Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties

Santiago Escobar[1], Catherine Meadows[2], and José Meseguer[2]

[1] Universidad Politécnica de Valencia, Spain. sescobar@dsic.upv.es
[2] Naval Research Laboratory, Washington, DC, USA. meadows@itd.nrl.navy.mil
[3] University of Illinois at Urbana-Champaign, USA. meseguer@cs.uiuc.edu

Abstract. In this tutorial, we give an overview of the Maude-NRL Protocol Analyzer (Maude-NPA), a tool for the analysis of cryptographic protocols using functions that obey different equational theories. We show the reader how to use Maude-NPA, and how it works, and also give some of the theoretical background behind the tool.

1 Introduction

The Maude-NPA is a tool and inference system for reasoning about the security of cryptographic protocols in which the cryptosystems satisfy different equational properties. The tool handles searches in the unbounded session model, and thus can be used to provide proofs of security as well as to search for attacks. It is the next generation of the NRL Protocol Analyzer [36], a tool that supported limited equational reasoning and was successfully applied to the analysis of many different protocols.

The area of formal analysis of cryptographic protocols has been an active one since the mid 1980's. The idea is to verify protocols that use encryption to guarantee secrecy, and that use authentication of data to ensure security, against an attacker (commonly called the *Dolev-Yao* attacker [17]) who has complete control of the network, and can intercept, alter, and redirect traffic, create new traffic on his/her own, perform all operations available to legitimate participants, and may have access to some subset of the longterm keys of legitimate principals. Whatever approach is taken, the use of formal methods has had a long history, not only for providing formal proofs of security, but also for uncovering bugs and security flaws that in some cases had remained unknown long after the original protocol's publication.

A number of approaches have been taken to the formal verification of cryptographic protocols. One of the most popular is model checking, in which the interaction of the protocol with the attacker is symbolically executed. Indeed, model-checking of secrecy (and later, authentication) in protocols in the bounded-session model (where a *session* is a single execution of a process representing an honest principal) has been shown to be decidable [42], and a number of bounded-session model checkers exist. Moreover, a number of unbounded model checkers either make use of abstraction to enforce decidability, or allow for the possibility of non-termination.

The earliest protocol analysis tools, such as the Interrogator [30] and the NRL Protocol Analyzer (NPA) [35], while not strictly speaking model checkers, relied on state exploration, and, in the case of NPA, could be used to verify security properties specified in a temporal logic language. Later, researchers used generic model checkers to analyze protocols, such as FDR [32] and later Murphi [40]. More recently the focus has been on special-purpose model-checkers developed specifically for cryptographic protocol analysis, such as Blanchet's ProVerif [8], the AVISPA tool [3], and Maude-NPA itself [23].

There are a number of possible approaches to take in the modeling of cryptoalgorithms used. In the simplest case, the free algebra model, cryptosystems are assumed to behave like black boxes: an attacker knows nothing about encrypted data unless it has the appropriate key. This is the approach taken, for example, by the above-cited use of Murphi and FDR to analyze cryptographic protocols, and current tools such as SATMC [4] and TA4SP [9] , both used in the AVISPA tool. However, such an approach, although it can work well for protocols based on generic shared key or public key cryptography, runs into problems with algorithms such as Diffie-Hellman or algorithms employing exclusive-or, which rely upon various algebraic properties such as the law of exponentiation of products, associativity-commutativity and cancellation. Without the ability to specify these properties, one needs to rely on approximations of the algorithms that may result in formal proofs of secrecy invalidated by actual attacks that are missed by the analysis (see, e.g., [41,43,46]). Thus there has been considerable interest in developing algorithms and tools for protocol analysis in the presence of algebraic theories [1,11,10,12,7].

Another way in which tools can differ is in the number of sessions. A *session* is defined to be one execution of a protocol role by a single principal. A tool is said to use the *bounded session model* if the user must specify the maximum number of sessions that can be generated in a search. It is said to use the *unbounded session model* if no such restrictions are required.

Secrecy is known to be decidable in the free theory together with the bounded session model [42], and undecidable in the free theory together with the unbounded session model [18]. The same distinction between bounded and unbounded sessions is known to hold for a number of different equational theories of interest, as well as for some authentication-related properties; see for example [10]. Thus, it is no surprise that most tools, whether or not they offer support for different algebraic theories, either operate in the bounded session model, or rely on abstractions that may result in reports of false attacks even when the protocol being analyzed is secure.

Maude-NPA is a model-checker for cryptographic protocol analysis that both allows for the incorporation of different equational theories and operates in the unbounded session model without the use of abstraction. This means that the analysis is *exact*. That is, (i) if an attack exists using the specified algebraic properties, it will be found; (ii) no false attacks will be reported; and (iii) if the tool terminates without finding an attack, this provides a *formal proof* that the protocol is secure for that attack modulo the specified properties. However,

it is always possible that the tool will not terminate; although, as explained in Section 7, a number of heuristics are included to drastically reduce the search space and make nontermination less likely. In order to have a steady, incremental approximation of the analysis, the user is also given the option of restricting the number of steps executed by Maude-NPA.

Maude-NPA is a backwards search tool, i.e., it searches backwards from a final insecure state to determine whether or not it is reachable from an initial state. This backwards search is *symbolic*, i.e., it does not start with a *concrete* attack state, but uses instead a symbolic *attack pattern*, i.e., a term with logical variables describing a general attack situation. The backwards search is then performed by *backwards narrowing*. Each backwards narrowing step denotes a state transition, such as a principal sending or receiving a message or the intruder manipulating a message, all in a backwards sense. Each backwards narrowing step takes a symbolic state (i.e., a term with logical variables) and returns a previous symbolic state in the protocol (again a term with logical variables). In performing a backwards narrowing step, the variables of the input term are appropriately instantiated in order to apply the concrete state transition, and the new previous state may contain new variables that are differentiated from any previously used variable to avoid confusion. To appropriately instantiate the input term, narrowing uses *equational unification*. As it is well-known from logic programming and automated deduction (see, e.g., [5]), unification is the process of solving equations $t = t'$. Standard unification solves these equations in a term algebra. Instead, *equational unification* (w.r.t. an equational theory E) solves an equation $t = t'$ in a free algebra for the equations E, that is, *modulo* the equational theory E. In the Maude-NPA, the equational theory E used depends on the protocol, and corresponds to the algebraic properties of the cryptographic functions (e.g. cancellation of encryption and decryption, Diffie-Hellman, or exclusive-or).

Sound techniques for equational unification are paramount to Maude-NPA, and much of the research behind it has concentrated on that. The idea is to develop, not only special–purpose equational unification techniques that can be used for specific theories, but general methods that can be extended to different theories that arise in actual practice. We thus consider a broad class of theories that incorporate those equational axioms commonly appearing in cryptosystems and for which Maude has dedicated unification algorithms [13], such as commutativity (C) or associativity-commutativity (AC) of some function symbols [19]. Our approach is designed to be as extensible as possible to different algebraic theories, and Maude-NPA is designed with this in mind. Maude-NPA has thus both dedicated and generic narrowing-based methods for solving unification problems in such theories [27], which under appropriate checkable conditions yield narrowing-based unification algorithms with a finite number of solutions [26]. Maude-NPA currently supports a number of algebraic theories, including exclusive-or, cancellation of encryption and decryption (both public and shared key), bounded associativity, and Diffie-Hellman exponentiation. We include examples of several of these in this tutorial; others can be found in the Maude-

NPA manual [23] and in our previous papers [21,19]. Moreover, we continue to explore new generic unification algorithms which we plan to incorporate into Maude-NPA in the future.

Since Maude-NPA allows reasoning in the unbounded session model, and because it allows reasoning about different equational theories (which typically generate many more solutions to unification problems than syntactic unification, leading to bigger state spaces), it is necessary to find ways of pruning the search space in order to prevent infinite or overwhelmingly large search spaces. A key technique for preventing infinite searches is the generation of *formal grammars* describing terms unreachable by the intruder; such formal grammars are described in [36,20] and in Section 7.7. However, grammars do not prune out all infinite searches, and there is a need for other techniques. Moreover, even when a search space is finite it may still be necessary to reduce it to a manageable size. State space reduction techniques for doing that have been provided in [22], with an empirical average state-space size reduction of 96% (i.e., the average size of the reduced state space is 4% of that of the original one). Furthermore, our often combined techniques are effective in obtaining a *finite* state space for all protocol analyses in many of our experiments.

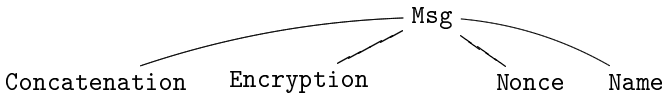
The rest of this tutorial is organized as follows. Section 2 gives definitions of some of the terminology that is used throughout this tutorial; we have tried to reduce definitions to a minimum and to illustrate each definition by means of examples. Sections 3 and 4 explain the basic mechanics of using Maude-NPA, including writing specifications and formulating queries. It gives the minimum amount of background needed for using the tool. Sections 5 and 6 describe how Maude-NPA actually works. They are intended for two types of readers: first the reader who wants to get better insight into how to use the tool to best advantage, and secondly for the reader who wants to understand the basic concepts behind Maude-NPA, without necessarily using the tool. Section 7, on state space reduction, is somewhat more specialized. It is intended for someone who is interested in the design of crypto protocol analysis tools and wants a more complete picture of the techniques Maude-NPA uses or plans to use, but is not really necessary to read it in order to be able to use the tool. Section 8 concludes the tutorial.

Maude-NPA is publicly available[4], including a user manual and some protocol examples. Maude-NPA is written in Maude, which is a publicly available[5] implementation of rewriting logic. To be more specific, Maude-NPA requires a version of Maude that includes the implementation of order-sorted unification modulo commutativity and associativity–commutativity [13], e.g., version 2.4 or later.

[4] At `http://maude.cs.uiuc.edu/tools/Maude-NPA`.
[5] At `http://maude.cs.uiuc.edu`.

2 Preliminaries

We follow the classical notation and terminology from [47] for term rewriting and from [37,38] for rewriting logic and order-sorted notions. We assume an *order-sorted signature* Σ with a finite poset of sorts $(\mathtt{S}, \leq)$ and a finite number of function symbols. For example, consider a signature with sorts `Msg`, `Encryption`, `Concatenation`, `Nonce`, `Fresh`, and `Name` that satisfy the following subsort order between sorts:

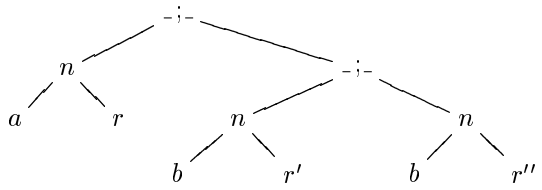


and function symbols `pk` (for "public" key encryption), `sk` (for "secret" or "private" key encryption), `n` (for nonces), and `_;_` (for concatenation) that satisfy the following sort declarations[6]:

$$\mathtt{pk} : \mathtt{Name} \times \mathtt{Msg} \to \mathtt{Encryption} \qquad \mathtt{n} : \mathtt{Name} \times \mathtt{Fresh} \to \mathtt{Nonce}$$
$$\mathtt{sk} : \mathtt{Name} \times \mathtt{Msg} \to \mathtt{Encryption} \qquad \mathtt{\_;\_} : \mathtt{Msg} \times \mathtt{Msg} \to \mathtt{Concatenation}$$

We assume an $\mathtt{S}$-sorted family $\mathcal{X} = \{\mathcal{X}_\mathtt{s}\}_{\mathtt{s} \in \mathtt{S}}$ of disjoint variable sets with each $\mathcal{X}_\mathtt{s}$ countably infinite. $\mathcal{T}_\Sigma(\mathcal{X})_\mathtt{s}$ is the set of terms of sort $\mathtt{s}$, and $\mathcal{T}_{\Sigma,\mathtt{s}}$ is the set of ground terms (i.e., without variables) of sort $\mathtt{s}$. We write $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{T}_\Sigma$ for the corresponding term algebras. We write $Var(t)$ for the set of variables present in a term t.

A term is viewed as a tree labeled with function symbols, where the arity of a symbol coincides with the number of its children in the tree (and the sort declaration for the symbol is satisfied by its children). For example, the term[7] $t = n(a, r)\ ;\ (n(b, r')\ ;\ n(b, r''))$, where a, b, and c are symbols with arity 0 (called constants) of sort `Name` and r, r', and r'' are variables of sort `Fresh`, is a term of sort `Concatenation` and has the following tree representation



A position p of a term t is described by a string of natural numbers that specifies a path from the root of the term to the desired subterm position. For

[6] Note that Maude allows function declarations of symbols with a user-defined syntax, where each underscore denotes the position of one function argument, e.g. "`_;_`" denotes a symbol ; with two arguments written in an infix notation.

[7] For a Maude term with symbols that contain underscores, a blank space must appear between the subterm corresponding to an underscore and any other part of the term, e.g. "`n(a,r) ; n(b,r')`".

example, in the term t above, position $p_1 = 1.1$ identifies the subterm a, and positions $p_2 = 2.1.1$ and $p_3 = 2.2.1$ both identify the subterm b, although different occurrences of it. The set of positions of a term t is written $Pos(t)$. The set of non-variable positions of a term t is written $Pos_\Sigma(t)$. The root of a term is Λ. The subterm of t at position p is $t|_p$, and $t[u]_p$ is the result of replacing $t|_p$ by u in t. For example, we have $t|_{p_1} = a$ and $t|_{p_2} = t|_{p_3} = b$.

A *substitution* σ is a sort-preserving mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}_\Sigma(\mathcal{X})$. The application of substitution σ to term t is denoted $\sigma(t)$. The identity substitution is *id*. Substitutions are naturally extended to homomorphic functions $\sigma : \mathcal{T}_\Sigma(\mathcal{X}) \to \mathcal{T}_\Sigma(\mathcal{X})$. The restriction of σ to a set of variables V is $\sigma|_V$. The composition of two substitutions σ, θ is $(\sigma\theta)(X) = \theta(\sigma(X))$ for $X \in \mathcal{X}$.

A Σ-*equation* is an unoriented pair of terms (t, t'), written $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_\mathtt{s}$ for some sort $\mathtt{s} \in \mathtt{S}$. Given Σ and a set E of Σ-equations such that $\mathcal{T}_{\Sigma,\mathtt{s}} \neq \emptyset$ for every sort $\mathtt{s}$, order-sorted equational logic induces a congruence relation $=_E$ on terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$ (see [38]). Throughout this tutorial we assume that $\mathcal{T}_{\Sigma,\mathtt{s}} \neq \emptyset$ for every sort $\mathtt{s}$. The *E-subsumption* order on terms $\mathcal{T}_\Sigma(\mathcal{X})_\mathtt{s}$, written $t \preccurlyeq_E t'$ (meaning that t' is more general than t), holds if there exists a substitution σ such that $t =_E \sigma(t')$. The order on terms $\mathcal{T}_\Sigma(\mathcal{X})_\mathtt{s}$ is naturally extended to substitutions, i.e., $\sigma \preccurlyeq_E \sigma'$ iff there exists a substitution θ such that $\sigma =_E \sigma'\theta$.

