Maude-NPA, Version 1.0

Santiago Escobar
sescobar@dsic.upv.es
Technical University of Valencia
Valencia, Spain

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

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

Contents

1 Introduction **3**

2 Setting Up and Using Maude-NPA **3**

3 Protocol Specifications **4**
3.1 Specifying the Protocol Syntax . . . 5
3.1.1 Sorts and Subsorts . . . 6
3.2 Algebraic Properties . . . 8
3.2.1 The `PROTOCOL-EXAMPLE-ALGEBRAIC` module . . . 8
3.2.2 Specifying C and AC Theories . . . 9
3.2.3 General Requirements for Algebraic Theories . . . 11
3.2.4 Some Examples of Admissible Theories . . . 14
3.3 Protocol Specification . . . 17
3.3.1 Protocol Specification Variables . . . 17
3.3.2 Dolev-Yao Rules . . . 17
3.3.3 Adding/deleting operations in the Dolev-Yao Strands . . . 18
3.3.4 Protocol Rules . . . 19
3.3.5 Attack States . . . 20
3.3.6 Attack States With Excluded Patterns: Never Patterns . . . 22
3.4 Grammars . . . 25

4 Maude-NPA commands for attack search **25**

5 Two More Examples **27**
5.1 Diffie-Hellman protocol . . . 27
5.2 Tiny xor protocol . . . 35

6 Known Limitations and Future Work **37**

References **37**

A Narrowing-based Finitary Equational Unification **39**
A.1 Built-in support for Unification Modulo Equational Axioms . . . 39
A.2 Narrowing-Based Equational Unification and its Limitations . . . 40
A.3 Narrowing-Based Equational Unification in the Maude-NPA . . . 40

B Specifying Grammars **41**
B.1 Reusing Grammars . . . 41
B.2 Adding New Initial Grammars . . . 42
B.3 Replacing Maude-NPA Initial Grammars . . . 42

C Grammar BNF Syntax **43**

D Other Optimizations **43**

E Commands Useful for Debugging **44**
E.1 Excluding Optimizations and Checks 44

F Example Protocol files **45**
F.1 Needham-Schroeder public key protocol 45
F.2 Diffie-Hellman protocol 46
F.3 Tiny XOR protocol 48

1 Introduction

This document describes Version 1.0 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 included in other tools. These include cancellation of encryption and decryption, Abelian groups (including exclusive-or), and exponentiation. Maude-NPA uses an approach similar to the original NRL Protocol Analyzer; it is based on unification, and performs backwards search from a final state to determine whether or not it is reachable. Unlike the original NPA, it has a theoretical basis in rewriting logic and narrowing, and offers support for a wider basis of equational theories that includes associative-commutative (AC) theories. A description of Maude-NPA's formal foundations in rewriting logic, together with a soundness proof, is given in [6]. Descriptions of how Maude-NPA handles different equational theories are given in [7, 8].

This document is organized as follows. In Section 2 we describe the mechanics of loading Maude-NPA and Maude-NPA specifications. In Section 3 we describe how a protocol is specified in Maude-NPA, using the Needham-Schroeder public key protocol as an example. We also give instructions for specifying AC equational theories of interest, illustrating the ideas with two examples: exclusive-or and modular exponentiation. In Section A we explain in detail the types of equational theories that Maude-NPA supports, and describe how the user can ensure that the requirements are met. In Section 4 we describe the commands that can be used, again using NSPK as an example. In Section 5 we describe how the tool can be applied to two other examples involving AC theories, one involving exclusive or, and another the basic Diffie-Hellman protocol. In Section 6 we describe some known limitations of the tool and plans for further extensions. In several appendices we give some commands that are mainly used for debugging Maude-NPA, we describe how to specify grammars, which cut down many useless states, give a brief description of other state-space reduction techniques supported by the tool, and include the full specification of the protocol examples used in this document.

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 `maude.cs.uiuc.edu`, and also to the Maude book [2] for more detailed documentation on Maude-related matters.

2 Setting Up and Using Maude-NPA

We assume that the user has installed a copy of Maude that includes the implementation of order-sorted C and AC unification, e.g., version 2.4 or later. After starting Maude, the user must load Maude-NPA. To do this, the user must be in the Maude-NPA directory. The user should type the command

```
load maude-npa
```

All Maude-NPA specification files must end in the suffix `.maude`, e.g., `foo.maude`. Instructions for writing specifications are given in Section 3. In order to load a specification one uses the `cd` command to change to the directory that the specification is in, and then types the command

```
load foo
```

If the user does not want to change to the directory where `foo.maude` sits in, it is necessary to specify the directory path when loading it, e.g., `load examples/foo`.

Once `foo.maude` is loaded, it is possible to search for attacks using the commands described in Section 4.

3 Protocol Specifications

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

We give a template for any Maude-NPA specification below. Throughout, lines beginning with three or more dashes (i.e., `---`) or three or more asterisks (i.e., `***`) are comments that are ignored by Maude.

```
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
  --- * Every sort must be a subsort of Msg
  --- * No sort can be a supersort of Msg
  --- * Variables of sort Fresh are really fresh
  ---   and no substitution is allowed on them
  --- * Sorts Msg and Public cannot be empty
  ----------------------------------------------------------
endfm

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

endfm

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

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

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

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

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

endfm

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

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

3.1 Specifying the Protocol Syntax

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

3.1.1 Sorts and Subsorts

We begin by specifying *sorts*. In general, sorts are used to specify different types of data, that are used for different purposes. We have a special sort called `Msg` that represents what a message is going to look like in our protocol. If a protocol makes no additional sort distinctions, i.e., it is an unsorted protocol, there will be no extra sorts, and every symbol will be of sort `Msg`. If only keys can be used for encryption, we would want to have a sort `Key`, and specify that an encryption operator `e` can only take a term of sort `Key` as its first argument, e.g., "`op e :  Key Msg -> Msg`."

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

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

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

`Msg` All sorts defined by the user must 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`. It is not necessary to define symbols of sort `Fresh`, i.e., the sort `Fresh` can be empty.

`Public` The sort `Public` is used to identify terms that are publically available, and therefore assumed known by the intruder[1]. This sort cannot be empty.

We begin with the definition of sorts. We use the Needham-Schroeder Public Key Protocol (NSPK) as the running example. This protocol uses public key cryptography, and the principals exchange encrypted data consisting of names and nonces. Thus we will define sorts to distinguish names, keys, nonces, and encrypted data. This is specified as follows:

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

[1]The `Public` sort will be automatically deduced in future versions of the tool.

The sorts `Nonce` and `Enc` are not strictly necessary, but they can make the 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 `Enc`, it will not accept a term of sort `Nonce`; technically because `Enc` is not declared as a subsort of `Nonce`. If we are looking for type confusion attacks, we would not want to include these sorts, and instead would declare everything as having sort `Msg` or `Name`. See [7] for an example of a type confusion attack.

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

We begin with the public/private encryption operators.

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

The `frozen` attribute is technically necessary to tell Maude not to attempt to apply rewrites at arguments of those symbols[2]. The `frozen` attribute must be included in all operator declarations in Maude-NPA specifications, excluding constants. The use of sort `Name` as an argument of public key encryption may seem odd at first. But it is used because we are implicitly associating a public key with a name when we apply the `pk` operator, and a private key with a name when we apply the `sk` operator. A different syntax specifying explicit keys could have been chosen for public/private encryption.

Next we specify some principal names. 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., of sort `Name` in NSPK) instead of constants. However, we may have a need to specify constant principal names in a goal state. For example, if we have an initiator and a responder, and we are not interested in the case in which the initiator and the responder are the same, we can prevent that by specifying the names of the initiator and the responder as different constants. Also, we may want to identify the intruder's name by a constant, so that we can cover the case in which principals are talking directly to the intruder.

For NSPK, we have three constants of sort `Name`, `a`, `b`, and `i`.

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

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

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

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

Note that the nonce operator has an argument of sort `Fresh` to ensure uniqueness. The argument of type `Name` is not strictly necessary, but it provides a convenient way of identifying which nonces are generated by which principal. This makes searches more efficient, since it allows us to keep track of the originator of a nonce throughout a search. 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. This would require us to define a new sort `Role`, with a subsort inclusion `Role < Msg`, and use it as follows:

```
op init : -> Role .
op resp : -> Role .

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

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 ";" has to be parsed as associated to the left; whereas "`gather (E e)`" indicates association to the right.

3.2 Algebraic Properties

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

3.2.1 The `PROTOCOL-EXAMPLE-ALGEBRAIC` module

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

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

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

The `nonexec` attribute is technically necessary to tell Maude not to use an equation or rule within its standard execution, since it will be used only in the narrowing implementation[3]. The `nonexec` attribute must be included in all user-defined equation and rule declarations of the protocol. Furthermore, attribute `owise` (i.e., otherwise) cannot be used and no order of application is assumed on the algebraic properties.

3.2.2 Specifying C and AC Theories

Since Maude-NPA uses special unification algorithms for the case of having commutative (C) or associative-commutative (AC) algebraic properties, these are specified not as standard equations but as *axioms* in the operator declarations. For example, suppose that we want to specify exclusive-or. We specify an infix associative-commutative operator "`<+>`" in the `PROTOCOL-EXAMPLE-SYMBOLS` module as follows:

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

where the associativity and commutativity axioms are declared as attributes of the `<+>` operator with the `assoc` and `comm` keywords. Similarly, we would specify an operator that is commutative but not associative with the `comm` keyword alone. [4]

We specify the cancellation rules for `<+>` in the `PROTOCOL-EXAMPLE-ALGEBRAIC` module as follows[5]:

```
vars X Y : Msg .

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

If we want to include a Diffie-Hellman mechanism, we need two operations. One is exponentiation, and the other is modular multiplication. Since Diffie-Hellman is

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

[4]It is also possible to specify and operator that is associative but not commutative using the `assoc` keyword, but this is not advised, since associative unification is not finitary. A form of *bounded* associativity is possible, but this is done differently, see [7] for details.

[5]Note that the first equational property, i.e., `X <+> X <+> Y = Y`, is not part of the exclusive-or theory but it is necessary for coherence, see Section 3.2.3.

a commonly used algorithm in cryptographic protocols, we give a complete specification here.

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

