convention, $\widehat{\bot} = \epsilon$, where ϵ is the empty multiset). We say that G is provable from P if there exists a proof tree, built over the proof system of Figure 1, with root $P \vdash_\Sigma G$, and such that every branch is terminated with an instance of the $\top_r$ axiom. The proof system of Figure 1 is a specialization of more general uniform proof systems for linear logic like Andreoli's focusing proofs [2] and Forum [28]. Rule *bc* is analogous to a resolution step in traditional logic programming languages. By the uniformity of the proof system, it can be executed only if the right-hand side of the current LO$_\forall$ sequent consists of atomic formulas. Consider now a branch of a proof terminated by the sequent $P \vdash_\Sigma \mathcal{B}, \mathcal{A}$. When a backchaining step over a clause $H \multimapinv \top$, $\widehat{H} = \mathcal{B}$, is possible, we immediately obtain an instance of the axiom $\top_r$, i.e., a *successful* (branch of) computation independently of the current *context* $\mathcal{A}$. This observation is formally stated in the following proposition, where $\preccurlyeq$ denotes the *multiset inclusion* relation.

PROPOSITION 1. *Given an LO$_\forall$ program P and two multisets of goals Δ, Δ' such that $\Delta \preccurlyeq \Delta'$, if $P \vdash_\Sigma \Delta$ then $P \vdash_\Sigma \Delta'$.*

Finally, rule $\forall_r$ can be used to *dynamically* introduce new *names* during the computation. The initial signature Σ must contain at least the constant, function, and predicate symbols of a given program P, and it can dynamically grow thanks to rule $\forall_r$. Namely, every time rule $\forall_r$ is fired, a new constant c is added to the current signature, and the resulting goal is proved in the new one. The idea is that all terms appearing on the right-hand side of a sequent are implicitly assumed to range over the relevant signature. This behaviour is standard in logic programming languages [32].

EXAMPLE 2. *Let Σ be a signature with a constant symbol a, a function symbol f and predicate symbols p, q, r, s. Let P be the program consisting of the clauses*

1. $r(w) \multimapinv q(f(w)) ⅋ s(w)$
2. $s(z) \multimapinv \forall x.p(f(x))$
3. $\bot \multimapinv q(u) \,\&\, r(v)$
4. $p(x) ⅋ q(x) \multimapinv \top$

The goal $s(a)$ is provable from P. The corresponding proof is shown in Figure 2 (where $bc^{(i)}$ denotes backchaining rule over clause number i of P). Notice that the notion of ground instance *is relative to the current signature. For instance, backchaining over clause 3 is possible because the corresponding signature contains the constant c, and therefore $\bot \multimapinv q(f(c)) \,\&\, r(c)$ is a valid instance of clause 3.* □

In the rest of the paper we will focus our attention on an *observational semantics* that captures the provability of a restricted form of LO$_\forall$ goals, namely goals consisting of a multiset of ground atomic formulas. Specifically, given a program P we define its *top-down* operational semantics as

$$\mathcal{O}(P) = \{\ \mathcal{A} \mid \ \mathcal{A} \text{ multiset of ground atoms in } A_{\Sigma_P},\ P \vdash_{\Sigma_P} \mathcal{A}\ \}$$

Notice that a multiset $\mathcal{A} = A_1, \ldots, A_k$ (in the r.h.s. of a sequent) is logically equivalent to the multiplicative disjunction $A_1 ⅋ \ldots ⅋ A_k$.

3. BOTTOM-UP EVALUATION FOR LO$_\forall$

In this section we introduce the basic ideas underlying the *bottom-up* evaluation scheme of LO$_\forall$ programs. For more details, the reader may refer to [10, 7]. As anticipated in the previous section, we are interested in observing the set of disjunctive atomic goals that are provable in a given program P. By the admissibility of weakening, we observe that if $\mathcal{A} \in \mathcal{O}(P)$ then $\mathcal{A} + \mathcal{C} \in \mathcal{O}(P)$ for any multiset $\mathcal{C}$ (of

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash_{\Sigma,c} \top}\ \top_r}{P \vdash_{\Sigma,c} p(f(c)), q(f(c))}\ bc^{(4)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash_{\Sigma,c} \top, s(c)}\ \top_r}{P \vdash_{\Sigma,c} p(f(c)), q(f(c)), s(c)}\ bc^{(4)}}{P \vdash_{\Sigma,c} p(f(c)), q(f(c)) \parr s(c)}\ \parr_r}{P \vdash_{\Sigma,c} p(f(c)), r(c)}\ bc^{(1)}}{P \vdash_{\Sigma,c} p(f(c)), q(f(c)) \,\&\, r(c)}\ \&_r}{P \vdash_{\Sigma,c} p(f(c))}\ bc^{(3)}}{P \vdash_{\Sigma} \forall x.p(f(x))}\ \forall_r}{P \vdash_{\Sigma} s(a)}\ bc^{(2)}
$$

Figure 2: An example of $\mathrm{LO}_\forall$ proof

atomic formulas). In other words, $\mathcal{O}(P)$ is *upward closed* w.r.t. multiset inclusion. We can exploit this property in order to "finitely" represent sets of provable goals by using the following idea. We will consider interpretations consisting of multisets of *non-ground* formulas and we will lift their denotation to the upward closure of their *ground* instances. Formally, given an $\mathrm{LO}_\forall$ program P, the (non-ground) *Herbrand base* of P, denoted $HB(P)$, is given by the *set of multisets* built over the signature Σ_P associated to the program P. An *interpretation* I is any subset of $HB(P)$. An interpretation I is *ground* whenever all multisets $\mathcal{A} \in I$ consists of ground atomic formulas.

Given an interpretation I, its *denotation* $[\![I]\!]$ is the family of *ground* interpretations $\{[\![I]\!]_\Sigma\}_{\Sigma \in Sig_P}$ defined as follows:

$$[\![I]\!]_\Sigma = Up_\Sigma(Inst_\Sigma(I)),$$

where the operator $Inst_\Sigma$ is defined as

$$Inst_\Sigma(I) = \{\mathcal{A}\theta \mid \mathcal{A} \in I,\, \theta \text{ subst. over } \Sigma\}$$

and the operator Up_Σ is defined as

$$Up_\Sigma(I) = \{\mathcal{A} + \mathcal{C} \mid \mathcal{A} \in I,\, \mathcal{C} \text{ multiset over } A_\Sigma\}.$$

Here we assume the substitution θ and the multiset $\mathcal{C}$ to be defined over Σ. Notice that, in the definition of $[\![\cdot]\!]_\Sigma$ the operations of instantiation and upward-closure are performed for every possible signature $\Sigma \in Sig_P$.

