MAUDE-PSL: A NEW INPUT LANGUAGE FOR MAUDE-NPA

BY

ANDREW RUSSEL CHOLEWA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisor:

Professor José Meseguer

Abstract

Maude-NPA is a narrowing-based model checker for analysing cryptographic protocols in the Dolev-Yao model modulo equations. Maude-NPA is a powerful analyzer that is sound and never returns spurious counter-examples. Maude-NPA is also very flexible, providing the user great flexibility in designing his/her own custom notation. Maude-NPA also supports a large variety of equational theories (any theory possessing the finite variant property, plus dedicated algorithms for homomorphism and exclusive or). However, Maude-NPA relies on a strand-based notation that, while very precise, is less familiar to users of the Alice-Bob notation. Furthermore, the input language itself is rather difficult to read and write. This makes Maude-NPA hard to use, and therefore a less attractive option for protocol verification despite its power. We propose a new input language called the Maude Protocol Specification Language (Maude-PSL). The Maude-PSL extends the Alice-and-Bob notation with the following additional pieces of information: the interpretation each principal has for every message he/she sends and receives, the information each principal is assumed to know at the start of the protocol execution, and the information the principal should know after execution. The Maude-PSL also provides simple yet expressive syntax for specifying intruder capabilities, secrecy attacks and authentication attacks. The Maude-PSL retains the flexible, Maude-like syntax for specifying the operators, type structure, and algebraic properties of a protocol. The semantics of the language is defined as a rewrite theory that rewrites Maude-PSL specifications into Maude-NPA strands. This provides a formal grounding of Maude-PSL specifications in a well understood model of cryptographic protocols.

To my parents, Bertram and Nancy Cholewa, for all the Saturday afternoons we spent at the local library.

Acknowledgments

This document would not have been possible without the help of several people. First, a special thanks to Santiago Escobar and Catherine Meadows for their criticisms, suggestions, and support of this project from inception to now. The Maude-PSL would be a far more inferior language without their insightful feedback. Next, thanks to Fan Yang for her help in developing an early prototype of the language. I'd also like to thank Stephen Skeirik for his willingness to read through drafts of this thesis and provide invaluable feedback. Finally, many thanks to my advisor, José Meseguer for his willingness to take me on as his student, for his support and patience throughout my graduate education, his many valuable suggestions for this protocol language, and most of all, for all that I have learned under his expert tutelage.

Table of Contents

Chapter 1

Introduction

In the modern day explosion of Internet-based commerce, a means of establishing secure communications between two geographically distant strangers has become tremendously important. For example, thousands of users across the globe need to be able to establish a secure connection with their banks, so that they may view their bank account information in the comfort and security of their own home (amongst other places). However, the nature of the Internet is such that anyone may be listening in on and interfering with any one of these communications, with neither the user nor the bank the wiser. As a result, over the past half a century, a tremendous amount of effort has been put into developing processes for establishing secure communications across insecure channels, called *cryptographic protocols*.

Unfortunately, developing secure protocols is a very difficult and subtle process. The security of a protocol may hinge on the absence of a single number. The smallest tweak may make an insecure protocol secure, or give a previously secure protocol a giant hole. To highlight the difficulty in designing a secure cryptographic protocol, a cautionary tale is that of the Needham-Schroeder public key protocol [23]. The protocol was developed in 1978, was proven secure, and was believed to be secure for many years. However, in 1998, twenty years after the protocol's introduction, Gavin Lowe found an exploit [19]. Due to the difficulty in proving a protocol secure, and the high stakes involved in ensuring that a cryptographic protocol is in fact secure, a lot of work has been done on trying to automate the proofs of security. The idea is that an automated, exhaustive proof should be able to catch the kind of subtle exploits that humans often fail to see immediately. As a result, of this research, a variety of tools have been developed, including but not limited to Maude-NPA[13], AVISPA[30], CPSA[9] and ProVerif [4]. However, the notation typically used by protocol designers to discuss cryptographic protocols (called here the Alice and Bob notation) is not precise enough to perform automated verification. As a result, a major component of all of these tools has been the development of a domain specific language for protocol specification.