An *E-unifier* for a Σ-equation $t = t'$ is a substitution σ s.t. $\sigma(t) =_E \sigma(t')$. Given two E-unifiers σ_1 and σ_2 for a Σ-equation $t = t'$, σ_2 is more general than σ_1, written $\sigma_1 \preccurlyeq_E \sigma_2$, if there exists τ such that $\sigma_2\tau =_E \sigma_1$, i.e., for each variable X, $(\sigma_2\tau)(X) =_E \sigma_1(X)$. A *complete* set of E-unifiers of an equation $t = t'$, written $CSU_E(t = t')$, is a set of unifiers of $t = t'$ such that for each E-unifier σ of $t = t'$, there exists $\tau \in CSU_E(t = t')$ such that $\sigma \preccurlyeq_E \tau$. We say that $CSU_E(t = t')$ is *finitary* if it contains a finite number of E-unifiers. We say that $CSU_E(t = t')$ is *the set of most general unifiers* if each unifier $\tau \in CSU_E(t = t')$ is maximal among all unifiers of $t = t'$ w.r.t. $\preccurlyeq_E$. For example, consider an infix symbol $\mathtt{\_*\_} : \mathtt{Msg} \times \mathtt{Msg} \to \mathtt{Msg}$ satisfying the following associativity and commutativity (AC) equational properties (where X, Y, Z are variables of sort $\mathtt{Msg}$):

$$X * Y = Y * X \qquad X * (Y * Z) = (X * Y) * Z$$

A complete set of most general AC-unifiers of the two terms $t = X * Y$ and $s = U * V$ (where X, Y, U, V are variables of sort $\mathtt{Msg}$) is

$$\begin{array}{lllll}
\sigma_1 = \{ & X \mapsto X', & Y \mapsto Y', & U \mapsto X', & V \mapsto Y' \} \\
\sigma_2 = \{ & X \mapsto X', & Y \mapsto Y', & U \mapsto Y', & V \mapsto X' \} \\
\sigma_3 = \{ & X \mapsto X', & Y \mapsto Y' * Y'', & U \mapsto X' * Y'', & V \mapsto Y' \} \\
\sigma_4 = \{ & X \mapsto X', & Y \mapsto Y' * Y'', & U \mapsto Y'', & V \mapsto X' * Y' \} \\
\sigma_5 = \{ & X \mapsto X' * X'', & Y \mapsto Y', & U \mapsto X'', & V \mapsto X' * Y' \} \\
\sigma_6 = \{ & X \mapsto X' * X'', & Y \mapsto Y', & U \mapsto X'' * Y', & V \mapsto X' \} \\
\sigma_7 = \{ & X \mapsto X' * X'', & Y \mapsto Y' * Y'', & U \mapsto X'' * Y'', & V \mapsto X' * Y' \}
\end{array}$$

Consider now the exclusive-or symbol $\mathtt{\_} \oplus \mathtt{\_} : \mathtt{Msg} \times \mathtt{Msg} \to \mathtt{Msg}$ and the constant $0 : \mathtt{Msg}$ satisfying the following xor-properties (where X, Y, Z are variables of

sort `Msg`):

$$X \oplus Y = Y \oplus X \qquad X \oplus X = 0$$
$$X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z \qquad X \oplus 0 = X$$

A complete set of most general xor-unifiers of the two terms $t = X \oplus Y$ and $s = U \oplus V$ (where X, Y, U, V are variables of sort `Msg`) is the unique unifier $\theta_1 = \{X \mapsto Y' \oplus U' \oplus V', Y \mapsto Y', U \mapsto U', V \mapsto V'\}$.

3 Protocol Specification in Maude-NPA

In this section, we describe how to specify a protocol and all its relevant items in the Maude-NPA. We postpone until Section 4 the topic of formal protocol analysis in the Maude-NPA.

3.1 Templates and File Organization for Protocol Specification

Protocol specifications are given in a single file (e.g., `foo.maude`), and consist of three Maude modules, having a fixed format and fixed module names. In the first module, the *syntax* of the protocol is specified, consisting of sorts, subsorts, and operators. The second module specifies the *algebraic properties* of the operators. Note that algebraic properties *cannot be arbitrary* and must satisfy some specific conditions described in Section 6.3. The third module specifies the *actual behavior of the protocol* using a strand-theoretic notation [28]. This third module includes the intruder strands (Dolev-Yao strands) and regular strands describing the behavior of principals. *Attack states*, describing behavior that we want to prove cannot occur, are also specified in this third module, but we postpone their presentation until Section 4.

We give a template for any Maude-NPA specification below. Throughout, lines beginning with three or more dashes (i.e., `---`) or three or more asterisks (i.e., `***`) are comments that are ignored by Maude. Furthermore, the Maude syntax is almost self-explanatory [14]. The general point is that each syntactic element –e.g. a sort, an operation, an equation, a rule– is declared with an obvious keyword: `sort`, `op`, `eq`, `rl`, etc., ended by a space and a period.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  protecting DEFINITION-PROTOCOL-RULES .
  ------------------------------------------------------------
  --- Overwrite this module with the syntax of your protocol.
  --- Notes:
  --- * Sorts Msg and Fresh are special and imported
  --- * Every sort must be a subsort of Msg
  --- * No sort can be a supersort of Msg
  --- * Variables of sort Fresh are really fresh
  ---   and no substitution is allowed on them
  --- * Sorts Msg and Public cannot be empty
  ------------------------------------------------------------
endfm
```

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .
  ------------------------------------------------------------------
  --- Overwrite this module with the algebraic properties
  --- of your protocol.
  --- * Use only equations of the form (eq Lhs = Rhs [nonexec] .)
  --- * Maude attribute owise cannot be used
  --- * There is no order of application between equations
  ------------------------------------------------------------------
endfm

fmod PROTOCOL-SPECIFICATION is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .
  protecting DEFINITION-PROTOCOL-RULES .
  protecting DEFINITION-CONSTRAINTS-INPUT .

  ----------------------------------------------------------
  --- Overwrite this module with the strands
  --- of your protocol and the attack states
  ----------------------------------------------------------

  eq STRANDS-DOLEVYAO =
      --- Add Dolev-Yao intruder strands here. Strands are
      --- properly renamed.
  [nonexec] .

  eq STRANDS-PROTOCOL =
      --- Add protocol strands here. Strands are properly renamed.
  [nonexec] .

  eq ATTACK-STATE(0) =
      --- Add attack state here
      --- More than one attack state can be specified, but each
      --- must be identified by a number (e.g. ATTACK-STATE(1) = ...
      --- ATTACK-STATE(2) = ... etc.)
  [nonexec] .
endfm

--- THE FOLLOWING COMMAND HAS TO BE THE LAST ACTION !!!!

select MAUDE-NPA .
```

In what follows we explain in detail how each of these three modules, comprising a Maude-NPA protocol specification, are specified.

3.2 Specifying the Protocol Syntax

The protocol syntax is specified in the module `PROTOCOL-EXAMPLE-SYMBOLS`. Note that, since we are using Maude also as the specification language, each declaration has to be ended by a space and a period.

We begin by specifying *sorts*. In general, sorts are used to specify different types of data, that are used for different purposes. We have a special sort called `Msg` that represents the messages in our protocol. If only keys can be used for encryption, we would want to have a sort `Key`, and specify that an encryption operator `e` can only take a term of sort `Key` as its first argument, which can be specified in Maude as follows:

```
op e : Key Msg -> Msg .
```

Sorts can also be *subsorts* of other sorts. Subsorts allow a more refined distinction of data within a concrete sort. For example, we might have a sort `Masterkey` which is a subsort of `Key`. Or two sorts `PublicKey` and `PrivateKey` that are subsorts of `Key`. These two relevant subsort relations can be specified in Maude as follows:

```
subsort MasterKey < Key .
subsorts PublicKey PrivateKey < Key .
```

Most sorts are user-defined. However, there are several special sorts that are automatically imported by any Maude-NPA protocol definition through the `DEFINITION-PROTOCOL-RULES` module. The user must make sure that certain constraints are satisfied for the sorts and subsorts specified in `PROTOCOL-EXAMPLE-SYMBOLS`, which are the following:

`Msg`. Every sort specified in a protocol must be *connected* to the sort `Msg`, i.e., for every sort `S` given by the user, and every term t of sort `S`, there must be a term t' in `Msg` and a position p in t' such that $t'|_p = t$. Any subsort of `Msg` is of course directly connected to `Msg`. No sort defined by the user can be a supersort of `Msg`. The sort `Msg` should not be empty, i.e., there should be enough function symbols and constants such that there is at least one ground term (i.e., a term without variables) of sort `Msg`.

`Fresh`. The sort `Fresh` is used to identify terms that must be unique. No sort can be a subsort of `Fresh`. It is typically used in protocol specifications as an argument of some data that must be unique, such as a nonce, or a session key, e.g., "`n(A,r)`" or "`k(A,B,r)`", where `r` is a variable of sort `Fresh`. It is not necessary to define symbols of sort `Fresh`, i.e., the sort `Fresh` can be empty, since only variables of such sort will be allowed.

`Public`. The sort `Public` is used to identify terms that are publicly available, and therefore assumed known by the intruder. No sort can be a supersort of `Public`. This sort cannot be empty.

To illustrate the definition of sorts, we use the Needham-Schroeder Public Key Protocol (NSPK) as the running example. This protocol uses public key cryptography, and the principals exchange encrypted data consisting of names and nonces. We recall the informal specification of NSPK as follows:

1. $A \rightarrow B : pk(B, A; N_A)$
2. $B \rightarrow A : pk(A, N_A; N_B)$
3. $A \rightarrow B : pk(B, N_B)$

where N_A and N_B are nonces generated by the respective principals.

Thus, we will define sorts to distinguish names, keys, nonces, and encrypted data. This is specified as follows:

```
--- Sort Information
 sorts Name Nonce Enc .
 subsort Name Nonce Enc < Msg .
 subsort Name < Public .
```

The sorts `Nonce` and `Enc` are not strictly necessary, but they can make the search more efficient, since Maude-NPA will not attempt to unify terms with incompatible sorts. For example, if in this specification a principal is expecting a term of sort `Enc`, he/she will not accept a term of sort `Nonce`; technically because `Nonce` is not declared as a subsort of `Enc`. If we are looking for type confusion attacks, we would not want to include these sorts, and instead would declare everything as having sort `Msg` or `Name`. See Section 3.3 for details on type confusion attacks.

We can now specify the different operators needed in NSPK. These are `pk` and `sk`, for public and private key encryption, the operator `n` for nonces, designated constants for the names of the principals, and concatenation using the infix operator "`_;_`".

We begin with the public/private encryption operators.

```
  --- Encoding operators for public/private encryption
 op pk : Name Msg -> Enc [frozen] .
 op sk : Name Msg -> Enc [frozen] .
```

The `frozen` attribute is technically necessary to tell Maude not to attempt to apply rewrites in arguments of those symbols. The `frozen` attribute *must be included in all operator declarations in Maude-NPA specifications*, excluding constants. The use of sort `Name` as an argument of public key encryption may seem odd at first. But it is used because we are implicitly associating a public key with a name when we apply the `pk` operator, and a private key with a name when we apply the `sk` operator. A different syntax specifying explicit keys could have been chosen for public/private encryption.

Next we specify some principal names. For NSPK, we have three constants of sort `Name`, `a` (for "Alice"), `b` (for "Bob"), and `i` (for the "Intruder").

```
  --- Principals
 op a : -> Name . --- Alice
 op b : -> Name . --- Bob
 op i : -> Name . --- Intruder
```

These are not all the possible principal names. Since Maude-NPA is an unbounded session tool, the number of possible principals is unbounded. This is achieved by using variables (i.e., variables of sort `Name` in this specification of NSPK) instead of constants. However, we may have a need to specify constant principal names in a goal state. For example, if we have an initiator and a responder, and we are not interested in the case in which the initiator and the responder are the same, we can prevent that by specifying the names of the initiator and the responder as different constants. Also, we may want to identify the intruder's name by a constant, so that we can cover the case in which principals are talking directly to the intruder.

We now need two more operators, one for nonces, and one for concatenation. The nonce operator is specified as follows.

```
--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .
```

Note that the nonce operator has an argument of sort `Fresh` to ensure uniqueness. The argument of type `Name` is not strictly necessary, but it provides a convenient way of identifying which nonces are generated by which principal. This makes searches more efficient, since it allows us to keep track of the originator of a nonce throughout a search.

Finally, we come to the message concatenation operator. In Maude-NPA, we specify concatenation via an infix operator "`_;_`" defined as follows:

```
--- Concatenation operator
op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .
```

The Maude operator attribute "`gather (e E)`" indicates that symbol "`_;_`" has to be parsed as associated to the left; whereas "`gather (E e)`" indicates association to the right (see [14, Section 3.9]). Note that this *gather* information is specified *only for parsing purposes*: it does *not* specify an associativity property, although the associativity of an operator is certainly expressible in Maude either as an explicit "equational rule", or as an "equational axiom" as explained in Section 3.4 below. Therefore, `_;_` in this NSPK example is just a *free* function symbol, which, when no parenthesis are added, is *parsed* in a left–associative way.

3.3 User-defined Subsorts of Sort `Msg` in Protocol Specification

The Maude-NPA tool always assumes that the sort `Msg` is the top sort, but it allows user-defined subsorts of `Msg` that can be specified by the user for a more accurate protocol specification and analysis. However, protocol modeling in Maude-NPA should pay careful attention to when and why a subsort of sort `Msg` can be used, since an overly general specification using sort `Msg` will negatively affect performance, whereas a too restrictive specification may jeopardize finding some attacks, since some attacks possible in a less restrictive specification may become excluded by using some subsort in the reception of a message.

The two relevant concepts here are: (i) whether a principal can check the sort of a bit string and (ii) whether a principal can check the structure of a term to find out if it matches some given pattern. In general, data received by a principal for which the principal cannot check its structure, but can check its sort, must be represented by a variable of the appropriate sort. For example, if a principal receives a message that he/she expects to be a nonce but he/she cannot check that is has sort `Nonce`, then such a message will be represented by a variable of sort `Msg`. Sometimes, even though we do not assume that type-checking is possible, we may want to use sorts to limit the size of the search space. How this can be done, and the ramifications of doing this, are discussed later in this tutorial.

Therefore, if we are interested in a type confusion attack for a given informal protocol specification, we would like a protocol specification in Mauge-NPA that makes no distinctions on messages, i.e., principals cannot perform type checking of messages (for instance, may not be able to detect whether a string of bits is a nonce) and may not even be able to separate messages into their different components. In this very general case, we are interested in a unsorted protocol, i.e., there will be no extra sorts and every symbol will be of sort `Msg`. See [21] for an example of a type confusion attack specified in Maude-NPA.

A third relevant concept here is the use of variables of sort `Fresh`. Every variable of sort `Fresh` denotes an unguessable message. Those variables of sort `Fresh` that a principal generates have to be explicitly written in the protocol specification (see Section 3.6). Therefore, every nonce that a principal generates is represented by a message containing a variable of sort `Fresh`, e.g. $n(A, r)$, with r a variable of sort `Fresh`. If a principal is expecting this nonce again, it will be represented in the protocol specification by the same term $n(A, r)$, meaning that the principal can check the sort of the message and can also check the structure of the message. This is safe because typically a principal can verify whether two nonces are the same, i.e., whether the sequences of their bits coincide. If a principal is expecting a nonce different from one of his/her nonces, it will be represented in the protocol specification by a variable N of sort `Nonce` or of sort `Msg`, depending on the type checking facilities available to the principal.

3.4 Algebraic Properties

There are two types of algebraic properties: (i) *equational axioms*, such as commutativity, or associativity-commutativity, called *axioms*, and (ii) *equational rules*, called *equations*. Equations are specified in the `PROTOCOL-EXAMPLE-ALGEBRAIC` module, whereas axioms are specified within the operator declarations in the `PROTOCOL-EXAMPLE-SYMBOLS` module, as illustrated in what follows.

An equation is oriented into a rewrite rule in which the left–hand side of the equation is *reduced* to the right–hand side. In writing equations, one needs to specify the variables involved, and their type. Variables can be specified globally for a module, e.g., "`var Z : Msg .`", or locally within the expression using it, e.g., a variable `A` of sort `Name` in "`pk(A:Name,Z)`". Several variables of the same sort can be specified together, as "`vars X Y Z1 Z2 : Msg .`". In NSPK, we

use two equations specifying the relationship between public and private key encryption, as follows:

```
var Z : Msg .
var A : Name .

--- Encryption/Decryption Cancellation
eq pk(A,sk(A,Z)) = Z [nonexec] .
eq sk(A,pk(A,Z)) = Z [nonexec] .
```

The `nonexec` attribute is technically necessary to tell Maude not to use an equation or rule within its standard execution, since it will be used only at the Maude-NPA level rather than at the Maude level. The `nonexec` attribute *must be included in all user-defined equation declarations of the protocol.* Furthermore, the Maude attribute `owise` (i.e., otherwise) *cannot be used* in equations, since no order of application is assumed for the algebraic equations.