Given an $\mathrm{LO}_\forall$ goal G, we need to define a notion of satisfiability w.r.t. to our definition of interpretation. For this purpose, in Fig 3 we introduce the satisfiability judgment

$$I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta,$$

where I is an interpretation, Δ is a multiset of goal formulas (a *context*), $\mathcal{C}$ is an output multiset of atomic formulas, and θ is an output substitution. In Figure 3 $\mathcal{A} \setminus \mathcal{B}$ denotes the multiset difference between $\mathcal{A}$ and $\mathcal{B}$, $|\mathcal{A}|$ denotes the *cardinality* of $\mathcal{A}$, $FV(\mathcal{A}, \mathcal{C})$ denotes the set of free variables in $\mathcal{A}+\mathcal{C}$, and $\preccurlyeq$ (as mentioned in the previous section) denotes *multiset inclusion.*

The judgment is used to compute the set of *resources* $\mathcal{C}$ (I denotes its upward closure) and the corresponding *variable bindings* that are needed for Δ to be provable in I. Intuitively, if $I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ holds then the sequent $P, P' \vdash_{\Sigma_P} \Delta\theta\gamma, \mathcal{C}\theta\gamma$, γ being a grounding substitution, is provable by augmenting P with the program P' consisting of clauses like $\mathcal{A} \multimapinv \top$ for any $\mathcal{A} \in [\![I]\!]$. Technically, the idea behind the definition is that the output multiset $\mathcal{C}$ and the output substitution θ are *minimal* so that they can be computed effectively given a program P, an interpretation I, and a signature Σ. The output substitution θ is needed in order to deal with clause instantiation, and its minimality is ensured by using most general unifiers in the definition.

Note: The notation $I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta$ requires that Δ, $\mathcal{C}$, and θ are defined over Σ. As a consequence, the newly introduced constant c in the $\forall$-case of the $\Vdash_\Sigma$ definition below *cannot be exported* through the output parameters $\mathcal{C}$ or θ. This way, universal quantification is always resolved *locally.* For simplicity, in Fig 3 we present only the formal definition of the judgement for goals without conjunction (see [7] for the complete definition). We recall that two multisets in general may have more than one (not necessarily equivalent) most general unifier and that using the notation $m.g.u.(\mathcal{B}', \mathcal{A}')$ we mean any unifier which is *non-deterministically* picked from the set of most general unifiers of $\mathcal{B}'$ and $\mathcal{A}'$.

We are now ready to define the symbolic fixpoint operator S_P working on our notion of interpretation. Given an interpretation I, the operator is defined as follows:

$$S_P(I) = \{(\widehat{H} + \mathcal{C})\,\theta \mid \quad (H \multimapinv G) \in Vrn(P),\; I \Vdash_{\Sigma_P} G \blacktriangleright \mathcal{C} \blacktriangleright \theta \;\}.$$

Here, $+$ denotes multiset union. Furthermore, given $H = A_1 \parr \ldots \parr A_k$, we recall that $\widehat{H}$ is the multiset $A_1, \ldots, A_k$. Finally, $Vrn(P)$ denotes the set of clauses that are variant (renamed with fresh variables) of clauses in P.

Example 3. *Let us consider a signature with a function symbol f and predicate symbols p, q, r, s. Let I be the interpretation consisting of the multiset $\{p(x), q(x)\}$ (for simplicity, hereafter we omit brackets in multiset notation), and P the program*

1. $r(w) \multimapinv q(f(w))$
2. $s(z) \multimapinv \forall x.p(f(x))$

Let's consider (a renaming of) the body of the first clause, $q(f(w'))$, and (a renaming of) the element in I, $p(x'), q(x')$. Using the atomic *clause for the $\Vdash_{\Sigma_P}$ judgment, with $\mathcal{A} =$*

$$
\begin{array}{lll}
\textit{axiom}: & I \Vdash_\Sigma \top, \Delta \blacktriangleright \epsilon \blacktriangleright nil; & \textit{anti}: \; I \Vdash_\Sigma \bot, \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta, \text{ if } I \Vdash_\Sigma \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta; \\
\textit{par}: & I \Vdash_\Sigma G_1 \mathbin{⅋} G_2, \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta, \text{ if } I \Vdash_\Sigma G_1, G_2, \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta; & \\
\textit{forall}: & I \Vdash_\Sigma \forall x.G, \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta, \text{ if } I \Vdash_{\Sigma,c} G[c/x], \Delta \blacktriangleright \mathcal{C} \blacktriangleright \theta, \text{ with } c \notin \Sigma & \\
\textit{atomic multiset}: & I \Vdash_\Sigma \mathcal{A} \blacktriangleright \mathcal{C} \blacktriangleright \theta, \text{ if there exist } \mathcal{B} \in I \text{ (variant)}, \mathcal{B}' \preccurlyeq \mathcal{B}, \mathcal{A}' \preccurlyeq \mathcal{A}, |\mathcal{B}'| = |\mathcal{A}'|, & \\
& \quad \mathcal{C} = \mathcal{B} \backslash \mathcal{B}', \text{ and } \theta = m.g.u.(\mathcal{B}', \mathcal{A}')_{|FV(\mathcal{A},\mathcal{C})}. &
\end{array}
$$

Figure 3: Satisfiability judgement for $\mathbf{LO}_\forall$ goals with ⅋, $\top$, $\bot$, and $\forall$.

$\mathcal{A}' = q(f(w'))$, $\mathcal{B} = p(x'), q(x')$, $\mathcal{B}' = q(x')$, we get

$$I \Vdash_{\Sigma_P} q(f(w')) \blacktriangleright p(x') \blacktriangleright [x' \mapsto f(w')].$$

Thus, the multiset $p(f(w')), r(w') \in S_P(I)$ (in fact, any of its instances is provable in P enriched with $p(x) ⅋ q(x) \multimap \top$). This is not the only possible result of applying S_P. In fact we can apply the first clause to I by choosing $\mathcal{A}' = \mathcal{B}' = \epsilon$ in the atomic *case of $\Vdash_{\Sigma_P}$. Thus, the multiset $\mathcal{A} = p(x'), q(x'), r(w)$ belongs to $\in S_P(I)$, too. Notice that the latter multiset denotes* redundant *information w.r.t. the denotations of $\mathcal{B} = p(x'), q(x')$. In fact $[\![\{\mathcal{A}\}]\!] \subseteq [\![\{\mathcal{B}\}]\!]$.*

Let's consider now (a renaming of) the body of the second clause, $\forall x.p(f(x))$, and another renaming of the single element in I, $p(x''), q(x'')$. From the $\forall$-case of $\Vdash_{\Sigma_P}$ definition, $I \Vdash_{\Sigma_P} \forall x.p(f(x)) \blacktriangleright \mathcal{C} \blacktriangleright \theta$ if $I \Vdash_{\Sigma_P,c} p(f(c)) \blacktriangleright \mathcal{C} \blacktriangleright \theta$, with $c \notin \Sigma_P$.