Cryptographic protocol specification is a tricky balancing act. On the one hand, the specification must be provided in a formal logic with all the details needed to perform automatic verification. On the other, it is often difficult to

debug specifications provided to formal tools, because these tools do not execute the protocol in the normal sense of the word. As a result, the specification needs to be clear enough, and simple enough to ensure that the specifier provides a correct specification. Furthermore, if the specification language is either too complex or too different from what protocol analyzers are used to, the tool may not be used due to the steep learning curve. One tool that exemplifies this exact problem is Maude-NPA. Maude-NPA is a powerful tool for cryptographic protocol analysis based on symbolic model checking. Maude-NPA provides the user the ability to define their own syntax, and to verify protocols modulo a large class of equational theories. Furthermore, Maude-NPA analysis is sound, and does not produce spurious attacks. However, despite this power and flexibility, the input language is fairly complex, requires the user to include a variety of extra syntactic "kibble" that is necessary for the tool to function, but is semantically meaningless with respect to the protocol. Furthermore, in Maude-NPA, protocols are formulated based on mathematical objects called "strands." While strands are not particularly complicated, they do formulate protocols from a point of view that protocol designers may not be used to. Finally, attacks are specified in Maude-NPA using fragments of Maude-NPA states, which not only exposes internal implementation details to the user, but also forces code duplication.

Therefore, to improve the usability of Maude-NPA, we present a new input language called the Maude Protocol Specification Language (Maude-PSL) for Maude-NPA. Rather than using strands, the Maude-PSL uses an extension of the Alice and Bob notation. This ensures that information about the destination and source of each message is preserved. Second, the syntax is made as minimal as possible. Everything that the user writes has to do with the protocol, the properties of the operations used by the protocol, and the environment in which the protocol is being executed. Third, attacks are specified without explicitly rewriting the protocol, but rather by referencing it. In particular, attacks only need to be modified if the variables used in the protocol are changed. So long as those stay constant, attacks are not affected by modifications to the protocol. The Maude-PSL also introduces additional more stringent static checks that are meant to capture as many subtle, otherwise hard-to-detect errors as possible.

The Maude-PSL is implemented on top of Maude-NPA by using the original, strand-based input language as an intermediate language. The translation from a Maude-PSL specification to the corresponding Maude-NPA specification is implemented using a combination of Python and Maude. At the Python level, we decompose the specification into the high level syntax and user-defined terms. The Python level then performs a variety of checks on the top level syntax. If the specification passes all of these checks, then Python uses the specification to generate a term to be rewritten by Maude. This term is then fed into Maude. Maude then performs certain checks that Python cannot easily handle (such as the syntactic correctness of user-defined terms). If the specification-derived-

term passes all of these checks as well, then Maude rewrites the term into a Maude-NPA specification. Python then writes this specification to a file. From this point on, the user may load the Maude-NPA specification into Maude-NPA, and interact with Maude-NPA in the exact same manner as if he/she had written the specification in Maude-NPA's original input language. Figure 1 provides a high-level, visual depiction of the translation process.

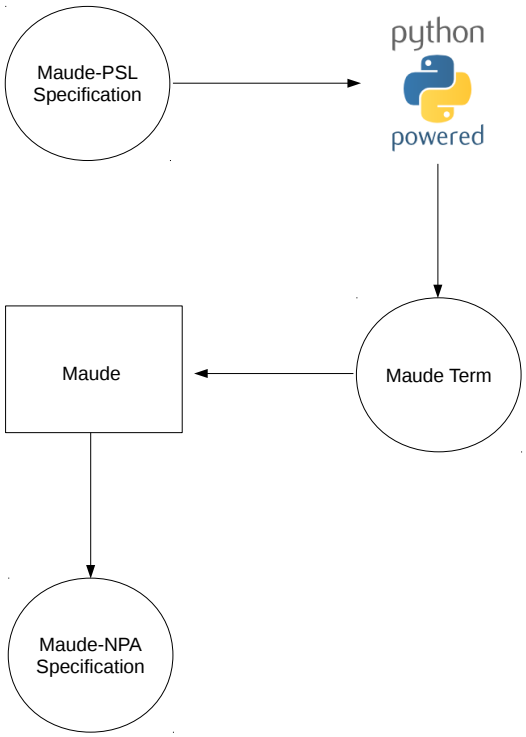

Figure 1.1: An overview of the translation from a Maude-PSL specification to a Maude-NPA specification.

The Maude-PSL tool can be found at
`http://maude.cs.uiuc.edu/tools/Maude-NPA/Maude-PSL/`.

Chapter 2

Background

The goal of the Maude-PSL is to provide a user-friendly specification language for use with Maude-NPA, an order-sorted rewriting-based tool for the verification of cryptographic protocols. Therefore, the user of the language must have some familiarity with order-sorted term rewriting, cryptographic protocol analysis, and Maude-NPA.

2.1 Rewriting Logic and Maude

2.1.1 Rewrite Theories

Term rewriting is a Turing-complete computational model based on the step-by-step modification of mathematical entities called *terms*. A program is modeled in rewriting logic using a *rewrite theory*. Maude is a declarative programming language where programs are (various flavors of) rewrite theories, called *modules* [7]. Since the Maude-PSL uses a fragment of Maude as a sub-language, a basic understanding of both rewriting logic and Maude are imperative to understanding how to use the Maude-PSL. Therefore, we shall introduce the two in tandem. We shall explain various flavors of rewriting logic, and write out example theories in Maude code.