```
sorts Name Nonce NeNonceSet Gen Exp Key GenvExp Enc Secret .
subsort Gen Exp < GenvExp .
subsort Name NeNonceSet GenvExp Enc Secret Key < Msg .
subsort Exp < Key .
subsort Nonce < NeNonceSet .
subsort Name < Public . --- This is quite relevant and necessary
subsort Gen < Public . --- This is quite relevant and necessary
```

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

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

There are several things to note about this Diffie-Hellman specification. The first is that, although modular multiplication has a unit and inverses, this is not included in our equational specification. Instead, we have only included the algebraic property that is necessary for Diffie-Hellman to work. The second is that we have specified types that will rule out certain kinds of intruder behavior. In actual fact, there is nothing that prevents an intruder from sending an arbitrary string to a principal and passing it off as an exponentiated term. The principal will then exponentiate that term. However, given our definition of the `exp` operator, only terms of type `GenvExp` can be exponentiated. This last restriction is necessary in order to assure that the unification algorithm is finitary. The details of this are explained in Section A.3. The omission of units and inverses is not necessary to ensure finitary unification, but rules out behavior of the intruder that is likely to be irrelevant for attacking the protocol, or that is likely to be easily detected (such as the intruder sending an `exp(g,0)`.

We note that if one is interested in obtaining a proof of security using these restrictive assumptions, one must provide a proof (outside of the tool) that security in the restricted model implies security in the more general model. This could be done along the lines of the proofs in [18, 15, 16].

3.2.3 General Requirements for Algebraic Theories

The Maude-NPA performs symbolic reachability analysis *modulo* the equational theory of the protocol. This makes Maude-NPA verification much stronger than verification methods based on a purely syntactic view of the algebra of messages as a term algebra using the standard Dolev-Yao model of perfect cryptography in which no algebraic properties are assumed. Indeed, it is well-known (see, e.g., [19]) 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.

As explained in Appendix A, the symbolic reachability analysis performed by Maude-NPA uses a technique called *narrowing* to perform unification of symbolic terms *modulo* the algebraic properties of the protocol. In order for this narrowing technique to provide a *finite* set of unifiers, five specific requirements must be met by any algebraic theory specifying cryptographic functions that the user provides. If these requirements are not satisfied, Maude-NPA may exhibit non-terminating and/or incomplete behavior, and any completeness claims about the results of the analysis cannot be guaranteed. We list below these five requirements, explain in detail what they mean, and show in Section 3.2.4 how they are all met by the examples presented in Section 3.2.2.

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 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` attributes; they are *not* declared as explicit equations. Instead, the equations E are declared with the `eq` keyword as we have 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 five requirements:

1. The axioms Ax can declare some binary operators in Σ to be commutative (with the `comm` attribute), or associative-commutative (with the `assoc` and `comm` attributes). No other combinations of axioms are allowed; that is, a function symbol has either no attributes, or only the `comm` attribute, or only the `assoc` and `comm` attributes.[6]

2. The equations E are confluent modulo Ax.

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

3. The equations E are terminating modulo Ax.

4. The equations E are coherent modulo Ax (see [14]).

5. The equations E are strongly right irreducible.

We now explain in detail what these requirements mean. Since Ax-unification is supported for the combinations of axioms Ax described in (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 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 and commutativity. 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.

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 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

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

Rather than explaining the *coherence modulo Ax* notion in general (the precise definition of the general notion can found in [14]), we explain in detail its meaning in the case where it is needed for the Maude-NPA, namely, the case of associative-commutative (AC) symbols. The best way to illustrate the meaning of coherence is by its *absence*. Consider, for example, an exclusive or operator $\oplus$ which has been declared AC. Now consider the equation $x \oplus x = 0$. This equation, if not completed by another equation, is *not* coherent modulo AC. What this means is that there will be term *contexts* in which the equation *should* be applied, but it cannot be applied. Consider, for example, the term $b \oplus (a \oplus b)$. intuitively, we should be able to apply the above equation to simplify it 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 using also the equation $x \oplus 0 = x$ we can slightly simplify to the equation $x \oplus x \oplus y = y$. This second variant of our equation will now apply to the term $b \oplus (a \oplus b)$, giving the simplification $b \oplus (a \oplus b) \longrightarrow_{E/Ax} a$.

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

Given a theory $T = (\Sigma, E \cup Ax)$, which we assume satisfies conditions (1)–(4) above, we call the set E of equations *strongly right irreducible* iff for each equation $t = t'$ in E, we cannot further simplify by the equations E modulo Ax either the term t', or any substitution instance $\theta(t')$ where the terms in the substitution θ cannot themselves be further simplified by the equations E modulo Ax. Obvious cases of such righthand-sides t' include:

- a single variable;
- a constant for which no equations exist;
- a *constructor term*, that is, a term whose function symbols have no associated equations.

But these are not the only possible cases where strong right irreducibility can be applied. Typing, particularly the use of sorts and subsorts in an order-sorted equational specification, can greatly help in obtaining right irreducibility. We refer the reader to [7, 8] for two examples of how order-sorted typing helps narrowing-based unification to become finitary. We discuss one of these examples, namely Diffie-Hellman, in the following section.

3.2.4 Some Examples of Admissible Theories

Since any user of the Maude-NPA should write specifications whose algebraic theories are admissible, i.e., satisfy requirements (1)–(5) in Section 3.2.3, it may be useful to illustrate how these requirements are met by several examples. This can give a Maude-NPA user a good intuitive feeling for how to specify algebraic theories that the Maude-NPA currently can handle. For this purpose, we revisit the theories already discussed in Section 3.2.

Let us begin with the theory of Encryption/Decryption:

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

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

In this case $Ax = \emptyset$. It is obvious that in this case the equations E *terminate*, since the *size* of a term as a tree (number of nodes) strictly decreases after the application of any of the above two rules, and therefore it is impossible to have an infinite chain of rewrites with the above equations. It is also easy to check that the equations are *confluent*: by the termination of E this can be reduced to checking confluence of critical pairs, which can be easily discharged by automated tools, or even by hand. Since $Ax = \emptyset$, coherence is a mute point. The equations are also *strongly right irreducible*, because in both cases they are the variable `Z`, and any instance of `Z` by a term that cannot be further simplified by the above equations, obviously cannot be further simplified by hypothesis.

Let us now consider the Exclusive Or Theory:

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

vars X Y : Msg .

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

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

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

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

Turning now to the Diffie-Hellman theory we have:

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

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

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

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

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

we see that the order-sorted unification of the subterm `exp(W:Gen,Y:NeNonceSet)` (which has sort `Exp`) and the lefthand side now *fails*, because the sorts `Gen` and `Exp` are mutually exclusive and cannot have any terms in common. Therefore there are no nontrivial critical pairs and the equation is confluent modulo *AC*. *Coherence* modulo *AC* is trivially satisfied, because the top operator of the equation (`exp`) is not an *AC* operator. As in the case of confluence modulo *AC*, the remaining issue of *strong right irreducibility* becomes particularly interesting in the order-sorted context. Note that in an untyped setting, an instance of the righthand side by applying a substitution whose terms cannot be further simplified *could* itself be simplified. For example, if we consider the untyped righthand side term `exp(W, Y`

`* Z)`, the substitution θ mapping `W` to `exp(Q,X)` and being the identity on `Y` and `Z` is itself irreducible by the equations, but when applied to `exp(W, Y * Z)` makes the corresponding instance reducible by the untyped version of the above equation. However, in the order-sorted setting in which our equation is defined, the equation is indeed strongly right irreducible. This is again because the sorts `Gen` and `Exp` are mutually exclusive and cannot have any terms in common, so that the variable `W:Gen` cannot be instantiated by any term having `gen` as its top operator.

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

```
op h: Msg -> Msg .

vars X Y : Msg .

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

The problem now is that the righthand side `h(X) <+> h(Y)` *fails* to be strongly right-irreducible. For example, the substitution θ mapping `X` to `U <+> V` and `Y` to `Y` is itself irreducible, but produces the instance `h(U <+> V) <+> h(Y)`, which is obviously reducible. Since strong irreducibility is only a *sufficient condition* for narrowing-based equational unification to be finitary, one could in principle hope that this homomorphism example might still have a narrowing-based finitary algorithm[7]. However, the hopes for such a finitary narrowing-based algorithm are dashed to the ground by results in both [3], about the homomorphism theory not having the "finite variant" property, and the variant-based unification methods in [9].

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

As a final *caveat*, if the user specifies a theory T where any of the above conditions (1)–(5) fail, besides the lack of completeness that would be caused by the failure of conditions (2)–(4), a likely consequence of failing to meet condition (5) will be that the tool will loop forever trying to solve a unification problem associated with just a single transition step in the symbolic reachability analysis process. However, we are

[7]The fact that an equational theory T does not have a finitary narrowing-based algorithm does not by itself preclude the existence of a finitary unification algorithm obtained by other methods. In fact, the homomorphic theory we have just described does have a finitary unification algorithm [1]; however this dedicated unification algorithm is not an instance of a generic narrowing-based algorithm. However, as already explained, in the Maude-NPA the theories for which finitary unification is supported are either order-sorted theories with built-in axioms of commutativity and associativity-commutativity, or theories modulo such built-in axioms that are confluent, terminating, and coherent modulo Ax, and that are also strongly right irreducible.

investigating conditions more general than (5) (such as the above-mentioned finite variant property) that will still guarantee that a T-unification problem always has a finite complete set of solutions. Future versions of Maude-NPA will relax condition (5) to allow more general conditions of this kind.

