Maude-NPA, Version 3.0.1

Santiago Escobar
sescobar@dsic.upv.es
DSIC-ELP, Universitat Politècnica de València
Valencia, Spain

Catherine Meadows
meadows@itd.nrl.navy.mil
Naval Research Laboratory
Washington, DC, USA

José Meseguer
meseguer@illinois.edu
University of Illinois at Urbana-Champaign
Urbana, IL, USA

July 30th, 2017

Contents

1 Introduction

This document describes version 3.0 of the Maude-NRL Protocol Analyzer (Maude-NPA) and gives instructions for its use. Maude-NPA is an analysis tool for cryptographic protocols that takes into account many of the algebraic properties of cryptosystems that are not supported in other tools. Maude-NPA uses an approach similar to its predecessor, the NRL Protocol Analyzer (NPA) [34], i.e., it is based on unification and performs backwards search from a final state to determine whether or not it is reachable. However, unlike the original NPA, it has a theoretical basis in rewriting logic and narrowing, and while NPA only could be used to reason equational theories involving a fixed set of rewrite rules, Maude-NPA can be used to reason about a wide range of equational theories. First of all, it provides built-in support for theories involving symbols with any combination of associativity (A), commutativity (C), and identity (U) axioms. Furthermore, by relying on variant-based equational unification [23], Maude-NPA allows users to augment the basic set of equational axioms supported with rewrite rules. Theories that can be supported this way include cancellation of encryption and decryption, Diffie-Hellman exponentiation, exclusive-or, and some approximations of homomorphic encryption. A description of Maude-NPA's formal foundations in rewriting logic, together with a soundness and completeness proof, are given in [16, 21]. The most detailed description of how Maude-NPA works is given in [19].

The current version 3.0 of Maude-NPA adds several useful new features to version 2.0, including:

1. *Equational Variant-based Unification in Full Generality.* Unification modulo a theory has been extended to its full generality for theories satisfying the *finite variant property* when the equational theory is convergent modulo any combination of associativity and/or commutativity and/or identity axioms (see Section 4).

2. *Protocol Composition.* Protocols are often obtained by combining several subprotocols. In Maude-NPA 3.0 such subprotocols can be specified modularly, and the security properties of their compositions can be analyzed by the tool (see Section 8).

3. *Process Algebra.* In addition to the strand space notation traditionally used by Maude-NPA (see Section 5), a more convenient notation based on process algebra allows easy specification of protocols with branching behavior and various kinds of choice and non-determinism (see Section 9). We offer the user the option of using the original Maude-NPA language or the process algebra language.

This document is organized into basic, advanced, and experimental sections. Sections that are labeled with "$*$", as well as all the appendices, are not necessary for getting started with Maude-NPA, and suggested for more advanced users. Sections that are labeled with "$**$" present work that is still in progress, but has

reached a point at which users may be interested in experimenting with it. In Section 2 we explain how Maude-NPA works using informal illustrations of different features of Maude-NPA. In Section 3 we describe the mechanics of loading Maude-NPA and writing and running Maude-NPA specifications, showing all the fragments of a protocol specification at a very high level. Since the inclusion of cryptographic properties is a remarkable feature in Maude-NPA, we describe them in Section 4 using cancellation of encryption and decryption, exclusive-or, modular exponentiation, and homomorphic encryption over concatenation as examples. In Section 5 we describe how a protocol is specified in Maude-NPA in full detail, using the Needham-Schroeder public key (NSPK) protocol as a running example. In Section 6, we describe how an attack pattern (the input to the search process) is specified in Maude-NPA in full detail, again using the NSPK protocol as an example. In Section 7, we describe the search commands that can be used, again using NSPK as an example. In Section 8, we describe an advanced feature of Maude-NPA for protocol composition, describing its syntax and semantics with some examples. In Section 9, we describe another advanced feature of Maude-NPA, currently under development, for specifying protocols using a process algebra notation, describing its syntax and semantics with some examples. In Section 10, we describe how the tool can be applied to several other examples. And, finally, in Section 11 we describe some known limitations of the tool and plans for further extensions. Several appendices are included, where we give some extra commands, describe how to specify grammars, and how to disable state-space reduction techniques for debugging.

Throughout this document we assume a minimum acquaintance with the basic syntactic conventions of the Maude language, an implementation of rewriting logic. We refer the user to the Maude manual that is available online at `http://maude.cs.uiuc.edu`, and also to the Maude book [7] for more detailed documentation on Maude-related matters.

Finally, we note that Maude-NPA is still a work in progress and we are constantly experimenting with new ways of optimizing it and improving its performance. The version in this release, Maude-NPA 3.0, has the best overall performance of the versions we have experimented with, but performs less well for certain protocols than some previously unreleased versions, particularly for some (but not all) protocols involving exclusive-or. We are working on improving the overall performance, and hope to provide versions with better performance in this area in subsequent 3.x releases. In this manual we will note cases where performance is negatively impacted in this way. This should help avoid confusion in cases in which Maude-NPA runs less well on a given protocol than unreleased versions that were previously cited in the literature.

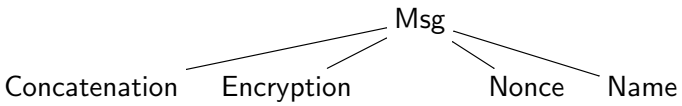
2 How Maude-NPA works

In this section, we explain how Maude-NPA works at a very high level without requiring the reader to know too many technical details.

2.1 Type information and function symbols

Given a protocol $\mathcal{P}$, Maude-NPA considers an order-sorted signature $\Sigma_{\mathcal{P}}$ defining the sorts and function symbols used in the protocol.

For example, consider a signature with sorts Msg, Encryption, Concatenation, Nonce, Fresh, and Name. The order-sorted information is provided as a subsort inclusion order between sorts:

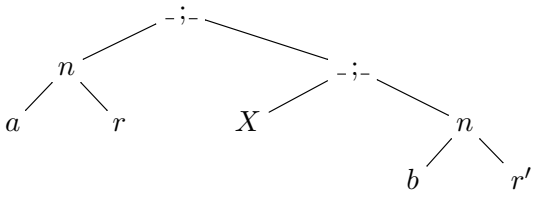


describing that, for example, any term of sort Concatenation is also of sort Msg. This is useful to specify *subtype polymorphism*, allowing one function definition that spans over different argument sorts and also allowing function definitions specific for different subsorts. Note that Fresh is not a subtype of Msg.

The signature also includes the following function symbols: `pk` (for "public" key encryption), `sk` (for "secret" or "private" key encryption), `n` (for nonces), and `_;_` (for concatenation). They satisfy the following typing declarations[1]:

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

For example, the term $t = n(a, r)\ ;\ (X\ ;\ n(b, r'))$, where a, b, and c are constants of sort Name, X is a variable of sort Msg, and r, r', and r'' are variables of sort Fresh, is a term of sort Concatenation and has the following tree representation



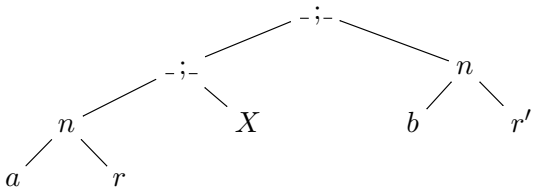
2.2 Equational theory

Maude-NPA uses not only an order-sorted signature $\Sigma_{\mathcal{P}}$ associated to a protocol $\mathcal{P}$ but an equational theory $E_{\mathcal{P}}$ specifying the *algebraic properties* of the function symbols in $\Sigma_{\mathcal{P}}$. They are useful for describing both the properties of the cryptographic functions in the protocol and the properties of any other regular symbol.

Maude-NPA works with $E_{\mathcal{P}}$-equivalence classes. For a term t, $[t]$ denotes its equivalence class (i.e., $t' \in [t] \Leftrightarrow t' =_{E_{\mathcal{P}}} t$, where $t' =_{E_{\mathcal{P}}} t$, means that t' may be transformed into t by a sequence of rules from $E_{\mathcal{P}}$. For example, if we assume

[1]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 its two arguments written in an infix notation.

that the symbol _;_ is associative, i.e., $x;(y;z) = (x;y);z$, the equivalence class $[t]$ contains not only the tree representation above but also the following one:



And if we assume that the symbol _;_ is associative and commutative, i.e., $x;y = y;x$, then the number of elements in the equivalence class $[t]$ contains all the permutations of both tree representations.

In addition, we can use equivalence classes to specify properties of the cryptographic symbols in the protocol. For example, the symbols `pk` and `sk` satisfy the decryption property, i.e., $sk(k, pk(k, m)) = m$. For the term $u = sk(a, pk(a, n(b, r)))$, the number of elements in the equivalence class $[u]$ is infinite: $sk(a, pk(a, n(b, r)))$ and $n(b, r)$ but also any term of the form $sk(k_1, pk(k_1, \ldots, sk(k_n, pk(k_n, sk(a, pk(a, n(b, r)))))) \cdots))$. In this case, Maude-NPA keeps only $n(b, r)$, which is called the *normalized* (or *simplified*) version of u. Maude-NPA makes a distinction between whether an algebraic property is used for simplification or not, which is explained in detail in Section 4.

2.3 Equational unification

The execution states associated to a protocol are modeled as elements of an initial algebra $T_{\Sigma_\mathcal{P}/E_\mathcal{P}}$ (i.e., the set of all the terms without variables modulo the algebraic properties $E_\mathcal{P}$). However, Maude-NPA does not work with concrete states in the initial algebra but with equivalence classes of *symbolic state patterns* $[t(x_1, \ldots, x_n)]$ on the free algebra $T_{\Sigma_\mathcal{P}/E_\mathcal{P}}(X)$ over a set of sorted variables X (i.e., the set of all the terms with variables modulo the algebraic properties $E_\mathcal{P}$).

Maude-NPA relies on equational unification to work with symbolic state patterns. Given an algebra $T_{\Sigma_\mathcal{P}/E_\mathcal{P}}(X)$, a substitution σ on $T_{\Sigma_\mathcal{P}/E_\mathcal{P}}(X)$ is a map from variables X to $T_{\Sigma_\mathcal{P}/E_\mathcal{P}}(X)$-terms such that σ is the identify on all but a finite set of variables. Given two terms u and v and an equational theory $E_\mathcal{P}$ associated to a protocol $\mathcal{P}$, a substitution σ is a $E_\mathcal{P}$-unifier of terms u and v (or a unifier modulo $E_\mathcal{P}$) if $\sigma(u) =_{E_\mathcal{P}} \sigma(v)$. A substitution σ is more general than another substitution θ if θ is a substitution instance of σ, i.e., there is a substitution γ such that for each variable x, $\sigma(x) =_{E_\mathcal{P}} \gamma(\theta(x))$. *A complete set of most general $E_\mathcal{P}$-unifiers* of two terms u and v satisfies the property that for any unifier of u and v, there is a more general substitution in the set.

For example, consider the decryption property above, i.e., $sk(k, pk(k, m)) = m$. The complete set of most general unifiers of the two terms $t = sk(a, X)$ and $s = n(b, r)$ (where X is a variable of sort Msg and r is a variable of sort Fresh) is $\sigma = \{X \mapsto pk(a, n(b, r))\}$.

Consider an associative and commutative (AC) symbol $\_ * \_$ defined on terms of sort Msg. 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 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 $\_ \oplus \_$ defined on terms of sort Msg and the constant 0 satisfying the following xor-properties (where X, Y, Z are variables of sort Msg):

$$\begin{array}{cc}
X \oplus Y = Y \oplus X & X \oplus X = 0 \\
X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z & X \oplus 0 = X
\end{array}$$

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

$$\theta_1 = \{X \mapsto Y' \oplus U' \oplus V', Y \mapsto Y', U \mapsto U', V \mapsto V'\}$$

Section 4 describes the equational unification capabilities included in Maude-NPA.

2.4 Maude-NPA search states

Maude-NPA performs reachability analysis by: (i) working with symbolic state patterns representing typically *infinite sets* of its ground instances; and (ii) performing equational-unification-based *symbolic execution*. What (i) means is that a pattern t describing a "symbolic state" defines a corresponding *set* $[\![t]\!]$ of actual *instances*. And what (ii) means is that unification works with equivalence classes of the symbolic states and all the possible unifiers must be explored.

For example, the Needham-Schroeder public key (NSPK) protocol is specified using the standard Alice-and-Bob notation as follows:

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

where A and B denote Alice and Bob principal identifiers, N_A and N_B denote the respective nonces, and $pub(A)$ and $pub(B)$ are the respective public keys.

Let us consider an informal representation of the symbolic states found by Maude-NPA where we include each participant as "*Name:*". The intruder is represented by "*Intruder:*" and contains a *set* of learned messages. Any other participant

is labeled by its role label, e.g. "*Alice:*", and contains a *list* of received and sent messages; a positive node $+(m)$ implies sending, and a negative node $-(m)$ implies receiving.

The following is an informal representation of one of the possible symbolic states found during the execution of the protocol, which represents a partial execution of an instance of Alice's role and an instance of Bob's role:

$$\begin{array}{rl} & \mathit{Roles}\ \& \\ \text{Alice:} & +(pk(i, A; N_A)), -(pk(A, N_A; N_B)) \\ \text{Bob:} & -(pk(B, A; N_A)), +(pk(A, N_A; N_B)) \\ \text{Intruder:} & \mathit{Knowledge} \end{array}$$

where *Roles* is a variable denoting unknown role executions, *Knowledge* is a variable denoting unknown intruder knowledge, A and B are variables of sort Name, i is a constant identifying the intruder, and N_A and N_B are variables of sort Nonce.

2.5 Reachability analysis

Maude-NPA performs *backwards* symbolic reachability analysis, i.e., if the protocol's usual "forwards" transitions are specified by rules of the form $l \to r$, then the reverse, "backwards" transitions are specified by reversed rules of the form $r \to l$.

The well-known man-in-the-middle attack is described by the following attack pattern, where an instance of Bob's role has participated, this instance has finished its execution, and the intruder has learned the nonce, N_B, generated by this Bob's instance; logical variables $A_?$ and $N_{A_?}$ are labeled with a question mark, and variables for extra role instances, Roles_1, and the extra intruder knowledge, $\mathit{Knowledge}_1$, are written explicitly. All these variables, $A_?$, $N_{A_?}$, Roles_1, and $\mathit{Knowledge}_1$ are existentially quantified, even if the symbol $\exists$ is not written explicitly in the informal attack pattern notation.

$$\begin{array}{rl} & \mathit{Roles}_1\ \& \\ \text{Bob:} & -(pk(B, A_?; N_{A_?})), +(pk(A_?, N_{A_?}; N_B)), -(pk(B, N_B)) \\ \text{Intruder:} & N_B\ \&\ \mathit{Knowledge}_1 \end{array}$$

Then, the *existence of an attack* on the given protocol from a symbolic attack state u exactly means that there is a substitution θ such that $\theta(u)$ can reach in a backwards direction a state $\theta(v)$ that it is initial (i.e., no more backwards steps can be done). But this is equivalent to the forwards meaning that from the initial state $\theta(v)$ we can reach $\theta(u)$ in some protocol execution. The extra ingredients necessary for reachability analysis are just to: (i) incorporate the Dolev-Yao intruder capabilities as transition rules, and (ii) adding new protocol sessions whenever necessary.

Figure 1 shows an informal representation of the full backwards *symbolic* execution from this attack pattern until an initial state is reached. The transition arrows include the honest or dishonest action being performed and the variable instantiation.

Note that although in Figure 1 the attack proceeds from the initial state (at the bottom) to the final state (at the top), the arrows go in the direction of the backwards search. Moreover, as the search proceeds, the variables in the state expressions become further and further instantiated. In order to see the attack state that is actually reached, we can compose all the partial substitutions generated through the backwards steps and apply it to the attack pattern, producing the following fully instantiated attack state:

$$\begin{array}{rl} & Roles_2 \ \& \\ \text{Alice:} & +(pk(i, A; N_A)), -(pk(A, N_A; N_B)), +(pk(i, N_B)) \\ \text{Bob:} & -(pk(B, A; N_A)), +(pk(A, N_A; N_B)), -(pk(B, N_B)) \\ \text{Intruder:} & N_B \ \& \ pk(B, N_B) \ \& \ pk(i, N_B) \ \& \ pk(A, N_A; N_B) \\ & \& \ pk(B, A; N_A) \ \& \ pk(i, A; N_A) \ \& \ Knowledge_6 \end{array}$$

It is easy now to figure out the fully instantiated execution path in a forwards sense from the initial state to this fully instantiated attack pattern (just by following Figure 1 from bottom to top).

Let us also illustrate equational unification within this protocol example. In the step called "Intruder extracts N_B from $pk(i, N_B)$", *backwards narrowing search* invokes equational unification between the following two terms N_B and $sk(i, M)$, i.e., between the challenge N_B and the intruder action of encrypting a message M using his private key i. The equational theory of NSPK is cancellation of encryption and decryption, described by the following equation $sk(A, pk(A, M)) = M$. And this unification problem $N_B =^? sk(i, M)$ has only one solution $\{M \mapsto pk(i, N_B)\}$ modulo the equational theory of cancellation of encryption and decryption.

2.6 Summary

The main features of Maude-NPA are as follows:

- User-definable protocol syntax. No predefined symbols. Flexible order-sorted structure.
- User-definable cryptographic properties of symbols. Reasoning modulo the cryptographic properties means working with equivalence classes instead of syntactic terms.
- Use of symbolic state patterns (terms with logical variables) instead of ground states (terms without variables). Reasoning with symbolic terms means working with a (possibly infinite) set of instances instead of just one term.
- Backwards symbolic reachability analysis. Providing attack patterns allows a smaller backwards search space than just executing the protocol forwards.
- Use of variant unification to reason about a large class of equational theories.

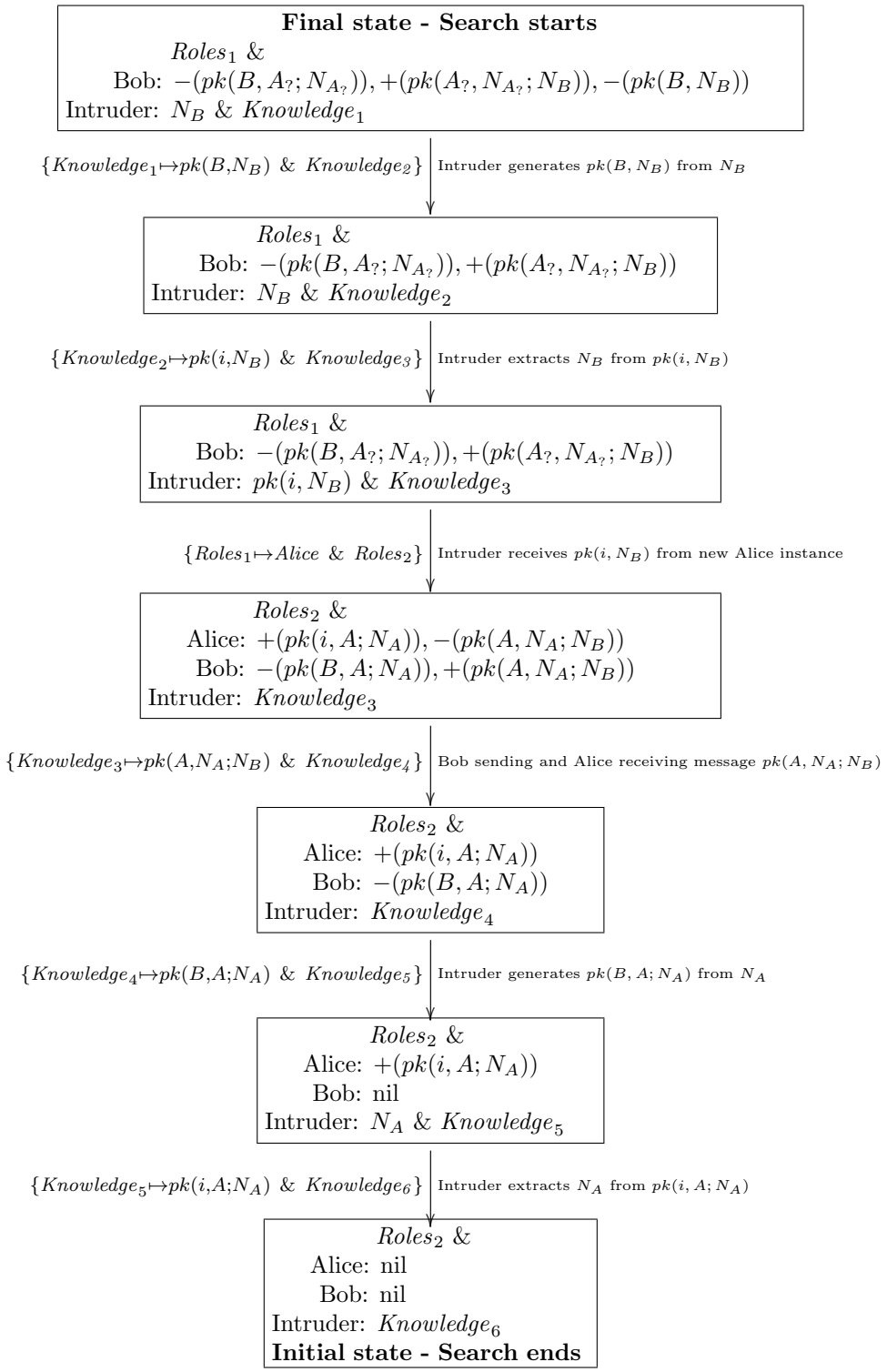


Figure 1: Symbolic execution of man-in-the-middle attack in NSPK

- Honest protocol actions and intruder capabilities are both represented as state transition rules. That is, intruder deduction and state space search are handled in the same way.
- Analysis for an unbounded number of sessions. The use of symbolic state patterns allows extra protocol sessions to be added on the fly.

3 Getting started with Maude-NPA

We assume that the user has installed a copy of Maude 2.7.1. After starting Maude, the user must load Maude-NPA. To do this, the user should be in the Maude-NPA directory. The user should then type the command

```
Maude> load maude-npa
```

In order to load a specification stored in file `foo.maude` one uses the `cd` command to change to the directory where the specification is located, and then types the command

```
Maude> load foo
```

All Maude-NPA specification files must end with the suffix `.maude`. If the user does not want to change to the directory where `foo.maude` sits in, a directory path should be appended when loading it, e.g., `load examples/foo`. Once the protocol specification file `foo.maude` is loaded, it is possible to search for attacks using the commands described in Section 7.

3.1 File structure

Protocol specifications are given in a single file (e.g., `foo.maude`). This file consists of three Maude modules, having a fixed format and fixed module names (see Figure 2). In the first module, the *syntax* of the protocol is specified, consisting of a so-called signature of *sorts* and *operators*. The second module specifies the *algebraic properties* of the operators. Note that algebraic properties must satisfy some specific conditions given in Section 4 and Appendix A. The third module specifies the *actual behavior of the protocol* using a strand-theoretic notation (see Section 5) or a process algebra notation (see Section 9). This module includes the intruder capabilities (also called the Dolev-Yao intruder) and regular strands or processes (describing the behavior of principals). It also contains *attack states* describing behavior that we want to prove cannot occur.

We give a template for Maude-NPA specifications in Figure 2. Throughout, lines beginning with three or more dashes (i.e., `---`) or three or more asterisks (i.e., `***`) indicate comments that are ignored by Maude. The `nonexec` attribute is technically necessary to tell Maude not to use an equation or rule within its standard execution[2].

[2]The `nonexec` attribute will be automatically introduced in future versions of the tool.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  --- Importing sorts Msg, Fresh, and Public
  protecting DEFINITION-PROTOCOL-RULES .
  ---------------------------------------------------------------
  --- Overwrite this module with the syntax of your protocol
  --- Notes:
  --- * Sorts Msg and Fresh are special and imported
  --- * New sorts can be subsorts of Msg
  --- * No sort can be a supersort of Msg
  --- * Variables of sort Fresh denote uniquely generated data
  --- * 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 variant equations eq Lhs = Rhs [variant] .)
  --- * Or use equations for dedicated unification algorithm
  --- * Attribute owise cannot be used
  ---------------------------------------------------------------
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 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 natural number
  [nonexec] .
endfm

--- THIS HAS TO BE THE LAST ACTION !!!!
select MAUDE-NPA .
```

Figure 2: Maude-NPA protocol template

The `nonexec` attribute must be included in all the user-defined equations declared in a Maude-NPA specification file, except for the equational axioms declared together with operators as equational attributes (see Section 4.1).

In what follows we briefly explain the different sections in these three modules using the Needham-Schroeder public key (NSPK) protocol as a running example. Detailed instructions for writing specifications are given in Section 5. Note that, since we are using Maude also as the specification language, following Maude's convention, each declaration has to be ended by a blank space and a period.

3.2 Sorts and Subsorts

We begin by specifying *sorts*. Sorts are used to specify different types of data, that are used for different purposes. We have a special sort called `Msg` that represents what messages are going to look like in our protocol. If a protocol makes no additional sort distinctions, i.e., if it is an unsorted protocol, there will be no extra sorts, and every symbol will be of sort `Msg`.

Sorts can also be *subsorts* of other sorts. Subsorts allow a more refined distinction of data within a given sort. The NSPK protocol uses public key cryptography, and the principals exchange encrypted data consisting of names and nonces. Thus we can define sorts to distinguish names, keys, nonces, and encrypted data:

```
sorts Name Nonce .
subsort Name Nonce < Msg .
subsort Name < Public .
```

The sorts `Nonce` and `Name` are not strictly necessary, but they can make the symbolic search more efficient. Maude-NPA will not attempt to unify terms with incompatible sorts. So, for example, in this specification, if a principal is expecting a term of sort `Name`, then it will not accept a term of sort `Nonce`; technically this is achieved by the fact that `Name` is not declared as a subsort of `Nonce`. However, if we are looking for type confusion attacks, we would not want to include these sorts, and would instead declare everything as having sort `Msg`. See [17] for an example of a type confusion attack.

Maude-NPA is very flexible and most sorts are user-defined. However, there are several special sorts that are automatically imported by any Maude-NPA protocol definition, and for which the user should make sure that certain constraints are satisfied. These are:

`Msg` Sorts defined by the user can be subsorts of `Msg`. No sort defined by the user can be a supersort of `Msg`. This sort cannot be empty, i.e., it is necessary to define at least one symbol of sort `Msg` or of a subsort of `Msg`.

`Fresh` The sort `Fresh` is used to identify terms that must be unique. It is typically used 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`. Only variables can be of sort `Fresh`, not constants or function symbols.

`Public` The sort `Public` is used to identify terms that are publicly available, and therefore assumed known by the intruder. This sort is a subsort of `Msg` and, therefore, it is not necessary to write "`subsort Public < Msg`". This sort cannot be empty.

3.3 Variable declarations

Variables and their sorts 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 in one line, as "`vars X Y Z1 Z2 : Msg .`".

3.4 Operator declarations

Maude-NPA is very flexible and allows several operator declarations. Operators can have either the standard *prefix* syntax (e.g., $f, g, h, \ldots$), the standard *infix* syntax (e.g., the binary infix symbol + is represented as $\_ + \_$ in Maude), or *mix-fix* syntax, allowing user-definable symbols where each underscore represents an argument position (e.g., `if_then_else_fi`). Furthermore, binary infix symbols, e.g. $\_ + \_$, can be declared with *equational attributes* such as associativity, commutativity, and identity (see Section 4.1).

For the NSPK protocol, there are two symbols, `pk` and `sk`, for public and private key encryption, the operator `n` for nonces, unguessable values for principals, and concatenation using the infix operator "$\_;\_$". We begin with the public/private encryption operators declared with prefix syntax.

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

The `frozen` attribute is technically necessary to tell Maude not to attempt to apply rewrites at arguments of those symbols[3]. The `frozen` attribute must be included in all operator declarations in Maude-NPA specifications, excluding constants and macros; macros will be discused in Section 5.4. The use of sort `Name` as an argument of public key encryption may seem odd at first. 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 (e.g., `enc(pubkey(A),M)` instead of `pk(A,M)`).

Next we specify some principal names. 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 NSPK) instead of constants. However, we may wish to specify constant

[3]The `frozen` attribute will be automatically introduced in future versions of the tool.

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* principal, 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.

For NSPK, we should declare, at least, three constants of sort `Name`: `a`, `b`, and `i`.

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

We need two more operators, one for nonces, and one for message concatenation. The nonce operator can be specified using prefix syntax 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 sort `Name` is not strictly necessary, but it provides a convenient way of identifying which nonces are generated by which principals. This makes searches more efficient, since it allows us to keep track of the originator of a nonce throughout a search. We could make things even more specific by keeping track of whether the originator was playing the role of initiator or responder, and including that as another argument (e.g., `n(A,init,x)` instead of `n(A,x)`).

Finally, we come to the 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 `_;_` should be parsed as associated to the left; whereas "`gather (E e)`" indicates association to the right. Note that this is just a parsing attribute to make the use of parenthesis unnecessary, and therefore it is different from the associativity equational property described in Section 4.1. In fact, `_;_` is *not* assumed to be associative in this version of NSPK protocol (see Section 10.5 for a version of Needham-Schroeder-Lowe public key (NSL) with associativity).

3.5 Algebraic Properties

The Maude-NPA performs symbolic reachability analysis *modulo* the equational theory expressing the algebraic properties of the protocol cryptographic functions. 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 (see, e.g., [40]) 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.

For example, the Needham-Schroeder public key (NSPK) protocol has a man-in-the-middle-attack found by the Maude-NPA (see Section 7), whereas the Needham-Schroeder-Lowe (NSL) protocol is proved to be secure by the Maude-NPA (see Section 10.2). However, when equational properties are added to some symbols in the NSL protocol, namely replacing concatenation by exclusive-or or making encryption homomorphic over concatenation, Maude-NPA is able to find two new attacks (see Sections 10.3, 10.4, and 10.5).

There are three types of algebraic properties that can be included in a protocol specification (see Section 4 for further details):

1. *operator equational attributes*, also called *axioms* in this manual, allowing symbols with any combination of associativity (A), commutativity (C), and identity (U) axioms (see Section 4.1);

2. *equations*, also called *variant equations* in this manual (see Sections 4.2 and 4.3), and

3. *metadata equations*; also called *dedicated equations* in this manual, to designate equations associated to dedicated unification algorithms (see Section 4.4).

Variant and dedicated equations are specified in the `PROTOCOL-EXAMPLE-ALGEBRAIC` module, whereas axioms are specified as equational attributes within the operator declarations in the `PROTOCOL-EXAMPLE-SYMBOLS` module. Note that only combinations of axioms and variant equations are allowed, and dedicated equations cannot be combined with anything else (i.e., no other operator attributes or variant equations can be added to the protocol specification if dedicated metadata equations are used).

In NSPK, we consider only cancellation between symmetric public encryption and decryption, which is defined by two variant equations as follows:

```
eq pk(A:Name,sk(A:Name,Z:Msg)) = Z:Msg [variant] .
eq sk(A:Name,pk(A:Name,Z:Msg)) = Z:Msg [variant] .
```

The attribute `variant` specifies that these equations should not be used as regular Maude equations, typically used for simplification, but are instead equations to be used for the variant-based unification algorithm available in Maude (see Appendix A.3 and [23, 11, 10] for details on this unification algorithm). Note that the Maude `owise` attribute for equations should *never* be used in variant equations.

3.6 Protocol and Intruder specifications

The protocol itself and the intruder capabilities are both specified in the `PROTOCOL-SPECIFICATION` module. They are specified using strands (see Section 5 for strand specification) or processes (see Section 9 for process specification). In this section we give a brief introduction to specifying protocol strands.

A *strand*, first defined in [24], is a sequence of positive and negative messages[4] describing a principal executing a protocol, or the intruder performing actions, e.g.,

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

where $r_1, \ldots, r_j$ are all the variables of sort `Fresh` uniquely generated in the strand, a positive node m^+ describes sending the message m, and a negative node m^- describes receiving a message that is an instance of the pattern m.

The vertical bar is used to distinguish between present and future when the strand appears in a state description. All messages appearing before the bar were sent/received in the past, and all messages appearing after the bar will be sent/received in the future. When a strand is used in a protocol specification as opposed to a state description the bar is irrelevant, and by convention it is assumed to be at the beginning of the strand, right after the initial `nil`.

For specifying the intruder capabilities, all intruder strands follow the same form: a sequence of negative variables followed by one positive message combining all previous variables under a function symbol. For example, one of the intruder capabilities is concatenation of two messages, represented by the following strand (the full intruder strand specification can be found in Section 5.1).

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

For specifying the honest protocol participants, we represent each role specification as a strand containing negative messages for the received data and positive messages for the sent data. We recall the informal specification of NSPK, as follows:

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

where N_A and N_B are nonces generated by the respective principals. Here there are just two roles, Alice and Bob, and we define two strands for these:

```
:: 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 ]
```

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

```
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
  ||
  nil
[nonexec] .
```

3.7 Attack States

The last part of a Maude-NPA specification file usually contains the *attack patterns*, which describe the final attack states that we are searching for in Maude-NPA. Unlike the case of strand specifications, we can provide more than one attack pattern by associating a different natural number to each one. Each attack pattern contains several sections separated by the symbol `||`. Only the first two sections can be filled in and correspond, respectively, to the attack state's expected set of strands and expected intruder knowledge. The other sections usually have the symbol `nil`.

For NSPK, the standard attack requires that an instance of Bob's role has participated in the interaction, this instance has finished its execution (by having the vertical bar at the end), and the intruder has learned the nonce, `n(b,r)`, generated by this Bob's instance. This attack is represented by the attack pattern in Figure 3.7 below, where we include Bob's instance in order to refer to the unique variable `r`.

This Maude-NPA attack pattern corresponds to the informal attack pattern given in Figure 1 in Section 2. The reader can see that the logical variable $A_?$ in Figure 1 is instantiated to constant `a` in this attack pattern and the logical variable $N_{A_?}$ in Figure 1 is denoted by variable `N` in this attack pattern. Moreover, the variables for extra role instances, $Roles_1$, and the extra intruder knowledge, $Knowledge_1$, both of Figure 1 are not included in the attack pattern above just for simplicity, but are internally added by Maude-NPA to each attack pattern given in a specification file. Note that extra attention should be given to variables of sort Fresh, such as `r` in the nonce `n(b,r)`, since they are not existentially quantified, as `N` is, or universally quantified, they are just treated as constants by Maude-NPA and can never be instantiated (serving the purpose of an infinite countable set of fresh constants).

It is also possible to include *never patterns* in attack patterns. A never pattern

describes a strand that should not be encountered in a backwards search, and is generally used for authentication properties. For example, suppose that we want to find a state in which A has executed an instance of a protocol, appearently with Bob, but Bob has not executed the corresponding instance with Alice. This can be specified using the following attack pattern with a never pattern:

```
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
|| never(
   *** for authentication
   :: r' ::
   [ nil |
      +(pk(b,a ; N)),
      -(pk(a, N ; n(b,r))),
      +(pk(b,n(b,r))), nil ]
   & S:StrandSet
   || K:IntruderKnowledge
   )
[nonexec] .
```

More detailed information on attack patterns and how to write them may be found in Section 6.

4 ∗Supported Equational Theories

Maude-NPA provides support for unification modulo four types of equational theories.

1. *Syntactic unification* for the empty theory.

2. *Axioms* for symbols obeying equational theories consisting of any combination of associativity (A), commutativity (C), and identity (U) axioms. Note that we can have multiple symbols obeying any combination of axioms: for example we could have a theory in which f and g satisfy C, h satisfies AC, and p and 0 satisfy ACU;

3. Axioms, or the empty theory, together with *variant equations*. These are rewrite rules (rules with an orientation). The requirements they must satisfy are specified in Section 4.5.

4. In some cases, where the algebraic properties do not fit into any of the two previous cases, or when there exists a special-purpose algorithm for an equational theory, *dedicated unification algorithms* have been designed and implemented in Maude-NPA. In this case, some *metadata equations* are expressly associated to the special equational theory in order for Maude-NPA to identify when such an special equational theory is being used by a protocol specification and to invoke the corresponding dedicated unification algorithm in the appropriate way.

In the following, we provide explanations and detailed examples of the three non-trivial algebraic properties.

4.1 Specifying Theories with Axioms

Since Maude-NPA uses special unification algorithms allowing different combinations of associativity (A), commutativity (C), and identity (U) axioms, these are specified not as standard equations but as *axioms* (also called *equational attributes* in the Maude programming language) incorporated as keywords within operator declarations. For example, suppose that we want to specify an exclusive-or operator (except for the self cancellation property which will be specified later by variant equations in Section 4.2). Then, since exclusive-or is associative-commutative, we can specify it with an infix operator "`_*_`" in the `PROTOCOL-EXAMPLE-SYMBOLS` module as follows:

```
op _*_ : Msg Msg -> Msg [frozen assoc comm]
```

where the associativity and commutativity axioms are declared as attributes of the `_*_` operator with the `assoc` and `comm` keywords. We would instead specify an operator that is commutative but not associative with the `comm` keyword alone, and an operator that is associative but not commutative with the `assoc` keyword alone.

When a symbol satisfies associativity but not commutivity (with or without identity), some problems may have an infinite number of most general unifiers. Thus associative unification in Maude is not complete, i.e., it is not always able to return a complete set of most general unifiers for every unification problem. Instead, it finds a set of unifiers whose size is determined by a bound on the search tree. In the case of incompleteness is encountered by the associative unification algorithm, Maude-NPA returns a warning (see Section A.1.1 for details), meaning that the search is not complete and it may be the case that, for the given attack pattern, there is an initial state that the tool cannot find. However, any initial state found by the tool will correspond to a real attack. Appendix B describes an example of a simple protocol using associativity that displays an incompleteness warning encountered during search.

4.2 Combining Variant Equations with Axioms: Exclusive or

Suppose that we want to specify the full theory of exclusive-or with self cancellation. As explained in Section 4.1, we can first specify an infix associative-commutative operator "_*_" in the PROTOCOL-EXAMPLE-SYMBOLS module as follows:

```
op _*_ : Msg Msg -> Msg [frozen assoc comm] .
op null : -> Msg .
```

We then specify the identity and cancellation rules for _*_ in the PROTOCOL-EXAMPLE-ALGEBRAIC module as follows:

```
eq X:Msg * X:Msg * Y:Msg = Y:Msg [variant] . --- Extended cancellation
eq X:Msg * X:Msg = null [variant] .          --- Cancellation
eq X:Msg * null = X:Msg [variant] .          --- Identity
```

Note that the first equational property, i.e., `X * X * Y = Y`, is not an essential part of the exclusive-or theory. It is an extended version of the cancellation property technically needed to make the equations *coherent* modulo associativity and commutativity, see Section 4.5. Note also that, for termination reasons, exclusive-or should not be declared as an ACU symbol in Maude-NPA but only as an AC symbol with an explicit variant equation for the identity property (see Section 4.5 for technical details).

4.3 Combining Variant Equations with Axioms: Diffie-Hellman

To specify Diffie-Hellman exponentiation, 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 key aspects of this theory in detail.

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

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

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

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

We then include the following variant equation 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) [variant] .
```

There are several things to note about this Diffie-Hellman specification. The first is that, although modular multiplication has a unit and an inverse symbol, these are not included in our equational specification. Instead, we have only included basic algebraic properties that are necessary for Diffie-Hellman to work, namely, associativity and commutativity of multiplication, and the above exponentiation equation. The second thing to note 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. The details of this are explained in Section 4.5 and Appendix A.3. The omission of units and inverses is not necessary to ensure finitary unification. The advantage of this omission is that it 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 and expressive model. This could be done along the lines of the proofs in [37, 32, 33].

4.4 Dedicated unification algorithms

We have also developed with some collaborators some dedicated specific unification algorithms for two purposes: (i) allow unification for equational theories not supported by the combination of axioms and variant equations (e.g. homomorphic encryption[5]) and (ii) to improve efficiency for commonly used theories (e.g. exclusive-or or Abelian groups). Currently, we have integrated two dedicated unification algorithms that we describe below.

4.4.1 Built-in Homomorphic Encryption

We model an encryption algorithm that is homomorphic over a binary operator (e.g. concatenation). We assume the operator satisfies the following generic algebraic property

$$e(X; Y, Key) = e(X, Key); e(Y, Key)$$

[5]However, some approximations of homomorphic encryption *can* be specified by suitable combinations of axioms and variant equations, see [42].

However, the actual chosen symbols e and `_;_` are user-definable. For example, suppose that we want to specify a version of the Needham-Schroeder-Lowe (NSL) protocol, which is a version of the Needham-Schroeder public key (NSPK) protocol fixed by Lowe to avoid flaws, but including the algebraic property that encryption is homomorphic over concatenation. Then, the dedicated equation added to the `PROTOCOL-EXAMPLE-ALGEBRAIC` module is as follows:

```
eq pk(X:Msg ; Y:Msg, K:Key) = pk(X:Msg, K:Key) ; pk(Y:Msg, K:Key)
     [nonexec label homomorphism metadata "builtin-unify"] .
```

This dedicated equation contains the `nonexec` attribute. However, two new syntactic features are added in order for the Maude-NPA to identify the appropriate dedicated equation and invoke the dedicated equational unification algorithm for it:

1. A label `homomorphism` is explicitly used to identify this equation. This will help in the future when different dedicated algorithms have to be combined.

2. A generic metadata `builtin-unify` is used to separate dedicated equations from variant equations. This will help in the future when variant equations are executed modulo this dedicated algorithm.

Note that this equational property forces the key to be the second argument of the encryption. Also, only one encryption symbol can be homomorphic over concatenation, that is, only one dedicated equation with the label `homomorphism` is allowed. Note that no other symbol in the protocol can have axioms.

4.4.2 $**$ Built-in Exclusive-or Operator

As demonstrated in Section 4.2, it is possible to define an exclusive-or operator using variant equations and associativity and commutativity. But efficient dedicated unification algorithms do exist for exclusive-or and we have integrated such an algorithm, described in [31]. However, although this unification algorithm is more efficient than variant-based unification, we can not at this point guarantee that this is the case for Maude-NPA. Maude-NPA is currently optimized for variant generation and variant-based unification and more integration of this dedicated unification may be needed. See [12, 13] for a discussion of the integration of this dedicated unification algorithm for exclusive-or in Maude-NPA.

We assume one exclusive-or operator $\_*\_$ and one identity operator 0. The actual chosen symbols 0 and $\_*\_$ are user-definable. Then, the dedicated equations added to the `PROTOCOL-EXAMPLE-ALGEBRAIC` module are as follows:

```
*** Exclusive or properties
eq 0 * X:NNSet = X:NNSet
     [nonexec label XOR-UNITY metadata "builtin-unify"] .
eq X:NNSet * X:NNSet = 0
     [nonexec label XOR-NilPotent metadata "builtin-unify"] .
```

These dedicated equations contain new labels `XOR-UNITY` and `XOR-NilPotent` explicitly used to identify the 0 and $*$ operators.

4.5 General Requirements for Variant Algebraic Theories

As explained in Appendix A, for theories which can be decomposed into a set of axioms and a set of equations and satisfy the requirements explained in this section, Maude-NPA uses a technique called *variant narrowing* to perform unification of symbolic terms *modulo* the variant equations and the axioms specified for the algebraic properties of the protocol. In order for this variant narrowing technique to be sound an complete and to provide a *finite* set of unifiers, six 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 can no longer be guaranteed. We list below these six requirements and explain in detail what they mean. We then explain in Section 4.6 how some protocol examples meet all these requirements.

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 (in Maude Σ is defined by the sort and subsort declarations and the operator declarations, as we have already illustrated with examples), 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 and/or identity 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 together with their corresponding operator declarations by means of the `assoc` and/or `comm` and/or `id:` equational attributes; they are *not* declared as explicit equations. Instead, the equations E are explicitly declared with the `eq` keyword as we have also illustrated with examples.

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 following six requirements:

1. The axioms Ax can declare some binary operators in Σ to have any combination of associativity (A), commutativity (C), and identity (U) axioms.
2. The equations E are confluent modulo Ax.
3. The equations E are terminating modulo Ax.
4. The equations E are coherent modulo Ax (see [30]).
5. The equations E are sort-decreasing.
6. The equations E have the finite variant property (see [23, 6]).

We now explain in detail what these requirements mean.

4.5.1 Rewriting modulo Axioms

Since Ax-unification is supported for the combinations of axioms Ax described in requirement (1), this implies that *Ax-matching* (the special case in which one of

the terms being unified is a ground term without any variables) is also supported, so that we in effect can use the equations E to rewrite terms *modulo* Ax. This is of course supported by Maude for axioms such as associativity, commutativity, and identity. Suppose, for example, that a + symbol has been declared commutative with the `comm` attribute, and that we have an equation in E of the form $x + 0 = x$. Then we can apply such an equation 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. We would express this rewrite step of simplification modulo commutativity with the arrow notation:

$$0 + 7 \rightarrow_{E/Ax} 7$$

where E is the set of equations containing the above equation $x + 0 = x$, and where Ax is the set of axioms containing the commutativity of $+$. Likewise, we denote by $\longrightarrow^*_{E/Ax}$ the reflexive-transitive closure of the one-step rewrite relation $\longrightarrow_{E/Ax}$ with the equations E modulo the axioms Ax. That is, $\longrightarrow^*_{E/Ax}$ corresponds to taking zero, one, or more rewrite steps with the equations E modulo Ax.

4.5.2 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 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 our above example we have, for instance, $0+7 =_{Ax} 7+0$.

The Church-Rosser checker included in The Maude Formal Environment [1] can be used to check that the equations of a module are confluent modulo axioms, assuming the equations are terminating modulo the axioms (see below).

4.5.3 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

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

The Maude Termination Tool (MTT) included in The Maude Formal Environment [1] can be used to check that the equations of a module are terminating modulo the axioms.

4.5.4 Coherence

The $\rightarrow_{E/Ax}$ relation is a relation on equivalence classes modulo Ax, and is generally undecidable. But if E is *strictly coherent modulo* Ax it becomes decidable and can be implemented efficiently via a procedure known as E, Ax rewriting, in which rewriting is performed on representatives of the equivalence classes. Rather than explaining the *strict coherence modulo* Ax notion in general (the precise definition of strict coherence, and its relation to $\rightarrow_{E/Ax}$ and $\rightarrow_{E,Ax}$ can found in [36]), we explain in detail its meaning in cases where it is needed for the Maude-NPA, namely, the cases of AC, ACU, and A 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 the above equation to simplify the term above to the term $a \oplus 0$ in one step. However, 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 we can slightly simplify to the equation $x \oplus x \oplus y = y$ by using the equation $x \oplus 0 = x$. This extended version of our original equation will now apply to the term $b \oplus (a \oplus b)$, giving the simplification $b \oplus (a \oplus b) \longrightarrow_{E,Ax} a$.

Here, we show how to guarantee coherence for the three theories most commonly used in Maude-NPA: AC, ACU, and A: [6]

1. For any symbol f which is AC, and for any equation of the form $f(u, v) = w$ in E, we just add the equation $f(f(u, v), x) = f(w, x)$, where x is a fresh new variable.

2. If f is ACU with identity symbol e, the original equation $f(u, v) = w$ is *replaced* by the extended equation $f(f(u, v), x) = f(w, x)$ shown above, instead of being added.

3. Likewise, if f is associative only, the following *extended equations* should be added: $f(x, f(u, v)) = f(x, w)$, $f(f(u, v), y) = f(w, y)$, and $f(x, f(f(u, v), y)) = f(x, f(w, y))$.

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 Ax, so that we need not add the extra equation. This can be checked by determining if the new equation $s = t$ can be derived from the already existing equations. Consider for example the exclusive-or theory $x \oplus 0 = x$, $x \oplus x = 0$ and

[6] Note that The Maude Formal Environment [1] has a Coherence checker but it does not apply to the coherence modulo axioms associated to Maude-NPA.

$x \oplus x \oplus y = y$, where $\oplus$ is AC. Consider the extended equation $(x \oplus 0) \oplus z = x \oplus z$ constructed using $x \oplus 0$ and 1) from above to guarantee coherence. Since $x \oplus 0 = x$, we have $(x \oplus 0) \oplus z = x \oplus z$, so the equation already follows from the original theory.

Also, if we assume that symbol $\oplus$ is ACU instead of AC, then the previous three equations $x \oplus 0 = x$, $x \oplus x = 0$ and $x \oplus x \oplus y = y$ for $*$ being AC can be simplified into one equation $x \oplus x \oplus y = y$ for $*$ being ACU with 0 as the identity symbol. Note that this equation is coherent modulo ACU but it is not terminating modulo ACU since for any term T, we have $T =_{ACU} 0 \oplus 0 \oplus T$ and $0 \oplus 0 \oplus T \rightarrow_{E/ACU} T$ using the equation $x \oplus x \oplus y = y$, Because of this, exclusive-or must be specified in Maude-NPA with an AC symbol and never with an ACU symbol.

4.5.5 Sort-decreasingness

Sort-decreasingness is also easier to explain by its absence, with an example. Suppose that for the cancellation of encryption and decryption we use one subsort Encoding of the sort Msg and the following definitions for symbols pk and sk:

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

Then the following equations for cancellation are *not* sort-decreasing, since the left-hand sides are defined as elements of sort Encoding but the application of the equations may return elements of a greater sort Msg:

```
eq pk(A:Name,sk(A:Name,Z:Msg)) = Z:Msg [variant] .
eq sk(A:Name,pk(A:Name,Z:Msg)) = Z:Msg [variant] .
```

In general, an equation $u = v$ is called *sort-decreasing* iff for any substitution instance $u\theta = v\theta$, if $v\theta$ has some sort S, then $u\theta$ must also have sort S. This is precisely what failed to happen in the example above, where S was chosen to be Msg.

The Church-Rosser checker included in The Maude Formal Environment [1] automatically checks whether the equations are sort-decreasing.

4.5.6 The Finite Variant Property

The previous version of Maude-NPA did not allow theories with the finite variant property in their full generality. Instead it imposed extra syntactic conditions on the equational theory so that the theory would fall within a limited class of theories guaranteed to have the finite variant property. In this version we use the full implementation of the folding variant narrowing strategy developed for Maude 2.7, which allows theories satisfying the admissibility conditions (1)–(5) above and having the finite variant property, with the only restriction that the axioms of the equational theory be any combination of associativity (A), commutativity (C), and identity (U).

Recall that, given a theory $T = (\Sigma, E \uplus Ax)$, where E is a set of rewrite rules and Ax is a set of axioms, we say that a term t in $T(\mathcal{X})$ is *normalized* or *simplified* if no rewrite rules can be applied to any members of the Ax-equivalence class t belongs to. We say that t is a *normal form* of s, if t can be produced from s via a finite number of E, Ax rewriting steps. Furthermore, if $E \uplus Ax$ satisfies the first five admissibility conditions given in Section 4.5, then every term in T has a unique normal form modulo Ax, and it can be found a finite number of decidable rewriting steps. We refer to this unique normal form of t as $t{\downarrow_{E,Ax}}$. We can then define the set of (E, Ax)-*variants* of a term t as the set of all pairs of the form $(\sigma, \sigma(t){\downarrow_{E,Ax}})$ where σ is a substitution and $\sigma(t){\downarrow_{E,Ax}}$ is the normal form of σt.

For example, given the equational theory $E \uplus AX$ for exclusive-or shown in Section 4.2, and the term $X{:}Msg * Y{:}Msg$, we can construct several of its variants as follows:

1. The pair $(\{X{:}Msg \mapsto a * b, Y{:}Msg \mapsto a * b,\ a * b * a * b)$ is normalized to $(\{X{:}Msg \mapsto a * b, Y{:}Msg \mapsto a * b, \text{null})$;

2. The pair $(\{X{:}Msg \mapsto a{*}b{*}U{:}Msg, Y{:}Msg \mapsto a{*}b, a{*}b{*}U{*}a{*}b)$ is normalized to $(\{X{:}Msg \mapsto a * b, Y{:}Msg \mapsto a * b, U)$, and;

3. The pair $(\{X{:}Msg \mapsto a * b * U{:}Msg, Y{:}Msg \mapsto a * b * V, a * b * U{:}Msg * a * b * V{:}Msg)$ is normalized $(\{X{:}Msg \mapsto a * b, Y{:}Msg \mapsto a * b\}, U{:}Msg * V{:}Msh)$.

We say that a variant (θ_1, t_1) of t is *more general* than a variant (θ_2, t_2), if there is a ρ such that

1. $t_1\rho = t_2$ modulo Ax, and;

2. $\theta_2 = \theta_1\rho{\downarrow_{E,Ax}}$ modulo Ax.

Thus, the variant $(\{X \mapsto Z, Y \mapsto Z\}, Z * Z{\downarrow_{E,Ax}})$ is strictly more general than $(\{X \mapsto a * b, Y \mapsto a * b\}, a * b * a * b{\downarrow_{E,Ax}})$ even though both $Z * Z$ and $a * b * a * b$ normalize to the same term null, because $\{X \mapsto Z, Y \mapsto Z\}$ is strictly mor general than $\{X \mapsto a * b, Y \mapsto a * b\}$.

A set of variants V_t of a term t with the property that for every variant $(\sigma t \sigma)$ of t, there is a more general variant of t in V_t is called a *set of most general* variants of t. For example, the following is a set of most general variants of $a * V$, where a is a constant and V is a variable:

$$\{(\iota, a * V), (V \mapsto a * U, U), (V \mapsto \text{null}, a), (V \mapsto a, \text{null}\}$$

Given a theory $T = (\Sigma, E \cup Ax)$, we say the decomposition $E \uplus Ax$ has the *finite variant property* if for every term t, there is a finite set V_t of most general (E, Ax)-variants of t. Let us illustrate this concept with a positive and a negative example of theories having, or failing to have, the finite variant property.

```
Maude> get variants in PROTOCOL-EXAMPLE-ALGEBRAIC : X:Msg * Y:Msg .

Variant #1
Msg: #1:Msg * #2:Msg
X:Msg --> #1:Msg
Y:Msg --> #2:Msg

Variant #2
Msg: %2:Msg * %3:Msg
X:Msg --> %1:Msg * %2:Msg
Y:Msg --> %1:Msg * %3:Msg

Variant #3
Msg: %2:Msg
X:Msg --> %1:Msg * %2:Msg
Y:Msg --> %1:Msg

Variant #4
Msg: %2:Msg
X:Msg --> %1:Msg
Y:Msg --> %1:Msg * %2:Msg

Variant #5
Msg: null
X:Msg --> %1:Msg
Y:Msg --> %1:Msg

Variant #6
Msg: %1:Msg
X:Msg --> null
Y:Msg --> %1:Msg

Variant #7
Msg: %1:Msg
X:Msg --> %1:Msg
Y:Msg --> null

No more variants.
```

Figure 3: Result of “get variants” command

The equational theory for exclusive-or shown in Section 4.2 does have the finite variant property. For example, the term $X * Y$ has a set of seven most general variants, which are shown in Figure 3 using the Maude command "`get variants`".