When specifying protocols in the Maude-PSL, we are concerned with equational theories. Equational theories are pairs $(\Sigma, \mathcal{E})$, where Σ, called the theory's *signature* consists of a set of symbols called *operators*, and $\mathcal{E}$, called the theory's *equations*, is a set of equalities between terms. For example, the equational theory of the natural numbers with addition may consist of a constant 0 representing the number zero, an operator s representing the successor (plus one) function, an operator $+$ representing addition, and the equations

$$x + 0 = x \tag{2.1}$$

$$0 + x = x \tag{2.2}$$

$$s(x) + y = s(x + y) \tag{2.3}$$

One can then use these equations to perform equational reasoning (replacing equals by equals) to prove terms equal modulo the equations. For example, the terms $s(0) + s(0)$ and $s(s(0))$ can be proven equal modulo $\mathcal{E}$, written $s(0) +$

$s(0) =_{\mathcal{E}} s(s(0))$ as follows: $s(0) + s(0) =_{2.3} s(0 + s(0)) =_{2.2} s(s(0))$

Unfortunately, equational reasoning is difficult to automate. Equations may be applied either left to right, or right to left, and it may not be obvious which direction should be applied at any given point. However, if we orient the two-directional equations into one-directional rules such that the rules are *confluent* and *terminating* (see Section 2.1.5), then we may safely use the one-directional rules instead of the two-directional equations to automate equational reasoning. Equational theories whose equations have been oriented are called *rewrite theories*, and are denoted (Σ, R), where R is the set of rules[1].

What follows is an exploration of some of the basic concepts in term rewriting. However, this section provides only the barest of glimpses into the richness and complexity of rewriting logic, and only covers those concepts needed to use the Maude-PSL. References to resources that provide a more in-depth treatment may be found in Section 2.1.6.

Signatures

Every operator $f \in \Sigma$ is associated with a natural number n called the *arity* of the operator. Operators with an arity of 0 are called *constants*. Operators of arity n are called *n-ary*. 1-ary and 2-ary operators have special names: *unary* and *binary* respectively.

Terms are defined recursively as follows:

1. If $f \in \Sigma$ is a constant, then f is a term.
2. If $t_1, \ldots t_n$ are terms, and $f \in \Sigma$ is an n-ary operator, then $f(t_1, \ldots, t_n)$ is a term.

Note that $t_1, \ldots t_n$ in rule 2 are referred to as the *arguments* of f.

Example 2.1.1. Suppose we have the signature $\{0, s, +\}$ where 0 is a constant, s is unary, and $+$ is binary (This signature will be defined in Maude when we cover many-sorted theories).

The following are terms:

- 0
- $s(0)$, where the argument of s is 0.
- $+(s(0), 0)$, where the arguments of $+$ are $s(0)$ and 0.
- $s(+(s(0), 0))$, where the argument of s is $+(s(0), 0)$.

The operators in the example above are expressed using *prefix* syntax, i.e. the operator appears before its arguments. It is also possible to express operators using *mixfix* syntax, in which operator appear interwoven with the operator's arguments. For example, we may define the $+$ operator above as $\_ + \_$.

[1] Technically, a rewrite theory is just a signature and a set of rules. The rules do not need to be derived from an equational theory. However, such rewrite theories are beyond the scope of this document.

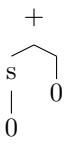

Figure 2.1: Tree of the term $s(0) + 0$

The underscores indicate the position at which arguments should be placed relative to the operator name. So, we may write the term $+(s(0), 0)$ as $s(0) + 0$. However, since each mixfix operator can be trivially turned into a prefix operator, we will assume all operators are prefix when discussing generic theories.

Terms may also be viewed as trees. Each node is an operator, and the children of each node are exactly the arguments of the corresponding operator. The root is the outermost operator, and the leaves are constants. See Figure 2.1 for a tree representation of $s(0) + 0$. The subtrees of a term tree are called the *subterms* of the term.

A position p in a term is a path from the root of the term tree to some node in the tree. The subterm t' of term t at position p, denoted t_p, is the subterm rooted at position p in the term tree. The empty position λ is the position of the root of the entire tree.

We assume that every signature has a countably infinite set X of constants called *variables*, disjoint from the rest of the signature. We only consider *finite* signatures. Finite signatures are those that can be decomposed into a disjoint union of a finite set of operators, and a countably infinite set of variables.

Substitutions

A *substitution* θ is a function from variables to terms. Every substitution may be uniquely extended to a function $\overline{\theta}$ from terms to terms as follows:

1. $\overline{\theta}(a) = a$ if a is a constant, and a is not in the domain of θ
2. $\overline{\theta}(x) = \theta(x)$.
3. $\overline{\theta}(f(t_1, \ldots, t_n)) = f(\overline{\theta}(t_1), \ldots, \overline{\theta}(t_n))$.