Since Maude-NPA uses built-in unification algorithms [13] for the case of operators having either no equational axioms, or the commutative (C) or associative-commutative (AC) equational axioms, these are specified not as standard equations but as *axioms* in the operator declarations. For example, suppose that we want to specify exclusive-or. We can specify an infix associative-commutative operator "`<+>`" in the `PROTOCOL-EXAMPLE-SYMBOLS` module as follows:

```
--- XOR operator and equational axioms
op _<+>_ : Msg Msg -> Msg [frozen assoc comm] .
op null : -> Msg .
```

where the *associativity and commutativity* axioms are declared as attributes of the `<+>` operator with the `assoc` and `comm` keywords. Similarly, we could specify an operator that is commutative but not associative with the `comm` keyword alone. [8]

We specify the equational rules for `<+>` in the `PROTOCOL-EXAMPLE-ALGEBRAIC` module as equations declared with the `eq` keyword as follows[9]:

```
vars X Y : Msg .

--- XOR equational rules
eq X <+> X <+> Y = Y [nonexec]  .
```

[8] In Maude, it is possible to specify and operator that is associative but not commutative using the `assoc` keyword, but in the Maude-NPA, symbols having the *assoc* but not the *comm* attribute *should not be specified.* The reason for this is that associative unification is not finitary (see, e.g., [5]) and is not supported in Maude. Various forms of *bounded* associativity with finitary unification are possible, but this is done differently, namely, by specifying bounded associativity with rules, see [21] for details.

[9] Note that the first equational rule, i.e., `X <+> X <+> Y = Y`, is not essentially needed for the exclusive-or theory, but it is necessary for rewriting purposes to ensure coherence, see Section 6.2.

```
eq X <+> X = null [nonexec]  .
eq X <+> null = X [nonexec]  .
```

If we want to include a Diffie-Hellman mechanism, we need two operations. One is exponentiation, and the other is modular multiplication. Since Diffie-Hellman is a commonly used algorithm in cryptographic protocols, we discuss it also here.

We begin by including several new sorts in `PROTOCOL-EXAMPLE-SYMBOLS`: `Gen`, `Exp`, `GenvExp`, and `NeNonceSet`.

```
sorts Name Nonce NeNonceSet Gen Exp Key GenvExp Enc Secret .
subsorts Gen Exp < GenvExp .
subsorts Name NeNonceSet GenvExp Enc Secret Key < Msg .
subsort Exp < Key .
subsort Nonce < NeNonceSet .
subsorts Name Gen < Public .
```

We now introduce three new operators. The first, `g`, is a constant that serves as the Diffie-Hellman *generator* of the multiplicative group. The second is exponentiation, and the third is an associative-commutative multiplication operation on nonces and products of such nonces.

```
op g : -> Gen .
op exp : GenvExp NeNonceSet -> Exp [frozen] .
op _*_ : NeNonceSet NeNonceSet -> NeNonceSet [frozen assoc comm] .
```

We then include the following equational property, to capture the fact that $z^{x^y} = z^{x*y}$:

```
eq exp(exp(W:Gen,Y:NeNonceSet),Z:NeNonceSet)
 = exp(W:Gen, Y:NeNonceSet * Z:NeNonceSet) [nonexec] .
```

There are several things to note about this Diffie-Hellman specification. The first is that, although modular multiplication has a unit and inverses, this is not included in our equational specification. Instead, we have only included the key algebraic rule that is necessary for Diffie-Hellman to work. The second is that we have specified types that will rule out certain kinds of intruder behavior. In actual fact, there is nothing that prevents an intruder from sending an arbitrary string to a principal and passing it off as an exponentiated term. The principal will then exponentiate that term. However, given our definition of the `exp` operator, only terms of type `GenvExp` can be exponentiated. This last restriction is necessary in order to ensure that the unification algorithm is finitary; see Section 6.3 for technical details. The omission of units and inverses is not necessary to ensure finitary unification, but rules out behavior of the intruder that is likely to be irrelevant for attacking the protocol, or that is likely to be easily detected (such as the intruder sending an `exp(g,0)`).

We note that if one is interested in obtaining a proof of security using these restrictive assumptions, one must provide a proof (outside of the tool) that security in the restricted model implies security in the more general model. This could be done along the lines of the proofs in [39,33,34].

3.5 Protocol Specification: Intruder Strands

The protocol itself and the intruder capabilities are both specified in the `PROTOCOL-SPECIFICATION` module. They are specified using strands. A *strand*, first defined in [28], is a sequence of positive and negative messages[10] describing a principal executing a protocol, or the intruder performing actions, e.g., the strand for Alice in NSPK is:

$$[\ pk(K_B,\ A; N_A)^+,\ pk(K_A, N_A; Z)^-,\ pk(K_B, Z)^+\ ]$$

where a positive node implies sending, and a negative node implies receiving. However, in our tool each strand is divided into the past and future parts by means of a vertical line. That is, the messages to the left of the vertical line were sent or received in the past, whereas the messages to the right of the line will be sent or received in the future. Also, we keep track of all the variables of sort `Fresh` generated by a concrete strand; see Section 5.2 for technical details. That is, all the variables $r_1, \ldots, r_j$ of sort `Fresh` generated by a strand are made explicit right before the strand, as follows:

$$:: r_1, \ldots, r_j :: [\ {m_1}^{\pm},\ \ldots,\ {m_i}^{\pm} \mid {m_{i+1}}^{\pm},\ \ldots,\ {m_k}^{\pm}\ ]$$

Note that there is some difference between the variables of sort `Fresh` generated in a strand and those appearing in a strand. In the specification of a role, they must coincide, in the sense that all the variables appearing in the description of a principal are generated during an execution (an instance) of such a strand. However, in a given protocol state, a strand may contain many more variables of sort `Fresh` than those specified at the left of the strand due to the message exchange between the principals.

We begin by specifying all the variables that are used in this module, together with the sorts of these variables. In the NSPK example, these are

```
vars X Y Z : Msg .
vars r r' : Fresh .
vars A B : Name .
vars N N1 N2 : Nonce .
```

After the variables are specified, the next thing to specify is the actions of the intruder, or Dolev-Yao rules [17]. These specify the operations an intruder can perform. Each such action can be specified by an intruder strand consisting of a sequence of negative nodes, followed by a single positive node. If the intruder can (non-deterministically) find more than one term as a result of performing one operation (as in deconcatenation), we specify each of these outcomes by separate strands. For the NSPK protocol, we have four operations: encryption with a public key (`pk`), decryption with a private key (`sk`), concatenation (`_;_`), and deconcatenation.

[10] We write $m^{\pm}$ to denote m^+ or m^-, indistinctively. We often write $+(m)$ and $-(m)$ instead of m^+ and m^-, respectively.

Encryption with a public key is specified as follows. Note that we use a principal's name to stand for the key. That is why names are of type `Name`. The intruder can encrypt any message using any public key.

```
:: nil:: [ nil | -(X), +(pk(A,X)), nil ]
```

Encryption with the private key is a little different. The intruder can only apply the `sk` operator using his own identity. So we specify the rule as follows, assuming that `i` denotes the name of the intruder:

```
:: nil:: [ nil | -(X), +(sk(i,X)), nil ]
```

Concatenation and deconcatenation are straightforward. If the intruder knows X and Y, he can find $X;Y$. If he knows $X;Y$ he can find both X and Y. Since each intruder strand should have at most one positive node, we need to use three strands to specify these actions:

```
:: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ]
:: nil :: [ nil | -(X ; Y), +(X), nil ]
:: nil :: [ nil | -(X ; Y), +(Y), nil ]
```

The final Dolev-Yao specification looks as follows. Note that our tool requires the use of the constant symbol `STRANDS-DOLEVYAO` as the repository of all the Dolev-Yao strands, and uses the associative-commutative symbol `_&_` as the union operator to form sets of strands. Note, also, that our tool considers that variables are not shared between strands, and thus will appropriately rename them when necessary.

```
eq STRANDS-DOLEVYAO
 = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
   :: nil :: [ nil | -(X ; Y), +(X), nil ] &
   :: nil :: [ nil | -(X ; Y), +(Y), nil ] &
   :: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
   :: nil :: [ nil | -(X), +(pk(Ke,X)), nil ]
[nonexec] .
```

Every operation that can be performed by the intruder, and every term that is initially known by the intruder, should have a corresponding intruder strand. For each operation used in the protocol we should consider whether or not the intruder can perform it, and specify a corresponding intruder strand that describes the conditions under which the intruder can perform it.

For example, suppose that the operation requires the use of exclusive-or. If we assume that the intruder can exclusive-or any two terms in its possession, we should represent this by the following strand:

```
:: nil :: [ nil | -(X), -(Y), +(X <+> Y), nil ]
```

If we want to give the intruder the ability to generate his own nonces, we should represent this by the following rule:

```
:: r :: [ nil | +(n(i,r)), nil ]
```

In general, it is a good idea to provide Dolev-Yao strands for all the operations that are defined, unless one is explicitly making the assumption that the intruder can *not* perform the operation. It is also *strongly recommended* that operations not used in the protocol should not be provided with Dolev-Yao strands. This is because the tool will attempt to execute rules associated with these strands, even if they are useless, and this will negatively affect performance.

3.6 Protocol Specification: Protocol Strands

In the Protocol Rules section of a specification we define the messages that are sent and received by the honest principals. We will specify one strand per role. However, since the Maude-NPA analysis supports an arbitrary number of sessions, each strand can be instantiated an arbitrary number of times.

In specifying protocol strands it is important to remember to specify them *from the point of view of the principal executing the role*. For example, in NSPK the initiator `A` starts out by sending her name and a nonce encrypted with `B`'s public key. She gets back something encrypted with her public key, but all she can tell is that it is her nonce concatenated with some other nonce. She then encrypts that other nonce under `B`'s public key and sends it out.

As explained in Section 3.3, we represent the construction of `A`'s nonce explicitly as `n(A,r)`, where `r` is a variable of sort `Fresh` therefore appearing in the header of `A`'s strand, and the extra nonce that she receives is represented by a variable `N` of sort `Nonce`. The entire strand for the initiator role is as follows:

```
:: r ::
[ nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; N)), +(pk(B, N)), nil ]
```

In the responder strand, the signs of the messages are reversed. Moreover, the messages themselves are represented differently. `B` starts out by receiving a name and some nonce encrypted under his key. He creates his own nonce, appends the received nonce to it, encrypts it with the key belonging to the name, and sends it out. He gets back his nonce encrypted under his own key. This is specified as follows:

```
:: r ::
[ nil | -(pk(B,A ; N)), +(pk(A, N ; n(B,r))), -(pk(B,n(B,r))), nil ]
```

Note that, as explained in Section 3.3, the point here is to only include things in a strand that a principal executing a strand can actually *verify*. The complete `STRANDS-PROTOCOL` specification for NSPK is as follows.

```
eq STRANDS-PROTOCOL =
  :: r ::
  [nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; N)), +(pk(B, N)), nil]
  &
  :: r ::
  [nil | -(pk(B,A ; N)), +(pk(A, N ; n(B,r))), -(pk(B,n(B,r))), nil]
[nonexec] .
```

As a final note, we remark that, if `B` received a message `Z` encrypted under a key he does not know, he would not be able to verify that he received `pk(A,Z)`, because he cannot decrypt the message. So the best we could say here is that `A` received some term `Y` of sort `Msg`.

The complete specification of the Needham-Schroeder shared-key protocol and the Diffie-Hellman protocol can be found in [23].

4 Protocol Analysis

In this section we describe how to analyze a protocol in practice. First, we explain how a protocol state looks like, and how an attack state is specified in the protocol. Then, we explain how the actual protocol analysis is performed. Technical details on how the backwards search is performed are postponed until Section 5.

4.1 Protocol States

In Maude-NPA, each state associated to the protocol execution (i.e., a backwards search) is represented by a term with four different components separated by the symbol || in the following order: (1) the set of current strands, (2) the current intruder knowledge, (3) the sequence of messages encountered so far in the backwards execution, and (4) some auxiliary data.

Strands || Intruder Knowledge || Message Sequence || Auxiliary Data

The first component, the set of current strands, indicates in particular how advanced each strand is in the execution process (by the placement of the bar). The second component contains messages that the intruder already knows (symbol `_inI`) and messages that the intruder currently doesn't know (symbol `_!inI`) but will learn in the future. Note that the set of strands and the intruder knowledge *grow along with* the backwards reachability search (see Section 5.2 for technical details) as follows:

- by propagation of the substitutions computed by unification modulo the equational theory,
- by introducing more protocol or intruder strands,
- by introducing more positive knowledge of the intruder (e.g., `M inI`), and
- by transforming positive knowledge into negative knowledge due to the backwards execution (e.g., `M inI` $\rightarrow$ `M !inI`).

The third component, the sequence of messages, is `nil` for any attack state at the beginning of the backwards search and records the actual sequence of messages exchanged so far in the backwards search from the attack state. This sequence grows as the backwards search continues and some variables may be instantiated in the backwards search. It gives a complete description of an attack when an initial state is reached but this component is intended for the benefit of

the user, and is not actually used in the backward search itself, except just by recording it. Finally, the last component contains information about the search space that the tool creates to help manage[11] its search. It does not provide any information about the attack itself, and is currently only displayed by the tool to help in debugging. More information about this last component can be found in Sections 5 and 7.

4.2 Initial states

An initial state is the final result of the backwards reachability process and is described as follows:

1. in an initial state, all strands have the bar at the beginning, i.e., all strands are of the form $:: r_1, \ldots, r_j :: [\ nil \mid {m_1}^{\pm}, \ \ldots, \ {m_k}^{\pm}\ ]$;
2. in an initial state, all the intruder knowledge is negative, i.e., all the items in the intruder knowledge are of the form (m `!inI`).

From an initial state, no further backwards reachability steps are possible, since there is nothing for the intruder to unlearn or to learn due to the following two facts: (i) since all the intruder knowledge is already negative (m `!inI`), no positive knowledge can be unlearned, and (ii) there is no further reception of messages (no negative node in a strand) that might introduce positive items (m `inI`) in the intruder knowledge. Note that in an initial state, the third component of a protocol state denotes the concrete message sequence from this initial state to the given attack state.

4.3 Unreachable states

Another interesting concept is that of an *unreachable state*. An unreachable state is a protocol state from which we cannot reach an initial state by further backwards reachability analysis. Interpreted in a forwards way, this of course means that it is an state that can *never* be reached from an initial state by forward execution. Early detection of unreachable states is essential to achieve effective protocol analysis. In Section 7, we describe several techniques for early detection of unreachable states. For instance, consider the following state found by the Maude-NPA for the NSPK:

```
:: nil :: [nil | -(pk(i, b ; n(b, r))), +(b ; n(b, r)), nil] &
:: nil :: [nil | -(n(b, r)), +(pk(b, n(b, r))), nil] &
:: nil :: [nil | -(b ; n(b, r)), +(n(b, r)), nil] &
:: r :: [nil | +(pk(i, b ; n(b, r))), nil] &
:: r :: [nil, -(pk(b, a ; N)), +(pk(a, N ; n(b, r)))
            | -(pk(b, n(b, r))), nil])
||
```

[11] Indeed, we use the fourth component of a protocol state to store the data related to the Super Lazy Intruder of Section 7.6.

```
pk(b, n(b, r)) !inI, pk(i, b ; n(b, r)) !inI, n(b, r) !inI,
(b ; n(b, r)) !inI
||
+(pk(i, b ; n(b, r))), -(pk(i, b ; n(b, r))), +(b ; n(b, r)),
-(b ; n(b, r)), +(n(b, r)), -(n(b, r)), +(pk(b, n(b, r))),
-(pk(b, n(b, r)))
||
nil
```

This state is unreachable, since there are two strands that are generating the same fresh variable `r` and this is impossible because the fresh data generated by each strand must be unique.

4.4 Attack States

Attack states describe not just single concrete attacks, but *attack patterns* (or if you prefer *attack situations*), which are specified symbolically as terms (with variables) whose instances are the final attack states we are looking for. Given an attack pattern, Maude-NPA tries to either find an instance of the attack pattern or prove that no such instance is possible. We can specify more than one attack state. Thus, we designate each attack state with a natural number.

When specifying an attack state, the user should specify only the first two components of the attack state: (i) a set of strands expected to appear in the attack, and (ii) some positive intruder knowledge. The other two state components should have just the empty symbol `nil`. Note that the attack state is indeed a term with variables; however, the user does not have to provide the variables denoting "the remaining strands", "the remaining intruder knowledge", and the two variables for the two last state components, since these variables are symbolically inserted by the tool (see Section 7.2 for technical details).

For NSPK, the standard attack is represented as follows:

```
eq ATTACK-STATE(0) =
  :: r ::
  [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))), -(pk(b,n(b,r))) | nil ]
  || n(b,r) inI
  || nil
  || nil
[nonexec] .
```

where we require the intruder to have learned the nonce generated by Bob. Therefore, we have to include Bob's strand in the attack in order to describe such specific nonce `n(b,r)`.