The key idea for the finite variant property, is that, given a term t and a (normalized) substitution θ, any pair $(\theta, \theta(t)\downarrow_{E,Ax})$ must be either equal to or a further instantiation of some most general variant of t. For example, the substitution $X \mapsto a * b * U, Y \mapsto a * b * V$ maps $X * Y$ to a term that reduces to $U * V$. This variant is a special case of Variant #2 produced by Maude-NPA. If we then consider a further instantiation $\{\texttt{U} \mapsto \texttt{c * d}, \texttt{V} \mapsto \texttt{c * d}\}$ of the term `a * b * U * a * b * V`, then the term `a * b * c * d * a * b * c * d` is simplified into `null`, with the composed substitution $X \mapsto a * b * c * d, Y \mapsto a * b * c * d$. This is a special case of Variant #5 produced by Maude-NPA.

Let us now illustrate this concept with a negative example. Consider a different equational theory with a sort `Elem` for constants that we will show does not have the finite variant property. We will refer to this theory as the *elem-xor* theory.

```
op _*_ : Msg Msg -> Msg [frozen assoc comm] .
op null : -> Msg .

sort Elem .
subsort Nonce < Elem < Msg .
ops a b c d : -> Elem .

eq X:Elem * X:Elem * Y:Msg = Y:Msg [variant] .
eq X:Elem * X:Elem = null [variant] .
eq Y:Msg * null = Y:Msg [variant] .
```

The key difference here is that the variable X: Msg in the xor could be instantiated not only to a constant but to any xor product of constants, but the variable X: Elem in the elem-xor theory can only be instantiated to one constant. Thus, Maude-NPA will produce the an infinite set of variants $\{(\sigma_k, t_k) | 1 \leq k \leq \infty\}$ of $X * Y$, where σ_k is of the form

$$\{X{:}\text{Msg} \mapsto Z_1{:}\text{Elem} * \ldots * Z_k{:}\text{Elem} * U_k{:}\text{Msg}, Y \mapsto X_1{:}\text{Elem} * \ldots * Z_k{:}\text{Elem} * V_k{:}\text{Msg}\}$$

and t_k is of the form $U{:}\text{Msg} * V{:}\text{Msg}$.

To see see why this is the case, consider the two variants $(\sigma_1, t_1) = (\{X \mapsto Z_1{:}\text{Elem} * U_1{:}\text{Msg}, Y \mapsto Z_1\text{Elem} * V_1{:}\text{Msg}, U_1{:}\text{Msg} * V_1{:}\text{Msg}\}$ and $(\sigma_2, t_2) = (\{X \mapsto Z_1{:}\text{Elem} * Z_2{:}\text{Elem} * U_2{:}\text{Msg}, Y \mapsto Z_1{:}\text{Elem} * Z_2\text{Elem} * V_2{:}\text{Msg}, U_2{:}\text{Msg} * V_2{:}\text{Msg}\}$. We will show that neither one is more general than the other. Clearly, (σ_2, t_2) is not more general than (σ_1, t_1), since σ_2 is not more general then σ_1 modulo Ax. To show the other way around we note that σ_1 *is* more general than σ_2; if we let $\rho = \{U_1 \mapsto X_2{:}\text{Elem} * U_2{:}\text{Msg}, V1 \mapsto Y_2{:}\text{Elem} * V_2{:}\text{Msg}$, then $\sigma_2 = \rho\sigma_1$. On the other hand, we have $t_1\rho = Z_2{:}\text{Elem} * U_2{:}\text{Msg} * Z_2{:}\text{Elem} * V_2\text{Msg}$, which is *not* equal to

$t_2 = U_2$:Msg $*$ U_2:Elem modulo Ax. So the second criterion for (σ_1, t_1) being more general then (σ_2, t_2) is not satisfied.

Whether or not an equational theory has the finite variant property is undecidable [5]. However, if the theory *does* have the finite variant property, it is quite easy to check that it does; and if it *does not*, it is also quite easy to get strong empirical evidence suggesting that it either it does not, or the number of variants is so large that they are not practical to compute. A semi-decision procedure that works well in practice was introduced in [6]. This check can be performed in Maude by using its `get variants` command. The procedure is as follows: if for each operator declaration $f : A_1\ A_2\ \cdots A_n \rightarrow A$, the set of variants obtained by the `get variants` command for the term $f(X_1 : A_1, \ldots, X_n : A_n)$ is finite, then the the theory *does* have the finite variant property. And if any term $f(X_1 : A_1, \ldots, X_n : A_n)$ does have an infinite set of variants (which in practice will be suggested by Maude not terminating its process of variant generation on screen), then the theory does *not* have the finite variant property.

4.6 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)–(6) in Section 4.5, it may be useful to illustrate how these requirements are met by several example theories. 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 3.5.

Note that just because Maude-NPA can *theoretically* handle an equational theory meeting conditions (1)–(6) above, this does not mean that a formal analysis can always be practical for all protocols. The number of states generated during backwards reachability search for a protocol depends on many factors, e.g. on how general the attack pattern is, how general the protocol is, which are the possible implications of the equational theory for the protocol and the attack pattern, etc. In some cases the number of states generated may be impractically large even though the search space is finite. We advise the user to start out by specifying the least complex theories possible when analyzing a protocol, and then incrementally increase their complexity. For example, the different versions of NSL described in Sections 10.3, 10.4, and 10.5 have different equational theories, but the corresponding search spaces are quite similar because they use the same attack pattern. An interesting measurement is the *variant complexity* of an equational theory, i.e., the sum of the number of variants generated for each pattern in the theory as demonstrated in Section 4.5.6. If an equational theory has a high variant complexity and this happens modulo axioms like AC or ACU, some unification call may have an exponential number of unifiers and, then, some theoretically possible analyses may become unfeasible in practice. However, this depends on the actual terms of each unification call, and it may be the case that by restricting the actual messages exchanged in a protocol, the analysis becomes feasible.

4.6.1 Cancellation of Encryption and Decryption

Let us begin with the theory of Encryption/Decryption:

```
op pk : Name Msg -> Msg [frozen] .
op sk : Name Msg -> Msg [frozen] .

eq pk(A:Name,sk(A:Name,Z:Msg)) = Z:Msg [variant] .
eq sk(A:Name,pk(A:Name,Z:Msg)) = Z:Msg [variant] .
```

In this case $Ax = \emptyset$. It is easy to see 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 in The Maude Formal Environment [1], or even by hand. Since $Ax = \emptyset$, coherence modulo axioms is a mute point. The equations have also the finite variant property, which can be verified by generating the variants in Maude for each pattern `pk(A:Name,M:Msg)` and `sk(A:Name,M:Msg)`.

4.6.2 Exclusive-or

Let us now consider the Exclusive Or Theory:

```
op _*_ : Msg Msg -> Msg [frozen assoc comm] .
op null : -> Msg .

eq X:Msg * X:Msg * Y:Msg = Y:Msg [variant] .
eq X:Msg * X:Msg = null [variant] .
eq X:Msg * null = X:Msg [variant] .
```

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, such as the Church-Rosser checker included in The Maude Formal Environment [1]. *Coherence* modulo AC is also easy. As already explained in Section 4.5.4, the first equation has to be added to the second to make the equations coherent modulo AC.

The equations also have the finite variant property, which was already demonstrated in Section 4.5.6.

4.6.3 Diffie-Hellman Exponentiation

Turning now to the Diffie-Hellman theory we have:

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

op g : -> Gen .
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) [variant] .
```

Again, this theory is AC. *Termination* modulo AC is easy to prove by the MTT tool in The Maude Formal Environment [1] or by using a polynomial ordering with AC polynomial functions. 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, which could be checked by the Church-Rosser checker in The Maude Formal Environment [1]. 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 can 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. The equation also has the finite variant property, which can be verified by generating the variants in Maude for each pattern `exp(W:GenvExp,Y:NeNonceSet)`.

4.7 Failure to meet conditions

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 either the two dedicated unification algorithms shown in Sections 4.4.1 and 4.4.2, or order-sorted theories of the form $T = (\Sigma, E \cup Ax)$ satisfying the admissibility requirements (1)–(6). Under assumptions (1)–(6), T-unification problems are always guaranteed to have a finite number of solutions.

As a final *caveat*, if the user specifies a theory T where any of the above conditions (1)–(6) fail, besides the lack of completeness that would be caused by the failure of

conditions (2)–(5), a likely consequence of failing to meet condition (6) will be that the tool may loop forever trying to solve a unification problem associated with even just a single transition step in the symbolic reachability analysis process.

5 Protocol Specification

Maude-NPA uses strands for both protocol specification and the specification of the intruder capabilities. A *strand*, first defined in [24], is a sequence of positive and negative messages[7] describing a principal executing a protocol, or the intruder performing actions, e.g.,

$$[\ m_1{}^{\pm},\ \ldots,\ m_i{}^{\pm}\ |\ m_{i+1}{}^{\pm},\ \ldots,\ m_k{}^{\pm}\ ]$$

where a positive node implies sending, and a negative node implies receiving. However, we have included two differences in our tool: (i) each strand is divided into the *past* and *future* parts, by means of the vertical bar "|" and (ii) we keep track of all the variables of sort `Fresh` generated by that concrete strand. That is, the messages to the left of the vertical bar were sent or received in the past, whereas the messages to the right of the bar will be sent or received in the future. Right before the strand, the variables $r_1, \cdots, r_i$ of sort `Fresh` are made explicit, as shown for the Needham-Schroeder public key (NSPK):

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

Note that variables are not shared between strands in a protocol specification; but they can be shared between actual strands in a protocol execution state.

5.1 Dolev-Yao Strands

The Dolev-Yao strands specify the intruder capabilities. An intruder strand consists 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 this 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.

Encryption with a public key is specified as follows. Note that we use a principal's name to stand for the key. The intruder can encrypt any message using any public key.

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

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

Encryption with the private key is a little different. The intruder can only apply the `sk` operator using his own identity and therefore his own secret key. So we specify the corresponding strand as follows.

```
:: 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 X and Y. Since each intruder strand can 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 symbol `STRANDS-DOLEVYAO` as the repository of all the Dolev-Yao strands, and the symbol `&` as the union operator for 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] .
```

5.2 When to Include/Exclude Operations in the Dolev-Yao Strands

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 would 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 would 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 operation that are defined, unless one is explicitly making the assumption that the intruder can *not* perform the given operation.

5.3 Protocol Strands

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

In specifying protocol strands it is important to remember to do so *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 term of sort `Nonce`. She then encrypts that term of sort `Nonce` under `B`'s public key and sends it out. In other words, data received by a principal whose structure is unverifiable by that principal must be represented by a variable of sort `Msg`.

In order to represent the initiator's strand, we model the construction of `A`'s nonce explicitly as `n(A,r)`, where `r` is a variable of sort `Fresh` belonging to `A`'s strand. The nonce she receives, though, is represented by a variable `N` of sort `Nonce`, as follows:

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

If we wanted to check for type confusion attacks we would replace `N` of sort `Nonce` by a variable `X` of sort `Msg` (where sort `Nonce` is a subsort of sort `Msg`). However his would give rise to a bigger search space due to the more general sort of the variable.

In the responder strand the signs of the messages are reversed. Moreover, the messages themselves are represented differently, since they are seen *from the receiver's point of view.* `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 above, the point here is to only specify things in a strand that a principal executing a strand can actually *verify.* If we say that a principal

receives a term of sort `Nonce`, we assume that the principal has some ability to determine whether something is a nonce, as opposed to some other type of message (perhaps by its length). We do not assume, however, that the principal is able to verify who created that nonce or when.

The complete `STRANDS-PROTOCOL` specification is as follows. We note that in this specification, when a principal receives a nonce that she did not create encrypted under her own public key, she is able to decrypt it and determine whether it is of sort `Nonce`.

```
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] .
```

Remember that our tool considers that variables are not shared between strands, and thus, will appropriately rename them when necessary.

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 in this case is that `A` received some term `Y` of sort `Msg`.

5.4 Macros

We may have protocols with very large messages that keep showing up during the specification. In this case, Maude-NPA allows for a simple trick. Consider for instance the Diffie-Hellman protocol. In this protocol, messages of the form "`A ; B ; exp(g,n(A,r))`" appear many times and we may define the following operator:

```
op m1 : Name Name Fresh -> Msg .
eq m1(A,B,r) = A ; B ; exp(g,n(A,r)) .
```

Then we can replace any occurrence of a message of that form by the expression `m1(A,B,r)`. Indeed, we can define as many macro operators as we need. For example, we may instantiate the previous `m1` to the case where we fix constants `a` and `b`.

```
op mab : Fresh -> Msg .
eq mab(r) = a ; b ; exp(g,n(a,r)) .
```

Caveats when using macros:

- We cannot define a macro with variables in the right-hand side that do not also appear on the left; e.g. "`eq nA = n(A,r) .`" is not allowed if `A` and `r` are variables.

- We have to be careful that the macros are confluent and terminating, since they are expressed as Maude equations (without the `variant` attribute). This can be guaranteed by having only one term defined for each macro, and avoiding cycles in macros.
- These Maude equations will be applicable all over the place during the execution of Maude-NPA, and they may yield unpredictable results[8]. For example, imagine the consequences of saying "`eq n(A,r) = n(i,r) .`", which replaces any occurrence of a nonce by a nonce generated by the intruder.

Generally speaking, macros should be used as *definitional extensions*, which abbreviate an already expressible term by a new function symbol equal to that term by definition.

The `red new-strands?` command is useful for displaying the actual strands that are produced using the macros.

6 Maude-NPA States and Attack Patterns

Attack patterns describe the final attack states we are looking for with Maude-NPA. However, for each attack pattern, Maude-NPA performs a backwards reachability analysis and we first need to describe the states generated during the backwards search.

6.1 Maude-NPA Search States

In Maude-NPA, each state found during the backwards analysis (i.e., a backwards search) is represented within different sections separated by the symbol `|` in the following order: (1) state Id, (2) set of current protocol and intruder strands, (3) intruder knowledge, (4) sequence of messages, and (5) extra data that will be described in Section 7.6. For instance, the following is a state found during the analysis of the NSPK protocol:

```
< 1 . 2 > (
:: nil ::
[ nil |
   -(pk(i, n(b, #0:Fresh))),
   +(n(b, #0:Fresh)), nil]  &
:: #0:Fresh ::
[ nil,
   -(pk(b, a ; #1:Nonce)),
   +(pk(a, #1:Nonce ; n(b, #0:Fresh))) |
   -(pk(b, n(b, #0:Fresh))), nil] )
|
n(b, #0:Fresh) !inI,
```

[8] See Section 9.4 on *equational abstractions* of the Maude manual available online at `http://maude.cs.uiuc.edu`.

```
pk(b, n(b, #0:Fresh)) inI,
pk(i, n(b, #0:Fresh)) inI
|
-(pk(i, n(b, #0:Fresh))),
+(n(b, #0:Fresh)),
-(pk(b, n(b, #0:Fresh)))
|
nil
```

The intruder knowledge represents what messages the intruder knows (symbol `_inI`) or doesn't yet know (symbol `_!inI`) at each state of a protocol execution. Note that the symbol `_!inI` represents that a message is not known now but will be known in the future[9]. The set of current strands indicates how advanced each strand is in the execution process (by the placement of the bar), and gives partial substitutions for the messages in each strand. Note that the set of strands and the intruder knowledge *grow along with* the backwards reachability search, in one case by introducing more protocol or intruder strands, and in the other case by introducing more positive intruder knowledge (e.g., `M inI`) or by transforming positive into negative knowledge due to the backwards execution (e.g., `M inI` ⇒ `M !inI`). The sequence of messages, which is nil at the beginning, gives the actual sequence of messages communicated. This also grows as the backwards search continues, and gives a complete description of an attack when an initial state is reached. This part is intended to provide the user assistance in interpreting the attack path, and is not actually used in the backwards search. Finally, the last part is used to store information about the search space that the tool creates to help manage its search.

6.2 Attack patterns

Attack patterns define final states used for backwards search. There are two important differences w.r.t. search states: (i) we use the symbol `||` as separators in an attack pattern instead of the symbol `|` used in generated states, and (ii) we have an extra section, so that now there are six sections in total instead of the five sections of a generated state.

The user can specify only the first two sections of an attack state: the set of strands expected to appear in the attack, and the intruder knowledge. The state id will be the number 0 and the other sections must have just the empty symbol `nil`.

For NSPK, the standard attack requires the intruder to have learned the nonce generated by Bob, and thus we have to include a finished execution of Bob's strand in the attack pattern (having the vertical bar at the end) in order to describe the specific nonce `n(b,r)`:

```
eq ATTACK-STATE(0) =
```

[9]Since we are performing *backwards* reachability analysis, we are *coming from the future* where messages are known by the intruder into the *past*, where messages are not known to the intruder. For example, in Figure 1 it is possible to see when a message, e.g. N_B, disappears from the intruder knowledge when moving one step backwards.

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

We note that Maude-NPA does *not* check that the strands in an attack pattern are instantiations of strands in a specification. This is left to the user.

6.3 Constraints

In Maude-NPA the user has the option of specifying *constraints* on variables appearing in an attack pattern. A constraint is expressed as the conjunction of equality constraints ($t_1 = t_2$) and disequality constraints (t_1 != t_2), separated by commas. Any term t appearing in a constraint is built by applying function symbols to variables appearing in the state.

For example, suppose that we want to specify that a responder in NSPK executes a strand, but the initiator cannot be the intruder. 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 ]
  || A != i, N != n(b,r)
  || nil
  || nil
  || nil
[nonexec] .
```

Maude-NPA handles equality and disequality constraints that appear in a state in different ways. In the case of an equality constraint, it attempts to solve the constraint using equational unification, as part of the backwards search process, generating one predecesor state for each unifier. However, Maude-NPA does not support disunification (still largely an open problem for AC theories), so the disequality constraints are only checked for unsatisfiability (i.e., whether the two terms in a disequality are actually equal modulo the protocol's equational theory) during the search. Final checking of constraints is saved until an initial state is reached, at which point the constraints will be completely instantiated and disunification

becomes trivial. For an in-depth discussion of how Maude-NPA handles disequality constraints in theories with finite variant decompositions, we refer the reader to [22].

Although Maude-NPA does not support disunification, it does support state space reduction techniques that are used to eliminate disequality constraints that are trivially satisfiable or unsatisfiable. These are primarily designed for constraints that appear at some point in the search process and get partially instantiated at another point in the search process; but apply also to constraints specified by the user as well.

For the above reasons, it is *very important* that, when a user is specifying a constraint as part of an attack state, special attention is given to the variables included in the constraint. For example, if all the variables in the constraint do not appear in the main body of the state, Maude-NPA will silently remove any such constraint, resulting in what appears to be false attacks. Another example is when the user specifies a constraint about the form a term takes. For example, suppose that we want to specify that a responder in NSPK executes a strand, apparently with an initiator `a`, but the nonce received is not generated by `a`, i.e., it does not satisfy the form `n(a,r')` for any variable `r'`. The user may think that this could 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
  || nil
[nonexec] .
```

However, Maude-NPA is able to find an initial state where an Alice strand is added with fresh variable `r''` and `N` is mapped to the nonce `n(a,r'')`, since this satisfies the disequality `n(a,r'') != n(a,r')`. Such constraints about the form of a term may be supported in the future, but for the time being they are better expressed using *never patterns*, which are described in Section 6.4.

6.4 Attack States With Excluded Patterns: Never Patterns

The last section of an attack pattern is used for what we call *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 have finished its execution without the corresponding initiator strand being present. For this there is an optional additional field in the attack state containing the never patterns. Never patterns are used to describe the strands that should not show up in a search.

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

```
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
 || never(
    *** for authentication
    :: r' ::
    [ nil |
       +(pk(b,a ; N)),
       -(pk(a, N ; n(b,r))),
       +(pk(b,n(b,r))), nil ]
    & S:StrandSet
    || K:IntruderKnowledge
    )
 [nonexec] .
```

Note that the never pattern does not contain the extra fields included in an attack pattern so that only strands and intruder knowledge are specified. Note that variables from the regular strands or the regular intruder knowledge may also appear in the never pattern as a way to further constrain the never pattern. For instance, the variable `r'` in the never pattern is not an error, since the regular strand in the attack state is the initiator strand, which generates the variable `r`, and the strand in the never pattern is a pattern that must be valid for any responder strand. Any responder strand would generate its own variable `r'`.

If we want to restrict the existence of a responder strand even more, so that no *partial responder strand* can show up, we should include a second never pattern ending in a positive message of the strand (since partial strands will always end in a positive message and represent incomplete strands):

```
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
 || never(
    *** for authentication
    (:: r' ::
```

```
    [ nil |
       +(pk(b,a ; N)),
       -(pk(a, N ; n(b,r))),
       +(pk(b,n(b,r))), nil ]
    & S:StrandSet
    || K:IntruderKnowledge)
    *** for authentication
    (:: r' ::
    [ nil |
       +(pk(b,a ; N)), nil ]
    & S:StrandSet
    || K:IntruderKnowledge)
    )
  [nonexec] .
```

Note that variable names used in different never patterns but not in the main body of the attack have no effect, since each never pattern is checked independently. The tool will now search for all backwards transition paths in which the intruder strand is executed, but no (partial) responder strand is present.

It is also possible to use never patterns to specify negative conditions on terms or strands. Recall the attack pattern in Section 6.3 where we are interested in a responder of the NSPK protocol, apparently talking to the initiator `a`, but the nonce received was not the initiator's (represented by `n(a,r')`). This can be done as follows, instead of using the disequality `N != n(a,r')` discussed in Section 6.3:

```
 eq ATTACK-STATE(2) =
  :: r ::
  [ nil,
     -(pk(b,a ; N)),
     +(pk(a, N ; n(b,r))),
     -(pk(b,n(b,r))) |
     nil ]
  || empty
  || nil
  || nil
  || never(
     *** for authentication
     (:: 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)
     )
  [nonexec] .
```

Note that it is very important that the never pattern uses the same variable `r` of sort Fresh as the responder strand, since we want to specify that exactly that

responder strand cannot satisfy the never pattern. Also, note that the variable `r'` of sort Fresh is *not considered as a constant*, as `r` does, but as a *meta variable* of sort Fresh used only for matching. That is, for any possible instantiation of the form $\{\texttt{N} \mapsto \texttt{n(a,r'')}\}$ for any variable `r''`, the term `n(a,r'')` does match with the expression `n(a,r')` of the never pattern.

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

```
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
 || never(
    *** cut down search for two nonces
    :: nil ::
    [ nil |
       -(N1 ; N2),
       +(pk(A, N1 ; N2)), nil ]
    & S:StrandSet
    || K:IntruderKnowledge
    )
 [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 mean 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.

For good examples 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 an authentication attack pattern for the Needham-Schroeder-Lowe protocol in Section 10.2, and for the Diffie-Hellman protocol in Section 10.6.

6.5 Attack Pattern Summary

In summary, we note the following conditions on attack pattern specifications:

- Strands in the attack pattern must have the bar at the end.
- 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 strand appears, or no term appears, the `empty` symbol is used, in the strands or intruder knowledge sections, respectively.
- The items that can appear in the intruder knowledge may include not only terms known by the intruder, but also disequality constraints on terms.
- If constraints are included into an attack pattern, any variable appearing in such constraints must also appear in the main body of the state.
- The two fields after the intruder knowledge must always be `nil`. These are fields that contain information that is built up in the backwards search, but is empty in the final attack state.
- The last field will usually be `nil`, except when a never pattern is included.
- The bar in any strand in the never pattern should be at the beginning of the strand. If it is not, the tool will enforce it.
- One may consider several prefixes of a strand as different never patterns if one wants to discard different output messages in the strand.
- The first two fields of any never pattern must end in variables of type `Strandset` and `Intruderknowledge`, respectively.
- Variables shared between the state and the never pattern will be considered as the same variable. However, variables shared between never patterns themselves are considered as different, so that in that case the never patterns are properly renamed.
- More than one never pattern can be used in an attack state. Each one must be delimited by its own set of parentheses.
- Substitutions to variables in a state can have unexpected effects on a never pattern, resulting in the discarding of more states than expected. For example, if a never pattern contains $d(K, X)$, and the cancellation rule for encryption/decryption is used, this could be reduced to Y by the substitution $X \mapsto e(K, Y)$. We therefore recommend that never patterns should be *strongly irreducible*, that is, there should be no irreducible substitutions to the variables in a never pattern irreducible For example, the never pattern used for authentication in Section 10.6 satisfies this property. Strong irreducibility of a term is equivalent to its having only one variant, and so it can be checked using Maude's "`get variants`" command (see Section 4.5.6 for information about variants).

6.6 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, together with its grammar-based techniques for reducing the size of the search space as well as other state-space reduction techniques (see [21]). Here we briefly explain how *grammars* work as a state space reduction technique and refer the reader to [35, 16] 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 $\theta(m)$ `!inI`, i.e., the intruder is not able to learn message m. 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. Even when they can't they may reduce the search space enough so finding an attack becomes feasible.

Unlike the grammars used in NPA, described in [35], and the earlier version of Maude-NPA described in [16], in which initial grammars needed to be specified by the user, the current version ofMaude-NPA generates initial grammars automatically. Each initial grammar, with the exception of a special initial grammar that is generated for AC operators, consists of a single seed term of the form $C \mapsto f(X_1, \cdots, X_n) \in \mathcal{L}$, 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 $(X_i$ `inI`$)$ (similar to expression X_i `inI` but used in a different context). However, Maude-NPA provides features to control such automatically generated grammars, e.g., by adding more seed terms. Appendix C gives a more detailed description of grammars and their features in the Maude-NPA.

7 Maude-NPA Commands for Attack Search

The tool provides different commands for searching for attacks: the commands `run`, `digest`, `summary`, `initials`, `debug`, and `ids`. They are invoked by reducing them in Maude, that is, by typing `red` followed by the 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.

7.1 The `run` command

This is the most basic command in Maude-NPA and returns the states at the frontier of the search tree for the specified depth. For example, the command

```
Maude> red 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 and return the leaves of that tree. Note that if a state above depth 10 did not have any children it will also be returned, since it is a leaf of the search tree. For instance, when we give[10] the command `run(0,4)` in Maude as shown below for the NSPK example, it returns:

```
Maude> reduce in MAUDE-NPA : run(0, 4) .
result IdSystemSet: (< 1 . 5 . 2 . 7 . 2 > (
:: nil ::
[ nil |
   -(pk(i, n(b, #0:Fresh))),
   +(n(b, #0:Fresh)), nil]  &
:: nil ::
[ nil |
   -(n(b, #0:Fresh)),
   +(pk(b, n(b, #0:Fresh))), nil]  &
:: #0:Fresh ::
[ nil |
   -(pk(b, a ; n(a, #1:Fresh))),
   +(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
   -(pk(b, n(b, #0:Fresh))), nil]  &
:: #1:Fresh ::
[ nil,
   +(pk(i, a ; n(a, #1:Fresh))) |
   -(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
   +(pk(i, n(b, #0:Fresh))), nil] )
|
pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh)) !inI,
pk(b, n(b, #0:Fresh)) !inI,
pk(i, n(b, #0:Fresh)) !inI,
n(b, #0:Fresh) !inI,
pk(b, a ; n(a, #1:Fresh)) inI
|
-(pk(b, a ; n(a, #1:Fresh))),
+(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
-(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
+(pk(i, n(b, #0:Fresh))),
-(pk(i, n(b, #0:Fresh))),
+(n(b, #0:Fresh)),
-(n(b, #0:Fresh)),
+(pk(b, n(b, #0:Fresh))),
-(pk(b, n(b, #0:Fresh)))
|
nil)
< 1 . 5 . 5 . 2 . 7 > (
:: nil ::
[ nil |
   -(pk(i, n(b, #0:Fresh) ; n(#1:Name, #2:Fresh))),
   +(n(b, #0:Fresh) ; n(#1:Name, #2:Fresh)), nil]  &
:: nil ::
[ nil |
   -(n(b, #0:Fresh)),
   +(pk(b, n(b, #0:Fresh))), nil]  &
:: nil ::
[ nil |
   -(n(b, #0:Fresh) ; n(#1:Name, #2:Fresh)),
   +(n(b, #0:Fresh)), nil]  &
```

[10] Note that after loading Maude-NPA (see Section 3), the active Maude module is the `MAUDE-NPA` module and each command is displayed with the "`in MAUDE-NPA :`" decoration.