Because the extension is unique, we may refer to both the original substitution θ and its extension $\overline{\theta}$ as θ without ambiguity. Furthermore, instead of writing $\theta(t)$ for t a term, we write $t\theta$.

A substitution θ is called *idempotent* if and only if for every term t, $t\theta\theta = t\theta$.

Rules

The rules of a rewrite theory consists of a set of ordered pairs of terms $\{(u_1 \to v_1), \ldots, (u_n \to v_n)\}$. Computation is performed by using these rules to rewrite terms into other terms. We assume that for every rule $u \to v$, $vars(v) \subseteq vars(u)$.

Given two terms t, t', we say that t *rewrites to* t' with rule $u \to v \in R$ at position p with substitution θ, denoted $t \to_{p,\theta,u\to v} t'$ ($t \to t'$ if the rules, position and substitution are understood) iff $u\theta = t_p$, and $v\theta = t'_p$. In this case, we say that u *matches* t, with *matching substitution* θ. We denote the transitive closure by $t \to^+ t'$, and the reflexive-transitive closure by $t \to^* t'$.

A term t is called *R-normalized* (or just normalized) iff there is no rule $u \to v$, no term t', and no substitution θ such that $t \to t'$. Usually, computation in a rewrite theory consists of *normalizing* some term t, i.e. rewriting t until the term is R-normalized, and then taking the normalized term(s) as the result(s) of the computation.

Example 2.1.2. Recall the signature $\{0, s, \_+\_\}$ from Example 2.1.1. Suppose we also have the following rules:

$$x + 0 \to x \tag{2.4}$$

$$0 + x \to x \tag{2.5}$$

$$s(x) + y \to s(x + y) \tag{2.6}$$

Then, we can rewrite the term $s(s(0)) + s(0)$ to the normalized term $s(s(s(0)))$ as follows:

$$\begin{aligned} &s(s(0)) + s(0) \to_{\{x\mapsto s(0), y\mapsto s(0)\},\lambda,\ 2.6} \\ &s(s(0) + s(0)) \to_{\{x\mapsto 0, y\mapsto s(0)\},\lambda.1,\ 2.6} \\ &s(s(0 + s(0))) \to_{\{x\mapsto 0, y\mapsto s(0)\},\lambda.1.1.1,\ 2.5} \\ &\quad s(s(s(0))) \end{aligned}$$

If 0 is treated as the natural number 0, s as the *successor* (or plus one) function on natural numbers, and $+$ as natural number addition, then we see that we have computed the value of the expression $2 + 1$!

Now, observe that any term may be used as an argument to any operator. Since rewriting itself only depends on pattern matching, this may lead to unexpected behavior when attempting to model systems (such as programs written in programming languages with types) that do place restrictions on the types of computation that may be performed on different types of data.

Example 2.1.3. Consider the theory from Example 2.1.2, plus the additional operators:$\{\top, \bot, \_ \wedge \_\}$ where $\top$ and $\bot$ are constants, while $\_ \wedge \_$ is binary, and the rules:

$$\top \wedge z \to z \tag{2.7}$$

$$z \wedge \top \to z \tag{2.8}$$

$$\bot \wedge z \to \bot \tag{2.9}$$

$$z \wedge \bot \to \bot \tag{2.10}$$

These rules are meant to model the boolean values *true* and *false*, while the rules define $\wedge$ to be the boolean operation *and*.

However, there is nothing stopping us from considering (and evaluating) the term $(s(0) + \top) \wedge (\top + 0)$ to $s(\top)$, even though conceptually the successor of $\top$ (or $\top$ plus a number for that matter) makes no sense.

In order to get around this problem, one would have to introduce explicit operators and rules that disallow this kind of mixing. However, this is tedious, time consuming, and error prone. A more attractive approach is to introduce a notion of types, called *sorts*. To distinguish a rewriting theory with sorts from one without, we refer to a rewrite theory without sorts as an *unsorted rewrite theory*.

2.1.2 Many Sorted Rewriting

A many sorted rewriting theory is very much like an unsorted rewriting theory, except with several extensions to account for the sorts.

First, the signature Σ is now a pair (S, F), where S is a finite set of sorts, and F a finite set of operators. Furthermore, each n-ary operator $f \in F$ now has $n+1$ sorts associated with it: $s_1, s_2, \ldots s_{n+1}$. The first n sorts are the sorts of the operator's arguments, while the last sort is the result sort. An n-ary operator f with sorts $s_1, \ldots, s_{n+1}$ is denoted $f : s_1, \ldots s_n \to s_{n+1}$.