There are several possibilities to build an attack state by combining what the intruder learned at the attack state and what honest strands are expected to have occurred at the attack state. The most common situation is to include positive intruder facts of the form (m `inI`) and finished strands (i.e., strands that have the bar at the end). Note that an attack state can also contain negative intruder

facts of the form (m `!inI`) although this is not common. Partially executed strands of the form

$$:: r_1, \ldots, r_i :: [\ nil,\ {m_1}^{\pm},\ \ldots,\ {m_i}^{\pm} \mid {m_{i+1}}^{\pm},\ \ldots,\ {m_k}^{\pm},\ nil\ ]$$

are represented in attack states by truncated strands in which only the terms before the bar appear. For instance, a situation in which the intruder learns Bob's nonce without requiring Bob to finish the protocol, is represented as:

```
eq ATTACK-STATE(0) =
  :: r :: [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))) | nil ]
  || n(b,r) inI
  || nil
  || nil
[nonexec] .
```

The intruder knowledge can also contain *inequalities* (the condition that a term is not equal to some other term), which we can also include in the intruder knowledge component of an attack state. For example, suppose that we want to specify that a responder executes a strand, apparently with an initiator `a`, but the nonce received is not generated by `a`. This can be done as follows:

```
eq ATTACK-STATE(0) =
  :: r ::
  [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))), -(pk(b,n(b,r))) | nil ]
  || N != n(a,r')
  || nil
  || nil
[nonexec] .
```

where t `!=` s means that for any ground substitution θ applicable to t and s (i.e., $\theta(t)$ and $\theta(s)$ are ground terms), $\theta(t)$ cannot be equal to $\theta(s)$ modulo the equational theory. Note that, since a is a constant and r' is a variable of the special sort `Fresh`, `N != n(a,r')` means that `N` cannot be a term of the form `n(a,r')`.

In summary, we note the following requirements on attack state specifications:

1. Strands in an attack state must have the bar at the end. This also applies to partially executed strands, for which the future messages are discarded.
2. If more than one strand appears in the attack state, they must be separated by the `&` symbol. If more than one term appears in the intruder knowledge, they must be separated by commas. If no strands appear, or no intruder items appear, the `empty` symbol should be used, in the strands or intruder knowledge components, respectively.
3. Items that can appear in the intruder knowledge may include not only terms known by the intruder, but also inequality conditions on terms. Terms unknown to the intruder are also possible but are not common in attack states.
4. The last two fields of an attack state must always be `nil`. These are fields that contain information that is built up in the backwards search, but should be empty in the attack state.

4.5 Attack States With Excluded Patterns: Never Patterns

It is often desirable to exclude certain patterns from transition paths leading to an attack state. For example, one may want to determine whether or not authentication properties have been violated, e.g., whether it is possible for a responder strand to appear without the corresponding initiator strand. For this there is an optional additional field in the attack state containing the never patterns. It is included at the end of the attack state specification.

Here is how we would specify an initiator strand without a corresponding responder in the NSPK protocol[12]:

```
eq ATTACK-STATE(1)
  = :: r ::
    [ nil, -(pk(b,a ; N)), +(pk(a,  N ; n(b,r))), -(pk(b,n(b,r))) | nil ]
    ||  empty
    || nil
    || nil
    butNeverFoundAny *** for authentication
    (:: r' ::
    [ nil | +(pk(b,a ; N)), -(pk(a,  N ; n(b,r))), +(pk(b,n(b,r))), nil ]
    & S:StrandSet
    || K:IntruderKnowledge
    || M:SMsgList
    || G:GhostList)
  [nonexec] .
```

The tool will now look for all paths in which the intruder strand is executed, but the corresponding responder strand is not. That is, when we provide an attack state and some never patterns, a backwards reachability sequence leading to an initial state will not contain any state that matches any of the never patterns, where the place of the bar at each strand appearing in a never pattern is not taken into consideration for matching purposes.

It is also possible to use never patterns to specify negative conditions on terms or strands. Suppose that we want to ask whether it is possible for a responder in the NSPK protocol to execute a session of the protocol, apparently with an initiator, but the nonce received was not the initiator's. This can be done as follows:

```
eq ATTACK-STATE(1)
  = :: r ::
```

[12] Note that variable `r'` in the never pattern is not an error. The regular strand in the attack state is the initiator strand, which generates variable `r`. The strand in the never pattern is a pattern that must be valid for any responder strand. Any responder strand would generate his variable `r''`, which would be part of the variable `N` written in the never pattern if such a pattern is matched. Indeed, note that the variable `r` written in the never pattern is different by definition from the variable `r` written in the regular strand, since we are defining an actual strand and a pattern to be matched.

```
    [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))),
             -(pk(b,n(b,r))) | nil ]
    || empty
    || nil
    || nil
    butNeverFoundAny
   (  ::: r ::
    [ nil | -(pk(b,a ; n(a,r'))), +(pk(a, n(a,r') ; n(b,r))),
             -(pk(b,n(b,r))), nil ] & S:StrandSet
    || K:IntruderKnowledge
    || M:SMsgList
    || G:GhostList )
  [nonexec] .
```

It is possible to include more than one never pattern in the specification of an attack state, but then each such pattern must be contained within a pair of parentheses, e.g.,

```
 butNeverFoundAny
 ( ... State 1 ... )
 ( ... State 2 ... )
```

Never patterns can also be used to cut the down the search space. Suppose, for example, that one finds in the above search that a number of states are encountered in which the intruder encrypts two nonces, but they never seem to provide any useful information. One can reduce the search space by ruling out such intruder behavior with the second never pattern in the following attack state:

```
 eq ATTACK-STATE(1)
   = :: r ::
     [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))),
              -(pk(b,n(b,r))) | nil ]
     || empty
     || nil
     || nil
     butNeverFoundAny
     (:: r' :: [nil | +(pk(b,a ; N)), -(pk(a, N ; n(b,r))),
                      +(pk(b,n(b,r))), nil] & S:StrandSet
     || K:IntruderKnowledge
     || M:SMsgList
     || G:GhostList)
     (:: nil :: [nil | -(N1 ; N2), +(pk(B, N1 ; N2)), nil] & S:StrandSet
     || K:IntruderKnowledge
     || M:SMsgList
     || G:GhostList)
  [nonexec] .
```

Note that adding never patterns to reduce the search space, as distinguished from their use for verifying *authentication* properties, means that failure to find

an attack does not necessarily imply that the protocol is secure. It simply means that any attack against the security property specified in the attack state must use at least one strand that is specified in the set of never patterns.

There are several things about never patterns that should be noted:

1. The bar in any strand in a never pattern should be at the beginning of the strand. If it is not, the tool does not report any error and places the bar at the beginning.
2. Variables in a never pattern are renamed by the tool to assure that they never appear in the main attack state specification nor in other never patterns.
3. The last two fields in a never pattern must be variables of type `SMsgList`, and `Ghostlist`, respectively, as illustrated in the above examples.
4. The first two fields must end in variables of type `Strandset` and `IntruderKnowledge`, respectively.
5. More than one never pattern can be used in an attack state. However, each one must be delimited by its own pair of parentheses.

For a good example of the use of never patterns, which makes the Maude-NPA search considerably more efficient without compromising the completeness of the reachability analysis, we refer the reader to the analysis of the Diffie-Hellman protocol in [23].

4.6 Maude-NPA Commands for Attack Search

The commands `run`, `summary`, and `initials` are the tool's commands for attack search. They are invoked by reducing them in Maude, that is, by typing the Maude `red` command followed by the corresponding Maude-NPA command, followed by a space and a period. To use them we must specify the attack state we are searching for and the number of backwards reachability steps we want to compute. For example, the Maude-NPA command

```
run(0,10)
```

tells Maude-NPA to construct the backwards reachability tree up to depth 10 for the attack state designated with natural number 0. The Maude-NPA `run` command yields the set of states found at the leaves of the backwards reachability tree of the specified depth that has been generated. When the user is not interested in the current states of the reachability tree, he/she can use the Maude-NPA `summary` command, which outputs just the number of states found at the leaves of the reachability tree and how many of those are initial states, i.e., solutions for the attack. For instance, when we give the reduce command in Maude with the Maude-NPA command `summary(0,2)` as shown below for the NSPK example, the tool returns:

```
red summary(0,2) .
result Summary: States>> 4 Solutions>> 0
```

The initial state representing the standard NSPK attack is found in seven steps. That is, if we type

```
red summary(0,7) .
```

the tool outputs:

```
red summary(0,7) .
result Summary: States>> 3 Solutions>> 1
```

A slightly different version of the `run` command, called `initials`, outputs only the initial states, instead of all the states at the leaves of the backwards reachability tree. Thus, if we type

```
red initials(0,7) .
```

for the NSPK example, our tool outputs the following initial state[13], which implies that the attack state has been proved reachable and the protocol is insecure:

```
Maude> red initials(0,7) .
result IdSystem: < 1 . 5 . 2 . 7 . 1 . 4 . 3 . 1 > (
:: nil :: [nil | -(pk(i, n(b, #1:Fresh))), +(n(b, #1:Fresh)), nil] &
:: nil :: [nil | -(pk(i, a ; n(a, #0:Fresh))), +(a ; n(a, #0:Fresh)), nil] &
:: nil :: [nil | -(n(b, #1:Fresh)), +(pk(b, n(b, #1:Fresh))), nil] &
:: nil :: [nil | -(a ; n(a, #0:Fresh)), +(pk(b, a ; n(a, #0:Fresh))), nil] &
:: #1:Fresh :: [nil | -(pk(b, a ; n(a, #0:Fresh))),
 +(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))), -(pk(b, n(b, #1:Fresh))), nil] &
:: #0:Fresh :: [nil | +(pk(i, a ; n(a, #0:Fresh))),
 -(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))), +(pk(i, n(b, #1:Fresh))), nil])
||
pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh)) !inI,
  pk(b, n(b, #1:Fresh)) !inI,
  pk(b, a ; n(a, #0:Fresh)) !inI,
  pk(i, n(b, #1:Fresh)) !inI,
  pk(i, a ; n(a, #0:Fresh)) !inI,
  n(b, #1:Fresh) !inI,
  (a ; n(a, #0:Fresh)) !inI
||
+(pk(i, a ; n(a, #0:Fresh))),
  -(pk(i, a ; n(a, #0:Fresh))),
  +(a ; n(a, #0:Fresh)),
  -(a ; n(a, #0:Fresh)),
  +(pk(b, a ; n(a, #0:Fresh))),
  -(pk(b, a ; n(a, #0:Fresh))),
  +(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
  -(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
```

[13] Maude-NPA associates an identifier, e.g. 1.5.2.7.1.4.3.1, to each state generated by the tool. These identifiers are for internal use and are not described here.

```
  +(pk(i, n(b, #1:Fresh))),
  -(pk(i, n(b, #1:Fresh))),
  +(n(b, #1:Fresh)),
  -(n(b, #1:Fresh)),
  +(pk(b, n(b, #1:Fresh))),
  -(pk(b, n(b, #1:Fresh)))
||
nil
```

This corresponds to the following textbook version of the attack:

1. $A \to I : pk(I, A; N_A)$
2. $I_A \to B : pk(B, A; N_A)$
3. $B \to A : pk(A, N_A; N_B)$, intercepted by I;
4. $I \to A : pk(A, N_A; N_B)$
5. $A \to I : pk(I, N_B)$
6. $I_A \to B : pk(B, N_B)$

It is also possible to generate an unbounded search by specifying the second argument of `run`, `initials`, or `summary` as `unbounded`. In that case the tool will run until it has shown that all the paths it has found either begin in initial states or in unreachable ones. This check may terminate in finite time, but in some cases may run forever.

We demonstrate this unbounded search with NSPK:

```
red summary(0,unbounded) .
result Summary: States>> 1 Solutions>> 1
```

This tells us, that Maude-NPA terminated with only one attack. If we want to see what that attack looks like, we would instead type

```
red run(0,unbounded) .
```

to get the attack displayed above.

The complete analysis of the Diffie-Hellman protocol can be found in [23], including the initial state proving that the protocol in insecure.

5 How Maude-NPA Works: Backwards Reachability

First, we recall some definitions on rewriting and narrowing in Section 5.1. Then, we explain in Section 5.2 how the backwards reachability analysis works in practice.

5.1 Rewriting and Narrowing

Continuing Section 2, we recall some definitions on term rewriting and illustrate each definition by means of examples.

A *rewrite rule* is an oriented pair of terms (l, r), written $l \to r$, where $l \notin \mathcal{X}$ and $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathtt{s}}$ for some sort $\mathtt{s} \in \mathtt{S}$. An *(unconditional) order-sorted rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$ with Σ an order-sorted signature, E a set of Σ-equations, and R a set of rewrite rules. The rewriting relation $\to_R$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \xrightarrow{p}_R t'$ (or $\to_R$) if $p \in Pos_\Sigma(t)$, $l \to r \in R$, $t|_p = \sigma(l)$, and $t' = t[\sigma(r)]_p$ for some σ. The relation $\to_{R/E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $=_E; \to_R; =_E$, where $\_;\_$ denotes relation composition. The relation $\to_{R/E}$ is much harder to implement and the weaker rewrite relation $\to_{R,E}$ is usually provided, as it happens in Maude. Assuming a finitary and complete matching algorithm modulo the equational theory E, the rewriting relation $\to_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is defined as $t \xrightarrow{p}_{R,E} t'$ (or $\to_{R,E}$) if $l \to r \in R$, $t|_p =_E \sigma(l)$, and $t' = t[\sigma(r)]_p$.

Maude supports matching modulo any combination of free, associative, commutative, associative-commutative, and associative-commutative with identity symbols, so that we in effect can rewrite terms *modulo* any combination of such axioms. Suppose, for example, an unconditional order-sorted rewrite theory $\mathcal{R} = (\Sigma, E, R)$ where Σ contains an infix symbol $+$, E contains the commutativity property for symbol $+$, and R contains a rule of the form $X + 0 = X$. Then we can apply such a rule to the term $0 + 7$ *modulo* commutativity, even though the constant 0 is on the left of the $+$ symbol. That is, the term $0 + 7$ *matches* the left-hand side pattern $X + 0$ *modulo* commutativity, i.e., $0 + 7 =_E (X + 0)\sigma$ where $\sigma = \{X \mapsto 7\}$. We would express this rewrite step modulo commutativity with the arrow notation:

$$0 + 7 \to_{R,E} 7$$

Likewise, we denote by $\to^*_{R,E}$ the reflexive-transitive closure of the one-step rewrite relation $\to_{R,E}$ with the rules R modulo the axioms E. That is, $\to^*_{R,E}$ corresponds to taking zero, one, or more rewrite steps with the rules R modulo E.

Narrowing generalizes term rewriting by allowing free variables in terms (as in logic programming) and by performing unification instead of matching in order to (non–deterministically) reduce a term. Intuitively, the difference between a rewriting step and a narrowing step is that in both cases we use a rewrite rule $l \to r$ to rewrite t at a position p in t, but narrowing finds values for the variables in the chosen subject term $t|_p$ before actually performing the rewriting step. The narrowing relation $\rightsquigarrow_R$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \overset{p}{\rightsquigarrow}_{\sigma,R} t'$ (or $\rightsquigarrow_{\sigma,R}$, $\rightsquigarrow_R$) if $p \in Pos_\Sigma(t)$, $l \to r \in R$, $\sigma \in CSU_\emptyset(t|_p = l)$, and $t' = \sigma(t[r]_p)$. Assuming that E has a finitary and complete unification algorithm, the narrowing relation $\rightsquigarrow_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \overset{p}{\rightsquigarrow}_{\sigma,R,E} t'$ (or $\rightsquigarrow_{\sigma,R,E}$, $\rightsquigarrow_{R,E}$) if $p \in Pos_\Sigma(t)$, $l \to r \in R$, $\sigma \in CSU_E(t|_p = l)$, and $t' = \sigma(t[r]_p)$.

Maude supports unification modulo different user-defined theories including any combination of free, commutative, and associative-commutative symbols; see Section 6.3 for the algebraic theories admissible for such equational unification procedure, and Section 6.4 for some examples of admissible theories. Suppose, for example, the same unconditional order-sorted rewrite theory used above for rewriting. Then, we can apply the rule $X + 0 = X$ above to the term $Z + 7$,

where X is a variable, *modulo* commutativity. That is, the term $Z+7$ *unifies* the left-hand side pattern $X+0$ *modulo* commutativity, i.e., $(Z+7)\theta =_E (X+0)\theta$ where $\theta = \{Z \mapsto 0, X \mapsto 7\}$. We would express this narrowing step modulo commutativity with the arrow notation:

$$Z + 7 \leadsto_{\theta,R,E} 7$$

We denote by $\leadsto^*_{\theta,R,E}$ the reflexive-transitive closure of the one-step narrowing relation $\leadsto_{R,E}$ with the rules R modulo the axioms E, where θ is the composition of all the unifiers obtained during the sequence.