```
:: #0:Fresh ::
[ nil,
   -(pk(b, a ; #3:Nonce)),
   +(pk(a, #3:Nonce ; n(b, #0:Fresh))) |
   -(pk(b, n(b, #0:Fresh))), nil]  &
:: #2:Fresh ::
[ nil |
   -(pk(#1:Name, i ; n(b, #0:Fresh))),
   +(pk(i, n(b, #0:Fresh) ; n(#1:Name, #2:Fresh))), nil] )
|
pk(b, n(b, #0:Fresh)) !inI,
pk(i, n(b, #0:Fresh) ; n(#1:Name, #2:Fresh)) !inI,
n(b, #0:Fresh) !inI,
(n(b, #0:Fresh) ; n(#1:Name, #2:Fresh)) !inI,
pk(#1:Name, i ; n(b, #0:Fresh)) inI
|
-(pk(#1:Name, i ; n(b, #0:Fresh))),
+(pk(i, n(b, #0:Fresh) ; n(#1:Name, #2:Fresh))),
-(pk(i, n(b, #0:Fresh) ; n(#1:Name, #2:Fresh))),
+(n(b, #0:Fresh) ; n(#1:Name, #2:Fresh)),
-(n(b, #0:Fresh) ; n(#1:Name, #2:Fresh)),
+(n(b, #0:Fresh)),
-(n(b, #0:Fresh)),
+(pk(b, n(b, #0:Fresh))),
-(pk(b, n(b, #0:Fresh)))
|
nil
```

7.2 The `digest` command

When the user is not interested in all the information contained in states but only in the *active* information, the alternative `digest` command can be used. This command omits the strands and the negative facts in the intruder knowledge. For instance, the reader can check the differences between the previous command `run(0,4)` and the following command `digest(0,4)`:

```
Maude> reduce in MAUDE-NPA : digest(0, 4) .
result IdSystemSet: (< 1 . 5 . 2 . 7 . 2 >
pk(b, a ; n(a, #1:Fresh)) inI
|
-(pk(b, a ; n(a, #1:Fresh))),
+(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
-(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
+(pk(i, n(b, #0:Fresh))),
-(pk(i, n(b, #0:Fresh))),
+(n(b, #0:Fresh)),
-(n(b, #0:Fresh)),
+(pk(b, n(b, #0:Fresh))),
-(pk(b, n(b, #0:Fresh)))
)
< 1 . 5 . 5 . 2 . 7 >
pk(#1:Name, i ; n(b, #0:Fresh)) inI
|
-(pk(#1:Name, i ; n(b, #0:Fresh))),
+(pk(i, n(b, #0:Fresh) ; n(#1:Name, #2:Fresh))),
-(pk(i, n(b, #0:Fresh) ; n(#1:Name, #2:Fresh))),
+(n(b, #0:Fresh) ; n(#1:Name, #2:Fresh)),
-(n(b, #0:Fresh) ; n(#1:Name, #2:Fresh)),
+(n(b, #0:Fresh)),
```

```
-(n(b, #0:Fresh)),
+(pk(b, n(b, #0:Fresh))),
-(pk(b, n(b, #0:Fresh)))
```

7.3 The `ids` command

If the information returned by the commands `run` or `digest` is not relevant, the user can simply show the identifiers of all the states found in the leaves of the reachability tree using the command `ids`. For instance, when we give the reduce command `ids(0,4)` in Maude as below for the NSPK example, it returns:

```
Maude> reduce in MAUDE-NPA : ids(0, 4) .
result IdSet: (1 . 5 . 2 . 7 . 2) : (1 . 5 . 5 . 2 . 7)
```

7.4 The `summary` command

When the user is not interested in the current states of the reachability tree, he/she can use the command `summary`, which outputs just the number of states found in the leaves of the reachability tree and how many of those are initial states, i.e., solutions to the attack. For instance, the sequence of summary commands associated to the NSPK example is as follows:

```
Maude> reduce in MAUDE-NPA : summary(0,1) .
result Summary: States>> 4 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(0,2) .
result Summary: States>> 6 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(0,3) .
result Summary: States>> 4 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(0,4) .
result Summary: States>> 2 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(0,5) .
result Summary: States>> 1 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(0,6) .
result Summary: States>> 2 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(0,7) .
result Summary: States>> 4 Solutions>> 1
Maude> reduce in MAUDE-NPA : summary(0,8) .
result Summary: States>> 4 Solutions>> 1
Maude> reduce in MAUDE-NPA : summary(0,9) .
result Summary: States>> 2 Solutions>> 1
Maude> reduce in MAUDE-NPA : summary(0,10) .
result Summary: States>> 1 Solutions>> 1
```

7.5 The `initials` command

We also provide a slightly different version of the `run` command that outputs only the initial states, instead of all the leaves. Thus, if we type

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

for the NSPK example our tool outputs the attack:

```
Maude> reduce in MAUDE-NPA : initials(0,7) .
result ShortIdSystem: < 1 . 5 . 2 . 7 . 2 . 4 . 2 . 1 > (
:: nil ::
[ nil |
   -(pk(i, n(b, #0:Fresh))),
   +(n(b, #0:Fresh)), nil]  &
:: nil ::
[ nil |
   -(pk(i, a ; n(a, #1:Fresh))),
   +(a ; n(a, #1:Fresh)), nil]  &
:: nil ::
[ nil |
   -(n(b, #0:Fresh)),
   +(pk(b, n(b, #0:Fresh))), nil]  &
:: nil ::
[ nil |
   -(a ; n(a, #1:Fresh)),
   +(pk(b, a ; n(a, #1:Fresh))), nil]  &
:: #0:Fresh ::
[ nil |
   -(pk(b, a ; n(a, #1:Fresh))),
   +(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
   -(pk(b, n(b, #0:Fresh))), nil]  &
:: #1:Fresh ::
[ nil |
   +(pk(i, a ; n(a, #1:Fresh))),
   -(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
   +(pk(i, n(b, #0:Fresh))), nil] )
|
pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh)) !inI,
pk(b, n(b, #0:Fresh)) !inI,
pk(b, a ; n(a, #1:Fresh)) !inI,
pk(i, n(b, #0:Fresh)) !inI,
pk(i, a ; n(a, #1:Fresh)) !inI,
n(b, #0:Fresh) !inI,
(a ; n(a, #1:Fresh)) !inI
|
+(pk(i, a ; n(a, #1:Fresh))),
-(pk(i, a ; n(a, #1:Fresh))),
+(a ; n(a, #1:Fresh)),
-(a ; n(a, #1:Fresh)),
+(pk(b, a ; n(a, #1:Fresh))),
-(pk(b, a ; n(a, #1:Fresh))),
+(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
-(pk(a, n(a, #1:Fresh) ; n(b, #0:Fresh))),
+(pk(i, n(b, #0:Fresh))),
-(pk(i, n(b, #0:Fresh))),
+(n(b, #0:Fresh)),
-(n(b, #0:Fresh)),
+(pk(b, n(b, #0:Fresh))),
-(pk(b, n(b, #0: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. $I \rightarrow A : pk(A, N_A; N_B)$
5. $A \rightarrow I : pk(I, N_B)$
6. $I_A \rightarrow B : pk(B, N_B)$

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 feature with NSPK:

```
Maude> 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 is like, we can then give the command "`red run(0,unbounded) .`" to get the initial state displayed above.

7.6 The `debug` command

States in Maude-NPA carry extra internal information that can be displayed using the `debug` command. For example, the never patterns associated to each state or internal information on different optimizations.

We do not show any output of this command, since the outputs are very large and the extra information is only useful for very advanced users.

7.7 Guiding the search

Another interesting feature that has been added to the backwards reachability analysis is the possibility of limiting the search space by search for the descendants of a given state identifier. All the previous commands `run`, `digest`, `summary`, `initials`, `debug`, and `ids` allow for a specific state identifier restricting the search.

For example, imagine that the initial state associated to the NSPK protocol has identifier "`1 . 5 . 2 . 7 . 2 . 4 . 2 . 1`". Then we could repeat the analysis with just the restriction of that state to see whether the state is still reachable or not after some modifications in the specification file.

```
Maude> reduce in MAUDE-NPA : summary[1 . 5 . 2 . 7 . 2 . 4 . 2 . 1](0,2) .
result Summary: States>> 1 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary[1 . 5 . 2 . 7 . 2 . 4 . 2 . 1](0,1) .
result Summary: States>> 1 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary[1 . 5 . 2 . 7 . 2 . 4 . 2 . 1](0,2) .
result Summary: States>> 1 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary[1 . 5 . 2 . 7 . 2 . 4 . 2 . 1](0,3) .
result Summary: States>> 1 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary[1 . 5 . 2 . 7 . 2 . 4 . 2 . 1](0,4) .
result Summary: States>> 1 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary[1 . 5 . 2 . 7 . 2 . 4 . 2 . 1](0,5) .
result Summary: States>> 1 Solutions>> 0
```

```
Maude> reduce in MAUDE-NPA : summary[1 . 5 . 2 . 7 . 2 . 4 . 2 . 1](0,6) .
result Summary: States>> 1 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary[1 . 5 . 2 . 7 . 2 . 4 . 2 . 1](0,7) .
result Summary: States>> 1 Solutions>> 1
```

7.8 General Comments on Search Commands

We can highlight some comments on the different search commands:

1. Use the `summary` command to detect whether the protocol specification is correct or not. One can specify an "attack state" that describes a correct execution of the protocol. If an initial state is not found, then the protocol specification may be incorrect. This procedure may be helped by commenting out the Dolev-Yao strands, which are not needed to find a correct execution. Note, however, that the Maude-NPA expects at least one Dolev-Yao strand to be present.

2. If, after using the `summary` commands, the search space is increasing level by level, then scrutinize the generated states to see whether it is exhibiting the expected behavior. Different levels of information are given by the `digest`, `run`, or `debug` commands.

3. If the search space associated to the protocol and the attack pattern is too big, the user may be interested in a more specific traversal of the search space using the guiding facilities.

4. It is also possible to use some scripting language (e.g. bash scripting facilities) to generate as many levels of the search tree as possible and analyze them later. For instance, the following bash script file incrementally generates the search space up to level 7, which is where the initial state is found, including different levels of details in the output file by using the `summary`, `digest`, `run`, and `initials` commands.

```
#!/bin/bash
maude271 <<EOF
load maude-npa.maude
load examples/Needham_Schroeder.maude
red summary(0,1) .
red digest(0,1) .
red run(0,1) .
red summary(0,2) .
red digest(0,2) .
red run(0,2) .
red summary(0,3) .
red digest(0,3) .
red run(0,3) .
red summary(0,4) .
red digest(0,4) .
```

```
red run(0,4) .
red summary(0,5) .
red digest(0,5) .
red run(0,5) .
red summary(0,6) .
red digest(0,6) .
red run(0,6) .
red summary(0,7) .
red digest(0,7) .
red run(0,7) .
red initials(0,7) .
EOF
```

8 ∗Protocol Composition

Protocols do not work alone, but together, one protocol relying on another to provide needed services. Many security problems in cryptographic protocols arise when such composition is done incorrectly or is not well understood. Maude-NPA allows the specification and analysis of protocol compositions through a specific syntax and semantics, which is discussed in this section. This new syntax is only necessary for protocol composition and does not modify any of the syntax described before.

Each protocol strand that will be used in a composition is given a *role identifier*, which is a constant uniquely identifying that protocol strand. For example, in the NSPK protocol, we could have two role identifiers, `init` and `resp`, for initiator and responder. We have added a new sort Role that the role identifiers must belong to.

We then extend each strand with *input* and *output synchronization* information. That is, a strand can have an input synchronization of the form

$$\{(a_1 a_2 \cdots a_i) \rightarrow b \ ;; \ Mode \ ;; \ Msg\}.$$

and an output synchronization of the form

$$\{a \rightarrow (b_1 b_2 \cdots b_j) \ ;; \ Mode \ ;; \ Msg\}$$

Note that expressions of the form $(a_1 a_2 \cdots a_i) \rightarrow b$ or $a \rightarrow (b_1 b_2 \cdots b_j)$ cannot contain variables.

For the input synchronization, the first field gives the identifiers of the strands $(a_1 a_2 \cdots a_i)$ that can be parents of the strand b, which is the identifier of the strand in which the input synchronization appears. Likewise, for the output synchronization, the first field gives the identifiers of the strands $(b_1 b_2 \cdots b_j)$ that can be children of the strand a.

The second field, or `Mode` field, gives the types of synchronization a strand role may be involved in. There are two possible modes: `1-1` for one-to-one compositions, and `1-*` for one-to-many compositions. A role is of mode 1-1 if each instantiation of a role can have only one child. A role is of mode 1-* if each instantiation of a role may have an unlimited number of children. For example, a role used in setting

up a shared master key can be the parent of many instantiations of roles used for setting up session keys (see an example in Section 8.2).

Finally the last field, or `Msg` field, is used to transfer information from the parent strand to the child strand. For example, a master key generation strand might have variables standing for the names of the initiator A and responder B and the master key MK that they share.

Two strands may be composed if the following conditions hold:

1. The output synchronization field of the parent is $a \rightarrow (b_1 \cdots b_{i-1}\ b\ b_{i+1} \cdots b_n)$ and the input synchronization of the child is $(a_1 \cdots a_{j-1}\ a\ a_{j+1} \cdots a_m) \rightarrow b$;

2. The `Mode` fields of the output synchronization of the parent and the input synchronization of the child are identical, and;

3. The `Msg` fields of the output synchronization of the parent and the input synchronization of the child are unifiable.

After extending strands with *input* and *output synchronization* information, a strand is now a term of one of the following forms:

1. $[nil, \overrightarrow{M}, nil]$, i.e. a standard strand that cannot be connected to either a parent or a child strand,

2. $[I_M, \overrightarrow{M}, nil]$, i.e. a *child strand* that can be connected to a parent strand,

3. $[nil, \overrightarrow{M}, O_M]$, i.e. a *parent strand* that can be connected to a child strand,

4. $[I_M, \overrightarrow{M}, O_M]$, i.e. a strand that can be connected to both a parent and a child strand, or

5. $[I_M, O_M]$, i.e. a strand that can be connected to both a parent and a child strand, but without sending or receiving any message, called a *void strand*.

Note that an attack pattern should also include input and output synchronization messages but only up to the specific attack. For example, it is an error to include a strand of the form `[-(m1), +(m2), {role1 -> role2 ;; 1-1 ;; m3} | nil]` in an attack pattern if there is no other strand starting with the same input synchronization message, since there is no way to synchronize this output message. In this case, we should write `[ -(m1), +(m2) | nil]`, leaving the output synchronization message out. Similarly, it is an error to include an input synchronization message in a strand if we are not planning to synchronize it with another (existing or new) strand.

Let us illustrate with examples the syntax and semantics for protocol composition.

8.1 NSL Distance Bounding Protocol (One-to-one composition)

In this example of a one-parent, one-child protocol composition, appeared in [27], the participants first use NSL to agree on a secret nonce.

1. $A \to B : \{N_A, A\}_{pub(B)}$
2. $B \to A : \{N_A, N_B, B\}_{pub(A)}$
3. $A \to B : \{N_B\}_{pub(B)}$

where $\{M\}_{pub(A)}$ means message M encrypted using the public key of principal with name A, N_A and N_B are nonces generated by the respective principals, and we use the comma as message concatenation.

The agreed nonce N_A is then used in the distance bounding protocol described below. The idea behind the protocol is that Bob uses the round trip time of a challenge-response protocol with Alice to compute an upper bound on her distance from him according to the following protocol:

4. $B \to A : N'_B$
   Bob records the time at which he sent N'_B
5. $A \to B : N_A \oplus N'_B$
   Bob records the time he receives the response and checks the equivalence $N_A = N_A \oplus N'_B \oplus N'_B$. If this holds, he uses the round-trip time of his challenge and response to estimate his distance from Alice

where $\oplus$ is the exclusive-or operator satisfying associativity (i.e., $X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$) and commutativity (i.e., $X \oplus Y = Y \oplus X$) plus the self-cancellation property $X \oplus X = 0$ and the identity property $X \oplus 0 = X$. Note that Bob is the initiator and Alice is the responder of the distance bounding protocol, in contrast to the NSL protocol.

The distance bounding example is a case of a one parent, one child protocol composition. Each instance of the parent NSL protocol can have only one child distance bounding protocol, since the distance bounding protocol depends upon the assumption that N_A is known only by A and B. But since the distance bounding protocol reveals N_A, it cannot be used with the same N_A more than once.

How an earlier version Maude-NPA found this attack is described in [20]:

a) Intruder I runs an instance of NSL with Alice as the initiator and I as the responder, obtaining a nonce N_A.

b) I then runs an instance of NSL with Bob with I as the initiator and Bob as the responder, using N_A as the initiator nonce.

c) $B \to I : N'_B$ where I does not respond, but Alice, seeing this, thinks it is for her.

d) $A \rightarrow I : N'_B \oplus N_A$ where Bob, seeing this thinks this is I's response.

If Alice is closer to Bob than I is, then I can use this attack to appear closer to Bob than he is. This attack is a textbook example of a composition failure. NSL has all the properties of a good key distribution protocol, but fails to provide all the guarantees that are needed by the distance bounding protocol.

The specification of the protocol strands using the protocol composition syntax described above is as follows where the symbol _*_ denotes the exclusive-or operator:

```
eq STRANDS-PROTOCOL
 = :: r :: --- NSL-Alice
   [ nil | +(pk(B, n(A,r) ; A)) ,
           -(pk(A, n(A,r) ; NB ; B )),
           +(pk(B, NB)),
           {init-nsl -> resp-db ;; 1-1 ;;  (A ; B ; n(A,r))}, nil ] &
   :: r :: --- NSL-Bob
   [ nil | -(pk(B,NA ; A)),
           +(pk(A, NA ; n(B,r) ; B)),
           -(pk(B,n(B,r))),
           {resp-nsl -> init-db ;; 1-1 ;; (A ; B ; NA)}, nil ] &
   :: r' :: --- Init-DB
   [ nil | {resp-nsl -> init-db ;; 1-1 ;; (A ; B ; NA)},
           +(n(B,r')),
           -(NA * n(B,r')), nil] &
   :: nil :: ---Resp-DB
   [ nil | {init-nsl -> resp-db ;; 1-1 ;; (A ; B ; NA) },
           -(N),
           +(NA * N), nil ]
[nonexec] .
```

The full specification of this protocol with attack states is provided in Section 10.7. This is one of the cases in which the current configuration of Maude-NPA does not perform as well as some of the previous ones that were used to analyze this protocol. In order to make the analysis feasible and still exhibit the attack we leave out the intruder strand for computing exclusive-or, which is not needed to execute the attack.

8.2 NSL Key Distribution Protocol (One-to-many composition)

Our next example is a one parent, many children protocol composition, also using NSL. This type of composition arises, for example, in key distribution protocols in which the parent protocol is used to generate a master key, and the child protocol is used to generate a session key. In this case, one wants to be able to run an arbitrary number of instances of the child protocol with the same master key.

In the distance bounding example the initiator of the distance bounding protocol was always the child of the responder of the NSL protocol and vice versa. In the key distribution example, the initiator of the session key protocol can be the child of

either the initiator or the responder of the NSL protocol. So, we have two possible child executions after NSL:

4. $A \to B : \{Sk_A\}_{h(N_A,N_B)}$	4. $B \to A : \{Sk_B\}_{h(N_A,N_B)}$
5. $B \to A : \{Sk_A; N'_B\}_{h(N_A,N_B)}$	5. $A \to B : \{Sk_B; N'_A\}_{h(N_A,N_B)}$
6. $A \to B : \{N'_B\}_{h(N_A,N_B)}$	6. $B \to A : \{N'_A\}_{h(N_A,N_B)}$

where Sk_A is the session key generated by principal A and h is again a collision-resistant hash function. This protocol is proved secure by our tool.

The specification of the strands of the NSL-KD protocol using the syntax for protocol composition via synchronization messages is as follows:

```
eq STRANDS-PROTOCOL
   --- NSL protocol
   :: r ::
   [ nil | +(pk(B, n(A,r) ; A)),
           -(pk(A, n(A,r) ; NB ; B )),
           +(pk(B, NB)),
           {init-nsl -> init-kd resp-kd ;; 1-* ;;
               A ; B ; h(n(A,r) , NB) }, nil ] &
   :: r ::
   [ nil | -(pk(B,NA ; A)),
           +(pk(A, NA ; n(B,r) ; B)),
           -(pk(B,n(B,r))),
           {resp-nsl -> init-kd resp-kd ;; 1-* ;;
              B ; A ; h(NA , n(B,r))}, nil ] &
   ---- KD protocol
   :: r' ::
   [ nil | { init-nsl resp-nsl -> init-kd ;; 1-* ;; C ; D ; MKe },
           +(e(MKe, skey(C, r'))),
           -(e(MKe, skey(C, r') ; N)),
           +(e(MKe, N)), nil] &
   :: r' ::
   [ nil | { init-nsl resp-nsl -> resp-kd ;; 1-* ;; C ; D ; MKe },
           -(e(MKe, K)),
           +(e(MKe, K ; n(C,r'))),
           -(e(MKe, n(C,r'))), nil ]
[nonexec] .
```

The full specification of this protocol with attack states is provided in Section 10.8.

9 **Process Algebra and Choice

Honest participants in protocol specifications do not always have a linear execution. In general their execution may include choice points causing the protocol to continue

along different paths. In order to support the specification of protocols having roles that may exhibit non-linear, branching behavior, Maude-NPA has been recently extended with a process algebra notation that is more expressive than the strand notation. Strands represent each role behavior as a *linear* sequence of message outputs and inputs but processes represent each role behavior as a possibly *non-linear* sequence of message outputs and inputs. In this section we briefly explain this new syntax and semantics.

This is work in progress, and so currently the process algebra version of Maude-NPA uses only a mixture of process algebra and strand space syntax. The honest principal specification is specified in the process algebra syntax. The intruder capabilities as well as the states generated by the tool still use the strand syntax. Attack patterns without never patterns may be specified using the process algebra or strand syntax, while attack patterns with never patterns currently can only be specified using the strand syntax.

This may seem confusing, but from Maude-NPA's point of view it is not. The process algebra semantics has been proved sound and complete with respect to the strand space semantics in [43], and Maude-NPA translates all process algebra specifications to strands. However, we strongly recommend that users specify any protocols involving explicit choice in the process algebra notation; although it is theoretically possible to specify these protocols in the strand notation, it is fact quite cumbersome and can easily lead to error. Strands used in attack patterns, however, can be found using the command `red new-strands?` once the specification has been loaded into Maude-NPA.

Through this process algebra notation, the Maude-NPA tool now supports a taxonomy of choices in which the categories of deterministic and non-deterministic choice are further subdivided. First of all, non-deterministic choice is subdivided into *explicit* and *implicit* non-deterministic choice. In explicit non-deterministic choice a role chooses either one branch or another at a choice point non-deterministically. In implicit non-deterministic choice a logical *choice variable* is introduced which may be non-deterministically instantiated by the role. Deterministic choice is subdivided into (explicit) *if-then-else* choice and *implicit deterministic choice*. In if-then-else choice a predicate is evaluated. If the predicate evaluates to true, then one branch is chosen, and if it evaluates to false, then the other branch is chosen. Deterministic choice with more than two choices can be modeled by nested of if-then-else choices. In implicit deterministic choice, a term pattern is used as an implicit guard, so that only messages matching such a pattern can be chosen i.e., accepted, by the role. Although implicit deterministic choice can be viewed a special case of if-then-else choice in which the second branch is empty, it is often simpler to treat it separately. Classifying choice in this way allows us to represent possible behaviors of a protocol by translating its process algebra description into a semantically equivalent finite set of strands modeling possible executions, while still allowing the variables used in implicit non-deterministic and deterministic choice to be instantiated in a possibly infinite number of ways.

In the *protocol process algebra* the behavior of both honest principals and the

intruder is represented by *labeled processes*. Therefore, a protocol is specified as a set of labeled processes. Each process performs a sequence of actions, namely, sending or receiving a message, and may perform deterministic or non-deterministic choices. The protocol process algebra's syntax is parameterized by a sort Msg of messages and has the following syntax:

$$\begin{aligned} \mathit{ProcConf} &::= \mathit{Proc} \mid \mathit{ProcConf}\ \&\ \mathit{ProcConf} \mid \emptyset \\ \mathit{Proc} &::= \mathit{nilP} \mid +\mathit{Msg} \mid -\mathit{Msg} \mid \mathit{Proc} \cdot \mathit{Proc} \mid \\ &\quad\ \mathit{Proc}\ ?\ \mathit{Proc} \mid \mathit{if}\ \mathit{Cond}\ \mathit{then}\ \mathit{Proc}\ \mathit{else}\ \mathit{Proc} \\ \mathit{Cond} &::= \mathit{Msg}\ \texttt{eq}\ \mathit{Msg} \mid \mathit{Msg}\ \texttt{neq}\ \mathit{Msg} \end{aligned}$$

- *ProcConf* stands for a *process configuration*, that is, a set of labeled processes. The symbol & is used to denote set union for sets of labeled processes. It is associative-commutative with $\emptyset$ as its identity element.

- *Proc* defines the actions that can be executed within a process. $+Msg$, and $-Msg$ respectively denote sending out or receiving a message *Msg*. We assume a single channel, through which all messages are sent or received by the intruder. "$Proc \cdot Proc$" denotes sequential composition of processes, where symbol `_._` is associative and has the empty process *nil* as identity. "*Proc* ? *Proc*" denotes an explicit nondeterministic choice, whereas "*if Cond then Proc else Proc*" denotes an explicit deterministic choice, whose continuation depends on the satisfaction of the constraint *Cond*.

- *Cond* denotes a constraint that will be evaluated in explicit deterministic choices. In this work we only consider constraints that are either equalities ($=$) or disequalities ($\neq$) between message expressions.

In order for Maude-NPA to accept process specifications, we have replaced the section `STRANDS-PROTOCOL` from the protocol template of Figure 2 by a new section `PROCESSES-PROTOCOL`. Figure 4 shows the new template for a protocol specification using the process algebra notation.

As a part of our work in progress, we are in the process of translating attack states into the process algebra notation. Currently, attack patterns without never patterns can be specified in either the process algebra notation or the strand notation. To use never patterns, you must specify attack patterns using the strand space notation. We describe how to specify both kinds of attack patterns below.

Attack patterns specified using the process algebra notation, appear under the label `ATTACK-PROCESS` instead of `ATTACK-STATE`. We have simplified the notation, removing two parts of an attack pattern that usually had the value `nil`. Note that an attack pattern cannot contain explicit nondeterminism (?) or explicit deterministic choice (`if`), since one and only one behavior is provided in an attack pattern. That is, imagine a process in the process specification of the form

$$-(m1)\ .\ +(m2)\ .\ \mathit{if}\ \mathit{exp}1\ =\ \mathit{exp}2\ \mathit{then}\ +(m3)\ \mathit{else}\ +(m4)$$

Then in the attack pattern one must specify

$$-(m1)\ .\ +(m2)\ .\ exp1 = exp2\ .\ +(m3)$$

or

$$-(m1)\ .\ +(m2)\ .\ exp1 \neq exp2\ .\ +(m4)$$

Examples of attack patterns in the process algebra notation appear in Sections 9.1 and 9.2.

Attack patterns in the strand space notation are labeled as `ATTACK-STATE` and specified using the strands translations of the process algebra specifications produced by Maude-NPA. These strands may be obtained by using the command `red new-strands?`. When using the strand space notation, one does not have to remove explicit choice, since this is already done by the translation from process algebras to strands. However, some work still has to be done for never patterns. As a result of the strong irreducibility requirement on never patterns (see Section 6.5), any expression in a condition must be replaced by a possible result of evaluating it. For example, consider again the process

$$-(m1)\ .\ +(m2)\ .\ if\ exp1\ =\ exp2\ then\ +(m3)\ else\ +(m4)$$

.

This will be translated into the two strands

$$::R::[nil, -(m1)\ ,\ +(m2),\ exp1\ =\ exp2,\ +(m3), nil]$$
$$::R::[nil, -(m1)\ ,\ +(m2),\ exp1\ \neq\ exp2,\ +(m4), nil]$$

If the expression $exp1$ above may be reduced one of two constants `yes` and `no` and $exp2$ to the constant `yes`, when we include any of the two previous strands in a never pattern, we need to write

$$::R::[\texttt{nil}, -(m1)\ ,\ +(m2),\ \texttt{yes} = \texttt{yes},\ +(m3), \texttt{nil}]$$
$$::R::[\texttt{nil}, -(m1)\ ,\ +(m2), \texttt{no} \neq \texttt{yes},\ +(m4), \texttt{nil}]$$

with appropriate substitutions to the variables of $m1$ and $m2$ if they share variables with $exp1$ and $exp2$.

See Section 9.2 for an example of an attack state with never patterns.

WARNING: Attack states and attack processes may appear in the same specification, but they should *never* be assigned the same index number. Maude-NPA translates both into the same the strand space notation, and after that it will not be able to tell which of the two attack patterns you intend to use.

In all process specifications we assume three disjoint kinds of variables:

- ***fresh variables***: these are not really variables in the standard sense, but *names* for *constant values* in a data type of unguessable values such as nonces. Throughout this manual we denote this kind of variables as $r, r_1, r_2, \ldots$.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  --- Importing sorts Msg, Fresh, and Public
  protecting DEFINITION-PROTOCOL-RULES .
  ---------------------------------------------------------------
  --- Overwrite this module with the syntax of your protocol
  --- Notes:
  --- * Sorts Msg and Fresh are special and imported
  --- * New sorts can be subsorts of Msg
  --- * No sort can be a supersort of Msg
  --- * Variables of sort Fresh denote uniquely generated data
  --- * 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 variant equations eq Lhs = Rhs [variant] .)
  --- * Or use equations for dedicated unification algorithm
  --- * Attribute owise cannot be used
  ---------------------------------------------------------------
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 strands here. Strands are properly renamed
  [nonexec] .
  eq PROCESSES-PROTOCOL
   = --- Add protocol specification here using process algebra
  [nonexec] .
  eq ATTACK-PROCESS(0)
   = --- Add attack state here using process algebra
     --- More than one attack state can be specified, but each must be
     --- identified by a natural number
  [nonexec] .
endfm

--- THIS HAS TO BE THE LAST ACTION !!!!
select MAUDE-NPA .
```