We now have a family of countable sets of variables, one for each sort: $\bigcup_{s \in S} X_s$.

Next, each term has associated with it at least one sort. We use $t : s$ to denote that t is of sort s. This sort is defined as part of the recursive definition of terms:

1. Every constant $a :\to s$ is a term of sort s
2. Let $f : s_1 \ldots s_n \to s_{n+1}$ be an n-ary operator, and $t_1, \ldots t_n$ be terms. Then, $f(t_1, \ldots, t_n)$ is a term of sort s_{n+1} iff $t_i : s_i$ for all $1 \leq i \leq n$.

A substitution θ is valid only if it preserves the sorts. In other words, it must be the case that for every x in the domain of θ, $x : s$ iff $x\theta : s$.

Example 2.1.4. Recall the theory from Example 2.1.3. Now, we shall write that theory as a many-sorted theory in Maude.

```
mod NAT-BOOL is
    sorts Nat Bool .

    op 0     : -> Nat .
    op s     : Nat -> Nat .
    op _+_   : Nat Nat -> Nat .
    op True  : -> Bool .
    op False : -> Bool .
```

```
    op _/\_  : Bool Bool -> Bool .

    vars X Y : Nat .
    var Z    : Bool .

    eq 0 + X      = X .
    eq X + 0      = X .
    eq s(X) + Y   = s(X + Y) .
    eq Z /\ True  = Z .
    eq True /\ Z  = Z .
    eq Z /\ False = False .
    eq False /\ Z = False .
endm
```

The keywords `op`, `var(s)`, and `eq` stand for "operator," "variable(s)," and "equation" respectively, while `/\` is an ASCII approximation of the boolean and operator "∧." The keywords `mod` and `endm` are used to begin and end a module of name `NAT-BOOL`. Despite the name, equations are in fact oriented from left to right as rules. By declaring rules to be equations, the programmer is claiming that the rules so defined are confluent and terminating (see Section 2.1.5 for more details).

Now, `s(0) + True) /\ (True + 0)` is not a valid Σ-term, because the `_+_` operator is only defined on pairs of terms of sort *Nat*. In other words, the above specification preserves the intuitive separation between natural numbers and booleans.

While a many-sorted theory allows us to naturally enforce the separation between different data types, it does not provide an easy means of encoding subtypes. For example, the fact that natural numbers are also integers, or the fact that encrypted and unencrypted messages are both still messages.

Example 2.1.5. Suppose we would like a theory that makes a distinction between integers and natural numbers, but still has them behave the same on shared data. In other words, we would like the natural numbers to be a subtype of integers. Our first attempt may look something like this:

```
mod NATS-INTS
  sorts Nat Int .
  op 0N   : -> Nat .
  op s    : Nat -> Nat .
  op _+_  : Nat Nat -> Nat .

  op 0I   : -> Int .
  op s    : Int -> Int .
  op p    : Int -> Int .
```

```
  op _+_  : Int Int -> Int .

  vars X1 X2 : Nat .
  vars Y1 Y2 : Int .

  eq X1 + 0N    = X1 .
  eq 0N + X1    = X1 .
  eq s(X1) + X2 = s(X1 + X2) .
  eq Y1 + 0I    = Y1 .
  eq 0I + Y1    = Y1 .
  eq s(Y1) + Y2 = s(Y1 + Y2) .
  eq p(Y1) + Y2 = p(Y1 + Y2) .
  eq s(p(Y1))   = Y1 .
  eq p(s(Y1))   = Y1 .
endm
```

where the `p` operator, or *predecessor* can be thought of as the inverse of the `s` or *successor* operation (i.e. `p` represents "minus one" as opposed to "plus one").

Obviously, there is a lot to be desired here. First, unless we are willing to tolerate an ambiguous signature (i.e. allow the term `s(0)` to have two parses: one as a `Nat`, the other as an `Int`) we do not actually have any overlap between natural numbers and integers. Second, there is a lot of repetition, both of operators and rules. If we would like to mix integers and naturals, then we would have even more rules. Finally, if we wish to transition between natural numbers and integers, we need to introduce an explicit function to perform that conversion.

There are smarter approaches to modeling subtyping in many-sorted theories. We can use predicates to model subtypes in much the same way that we can simulate sorts in an unsorted theory. However, doing this manually is again tedious, error prone, and can clutter up the theory with bookkeeping that we would rather not deal with. Therefore, it would be nice if we had some notion of subtypes built into the logic.

2.1.3 Order-Sorted Rewriting

In order-sorted rewriting, we extend many sorted rewriting with a notion of *subsorts*. The signature of an order-sorted rewrite theory is now a pair $((S, <), F)$, where S and F are as described in Section 2.1.2, and $<$ is a partial order on sorts. The partial order has a simple meaning: the sort s_1 is a subsort of s_2, written $s_1 < s_2$ if and only if for every Σ-term t of sort s_1, t is also of sort s_2. In other words, the set of all terms of sort s_1 is a subset of the set of all terms of sort s_2.