In Maude-NPA, narrowing is used at two levels: for backwards reachability analysis from a term with variables (i.e., an attack state), and for equational unification. We postpone until Section 6 how the narrowing-based equational unification is performed in Maude-NPA.

5.2 Backwards Reachability Analysis

Given a protocol $\mathcal{P}$ and an equational theory $E_\mathcal{P}$, Maude-NPA performs backwards narrowing reachability analysis from a state St representing an attack pattern (i.e., a term with variables) using the relation $\leadsto_{R^{-1}_\mathcal{P},E_\mathcal{P}}$, where the rewrite rules $R_\mathcal{P}$ are obtained from the protocol strands $\mathcal{P}$. The rewrite theory $\mathcal{R}_\mathcal{P} = (\Sigma, E_\mathcal{P}, R_\mathcal{P})$ is *topmost*, i.e., there is a sort `State` such that for each $l \to r \in R_\mathcal{P}$, $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_{\texttt{State}}$, $r \notin \mathcal{X}$, and no operator in Σ has `State` as an argument sort. Intuitively, a topmost rewrite theory describes a specification of a system such that a rewrite or narrowing step can only be performed at the root of each term denoting a state. The fact that $\mathcal{R} = (\Sigma, E_\mathcal{P}, R_\mathcal{P})$ is a topmost rewrite theory has several advantages:

1. The rewriting relation $\to_{R_\mathcal{P}/E_\mathcal{P}}$ can be safely simulated by the rewriting relation $\to_{R_\mathcal{P},E_\mathcal{P}}$.
2. Similarly, the narrowing relation $\leadsto_{R_\mathcal{P},E_\mathcal{P}}$ achieves the same effect as a more general narrowing relation $\leadsto_{R_\mathcal{P}/E_\mathcal{P}}$ (see [48]).
3. We obtain the following *completeness* result between narrowing ($\leadsto_{R_\mathcal{P},E_\mathcal{P}}$) and rewriting ($\to_{R_\mathcal{P}/E_\mathcal{P}}$):

   Theorem 1 (Topmost Completeness). [48] *Let $\mathcal{R} = (\Sigma, E, R)$ be a topmost rewrite theory such that E has a finitary unification algorithm, $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$, and let σ be a substitution such that $\sigma(t) \to^*_{R/E} t'$. Then, there are substitutions θ, τ and a term t'' such that $t \leadsto^*_{\theta,R,E} t''$, $\sigma(t) =_E \tau(\theta(t))$, and $t' =_E \tau(t'')$.*

A *state* in the protocol execution is a set of Maude-NPA strands unioned together with an associative and commutativity union operator _&_ with identity operator `empty`, along with an additional term describing the intruder knowledge at that point. Note that two extra state components are associated to a Maude-NPA state in Section 4, but they are irrelevant and useful only for user debugging of the protocol, so we omit them in this section and in Section 7.

The *intruder knowledge* is represented as a set of facts unioned together with an associative and commutativity union operator `_,_` with identity operator `empty`. There are two kinds of intruder facts: positive knowledge facts (the intruder knows m, i.e., (m `inI`)), and negative knowledge facts (the intruder does not yet know m but will know it in a future state, i.e., (m `!inI`)), where m is a message expression.

Strands communicate between them via a unique shared channel, i.e., by sending messages to the channel and retrieving messages from the channel. However, we do not explicitly represent the channel in our model. Instead, since the intruder is able to learn any message present in the channel, we use the intruder knowledge as the channel. When the intruder observes a message in the channel, then he/she learns it. The intruder has the usual ability to read and redirect traffic, and can also perform operations, e.g., encryption, decryption, concatenation, exclusive or, exponentiation, etc., on messages that it has received. As explained in Section 3, the nature and algebraic properties of such operations depend on the given cryptographic theory $E_{\mathcal{P}}$. Intruder operations are described in terms of the intruder sending messages to itself, which are represented as different strands, one for each action. All intruder and protocol strands are described symbolically, using a mixture of variables and constants, so a single state specification as a term with variables symbolically represents many concrete state instances. There is no restriction on the number of principals, number of sessions, nonces, or time, i.e., no data abstraction or approximation is performed.

The user can make use of a special sort `Fresh` in the protocol-specific signature Σ for representing fresh unguessable values, e.g., for nonces. The meaning of a variable of sort `Fresh` is that it will never be instantiated by an E-unifier generated during the backwards reachability analysis. This ensures that if nonces are represented using variables of sort `Fresh`, they will never be merged and no approximation for nonces is necessary. We make the `Fresh` variables generated by a strand explicit by writing $(r_1, \ldots, r_k : \texttt{Fresh})\ [msg_1^{\pm}, \ldots, msg_n^{\pm}]$, where $r_1, \ldots, r_k$ are all the variables of sort `Fresh` generated by $msg_1^{\pm}, \ldots, msg_n^{\pm}$. If it does not generate any fresh variable, we add nothing to the strand. As explained in Section 3, the types and algebraic properties of the operators used in messages (cryptographic and otherwise) are described as an equational theory $E_{\mathcal{P}}$.

In principle, the rewrite rules $R_{\mathcal{P}}$ obtained from the protocol strands $\mathcal{P}$ are represented by the following rewrite rules[14]:

$$SS\,\&\,[L \mid M^-, L']\,\&\,((M\ \texttt{inI}), IK) \to SS\,\&\,[L, M^- \mid L']\,\&\,((M\ \texttt{inI}), IK) \quad (1)$$

$$SS\,\&\,[L \mid M^+, L']\,\&\,IK \to SS\,\&\,[L, M^+ \mid L']\,\&\,IK \quad (2)$$

$$SS\,\&\,[L \mid M^+, L']\,\&\,IK \to SS\,\&\,[L, M^+ \mid L']\,\&\,((M\ \texttt{inI}), IK) \quad (3)$$

[14] The top level structure of the state is a multiset of strands formed with the _&_ union operator. The protocol and intruder rewrite rules are "essentially topmost" in that, using an extension variable matching the "remaining strands" they can always rewrite the whole state. Therefore, as explained in [48], completeness results of narrowing for topmost theories also apply to them.

In a *forward execution* of the protocol strands, Rule (1) synchronizes an input message with a message already learned by the intruder, Rule (2) accepts output messages but the intruder's knowledge is not increased, and Rule (3) accepts output messages and the intruder's knowledge is positively increased. New strands will be added to the state by explicit introduction through dedicated rewrite rules (one for each honest or intruder strand), e.g.

$$SS \,\&\, ((sk(i,M)\ \texttt{inI}), IK) \rightarrow SS \,\&\, [M^- \mid sk(i,M)^+] \,\&\, ((sk(i,M)\ \texttt{inI}), IK)$$

However, we are interested in a *backwards* execution of the protocol from an attack state that contains *positive* facts in the intruder knowledge. Therefore, the rule increasing the intruder knowledge, Rule (3), must make explicit *when* the intruder learned message M:

$$SS \,\&\, [L \mid M^+, L'] \,\&\, ((M\ \texttt{!inI}), IK) \rightarrow SS \,\&\, [L, M^+ \mid L'] \,\&\, ((M\ \texttt{inI}), IK) \qquad (4)$$

Note that this is recorded in the previous state by the negative fact (M !inI), which can be paraphrased as: "the intruder does not yet know M, but will learn it in the future".

For the introduction of new additional strands, we simply record when the intruder learned message M in each of the rewrite rules explicitly introducing new strands but, since they are applied backwards, reversing the introduction of the new strand:

$$\{\ [l_1 \,|\, u^+,\, l_2\,] \,\&\, \{(u\ \texttt{!inI}), K\} \rightarrow \{(u\ \texttt{inI}), K\} \mid [\,l_1,\ u^+,\ l_2\,] \in \mathcal{P}\} \qquad (5)$$

For example, the following Dolev-Yao action:

$$SS \,\&\, [M^- \mid sk(i,M)^+] \,\&\, ((sk(i,M)\ \texttt{!inI}), IK) \rightarrow SS \,\&\, ((sk(i,M)\ \texttt{inI}), IK) \qquad (6)$$

Thus, we can conclude that the set of rewrite rules obtained from the protocol strands that are used for backwards narrowing reachability analysis is $R_{\mathcal{P}} = \{(1), (2), (4)\} \cup (5)$.

For example, consider the the NSPK attack state:

$$(r : \texttt{Fresh})[\ pk(b, a; N)^-,\ pk(a, N; n(b,r))^+,\ pk(b, n(b,r))^-\ \mid nil\ ]\ \&\ (n(b,r)\ \texttt{inI})$$

and the Rule (6). We can apply this rule to our attack state modulo the equational theory for NSPK. That is, the attack term above unifies with the left-hand side of the rule modulo the following cancellation equational rules $E_{\mathcal{P}}$:

$$pk(A, sk(A, Z)) = Z \qquad\qquad sk(A, pk(A, Z)) = Z$$

To be more concrete, the term $n(b,r)$ unifies[15] with the term $sk(i,M)$ modulo the cancellation equational rules yielding the unifier $\theta = \{M \mapsto pk(i, n(b,r))\}$.

[15] This equational unification procedure is also performed by narrowing and is explained in Section 6 below.

Therefore, we obtain the following predecessor state using the narrowing relation $\leadsto_{\theta, R_{\mathcal{P}}^{-1}, E_{\mathcal{P}}}$ from the NSPK attack state:

$$[\ pk(i, n(b,r))^- \mid n(b,r)^+\ ]\ \&$$
$$(r : \texttt{Fresh})[\ pk(b, a; N)^-,\ pk(a, N; n(b,r))^+,\ pk(b, n(b,r))^- \mid nil\ ]\ \&\ (n(b,r)\ \texttt{!inI})$$

In a backwards execution of the protocol using narrowing we start from an attack pattern, i.e., a term with variables, containing: (i) some of the strands of the protocol to be executed backwards, (ii) a variable SS denoting a set of additional strands, (iii) some terms the intruder knows at the attack state, i.e., of the form $(t\ \texttt{inI})$, and (iv) a variable IK denoting a set of additional intruder facts. We then perform narrowing with the rules $R_{\mathcal{P}}$ in reverse to move the bars of the strands to the left until either we find an initial state, or cannot perform any other useful backwards narrowing steps. Note that variables SS and IK will be instantiated by backwards narrowing to additional strands and additional intruder facts, respectively, in order to find an initial state. Indeed, these variables SS and IK are not required for the specification of the attack state as explained in Section 4.4; see Section 7.2 for further technical details.

6 How Maude-NPA Works: Equational Unification

Sound techniques for equational unification are paramount to Maude-NPA, since it performs symbolic reachability analysis *modulo* the equational theory of the protocol. This makes Maude-NPA verification much stronger than verification methods based on a purely syntactic view of the algebra of messages as a term algebra using the standard Dolev-Yao model of perfect cryptography in which no algebraic properties are assumed. Indeed, it is well-known that various protocols that have been proved secure under the standard Dolev-Yao model can be broken by an attacker who exploits the algebraic properties of some cryptographic functions (see, e.g., [41,43,46]).

In the standard Dolev-Yao model, symbolic reachability analysis typically takes the form of representing sets of states symbolically as terms with logical variables, and then performing *syntactic unification* with the protocol rules to explore reachable states. This can be done in either a forwards or a backwards fashion. In the Maude-NPA (which can also be used for analyses under the standard Dolev-Yao model when no algebraic properties are specified) symbolic reachability analysis is performed in a *backwards* fashion (as already explained in Section 5), beginning with a symbolic representation of an attack state, and searching for an initial state, which then provides a proof that an attack is possible; or a proof that no such attack is possible if all such search paths fail to reach an initial state.

However, if the Maude-NPA analyzes a protocol for which algebraic properties *have* been specified by an equational theory T, the same symbolic reachability analysis is performed in the same fashion, but now *modulo* T. What this means precisely is that, instead of performing syntactic unification between a

term representing symbolically a set of states and the righthand-side (in the backwards reachability case) of a protocol rule, we now perform *equational unification* with the theory T, (also called T-*unification*, or *unification modulo* T) between the same term and the same righthand side of a protocol rule. In what follows we explain several things regarding T-unification in the Maude-NPA:

- In Section 6.1, equational axioms for which the Maude-NPA provides built-in support for equational unification.
- In Section 6.2, narrowing-based equational unification in general, which is however infeasible for Maude-NPA analysis when the number of unifiers generated is infinite.
- In Section 6.3, the most general class of equational theories for which the Maude-NPA can currently support unification by narrowing, with the important requirement of the number of unifier solutions being *finite*. These are called *admissible* theories.
- Some examples of admissible theories in Section 6.4.

6.1 Built-in support for Unification Modulo Equational Axioms

The Maude-NPA has built-in support for unification modulo certain equational theories T thanks to the underlying Maude infrastructure. Specifically, the Maude-NPA automatically supports unification modulo T for T any order-sorted theory of the form $T = (\Sigma, Ax)$, where Σ is a *signature* declaring sorts, subsorts, and function symbols (as we have already illustrated with examples in Section 3.2) and Ax is a collection of equational *axioms* where some binary operators f in the signature Σ may have axioms in Ax for either *commutativity* ($f(x, y) = f(y, x)$), or commutativity and *associativity* ($f(x, f(y, z)) = f(f(x, y), z)$). Associativity alone is not supported, because it is well-known that unification problems modulo associativity may have an *infinite* number of unifiers (see, e.g., [5]). As already illustrated in Section 3.4, the way associativity, and/or commutativity axioms are specified in Maude for a function symbol f is not by giving those axioms explicitly, but by declaring f in Maude with the `assoc` and/or `comm` attributes. For example a function symbol `f` of sort `S` which is associative and commutative is specified in Maude as follows:

```
op f : S S -> S [assoc comm] .
```

6.2 Narrowing-Based Equational Unification and its Limitations

Of course, many algebraic theories T of interest in protocol analysis fall outside the scope of the above-mentioned class of theories T based on combinations of associativity and/or commutativity axioms, for which the Maude-NPA provides automatic built-in support. Therefore, the burning issue is how to support more general classes of algebraic theories in the Maude-NPA.

In this regard, a very useful, generic method to obtain T-unification algorithms is narrowing [29,31]. Mathematically, an algebraic theory T is a pair of

the form $T = (\Sigma, E \cup Ax)$, where Σ is a signature declaring sorts, subsorts, and function symbols, and where $E \cup Ax$ is a set of *equations*, that we assume is split into a set Ax of equational axioms such as our previous combinations of associativity and/or commutativity axioms, and a set E of oriented equations to be used from left to right as rewrite rules. In Maude, the axioms Ax are declared by means of the `assoc` and/or `comm` attributes. Instead, the equations E are declared with the `eq` keyword as we have illustrated with examples in Section 3.4.

An algebraic theory $T = (\Sigma, E \cup Ax)$ must satisfy the following four requirements:

1. The axioms Ax can declare some binary operators in Σ to be commutative (with the `comm` attribute), or associative-commutative (with the `assoc` and `comm` attributes).
2. The equations E are confluent modulo Ax.
3. The equations E are terminating modulo Ax.
4. The equations E are coherent modulo Ax.

We now explain in detail what these requirements mean. First, the equational theory $T = (\Sigma, E \cup Ax)$ is viewed as an unconditional order-sorted *rewrite theory* $\mathcal{R}_T = (\Sigma, Ax, E)$, so that we perform rewrite steps with the rewrite rules E modulo Ax, i.e., the reduction relation $\rightarrow_{E,Ax}$ explained in Section 5.1, where, in the notation $\rightarrow_{R,E}$, now $R = E$ and $E = Ax$.

Confluence. The equations E are called *confluent* modulo Ax if and only if for each term t in the theory $T = (\Sigma, E \cup Ax)$, if we can rewrite t with E modulo Ax in two different ways as: $t \longrightarrow^*_{E,Ax} u$ and $t \longrightarrow^*_{E,Ax} v$, then we can always further rewrite u and v to a common term up to identity modulo Ax. That is, we can always find terms u', v' such that:

- $u \longrightarrow^*_{E,Ax} u'$ and $v \longrightarrow^*_{E,Ax} v'$, and
- $u' =_{Ax} v'$

That is, u' and v' are essentially the same term, in the sense that they are equal modulo the axioms Ax. In the example of Section 5.1, we have, for instance, $0 + 7 =_{Ax} 7 + 0$.