Now, we can apply the atomic *clause for $\Vdash_{\Sigma_P,c}$. Unfortunately, we can't choose $\mathcal{A}'$ to be $p(f(c))$ and $\mathcal{B}'$ to be $p(x'')$. In fact, by unifying $p(f(c))$ with $p(x'')$, we should get the substitution $\theta = [x'' \mapsto f(c)]$ and the output multiset $q(x'')$ (notice that x'' is a free variable in the output multiset) and this is not allowed because the substitution θ must be defined on Σ_P, in order for $I \Vdash_{\Sigma_P} \forall x.p(f(x)) \blacktriangleright \mathcal{C} \blacktriangleright \theta$ to be meaningful. It turns out that the only way to use the second clause for $\Vdash_{\Sigma_P,c}$ is to choose $\mathcal{A}' = \mathcal{B}' = \epsilon$. In fact, notice that goals of the form $p(c_1), r(c_2)$ are not provable in P enriched with the axiom $p(x) ⅋ q(x) \multimap \top$.* □

Notice that the S_P operator is defined using the judgment $\Vdash_{\Sigma_P}$. This corresponds to the idea that we are interested in observing only provable goals that are *visible* outside the scope of programs with universal quantification. The constants that are introduced during a derivation, in fact, cannot be exported outside the scope of the corresponding subderivation. The operator S_P is monotonic and continuous over the set of interpretations ordered with respect to inclusion of their denotations [7]. The fixpoint semantics $\mathcal{F}(P)$ of an $\mathrm{LO}_\forall$ program P is defined then as the *least fixpoint* of the operator S_P. Furthermore, the following property (proved in [7]) holds.

THEOREM 4 (SOUNDNESS AND COMPLETENESS). *For every $LO_\forall$ program P, $O(P) = [\![\mathcal{F}(P)]\!]_{\Sigma_P}$.*

An effective *subsumption* test (defined in accordance with rich denotations) between multisets of non-ground goals can be defined using multiset operations as follows: $\mathcal{A}$ *entails* $\mathcal{B}$ if there exists $\mathcal{B}' \preccurlyeq \mathcal{B}$ such that $\mathcal{A}$ is *more general* than a permutation of $\mathcal{B}'$ (here they are both viewed as a list of terms). This effective test can be used as a symbolic termination test for the least fixpoint computation built on top of the effective operator S_P. The resulting machinery represents then the *core* of our bottom-up evaluation procedure for $\mathrm{LO}_\forall$ programs. Sufficient conditions for termination are discussed in detail in [7].

4. SPECIFYING AUTHENTICATION PROTOCOLS

In Section 5, we will discuss the specification and analysis of different examples of protocols. Our specification language has a natural correspondence with multiset rewriting systems proposed in [11]. In this section we introduce, rather informally, some generalities about the way we will use to specify protocols in Section 5.

First of all, we need a representation for the entities (e.g. principals and messages) involved. In particular, we will use a notation like

$$pr(id, s)$$

to denote a principal with identifier *id* and internal state s. The internal state s can store information about an ongoing execution of any given protocol (for instance, the identifier of another principal, which step of the protocol has been executed, the *role* of the principal, and so on). Typically, the state s will be a term like *init* (indicating the initial state of a principal, before protocol execution), or a term like

$$step_i(data),$$

where the constructor $step_i$ denotes which is the last step executed and *data* represents the internal data of a given principal. In general, we allow more than one atom $pr(id, \_)$ inside a given configuration. In this way, we can model the possibility of a given principal to take part into different protocol runs, possibly with different *roles*. Messages sent over a given network can in turn be represented by terms like

$$n(mess\_content),$$

where *mess_content* is the content of the message. Depending on the particular protocol under consideration, we can fix a specific format for messages. For instance, a message encrypted with the public key of a principal a could be represented as the term $enc(pubk(a), mess\_content)$.

Finally, we will use the Dolev-Yao *intruder* model (see [11]) and the associated assumptions. In particular, we need a way to store the intruder knowledge. We will use terms such as

$$m(inf)$$

to represent the information in possession of the intruder (m stands for the internal *memory* of the intruder). At any

$$
\begin{array}{ll}
p_1) & pr(A, init) \parr pr(B, init) \multimap pr(A, step1(B)) \parr pr(B, init) \parr n(plain(A)) \\
p_2) & pr(B, init) \parr n(plain(A)) \multimap \forall N1.\forall N2.(pr(B, step2(A, N1)) \parr n(plain(N1, N2))) \\
p_3) & pr(A, step1(B)) \parr n(plain(N1, N2)) \multimap \forall S.(pr(A, step3(B, S)) \parr \\
 & \quad n(enc(pubk(B), N1, N2, S))) \\
p_4) & pr(B, step2(A, N1)) \parr n(enc(pubk(B), N1, X, Y)) \multimap pr(B, step4(A)) \parr \\
 & \quad n(plain(N1, X), enc(pubk(B), X, Y, N1))
\end{array}
$$

Figure 4: Specification of the *ffgg* protocol

given instant of time, we can think of the current *state* of a given system as a *multiset* of atoms representing principals and messages currently on the network, and the intruder knowledge. Following [11], we represent the environment in which protocol execution takes place by means of: a *protocol theory*, which includes rules for every protocol role (typically, one rule for every step of the protocol), and an *intruder theory*, which formalizes the set of possible actions of a malicious intruder who tries to break the protocol. In addition, it is possible to have additional rules for the environment. Rules assume the general format

$$F_1 \parr \ldots \parr F_n \multimap \forall X_1 \ldots \forall X_k.(G_1 \parr \ldots \parr G_m)$$

where F_i, G_i are atomic formulas (representing e.g. principals or messages) and X_i are variables. As explained in Section 2, the standard semantics for the universal quantifier requires new values to be chosen before application of a rule. We use this behaviour to encode nonce generation during protocol runs. As a result, we get *for free* the assumption (required by the Dolev-Yao model) that nonces are not *guessable*. In the following we will use these notational conventions: free variables inside a rule are always implicitly universally quantified, and variables are written as *upper-case* identifiers.

As far as the specification of the *initial states* is concerned, we allow a *partial* specification of the initial states. This strategy is more flexible in that it may help us to find additional hypotheses under which a given attack might take place. As a general rule, the partial specification of the initial states we have chosen requires every principal to be in his/her initial state (represented by the term *init*) at the beginning of protocol execution.