Figure 4: Maude-NPA protocol template using process algebra

- ***choice variables***: variables first appearing in a *sent message* $+M$, which can be instantiated to any value arbitrarily chosen from a possibly infinite domain. A choice variable indicates an *implicit non-deterministic choice*. Given a protocol with choice variables, each possible substitution of these variables denotes a possible continuation of the protocol. This kind of variables will be written as regular variables but we append the symbol "?" after the variable name for readability purposes, e.g. $X_?$, $Y_?$, ...
- ***pattern variables***: variables first appearing in a *received message* $-M$. These variables will be instantiated when matching sent and received messages. *Implicit deterministic choices* are indicated by pattern variables, since failing to match the pattern may lead to the rejection of a message. The pattern plays the implicit role of a guard, so that, depending on the different ways of matching, the protocol can have different continuations. These variables will be written as regular variables, e.g. $A, B, N_A, \ldots$.

Note that fresh variables are distinguished from other variables by having a specific sort Fresh. Choice variables or pattern variables can never have sort Fresh.

9.1 Encryption Mode

Consider the following protocol examples, which exhibits all four types of choice, implicit non-determinism, explicit non-determinism, implicit determinism, and explicit determinism. The informal representation of the protocol is as follows:

$$
\begin{array}{ll}
(\mathit{Init}) & ((+(A_?\,;\,B_?\,;\,\mathit{pub})\cdot -(\mathit{pk}(A_?,B_?\,;\,\mathit{SK})) \\
& \quad ? \\
& \quad (+(A_?\,;\,B_?\,;\,\mathit{SharedKey})\cdot -(e(\mathit{key}(A_?,B_?),B_?\,;\,\mathit{SK})) \\
(\mathit{Resp}) & -(A\,;\,B\,;\,\mathit{TEnc})\cdot \\
& \quad \mathit{if}\ \mathit{TEnc}=\mathit{pub} \\
& \quad\quad \mathit{then}\ (+(\mathit{pk}(A,B\,;\,\mathit{skey}(A,B,r')))) \\
& \quad\quad \mathit{else}\ (+(e(\mathit{key}(A,B),B\,;\,\mathit{skey}(A,B,r'))))
\end{array}
$$

In the initiator role the principal names are chosen using implicit nondeterministic choice. This is represented by choice variables of the form $X_?$. The initiator role then uses the explicit nondeterministic choice operator ? to determine whether or not to initiate a public or shared key version of the protocol. The responder role in turn uses implicit deterministic choice to determine whether to proceed after receiving the first message, proceeding only if that message satisfies the pattern specified by $A\,;\,B\,;\,\mathit{TEnc}$, where A, B, and TEnc are *pattern variables*. It then uses if-then-else deterministic choice to decide whether to execute the public or shared key version of the protocol, depending on TEnc.

```
eq PROCESSES-PROTOCOL
 = (
    ( +(A ; B ;  pubkey) .
```

```
        -(pk(A, B ; SK)) .
        +(pk(B, A ; SK ; n(A,r))) .
        -(pk(A, B ; n(A,r)))
      )
      ?
      ( +(A ; B ;  shkey) .
        -(she(key(A, B), SK )) .
        +(she(key(A, B), SK ; n(A,r))) .
        -(she(key(A,B), n(A,r))) )
      )
     &
     ( -(A ; B ; mode) .
       (if (mode eq pubkey)
        then ( +(pk(A, B ; skey(B, r))) .
               -(pk(B, A ; skey(B,r) ; N)) .
               +(pk(A, B ; N)) )
        else ( +(she(key(A, B), skey(B,r))) .
               -(she(key(A, B), skey(B,r) ; N)) .
               +(she(key(A,B), N)) )
       )
     )
[nonexec] .
```

The full specification of this protocol with attack states is provided in Section 10.9.

9.2 Rock-Paper-Scissors

A very simple protocol that involves several choices is the famous Rock-Paper-Scissors game, in which Alice and Bob are the two players of the game. In this game, Alice and Bob commit to each other their hand shapes, which are later on revealed to each other after both players have committed them. The result of the game is then agreed upon between the two players according to the rule: rock beats (blunts) scissors, scissors beat (cuts) paper, and paper beats (wraps) rock. They finish by verifying with each other that they both reached the same conclusion. Thus, at the end of the protocol each party should know the outcome of the game and whether or not the other party agrees to the outcome. This protocol exhibits *explicit deterministic choice*, because the result of the game depends on the evaluation of the committed hand shapes according to the game rules. Note that this protocol also exhibits *implicit nondeterministic choice*, since the hand shapes of the players are chosen by the players themselves during the game.

The protocol proceeds as follows. First, both initiator and responder choose their hand shapes and send them to each other using a secure commitment scheme. Next, they both send each other the nonces that are necessary to open the commitments. Each of them then compares the two hand shapes and decides if the initiator wins, the responder wins, or there is a tie. The initiator then sends the responder the outcome. When the responder receives the initiator's verdict, it compares it against

its own. It responds with "finished" if it agrees with the initiator and "cheater" if it doesn't. All messages are signed and encrypted, and the initiator's and responder's nonces are included in the messages concerning the outcome of the game. The specification of the protocol is given below.

```
eq PROCESSES-PROTOCOL =
  --- initiator
  +(pk(B, sig(A, com(n(A,r), XA)))) .
  -(pk(A, sig(B, ComXB))) .
  +(pk(B, sig(A, n(A , r)))) .
  -(pk(A, sig(B, NB))) .
  (if ((item? open(NB, ComXB)) eq  ok)
   then if ((XA beats open(NB, ComXB)) eq ok)
        then +(pk(B, sig(A, n(A, r)  ; win)))
        else if ((open(NB, ComXB) beats XA) eq ok)
             then +(pk(B, sig(A, n(A, r) ; lose )))
             else +(pk(B, sig(A, n(A, r) ; tie)))
   else nilP ) .
  -(pk(A, sig(B, n(A,r) ; NB)) ; S:Status)
  &
  --- responder
  -(pk(B, sig(A, ComXA))) .
  +(pk(A, sig(B, com(n(B,r), XB)))) .
  -(pk(B, sig(A, NA))) .
  +(pk(A, sig(B, n(B, r)))) .
  -(pk(B, sig(A, NA ; R))) .
  (if ((item? open(NA, ComXA)) eq ok)
   then if (R eq win)
        then if ((open(NA, ComXA) beats XB) eq ok)
             then +(pk(A, sig(B, NA ; n(B,r))) ; finished)
             else +(pk(A, sig(B, NA ; n(B,r))) ; cheater)
        else if (R eq lose)
                then if ((XB beats open(NA, ComXA)) eq ok)
                     then +(pk(A, sig(B, NA ; n(B,r))) ; finished)
                     else +(pk(A, sig(B, NA ; n(B,r))) ; cheater)
                else if (R eq tie)
                     then if (XB eq open(NA, ComXA))
                          then +(pk(A, sig(B, NA ; n(B,r))) ; finished)
                          else +(pk(A, sig(B, NA ; n(B,r))) ; cheater)
                     else nilP
    else nilP )
 [nonexec] .
```

One interesting feature of the Rock-Scissors-Paper protocol, is that, in order to verify that the commitment has been opened successfully (that is, that the nonce received is the nonce used to create the commitment) one must verify that the result of opening it is well-formed; that is, that it is equal to "rock", "scissors", or "paper". This can be done via the evaluation of predicates. First, we create a sort Item and declare the constants "rock", "scissors", and "paper" to be of sort Item. Then we

create a variable `X:Item` of sort Item. We then define a predicate `item?` such that `item? X:Item` evaluates to true. Since only terms of sort Item can be unified with `X:Item`, this predicate can be used to check whether or not a term is of sort Item.

The full specification of this protocol with attack states is provided in Section 10.10.

10 Examples

In the following, we describe how the Maude-NPA analyzes several examples whose algebraic properties have already been defined above.

10.1 Needham-Schroeder Public Key

This protocol has been used as a running example through the manual. We recall the informal specification of NSL, as follows:

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

The protocol is specified in Maude-NPA 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] .
```

The attack pattern representing the intruder learning the responder's nonce is 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
     ||
```

```
    nil
  [nonexec] .
```

And the search space associated to this attack pattern is as follows:

```
Maude> reduce in MAUDE-NPA : summary(1) .
result Summary: States>> 4 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(2) .
result Summary: States>> 6 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(3) .
result Summary: States>> 4 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(4) .
result Summary: States>> 2 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(5) .
result Summary: States>> 1 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(6) .
result Summary: States>> 2 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(7) .
result Summary: States>> 4 Solutions>> 1
Maude> reduce in MAUDE-NPA : summary(8) .
result Summary: States>> 4 Solutions>> 1
Maude> reduce in MAUDE-NPA : summary(9) .
result Summary: States>> 2 Solutions>> 1
Maude> reduce in MAUDE-NPA : summary(10) .
result Summary: States>> 1 Solutions>> 1
```

The well-known man-in-the-middle attack is found, see Section 7.5.

10.2 Needham-Schroeder-Lowe: A Secure Protocol

If we consider Lowe's fix of the Needham-Schroeder protocol, Maude-NPA is able to prove that the protocol is secure, that is, no initial state is found and the search space is finite.

We recall the informal specification of NSL, as follows:

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

Note that the only change in NSL protocol w.r.t. the NSPK protocol is that the responder B sends both nonces together with his name, instead of sending just both nonces.

The protocol is specified in Maude-NPA as follows, keeping the sort and operator declarations as well as the intruder strands of the NSPK protocol.

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

The attack pattern representing the intruder able to learn the responder's nonce (similar to the NSPK attach pattern) is as follows:

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

And the search space associated to this attack pattern is as follows:

```
Maude> reduce in MAUDE-NPA : summary(0,1) .
result Summary: States>> 4 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(0,2) .
result Summary: States>> 7 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(0,3) .
result Summary: States>> 6 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(0,4) .
result Summary: States>> 2 Solutions>> 0
Maude> reduce in MAUDE-NPA : summary(0,5) .
result Summary: States>> 0 Solutions>> 0
```

We can also specify an authentication attack using a never pattern:

```
  eq ATTACK-STATE(1)
   = :: r ::
     [ nil, -(pk(b,a ; N)),
            +(pk(a, N ; n(b,r) ; b)),
            -(pk(b,n(b,r))) | nil ]
     || empty
```

```
    || nil
    || nil
    || never(
       (:: r' ::
       [ nil | +(pk(b,a ; N1)),
               -(pk(a,N1 ; n(b,r) ; b)),
               +(pk(b, n(b,r))), nil ]
        & SS:StrandSet)
        || IK:IntruderKnowledge )
  [nonexec] .
```

And the search space associated to this attack pattern is as follows:

```
reduce in MAUDE-NPA : summary(1,0) .
result Summary: States>> 1 Solutions>> 0
reduce in MAUDE-NPA : summary(1,1) .
result Summary: States>> 2 Solutions>> 0
reduce in MAUDE-NPA : summary(1,2) .
result Summary: States>> 4 Solutions>> 0
reduce in MAUDE-NPA : summary(1,3) .
result Summary: States>> 5 Solutions>> 0
reduce in MAUDE-NPA : summary(1,4) .
result Summary: States>> 2 Solutions>> 0
reduce in MAUDE-NPA : summary(1,5) .
result Summary: States>> 0 Solutions>> 0
```

10.3 Needham-Schroeder-Lowe with Homomorphic Encryption

When we consider the secure Needham-Schroeder-Lowe protocol from Section 10.2 and make encryption homomorphic over concatenation, the protocol becomes insecure. In order to be able to use the homomorphic unification available in Maude-NPA, the encryption operator must swap its arguments in contrast to the protocol definition of Section 10.2. That is, the informal specification of NSL is as follows:

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

There are a number of ways in which either A or B can be tricked into believing that they have successfully completed a run of the protocol with one another, when in fact this has not happened. Here is one of the simplest:

1. $I_A \to B : pke(N_I; A, B)$
2. $B \to I_A : pke(N_I; N_B; B, A)$

   This message is intercepted by the intruder, who, thanks to the homomorphic property, is able to extract $pke(N_B, A)$. He uses this to initiate the protocol

with A, posing as B. Again the intruder uses the homomorphic property to build the following message:

3. $I_B \rightarrow A : pke(N_B; B, A)$
4. $A \rightarrow I_B : pke(N_A; N_B; A, B)$

   The intruder is now able to extract $pke(N_B, B)$ and use it to complete its impersonation of A to B.

5. $I_A \rightarrow B : pke(N_B, B)$.

The protocol is specified in Maude-NPA as follows. First the sort and operator declarations, similar to the NSPK and NSL protocols.

```
sorts Name Nonce Key .
subsort Name Nonce Key < Msg .
subsort Name < Key .
subsort Name < Public .

op pk : Msg Key -> Msg [frozen] .

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

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

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

The equational property was specified in Section 4.4.1.

```
eq pk(X:Msg ; Y:Msg, K:Key) = pk(X:Msg, K:Key) ; pk(Y:Msg, K:Key)
     [nonexec label homomorphism metadata "builtin-unify"] .
```

The Dolev-Yao intruder capabilities here reflect the cancellation of encryption and decryption explicitly. We recall that this is necessary because the homomorphic unification algorithm cannot be used in conjunction with any other theory.

```
vars X Y : Msg .
vars A B : Name .
var Ke : Key .
var r : Fresh .

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), +(pk(X,Ke)), nil ] &
   :: nil :: [ nil | -(pk(X,i)), +(X), nil ] &
```

```
    :: r ::   [ nil | +(n(i,r)), nil ] &
    :: nil :: [ nil | +(A), nil ]
  [nonexec] .
```

The protocol is described as follows (very similar to the NSL protocol).

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

The attack pattern representing the intruder able to learn the responder's nonce (similar to the NSPK attach pattern) is as follows:

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

The following initial state from the previous attack pattern is found after seven backwards reachability steps[11]. Maude-NPA terminates its search after fifteen backwards reachability steps, since by that point all the states it is trying to reach are either initial or proved unreachable.

```
(< 1 . 2 . 9 . 12 . 10{1} . 6 . 1 . 1 > (
:: nil ::
[ nil |
   -(pk(i, b)),
   -(pk(n(b, #0:Fresh), b)),
   +(pk(i, b) ; pk(n(b, #0:Fresh), b)), nil]  &
:: nil ::
[ nil |
   -(pk(n(b, #0:Fresh), i)),
   +(n(b, #0:Fresh)), nil]  &
:: nil ::
[ nil |
```

[11] The keyword `generatedByIntruder` in the actual message list exchanged by the principals is included only for debugging purposes and does not imply any exchange between principals.

```
   -(pk(n(b, #0:Fresh), i) ; pk(n(b, #1:Fresh), i) ; pk(b, i)),
   +(pk(n(b, #0:Fresh), i)), nil]  &
:: #1:Fresh ::
[ nil |
   -(pk(i ; n(b, #0:Fresh), b)),
   +(pk(n(b, #0:Fresh) ; n(b, #1:Fresh) ; b, i)), nil]  &
:: #2:Fresh ::
[ nil |
   +(pk(a ; n(a, #2:Fresh), b)),
   -(pk(n(a, #2:Fresh) ; n(b, #0:Fresh) ; b, a)),
   +(pk(n(b, #0:Fresh), b)), nil]  &
:: #0:Fresh ::
[ nil |
   -(pk(a ; n(a, #2:Fresh), b)),
   +(pk(n(a, #2:Fresh) ; n(b, #0:Fresh) ; b, a)),
   -(pk(n(b, #0:Fresh), b)), nil] )
|
pk(i, b) !inI,
pk(n(b, #0:Fresh), b) !inI,
pk(n(b, #0:Fresh), i) !inI,
pk(a ; n(a, #2:Fresh), b) !inI,
pk(n(a, #2:Fresh) ; n(b, #0:Fresh) ; b, a) !inI,
pk(n(b, #0:Fresh) ; n(b, #1:Fresh) ; b, i) !inI,
n(b, #0:Fresh) !inI,
(pk(i, b) ; pk(n(b, #0:Fresh), b)) !inI
|
+(pk(a ; n(a, #2:Fresh), b)),
-(pk(a ; n(a, #2:Fresh), b)),
+(pk(n(a, #2:Fresh) ; n(b, #0:Fresh) ; b, a)),
-(pk(n(a, #2:Fresh) ; n(b, #0:Fresh) ; b, a)),
+(pk(n(b, #0:Fresh), b)),
generatedByIntruder(pk(i, b)),
-(pk(i, b)),
-(pk(n(b, #0:Fresh), b)),
+(pk(i, b) ; pk(n(b, #0:Fresh), b)),
-(pk(i ; n(b, #0:Fresh), b)),
+(pk(n(b, #0:Fresh) ; n(b, #1:Fresh) ; b, i)),
-(pk(n(b, #0:Fresh), i) ; pk(n(b, #1:Fresh), i) ; pk(b, i)),
+(pk(n(b, #0:Fresh), i)),
-(pk(n(b, #0:Fresh), i)),
+(n(b, #0:Fresh)),
-(pk(n(b, #0:Fresh), b))
|
nil)
```

10.4 Needham-Schroeder-Lowe with Exclusive-or

Similarly to the previous section, if we replace concatenation by exclusive-or, the protocol becomes insecure. The informal specification of NSL is as follows:

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

The attack can be performed as follows:

1. $A \rightarrow I_B : pk(i, N_A; A)$
   The intruder extracts the nonce from the initiator and starts another session with the responder.

2. $I_A \rightarrow B : pk(B, N_A; A)$

3. $B \rightarrow A : pk(A, N_A; N_B * B)$
   The initiator is expecting a message of the form $pk(A, N_A; N_I * I)$ where N_I is unknown. But the message $pk(A, N_A; N_B * B)$ can also be interpreted as $pk(A, N_A; N_B * B * i * i)$ where $N_I = N_B * B * i$.

4. $A \rightarrow I_B : pk(i, N_B * B * i)$
   The intruder is able to extract N_B by decrypting and composing the message with i and B, which are known to the intruder.

The protocol is specified in Maude-NPA as follows. First the sort and operator declarations, similar to the NSPK and NSL protocols.

```
sorts Name Nonce NNSet .
subsort Name Nonce NNSet < Msg .
subsort Name < Public .
subsort Name Nonce < NNSet .

op pk : Name Msg -> Msg [frozen] .
op sk : Name Msg -> Msg [frozen] .

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

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

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

op _*_ : NNSet NNSet -> NNSet [assoc comm frozen] .
op null : -> NNSet .
```

The equational properties are the exclusive-or and the cancellation of encryption and decryption.

```
*** Encryption/Decryption Cancellation
eq pk(A:Name,sk(A:Name,Z:Msg)) = Z:Msg [variant] .
eq sk(A:Name,pk(A:Name,Z:Msg)) = Z:Msg [variant] .

*** Exclusive or properties
eq XN:NNSet * XN:NNSet = null                [variant] .
eq XN:NNSet * XN:NNSet * YN:NNSet = YN:NNSet [variant] .
eq XN:NNSet * null = XN:NNSet                [variant] .
```

The Dolev-Yao intruder capabilities here reflect the cancellation of encryption and decryption explicitly.

```
vars X Y : Msg .
vars A B : Name .
vars XN YN : NNSet .
var r : Fresh .

eq STRANDS-DOLEVYAO
 = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
   :: nil :: [ nil | -(X ; Y), +(X), nil ] &
   :: nil :: [ nil | -(X ; Y), +(Y), nil ] &
   :: nil :: [ nil | -(XN), -(YN), +(XN * YN), nil ] &
   :: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
   :: nil :: [ nil | -(X), +(pk(A,X)), nil ] &
   :: nil :: [ nil | +(null), nil ] &
   :: r   :: [ nil | +(n(i,r)), nil ] &
   :: nil :: [ nil | +(A), nil ]
```

The protocol is described as follows (very similar to the NSL protocol).

```
eq STRANDS-PROTOCOL
= :: r ::  *** Bob ***
  [nil | +(pk(B, n(A,r) ; A)),
         -(pk(A, n(A,r) ; B * YN)),
         +(pk(B, YN)), nil]
  &
  :: r' :: *** Alice ***
  [nil | -(pk(B, XN ; A)),
         +(pk(A, XN ; B * n(B,r'))),
         -(pk(B,n(B,r'))), nil]
```

The attack pattern representing the intruder able to learn the responder's nonce (similar to the NSPK attach pattern) is as follows:

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

The following initial state from the previous attack pattern is found after eight backwards reachability steps. Maude-NPA also terminates its search[12] after thirteen backwards reachability steps, since by that point all the states it is trying to reach are either initial or proved unreachable.

```
result ShortIdSystem: < 1 . 9 . 7[2] . 2 . 7[3] . 1 . 6 . 2 . 1 > (
:: nil ::
[ nil |
  -(pk(i, n(a, #0:Fresh) ; a)),
  +(n(a, #0:Fresh) ; a), nil]  &
:: nil ::
[ nil |
  -(pk(i, b * i * n(b, #1:Fresh))),
  +(b * i * n(b, #1:Fresh)), nil]  &
:: nil ::
[ nil |
  -(n(a, #0:Fresh) ; a),
  +(pk(b, n(a, #0:Fresh) ; a)), nil]  &
:: nil ::
[ nil |
  -(n(b, #1:Fresh)),
  +(pk(b, n(b, #1:Fresh))), nil]  &
:: nil ::
[ nil |
  -(b * i),
  -(b * i * n(b, #1:Fresh)),
  +(n(b, #1:Fresh)), nil]  &
:: #1:Fresh ::
[ nil |
  -(pk(b, n(a, #0:Fresh) ; a)),
  +(pk(a, n(a, #0:Fresh) ; b * n(b, #1:Fresh))),
  -(pk(b, n(b, #1:Fresh))), nil]  &
:: #0:Fresh ::
[ nil |
  +(pk(i, n(a, #0:Fresh) ; a)),
  -(pk(a, n(a, #0:Fresh) ; b * n(b, #1:Fresh))),
  +(pk(i, b * i * n(b, #1:Fresh))), nil] )
|
pk(a, n(a, #0:Fresh) ; b * n(b, #1:Fresh)) !inI,
pk(b, n(a, #0:Fresh) ; a) !inI,
pk(b, n(b, #1:Fresh)) !inI,
pk(i, n(a, #0:Fresh) ; a) !inI,
pk(i, b * i * n(b, #1:Fresh)) !inI,
(n(a, #0:Fresh) ; a) !inI,
n(b, #1:Fresh) !inI,
(b * i) !inI,
(b * i * n(b, #1:Fresh)) !inI
|
+(pk(i, n(a, #0:Fresh) ; a)),
-(pk(i, n(a, #0:Fresh) ; a)),
+(n(a, #0:Fresh) ; a),
-(n(a, #0:Fresh) ; a),
+(pk(b, n(a, #0:Fresh) ; a)),
generatedByIntruder(b * i),
-(pk(b, n(a, #0:Fresh) ; a)),
+(pk(a, n(a, #0:Fresh) ; b * n(b, #1:Fresh))),
-(pk(a, n(a, #0:Fresh) ; b * n(b, #1:Fresh))),
+(pk(i, b * i * n(b, #1:Fresh))),
```

[12]The keyword `resuscitated` in the actual message list exchanged by the principals is included only for debugging purposes and does not imply any exchange between principals.

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

10.5 Needham-Schroeder-Lowe with Type Confusion

Like in the previous section, if we declare instead concatenation to be associative and use variables of the more general sort Msg, the protocol becomes insecure. The informal specification of NSL is as follows:

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

The attack can be performed as follows:

1. $I_A \to B : pk(b, i; A)$
   The intruder impersonates Alice and sends his name concatenated with Alice's name to Bob. Bob believes it is the starting message where the constant `i` corresponds to Alice's nonce. This is clearly a type confusion, only possible if Bob cannot check whether a bit string is a nonce.
2. $B \to I_A : pk(A, i; (N_B; B))$
   Bob replies with the standard message containing Alice's nonce, Bob's nonce and Alice's name.
3. $I \to A : pk(A, (i; N_B); B))$
   The intruder takes the message received from Bob and starts a new session by forwarding it to Alice. However, Alice it is going to interpret the message in a different way due to the associativity property, believing that $i; N_B$ is Bob's nonce.
4. $A \to B : pk(i, (i; N_B); N_A; A)$ The intruder now has obtained all the elements.
5. $I_A \to B : pk(B, N_B)$ The intruder finally returns Bob's nonce, since it can be extracted easily from the previous message.

The protocol is specified in Maude-NPA as follows. First the sort and operator declarations, similar to the NSPK and NSL protocols except that concatenation is specified with the associativity axiom.