Termination. The equations E are called *terminating* modulo Ax if and only if all rewrite sequences terminate; that is, if and only if we never have an infinite sequence of rewrites

$$t_0 \rightarrow_{E,Ax} t_1 \rightarrow_{E,Ax} t_2 \ldots t_n \rightarrow_{E,Ax} t_{n+1} \ldots$$

Coherence. Rather than explaining the *coherence modulo* Ax notion in general (the precise definition of the general notion can found in [31]), we explain in detail its meaning in the case where it is needed for the Maude-NPA, namely,

the case of associative-commutative (AC) symbols. The best way to illustrate the meaning of coherence is by its *absence*. Consider, for example, an exclusive or operator $\oplus$ which has been declared AC. Now consider the equation $X \oplus X = 0$. This equation, if not completed by another equation, is *not* coherent modulo AC. What this means is that there will be term *contexts* in which the equation *should* be applied, but it cannot be applied. Consider, for example, the term $b \oplus (a \oplus b)$. Intuitively, we should be able to apply to it the above equation to simplify it to the term $a \oplus 0$ in one step. However, since we are using the weaker rewrite relation $\rightarrow_{E,Ax}$ instead of the stronger but much harder to implement relation $\rightarrow_{E/Ax}$, we cannot! The problem is that the equation cannot be applied (even if we match modulo AC) to either the top term $b \oplus (a \oplus b)$ or the subterm $a \oplus b$. We can however make our equation *coherent* modulo AC by adding the extra equation $X \oplus X \oplus Y = 0 \oplus Y$, which, using also the equation $X \oplus 0 = X$, we can slightly simplify to the equation $X \oplus X \oplus Y = Y$. This second variant of our equation will now apply to the term $b \oplus (a \oplus b)$, giving the simplification $b \oplus (a \oplus b) \longrightarrow_{E,Ax} a$. Technically, what coherence means is that the weaker relation $\rightarrow_{E,Ax}$ becomes semantically equivalent to the stronger relation $\rightarrow_{E/Ax}$.

For the Maude-NPA, coherence is only an issue for AC symbols. And there is always an easy way, given a set E of equations, to make them AC-coherent. The method is as follows. For any symbol f which is AC, and for any equation of the form $f(u, v) = w$ in E, we add also the equation $f(f(u, v), X) = f(w, X)$, where X is a new variable not appearing in u, v, w. In an order-sorted setting, we should give to X *the biggest sort possible*, so that it will apply in all generality. As an additional optimization, note that some equations may already be coherent modulo AC, so that we need not add the extra equation. For example, if the variable X has the biggest possible sort it could have, then the equation $X \oplus 0 = X$ is already coherent, since X will match "the rest of the $\oplus$-expression," regardless of how big or complex that expression might be, and of where in the expression a constant 0 occurs. For example, this equation will apply modulo AC to the term $(a \oplus (b \oplus (0 \oplus c))) \oplus (c \oplus a)$, with x matching the term $(a \oplus (b \oplus c)) \oplus (c \oplus a)$, so that we indeed get a rewrite $(a \oplus (b \oplus (0 \oplus c))) \oplus (c \oplus a) \rightarrow_{E,Ax} (a \oplus (b \oplus c)) \oplus (c \oplus a)$.

Limitations. Although narrowing is a very general method to generate T-unification algorithms, general narrowing has a serious limitation. The problem is that, in general, narrowing with an equational theory $T = (\Sigma, E \cup Ax)$ satisfying requirements (1)–(4) above may yield an *infinite* number of unifiers. Since, for T the algebraic theory of a protocol, T-unification must be performed by the Maude-NPA at each *single step* of symbolic reachability analysis, narrowing is in general not practical as a unification procedure, unless the theory T satisfies the additional requirement that there always exists a *finite* set of unifiers that provide a complete set of solutions; and that such a finite set of solutions can be effectively computed by narrowing. We discuss this extra important requirement in what follows.

6.3 General Requirements for Algebraic Theories

The Maude-NPA's unification technique is based on narrowing and in order to provide a *finite* set of unifiers, five specific requirements must be met by any algebraic theory specifying cryptographic functions that the user provides. If these requirements are not satisfied, Maude-NPA may exhibit non-terminating and/or incomplete behavior, and any completeness claims about the results of the analysis cannot be guaranteed. We call theories that satisfy these criteria *admissible* theories. We show in Section 6.4 how all the examples presented in Section 3.4 are admissible theories.

In the Maude-NPA we call an algebraic theory $T = (\Sigma, E \cup Ax)$ specified by the user for the cryptographic functions of the protocol *admissible* if it satisfies the four requirements specified in Section 6.2 and *strongly right irreducibility*, i.e., the following five requirements:

1. The axioms Ax can declare some binary operators in Σ to be commutative (with the `comm` attribute), or associative-commutative (with the `assoc` and `comm` attributes). No other combinations of axioms are allowed; that is, a function symbol has either no attributes, or only the `comm` attribute, or only the `assoc` and `comm` attributes.[16]
2. The equations E are confluent modulo Ax.
3. The equations E are terminating modulo Ax.
4. The equations E are coherent modulo Ax.
5. The equations E are strongly right irreducible.

Confluence, termination, and coherence were explained in Section 6.2. We now explain in detail what strongly right irreducibility means. Given a theory $T = (\Sigma, E \cup Ax)$, which we assume satisfies requirements (1)-(4) above, we call the set E of equations *strongly right irreducible* iff for each equation $t = t'$ in E, we cannot further simplify by the equations E modulo Ax either the term t', or any substitution instance $\theta(t')$ where the terms in the substitution θ cannot themselves be further simplified by the equations E modulo Ax. Obvious cases of such righthand-sides t' include:

- a single variable;
- a constant for which no equations exist;
- more generally, a *constructor term*, i.e., a term whose function symbols have no associated equations.

But these are not the only possible cases where strong right irreducibility can be applied. Typing, particularly the use of sorts and subsorts in an order-sorted equational specification, can greatly help in attaining strong right irreducibility. We refer the reader to [21,19] for two examples of how order-sorted typing helps narrowing-based unification to become finitary. We discuss one of these examples, namely Diffie-Hellman, in the following section.

[16] In future versions of the Maude-NPA we also plan to support operators having the `assoc`, `comm` and `id` attributes, adding an identity axiom. At present, identity axioms in Maude-NPA theories should only be defined by means of equations.

Finally, let us motivate with an example how this narrowing-based equational unification works. Consider the exclusive-or theory:

$$\begin{aligned} X \oplus X \oplus Y &= Y \\ X \oplus X &= 0 \\ X \oplus 0 &= X \end{aligned}$$

viewed as the order-sorted rewrite theory $\mathcal{R} = (\Sigma, Ax, E)$ where Ax contains the associativity and commutativity of $\oplus$ and E contains the three equational rules above. Given the equational problem $t \stackrel{?}{=} s$, where $t = X \oplus Y$ and $s = U \oplus V$, we perform narrowing on t with the equations E modulo the axioms Ax. We can apply a renamed version (e.g. $Z \oplus Z \oplus W$) of the first equation to the term $X \oplus Y$. That is, the term $X \oplus Y$ unifies modulo AC with the term $Z \oplus Z \oplus W$ and one of the unifiers is $\sigma = \{X \mapsto W' \oplus Z', Y \mapsto Z', W \mapsto W', Z \mapsto Z'\}$. We express this narrowing step as

$$X \oplus Y \leadsto_{\sigma, E, Ax} W'$$

Note that, according to [27], we simply unify the terms W' and $U \oplus V$ modulo AC to finish the equational unification procedure, obtaining $\sigma' = \{X \mapsto (U \oplus V) \oplus Z', Y \mapsto Z'\}$ as an equational unifier of terms t and s modulo the equational theory for exclusive-or. This unifier is not necessarily the only one possible: given an equation $t = t'$, by successive narrowing of t and t' followed by attempts of Ax-unification, other unifiers can be computed. However, since the exclusive or theory satisfies conditions (1)-(5) above, we are guaranteed that there will be a finite complete set of unifiers obtained this way by narrowing.

6.4 Some Examples of Admissible Theories

Since any user of the Maude-NPA should write specifications whose algebraic theories are admissible, i.e., satisfy requirements (1)–(5) in Section 6.3, it may be useful to illustrate how these requirements are met by several examples. This can give a Maude-NPA user a good intuitive feeling for how to specify algebraic theories that the Maude-NPA currently can handle. For this purpose, we revisit the theories already discussed in Section 3.4.

Let us begin with the theory of Encryption/Decryption:

```
var Z : Msg .
var A : Name .

*** Encryption/Decryption Cancellation
eq pk(A,sk(A,Z)) = Z [nonexec] .
eq sk(A,pk(A,Z)) = Z [nonexec] .
```

In this case $Ax = \emptyset$. It is obvious that in this case the equations E *terminate*, since the *size* of a term as a tree (number of nodes) strictly decreases after the application of any of the above two rules, and therefore it is impossible to have an

infinite chain of rewrites with the above equations. It is also easy to check that the equations are *confluent*: by the termination of E this can be reduced to checking confluence of critical pairs, which can be easily discharged by automated tools [16], or even by hand. Since $Ax = \emptyset$, coherence is a mute point. The equations are also *strongly right irreducible*, because in both cases they are the variable `Z`, and any instance of `Z` by a term that cannot be further simplified by the above equations, obviously cannot be further simplified by hypothesis.

Let us now consider the Exclusive Or Theory:

```
--- XOR operator
op _<+>_ : Msg Msg -> Msg [frozen assoc comm] .
op null : -> Msg .

vars X Y : Msg .

--- XOR equational properties
eq X <+> X <+> Y = Y [nonexec]  .
eq X <+> X = null [nonexec]  .
eq X <+> null = X [nonexec]  .
```

In this case $Ax = AC$. *Termination* modulo AC is again trivial, because the size of a term strictly decreases after applying any of the above equations modulo AC. Because of termination modulo AC, *confluence* modulo AC can be reduced to checking confluence of critical pairs, which can be discharged by standard tools [16]. *Coherence* modulo AC is also easy. As already explained, the first equation has to be added to the second to make it coherent. As also explained above, since the sort `Msg` is the biggest possible for the exclusive or operator, the variable `X` in the last equation has the biggest possible sort it can have, and therefore that equation is already coherent, so that there is no need to add an extra equation of the form

```
eq X <+> null <+> Y = X <+> Y [nonexec]  .
```

because modulo AC such an equation is in fact *an instance* of the third equation (by instantiating `X` to the term `X <+> Y`). Finally, *strong right irreducibility* is also obvious, since the righthand sides are either variables, or the constant `null`, for which no equations exist.

Turning now to the Diffie-Hellman theory we have:

```
sorts Name NeNonceset Nonce Gen  Exp GenvExp Enc .
subsort Name NeNonceset Enc  Exp < Msg .
subsort Nonce < NeNonceset .
subsort Gen Exp < GenvExp .
subsort Name  Gen < Public .

op g : -> Gen  [frozen] .
op exp : GenvExp NeNonceSet -> Exp [frozen] .
op _*_ : NeNonceSet NeNonceSet -> NeNonceSet [frozen assoc comm] .
```

```
eq exp(exp(W:Gen,Y:NeNonceSet),Z:NeNonceSet)
 = exp(W:Gen, Y:NeNonceSet * Z:NeNonceSet) [nonexec] .
```

Again, this theory is AC. *Termination* modulo AC is easy to prove by using a polynomial ordering with AC polynomial functions (see [47]). For example, we can associate to `exp` the polynomial $x + y + 1$, and to `*` the polynomial $x + y$. Then the proof of termination becomes just the polynomial inequality $w + y + z + 2 > w + y + z + 1$. Because of termination modulo AC, *confluence* modulo AC can be reduced to checking the confluence of critical pairs. Here things become interesting. In an untyped setting, the above equation would have a nontrivial overlap with itself (giving rise to a critical pair), by unifying the lefthand side with the subterm `exp(W:Gen,Y:NeNonceSet)`. However, because of the subsort and operator declarations

```
subsort Gen Exp < GenvExp .
op exp : GenvExp NeNonceSet -> Exp [frozen] .
```

we see that the order-sorted unification of the subterm `exp(W:Gen,Y:NeNonceSet)` (which has sort `Exp`) and the lefthand side now *fails*, because the sorts `Gen` and `Exp` are mutually exclusive and cannot have any terms in common. Therefore there are no nontrivial critical pairs and the equation is confluent modulo AC. *Coherence* modulo AC is trivially satisfied, because the top operator of the equation (`exp`) is not an AC operator. As in the case of confluence modulo AC, the remaining issue of *strong right irreducibility* becomes particularly interesting in the order-sorted context. Note that in an untyped setting, an instance of the righthand side by applying a substitution whose terms cannot be further simplified *could* itself be simplified. For example, if we consider the untyped righthand side term `exp(W, Y * Z)`, the substitution θ mapping `W` to `exp(Q,X)` and being the identity on `Y` and `Z` is itself irreducible by the equations, but when applied to `exp(W, Y * Z)` makes the corresponding instance reducible by the untyped version of the above equation. However, in the order-sorted setting in which our equation is defined, the equation is indeed strongly right irreducible. This is again because the sorts `Gen` and `Exp` are mutually exclusive and cannot have any terms in common, so that the variable `W:Gen` cannot be instantiated by any term having `exp` as its top operator.

It may perhaps be useful to conclude this section with an example of an algebraic theory that *cannot* be supported in the current version of Maude-NPA. Consider, the extension of the above exclusive or theory in which we add a *homomorphism* operator and the obvious homomorphism equation:

```
op h : Msg -> Msg .

vars X Y : Msg .

eq h(X <+> Y) = h(X) <+> h(Y) [nonexec] .
```

The problem now is that the righthand side `h(X) <+> h(Y)` *fails* to be strongly right-irreducible. For example, the substitution θ mapping `X` to `U <+> V` and `Y` to

`Y` is itself irreducible, but produces the instance `h(U <+> V) <+> h(Y)`, which is obviously reducible. Since strong irreducibility is only a *sufficient condition* for narrowing-based equational unification to be finitary, one could in principle hope that this homomorphism example might still have a narrowing-based finitary algorithm[17]. However, the hopes for such a finitary narrowing-based algorithm are dashed to the ground by results in both [15], about the homomorphism theory not having the "finite variant" property, and the variant-based unification methods in [25].

In summary, the main point we wish to emphasize is that the equational theories T for which the current version of Maude-NPA will work properly are order-sorted theories of the form $T = (\Sigma, E \cup Ax)$ satisfying the admissibility requirements (1)–(5). Under assumptions (1)–(5), T-unification problems are always guaranteed to have a finite number of solutions and the Maude-NPA will find them by narrowing.

As a final *caveat*, if the user specifies a theory T where any of the above requirements (1)–(5) fail, besides the lack of completeness that would be caused by the failure of conditions (2)–(4), a likely consequence of failing to meet condition (5) will be that the tool may loop forever trying to solve a unification problem associated with just a single transition step in the symbolic reachability analysis process. However, we are investigating conditions more general than (5) (such as the above-mentioned finite variant property) that will still guarantee that a T-unification problem always has a finite complete set of solutions. Future versions of Maude-NPA will relax condition (5) to allow more general conditions of this kind.

7 State Space Reduction Techniques

In this section, we describe the different state-reduction techniques identifying unproductive backwards narrowing reachability steps. First, let us briefly recall a protocol state in Maude-NPA. A *state* in the protocol execution is a set of Maude-NPA strands unioned together with an associative and commutativity union operator `_&_` with identity operator `empty`, along with an additional term describing the intruder knowledge at that point. The *intruder knowledge* is represented as a set of facts unioned together with an associative and commutativity union operator `_,_` with identity operator `empty`. There are mainly two

[17] The fact that an equational theory T does not have a finitary narrowing-based algorithm does not by itself preclude the existence of a finitary unification algorithm obtained by other methods. In fact, the homomorphic theory we have just described does have a finitary unification algorithm [2]; however this dedicated unification algorithm is not an instance of a generic narrowing-based algorithm. However, as already explained, in the Maude-NPA the theories for which finitary unification is currently supported are either order-sorted theories with built-in axioms of commutativity and associativity-commutativity, or theories modulo such built-in axioms that are confluent, terminating, and coherent modulo Ax, and that are also strongly right irreducible.

kinds of intruder facts: positive knowledge facts (m `inI`), and negative knowledge facts (m `!inI`). Strands communicate between them via a unique shared channel represented by the intruder knowledge.