Finally, we conclude this section by collecting together some rules which are common to *all* the examples presented in Section 5. In particular, we have two rules for the environment:

$$
\begin{array}{ll}
e_1) & \bot \multimap \forall ID.(pr(ID, init)) \\
e_2) & pr(Z, S) \multimap pr(Z, S) \parr pr(Z, init)
\end{array}
$$

The first one allows the *non-deterministic* creation of new principals (we use the universal quantifier to generate new identifiers for them), whereas the second rule allows creation of a new instance of a given principal (this allows a principal to start another execution of a given protocol with a new and possibly different *role*). Both rules can be fired at run-time, i.e., during the execution of a given protocol. Thus, we will always work in an *open* environment with multiple sessions running in parallel between several agents. We use the term *init* to denote the initial state of any given principal. We also have the following two rules for the intruder theory:

$$
\begin{array}{ll}
t_1) & pr(Z, S) \multimap pr(Z, S) \parr m(Z) \\
t_2) & \bot \multimap \forall N.(m(N))
\end{array}
$$

The first one allows the intruder to store the identifier of any principal, whereas the second rule formalizes the capability of the intruder to generate new values (e.g. nonces). Before explaing what kind of properties we can reason about using the bottom-up evaluation scheme, let us describe the specification of our main case-study.

4.1 Millen's *ffgg* protocol

Although an artificial protocol, Millen's *ffgg* protocol [30] provides an example of a *parallel session* attack, which requires running at least two processes for the same role. It has been proved (see [30]) that no *serial* attacks exist, i.e., the protocol is secure if processes are serialized. The protocol is described informally as follows.

$$
\begin{array}{llcl}
1. & A \rightarrow B & : & A \\
2. & B \rightarrow A & : & N_1, N_2 \\
3. & A \rightarrow B & : & \{N_1, N_2, S\}_{K_b} \% \{N_1, X, Y\}_{K_b} \\
4. & B \rightarrow A & : & N_1, X, \{X, Y, N_1\}_{K_b}
\end{array}
$$

N_1 and N_2 stand for nonces, created by principal B and included in message 2. The $m\%m'$ notation, introduced in [24], used in message 3 represents a message which has been created by the sender according to format m, but is interpreted as m' by the receiver. In this case, the intuition is that upon receiving message 3, B checks that the first component does correspond to the first of the two nonces previously created, while no check at all is performed on the second component of the message. In message 3, S stands for a secret, of the same length as a nonce, which is in possession of A. The security property one is interested to analyze is whether the secret S can be disclosed to a malicious intruder.

We have implemented the *ffgg* protocol through the specification shown in Figure 4, while the intruder theory is presented in Figure 5.

The specification consists of a set of protocol rules (rules p_1 through p_4 in Figure 4) and an intruder theory (rules i_1 through i_8 in Figure 5). We remind the reader that the four rules e_1, e_2, t_1 and t_2 discussed in Section 4 are *in addition* to the present rules.

Protocol rules directly correspond to the informal description of the *ffgg* protocol previously presented. We have followed the conventions outlined in Section 4 to model the internal state of principals. In particular, we have a term *init* denoting the initial state of a principal, and the constructors *step*1, *step*2, *step*3 and *step*4 to model the different steps of a protocol run. At every step, each principal needs to remember the identifier of the other principal he/she is

$i_1)$ $n(plain(X)) \circ\!- m(plain(X))$

$i_2)$ $n(plain(X,Y)) \circ\!- m(plain(X)) \parr m(plain(Y))$

$i_3)$ $n(enc(X,Y,Z,W)) \circ\!- m(enc(X,Y,Z,W))$

$i_4)$ $n(plain(X,Y), enc(U,V,W,Z)) \circ\!- m(plain(X)) \parr m(plain(Y)) \parr m(enc(U,V,W,Z))$

$i_5)$ $m(plain(X)) \circ\!- m(plain(X)) \parr n(plain(X))$

$i_6)$ $m(plain(X)) \parr m(plain(Y)) \circ\!- m(plain(X)) \parr m(plain(Y)) \parr n(plain(X,Y))$

$i_7)$ $m(enc(X,Y,Z,W)) \circ\!- m(enc(X,Y,Z,W)) \parr n(enc(X,Y,Z,W))$

$i_8)$ $m(plain(X)) \parr m(plain(Y)) \parr m(enc(U,V,W,Z)) \circ\!- m(plain(X)) \parr m(plain(Y)) \parr m(enc(U,V,W,Z)) \parr n(plain(X,Y), enc(U,V,W,Z))$

Figure 5: Intruder theory for the *ffgg* protocol

$$u)\ pr(alice, step3(bob,S)) \parr pr(bob, step4(alice)) \parr m(plain(S)) \circ\!- \top$$

Figure 6: A logical representation of an infinire set of unsafe configuration for the *ffgg* protocol.

executing the protocol with. In addition, at step 2 the responder stores the first nonce created (in order to be able to perform the required check, see rule p_4), and at step 3 the initiator of the protocol remembers the secret S. We have modeled the secret S using the universal quantifier, as for nonces. In this way, we can get for free the requirement that the secret initially is only known to the principal who possesses it. Finally, we have term constructors $plain(\ldots)$ and $enc(\ldots)$ (to be precise, we should say a *family* of term constructors, we find it convenient to overload the same symbol with different *arities*) to distinguish *plain* messages from *encrypted* messages.

The intruder theory is made up of rules i_1 through i_8 in Figure 5. It is an instance of the general Dolev-Yao intruder theory (see e.g. [11]). Let us discuss it in more detail. Rules i_1 through i_4 are *decomposition* rules, whereas rules i_5 through i_8 are *composition* rules. We have four rules for each of the two different kinds (composition and decomposition) of messages, dealing with the different formats of messages used in the *ffgg* protocol. For instance, rule i_1 deals with decomposition of plain messages with one component, whereas rule i_4 deals with decomposition of messages with two plain components and one encrypted component, and so on. Clearly, the intruder cannot furtherly decompose encrypted components, which are stored exactly as they are, whereas plain messages are decomposed into their atomic constituents. The intruder theory we have presented is an instance of the general Dolev-Yao intruder theory, in that intruder rules have been tailored to the particular form of messages used in the specific protocol under consideration, an optimization often taken by verification methods [21]. This hypothesis can be relaxed (as we did for the analysis of Needham-Schroeder protocol, see Section 5). The present specification is sufficient for our purposes.

5. VERIFICATION OF SECURITY PROPERTIES

In order to understand how to apply our bottom-up evaluation strategy for the analysis of security protocols, we first need the following general observation. Several practical examples of *safety properties* present the following interesting feature: their negation can be represented by means of the upward closure of a collection of *minimal violations*, e.g., as for mutual exclusion properties of communication protocols [1]. Thanks to this property, it often becomes possible to *finitely* represent infinite collections of unsafe configurations. Symbolic procedures can be applied then in order to saturate the set of predecessor states (by iteratively applying a transition relation *backwards*[2]). Using this method and assuming that a fixpoint is eventually reached, it is possible then to establish which initial states lead to violations of the property.