3.3 Protocol Specification

The protocol itself and the intruder capabilities are both specified in the `PROTOCOL-SPECIFICATION` module. They are specified using strands. A *strand*, first defined in [12], is a sequence of positive and negative messages[8] describing a principal executing a protocol, or the intruder performing actions, e.g.,

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

where a positive node implies sending, and a negative node implies receiving. However, in our tool each strand is divided into the past and future parts, and 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 line were sent or received in the past, whereas the messages to the right of the line will be sent or received in the future. Right before the strand, the variables $r_1, \cdots, r_i$ of sort `Fresh` are made explicit, as follows:

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

3.3.1 Protocol Specification Variables

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

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

3.3.2 Dolev-Yao Rules

After the variables are specified, the next thing to specify is the intruder, or Dolev-Yao rules. These specify the operations an intruder can perform. 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.

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

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

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

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

```
:: 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 rules to specify these:

```
:: 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 symbol `STRANDS-DOLEVYAO` as the repository of all the Dolev-Yao strands, and 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] .
```

3.3.3 Adding/deleting 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 rule. For each operation used in the protocol, we should consider whether or not the intruder can perform it, and produce 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 rules for all the operation that are defined, unless one is explicit making the assumption that the intruder can *not* perform the operation. It is also *strongly recommended* that operations not used in the protocol should not be provided Dolev-Yao strands. This is because the tool will attempt to execute these rules, even if they are useless, and so they will negatively affect its performance.

3.3.4 Protocol Rules

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

We recall the informal specification of NSPK, as follows:

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

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

In specifying protocol rules it is important to remember to specify them from the point of view of the principal executing the role. For example, in NSPK the initiator `A` starts out by sending her name and a nonce encrypted with `B`'s public key. She gets back something encrypted with her public key, but all she can tell is that it is her nonce concatenated with some other 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 for which the principal could not match its structure must be represented by a variable of sort `Msg`.

In order to represent this, we represent 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` by a variable `X` of sort `Msg`, which implies a bigger search space.

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

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

Note that, as explained above, the point here is to only include 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 that 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] .
```

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

3.3.5 Attack States

The last thing we specify are the *attack states*, which describe the final attack states we are looking for with Maude-NPA. Unlike the case of the Dolev-Yao and protocol strands, we can specify more than one attack state. Thus, we designate each attack state with a natural number.

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

```
  :: nil ::
  [ nil, -(#2:Msg ; 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, pk(b, n(b, #0:Fresh)) inI
||
+(n(b, #0:Fresh)), -(pk(b, n(b, #0:Fresh)))
||
nil
```

The intruder knowledge represents what the intruder knows (symbol `_inI`) or doesn't know (symbol `_!inI`) at each state. However, the symbol `_!inI` represents it is not known now, but it would be known in the future. 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 knowledge of the intruder (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 passed. 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 for the benefit of the user, and is not actually used in the backward search. Finally, the last part is used to contain information about the search space that the tool creates to help manage its search. It does not provide any information about the attack itself, and is currently only displayed by the tool to help in debugging. More information about this can be found in Appendix D.

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

For NSPK, the standard attack is represented as follows:

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

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

We can also specify inequalities (the condition that a term not be equal to something) in the intruder knowledge section. For example, suppose that we want to specify that a responder executes a strand, apparently with an initiator `a`, but the nonce received is not generated by `a`. This would be done as follows:

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

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

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

1. Strands in the attack state must have the bar at the end.

2. If more than one strand appears in the attack state, they must be separated by the `&` symbol. If more than one term appears in the intruder knowledge, they must be separated by commas. If no strands appear, or no terms appear, the `empty` symbol is used, in the strands or intruder knowledge sections, respectively, e.g.,

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

3. The items that can appear in the intruder knowledge may include not only terms known by the intruder, but also inequality conditions on terms.

4. The last two fields of an attack state must always be `nil`. These are fields that contain information that is built up in the backwards search, but is empty in the final attack state.

3.3.6 Attack States With Excluded Patterns: Never Patterns

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

Here is how we would specify an initiator strand without a responder in the NSPK protocol[9]:

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

The tool will now look for all paths in which the intruder strand is executed, but the corresponding responder strand is not.

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

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

We note that it is possible to put more than one never pattern in a search space, but then each such pattern must be contained in pair of parentheses, e.g.,

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

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

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

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

Note that adding `Never` patterns to reduce the search space, as distinguished from their use for verifying *authentication* properties, means that failure to find an attack does not necessarily 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.

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

1. The bar in any strand in the never pattern should be at the beginning of the strand. If it is not, we will enforce it.

2. Variables in a never pattern are never shared with the variables in the main attack state specification nor with variables in other never patterns.

3. The last two fields in a never pattern must be variables of type `SMsgList`, and `Ghostlist`, respectively, as given in the above.

4. The first two fields must end in variables of type `Strandset` and `Intruderknowledge`, respectively.

5. More than one never pattern can be used in an attack state. Each one must be delimited by its own set of parentheses.

For a good example of the use of never patterns, which makes the Maude-NPA search considerably more efficient without compromising the completeness of the reachability analysis, we refer the reader to the analysis of the Diffie-Hellman protocol in Section 5.1.

3.4 Grammars

The Maude-NPA's ability to reason well 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. Here we briefly explain how grammars work as a state space reduction technique and refer the reader to [17, 6] 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, or, at least, can drastically reduce the search space.

Unlike the grammars used in NPA, described in [17], and the version of Maude-NPA described in [6], in which initial grammars needed to be specified by the user, Maude-NPA now generates initial grammars automatically. Each initial grammar consists of a single seed term of the form $C \mapsto f(X_1, \cdots, X_n) \in \mathcal{L}$, 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., adding more seed terms. Appendix B gives a more detailed description of grammars and its features in the Maude-NPA.

4 Maude-NPA commands for attack search

The commands `run`, `summary`, and `initials` are the tool's commands for attack search. They are invoked by reducing them in Maude, that is, by typing `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, e.g.,

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

tells Maude-NPA to construct the backwards reachability tree up to depth 10 for the attack state designated with natural number 0. The command `run` yields the set of states found in the leaves of the backwards reachability tree of the specified depth that has been generated. When the user is not interested in the current states of the reachability tree, he/she can use the 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, when we give the reduce command `summary(0,2)` in Maude as below for the NSPK example, it returns:

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

The initial state representing the standard NSPK attack is found in 7 steps. That is, if we type

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

our tool outputs:

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

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

```
  red initials(0,14) .
```

for the NSPK example our tool outputs the attack[10]:

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

[10]Note that Maude-NPA associates an identifier, e.g. 1.5.2.7.1.4.3.1, to each state generated by the tool. These identifiers are for internal use and are not described in this manual.

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

This corresponds to the following textbook version of the attack:

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

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

We demonstrate with NSPK:

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

This tells us, that Maude-NPA terminated with only one attack. If we want to see what that attack is, we reduce the command `run(0,unbounded)` to get the attack displayed above.