```
sorts Name Nonce NNSet .
subsort Name Nonce NNSet < Msg .
subsort Name < Public .
subsort Name Nonce < NNSet .

op pk : Name Msg -> Msg [frozen] .
op sk : Name Msg -> Msg [frozen] .

op _;_ : Msg Msg -> Msg [assoc frozen] .

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

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

The equational properties are just the cancellation of encryption and decryption.

```
*** Encryption/Decryption Cancellation
eq pk(A:Name,sk(A:Name,Z:Msg)) = Z:Msg [variant] .
eq sk(A:Name,pk(A:Name,Z:Msg)) = Z:Msg [variant] .
```

The Dolev-Yao intruder capabilities here are the same as the original NSL.

```
var Ke : Key .
vars X Y Z : Msg .
vars A B : Name .

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 ] &
   :: nil :: [ nil | +(A), nil ]
[nonexec] .
```

The protocol is described as follows, very similar to the NSL protocol except that we use variables of sort Msg instead of variables of sort Nonce.

```
vars r r' : Fresh .
vars NA NB : Msg .
eq STRANDS-PROTOCOL
 = :: r :: *** Bob ***
   [ nil | +(pk(B,A ; n(A,r))),
           -(pk(A,n(A,r) ; NB ; B)),
           +(pk(B, NB)), nil ] &
   :: r :: *** Alice ***
   [ nil | -(pk(B,A ; NA)),
```

```
            +(pk(A, NA ; n(B,r) ; B)),
            -(pk(B,n(B,r))), nil ]
  [nonexec] .
```

The attack pattern representing the intruder being able to learn the responder's nonce is identical to the original NSL is as follows:

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

The following initial state from the attack pattern is found in five steps.

```
result ShortIdSystem: < 1 . 5 . 5 . 2 . 10 . 1 > (
:: nil ::
[ nil |
   -(pk(i, n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a)),
   +(n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a), nil]  &
:: nil ::
[ nil |
   -(n(b, #0:Fresh)),
   +(pk(b, n(b, #0:Fresh))), nil]  &
:: nil ::
[ nil |
   -(n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a),
   +(n(b, #0:Fresh)), nil]  &
:: #0:Fresh ::
[ nil |
   -(pk(b, a ; i)),
   +(pk(a, i ; n(b, #0:Fresh) ; b)),
   -(pk(b, n(b, #0:Fresh))), nil]  &
:: #1:Fresh ::
[ nil |
   -(pk(a, i ; n(b, #0:Fresh) ; b)),
   +(pk(i, n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a)), nil] )
|
pk(a, i ; n(b, #0:Fresh) ; b) !inI,
pk(b, n(b, #0:Fresh)) !inI,
pk(b, a ; i) !inI,
pk(i, n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a) !inI,
n(b, #0:Fresh) !inI,
(n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a) !inI
|
```

```
generatedByIntruder(pk(b, a ; i)),
-(pk(b, a ; i)),
+(pk(a, i ; n(b, #0:Fresh) ; b)),
-(pk(a, i ; n(b, #0:Fresh) ; b)),
+(pk(i, n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a)),
-(pk(i, n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a)),
+(n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a),
-(n(b, #0:Fresh) ; b ; n(a, #1:Fresh) ; a),
+(n(b, #0:Fresh)),
-(n(b, #0:Fresh)),
+(pk(b, n(b, #0:Fresh))),
-(pk(b, n(b, #0:Fresh)))
|
nil
```

10.6 Diffie-Hellman Protocol

The informal textbook-level description of the protocol is as follows.

1. $A \rightarrow B : A \ ; \ B \ ; \ g^{N_A}$
2. $B \rightarrow A : A \ ; \ B \ ; \ g^{N_B}$
3. $A \rightarrow B : e(g^{N_B \cdot N_A}, secret)$

The initiator `A` starts out by sending her name, the name of `B`, and `g` raised to the `NA` where `g` is the generator of the Diffie-Hellman group being used, and `NA` is a nonce. She is supposed to get back the concatenation of her name, the name of `B`, and a generator `g` raised to `B`'s nonce `NB`. However, all she can tell is that she receives the two names and an exponentiation, called `XE`. Then, she replies by encrypting a secret (to be shared with `B`) with `XE` raised to her nonce `NA`.

The sorts used in this protocol are as follows, where sorts `GenvExp`, `Gen`, and `Exp` have been explained in Section 4.3 above.

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

The operations used are as follows, where operators `g` and `exp` have been explained in Section 4.3 above.

```
--- Secret
op sec : Name Fresh -> Secret [frozen] .
--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .
--- Intruder
ops a b i : -> Name .
--- Encryption
```

```
op e : Key Msg -> Msg [frozen] .
op d : Key Msg -> Msg [frozen] .
--- Exp
op exp : GenvExp NeNonceSet -> Exp [frozen] .
--- Gen
op g : -> Gen .
--- NeNonceSet
subsort Nonce < NeNonceSet .
op _*_ : NeNonceSet NeNonceSet -> NeNonceSet [frozen assoc comm] .
--- Concatenation
op _;_ : Msg Msg -> Msg [frozen gather (e E)] .
```

The algebraic equations besides associative-commutative are specified as follows:

```
eq exp(exp(W:Gen,Y:NeNonceSet),Z:NeNonceSet)
 = exp(W:Gen, Y:NeNonceSet * Z:NeNonceSet) [variant] .
eq e(K:Key,d(K:Key,M:Msg)) = M:Msg [variant] .
eq d(K:Key,e(K:Key,M:Msg)) = M:Msg [variant] .
```

The Dolev-Yao intruder capabilities associated with these operation symbols are described as follows.

```
vars M M1 M2 : Msg .
vars NS1 NS2 : NeNonceSet .
var GE : GenvExp .
vars A B : Name .
var Ke : Key .
var r : Fresh .

eq STRANDS-DOLEVYAO =
   :: nil :: [ nil | -(M1 ; M2), +(M1), nil ] &
   :: nil :: [ nil | -(M1 ; M2), +(M2), nil ] &
   :: nil :: [ nil | -(M1), -(M2), +(M1 ; M2), nil ] &
   :: nil :: [ nil | -(Ke), -(M), +(e(Ke,M)), nil ] &
   :: nil :: [ nil | -(Ke), -(M), +(d(Ke,M)), nil ] &
   :: nil :: [ nil | -(NS1), -(NS2), +(NS1 * NS2), nil ] &
   :: nil :: [ nil | -(GE), -(NS), +(exp(GE,NS)), nil ] &
   :: r ::   [ nil | +(n(i,r)), nil ] &
   :: nil :: [ nil | +(g), nil ] &
   :: nil :: [ nil | +(A), nil ]
[nonexec] .
```

The informal textbook-level description above is specified in Maude-NPA as follows, taking into account that a received exponentiation that is unknown to a principal is represented by a variable. The strand for principal `A` is:

```
   :: r,r' ::
   [nil | +(A ; B ; exp(g,n(A,r))),
          -(A ; B ; XE),
          +(e(exp(XE,n(A,r)),sec(A,r'))), nil]
```

Note that A uses two fresh variables, one for the nonce and another for the secret data.

The strand for principal B is:

```
  :: r ::
  [nil | -(A ; B ; XE),
         +(A ; B ; exp(g,n(B,r))),
         -(e(exp(XE,n(B,r)),Sr)), nil]
```

For this protocol, the next thing to appear is not an attack state but the following:

```
eq EXTRA-GRAMMARS
  = (grl empty => (NS * n(a,r)) inL . ;
     grl empty => n(a,r) inL . ;
     grl empty => (NS * n(b,r)) inL . ;
     grl empty => n(b,r) inL .
     ! S2 )
  [nonexec] .
```

This is an additional *initial grammar*, that Maude-NPA uses to generate a class of unreachable states. Initial grammars are usually generated automatically; however, for some theories, such as the Diffie-Hellman equational theory, it is useful to specify additional ones. More information about grammars and methods for specifying them is given in Appendix C.

The attack state is represented by the following authentication pattern, which specifies that the B's strand is required to appear in the possible initial state, but A's corresponding strand does not. Thus B's strand is specified in the body of the state, and A's is specified as a never pattern.

```
  eq ATTACK-STATE(0)
   = :: r ::
     [nil, -(a ; b ; XE),
           +(a ; b ; exp(g,n(b,r))),
           -(e(exp(XE,n(b,r)),sec(a,r'))) | nil]
     || empty
     || nil
     || nil
     || never
     *** Pattern for authentication
     (:: R:FreshSet ::
     [nil | +(a ; b ; XE),
            -(a ; b ; exp(g,n(b,r))),
            +(e(YE,sec(a,r'))), nil]
      & S:StrandSet || K:IntruderKnowledge)
  [nonexec] .
```

The search terminates in nine backwards narrowing steps and three attacks are found. We list the first:

```
< 1[2] . 5 . 6 . 13 . 3 . 3 . 1 . 6 . 1 > (
:: nil ::
[ nil |
   -(exp(g, n(b, #0:Fresh))),
   -(#1:NeNonceSet),
   +(exp(g, #1:NeNonceSet * n(b, #0:Fresh))), nil]  &
:: nil ::
[ nil |
   -(exp(g, #1:NeNonceSet * n(b, #0:Fresh))),
   -(sec(a, #2:Fresh)),
   +(e(exp(g, #1:NeNonceSet * n(b, #0:Fresh)), sec(a, #2:Fresh))), nil]  &
:: nil ::
[ nil |
   -(exp(#3:Exp, n(a, #4:Fresh))),
   -(e(exp(#3:Exp, n(a, #4:Fresh)), sec(a, #2:Fresh))),
   +(sec(a, #2:Fresh)), nil]  &
:: nil ::
[ nil |
   -(a ; b ; exp(g, n(b, #0:Fresh))),
   +(b ; exp(g, n(b, #0:Fresh))), nil]  &
:: nil ::
[ nil |
   -(b ; exp(g, n(b, #0:Fresh))),
   +(exp(g, n(b, #0:Fresh))), nil]  &
:: #0:Fresh ::
[ nil |
   -(a ; b ; exp(g, #1:NeNonceSet)),
   +(a ; b ; exp(g, n(b, #0:Fresh))),
   -(e(exp(g, #1:NeNonceSet * n(b, #0:Fresh)), sec(a, #2:Fresh))), nil]  &
:: #2:Fresh,#4:Fresh ::
[ nil |
   +(a ; #5:Name ; exp(g, n(a, #4:Fresh))),
   -(a ; #5:Name ; #3:Exp),
   +(e(exp(#3:Exp, n(a, #4:Fresh)), sec(a, #2:Fresh))), nil] )
|
#1:NeNonceSet !inI,
sec(a, #2:Fresh) !inI,
e(exp(g, #1:NeNonceSet * n(b, #0:Fresh)), sec(a, #2:Fresh)) !inI,
e(exp(#3:Exp, n(a, #4:Fresh)), sec(a, #2:Fresh)) !inI,
exp(g, n(b, #0:Fresh)) !inI,
exp(g, #1:NeNonceSet * n(b, #0:Fresh)) !inI,
exp(#3:Exp, n(a, #4:Fresh)) !inI,
(a ; b ; exp(g, #1:NeNonceSet)) !inI,
(a ; b ; exp(g, n(b, #0:Fresh))) !inI,
(a ; #5:Name ; #3:Exp) !inI,
(b ; exp(g, n(b, #0:Fresh))) !inI,
irr(e(exp(g, #1:NeNonceSet * n(b, #0:Fresh)), sec(a, #2:Fresh))),
inst(#3:Exp),
inst(#1:NeNonceSet)
|
+(a ; #5:Name ; exp(g, n(a, #4:Fresh))),
generatedByIntruder(exp(#3:Exp, n(a, #4:Fresh))),
generatedByIntruder(a ; #5:Name ; #3:Exp),
-(a ; #5:Name ; #3:Exp),
+(e(exp(#3:Exp, n(a, #4:Fresh)), sec(a, #2:Fresh))),
generatedByIntruder(#1:NeNonceSet),
generatedByIntruder(a ; b ; exp(g, #1:NeNonceSet)),
-(a ; b ; exp(g, #1:NeNonceSet)),
+(a ; b ; exp(g, n(b, #0:Fresh))),
-(a ; b ; exp(g, n(b, #0:Fresh))),
+(b ; exp(g, n(b, #0:Fresh))),
-(b ; exp(g, n(b, #0:Fresh))),
```

```
+(exp(g, n(b, #0:Fresh))),
-(exp(g, n(b, #0:Fresh))),
-(#1:NeNonceSet),
+(exp(g, #1:NeNonceSet * n(b, #0:Fresh))),
-(exp(#3:Exp, n(a, #4:Fresh))),
-(e(exp(#3:Exp, n(a, #4:Fresh)), sec(a, #2:Fresh))),
+(sec(a, #2:Fresh)),
-(exp(g, #1:NeNonceSet * n(b, #0:Fresh))),
-(sec(a, #2:Fresh)),
+(e(exp(g, #1:NeNonceSet * n(b, #0:Fresh)), sec(a, #2:Fresh))),
-(e(exp(g, #1:NeNonceSet * n(b, #0:Fresh)), sec(a, #2:Fresh)))
|
nil
```

We note, however, that this is *not* the famous man-in-the-middle attack on unauthenticated Diffie-Hellman. Instead, it is a much more trivial attack in which the attacker removes the appended names from the initiator's message and substitutes some others. Thus, the responder is sharing a key with an honest initiator, just not the initiator he thinks. Furthermore, another of the three attacks that Maude-NPA displays is only a slight variant.

The reason why Maude-NPA finds the trivial attack and not the man-in-the-middle attack is because of the way Maude-NPA optimizes its search. If it finds two states S_1 and S_2 such that the unreachability of S_1 implies the unreachability of S_2 it discards S_2 and keeps S_1. If S_1 leads to an attack, then it could be S_2 would have led to a different attack. In other words, if a protocol is insecure, Maude-NPA will find at least one attack, but it is not guaranteed to find all attacks possible.

Let's try asking Maude-NPA a different question. An intruder may not only want to mislead principals about who they are talking to, but may also want to find out the secret himself. So we ask the following question:

```
  eq ATTACK-STATE(1)
   = :: r ::
      [nil, -(a ; b ; XE),
            +(a ; b ; exp(g,n(b,r))),
            -(e(exp(XE,n(b,r)),sec(a,r'))) | nil]
      || sec(a,r') inI
      || nil
      || nil
      || nil
  [nonexec] .
```

and we get the Man-in-the-Middle attack as follows[13]:

```
< 2] . 9 . 6 . 15 . 13 . 3 . 12 . 2 . 2 . 1 > (
:: nil ::
[ nil |
   -(exp(g, n(b, #0:Fresh))),
   -(#1:NeNonceSet),
   +(exp(g, #1:NeNonceSet * n(b, #0:Fresh))), nil]  &
```

[13]The keyword `resuscitated` in the actual message list exchanged by the principals is included only for debugging purposes and does not imply any exchange between principals.

```
:: nil ::
[ nil |
   -(exp(g, #1:NeNonceSet * n(b, #0:Fresh))),
   -(#2:NeNonceSet),
   +(exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh))), nil]  &
:: nil ::
[ nil |
   -(exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh))),
   -(sec(a, #3:Fresh)),
   +(e(exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh)), sec(a, #3:Fresh))), nil]  &
:: nil ::
[ nil |
   -(exp(#4:Exp, n(a, #5:Fresh))),
   -(e(exp(#4:Exp, n(a, #5:Fresh)), sec(a, #3:Fresh))),
   +(sec(a, #3:Fresh)), nil]  &
:: nil ::
[ nil |
   -(a ; b ; exp(g, n(b, #0:Fresh))),
   +(b ; exp(g, n(b, #0:Fresh))), nil]  &
:: nil ::
[ nil |
   -(b ; exp(g, n(b, #0:Fresh))),
   +(exp(g, n(b, #0:Fresh))), nil]  &
:: #0:Fresh ::
[ nil |
   -(a ; b ; exp(g, #1:NeNonceSet * #2:NeNonceSet)),
   +(a ; b ; exp(g, n(b, #0:Fresh))),
   -(e(exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh)), sec(a, #3:Fresh))), nil]  &
:: #3:Fresh,#5:Fresh ::
[ nil |
   +(a ; #6:Name ; exp(g, n(a, #5:Fresh))),
   -(a ; #6:Name ; #4:Exp),
   +(e(exp(#4:Exp, n(a, #5:Fresh)), sec(a, #3:Fresh))), nil] )
|
#1:NeNonceSet !inI,
#2:NeNonceSet !inI,
sec(a, #3:Fresh) !inI,
e(exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh)), sec(a, #3:Fresh)) !inI,
e(exp(#4:Exp, n(a, #5:Fresh)), sec(a, #3:Fresh)) !inI,
exp(g, n(b, #0:Fresh)) !inI,
exp(g, #1:NeNonceSet * n(b, #0:Fresh)) !inI,
exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh)) !inI,
exp(#4:Exp, n(a, #5:Fresh)) !inI,
(a ; b ; exp(g, n(b, #0:Fresh))) !inI,
(a ; b ; exp(g, #1:NeNonceSet * #2:NeNonceSet)) !inI,
(a ; #6:Name ; #4:Exp) !inI,
(b ; exp(g, n(b, #0:Fresh))) !inI,
irr(e(exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh)), sec(a, #3:Fresh))),
inst(#4:Exp),
inst(#1:NeNonceSet),
inst(#2:NeNonceSet)
|
generatedByIntruder(#2:NeNonceSet),
generatedByIntruder(#1:NeNonceSet),
generatedByIntruder(a ; b ; exp(g, #1:NeNonceSet * #2:NeNonceSet)),
-(a ; b ; exp(g, #1:NeNonceSet * #2:NeNonceSet)),
+(a ; b ; exp(g, n(b, #0:Fresh))),
-(a ; b ; exp(g, n(b, #0:Fresh))),
+(b ; exp(g, n(b, #0:Fresh))),
+(a ; #6:Name ; exp(g, n(a, #5:Fresh))),
generatedByIntruder(exp(#4:Exp, n(a, #5:Fresh))),
generatedByIntruder(a ; #6:Name ; #4:Exp),
```

```
-(a ; #6:Name ; #4:Exp),
+(e(exp(#4:Exp, n(a, #5:Fresh)), sec(a, #3:Fresh))),
-(b ; exp(g, n(b, #0:Fresh))),
+(exp(g, n(b, #0:Fresh))),
-(exp(g, n(b, #0:Fresh))),
-(#1:NeNonceSet),
+(exp(g, #1:NeNonceSet * n(b, #0:Fresh))),
-(exp(g, #1:NeNonceSet * n(b, #0:Fresh))),
-(#2:NeNonceSet),
+(exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh))),
-(exp(#4:Exp, n(a, #5:Fresh))),
-(e(exp(#4:Exp, n(a, #5:Fresh)), sec(a, #3:Fresh))),
+(sec(a, #3:Fresh)),
-(exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh))),
-(sec(a, #3:Fresh)),
+(e(exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh)), sec(a, #3:Fresh))),
-(e(exp(g, #1:NeNonceSet * #2:NeNonceSet * n(b, #0:Fresh)), sec(a, #3:Fresh)))
|
nil
```

The lesson to be learned here is that one must be sure to query Maude-NPA about *all* the properties that a protocol is supposed to have before concluding that it is secure.

10.7 NSL Distance Bounding Protocol

The NSL Distance Bounding protocol is one of the protocols for which the current version of Maude-NPA performs very poorly, mainly because of the lack of restrictions put on the inputs to the exclusive-or function in the distance bounding part of the protocol. Since the number of variants of an exclusive-or term grows rapidly with the size of the term, this leads to Maude-NPA generating a large number of potential unifiers, which has a negative impact on performance. In the version of the protocol presented below, we avoid this problem, and still find the attack, by omitting the exclusive-or Dolev-Yao strand, which is not needed for the attack.

The informal textbook-level description of the protocol is divided into the two protocols. The NSL protocol is specified as follows:

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

The distance bounding part is as follows:

4. $B \rightarrow A : N'_B$

   Bob records the time at which he sent N'_B

5. $A \rightarrow B : N_A \oplus N'_B$

   Bob records the time he receives the response and checks the equivalence $N_A = N_A \oplus N'_B \oplus N'_B$. If this holds, he uses the round-trip time of his challenge and response to estimate his distance from Alice

The sorts used in this protocol are as follows.

```
sorts Name Nonce NonceSet Enc .
subsort Name NonceSet Enc < Msg .
subsort Nonce < NonceSet .
subsort Name < Public .
```

The operations used are as follows.

```
--- Roles
ops init-nsl resp-nsl : -> Role .
ops init-db  resp-db :  -> Role .
--- Encoding operators for public/private encryption
op pk : Name Msg -> Enc [frozen] .
op sk : Name Msg -> Enc [frozen] .
--- Principals
op a : -> Name . --- Alice
op b : -> Name . --- Bob
op i : -> Name . --- Intruder
--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .
--- Concatenation operator
op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .
--- Exclusive-or operator
op _*_ : NonceSet NonceSet -> NonceSet  [assoc comm frozen] .
op null : -> NonceSet .
```

The algebraic equations besides associative-commutative are specified as follows:

```
vars X Y Z : Msg .
vars A B : Name .
vars XN YN : NonceSet .

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

*** Exclusive or properties
eq null * XN = XN [variant] .
eq XN * XN = null [variant] .
eq XN * XN * YN = YN [variant] .
```

The Dolev-Yao intruder capabilities associated with these operation symbols are described as follows.

```
var r : Fresh .
var A : Name .
vars NS NS' : NonceSet .
vars X Y : Msg .
```

```
eq STRANDS-DOLEVYAO =
   :: nil :: [ nil | -(NS), -(NS'), +(NS * NS'), nil ] &
   :: 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(A,X)), nil ] &
   :: nil :: [ nil | +(A) , nil ] &
   :: r :: [ nil | +(n(i,r)), nil ]
[nonexec] .
```

Note that the first strand, to compute an exclusive-or, is unnecessary to find the associated attack of Section 8.1 and, indeed, produces a huge search space with many states that are generated and later discarded by the tool.

The informal textbook-level description above is specified in Maude-NPA as follows.

```
eq STRANDS-PROTOCOL
 = :: r :: --- NSL-Alice
   [ nil | +(pk(B, n(A,r) ; A)) ,
           -(pk(A, n(A,r) ; NB ; B )),
           +(pk(B, NB)),
           {init-nsl -> resp-db ;; 1-1 ;;  (A ; B ; n(A,r))}, nil ] &
   :: r :: --- NSL-Bob
   [ nil | -(pk(B,NA ; A)),
           +(pk(A, NA ; n(B,r) ; B)),
           -(pk(B,n(B,r))),
           {resp-nsl -> init-db ;; 1-1 ;; (A ; B ; NA)}, nil ] &
   :: r' :: --- Init-DB
   [ nil | {resp-nsl -> init-db ;; 1-1 ;; (A ; B ; NA)},
           +(n(B,r')),
           -(NA * n(B,r')), nil] &
   :: nil :: ---Resp-DB
   [ nil | {init-nsl -> resp-db ;; 1-1 ;; (A ; B ; NA) },
           -(N),
           +(NA * N), nil ]
[nonexec] .
```

The attack state is represented by the following pattern associated to the attack described in Section 8.1.

```
eq ATTACK-STATE(0)
   = :: r ::
     [ nil,  +(pk(i,n(a,r) ; a)),
             -(pk(a,n(a,r) ; NC ; i)),
             +(pk(i, NC)),
             {init-nsl -> resp-db ;; 1-1 ;; a ; i ; n(a,r) } | nil] &
     :: r'' ::
     [ nil,  {resp-nsl -> init-db ;; 1-1 ;; i ; b ; n(a,r)},
```

```
            +(n(b,r'')),
            -(n(a,r) * n(b,r''))  | nil]
     || empty
     || nil
     || nil
     || nil
 [nonexec] .
```

The search terminates in eighteen backwards narrowing steps and two attacks are found. We list the second:

```
< 1 . 6 . 1 . 1 . 1 . 2 . 7 . 10 . 9 . 11 . 8 . 4 . 3 . 9 . 10 . 5 . 2 . 1 > (
:: nil ::
[ nil |
  -(pk(i, n(a, #0:Fresh) ; a)),
  +(n(a, #0:Fresh) ; a), nil]  &
:: nil ::
[ nil |
  -(pk(i, n(a, #0:Fresh) ; n(b, #1:Fresh) ; b)),
  +(n(a, #0:Fresh) ; n(b, #1:Fresh) ; b), nil]  &
:: nil ::
[ nil |
  -(n(a, #0:Fresh)),
  -(i),
  +(n(a, #0:Fresh) ; i), nil]  &
:: nil ::
[ nil |
  -(n(a, #0:Fresh)),
  -(#2:Nonce ; i),
  +(n(a, #0:Fresh) ; #2:Nonce ; i), nil]  &
:: nil ::
[ nil |
  -(n(b, #1:Fresh)),
  +(pk(b, n(b, #1:Fresh))), nil]  &
:: nil ::
[ nil |
  -(n(a, #0:Fresh) ; a),
  +(n(a, #0:Fresh)), nil]  &
:: nil ::
[ nil |
  -(n(a, #0:Fresh) ; i),
  +(pk(b, n(a, #0:Fresh) ; i)), nil]  &
:: nil ::
[ nil |
  -(n(a, #0:Fresh) ; #2:Nonce ; i),
  +(pk(a, n(a, #0:Fresh) ; #2:Nonce ; i)), nil]  &
:: nil ::
[ nil |
  -(n(a, #0:Fresh) ; n(b, #1:Fresh) ; b),
  +(n(b, #1:Fresh) ; b), nil]  &
:: nil ::
[ nil |
  -(n(b, #1:Fresh) ; b),
  +(n(b, #1:Fresh)), nil]  &
:: nil ::
[ nil |
  {init-nsl -> resp-db ;; 1-1 ;; a ; i ; n(a, #0:Fresh)},
  -(n(b, #3:Fresh)),
  +(n(a, #0:Fresh) * n(b, #3:Fresh)), nil]  &
:: #1:Fresh ::
```

```
[ nil |
  -(pk(b, n(a, #0:Fresh) ; i)),
  +(pk(i, n(a, #0:Fresh) ; n(b, #1:Fresh) ; b)),
  -(pk(b, n(b, #1:Fresh))),
  {resp-nsl -> init-db ;; 1-1 ;; i ; b ; n(a, #0:Fresh)}, nil]  &
:: #3:Fresh ::
[ nil |
  {resp-nsl -> init-db ;; 1-1 ;; i ; b ; n(a, #0:Fresh)},
  +(n(b, #3:Fresh)),
  -(n(a, #0:Fresh) * n(b, #3:Fresh)), nil]  &
:: #0:Fresh ::
[ nil |
  +(pk(i, n(a, #0:Fresh) ; a)),
  -(pk(a, n(a, #0:Fresh) ; #2:Nonce ; i)),
  +(pk(i, #2:Nonce)),
  {init-nsl -> resp-db ;; 1-1 ;; a ; i ; n(a, #0:Fresh)}, nil] )
|
pk(a, n(a, #0:Fresh) ; #2:Nonce ; i) !inI,
pk(b, n(b, #1:Fresh)) !inI,
pk(b, n(a, #0:Fresh) ; i) !inI,
pk(i, n(a, #0:Fresh) ; a) !inI,
pk(i, n(a, #0:Fresh) ; n(b, #1:Fresh) ; b) !inI,
n(a, #0:Fresh) !inI,
n(b, #1:Fresh) !inI,
n(b, #3:Fresh) !inI,
(#2:Nonce ; i) !inI,
(n(a, #0:Fresh) ; a) !inI,
(n(a, #0:Fresh) ; i) !inI,
(n(a, #0:Fresh) ; #2:Nonce ; i) !inI,
(n(a, #0:Fresh) ; n(b, #1:Fresh) ; b) !inI,
(n(b, #1:Fresh) ; b) !inI,
(n(a, #0:Fresh) * n(b, #3:Fresh)) !inI
|
+(pk(i, n(a, #0:Fresh) ; a)),
-(pk(i, n(a, #0:Fresh) ; a)),
+(n(a, #0:Fresh) ; a),
-(n(a, #0:Fresh) ; a),
+(n(a, #0:Fresh)),
-(n(a, #0:Fresh)),
-(i),
+(n(a, #0:Fresh) ; i),
-(n(a, #0:Fresh) ; i),
+(pk(b, n(a, #0:Fresh) ; i)),
-(pk(b, n(a, #0:Fresh) ; i)),
+(pk(i, n(a, #0:Fresh) ; n(b, #1:Fresh) ; b)),
-(pk(i, n(a, #0:Fresh) ; n(b, #1:Fresh) ; b)),
+(n(a, #0:Fresh) ; n(b, #1:Fresh) ; b),
-(n(a, #0:Fresh) ; n(b, #1:Fresh) ; b),
+(n(b, #1:Fresh) ; b),
generatedByIntruder(#2:Nonce ; i),
-(n(a, #0:Fresh)),
-(#2:Nonce ; i),
+(n(a, #0:Fresh) ; #2:Nonce ; i),
-(n(b, #1:Fresh) ; b),
+(n(b, #1:Fresh)),
-(n(a, #0:Fresh) ; #2:Nonce ; i),
+(pk(a, n(a, #0:Fresh) ; #2:Nonce ; i)),
-(n(b, #1:Fresh)),
+(pk(b, n(b, #1:Fresh))),
-(pk(a, n(a, #0:Fresh) ; #2:Nonce ; i)),
+(pk(i, #2:Nonce)),
-(pk(b, n(b, #1:Fresh))),
```

```
+(n(b, #3:Fresh)),
-(n(b, #3:Fresh)),
+(n(a, #0:Fresh) * n(b, #3:Fresh)),
-(n(a, #0:Fresh) * n(b, #3:Fresh))
|
nil
```