There are three reasons for detecting unproductive backwards narrowing reachability steps (i.e., the relation $St \rightsquigarrow_{\sigma, R_{\mathcal{P}}^{-1}, E_{\mathcal{P}}} St'$ described in Section 5.2). One is to reduce, if possible, the initially infinite search space to a finite one, as in the use of grammars. Another is to reduce the size of a (possibly finite) search space by eliminating unreachable states early, i.e., before they are eliminated by exhaustive search. This elimination of unreachable states can have an effect far beyond eliminating a single node in the search space, since a single unreachable state may appear multiple times and/or have multiple descendants. Finally, it is also possible to use various partial order reduction techniques that can further shrink the number of states that need to be explored.

7.1 Public data

The simplest optimization possible is when we are searching for some data that it is considered public using a subsort definition, e.g. "`subsort Name < Public`". That is, given a state St that contains an expression (t `inI`) in the intruder knowledge where t is of sort `Public`, we can remove the expression (t `inI`) from the intruder knowledge, since the backwards reachability steps taken care of such a (t `inI`) are trivially leading to an initial state but their inclusion in the message sequence is unnecessary.

7.2 Limiting Dynamic Introduction of New Strands

Our second optimization helps explain why attack states given in Section 5.2 contain a variable SS denoting a set of strands and a variable IK denoting a set of intruder facts, whereas the attack states explained in Section 4.4 do not contain those variables.

As pointed out in Section 5.2, Rules of type (5) allow the dynamic introduction of new strands. However, new strands can also be introduced by unification of a state containing a variable SS denoting a set of strands and one of the Rules (1), (2), and (4), where variables L and L' denoting lists of input/output messages will be introduced by instantiation of SS. The same can happen with new intruder facts of the form (X `inI`), where X is a variable. That is, consider a state containing a variable SS denoting a set of strands and a variable IK denoting a set of intruder knowledge, and the Rule (1):

$$SS' \,\&\, [L \mid M^-, L'] \,\&\, ((M \texttt{ inI}), IK') \rightarrow SS' \,\&\, [L, M^- \mid L'] \,\&\, ((M \texttt{ inI}), IK')$$

The following backwards narrowing step applying such a rule can be performed from the state above using the unifier $\sigma = \{SS \mapsto SS' \,\&\, [L, M^- \mid L'], IK \mapsto ((M \texttt{ inI}), IK')\}$

$$SS \;\&\; IK \overset{\sigma}{\rightsquigarrow}_{R,E} SS' \,\&\, [L \mid M^-, L'] \,\&\, ((M \texttt{ inI}), IK')$$

but this backwards narrowing step is unproductive, since it is not guided by the information in the attack state. Indeed, the same rule can be applied again using variables SS' and IK' and this can be repeated many times.

In order to avoid a huge number of unproductive narrowing steps, we allow the introduction of new strands and/or new intruder facts only by rule application instead of just by unification. For this, we do two things:

1. we remove any of the following variables from attack states; SS denoting a set of strands, IK denoting a set of intruder facts, and L, L' denoting a set of input/output messages; and
2. we replace Rule (1) by the following Rule (7), since we do no longer have a variable denoting a set of intruder facts that has to be instantiated:

$$SS \,\&\, [L \mid M^-, L'] \,\&\, \{(M \texttt{ inI}), IK\} \rightarrow SS \,\&\, [L, M^- \mid L'] \,\&\, \{IK\} \qquad (7)$$

Note that in order to replace Rule (1) by Rule (7) we have to assume that the intruder knowledge is a set of intruder facts without repeated elements, i.e., the union operator `_,_` is $ACUI$ (associative-commutative-identity-idempotent). This is completeness-preserving, since it is in line with the restriction in [20] that the intruder learns a term only once; if the intruder needs to use a term twice he must learn it the first time it is needed; if he learns a term and needs to learn it again in the backwards search, the state will be discarded as unreachable. Therefore, the set of rewrite rules used for backwards narrowing is $R_{\mathcal{P}} = \{(7), (2), (4)\} \cup (5)$.

Furthermore, one may imagine that Rule (4) and Rules of type (5) must also be modified in order to remove the (M `inI`) expression from the intruder knowledge of the right-hand side of each rule. However, this is wrong, since, by keeping the (M `inI`), we force the backwards application of such rule only when there is indeed a message for the intruder to be learned. This provides some sort of on-demand evaluation of the protocol.

7.3 Partial Order Reduction Giving Priority to Input Messages

The different rewrite rules on which the backwards narrowing search from an attack state is based are in general executed nondeterministically. This is because the order of execution can make a difference as to what subsequent rules can be executed. For example, an intruder cannot receive a term until it is sent by somebody, and that send action within a strand may depend upon other receives in the past. There is one exception, Rule (7) (originally Rule (1)), which, in a backwards search, only moves a negative term appearing right before the bar into the intruder knowledge. The execution of this transition in a backwards search does not disable any other transitions; indeed, it only enables send transitions. Thus, it is safe to execute it at each stage *before* any other transition. For the same reason, if several applications of Rule 7 are possible, it is safe to execute them all at once before any other transition. Requiring all executions of Rule 7 to execute first thus eliminates interleavings of Rule 7 with send and receive

transitions, which are equivalent to the case in which Rule 7 executes first. In practice, this typically cuts down in half the search space size.

Similar strategies have been employed by other tools in forward searches. For example, in [45], a strategy is introduced that always executes send transitions first whenever they are enabled. Since a send transition does not depend on any other component of the state in order to take place, it can safely be executed first. The original NPA also used this strategy; it had a receive transition which had the effect of adding new terms to the intruder knowledge, and which always was executed before any other transition once it was enabled.

7.4 Detecting Inconsistent States Early

There are several types of states that are always unreachable or inconsistent. If the Maude-NPA attempts to search beyond them, it will never find an initial state. For this reason, we augment the Maude-NPA search engine to always mark the following types of states as unreachable, and not search beyond them any further:

1. A state St containing two contradictory facts (t `inI`) and (t `!inI`) for a term t.
2. A state St whose intruder knowledge contains the fact (t `!inI`) and a strand of the form $[m_1^{\pm}, \ldots, t^{-}, \ldots, m_{j-1}^{\pm} \mid m_j^{\pm}, \ldots, m_k^{\pm}]$.
3. A state St containing a fact (t `inI`) such that t contains a fresh variable r and the strand in St indexed by r, i.e., $(r_1, \ldots, r, \ldots, r_k : \texttt{Fresh})$ $[m_1^{\pm}, \ldots, m_{j-1}^{\pm} \mid m_j^{\pm}, \ldots, m_k^{\pm}]$, cannot produce r, i.e., r is not a subterm of any output message in $m_1^{\pm}, \ldots, m_{j-1}^{\pm}$.
4. A state St containing a strand of the form $[m_1^{\pm}, \ldots, t^{-}, \ldots, m_{j-1}^{\pm} \mid m_j^{\pm}, \ldots, m_k^{\pm}]$ for some term t such that t contains a fresh variable r and the strand in St indexed by r cannot produce r.

Note that case 2 will become an instance of case 1 after some backwards narrowing steps, and the same happens with cases 4 and 3. The proof of inconsistency of cases 1 and 3 is obvious.

7.5 Transition Subsumption

Partial order reduction techniques (POR) are common in state exploration techniques due to their simplification properties. However, partial order techniques for narrowing-based state exploration have not been explored in detail, although they may be extremely relevant and even simplify much more than in standard state exploration techniques, based on ground terms rather than terms with variables. For instance, the simple concept of two states being equivalent modulo renaming of variables does not apply to standard state exploration techniques whereas it does apply to narrowing-based state exploration. In [24], Escobar and Meseguer explored narrowing-based state exploration and POR techniques, which may transform an infinite-state system into a finite one. However, the

Maude-NPA needs a dedicated POR technique that should combine different previous ideas in order to be applicable to the concrete execution model.

Let us motivate this with an example before giving more technical explanations. Consider again the NSPK attack state:

$$(r : \texttt{Fresh})[\ pk(b, a; N)^-, pk(a, N; n(b, r))^+, pk(b, n(b, r))^- \mid nil\ ]\ \&\ (n(b, r)\ \texttt{inI})$$

After a couple of backwards narrowing steps, the Maude-NPA finds the following state:

$$\begin{array}{l}[\ nil \mid n(b, r)^-, pk(b, n(b, r))^+\ ]\ \& \\ (r : \texttt{Fresh})[\ pk(b, a; N)^-, pk(a, N; n(b, r))^+ \mid pk(b, n(b, r))^-\ ]\ \& \\ ((pk(b, n(b, r))\ \texttt{!inI}), (n(b, r)\ \texttt{inI}))\end{array}$$

which corresponds to the intruder generating (i.e., learning) the message $pk(b, n(b, r))$ from the message $n(b, r)$, which he/she already knows; and the following state

$$\begin{array}{l}(r : \texttt{Fresh})[\ pk(b, a; N)^-, pk(a, N; n(b, r))^+ \mid pk(b, n(b, r))^-\ ]\ \& \\ (r' : \texttt{Fresh})[pk(b, A'; n(A', r'))^+ \mid pk(A', n(A', r'); n(b, r))^-, pk(b, n(b, r))^+\ ]\ \& \\ ((pk(b, n(b, r))\ \texttt{!inI}), (pk(A', n(A', r'); n(b, r))\ \texttt{inI}), (n(b, r)\ \texttt{inI}))\end{array}$$

which corresponds to the responder (identified by variable r) talking to an initiator (identified by variable r'). However, this second state is implied by the first state. Intuitively, the elements present in the first state that are relevant for the backwards reachability are both included in the second state, namely the (n(b,r) `inI`) item and the message $pk(b, a; N)^-$ that will be converted at some point into (pk(b,a;N) `inI`). Indeed, the unreachability of the following "kernel" state implies the unreachability of both states, although this kernel state is never computed by the Maude-NPA:

$$(r : \texttt{Fresh})[\ pk(b, a; N)^-, pk(a, N; n(b, r))^+ \mid pk(b, n(b, r))^-\ ]\ \&\ (n(b, r)\ \texttt{inI})$$

Note that the converse is not true, i.e., the second state does not imply the first one, since it contains one more intruder item relevant for backwards reachability purposes, namely (pk(A',n(A',r');n(b,r)) `inI`).

In the following, we write $IK^{\in}$ (resp. $IK^{\notin}$) to denote the subset of intruder facts of the form $(t\ \texttt{inI})$ (resp. $(t\ \texttt{!inI})$) appearing in the set of intruder facts IK. We abuse the set-theoretic notation and write $IK_1 \subseteq_{E_\mathcal{P}} IK_2$ for IK_1 and IK_2 sets of intruder facts to denote that all the intruder facts of IK_1 appear in IK_2 (modulo $E_\mathcal{P}$).

Definition 1. *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E_\mathcal{P}, R_\mathcal{P})$ representing protocol $\mathcal{P}$, and given two non-initial states $St_1 = SS_1\,\&\,\{IK_1\}$ and $St_2 =$*

$SS_2 \,\&\, \{IK_2\}$, *we write* $St_1 \triangleright St_2$ *(or* $St_2 \triangleleft St_1$*) if* $IK_1^{\in} \subseteq_{E_\mathcal{P}} IK_2^{\in}$, *and for each non-initial strand* $[\, m_1^{\pm}, \ldots, m_{j-1}^{\pm} \mid m_j^{\pm}, \ldots, m_k^{\pm} \,] \in SS_1$, *there exists* $[\, m_1^{\pm}, \ldots, m_{j-1}^{\pm} \mid m_j^{\pm}, \ldots, m_k^{\pm}, m_{k+1}^{\pm}, \ldots, m_{k'}^{\pm} \,] \in SS_2$. *Note that the comparison of the non-initial strand in* SS_1 *with the strands in* SS_2 *is performed modulo* $E_\mathcal{P}$.

Definition 2 ($\mathcal{P}$-subsumption relation). *Given a topmost rewrite theory* $\mathcal{R} = (\Sigma, E_\mathcal{P}, R_\mathcal{P})$ *representing protocol* $\mathcal{P}$ *and two non-initial states* St_1, St_2. *We write* $St_1 \preceq_\mathcal{P} St_2$ *and say that* St_1 *is* $\mathcal{P}$-subsumed *by* St_2 *if there is a substitution* θ *s.t.* $St_1 \triangleleft \theta(St_2)$.

This technique is used as follows: we keep all the states of the backwards narrowing-based tree and compare each new leaf of the tree with all the previous states in the tree. If a leaf is $\mathcal{P}$-subsumed by a previously generated node in the tree, we discard such leaf.

7.6 The Super Lazy Intruder

Sometimes terms appear in the intruder knowledge that are trivially learnable by the intruder. These include terms initially available to the intruder (such as names) and variables. In the case of variables, the intruder can substitute any arbitrary term of the same sort as the variable,[18] and so there is no need to try to determine all the ways in which the intruder can do this. For this reason it is safe, at least temporarily, to drop these terms from the state. We will refer to those terms as *lazy intruder* terms. The problem of course, is that later on in the search the variable may become instantiated, in which case the term now becomes relevant to the search. In order to avoid this problem, we take an approach similar to that of the lazy intruder of Basin et al. [6] and extend it to a more general case, that we call the *super-lazy terms*. We note that this use of what we here call the super-lazy intruder was also present in the original NPA.

Super-lazy terms are defined inductively as the union of the set of lazy terms, i.e., variables, with the set of terms that are produced out of other super-lazy terms using operations available to the intruder. That is, $e(K, X)$ is a super-lazy term if the intruder can perform the e operation, and K and X are variables. More precisely, the set of super-lazy intruder terms is defined as follows.

Definition 3. *Given a topmost rewrite theory* $\mathcal{R} = (\Sigma, E_\mathcal{P}, R_\mathcal{P})$ *representing protocol* $\mathcal{P}$, *and a state* St *where* $IK^{\notin}(St) = \{x \mid (x \texttt{ !inI}) \in St\}$, *its set of* super-lazy terms w.r.t. St *(or simply super-lazy terms) is defined as the union of the following:*

- *the set of variables of sort* `Msg` *or one of its subsorts,*
- *the set of terms* t *appearing in strands of the form* $[t^+]$, *and*

[18] This, of course, is subject to the assumption that the intruder can produce at least one term of that sort. But since the intruder is assumed to have access to the network and to all the operations available to an honest principal, this is a safe assumption to make.

– *the set of terms of the form* $f(t_1, \ldots, t_n)$ *where* $\{t_1, \ldots, t_n\}$ *are super-lazy intruder terms w.r.t.* St, $\{t_1, \ldots, t_n\} \not\subseteq IK^{\not\in}(St)$, *and there is an intruder strand* $[(X_1)^-, \ldots, (X_n)^-, (f(X_1, \ldots, X_n))^+]$ *with* $X_1, \ldots, X_n$ *variables.*

The idea behind the super-lazy intruder is that, given a term made out of lazy intruder terms, such as "$a; e(K, Y)$", where a is a public name and K and Y are variables, the term "$a; e(K, Y)$" is also a (super) lazy intruder term by applying the operations e and $\_;\_$.

Let us first briefly explain how the (super) lazy intruder mechanism works before formally describing it. When we detect a state St with a super lazy term t, we replace the intruder fact $(t\ \texttt{inI})$ in St by a new expression $ghost(t)$ and keep the modified version of St in the history of states used by the transition subsumption of Section 7.5. If later in the search tree we detect a state St' containing an expression $ghost(t)$ such that t is no longer a super lazy intruder term (or *ghost expression*), then t has been instantiated in an appropriate way and we must reactivate the original state St that introduced the $ghost(t)$ expression (and that precedes St' in the narrowing tree) with the new binding for variables in t applied. That is, we "roll back" and replace the current state St' with an instantiated version of state St.

However, if the substitution θ binding variables in t includes variables of sort `Fresh`, since they are unique in our model, we have to keep them in the reactivated version of St. Therefore, the strands indexed by these fresh variables must be included in the "rolled back" state, even if they were not there originally. Moreover, they must have the bar at the place it was when the strands were originally introduced. We show below how this is accomplished.