This observation can be applied in our setting in order to specify interesting *security properties*. As an example, in the *ffgg* protocol we consider a configuration *unsafe* if there exist *at least* two honest principals, say *alice* and *bob*, who have run the protocol to completion (i.e., they have completed, respectively, step 4 and step 3) and the secret S has been disclosed to the intruder (i.e., it is eventually stored in the intruder's internal memory). In our setting a configuration is represented as a multiset of atomic formulas. In order to symbolically represent *all possible configurations* in which a violation might occur, we can use then the $LO_\forall$ clause in Figure 6. Every *top-down* derivation leading from an initial goal (state) to an instance of the axiom $\top_r$ obtained by applying the rule u will represent a possible *attack* to the protocol security. It is important to note that, by the admissibility of weakening, the previous $LO_\forall$ rule can be used to represent unsafe configurations for *any number of principals* involved in sessions running in parallel with the session carried over by *alice* and *bob*. Exploring all possible *top-down* derivations however corresponds to an exhaustive search of the state space of the specification and it would force us to fix a given initial configuration.

Contrary, by evaluating *bottom-up* the $LO_\forall$ program obtained by merging the protocol and intruder theory with

[2]Given that sets of unsafe configurations are encoded via logical axioms, computing the *backward* reachability set of a transition relation amounts to evaluating the corresponding logic program *bottom-up*

$$
\begin{array}{cl}
\hline & \top_r \\
P \vdash_{\Sigma_3} \top, bob^4_{al}, m(al, n1, n2, n3, n4, \{n3, s, n1\}_{K_{bob}}, \{s, n1, n3\}_{K_{bob}}) & \\
\hline & bc^{(u)} \\
P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, bob^4_{al}, m(al, n1, n2, n3, n4, s, \{n3, s, n1\}_{K_{bob}}, \{s, n1, n3\}_{K_{bob}}) & \\
\hline & bc^{(i_4)} \\
P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, bob^4_{al}, m(al, n1, n2, n3, n4, \{n3, s, n1\}_{K_{bob}}), n(n3, s, \{s, n1, n3\}_{K_{bob}}) & \\
\hline & bc^{(p_4)} \\
P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, bob^2_{al,n3}, m(al, n1, n2, n3, n4, \{n3, s, n1\}_{K_{bob}}), n(\{n3, s, n1\}_{K_{bob}}) & \\
\hline & bc^{(i_7)} \\
P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, bob^2_{al,n3}, m(al, n1, n2, n3, n4, \{n3, s, n1\}_{K_{bob}}) & \\
\hline & bc^{(i_4)} \\
P \vdash_{\Sigma_3} al^3_{bob,s}, bob^4_{al}, bob^2_{al,n3}, m(al, n1, n2, n3, n4), n(n1, n3, \{n3, s, n1\}_{K_{bob}}) & \\
\hline & bc^{(p_4)} \\
P \vdash_{\Sigma_3} al^3_{bob,s}, bob^2_{al,n1}, bob^2_{al,n3}, m(al, n1, n2, n3, n4), n(\{n1, n3, s\}_{K_{bob}}) & \\
\hline & bc^{(p_3)} \\
P \vdash_{\Sigma_2} al^1_{bob}, bob^2_{al,n1}, bob^2_{al,n3}, m(al, n1, n2, n3, n4), n(n1, n3) & \\
\hline & bc^{(i_6)} \\
P \vdash_{\Sigma_2} al^1_{bob}, bob^2_{al,n1}, bob^2_{al,n3}, m(al, n1, n2, n3, n4) & \\
\hline & bc^{(i_2)} \\
P \vdash_{\Sigma_2} al^1_{bob}, bob^2_{al,n1}, bob^2_{al,n3}, m(al, n1, n2), n(n3, n4) & \\
\hline & bc^{(p_2)} \\
P \vdash_{\Sigma_1} al^1_{bob}, bob^2_{al,n1}, bob^{init}, m(al, n1, n2), n(al) & \\
\hline & bc^{(i_5)} \\
P \vdash_{\Sigma_1} al^1_{bob}, bob^2_{al,n1}, bob^{init}, m(al, n1, n2) & \\
\hline & bc^{(i_2)} \\
P \vdash_{\Sigma_1} al^1_{bob}, bob^2_{al,n1}, bob^{init}, m(al), n(n1, n2) & \\
\hline & bc^{(p_2)} \\
P \vdash_{\Sigma} al^1_{bob}, bob^{init}, bob^{init}, m(al), n(al) & \\
\hline & bc^{(p_1)} \\
P \vdash_{\Sigma} al^{init}, bob^{init}, bob^{init}, m(al) & \\
\hline & bc^{(t_1)} \\
P \vdash_{\Sigma} al^{init}, bob^{init}, bob^{init} & \\
\hline & bc^{(e_2)} \\
P \vdash_{\Sigma} al^{init}, bob^{init} &
\end{array}
$$

Figure 7: A parallel session attack to the *ffgg* protocol: al=$alice$; p^i_d:principal p after execution of step i with internal data d; p^{init} principal in initial state; we have omitted the *plain* term constructor for plain messages; we have noted encrypted messages using the usual protocol notation; $\mathcal{M}(x, y, \ldots)$ stands for the multiset $m(x), m(y), \ldots$; finally, $\Sigma_1 = \Sigma, n1, n2$, $\Sigma_2 = \Sigma, n1, n2, n3, n4$, and $\Sigma_3 = \Sigma, n1, n2, n3, n4, s$.

the symbolic representation of unsafe states like the clause u, we obtain the same effect using *backward reachability* for a complex specification (with quantification and so on) carried over in a completely open environment. Furthermore, if a fixpoint is reached (this is not guaranteed in general) we can derive *conditions* on the initial states under which unsafe configurations will not be reached. In other words we can establish the following connection between *bottom-up* evaluation (i.e., the semantics $\mathcal{F}(P)$ defined in Section 3) of an LO$_\forall$ specification of an authentication protocol and verification of security properties. Let $Iinit$ be a collection of multisets of ground atomic formulas (the initial states of a protocol), T_p be the LO$_\forall$ theory encoding a protocol $\mathcal{P}$, T_i the LO$_\forall$ intruder theory, and let U be a collection of LO$_\forall$ clauses $\mathcal{A}_1 \multimap \top, \ldots \mathcal{A}_k \multimap \top$ (the minimal violations of a security property $\mathcal{S}$). Furthermore, let $T = T_p \cup T_i \cup U$.

Proposition 5 (Ensuring Security). *The protocol $\mathcal{P}$ is secure w.r.t. the intruder with capabilities $\mathcal{T}_i$, initial configurations $Init$, and the property $\mathcal{S}$, if and only if $Init \cap [\![\mathcal{F}(T)]\!] = \emptyset$.*

We remark that an *if and only if* condition holds in the previous proposition, i.e. the bottom-up evaluation algorithm is correct and complete. As a corollary, we get the following property (*only if* direction in the previous proposition), useful for debugging purposes.