5 Two More Examples

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

5.1 Diffie-Hellman protocol

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 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 sorts used in this protocol are as follows, where sorts `GenvExp`, `Gen`, and `Exp` have been explained in Section 3.2.2 above.

```
--- Sort Information
sorts Name Nonce NeNonceSet Gen Exp Key GenvExp Enc Secret .
subsort Gen Exp < GenvExp .
subsort Name NeNonceSet GenvExp Enc 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 3.2.2 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 -> Enc [frozen] .
op d : Key Msg -> Enc [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 theories besides associative-commutative are specified as follows:

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

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

```
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 ]
[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 the other for the secret data. And 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]
```

The attack state is represented by the following, which states that only the strand of `B` is required to appear in the possible initial state, and that no intruder knowledge is required. Moreover, for authentication purposes, the initiator strand is avoided using a never pattern[11]:

```
:: 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
butNeverFoundAny
*** 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 || M:SMsgList || G:GhostList)
```

[11]Note that attack states and never patterns are appropriately renamed to avoid variable clash.

The search terminates in 12 backwards narrowing steps and two attacks are found. We list the first[12]:

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

[12]Note that expressions such as `irr(X:Msg)` or `inst(X:Msg)` included into the intruder knowledge require that message `X:Msg` has to be strongly irreducible (see definition in Page 13). The difference between `irr` and `inst` is that `inst` considers only new variables introduced by instantiation whereas `irr` any term in general. These constraints are dynamically checked, in order to discard states not satisfying them.

```
    -(#1:Name),
    -(exp(g, n(b, #2:Fresh))),
    +(#1:Name ; exp(g, n(b, #2:Fresh))),
    -(a),
    -(#1:Name ; exp(g, n(b, #2:Fresh))),
    +(a ; #1:Name ; exp(g, n(b, #2:Fresh))),
    -(a ; #1:Name ; exp(g, n(b, #2:Fresh))),
    +(e(exp(g, n(a, #0:Fresh) <+> n(b, #2:Fresh)), sec(a, #3:Fresh))),
    -(e(exp(g, n(a, #0:Fresh) <+> n(b, #2:Fresh)), sec(a, #3: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, the other attack that Maude-NPA displays is only a slight variant.

The reason 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 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, empty
     || nil
     || nil
  [nonexec] .
```

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

```
result IdSystem: < (1[1]) . 5 . 5 . (16{1}) . (5{1}) . 10 . 10 . 9 . 9 . 9 . 1 . 5 . 2 > (
:: nil :: [nil | -(exp(g, n(a, #2:Fresh))), -(#3:NeNonceSet),
        +(exp(g, #3:NeNonceSet <+> n(a, #2:Fresh))), nil] &
:: nil :: [nil | -(exp(g, n(b, #0:Fresh))), -(#5:NeNonceSet),
        +(exp(g, #5:NeNonceSet <+> n(b, #0:Fresh))), nil] &
:: nil :: [nil | -(exp(g, #3:NeNonceSet <+> n(a, #2:Fresh))),
        -(e(exp(g, #3:NeNonceSet <+> n(a, #2:Fresh)), sec(a, #4:Fresh))), +(sec(a, #4:Fresh)), nil] &
:: nil :: [nil | -(exp(g, #5:NeNonceSet <+> n(b, #0:Fresh))), -(sec(a, #4:Fresh)),
        +(e(exp(g, #5:NeNonceSet <+> n(b, #0:Fresh)), sec(a, #4:Fresh))), nil] &
:: nil :: [nil | -(a ; b ; exp(g, n(b, #0:Fresh))),
        +(b ; exp(g, n(b, #0:Fresh))), nil] &
:: nil :: [nil | -(a ; #1:Name ; exp(g, n(a, #2:Fresh))),
```

[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.

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

The search space generated by Maude-NPA can be very large, and it is convenient to use never patterns in the attack state to restrict the search space, as well as to define that attack state. Consider the following attack state with never patterns.

```
:: 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, empty     || nil
|| nil
butNeverFoundAny
*** Pattern to avoid infinite search space
     (:: nil ::
 [ nil | -(exp(GE,NS1 <+> NS2)), -(NS3),
     +(exp(GE,NS1 <+> NS2 <+> NS3)), nil ]
 & S:StrandSet || K:IntruderKnowledge || M:SMsgList || G:GhostList)
*** Pattern to avoid unreachable states
(:: nil ::
 [nil | -(exp(#1:Exp, N1:Nonce)),
        -(sec(A:Name, #2:Fresh)),
        +(e(exp(#1:Exp, N2:Nonce), sec(A:Name, #2:Fresh))), nil]
 & S:StrandSet || K:IntruderKnowledge || M:SMsgList || G:GhostList)
*** Pattern to avoid unreachable states
(:: nil ::
 [nil | -(exp(#1:Exp, N1:Nonce)),
     -(e(exp(#1:Exp, N1:Nonce), S:Secret)),
     +(S:Secret), nil]
 & S:StrandSet || K:IntruderKnowledge || M:SMsgList || G:GhostList)
*** Pattern to avoid unreachable states
(S:StrandSet
 || (#4:Gen != #0:Gen), K:IntruderKnowledge ||
     M:SMsgList || G:GhostList)
```

Let us explain the never patterns used above. There are two kinds of never patterns: (1) concrete intruder actions that lead to infinite search patterns for this protocol and need to be removed to guarantee termination, and (2) unreachable states that appear in the search space, that will ultimately be shown unreachable but nevertheless increase its size. We note, however, that the use of the first type of never pattern can mean that failure to find an attack no longer guarantees security. It is most useful when testing new equational theories whose behavior is not that well understood yet. We describe each of these never patterns in detail below:

- The intruder strand

  ```
  [ nil | -(exp(GE,NS1 <+> NS2)), -(NS3),
       +(exp(GE,NS1 <+> NS2 <+> NS3)), nil ]
  ```

  leads to an infinite search space involving larger and larger nonce sets. Since the protocol requires only a nonce set of size two to execute, it appears safe as a first approximation to exclude any intruder strands that create a nonce set of size three or more.

- The intruder strand

```
[nil | -(exp(#1:Exp, N1:Nonce)), -(sec(A:Name, #2:Fresh)),
       +(e(exp(#1:Exp, N2:Nonce), sec(A:Name, #2:Fresh))), nil]
```

is unreachable, since there is no way that the negative term `exp(#1:Exp,Nonce)` can be irreducible, as required. [14]

- The intruder strand