Example 2.1.6. The following is the order-sorted equivalent of the theory from Example 2.1.5

```
sorts Nat Int .
subsort Nat < Int .
op 0   : -> Nat .
op s   : Nat -> Nat .
op s   : Int -> Int .
op p   : Int -> Int .
op _+_ : Int Int -> Int .

vars X Y : Int .
eq s(p(X))  = X .
eq p(s(X))  = X .
eq X + 0    = X .
eq 0 + X    = X .
eq s(X) + Y = s(X + Y) .
eq p(X) + Y = p(X + Y) .
```

Since natural numbers are also integers, the equations apply to natural numbers as well as integers, so we may evaluate `s(0) + s(0)` to `s(s(0))`, `p(0) + p(0)` to `p(p(0))`, and `s(0) + p(0)` to 0 using the same equations for addition.

2.1.4 Rewriting Modulo Axioms

While orienting equations into rules provides an intuitive and general means for automating equational reasoning, some equations, such as commutativity and associativity, do not lend themselves well to orientation.

Example 2.1.7. Suppose we would like to explicitly make addition associative and commutative. Our first instinct would be to define the following module.

```
mod NAT is
    sorts Nat .

    op 0 : -> Nat .
    op s : Nat -> Nat .
    op _+_ : Nat Nat -> Nat .

    vars X Y Z : NAT .
    eq X + Y = Y + X .
    eq (X + Y) + Z = X + (Y + Z) .
    eq 0 + X = X .
    eq s(X) + Y = s(X + Y) .
endm
```

There are two problems with this. First, every term of the form `X + Y` will have an infinite rewrite sequence $X + Y \to Y + X \to X + Y \to \ldots$, because we can always map X to Y and Y to X. This is clearly not desirable. Second, the associativity equation does not give us true associativity. The associativity equation groups all elements of the additions except the first into one term. As a result, we are essentially forced to treat a sequence of multiple additions as a poor man's dequeue: we can add additions to the front and back, but can only access the first number in the addition in the pattern of a rule. However, associativity is supposed to allow us to freely group elements within the addition, allowing us to isolate arbitrary elements in the middle. So for the term $s(0) + s(0) + 0 + 0 + 0$ we should be able to work with any of $s(0) + (s(0) + 0 + 0 + 0)$, $(s(0) + s(0)) + 0 + (0 + 0)$, $(s(0) + s(0) + 0 + 0) + 0$ and so on.

To get around this, we can split our equations into a disjoint union: $B \uplus R$. B is a set of equations, called *axioms* that we do not try to orient, whereas R are the oriented equations. Instead we use a different method to reason about the equality of terms modulo B (typically an algorithm designed explicitly for the equations in B). Furthermore, we will now apply the rules in R *modulo* B. We define the relation $\to_{R/B}$ to be $=_B; \to_R; =_B$ where ; is relation composition in diagrammatic order, and we perform all rewriting as $t \to_{R/B} t'$. A rewrite theory modulo axioms is written as a triple (Σ, B, R) where Σ is the signature of the theory, B is the set of axioms, and R the set of rewrite rules.

Example 2.1.8. Maude supports associativity, commutativity, and unit (ACU) as axioms (among others, however Maude-NPA only supports commutativity, associativity with commutativity, and unit with associativity and commutativity). So in order to make addition commutative and associative, we add the following operator attributes:

```
fmod NAT is
    sorts Nat .

    op 0 : -> Nat .
    op s : Nat -> Nat .
    op _+_ : Nat Nat -> Nat [assoc comm] .

    vars X Y : Nat .
    eq 0 + X = X .
    eq s(X) + Y = s(X + Y) .
endfm
```

Note the `assoc` (associativity) and `comm` (commutativity) between brackets at the end of the `_+_` operator. Furthermore, observe that we have removed the equation `X + 0 = X` . This is because the `comm` attribute makes this equation unnecessary. Now, if we have the term $X + 0$, Maude will rewrite it as follows: $X + 0 =_{\{comm, assoc\}} 0 + X \to_R X =_{\{comm, assoc\}} X$.