10.8 NSL Key Distribution Protocol

The informal textbook-level description of the protocol is divided into the two protocols. The NSL protocol is specified as follows:

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

In this protocol composition, the initiator of the session key protocol can be the child of either the initiator or the responder of the NSL protocol. So, we have two possible child executions after NSL:

4. $A \to B : \{Sk_A\}_{h(N_A,N_B)}$	4. $B \to A : \{Sk_B\}_{h(N_A,N_B)}$
5. $B \to A : \{Sk_A; N'_B\}_{h(N_A,N_B)}$	5. $A \to B : \{Sk_B; N'_A\}_{h(N_A,N_B)}$
6. $A \to B : \{N'_B\}_{h(N_A,N_B)}$	6. $B \to A : \{N'_A\}_{h(N_A,N_B)}$

The sorts used in this protocol are as follows.

```
sorts Name Nonce MKey SKey .
subsort Name Nonce MKey SKey < Msg .
subsort Name < Public .
```

The operations used are as follows.

```
--- Roles
ops init-nsl resp-nsl : -> Role .
ops init-kd  resp-kd :  -> Role .
--- Encoding operators for public/private encryption
op pk : Name Msg ->  Msg [frozen] .
op sk : Name Msg ->  Msg [frozen] .
--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .
--- Concatenation operator
op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .
--- Hash operator
op h :  Nonce Nonce -> MKey [ frozen ]  .
--- Key operator
op skey : Name Fresh -> SKey  [ frozen ] .
```

```
--- Encryption Operators
op e : MKey Msg  -> Msg [frozen] .
op d : MKey Msg ->  Msg [frozen] .
 --- Principals
op a : -> Name . --- Alice
op b : -> Name . --- Bob
op i : -> Name . --- Intruder
```

The algebraic equations besides associative-commutative are specified as follows:

```
vars Z : Msg .
var A  : Name .
 vars MKe : MKey .

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

eq d(MKe,e(MKe,Z)) = Z [variant] .
eq e(MKe,d(MKe,Z)) = Z [variant] .
```

The Dolev-Yao intruder capabilities associated with these operation symbols are described as follows.

```
var r : Fresh .
var A : Name .
vars NS NS' : NonceSet .
vars X Y : Msg .
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(A,X)), nil ] &
   :: nil :: [ nil | +(A) , nil ] &
   :: r :: [ nil | +(n(i,r)), nil ] &
   :: nil :: [ nil | -(MKe), -(X), +(e(MKe,X)), nil ] &
   :: nil :: [ nil | -(MKe), -(X), +(d(MKe,X)), nil ] &
   :: r :: [ nil | -(N), +(h(n(i,r), N)), nil ] &
   :: r :: [ nil | -(N), +(h(N, n(i,r))), nil ] &
   :: r :: [ nil | +(skey(i,r)), nil ]
[nonexec] .
```

The informal textbook-level description above is specified in Maude-NPA as follows.

```
eq STRANDS-PROTOCOL
   --- NSL protocol
   :: r ::
```

```
    [ nil | +(pk(B, n(A,r) ; A)),
            -(pk(A, n(A,r) ; NB ; B )),
            +(pk(B, NB)),
            {init-nsl -> init-kd resp-kd ;; 1-* ;;
                A ; B ; h(n(A,r) , NB) }, nil ] &
    :: r ::
    [ nil | -(pk(B,NA ; A)),
            +(pk(A, NA ; n(B,r) ; B)),
            -(pk(B,n(B,r))),
            {resp-nsl -> init-kd resp-kd ;; 1-* ;;
               B ; A ; h(NA , n(B,r))}, nil ] &
    ---- KD protocol
    :: r' ::
    [ nil | { init-nsl resp-nsl -> init-kd ;; 1-* ;; C ; D ; MKe },
            +(e(MKe, skey(C, r'))),
            -(e(MKe, skey(C, r') ; N)),
            +(e(MKe, N)), nil] &
    :: r' ::
    [ nil | { init-nsl resp-nsl -> resp-kd ;; 1-* ;; C ; D ; MKe },
            -(e(MKe, K)),
            +(e(MKe, K ; n(C,r'))),
            -(e(MKe, n(C,r'))), nil ]
  [nonexec] .
```

The attack state is represented by the following pattern where we ask whether the intruder can learn the session key.

```
eq ATTACK-STATE(0)
  =  :: r' ::
     [ nil,  { init-nsl -> init-kd ;; 1-* ;;  a ; b ; MKe },
             +(e(MKe, skey(a, r'))) ,
             -(e(MKe, skey(a, r') ;  n(b,r))),
             +(e(MKe,  n(b,r)))  | nil] &
      :: r ::
     [ nil,   { resp-nsl -> resp-kd ;; 1-* ;;  a ; b ;  MKe },
             -(e(MKe, skey(a,r'))),
             +(e(MKe, skey(a,r') ; n(b,r))),
             -(e(MKe, n(b,r))) | nil ]
     || skey(a, r') inI
     || nil
     || nil
     || nil
  [nonexec] .
```

Maude-NPA terminates its search after twenty four reachability steps, since by that point all the states it is trying to reach are either initial or proved unreachable.

10.9 Encryption Mode

The informal process-algebra description of the protocol is as follows.

$$
\begin{array}{ll}
(\mathit{Init}) & ((+(A_?\,;\,B_?\,;\,\mathit{pub})\cdot -(\mathit{pk}(A_?,B_?\,;\,\mathit{SK})) \\
& \quad ? \\
& \quad (+(A_?\,;\,B_?\,;\,\mathit{SharedKey})\cdot -(e(\mathit{key}(A_?,B_?),B_?\,;\,\mathit{SK})) \\
(\mathit{Resp}) & \quad -(A\,;\,B\,;\,\mathit{TEnc})\cdot \\
& \quad \mathit{if}\ \mathit{TEnc}=\mathit{pub} \\
& \qquad \mathit{then}\ (+(\mathit{pk}(A,B\,;\,\mathit{skey}(A,B,r')))) \\
& \qquad \mathit{else}\ (+(e(\mathit{key}(A,B),B\,;\,\mathit{skey}(A,B,r'))))
\end{array}
$$

The sorts used in this protocol are as follows.

```
sorts Name Nonce SKey Key Mode  .
subsort Name Nonce SKey Key Mode  < Msg .
subsort Name < Public .
```

The operations used are as follows.

```
--- Principals
op a : -> Name . --- Alice
op b : -> Name . --- Bob
op i : -> Name . --- Intruder
--- Modes
ops pubkey shkey : -> Mode .
--- Nonce operator
op n : Name Fresh -> Nonce [frozen] .
--- Concatenation operator
op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .
--- Key
op key : Name Name -> Key [frozen] .
--- Session Key
op skey : Name Fresh -> SKey [frozen] .
--- Public encryption
op pk : Name Msg -> Msg [frozen] .
op sk : Name Msg -> Msg [frozen] .
--- Shared key encryption
op she : Key Msg -> Msg [frozen] .
op shd : Key Msg -> Msg [frozen] .
```

The algebraic equations besides associative-commutative are specified as follows:

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

eq pk(A, sk(A, Z)) = Z [variant] .
eq sk(A, pk(A, Z)) = Z [variant] .
```

```
eq she(Ke, shd(Ke, Z)) = Z [variant] .
eq shd(Ke, she(Ke, Z)) = Z [variant] .
```

The Dolev-Yao intruder capabilities associated with these operation symbols are described as follows.

```
vars X Y : Msg .
var r : Fresh .
var A : Name .
var Ke : Key .

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), +(pk(A, X)), nil ] &
   :: nil :: [ nil | -(X), +(sk(i, X)), nil ] &
   :: nil :: [ nil | -(Ke), -(X), +(she(Ke, X)), nil ] &
   :: nil :: [ nil | -(Ke), -(X), +(shd(Ke, X)), nil ] &
   :: nil :: [ nil | +(key(i, A)), nil ] &
   :: nil :: [ nil | +(key(A, i)), nil ] &
   :: r   :: [ nil | +(skey(i,r)), nil ] &
   :: r   :: [ nil | +(n(i,r)), nil ] &
   :: nil :: [ nil | +(A) , nil ]
[nonexec] .
```

The informal process-algebra description above is specified in Maude-NPA as follows.

```
eq PROCESSES-PROTOCOL
 = (
    ( +(A ; B ;  pubkey) .
      -(pk(A, B ; SK)) .
      +(pk(B, A ; SK ; n(A,r))) .
      -(pk(A, B ; n(A,r)))
    )
    ?
    ( +(A ; B ;  shkey) .
      -(she(key(A, B), SK )) .
      +(she(key(A, B), SK ; n(A,r))) .
      -(she(key(A,B), n(A,r))) )
    )
   &
   ( -(A ; B ; mode) .
     (if (mode eq pubkey)
      then ( +(pk(A, B ; skey(B, r))) .
             -(pk(B, A ; skey(B,r) ; N)) .
             +(pk(A, B ; N)) )
```

```
        else ( +(she(key(A, B), skey(B,r))) .
               -(she(key(A, B), skey(B,r) ; N)) .
               +(she(key(A,B), N)) )
      )
    )
[nonexec] .
```

The attack states are used to ask whether the intruder can learn the session key in either of the possible choices of the protocol.

```
 --- intruder learns the session key in shared key
 eq ATTACK-PROCESS(0)
  = -(a ; b ; mode) .
    (mode neq pubkey) .
    +(she(key(a, b), skey(b,r))) .
    -(she(key(a, b), skey(b,r) ; N)) .
    +(she(key(a, b), N))
    || skey(b,r) inI
    || nil
[nonexec] .

 --- intruder learns the session key in public key
 eq ATTACK-PROCESS(1)
  = -(a ; b ; mode) .
    (mode eq pubkey) .
    +(pk(a, b ; skey(b, r))) .
    -(pk(b, a ; skey(b,r) ; N)) .
    +(pk(a, b ; N))
    || skey(b,r) inI
    || nil
[nonexec] .
```

Maude-NPA terminates its search after ten reachability steps for both attack patterns, since by that point all the states it is trying to reach are either initial or proved unreachable.

10.10 Rock-Paper-Scissors

In this cryptographic version of the simple but popular game, both initiator and responder first choose their hand shapes and send them to each other using a secure commitment scheme. Next, they both send each other the nonces that are necessary to open the commitments. Each of them then compares the two hand shapes and decides if the initiator wins, the responder wins, or there is a tie. The initiator then sends the responder the outcome. When the responder receives the initiator's verdict, it compares it against its own. It responds with "finished" if it agrees with the initiator and "cheater" if it doesn't. All messages are signed and encrypted, and the initiator's and responder's nonces are included in the messages concerning the

outcome of the game. The actual messages sent and choices made are described in more detail below.

The sorts used in this protocol are as follows.

```
  sorts Name Nonce Item Result OK ComMsg Status .
  subsorts Name Nonce Result OK ComMsg Item < Msg .
  subsorts Name Result Item Status < Public .
```

The operations used are as follows.

```
  ---Two outcomes: finished correctly of a player cheated
  ops finished cheater :-> Status .
--- Hand shapes
  ops rock scissors paper : -> Item .
--- Result
  ops win lose tie : -> Result .
 ---Result if check verifies
  op ok : -> OK .
--- Predicates for verification of properties
  op _beats_ : Msg Msg -> OK   [frozen] .
  op _beats_ : Item Item -> OK [frozen] .
   op item?_ :  Msg -> OK [frozen] .
--- Names
  op a : -> Name . --- Name for the Initiator
  op b : -> Name . --- Name for the Responder
  op i : -> Name . --- Name for the Intruder
 --- Nonce operator
  op n : Name Fresh -> Nonce [frozen] .
  --- Concatentation  operator
  op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .

  --- Encryption
  op sk : Name Msg -> Msg [frozen] .
  op pk : Name Msg -> Msg [frozen] .
  --- Operations for committing and opening a nonce
  op com :  Nonce  Item   -> ComMsg            [frozen] .
  op open : Nonce ComMsg  -> Msg               [frozen] .
  --- Public Key Signature
  op sig : Name Msg -> Msg      [frozen] .
```

The algebraic equations besides associative-commutative are specified as follows:

```
  var Z : Msg .
  var A : Name .
  var N : Nonce .
  var H : Item .
  *** Encryption/Decryption Cancellation
  eq pk(A,sk(A,Z)) = Z [variant] .
  eq sk(A,pk(A,Z)) = Z [variant] .
```

```
*** Opening a commitment
eq open(N, com(N,H)) = H [variant] .
*** Checking whether an item is received
eq item? H = ok [variant] .
*** Beats predicate
eq rock beats scissors = ok  [variant] .
eq scissors beats paper = ok [variant] .
eq paper beats rock = ok     [variant] .
```

The Dolev-Yao intruder capabilities associated with these operation symbols are described as follows.

```
vars X Y Z : Msg .
var r r' : Fresh .
vars NA NB N : Nonce .
vars XA XB : Item .
var ComXA ComXB : ComMsg  .
var A B : Name .
var R R' R1 R2 : Result .
var S : Status .

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) , +(sig(i, X)), nil] &
   :: nil :: [ nil | -(X), +(sk(i,X)), nil ]      &
   :: nil :: [ nil | -(X), +(pk(A,X)), nil ]      &
   :: nil :: [ nil | -(N), -(ComXA), +(open(N, ComXA)), nil] &
   :: nil :: [ nil | -(N), -(XA), +(com(N, XA)), nil] &
   :: r   :: [ nil | +(n(i, r)), nil] &
   :: nil :: [ nil | +(A), nil] &
   :: nil :: [ nil | +(R), nil] &
   :: nil :: [ nil | +(XA), nil]
[nonexec] .
```

The formal process-algebra description above is specified in Maude-NPA as follows.

```
eq PROCESSES-PROTOCOL =
  --- initiator
  +(pk(B, sig(A, com(n(A,r), XA)))) .
  -(pk(A, sig(B, ComXB))) .
  +(pk(B, sig(A, n(A , r)))) .
  -(pk(A, sig(B, NB))) .
  (if ((item? open(NB, ComXB)) eq  ok)
   then if ((XA beats open(NB, ComXB)) eq ok)
        then +(pk(B, sig(A, n(A, r)  ; win)))
        else if ((open(NB, ComXB) beats XA) eq ok)
             then +(pk(B, sig(A, n(A, r) ; lose )))
```

```
            else +(pk(B, sig(A, n(A, r) ; tie)))
     else nilP ) .
    -(pk(A, sig(B, n(A,r) ; NB)) ; S:Status)
    &
    --- responder
    -(pk(B, sig(A, ComXA))) .
    +(pk(A, sig(B, com(n(B,r), XB)))) .
    -(pk(B, sig(A, NA))) .
    +(pk(A, sig(B, n(B, r)))) .
    -(pk(B, sig(A, NA ; R))) .
    (if ((item? open(NA, ComXA)) eq ok)
     then if (R eq win)
          then if ((open(NA, ComXA) beats XB) eq ok)
               then +(pk(A, sig(B, NA ; n(B,r))) ; finished)
               else +(pk(A, sig(B, NA ; n(B,r))) ; cheater)
          else if (R eq lose)
                 then if ((XB beats open(NA, ComXA)) eq ok)
                      then +(pk(A, sig(B, NA ; n(B,r))) ; finished)
                      else +(pk(A, sig(B, NA ; n(B,r))) ; cheater)
                 else if (R eq tie)
                      then if (XB eq open(NA, ComXA))
                           then +(pk(A, sig(B, NA ; n(B,r))) ; finished)
                           else +(pk(A, sig(B, NA ; n(B,r))) ; cheater)
                      else nilP
     else nilP )
  [nonexec] .
```

The first attack state, written in the process algebra notation, is one in which the intruder, playing with the responder, learns the responder's commitment nonce before it is revealed.

```
eq ATTACK-PROCESS(1) =
   -(pk(b,sig(i, ComXA:ComMsg))) .
   +(pk(i,sig(b, com(n(b, r:Fresh), XB:Item)))) .
   -(pk(b,sig(i, NA:Nonce)))
   ||  n(b, r:Fresh) inI
   || nil
[nonexec] .
```

Maude-NPA terminates its search after the first reachability step, without finding an initial state, since all the states it is trying to reach are proved unreachable (e.g. by the grammars).

The next attack state, written in the strand notation, is one in which the responder believes that it has played a game with the initiator, and it has won the game, but there is no corresponding strand for the initiator.

Using that `red new-strands?` command, we get the following strands for an a responder who accepts the initiator's result and believes it has lost, and an initiator who has won:

```
:: r:Fresh ::
[ nil |
   -(pk(B:Name, sig(A:Name, ComXA:ComMsg))),
   +(pk(A:Name, sig(B:Name, com(n(B:Name, r:Fresh), XB:Item)))),
   -(pk(B:Name, sig(A:Name, NA:Nonce))),
   +(pk(A:Name, sig(B:Name, n(B:Name, r:Fresh)))),
   -(pk(B:Name, sig(A:Name, NA:Nonce ; n(B:Name, r:Fresh) ; R:Result))),
   (item? open(NA:Nonce, ComXA:ComMsg)) eq ok,
   R:Result eq win,
   (open(NA:Nonce, ComXA:ComMsg) beats XB:Item) eq ok,
   +(pk(A:Name, sig(B:Name, NA:Nonce ; n(B:Name, r:Fresh) ; finished))), nil]

:: r':Fresh ::
[ nil |
   +(pk(B:Name, sig(A:Name, com(n(A:Name, r':Fresh), XA:Item)))),
   -(pk(A:Name, sig(B:Name, ComXB:ComMsg))),
   +(pk(B:Name, sig(A:Name, n(A:Name, r':Fresh)))),
   -(pk(A:Name, sig(B:Name, NB:Nonce))),
   (item? open(NB:Nonce, ComXB:ComMsg)) eq ok,
   (XA:Item beats open(NB:Nonce, ComXB:ComMsg)) eq ok,
   +(pk(B:Name, sig(A:Name, n(A:Name, r':Fresh) ; NB:Nonce ; win))),
   -(pk(A:Name, sig(B:Name, n(A:Name, r':Fresh) ; NB:Nonce ; S:Status))), nil]
```

We cannot simply use these two strands (with the appropriate variable substitutions) to define the attack state though. Recall the strong irreducibility requirement on never patterns in Section 6.5. In order to obtain this for the rock-scissors-paper protocol, we must substitute values to the variables that will make the predicates either always reduce to `ok` or never reduce to `ok`. For the `item` predicate this will be achieved by replacing `ComXA:ComMsg` with `com(n(A:Name, r':Fresh), XA:Item)` and `ComXB:ComMsg` with `com(n(B:Name, r:Fresh), XB:Item)`. But in order to make the `beats` predicate evaluate to `ok` we will need to values for `XA:Item` and `XB:Item` so that `XA:Item` beats `XB:Item` is guaranteed. In this example we replace `XA:Item` with `rock` and `XB:Item` with `scissors`. We give the specification of the attack pattern below.

```
eq ATTACK-STATE(3) =
:: #1:Fresh ::
[ nil ,
   -(pk(b,sig(a, ComXA:ComMsg))),
   +(pk(a,sig(b, com(n(b, #1:Fresh), scissors)))),
   -(pk(b,sig(a, NA:Nonce))),
   +(pk(a,sig(b, n(b, #1:Fresh)))),
   -(pk(b,sig(a, NA:Nonce ; n(b, #1:Fresh) ; win))),
   (ok eq ok,
   (item? open(NA:Nonce, ComXA:ComMsg)) eq ok,
  win  eq win),
   (XB:Item beats open(NA:Nonce, ComXA:ComMsg)) eq ok,
```