  ```
  [nil | -(exp(#1:Exp, N1:Nonce)), -(e(exp(#1:Exp, N1:Nonce), S:Secret)),
         +(S:Secret), nil]
  ```

  is unreachable, for the same reason.

- Any generated state containing the constraint `#4:Gen != #0:Gen` is unreachable, since there is only one term of sort `Gen`.

We try running this attack state with never patterns, and compare it with the output of the same attack state with never patterns:

Attack State	1	2	3	4	5	6	7	8	9	10	11	12
w/o nevers	4	6	11	10	15	17	25	17	6	3	2	1
with nevers	3	4	5	5	7	7	7	5	4	3	2	1

Although never patterns can have a dramatic effect on search space size, they should be used with care, because improper use of never patterns can cause Maude-NPA to miss a genuine attack. However, using never patterns to reduce search space size can be very useful in debugging protocols, since they allow the user to pinpoint weaknesses without performing a full-scale Maude-NPA search.

One important technique used in both the NRL-Protocol Analyzer and the Maude-NPA is the use of *grammars* [17, 6] to characterize sets of *unreachable* states, that is, states from which, through backwards search, it is provably impossible to reach an initial state. Grammars can drastically cut down the search space and often allow backwards search to terminate in many cases where unrestricted search would not. Appendix B gives a more detailed description of grammars in the Maude-NPA.

If nothing is specified by the user in a protocol specification, the Maude-NPA automatically generates grammars for such a protocol in a way entirely transparent to the user. However, to further constrain the search space without losing completeness, it is possible for the user to suggest additional grammars, as "initial grammars", that are both tested and completed by the Maude-NPA into "final grammars" where all the generated terms are provably unreachable.

A good case in point is the present Diffie-Hellman example, where one such initial grammars is indeed useful in further constraining the attack search. We do not give the grammar here, but instead refer to Appendix B.2, where such a grammar is presented and explained, and to Appendix F.2, where the full protocol specification for the Diffie-Hellman example, including the specification of the above-mentioned initial grammar, is given.

Besides grammars, the Maude-NPA uses a variety of other state-space reduction techniques. When used in combination with grammars, these can drastically cut down the search space size. Indeed, in a good number of cases these techniques reduce what would be an infinite number of states to a *finite* number, so that termination of the Maude-NPA without finding an attack provides *full verification* that the attack searched for is impossible under

[14]Recall that received messages (negative terms) are always supposed to be in irreducible form.

the assumptions of the specification. The state space reduction techniques are intended to be transparent to the user, so we do not discuss them in the main body of the manual. However, for the sake of documentation, we include a brief discussion of them, with references, in Appendix D.

5.2 Tiny xor protocol

We consider a very simple protocol using xor. That is, two principals exchange just messages consisting of names of principals glued together with the xor operator. The informal textbook-level description of the protocol is as follows.

1. $A \to B : A$
2. $B \to A : B$
3. $A \to B : A$
4. $B \to A : B$
5. $A \to B : A \oplus A \oplus B$

The point to consider is that we assume that principals cannot check that they have received the name of a principal and, therefore, they use a variable of sort `Msg`.

The sort used in this protocol is just `Name`.

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

The operations used are as follows

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

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

whose algebraic properties are described as follows.

```
eq X:Msg <+> X:Msg = null .
eq X:Msg <+> X:Msg <+> Y:Msg = Y:Msg .
eq X:Msg <+> null = X:Msg .
```

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

```
 :: nil :: [ nil | -(M1), -(M2), +(M1 <+> M2), nil ] &
 :: nil :: [ nil | +(A), nil ] &
 :: nil :: [ nil | +(null), nil ]
```

The informal textbook-level description above is encoded in our tool as follows, taking into account that a received message that is unknown to a principal is represented by a variable. The strand for principal `A` is:

```
:: nil :: [nil | +(A), -(B1), +(A), -(B2), -(B1 <+> B2), nil] &
```

Note that `A` uses two fresh variables, one for the nonce and the other for the secret data. And the strand for principal `B` is:

```
:: nil :: [nil | -(A1), +(B), -(A2), +(B), +(A1 <+> A2 <+> B <+> B), nil]
```

The attack state is represented by the following term.

```
:: nil :: [nil, +(A), -(B1), +(A), -(B2), -(B1 <+> B2) | nil]
|| empty
|| nil
|| nil
```

Five different attacks are found in just five backwards narrowing steps.