Furthermore, if any of the strands thus introduced have other variables of sort `Fresh` as subterms, then the strands indexed by those variables must be included too, and so on. Thus, when a state St' properly instantiating a ghost expression $ghost(t)$ is found, the procedure of rolling back to the original state St that gave rise to that ghost expression implies not only applying the bindings for the variables of t to St, but also introducing in St all the strands from St' that produced fresh variables and that either appear in the variables of t or are recursively connected to them.

First, before formally defining the super-lazy intruder technique, we must modify Rules of type 5 introducing new strands:

$$\{\, [\, l_1 \,|\, u^+ \,] \,\&\, \{(u\ \texttt{!inI}), K\} \rightarrow \{(u\ \texttt{inI}), K\} \text{ s.t. } [\, l_1,\, u^+,\, l_2 \,] \in \mathcal{P}\} \qquad (8)$$

Therefore, the set of rewrite rules used by narrowing in reverse are now $R_{\mathcal{P}} = \{(7), (2), (4)\} \cup (8)$. Note that Rules of type (5) introduce strands $[\, l_1 \mid u^+, l_2 \,]$, whereas here Rules of type (8) introduce strands $[\, l_1 \mid u^+ \,]$. This slight modification allows to safely move the position of the bar back to the place where the strand was introduced.

We extend the intruder knowledge to allow an extra fact $ghost(t)$. Indeed, this corresponds to the fourth component of a Maude-NPA state explained in Section 4. We first describe how to reactivate a state. Given a strand $s = (r_1, \ldots, r_k :$

Fresh) $[m_1^{\pm}, \ldots \mid \ldots, m_n^{\pm}]$, when we want to move the bar to the rightmost position (denoting a final strand), we write $s{\gg} = (r_1, \ldots, r_k : \texttt{Fresh})\ [m_1^{\pm}, \ldots, m_n^{\pm} \mid nil]$.

Definition 4. *Given a state St containing an intruder fact ghost(t) for some term t with variables, we define the set of strands associated to t, denoted $SS_{St}(t)$, as follows: for each strand s in St of the form $(r_1, \ldots, r_k : \texttt{Fresh})\ [m_1^{\pm}, \ldots \mid \ldots, m_n^{\pm}]$, if there is $i \in \{1, \ldots, k\}$ s.t. $r_i \in Var(t)$, then $s{\gg} \in SS_{St}(t)$; or if there is another strand $s' \in SS_{St}(t)$ of the form $(r'_1, \ldots, r'_{k'} : \texttt{Fresh})\ [w_1^{\pm}, \ldots \mid \ldots, w_{n'}^{\pm}]$, $i \in \{1, \ldots, k\}$, and $j \in \{1, \ldots, n'\}$ s.t. $r_i \in Var(w_j)$, then $s{\gg} \in SS_{St}(t)$.*

Improving the super lazy intruder. When we detect a state St with a super lazy term t, we may want to analyze whether the variables of t may be eventually instantiated or not before creating a ghost state. Therefore, if for each strand $[m_1^{\pm}, \ldots, m_{j-1}^{\pm} \mid m_j^{\pm}, \ldots, m_k^{\pm}]$ in St and each $i \in \{1, \ldots, j-1\}$, $Var(t) \cap Var(m_i) = \emptyset$, and for each term $(w\ \texttt{inI})$ in the intruder knowledge, $Var(t) \cap Var(w) = \emptyset$, then we can clearly infer that the variables of t can never be instantiated and adding a ghost to state St is unnecessary.

Interaction with transition subsumption. When a ghost state is reactivated, we see from the above definition that such a reactivated state will be $\mathcal{P}$-subsumed by the original state that raised the ghost expression. Therefore, the transition subsumption of Section 7.5 has to be slightly modified to avoid checking a resuscitated state with its predecessor ghost state, i.e., $St_1 \succeq'_{\mathcal{P}} St_2$ iff $St_1 \succeq_{\mathcal{P}} St_2$ and St_2 is *not* a resuscitated version of St_1.

7.7 Grammars

The Maude-NPA's ability to reason effectively about low-level algebraic properties is a result of its combination of symbolic reachability analysis using narrowing modulo equational properties (see Section 6), together with its grammar-based techniques for reducing the size of the search space, as well as other state space reduction techniques explained in this section. Here we briefly explain how grammars work as a state space reduction technique and refer the reader to [35,20] for further details.

Automatically generated grammars $\langle G_1, \ldots, G_m \rangle$ represent unreachability information (or co-invariants), i.e., typically infinite sets of states unreachable for the intruder. That is, given a message m and an automatically generated grammar G, if $m \in G$, then there is no initial state St_{init} and substitution θ such that the intruder knowledge of St_{init} contains the fact $\theta(m)\ !\texttt{inI}$, i.e., the intruder will never be able to learn message m in the future. These automatically generated grammars are very important in our framework, since in the best case they can reduce the infinite search space to a finite one, or, at least, can drastically reduce the search space.

Let us motivate this with the NSPK attack state:

$$(r : \texttt{Fresh})[\ pk(b, a; N)^-, pk(a, N; n(b, r))^+, pk(b, n(b, r))^-\ |\ nil\ ]\ \&\ (n(b, r)\ \texttt{inI})$$

After a couple of backwards narrowing steps, the Maude-NPA finds the following state:

$$\begin{aligned}
&[\ nil\ |\ (n(b, r); M)^-, n(b, r)^+\ ]\ \& \\
&(r : \texttt{Fresh})[\ pk(b, a; N)^-, pk(a, N; n(b, r))^+\ |\ pk(b, n(b, r))^-\ ]\ \& \\
&((n(b, r)\ \texttt{!inI}), (pk(b, n(b, r))\ \texttt{inI}), ((n(b, r); M)\ \texttt{inI}))
\end{aligned}$$

which corresponds to the intruder generating (i.e., learning) the message $n(b, r)$ from a bigger message $(n(b, r); M)$, although the contents of variable M have not yet been found by the backwards reachability analysis. This process of adding more and more intruder strands that look for terms $((n(b, r); M); M')$, $(((n(b, r); M); M'); M'')$, ... can go infinitely. Note that if we carefully check the strands for the NSPK protocol given in Page 17, we can see that the honest strands never produce a message of the form "*Nonce* ; *Message*" or such a message is under a public key encryption (and thus he/she cannot get into the contents), so the previous state is clearly unreachable and can be discarded. The grammar, which is generated by Maude-NPA, capturing the previous state as unreachable is as follows:

```
grl M inL => pk(A, M) inL . ;
grl M inL => sk(A, M) inL . ;
grl M inL => (M' ; M) inL . ;
grl M inL => (M ; M') inL . ;
grl M notInI, M notLeq n(i, r) => (i ; M) inL . ;
grl M notInI, M notLeq n(i, r) => (M ; M') inL . ;
grl M notInI, M notLeq n(i, r) => pk(A, B ; M) inL .
```

Intuitively, the last production rule in the grammar above says that any term of the form $pk(A, B; M)$ cannot be learned by the intruder if it is different to $pk(A, B; n(i, r))$ (i.e., it does not match such pattern) and the constraint (M !inI) appears explicitly in the intruder knowledge of the current state being checked for unreachability. Moreover, any term of any of the following forms: $pk(A, M)$, $sk(A, M)$, $(M'; M)$, or $(M; M')$ cannot be learned by the intruder if subterm M is also not learnable by the intruder.

Unlike the grammars used in NPA, described in [35], and the version of Maude-NPA described in [20], in which initial grammars needed to be specified by the user, Maude-NPA now generates initial grammars *automatically*. Each initial grammar consists of a single seed term of the form `grl` C `=>` $f(X_1, \cdots, X_n)$ `inL`, where f is an operator symbol from the protocol specification, the X_i are variables, and C is either empty or consists of the single constraint (X_i `notInI`), similar to expression (X_i `!inI`) but used in a different context. For instance, the seed term for the grammar above is "`grl M notInI =>`

`(M ; M') inL`". However, Maude-NPA provides features to control such automatically generated grammars, e.g., adding more seed terms; see [23] for further details.

8 Conclusions

In this tutorial, we have given an overview of the Maude-NRL Protocol Analyzer (Maude-NPA). We have explained the basic mechanics of using Maude-NPA, including writing specifications and formulating queries, that gives the minimum amount of background for using the tool. We have also explained how the Maude-NPA actually works at the backwards reachability level and at the equational unification level. Finally, we have described the different state-reduction techniques identifying unproductive backwards narrowing reachability steps. This, we believe, is enough to give a basic understanding of how Maude-NPA works, and also to get started using it.

Maude-NPA is still under development, and we are currently working on a number of ways of extending its applicability and usability, including more sophisticated narrowing-based algorithms, as mentioned in this tutorial, and a graphical user interface [44] that allows a user to examine a Maude-NPA search tree in detail. However, it still remains grounded in the principles described in this tutorial, and we believe the ideas and techniques we have presented here will remain a useful introduction even as it matures.

Acknowledgments

José Meseguer has been partially supported by NSF grants CNS 07-16638 and CNS 08-31064. Santiago Escobar has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grant TIN 2007-68093-C02-02, and Generalitat Valenciana GVPRE/2008/113.

References

1. M. Abadi and V. Cortier. Deciding knowledge in security protocols under equational theories. *Theoretical Computer Science*, 367(1-2):2–32, 2006.
2. S. Anantharaman, P. Narendran, and M. Rusinowitch. Unification modulo CUI plus distributivity axioms. *Journal of Automated Reasoning*, 33(1):1–28, 2004.
3. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P. Heam, O. Kouchnarenko, J. Mantovani, S. Moedersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Vigano, and L. Vigneron. The Avispa tool for the automated validation of internet security protocols and applications. In *Proceedings of CAV 05*. Springer-Verlag, 2005.
4. A. Armando, L. Compagna, and Y. Lierler. SATMC: a SAT-based model checker for security protocols. In *Proceedings of the 9th European Conference on Logic in Artificial Intelligence(JELIA'04)*, Lecture Notes in Computer Science, Lisbon, Portugal, September 2004. Springer.

5. F. Baader and W. Snyder. Unification theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. Elsevier and MIT Press, 2001.
6. D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.
7. M. Baudet, V. Cortier, and S. Delaune. YAPA: A generic tool for computing intruder knowledge. In R. Treinen, editor, *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, Lecture Notes in Computer Science, Brasília, Brazil, June-July 2009. Springer. To appear.
8. B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
9. Y. Boichut, P.-C. Héam, O. Kouchnarenko, and F. Oehl. Improvements on the Genet and Klay technique to automatically verify security protocols. In *Proceedings of Automated Verification of Infinite States Systems (AVIS04)*. ENTCS, 2004.
10. S. Bursuc and H. Comon-Lundh. Protocol security and algebraic properties: decision results for a bounded number of sessions. In R. Treinen, editor, *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, Lecture Notes in Computer Science, Brasília, Brazil, June-July 2009. Springer. To appear.
11. Y. Chevalier and M. Rusinowitch. Hierarchical combination of intruder theories. *Inf. Comput.*, 206(2-4):352–377, 2008.
12. Ş. Ciobâcă, S. Delaune, and S. Kremer. Computing knowledge in security protocols under convergent equational theories. In R. Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction (CADE'09)*, Lecture Notes in Artificial Intelligence, Montreal, Canada, Aug. 2009. Springer. To appear.
13. M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Unification and Narrowing in Maude 2.4. In R. Treinen, editor, *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, Lecture Notes in Computer Science. Springer, June-July 2009. To appear.
14. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude – A High-Performance Logical Framework*. Springer LNCS Vol. 4350, 2007.
15. H. Comon-Lundh and S. Delaune. The finite variant property: How to get rid of some algebraic properties. In J. Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.
16. E. Contejean and C. Marché. The CiME system: tutorial and user's manual. Manuscript, Université Paris-Sud, Centre d'Orsay.
17. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transaction on Information Theory*, 29(2):198–208, 1983.
18. N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Multiset rewriting and the complexity of bounded security. *Journal of Computer Security*, pages 677–722, 2004.
19. S. Escobar, J. Hendrix, C. Meadows, and J. Meseguer. Diffie-Hellman cryptographic reasoning in the Maude-NRL protocol analyzer. In *Proc. 2nd International Workshop on Security and Rewriting Techniques (SecReT 2007)*, 2007.

20. S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Compute Science*, 367(1–2):162–202, 2006.
21. S. Escobar, C. Meadows, and J. Meseguer. Equational cryptographic reasoning in the Maude-NRL protocol analyzer. In *Proc. 1st International Workshop on Security and Rewriting Techniques (SecReT 2006)*, pages 23–36. ENTCS 171(4) , Elsevier, 2007.
22. S. Escobar, C. Meadows, and J. Meseguer. State space reduction in the Maude-NRL Protocol Analyzer. In S. Jajodia and J. López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2008.
23. S. Escobar, C. Meadows, and J. Meseguer. *Maude-NPA, version 1.0*. University of Illinois at Urbana-Champaign, March 2009. Available at `http://maude.cs.uiuc.edu/tools/Maude-NPA`.
24. S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA'07)*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168, 2007.
25. S. Escobar, J. Meseguer, and R. Sasse. Variant narrowing and equational unification. Technical Report UIUCDCS-R-2007-2910, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 2007.
26. S. Escobar, J. Meseguer, and R. Sasse. Effectively checking the finite variant property. In A. Voronkov, editor, *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2008.
27. S. Escobar, J. Meseguer, and R. Sasse. Variant narrowing and equational unification. In G. Rossu, editor, *Proc. 7th. Intl. Workshop on Rewriting Logic and its Applications*, ENTCS to appear. Elsevier, 2008.
28. F. J. T. Fabrega, J. Herzog, and J. Guttman. Strand Spaces: What Makes a Security Protocol Correct? *Journal of Computer Security*, 7:191–230, 1999.
29. J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings, Fifth Conference on Automated Deduction*, pages 318–334. Springer-Verlag, 1980. LNCS, Volume 87.
30. S. F. J.K. Millen, S.C. Clark. The interrogator: Protocol secuity analysis. *IEEE Transactions on Software Engineering*, pages 274–288, Feb. 1987.
31. J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In *Proc. ICALP'83*, pages 361–373. Springer LNCS 154, 1983.
32. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, 17:93–102, 1996.
33. C. Lynch and C. Meadows. On the relative soundness of the free algebra model for public key encryption. In *Workshop on Issues in Theory of Security 2004*, 2004.
34. C. Lynch and C. Meadows. Sound approximations to Diffie-Hellman using rewrite rules. In *Proceedings of the International Conference on Information and Computer Security (ICICS)*. Springer-Verlag, 2004.
35. C. Meadows. Language generation and verification in the NRL protocol analyzer. In *Ninth IEEE Computer Security Foundations Workshop, March 10 - 12, 1996, Dromquinna Manor, Kenmare, County Kerry, Ireland*, pages 48–61. IEEE Computer Society, 1996.

36. C. Meadows. The NRL protocol analyzer: An overview. *Journal of logic programming*, 26(2):113–131, 1996.
37. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
38. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.
39. J. Millen. On the freedom of decryption. *Information Processing Letters*, 86(3), 2003.
40. J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murphi. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1997.
41. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128, 1998.
42. M. Rusinowitch and M. Turuani. Protocol insecurity with a finite number of sessions and composed keys is NP-complete. In 14^{th} *IEEE Computer Security Foundations Workshop*, pages 174–190, 2001.
43. P. Y. A. Ryan and S. A. Schneider. An attack on a recursive authentication protocol. a cautionary tale. *Inf. Process. Lett.*, 65(1):7–10, 1998.
44. S. Santiago, C. Talcott, S. Escobar, C. Meadows, and J. Meseguer. A graphical user interface for Maude-NPA. Technical Report DSIC-II/02/09, Universidad Politécnica de Valencia, June 2009.
45. V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *11th Computer Security Foundations Workshop — CSFW-11*. IEEE Computer Society Press, 1998.
46. S. Stubblebine and C. Meadows. Formal characterization and automated analysis of known-pair and chosen-text attacks. *IEEE Journal on Selected Areas in Communications*, 18(4):571–581, 2000.
47. TeReSe, editor. *Term Rewriting Systems*. Cambridge University Press, Cambridge, 2003.
48. P. Thati and J. Meseguer. Symbolic reachability analysis using narrowing and its application verification of cryptographic protocols. *J. Higher-Order and Symbolic Computation*, 20(1–2):123–160, 2007.