```
   +(pk(b,pk(a,sig(b, NA:Nonce ; n(b, #1:Fresh) ; finished)))) | nil]
   || empty
     || nil
     || nil
     || never ( :: #0:Fresh ::
[ nil |
   +(pk(b,sig(a, com(NA:Nonce, rock)))),
   -(pk(a,sig(b, com(n(b, #1:Fresh), scissors)))),
   +(pk(b,sig(a, NA:Nonce))),
   -(pk(a,sig(b, n(b, #1:Fresh)))),
  ( ok eq ok,
   ok eq ok),
   +(pk(b,sig(a, NA:Nonce ; n(b, #1:Fresh) ; win))), nil]
    & S:StrandSet || K:IntruderKnowledge)
   [nonexec] .
```

The search terminates after 12 steps without finding an attack.

As the reader can see, specifying never patterns with choice can be tricky. We are currently working to make never patterns more streamlined, and to integrate them with the process algebra notation.

10.11 Other Protocol Examples

Descriptions of how Maude-NPA handles different protocols, equational theories or security properties are given in several papers: bounded associativity [17], Diffie-Hellman [14], variant-based exclusive-or [39], built-in homomorphic encryption [15], built-in exclusive-or [12, 13], variant-based homomorphic encryption [42], IBM CCA Security API [25], indistinguishability properties [38], and PKCS#11 API [26].

One class of problems that Maude-NPA has been used to analyzed is Cryptographic Application Programming Interfaces (cryptographic API), in particular IBM CCA Security API [25] and the PKCS#11 [26]. An API is a set of instructions by which a developer of an application may allow it to take advantage of the cryptographic functionality of a secure module. These APIs allow an application to perform such functions as creating keys, using keys to encrypt and decrypt data, and export and import keys to and from other devices. Cryptographic APIs should also enforce security policies, in particular, no application should be able to retrieve a key in the clear. We have been able to reproduce the attacks found in the literature while using more realistic specifications than in previous work, e.g. a realistic exclusive-or operator, or fewer restrictions or simplifications that was required in previous work.

On the other hand, there are cryptographic properties that Maude-NPA cannot handle; in this case we have explored approximations of the equational theories involve [42]. on approximating those properties is very useful. Maude-NPA requires equational theories to satisfy several properties. Fortunately, many cryptographic theories of interest satisfy these properties. But there is one important family of theories that fails to have a decomposition that satisfies our requirements: the theory

H representing a homomorphic property of the form $f(X \ ? \ Y) = f(X) \ ? \ f(Y)$ where ? where ? also obeys certain other properties, e.g. it is an ? is also an Abelian group operator. We call that theory AGH. In that case, Maude-NPA's built-in unification algorithm for homomorphic encryption (see Section 4.4.1) cannot be applied. AGH theory is a property belonging to a number of different cryptographic algorithms, starting with RSA in the late 70s. From early on it was realized to have a number of potential applications, including anonymous payment systems, computation on encrypted data, and voting. Since AGH goes beyond the built-in homomorphic unification algorithm, several under and over approximations of AGH were reported in [42]. All satisfied the properties on variant equations required by Maude-NPA.

11 Known Limitations and Future Work

In this section we describe some known limitations, along with the work we plan to do in the future to address them. Where workarounds exist, we also describe those.

1. In some cases, the automatic grammar generation fails to terminate, although these cases are rare. At this point, the only way of addressing this problem is to specify the initial grammars oneself, instead of having the tool generate them. How this is done is explained in Appendix C. In practice, sometimes the process fails to terminate because some Dolev-Yao strands are missing in the specification.

2. In other cases, although the grammar generation terminates, it takes a long time, and it is tedious to recompute every time a specification is loaded. One can save the grammars by first reducing the `genGrammars` command in Maude, and copying and pasting the results to the specification. How this is done is described in Appendix C.

3. In some cases grammars may fail to capture infinite paths in which the attacker attempts to find every more complex terms, without interacting with the honest principals. This is especially likely in theories involving Abelian groups, including exclusive-or, and it is a limitation of our current grammar generation algorithm, which was not originally designed with AC theories in mind. In the future we plan to improve these grammars.

4. If the requirements of a user-defined equational theory given in Section 4.5 are not satisfied, Maude-NPA may exhibit non-terminating and/or incomplete behavior, and any completeness claims about the results of the analysis can no longer be guaranteed. Using under- and over-approximations of the theories which satisfy the properties required by Maude-NPA can sometimes help us in this situation.

5. In some protocols, the number of unifiers produced for each step of the backwards search, although finite, is so large that the search space becomes extremely large in width and exhausts the resources of the machine running

Maude-NPA. This is currently happening for several protocols using exclusive-or, see Section 10.7. It is possible to obtain a reduced number of unifiers by a judicious use of sorts. For more details on this, see Appendix A and [17] for bounded associativity. In many cases, the extra unifiers are redundant and can be safely discarded by using sort information.

References

[1] The Maude Formal Environment. Available at `https://code.google.com/archive/p/maude-formal-environment/`.

[2] María Alpuente, Santiago Escobar, and José Iborra. Termination of narrowing revisited. *Theoretical Computer Science*, 410(46):4608–4625, 2009.

[3] Siva Anantharaman, Hai Lin, Christopher Lynch, Paliath Narendran, and Michaël Rusinowitch. Cap unification: application to protocol security modulo homomorphic encryption. In *ASIACCS*, pages 192–203. ACM, 2010.

[4] Franz Baader and Klaus U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *CADE*, volume 607 of *LNCS*, pages 50–65. Springer, 1992.

[5] Christopher Bouchard, Kimberly A. Gero, Christopher Lynch, and Paliath Narendran. On forward closure and the finite variant property. In *FroCos*, pages 327–342, 2013.

[6] Andrew Cholewa, Jose Meseguer, and Santiago Escobar. Variants of variants and the finite variant property. Technical report, University of Illinois at Urbana-Champaign, http://hdl.handle.net/2142/47117, 2014.

[7] 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.

[8] Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: How to get rid of some algebraic properties. In Jürgen Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.

[9] N. Dershowitz, S. Mitra, and G. Sivakumar. Decidable matching for convergent systems (preliminary version). In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*, pages 589–602. Springer, 1992.

[10] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. Built-in variant generation and unification, and their applications in maude 2.7. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 183–192. Springer, 2016.

[11] Francisco Durán, Steven Eker, Santiago Escobar, José Meseguer, and Carolyn L. Talcott. Variants, unification, narrowing, and symbolic reachability in maude 2.6. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 31–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[12] Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher Lynch, Catherine A. Meadows, José Meseguer, Paliath Narendran, Sonia Santiago, and Ralf Sasse. Effective symbolic protocol analysis via equational irreducibility conditions. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, volume 7459 of *Lecture Notes in Computer Science*, pages 73–90. Springer, 2012.

[13] Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher Lynch, Catherine A. Meadows, José Meseguer, Paliath Narendran, Sonia Santiago, and Ralf Sasse. Asymmetric unification: A new unification paradigm for cryptographic protocol analysis. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 231–248. Springer, 2013.

[14] 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.

[15] S. Escobar, D. Kapur, C. Lynch, C. Meadows, J. Meseguer, P. Narendran, and R. Sasse. Protocol analysis in Maude-NPA using unification modulo homomorphic encryption. In Michael Hanus, editor, *13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP 2011)*. ACM Press, 2011.

[16] 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.

[17] 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.

[18] 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.

[19] Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, LNCS vol. 5705, pages 1–50. Springer, 2009.

[20] Santiago Escobar, Catherine Meadows, José Meseguer, and Sonia Santiago. Sequential protocol composition in Maude-NPA. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *ESORICS*, volume 6345 of *Lecture Notes in Computer Science*, pages 303–318. Springer, 2010.

[21] Santiago Escobar, Catherine A. Meadows, José Meseguer, and Sonia Santiago. State space reduction in the Maude-NRL protocol analyzer. *Information and Computation*, 238:157–186, 2014.

[22] Santiago Escobar, Catherine A. Meadows, José Meseguer, and Sonia Santiago. Symbolic protocol analysis with disequality constraints modulo equational theories. In Chiara Bodei, Gian Luigi Ferrari, and Corrado Priami, editors, *Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, volume 9465 of *Lecture Notes in Computer Science*, pages 238–261. Springer, 2015.

[23] Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. *The Journal of Logic and Algebraic Programming*, 81(7-8):898–928, 2012.

[24] F. J. Thayer Fabrega, J. Herzog, and J. Guttman. Strand Spaces: What Makes a Security Protocol Correct? *Journal of Computer Security*, 7:191–230, 1999.

[25] Antonio González-Burgueño, Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. Analysis of the IBM CCA security API protocols in maude-npa. In Liqun Chen and Chris J. Mitchell, editors, *Security Standardisation Research - First International Conference, SSR 2014, London, UK, December 16-17, 2014. Proceedings*, volume 8893 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 2014.

[26] Antonio González-Burgueño, Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. Analysis of the pkcs#11 API using the maude-npa tool. In Liqun Chen and Shin'ichiro Matsuo, editors, *Security Standardisation Research - Second International Conference, SSR 2015, Tokyo, Japan, December 15-16, 2015, Proceedings*, volume 9497 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2015.

[27] J. D. Guttman, J. C. Herzog, V. Swarup, and F. J. Thayer. Strand spaces: From key exchange to secure location. In Carolyn Talcott, editor, *Workshop on Event-Based Semantics*, 2008. Position papers available at `http://blackforest.stanford.edu/eventsemantics/`.

[28] Joe Hendrix and José Meseguer. Order-sorted equational unification revisited. *Electr. Notes Theor. Comput. Sci.*, 2008. To appear in Proc. of RULE 2008.

[29] Jean-Marie Hullot. Canonical forms and unification. In Wolfgang Bibel and Robert Kowalski, editors, *Proceedings, Fifth Conference on Automated Deduction*, pages 318–334. Springer-Verlag, 1980. LNCS, Volume 87.

[30] 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.

[31] Zhiqiang Liu and Christopher Lynch. Efficient general unification for XOR with homomorphism. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2011.

[32] Christopher Lynch and Catherine Meadows. On the relative soundness of the free algebra model for public key encryption. In *Workshop on Issues in Theory of Security 2004*, 2004.

[33] Christopher Lynch and Catherine Meadows. Sound approximations to Diffie-Hellman using rewrite rules. In *Proceedings of the International Conference on Information and Computer Security (ICICS)*. Springer-Verlag, 2004.

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

[35] Catherine 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] Jose Meseguer. Strict coherence of conditional rewriting modulo axioms. Technical report, University of Illinois at Urbana-Champaign, Available at `http://hdl.handle.net/2142/50288`, 2015.

[37] Jonathan Millen. On the freedom of decryption. *Information Processing Letters*, 86(3), 2003.

[38] Sonia Santiago, Santiago Escobar, Catherine A. Meadows, and José Meseguer. A formal definition of protocol indistinguishability and its verification using

Maude-NPA. In Sjouke Mauw and Christian Damsgaard Jensen, editors, *Security and Trust Management - 10th International Workshop, STM 2014, Wroclaw, Poland, September 10-11, 2014. Proceedings*, volume 8743 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2014.

[39] Ralf Sasse, Santiago Escobar, Jose Meseguer, and Catherine Meadows. Protocol analysis modulo combination of theories: A case study in Maude-NPA. In *6th International Workshop on Security and Trust Management (STM'2010), Revised Selected Papers*. Springer, 2010.

[40] Stuart Stubblebine and Catherine 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.

[41] E. Viola. E-unifiability via narrowing. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *Theoretical Computer Science, 7th Italian Conference, ICTCS 2001, Torino, Italy, October 4-6, 2001, Proceedings*, volume 2202 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2001.

[42] Fan Yang, Santiago Escobar, Catherine A. Meadows, José Meseguer, and Paliath Narendran. Theories of homomorphic encryption, unification, and the finite variant property. In Olaf Chitil, Andy King, and Olivier Danvy, editors, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 123–133. ACM, 2014.

[43] Fan Yang, Santiago Escobar, Catherine A. Meadows, José Meseguer, and Sonia Santiago. Strand spaces with choice via a process algebra semantics. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 76–89, 2016.

A Equational Unification

Section 4 describes the equational theories supported by Maude-NPA. It gives an overview of the three possibilities for equational theories allowed in Maude-NPA: (i) axioms, (ii) variant equations, and (iii) dedicated unification algorithms. This appendix provides further information on how unification modulo these theories is implemented in Maude-NPA.

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, 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. The following sections explain several things regarding T-unification in the Maude-NPA:

- Unification modulo equational axioms for which the Maude-NPA provides built-in support for equational unification; different symbols with any combination of associativity (A), commutativity (C), and identity (U).
- Narrowing-based equational unification in general, which is however unfeasible for Maude-NPA analysis when the number of unifiers generated is infinite; and
- The most general case of equational theories satisfying the finite variant property for which the Maude-NPA can currently support unification by narrowing, with the important requirement of the number of unifier solutions being *finite*, namely, the admissible theories described in Section 4.5.
- Specialized unification algorithms also provided for theories that cannot be handled by narrowing and that do not have built-in support in Maude.
- Integration of different unification algorithms.

A.1 Built-in support for Unification Modulo Equational Axioms

Maude-NPA has built-in support for unification modulo certain equational theories T thanks to the underlying Maude infrastructure [11]. Specifically, Maude-NPA automatically supports unification modulo T for T any order-sorted theory of the form $T = (\Sigma, Ax)$, where Ax is a collection of equational *axioms* where some binary operators f in the signature Σ may have any combination of the following axioms in Ax

- *associativity*[14] (A) represented as $f(x, f(y, z)) = f(f(x, y), z)$,
- *commutativity* (C) represented as $f(x, y) = f(y, x)$,

[14]The computed set of unifiers is always finite but sometimes may be incomplete, see Section A.1.1.

- identity (U) represented as $f(x, e) = x$ and $f(e, x) = x$ (though only one is necessary if f is commutative).

As already illustrated in Section 4.1, the way associativity, and/or commutativity, and/or identity 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` and/or `id:` attributes. For example a function symbol `f` of sort `S` which is associative, commutative, and with identity element `nil` is specified in Maude as follows:

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

A.1.1 Limited built-in associative unification

The reader should be aware that associativity is supported, but with certain restrictions.

For the associative theory there are unification problems for which no finite complete set of unifiers exists. Thus, the Maude algorithm only computes unifiers up to a certain bound that is calculated internally. It can, however, verify whether the set of unifiers is complete or not. If the set of unifiers is not complete, Maude displays a warning of the following form in the screen, where `F` is replaced by the associative symbol with an incompleteness problem.

```
Warning: Unification modulo the theory of operator F has encountered
an instance for which it may not be complete.
```

Maude-NPA will continue to operate normally. The user should simply understand that the search space explored by Maude-NPA is incomplete and it may be the case that, for the given attack pattern, there is an initial state that the tool cannot find. However, (i) any initial state found by the tool corresponds to finding a real attack and (ii) if the tool does not output such a warning, the analysis *is* complete. Appendix B describes an example of an simple protocol displaying the above warning during search.

A.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 and/or identity axioms, for which the Maude-NPA provides automatic built-in support.

In this regard, a very useful, generic method to obtain T-unification algorithms is *narrowing* [29, 30]. In order for narrowing to provide a T-unification algorithm, the theory T has to be of the form $T = (\Sigma, E \uplus Ax)$, where Ax is a collection of equational axioms such as our previous combinations of associativity and/or commutativity and/or identity axioms for which a *finitary* Ax-unification algorithm exists (that is, any Ax-unification problem has a finite number of unifiers providing a complete set of solutions), and E is a collection of equations that, as rewrite rules, are:

1. confluent modulo Ax
2. terminating modulo Ax, and
3. strictly coherent modulo Ax (see [36]).

The precise meaning of these three requirements was explained in detail in Section 4.5.

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 $T = (\Sigma, E \uplus Ax)$ satisfying requirements (1)–(3) above yields 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 *in finite time* by narrowing. We discuss this extra important requirement in what follows.

A.3 Narrowing-Based Equational Unification in the Maude-NPA

Sufficient conditions for narrowing-based T-unification to provide a finite, complete set of solutions are known. For example, for the case when $T = (\Sigma, E \cup Ax)$ and $Ax = \emptyset$ such sufficient conditions go back to [29, 9]; see [2] for a recent survey. The case when $Ax \neq \emptyset$ is considerably more challenging (see, e.g., [8, 41, 18, 23]). The essential condition for Maude-NPA is the *finite variant property* [8]. As illustrated by the examples in this manual, this condition is satisfied by a number of useful cryptographic theories.

The T-unification algorithm that it is used then T has the finite variant property and Ax has a finitary unification algorithm is called *folding variant narrowing*. It is defined and explained in [23] and is efficiently implemented in Maude 2.7. Maude-NPA uses folding variant narrowing unification of theories satisfying the finite variant property to perform equational unification modulo the protocol equational theory.

As already mentioned in Sections 4.5 and 4.6, the Maude-NPA's support for order-sorted specifications is very helpful in achieving a finite complete set of unifiers by narrowing. This is because unification problems that may have an infinite number of unifiers in an untyped setting can sometimes have only a finite set of unifiers in a setting with types and subtypes. The key reason is that many of the untyped unifiers do not even typecheck. In the Maude-NPA, order-sortedness can sometimes be directly used to one's advantage to obtain theories T that have finitary T-unification algorithms. Furthermore, order-sortedness can greatly help in having smaller search spaces for symbolic reachability, since many unifiers that would have to be explored in an untyped setting are weeded out by the inherent type checking of order-sorted unification.

A.4 Unification for Homomorphic Encryption over a Free Operator

Although the ease of implementation of narrowing-based unification makes it very useful for exploration and experimentation, and interesting cryptographic theories satisfy the finite variant property, ultimately we also want to be able to make use of more efficient special-purpose algorithms. Moreover, there is a class of equational theories that appears prominently in cryptographic protocols applied to privacy-preserving computation: operators that are homomorphic with respect to another, e.g., $q(X * Y) = q(X) * q(Y)$. Theories like these can be shown to lack the finite variant property whether or not $*$ is a free operator or obeys the axioms for an Abelian group.[15] In these cases narrowing-based unification does not provide a finitary E-unification algorithm, and we must seek a different method[16].

An algorithm for unification modulo the homomorphic encryption theory E_h defined by the single oriented equation $e(X;Y,Z) \rightarrow e(X,Z);e(Y,Z)$ in a signature containing symbols e, $\_;\_$, and uninterpreted function symbols was given in [3]. The inference rules given in [3] were proved to be sound, complete, and terminating, meaning that all solved forms created by the algorithm are correct solutions of the unification problem and that for every solution of the unification problem there is a more general solution created by the algorithm. This algorithm was implemented in Maude using the metalevel facilities, providing an algorithm parametric on the symbols e and $\_;\_$ chosen by the user. The implementation and its integration into Maude-NPA were described in [15].

The implemented algorithm was untyped and an order-sorted version of the unification algorithm was automatically derived following the methodology proposed in [28]. This methodology applies a general algorithm by which, under mild conditions on the theory E, an order-sorted E-unification algorithm can be automatically obtained by: (i) associating to E its unsorted version $\bar{E}$; (ii) computing a complete set of (unsorted) $\bar{E}$-unifiers for the given E-unification problem; and (iii) typing and filtering out the unsorted $\bar{E}$-unifiers to obtain a complete set of order-sorted E-unifiers using the generic *sort propagation algorithm* described in [28]. This algorithm has also been integrated into the Maude-NPA infrastructure but can also be applied in any other contexts in which one wants to derive an order-sorted unification algorithm from its unsorted version.

Finally, we combine E_h-unification with a typed version of ACU-unification. The latter is needed because Maude-NPA states are multisets of terms, which are associative-commutative and have the empty multiset as the identity. This combination is supported by Maude-NPA by means of an order-sorted variant of the standard combination method for disjoint theories à la Baader and Schultz [4], so that in the end typed $E_h \cup ACU$-unification is achieved. A more complete description of how this is done is given in [39].

[15] Comon and Delaune only prove the result for the exclusive-or case in [8], but their proof can easily be extended to the other cases.

[16] See, however [42] for alternative specifications of homomorphic encryption that *do* have the finite variant property and have been used in Maude-NPA to analyze various protocols.

A.5 Integration of Different Equational Unification Algorithms

Integrating equational unification into protocol analysis is challenging for several reasons. First of all, in principle we need to have a different $E_{\mathcal{P}}$-unification algorithm for each protocol $\mathcal{P}$; second, experience with the Maude-NPA tool has shown the great advantages (typically leading to a much smaller search space) of *typed unification*, where variables have types (or *sorts*) and types can be arranged in *subtype hierarchies*; for example, to properly specify a protocol we may wish to distinguish different subtypes —e.g., for nonces, keys, or principal names— of a general type for messages; third, we often need to *combine* several such unification algorithms, for example when composing together various subprotocols or taking into account the associative-commutative-identity (ACU) axioms of the state constructors (see Section 5). This is made even more challenging by the fact that, in order to allow the option of verifying different kinds of implementations (e.g. the case in which a key is indistinguishable from a nonce), typing is mostly left to the discretion of the user.

Given the wide range of protocols and protocol combinations that need to be analyzed, a *modular* approach to the development of $E_{\mathcal{P}}$-unification algorithms is very much needed. Such a modular approach and its necessary infrastructure are now under development. Besides using the known techniques for combining unification algorithms for disjoint theories à la Baader and Schultz [4], Maude-NPA employs a more general methodology and associated tool infrastructure (in the Maude-NPA) in which unification algorithms can be combined and developed at three different levels and in a not necessarily disjoint way: (i) a basic library of commonly occurring theories and their combinations —currently including combinations of associativity, commutativity, and identity symbols— is efficiently supported by the Maude tool at the C++ level; (ii) unification algorithms for special-purpose cryptographic theories can be developed in a declarative way in Maude itself using its metalevel facilities as done in Sections 4.4.1 and A.4 for the homomorphic encryption theory E_h and in Section 4.4.2 for exclusive-or; and (iii) it is often possible to decompose an equational theory $E_{\mathcal{P}}$ as a disjoint union $E_{\mathcal{P}} = E \uplus Ax$, (where E and Ax may share some function symbols), and where a dedicated Ax-unification algorithm exists. If E is viewed as a set of rewrite rules that is convergent, coherent and has the finite variant property modulo Ax, *folding variant narrowing modulo* Ax with the equations E oriented into rules provides a finitary $E_{\mathcal{P}}$-unification algorithm [23].

B Protocol using associativity with incomplete search

The last version of Maude-NPA allows unification modulo associativity, as described in Section 4.1 and Appendix A.1.1, but it is not always able to return a complete set of most general unifiers for every unification problem. This appendix describes an example of a simple protocol using associativity that displays an incompleteness warning encountered during search.

Let us consider a simple protocol with an associative unification call during the

search space that has an infinite number of unifiers, provoking Maude to display the incompleteness warning and returning just a finite set of unifiers.

Let us reuse the same operator declaration than NSPK with associativity given in Section 10.5.

```
sorts Name Nonce NNSet .
subsort Name Nonce NNSet < Msg .
subsort Name < Public .
subsort Name Nonce < NNSet .

op pk : Name Msg -> Msg [frozen] .
op sk : Name Msg -> Msg [frozen] .

op _;_ : Msg Msg -> Msg [assoc frozen] .

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

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

The simple protocol is described as follows.

```
eq STRANDS-PROTOCOL
 = :: nil :: *** Initiator ***
   [ nil | -(pk(A,Z)),
           +(pk(B,Z)),
           -(pk(A,Y)),
           +(pk(B,Y)), nil ] &
   :: r :: *** Responder ***
   [ nil | -(pk(B,X)),
           +(pk(A,n(b,r) ; X)),
           -(pk(B,X ; n(b,r))), nil ]
[nonexec] .
```

The initiator is just a dummy strand that receives some message Z (for example from a server not specified in this protocol), forwards it to the responder, waits for another message Y coming from the responder, and forwards the same message Y to the responder. However, the responder strand is the one causing an incomplete associative unification problem because it receives some message X, adds a nonce before X, forwards the concatenation to the initiator, and finally waits to receive the same message X but concatenated with the nonce at the end instead of before it.

When we provide the following attack pattern

```
eq ATTACK-STATE(0)
 = :: r ::
   [ nil, -(pk(B,X)), +(pk(A,n(b,r) ; X)), -(pk(B,X ; n(b,r))) | nil ]
```

```
    || empty
    || nil
    || nil
    || nil
  [nonexec] .
```

Maude displays the following warning during the search

```
reduce in MAUDE-NPA : summary(1) .
Warning: Unification modulo the theory of operator _;_ has encountered
an instance for which it may not be complete.
result Summary: States>> 10 Solutions>> 0
```

This protocol provokes an associative unification call of the form `E ; X =? X ; E`, where `X` is a list variable but `E` is just an element variable. These unification calls have an infinite family of most general unifiers $\{\texttt{X} \mapsto \texttt{E}^n\}$ for $\texttt{E}^n$ being a list of n consecutive copies of the `E` element. This means that the search space is infinite in width at depth 1 for the attack pattern above and Maude returns just a finite subset of that infinite set, allowing only a partial exploration of the protocol search space. See Section 12.4.6 of the Maude manual available online at `http://maude.cs.uiuc.edu` for details on how the finite approximation of the infinite set of unifiers is calculated.

C Specifying Grammars

Grammars are used in Maude-NPA to eliminate various infinite search paths that can be provably guaranteed to never reach an initial state [16]. By an *initial grammar* we mean a grammar conjecturing a set of unreachable states. The conjecture of an initial grammar does not have to be correct and is just an initial guess. Instead, a *final grammar* is a grammar that has been checked by the Maude-NPA to correctly generate a set of states whose elements are all unreachable. Final grammars are generated iteratively by the Maude-NPA from initial grammars. The default in Maude-NPA is to generate both the initial and final grammars *completely automatically*, at the beginning of the first attack search after a specification is loaded (see Section 7 for a description of how to perform attack searches). The intent is for grammars to be completely transparent to the user. However, there are cases in which the user may want to reuse grammars, add initial grammars, or replace the initial grammars generated by the Maude-NPA with his or her own ones. We describe how to do all this below. We also describe the user-defined initial grammar used to cut down the search space of the attach search of the Diffie-Hellman protocol discussed in Section 10.6.

C.1 Reusing Grammars

The generation of grammars may be time-consuming, and the user may want to avoid having to do this every time a specification is reloaded. This can be avoided

by adding the grammars to the specification. One first displays the grammars by reducing the `displayGrammars` constant in Maude-NPA typing:

```
red displayGrammars .
```

Maude-NPA will then produce output of the form

```
result GrammarList:

... Grammars ...
```

To reuse the grammars displayed by Maude-NPA in this way in a subsequent execution of the protocol, the user should "cut and paste" these grammars in an equation of the form:

```
eq GENERATED-GRAMMARS =

... Grammars ...

[nonexec] .
```

where `...Grammars..` is the text that was generated by the `genGrammars` command. The `GENERATED-GRAMMARS` equation is added to the module `PROTOCOL-SPECIFICATION` in the general template described in Section 5, right before the attack state specifications. Maude-NPA will now treat these as the generated grammars and will not attempt to generate any grammars of its own.

Note that the grammar generation can fail to generate a grammar for a concrete initial grammar (either an internally generated initial grammar or a user-defined initial grammar, which are described below). Such failed grammars are not included in the `displayGrammars` command. In order to obtain the whole list of generated grammars, the user should use the `genGrammars` command:

```
red genGrammars .
```

Failure grammars are identified by terms starting with `errorNoHeuristicApplied`, `errorIntegratingExceptions`, and `errorInconsistentExceptionsInGrammarRule`.

Failure grammars cannot be included in the `GENERATED-GRAMMARS` equation.

C.2 Adding New Initial Grammars

There are still some cases in which the initial grammars generated by Maude-NPA are not sufficient. In such a case the user can add his or her own initial grammars. For example, the Diffie-Hellmman protocol specified in Section 10.6 requires the following initial grammar, which is not yet automatically generated by Maude-NPA:

```
grl empty => (NS * n(a,r)) inL . ;
grl empty => n(a,r) inL . ;
grl empty => (NS * n(b,r)) inL . ;
grl empty => n(b,r) inL .
```

This initial grammar indicates that the concrete nonces generated by the initiator and the responder cannot be learned by the intruder, independently of whether they are combined with other nonces.

This can be done by adding an `EXTRA-GRAMMARS` equation to the `PROTOCOL-SPECIFICATION` module of the three-module template and specifying the initial grammars there as the value of `EXTRA-GRAMMARS`, as in the following:

```
eq EXTRA-GRAMMARS
  = (grl empty => (NS * n(a,r)) inL . ;
     grl empty => n(a,r) inL . ;
     grl empty => (NS * n(b,r)) inL . ;
     grl empty => n(b,r) inL .
     ! S2 )
  [nonexec] .
```

Originally, initial grammars consisted of the definition of a single term (called the *seed-term*) but, as we can see from the example above, an initial grammar can now be any syntactically correct grammar. Giving a complete set of directives on writing grammars is beyond the scope of this document, but we give a BNF specification of grammars in Appendix D. Strategies `S1` or `S2` for grammar generation are chosen depending on the conditions (`empty` requires strategy `S2`, whereas (`Msg notInI` requires strategy `S1`). If the user wants to see what the initial grammars generated by Maude-NPA look like, this can be done by reducing the expression `genGrammars(0)` in Maude (i.e., typing "`red genGrammars(0) .`"), where `0` indicates the number of grammar generation steps allowed and `unbounded` is the constant used in regular grammar generations.

We note that `GENERATED-GRAMMARS` has precedence over `EXTRA-GRAMMARS`. If both are found in a specification, `GENERATED-GRAMMARS` will be used and `EXTRA-GRAMMARS` will be ignored.

C.3 Replacing Maude-NPA Initial Grammars

In some cases one may want to replace the Maude-NPA initial grammars entirely. In this case, one uses `INITIAL-GRAMMARS` but enters one's own initial grammar specifications, following Appendix D, instead of the ones generated by Maude-NPA. This feature is only recommended for debugging Maude-NPA.

D Grammar BNF Syntax

In this Appendix we give a BNF specification of the syntax of Maude-NPA grammars. For a more complete discussion of grammars and how they work, see [16].

```
GrammarSpecList -> GrammarSpec | GrammarSpec "|" GrammarSpecList
GrammarSpec -> "(" Grammar "!" Strategy ")"
Strategy -> "S1" | "S2"
Grammar -> GrammarRule | GrammarRule ";" Grammar
```

```
GrammarRule _-> "grl" Conditions "=>" Term "inL ."
Conditions -> "empty" | Condition | Condition "," Conditions
Condition -> Term "notInI" | Term "inL" | Term "notLeq" Term
```

We do not provide a BNF definition of the production `Term`; that is just any term of sort `Msg` specifiable in the user-defined protocol syntax.

E Commands Useful for Debugging

The following commands are mainly useful for debugging Maude-NPA; we include them for the sake of completeness.

For debugging purposes, it is possible to disable optimization techniques and validity checks on the data selectively. A detailed description of all the optimization and validity checks is available in [21].

One adds another argument to the run or summary command, which includes the optimization techniques to be disabled. For example, if one wants to disable grammars and the inconsistency optimization techniques (the latter marks as unreachable states that violate certain consistency properties while looking for the second state in a backwards search), this is given as follows:

```
red run(0,2,-grammars -inconsistency) .
```

The optimization techniques that can be turned off are the following

1. `-grammars` turns off the grammars.
2. `-inconsistency` turns off the following inconsistency check:

   A state St containing two contradictory facts (t `inI`) and (t `!inI`) for a term t.
3. `-inputAndNotLearned` turns off the following inconsistency check:

   A state St whose intruder knowledge contains the fact (t `!inI`) and a strand of the form $[m_1^{\pm}, \ldots, t^{-}, \ldots, m_{j-1}^{\pm} \mid m_j^{\pm}, \ldots, m_k^{\pm}]$.
4. `-alreadySent` turns off the following inconsistency check:

   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., $(r_1, \ldots, r, \ldots, r_k : \mathsf{Fresh})$ $[m_1^{\pm}, \ldots, m_{j-1}^{\pm} \mid m_j^{\pm}, \ldots, m_k^{\pm}]$, cannot produce r, i.e., r is not a subterm of any output message in $m_1^{\pm}, \ldots, m_{j-1}^{\pm}$.
5. `-secretData` turns off the following inconsistency check:

   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.

6. `-implication` turns off the transition subsumption
7. `-equationalRed` turns off the check that negative terms are irreducible w.r.t. the equational theory
8. `-freshInstantiated`, turns off the check that fresh variables are never instantiated
9. `-inputFirst` turns off the optimization to give priority to input messages
10. `-variantsBefore` turns off the generation of the variants of a state before a new backwards narrowing step
11. `-variantsAfter` turns off the generation of the variants of a state after a new backwards narrowing step, and before the optimizations are applied
12. `-simplifyDiff` turns off the simplification of disequality constraints
13. `-inconsistencyDiff` turns off the inconsistency check on disequality constraints
14. `-variantDiff` turns off the generation of the variants of disequality constraints
15. `-removetDiff` turns off the removal of trivially true or false disequality constraints
16. `-ghost` turns off the super-lazy intruder.
17. `-never` turns off the never patterns appearing in attack states.
18. `noopt` turns off all of the optimizations. Note that `-` is not used here.