We can also eliminate the equation $X + 0 = X$ by introducing the identity axiom:

```
fmod NAT is
    sort Nat .

    op 0 : -> Nat .
    op s : Nat -> Nat .
    op _+_ : Nat Nat -> Nat [assoc comm id: 0] .

    vars X Y : Nat .
    eq s(X) + Y = s(X + Y) .
endfm
```

Now, the equation `eq 0 + X = X .` is not necessary, because $s(0) + s(0) \rightarrow_R s(0 + s(0)) =_{\text{id: } 0} s(s(0))$. However, the identity attribute is a little bit dangerous, because it allows Maude to not only *remove* an identity from a term, but also to *add* an identity to a term. Therefore, if the user is not careful, he/she can very easily make Maude loop. For example, suppose we try to reduce the term $s(0)$ in the module above. Intuitively, we would expect the result to be $s(0)$. However, Maude will in fact loop on the following sequence:

$$s(0) =_{ACU} s(0) + 0 \rightarrow s(0 + 0) =_{ACU} s(0)$$

It is possible to avoid this problem through a careful application of sorts. For example, consider the following module.

```
fmod NAT is
    sorts Zero Nat NzNat .
    subsorts Zero NzNat < Nat .

    op 0 : -> Zero .
    op s : Nat -> NzNat .
    op _+_ : Nat Nat -> Nat
        [assoc comm id: 0] .

    vars X : Nat .

    var Y : NzNat .

    eq s(X) + Y = s(X + Y) .

endfm
```

Here, because 0 is not of sort *NzNat*, the equation $s(X) + Y = s(X + Y)$ does not apply to the term $s(0) + 0$. Therefore, $s(0) =_{ACU} s(0) + 0 =_{ACU} s(0)$

with no intermediate rewrite step possible.

However, implementing this fix is predicated on the knowledge that there is a problem in the first place. Since Maude-NPA is attempting to solve an undecidable problem, there is no guarantee that Maude-NPA will terminate. Therefore, a failure of Maude-NPA to terminate in a reasonable timespan says nothing about the correctness of the specification. So in the interest of minimizing the chance of error, the identity attribute is strongly discouraged.

Technically Maude does not perform $\rightarrow_{R/B}$ rewriting because $\rightarrow_{R/B}$ is undecidable in general. Instead, Maude performs the (in general) weaker rewriting relation $\rightarrow_{R,B}$ [16]. Although the $\rightarrow_{R,B}$ relation is in general weaker, there is a class of rewrite theories for which $\rightarrow_{R,B}$ and $\rightarrow_{R/B}$ are equivalent in power with respect to computing normal forms. These theories are called *coherent modulo the axioms*. Decision procedures exist for making well-formed equational theories coherent modulo C, AC, and ACU [26]. Maude-PSL leverages previous implementations of these procedures in Maude to automatically make all user-provided theories coherent. Therefore an understanding of the distinction between $\rightarrow_{R/B}$ and $\rightarrow_{R,B}$ is not necessary to correctly use the Maude-PSL. Therefore, since coherence is rather subtle, its discussion is omitted. The interested reader may wish to look at Jouannaud, Kirchner, and Kirchner's work on coherence [16], or Meseguer's overview of rewriting logic [20] for more details about coherence.

2.1.5 Confluence, Termination, and Least Sorted

One of the greatest strengths of Maude-NPA is the ability to reason about cryptographic protocols modulo an equational theory. Therefore, one of the purposes of the Maude-PSL is to allow us easily specify which equational theory we would like to reason modulo. We do this by defining an order-sorted rewrite theory, very similar to the modules in Example 2.1.8. However, in order to ensure that our order-sorted rewrite theory correctly models the desired equational theory, the rewrite theory must have two properties: *termination*, and *confluence*.

Termination

A rewrite theory (Σ, B, R) is *terminating* if for every term t, there is no infinite rewrite sequence $t \rightarrow t_1 \rightarrow \ldots \rightarrow t_i \rightarrow \ldots$. In other words, every rewrite sequence starting at t is of the form $t \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$, for some $n \in \mathbb{N}$, and t_n in normal form.

Confluence

A rewrite theory (Σ, B, R) is *confluent* if for any term t, we have that $t \rightarrow^*_{R/B} t_1$ and $t \rightarrow^*_{R/B} t_2$, implies that there exists a term t' such that $t_1 \rightarrow^*_{R/B} t'$ and $t_2 \rightarrow^*_{R/B} t'$. If the theory is terminating, then a rewrite theory is confluent if

and only if every term t has a unique normal form up to B-equality, called the *canonical form* of t, denoted $t\downarrow_{R/B}$, or just $t\downarrow$ if R and B are understood. We write $t \rightarrow^{!}_{R/B} t'$ to indicate that t is being rewritten to canonical form t'.

In other words, if a theory is confluent and terminating, then the theory's only nondeterminism is "don't care nondeterminism:" we will always (eventually) reach the same result, namely $t\downarrow$, regardless of the order in which we apply our rules.

It should be noted that if the axioms B are non-empty, then we care only about termination and confluence *modulo* B.