Proposition 6 (Proving Insecurity). *If there exists $\mathcal{A}$ such that $\mathcal{A} \in Init \cap [\![\mathcal{F}(T)]\!]$, then there exists an* attack *that leads from the initial configuration $\mathcal{A}$ to an unsafe configuration $\mathcal{B} \in [\![U]\!]$.*

5.1 Some Practical Experiments

A simple prototype which implements the bottom-up evaluation procedure for LO$_\forall$ programs has been implemented in SML as described in [10]. Running our bottom-up evaluation algorithm on the *ffgg* specification, we automatically find a violation to the security property of Figure 6.

As in traditional model checking, counterexamples traces can be automatically generated whenever a violation is found. In particular, the trace corresponding to the above attack is shown in Figure 7 (we only post-processed the output of our verification tool to show the trace in a more human-readable form). The trace in Figure 7 corresponds to an $LO_\forall$ derivation which leads from an initial state to a state violating the security property of Figure 6. The attack is exactly the *parallel session* one described in [30]. We note that this attack is also an example of a *type flaw* attack, in that it relies on the secret S be passed as a nonce (under the hypothesis that the lengths of the respective fields are the same).

In order to let the reader better understand the connection between *bottom-up* evaluation used by our verification algorithm and the *top-down* derivation shown in Figure 7, we present below some of the steps performed by the bottom-up evaluation algorithm. In the following we follow the same syntactical notations as in Figure 7. Bottom-up evaluation starts from axiom u, i.e. we assert the following provable multiset:

$$m_1) \quad al^3_{bob,S}, bob^4_{al}, m(S),$$

where S is a *free* variable. Different clauses are applicable at this point. Among them, decomposition rule i_4. We can apply a variant of i_4, let it be

$$n(X', Y', \{V', W', Z'\}_{K_U}) \multimap m(X') \parr m(Y') \parr m(\{V', W', Z'\}_{K_U})$$

to m_1, in the following manner: unify S with Y' (hence unifying $m(S)$ in the multiset with $m(Y')$ in the clause body) and consider the other two atoms in the body (i.e. $m(X')$ and $m(\{V', W', Z'\}_{K_U})$) as being implicitly contained in m_1 (remember that interpretations are to be considered *upward-closed*). By applying the resulting clause *backwards* (i.e. the body is replaced by the head) we get the multiset

$$m_2) \quad al^3_{bob,S}, bob^4_{al}, n(X', S, \{V', W', Z'\}_{K_U}).$$

Multiset m_2 is accumulated into the current set of provable goals (other multisets can be obtained by applying the remaining program clauses). Now, consider the application of a variant of protocol rule p_4, let it be

$$(B'')^2_{A'',N1''} \parr n(\{N1'', X'', Y''\}_{K_{B''}}) \multimap (B'')^4_{A''} \parr n(N1'', X'', \{X'', Y'', N1''\}_{K_{B''}})$$

to m_2, in the following way: unify $N1''$ with X' and Z', X'' with S and V', Y'' with W', and B'' with U (thus unifying the atom $n(N1'', X'', \{X'', Y'', N1''\}_{K_{B''}})$ with the atom $n(X', S, \{V', W', Z'\}_{K_U})$). Furthermore, assume the atom $(B'')^4_{A''}$ to be implicitly contained in m_2. We get the multiset

$$m_3) \quad al^3_{bob,S}, bob^4_{al}, (B'')^2_{A'',N1''}, n(\{N1'', S, Y''\}_{K_{B''}})$$

which is in turn accumulated as a provable goal. We invite the reader to observe the correspondence between the *bottom-up* construction we are sketching and the *top-down* construction illustrated in Figure 7. Notice that the sequence of rules we are applying is the same but in the reversed order, i.e. axiom u, then rules i_4 and p_4, and so on (clearly, we are illustrating only one among the possible bottom-up derivations). Furthermore, every atom that we described as *implicitly contained* in the current multiset corresponds to one of the atoms in the top sequent of Fig. 7. In other words, the bottom-up computation starts from a multiset representing the *minimal* violations of the security property under consideration (i.e., axiom u), whereas any additional atom that turns out to be involved in the proof (see top-sequent in Fig. 7) is (implicitly) added, so to say, in a *lazy* manner as the bottom-up construction proceeds. Variable bindings can also be (implicitly) enforced during the bottom-up construction. For instance, the atom $(B'')^4_{A''}$ (which we assumed to be implicitly contained in m_2) corresponds to the atom bob^4_{al} in the top sequent of Fig. 7. Eventually, variable B'' (which is contained, e.g., in m_3) will be unified with bob, and similarly, A'' will be unified with al. We conclude the illustration of the bottom-up construction with an example of application of a clause involving universal quantification. Proceeding as above, eventually we get (a variant) of the multiset

$$m_6) \quad al^3_{bob,S}, bob^2_{al,N'}, bob^2_{A,N''}, n(\{N', N'', S\}_{K_{bob}}).$$

Now, we can apply a variant of protocol rule p_3 (see the corresponding inference in Fig. 7), let it be

$$(A')^1_{B'} \parr n(N1', N2') \multimap \forall S'. ((A')^3_{B',S'} \parr n(\{N1', N2', S'\}_{K_{B'}}))$$

to m_6, by unifying A' with al, B' with bob, S' with S, N' with $N1'$ and N'' with $N2''$. We get the following multiset:

$$m_7) \quad al^1_{bob}, bob^2_{al,N'}, bob^2_{A,N''}, n(N', N').$$

Notice that the bottom-up inference requires a new constant, let it be c, to be introduced in place of the universally quantified variable S' in the body of the above clause. According to rule $forall$ for the satisfiability judgment (see Figure 3) a static check must be performed in order to ensure that the output multiset and unifier do not contain the constant c. This check is successfully passed, thus the above inference is perfectly legal. The bottom-up construction goes on in this way until the multiset al^{init}, bob^{init} (corresponding to the bottom sequent in Figure 7) is reached.

We conclude by mentioning that we have also performed some further experiments regarding Millen's *ffgg* protocol, which we don't discuss in detail. In particular, we wanted to ascertain the role of the two nonces N_1 and N_2 in the *ffgg* protocol. According to the informal notation for the protocol introduced at the beginning of this section, principal B only checks that the first component of message T is the nonce N_1, whereas no check is performed for the second component. We have verified that imposing the check on the second component, the *ffgg* protocol is safe w.r.t. the security property and the intruder theory we have presented, while removing all checks, as expected, introduces *serial* attacks.

We think that this example is a good illustration of the capabilities of our general framework. In fact, using the *backward* evaluation strategy championed in this paper, we are able to automatically find a parallel session attack, *without enforcing any particular search strategy* of our evaluation algorithm (i.e. the same algorithm can be used to find serial or parallel attacks). Furthermore, according to [30] the *ffgg* protocol can be generalized to protocols which only admit *higher-order* parallel attacks (i.e., attacks which take place only in presence of *three or more* concurrent roles for the same principal). Using the same algorithm, and the same protocol and intruder theories as before, we can automatically find such attacks, if any exists. This distinguishes our methodology from most approaches based on model-checking, which operate on a finite-state abstraction of a given protocol, and require the number of principals and the number of roles to be fixed *in advance*.