```
(:: nil :: [nil | +(#0:Name), -(null), +(#0:Name), -(#1:Msg), -(#1:Msg), nil]
||
empty
||
+(#0:Name), -(null), +(#0:Name), -(#1:Msg), -(#1:Msg)
||
nil)
(:: nil :: [nil | +(#0:Name), -(#1:Msg), +(#0:Name), -(#1:Msg), -(null), nil]
||
inst(#1:Msg)
||
+(#0:Name), -(#1:Msg), +(#0:Name), -(#1:Msg), -(null)
||
nil)
(:: nil :: [nil | +(#0:Name), -(#1:Msg), +(#0:Name), -(#1:Msg * #2:Msg), -(#2:Msg), nil]
||
inst(#1:Msg), inst(#2:Msg)
||
+(#0:Name), -(#1:Msg), +(#0:Name), -(#1:Msg * #2:Msg), -(#2:Msg)
||
nil)
(:: nil :: [nil | +(#0:Name), -(#1:Msg * #2:Msg), +(#0:Name), -(#1:Msg), -(#2:Msg), nil]
||
inst(#1:Msg), inst(#2:Msg)
||
+(#0:Name), -(#1:Msg * #2:Msg), +(#0:Name), -(#1:Msg), -(#2:Msg)
||
nil)
(:: nil :: [nil | +(#0:Name), -(#1:Msg * #2:Msg), +(#0:Name), -(#1:Msg * #3:Msg),
                  -(#2:Msg * #3:Msg), nil]
||
inst(#1:Msg), inst(#2:Msg), inst(#3:Msg)
||
+(#0:Name), -(#1:Msg * #2:Msg), +(#0:Name), -(#1:Msg * #3:Msg), -(#2:Msg * #3:Msg)
||
nil)
```

6 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 Section B.
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 B.
3. The unification algorithm of a user-defined equational theory does not terminate for arbitrary equational theories. For a detailed description of the theories of interest for which it does and does not terminate, and our plans to extend the current class of supported theories to theories satisfying the finite variant property, see Section A.
4. In some cases, when a unification algorithm would produce an infinite number of unifiers, it is possible to obtain a *finitary* unification algorithm by a judicious use of sorts. For more details on this, see Section A.
5. In some finitary cases, the number of unifiers produced, although finite, is so large that it crashes Maude-NPA. This is currently happening for many protocols using exclusive-or, for example. In many cases, these unifiers are redundant.

References

[1] Siva Anantharaman, Paliath Narendran, and Michaël Rusinowitch. Unification modulo *cui* plus distributivity axioms. *Journal of Automated Reasoning*, 33(1):1–28, 2004.

[2] 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.

[3] Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: How to get rid of some algebraic properties. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.

[4] Nachum Dershowitz, Subrata 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.

[5] S. Escobar and C. Meadows. State space reduction in the Maude-NRL protocol analyzer. In *European Symposium on Research in Computer Security, ESORICS 2008, Proceedings*, LNCS to appear. Springer, 2008.

[6] 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.

[7] 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.

[8] S. Escobar, 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)*, to appear.

[9] 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.

[10] S. Escobar, J. Meseguer, and R. Sasse. Variant narrowing and equational unification. In G. Rossu, editor, *Proc. 7th. Intl. Workshop on Rewriting Logic and its Applications*, ENTCS to appear. Elsevier, 2008.

[11] Santiago Escobar, José Meseguer, and Ralf Sasse. Effectively checking the finite variant property. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2008.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] Emanuele 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.

A Narrowing-based Finitary Equational Unification

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. In what follows we explain three things regarding T-unification in the Maude-NPA:

- Equational axioms for which the Maude-NPA provides built-in support for equational unification.
- Narrowing-based equational unification in general, which is however infeasible for Maude-NPA analysis when the number of unifiers generated is infinite; and
- The most general case of equational theories for which the Maude-NPA can currently support unification by narrowing, with the important requirement of the number of unifier solutions being *finite*, namely, the admissible theories described in Section 3.2.3.

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

The Maude-NPA has built-in support for unification modulo certain equational theories T thanks to the underlying Maude infrastructure. Specifically, the Maude-NPA automatically supports unification modulo T for T any order-sorted theory of the form $T = (\Sigma, Ax)$, where Ax is a collection of equational *axioms* where some binary operators f in the signature Σ may have axioms in Ax for either *commutativity* ($f(x, y) = f(y, x)$), or commutativity and *associativity* ($f(x, f(y, z)) = f(f(x, y), z)$). Associativity alone is not supported, because it is well-known that unification problems modulo associativity may have an *infinite* number of unifiers. As already illustrated in Section 3.2.2, the way associativity, and/or commutativity axioms are specified in Maude for a function symbol f is not by giving those axioms explicitly, but by declaring f in Maude with the `assoc` and/or `comm` attributes. For example a function symbol `f` of sort `S` which is associative and commutative is specified in Maude as follows:

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

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 axioms, for which the Maude-NPA provides automatic built-in support. Therefore, the burning issue is how to support more general classes of algebraic theories in the Maude-NPA.

In this regard, a very useful, generic method to obtain T-unification algorithms is *narrowing* [13, 14]. In order for narrowing to provide a T-unification algorithm, the theory T has to be of the form $T = (\Sigma, E \cup Ax)$, where Ax is a collection of equational axioms such as our previous combinations of associativity and/or commutativity 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. coherent modulo Ax (see [14]).

The precise meaning of these three requirements was explained in detail in Section 3.2.3.

Although narrowing is a very general method to generate T-unification algorithms, general narrowing has a serious limitation. The problem is that, in general, narrowing with an equational theory $T = (\Sigma, E \cup Ax)$ satisfying requirements (1)–(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 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 [13, 4]. The case when $Ax \neq \emptyset$ is considerably more challenging (see, e.g., [3, 20, 9]). However, the condition of *strong right irreducibility* in [9], applies to both cases ($Ax = \emptyset$ and $Ax \neq \emptyset$) and is easy to check. This condition was already explained in detail in Section 3.2.3 and ensures that there is always a finite number of unifiers modulo $E \cup Ax$.

As already mentioned in Sections 3.2.3 and 3.2.4, the Maude-NPA's support for order-sorted specifications is very helpful in achieving strong right irreducibility, and therefore in reaching the desired goal of obtaining 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 strongly right irreducible theories T that have finitary T-unification algorithms. Furthermore, order-sortednedss 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 more general condition than strong right irreducibility, call the *finite variant property* has been recently proposed by Comon and Delaune [3]. This condition is satisfied by a number of useful cryptographic theories. Furthermore, in [10] it is shown how a finitary unification algorithm can be obtained in these theories. Methods for checking the finite variant property are proposed in [11]. In a future version of Maude-NPA we plan to support narrowing-based unification for the broader class of theories enjoying the finite variant property.

B 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 [6]. 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 4 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 5.1.

B.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 `genGrammars` constant in Maude-NPA typing:

```
red genGrammars .
```

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. Note that the `genGrammars` command can fail to generate a grammar for a concrete initial grammar. Such failure grammars are identified by terms starting with `errorNoHeuristicApplied` or `errorIntegratingExceptions`. Failure grammars cannot be included in the `GENERATED-GRAMMARS` equation. The `GENERATED-GRAMMARS` equation is added to the module `PROTOCOL-SPECIFICATION` in the general template described in Section 3, right before the attack state specifications. Maude-NPA will now treat these as the initial grammars.

B.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 5 requires the following initial grammar, which is not yet 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 .
      ! S2 )
```

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 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 C. If the user wants to see what the initial grammars generated by Maude-NPA look like, this is 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.

B.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 grammar specifications, following Appendix C, instead of the ones generated by Maude-NPA. This feature is only recommended for debugging Maude-NPA.