Protocol	Invar	Steps	Size	MSize	Time	Verified
Millen's ffgg		14	306	677	1335	*attack*
Millen's ffgg (*corrected*)		14	27	810	2419	*yes*
Needham-Schroeder (*strong correctness*)		13	294	323	45	*attack*
Needham-Schroeder		13	304	755	516	*yes*
(*weak correctness*)	$\surd$	12	299	299	63	*yes*
Corrected Needham-Schroeder		14	1575	1575	791	*yes*
(*strong correctness*)	$\surd$	9	402	402	31	*yes*
Otway-Rees		5	10339	10339	4272	*attack*

Figure 8: Analysis of authentication protocols: experimental results

Another advantage of using backward reasoning is related to the generation of fresh nonces. Forward exploration needs to explicity manage generation of fresh names. The backward application of $LO_\forall$ rules allows us instead to observe only formulas defined over the signature of the original program. In fact, suppose that a rule body contains universally quantified variables that have to be matched with an interpretation computed during the bottom-up evaluation of a program. By the definition of the satisfiability judgement (Fig. 3) and of the S_P operator, we can restrict ourselves to a *local* top-down derivation in which we simplify the body of the clause (see rule *forall* and *par* of Fig. 3) and then we match the resulting multiset of formulas against the current interpretation (see rule *atomic multiset* of Fig. 3). In the end a *static check* is made in order to ensure that the *output multiset* and the *resulting unfier* do not contain the constants which have been introduced. In other words the effect of quantification in the body of a clause is simply that of *restricting* the set of possible *predecessor* configurations. Notice that, on the contrary, using top-down evaluation, the current signature is enriched by adding a new constant every time the rule for the universal quantifier is used.

In addition to Millen's protocol we have used our method to study other more classical examples of authentication protocols. As an example we have analyzed the Needham-Schroder protocol (discovering Lowe's attack [23]), verified the corresponding corrected version, and analyzed the Otway-Rees protocol. We stress that our algorithm can be used either to find attacks of given protocols, or to prove that no attacks may exist (clearly, w.r.t. a given protocol theory and a given intruder theory). In other words, the bottom-up evaluation algorithm presented in Section 3 is correct and complete: is no proof is found (w.r.t. to the given protocol and intruder theories), then no proof at all may exist.

All the experiments (performed on a Pentium II 233MhZ under Linux 2.0.32, running Standard ML of New Jersey, Version 110.0.7) are summarized in Figure 8. The tag **Invar** indicates the use of *invariant strengthening* (the set of unsafe states is enriched with the negation of other invariants), a conservative technique that can speed up the fixpoint computation. Furthermore, **Size** denotes the number of multisets inferred during the evaluation, **MSize** the maximal number of multisets computed at any step, and **Time** the execution time in seconds. More details on these experiments can be found in [10] (available on the first author's web page http://www.disi.unige.it/person/BozzanoM/).

6. CONCLUSIONS AND RELATED WORK

In this paper we have presented security protocols as a possible application field for our methodology based on a linear logic-based specification language and on a bottom-up evaluation strategy. Our verification procedure is tailored to study security violations which can be specified by means of *minimality conditions*. While this may rule out interesting properties, e.g. questions of *belief* [4], the proposed approach can be used to study secrecy and confidentiality properties. No artificial limit is imposed on the number of simultaneous sessions we are able to analyze.

We have performed some experiments on different authentication protocols which show that the methodology we propose can be effective either to find *attacks* or to validate existing protocols. We plan to overcome some current limitations of our approach, in particular we plan to refine and automatize the specification phase of protocols and of the intruder theory. Specifically, we want to study a (possibly automatic) translation between the usual informal description of protocols and our representation. As shown in the paper, a one-to-one translation (one rule for every step) could be enough, provided we have a way to store the information about the internal state of principals. For efficiency reasons, it could also be worth to investigate some optimizations, in particular to the intruder theory (concerning, e.g., the rules for composition and decomposition). We plan to use techniques like *folding / unfolding* to automatize this process.

Another topic we would like to investigate is *typed multiset rewriting* [13], which extends multiset rewriting with a typing theory based on dependent types with subsorting. Dependent types can be used to enforce dependency between an encryption key and its owner. The paper [13] also presents some extensions which increase the flexibility of multiset rewriting specifications, e.g. using memory predicates to remember information across role executions.

Finally, an open question is that of non-termination. In the few examples we have presented, our algorithm is always terminating, even without invariant strengthening. Although secrecy has been proved to be undecidable, even for finite-length protocols with data of bounded complexity [11], one may ask if a more restricted subclass of protocols exists, for which the verification algorithm presented here is always terminating.

We conclude by discussing some related work. A wide research area in security protocol analysis is related to rewriting. For instance, we mention [14], which specifies security protocols as rewriting theories which can be executed in the ELAN system. A similar approach is followed in [17], where the target executable language is instead Maude. Perhaps the more interesting work in this class is [21]. This work presents an automatic compilation process from security protocol descriptions into rewrite rules. The resulting specifications are then executed using the daTac theorem prover. As a difference with [17], which is based on *matching*, the execution strategy of [21] relies on *narrowing* and AC unification. Our approach, based on multiset unification, is clearly closer to the latter approach, although currently we do not support equational theories. All of the above approaches are limited to protocol debugging, therefore they can find attacks mounted on a given protocol, but they cannot be used to analyze correctness. Also, a crucial difference is that all the above works are based on a *forward* breadth-first-search strategy, while effectiveness of our verification algorithm strongly relies on a *backward* search strategy. Another approach which shares some similarity with ours is [16], where a specification for security protocols based on rewriting and encoded in a subset of intuitionistic logic is presented. The author uses universal quantification to generate nonces, like us, and embedded implication to store the knowledge of agents. This approach is still limited to protocol debugging.

An alternative approach to verifying security protocols is based on model checking. For instance, the FDR model checking tool was used by Lowe [23] to analyze the Needham-Schroeder public-key protocol. Other works which fall into this class are [25, 34]. All these approaches have in common the use of some kind of abstraction to transform the original problem into a *finite-state* model-checking problem, which is then studied by performing a *forward* reachability analysis. Using a finite-state approximation has the advantage of guaranteeing termination, however it only allows one to analyze a fixed number of concurrent protocol runs, an approach which is infeasible as this number increases. As a difference, we use a *symbolic* representation for *infinite* sets of states and a *backward* reachability verification procedure, which avoid putting limitations on the number of parallel sessions we are able to analyze.

Theorem proving techniques are used in [33], where protocols are inductively defined as sets of traces, and formally analyzed using the theorem prover Isabelle. Here, analysis is a *semi-automatic* process which can take several days. The NRL protocol analyzer [26] provides a mixed approach. It is based on protocol specifications given via Prolog rules, and enriched via a limited form of term rewriting and narrowing to manage symbolic encryption equations. Similarly to us, verification is performed by means of a symbolic model-checker which relies on a *backward* evaluation procedure which takes as input a set of insecure states. The analyzer needs to be fed with some inductive lemmas by the user, in the same way theorem provers need to be guided by the user during the proof search process.

In [8], the author proposes an optimized specification of security protocols based on an "attacker view" of protocol security, specified by means of Prolog rules, as in [26]. The approach is effective, and has been applied to prove correctness of a number of real protocols. The verification algorithm performs a *backward depth-first* search, which seems to be closely related to our evaluation strategy, and uses an intermediate code optimization using a technique similar to *unfolding*, which we plan to study as future work. On the other hand, we think that the multiset rewriting formalism which we use is more amenable to an automatic translation from the usual protocol notation. Ensuring faithfulness between the intended semantics of a protocol and its specification is necessary to prove correctness. Also, with respect to [8], we use a cleaner treatment for nonces, and we don't have to use approximations (which may introduce false attacks) except for *invariant strengthening*, which can be controlled by the user.

Finally, we mention some works concerning the process of translation from the usual informal notation for protocols, which we plan to study as part of our future work. Existing approaches include `Casper` [24], a compiler from protocol specifications into the CSP process algebra, oriented towards verification in FDR, and `CAPSL` [29], a specification language which can be compiled into an intermediate language and used to feed tools like Maude [17] or the NRL analyzer [26]. Finally, [21] presents an automatic compilation process into rewriting rules which is able to manage infinite-state models.

Concerning the application of linear logic to verification, we would like to mention the work in [18], where phase semantics is used to prove properties of specification of concurrent constraint programs. The phase semantics for LO proposed by Andreoli could be the possible connection between the manual 'semantic-driven' method of [18] and our automated 'syntactic-driven' method that could be interesting to investigate.

The technical details of the bottom-up evaluation strategy for $LO_\forall$ programs is described in [7] (as practical example, in [7] we have studied a parameterized *mutual exclusion* protocol). The first author's PhD thesis [10] also contains a detailed presentation and proofs for the results presented in this paper, and all the details of the experimental results mentioned in Section 5.1. Some preliminary results (e.g. Needham-Schroeder protocol) were also discussed in [9].

7. REFERENCES

[1] P. A. Abdulla, K. Cerans, B. Jonsson, Y.-K. Tsay Algorithmic Analysis of Programs with Well Quasi-ordered Domains. *Information and Computation* 160(1-2): 109-127, 2000.

[2] J.-M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[3] J.-M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.

[4] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. T.R. 39, Digital Equipment Corporation Systems Research Center, 1989.

[5] M. Bozzano, G. Delzanno, and M. Martelli. A Bottom-up Semantics for Linear Logic Programs. In *Proceedings of PPDP'00*, pages 92–102, 2000.

[6] M. Bozzano, G. Delzanno, and M. Martelli. An Effective Fixpoint Semantics for Linear Logic Programs. *Theory and Practice of Logic Programming*, 2(1):85–122, 2002.

[7] M. Bozzano, G. Delzanno, and M. Martelli. Model Checking Linear Logic Specifications, manuscript submitted for publication, March 2002 (See http://www.disi.unige.it/person/DelzannoG/).

[8] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. of CSFW'01*, pages 82–96, 2001.

[9] M. Bozzano. Ensuring Security through Model Checking in a Logical Environment (Preliminary Results). In *Proc. of the ICLP/CP 01 workshop SAVE'01*, December 2001.

[10] M. Bozzano. *A Logic-Based Approach to Model Checking of Parameterized and Infinite-State Systems.* PhD thesis, D.I.S.I., Univ. di Genova, 2002. Available at http://www.disi.unige.it/person/BozzanoM/

[11] I. Cervesato, N.A. Durgin, P.D. Lincoln, J.C. Mitchell, and A. Scedrov. A Meta-notation for Protocol Analysis. In *Proc. of CSFW'99*, pages 55–69, 1999.

[12] I. Cervesato. Petri Nets and Linear Logic: a Case Study for Logic Programming. In *Proc. of GULP-PRODE'95*, pages 313–318, 1995.

[13] I. Cervesato. Typed Multiset Rewriting Specifications of Security Protocols. In *Proc. of MFCSIT'00*, Cork, Ireland, 2001.

[14] H. Cirstea. Specifying Authentication Protocols Using Rewriting and Strategies. In *Proc. of PADL'01*, *LNCS 1990*, pages 138–152, 2001.

[15] J. Clark and J. Jacob. A Survey of Authentication Protocol Literature, 1997. Web Draft Version.

[16] G. Delzanno. Specifying and Debugging Security Protocols via Hereditary Harrop Formulas and λProlog - A Case-study -. In *Proc. of FLOPS'01*, *LNCS 2024*, pages 123–137, 2001.

[17] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In *Proc. of CSFW'98*, 1998.

[18] F. Fages, P. Ruet, and S. Soliman. Phase Semantics and Verification of Concurrent Constraint Programs. In *Proc. of LICS'98*, 1998.

[19] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113, 1993.

[20] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1:1–102, 1987.

[21] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Verifying Security Protocols. In *Proc. of LPAR'00*, *LNCS 1955*, pages 131–160, 2000.

[22] N. Kobayashi and A. Yonezawa. Asynchronous Communication Model based on Linear Logic. *Formal Aspects of Computing*, 7(2):113–149, 1995.

[23] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Proc. of TACAS'96*, *LNCS 1055*, pages 147–166, 1996.

[24] G. Lowe. Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998.

[25] W. Marrero, E. Clarke, and S. Jha. Model Checking for Security Protocols. T.R. CMU-CS-97-139, School of Computer Science, Carnegie Mellon University, 1997.

[26] C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[27] D. Miller. The π-Calculus as a Theory in Linear Logic: Preliminary Results. In *Proc. of the Workshop on Extensions of Logic Programming*, *LNCS* 660, pages 242–265, 1992.

[28] D. Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[29] J. K. Millen. CAPSL: Common Authentication Protocol Specification Language. T. R. MP 97B48, The MITRE Corporation, 1997.

[30] J. K. Millen. A Necessarily Parallel Attack. In *Proc. of Workshop on Formal Methods and Security Protocols*, Trento, Italy, 1999.

[31] R. McDowell, D. Miller, and C. Palamidessi. Encoding Transition Systems in Sequent Calculus. In *Proc. of Linear Logic 96 Tokyo Meeting*, *ENTCS 3*, 1996.

[32] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[33] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.

[34] A. W. Roscoe and P. J. Broadfoot. Proving Security Protocols with Model Checkers by Data Independence Techniques. *Journal of Computer Security*, 7(2-3):147–190, 1999.