C 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 [6].

```
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.

D Other Optimizations

In this section we describe the various optimization techniques, besides grammars, that the Maude-NPA uses to reduce the size of the search space. All of these, except for the second partial order reduction, are described in detail in [5].

1. Partial Order Reductions

   a) One of the possible backwards steps in a Maude-NPA analysis involves moving a negative strand into the intruder knowledge. If any of these steps are possible, they are always executed first.

   b) If there are two or more terms in the intruder knowledge that share no variables, and both can be matched by the output of intruder strands, only one order of execution is used.

2. Detecting Inconsistent States Early

   Certain types of states containing inconsistent information are eliminated early.

   a) A state St containing two contradictory facts (t `inI`) and (t `!inI`) for a term t.

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

   c) 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 $m_1^\pm, \ldots, m_{j-1}^\pm$.

   d) 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.

3. Transition Subsumption

   There are a number of cases in which it is possible to tell that a state S_1 is reachable only if another state S_2 is. Roughly speaking, this occurs when S_1 subsumes a substate of S_2. In that case, S_2 is deleted and only S_1 is kept. Whenever a new state is found, this transition subsumption check is done on it with all existing states, including states appearing higher up in the search tree, to determine whether or not it should be kept.

4. Super Lazy Intruder

   If variable terms, publicly known constants (such as names), or terms constructed out of variables and publicly known constants appear in the intruder knowledge, then there is no need to search for them since it is trivial for the intruder to produce them. However, it is possible that later on in the search they may become instantiated, and it will then be necessary to reintroduce them. The solution is to remove these terms (called *super-lazy terms*) but keep the state in which they appear around as a ghost state which can be resuscitated if the variables in the terms are instantiated. The ghost states are stored in the fourth argument of the attack state in the backwards search.

E Commands Useful for Debugging

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

E.1 Excluding Optimizations and Checks

For debugging purposes, it is possible to disable optimization techniques and validity checks on the data selectively. 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 inconsistency check 2.a described above
3. `-inputAndNotLearned` turns off inconsistency check 2.b
4. `-alreadySent` turns off inconsistency check 2.c
5. `-secretData` turns off inconsistency check 2.d
6. `-implication` turns off the transition subsumption
7. `-equationalRed` turns off the check that negative terms are irreducible
8. `-freshInstantiated`, turns off the check that fresh variables are never instantiated
9. `-inputFirst` turns off the partial order reduction 1.a
10. `-partialOrder` turns off the partial order reduction 1.b
11. `-ghost` turns off the super-lazy intruder.
12. `none` turns off all of the optimizations. Note that `-` is not used here.

F Example Protocol files

This appendix contains the full protocol specifications for the protocol examples used in this manual.

F.1 Needham-Schroeder public key protocol

The contents of the file describing this protocol are as follows:

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  --- Importing sorts Msg, Fresh, Public, and GhostData
  protecting DEFINITION-PROTOCOL-RULES .

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

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

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

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

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

endfm

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .

  var Z : Msg .
  var Ke : Key .

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

endfm

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

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

  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] .

  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] .

  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, empty
     || nil
     || nil
  [nonexec] .

endfm

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

F.2 Diffie-Hellman protocol

The contents of the file describing this protocol are as follows:

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  --- Importing sorts Msg, Fresh, Public
  protecting DEFINITION-PROTOCOL-RULES .

  --- Sort Information
  sorts Name Nonce NeNonceSet Gen Exp Key GenvExp Enc Secret .
  subsort Gen Exp < GenvExp .
  subsort Name NeNonceSet GenvExp Enc Secret Key < Msg .
  subsort Exp < Key .
  subsort Name < Public . --- This is quite relevant and necessary
  subsort Gen < Public . --- This is quite relevant and necessary

  --- 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 -> Enc [frozen] .
  op d : Key Msg -> Enc [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)] .

endfm

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .

  eq exp(exp(W:Gen,Y:NeNonceSet),Z:NeNonceSet)
   = exp(W:Gen, Y:NeNonceSet <+> Z:NeNonceSet) .
  eq e(K:Key,d(K:Key,M:Msg)) = M:Msg .
  eq d(K:Key,e(K:Key,M:Msg)) = M:Msg .

endfm

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

  vars NS1 NS2 NS3 NS : NeNonceSet .
  var NA NB N : Nonce .
  var GE : GenvExp .
  var G : Gen .
  vars A B : Name .
  vars r r' r1 r2 r3 : Fresh .
  var Ke : Key .
  vars XE YE : Exp .
  vars M M1 M2 : Msg .
  var Sr : Secret .

  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 ]
  [nonexec] .

  eq STRANDS-PROTOCOL =
     :: r,r' ::
     [nil | +(A ; B ; exp(g,n(A,r))),
            -(A ; B ; XE),
            +(e(exp(XE,n(A,r)),sec(A,r'))), nil] &
     :: r ::
     [nil | -(A ; B ; XE),
            +(A ; B ; exp(g,n(B,r))),
            -(e(exp(XE,n(B,r)),Sr)), nil]
  [nonexec] .

  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] .

  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
     butNeverFoundAny
     *** 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 || M:SMsgList || G:GhostList)
     *** Pattern to avoid infinite search space
          (:: nil ::
      [ nil | -(exp(GE,NS1 <+> NS2)), -(NS3), +(exp(GE,NS1 <+> NS2 <+> NS3)), nil ]
      & S:StrandSet || K:IntruderKnowledge || M:SMsgList || G:GhostList)
     *** Pattern to avoid unreachable states
     (:: nil ::
      [nil | -(exp(#1:Exp, N1:Nonce)),
             -(sec(A:Name, #2:Fresh)),
             +(e(exp(#1:Exp, N2:Nonce), sec(A:Name, #2:Fresh))), nil]
      & S:StrandSet || K:IntruderKnowledge || M:SMsgList || G:GhostList)
     *** Pattern to avoid unreachable states
     (:: nil ::
      [nil | -(exp(#1:Exp, N1:Nonce)), -(e(exp(#1:Exp, N1:Nonce), S:Secret)),
             +(S:Secret), nil]
      & S:StrandSet || K:IntruderKnowledge || M:SMsgList || G:GhostList)
     *** Pattern to avoid unreachable states
     (S:StrandSet
      || (#4:Gen != #0:Gen), K:IntruderKnowledge || M:SMsgList || G:GhostList)
  [nonexec] .
endfm

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

F.3 Tiny XOR protocol

The contents of the file describing this protocol are as follows:

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  --- Importing sorts Msg, Fresh, Public
  protecting DEFINITION-PROTOCOL-RULES .

  ----------------------------------------------------------
  --- Overwrite this module with the syntax of your protocol
  --- Notes:
  --- * Sort Msg and Fresh are special and imported
  --- * Every sort must be a subsort of Msg
  --- * No sort can be a supersort of Msg
  ----------------------------------------------------------

  --- Sort Information
  sorts Name .
  subsort Name < Msg .
  subsort Name < Public .

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

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

endfm

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .

  ----------------------------------------------------------
  --- Overwrite this module with the algebraic properties
  --- of your protocol
  ----------------------------------------------------------

  eq X:Msg <+> X:Msg = null .
  eq X:Msg <+> X:Msg <+> Y:Msg = Y:Msg .
  eq X:Msg <+> null = X:Msg .

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
  ----------------------------------------------------------
  var A B : Name .
  var r r' : Fresh .
  vars X Y M M1 M2 M3 : Msg .
  vars A1 A2 B1 B2 : Msg .

  eq STRANDS-DOLEVYAO
   = :: nil :: [ nil | -(M1), -(M2), +(M1 <+> M2), nil ] &
     :: nil :: [ nil | +(null), nil ]
  [nonexec] .

  eq STRANDS-PROTOCOL
   = :: nil ::
     [nil | +(A), -(B1), +(A), -(B2), -(B1 <+> B2), nil] &
     :: nil ::
     [nil | -(A1), +(B), -(A2), +(B), +(A1 <+> A2 <+> B <+> B), nil]
  [nonexec] .


  eq ATTACK-STATE(0) =
     :: nil ::
     [nil, +(A), -(B1), +(A), -(B2), -(B1 <+> B2) | nil]
     || empty
     || nil
     || nil
  [nonexec] .
endfm
```

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