Example 2.1.9. Consider a theory of sets of natural numbers:

```
mod NAT-SET is

sorts Nat Set .
subsort Nat < Set .

op 0   : -> Nat .
op s   : Nat -> Nat .
op mt  : -> Set .
op _,_ : Set Set -> Set [assoc comm]

var S : Set .
eq S, S = S .
endm
```

Then, the set $0, 0, s(0) \rightarrow^{!}_{R/B} 0, s(0)$. Observe that one can rewrite $0, s(0)$ as $s(0), 0$. However, because both terms are equal modulo the axioms, they may be treated as "one" term, of which $0, s(0)$ is simply a representative.

Confluence and termination gives us the following key property, called the *Church-Rosser property.* Let $(\Sigma, \mathcal{E})$ be an equational theory, and (Σ, B, R) be an order-sorted rewrite theory obtained from $\mathcal{E}$ by orienting some equations into R and treating the rest as axioms. Then, if (Σ, B, R) is confluent and terminating modulo B, then for any two terms t, t', $t =_{\mathcal{E}} t'$ iff $t\downarrow_{R/B} =_B t'\downarrow_{R/B}$. In other words, equality may be decided by blindly rewriting both terms to canonical form, and then comparing the two canonical forms for equality modulo B.

2.1.6 Rewriting Resources

For additional information about (unsorted) rewriting, see the book *Term Rewriting and All That* by Franz Baader and Tobias Nipkow [3] or *Advanced Topics in Term Rewriting* by Enno Ohlebusch [24]. For details on order-sorted rewriting with axioms, see the survey paper *Twenty Years of Rewriting Logic* by José Meseguer [20].

The Maude Manual by Manuel Clavel, et. al. contains additional information about Maude [7]. A copy of the Maude Manual may be found on the Maude website: `http://maude.cs.illinois.edu`.

2.2 Cryptographic Protocol Analysis

The entire purpose of the Maude-PSL is to specify cryptographic protocols, which are then verified by Maude-NPA. Therefore, this section is meant to provide the reader with a brief introduction to the concepts in cryptography that he/she will need to use to understand the Maude-PSL.

2.2.1 Basic Concepts in Cryptography

Terminology

First, we need to introduce some basic terminology.

- *plain/ciphertext* - Plaintext is an unencrypted message that can be understood as-is. Ciphertext, is an encrypted message, which appears to be meaningless until decrypted.
- *symmetric key* - An unguessable object used by the encryption and decryption operations to generate the ciphertext (resp. plaintext) from the plaintext (resp. ciphertext).
- *principal* - An entity (such as a person, computer, or server) that is capable of sending messages over a network.
- *protocol* - A distributed algorithm generating a sequence of message passes between two or more principals.
- *nonce* - A value guaranteed to be unique, and unguessable (typically a sufficiently large random number). Nonces are typically used to give a message a watermark unique to the protocol execution for which the message was generated. This ensures that an intruder cannot blindly use a message acquired in a previous session as part of an attack on a future session (called a replay attack).
- *asymmetric key* - A pair of keys p, s with the property that any message encrypted with p can be decrypted only with s, and vice versa. However p (resp. s) cannot decrypt anything encrypted with p (resp. s), unlike with symmetric keys. Typically, one key is made publicly available (p), and the other (s) is kept secret. Then, anyone who wishes to securely send the owner of s a message encrypts the message with p. If the owner of p and s wishes to sign a message, he/she encrypts the message with s.

Alice and Bob

Next, we introduce one of the most popular notations for providing a loose specification of protocols: the Alice and Bob notation. In the Alice and Bob notation, a protocol is specified as a numbered sequence of message passes of the form

$$\begin{aligned} &1.\ A \to B : M_1 \\ &2.\ B \to A : M_2 \\ &3.\ A \to S : M_3 \\ &\vdots \end{aligned}$$

where each of A, B, and S represent three different principals (typically referred to as "Alice," "Bob," and "the Server" respectively), and each M_i is a message. Each line $A \to B : M_1$ says that Alice is sending the message M_1 to Bob.

Example 2.2.1. Consider the *Needham-Schroeder* public key authentication protocol, spelled out below [23]. Note that as originally presented, this protocol included steps in which Alice and Bob each interact with a third party server to obtain the other's public key. To simplify the protocol, we are going to leave out the interactions with the server, and assume that Alice and Bob already know the other's public key.

A represents the principal Alice, and B represents the principal Bob. $M_1; M_2$ is the concatenation of messages M_1 and M_2. $e(K_1, M_1)$ is the encryption of message M_1 with key K_1. PK_A (resp. PK_B) is the public key of A (resp. B). SK_A and SK_B are the associated secret keys. N_A is a nonce generated by A, and N_B is a nonce generated by B.

$$\begin{aligned} &1.A \to B : e(PK_B, A; N_A) \\ &2.B \to A : e(PK_A, N_A; N_B) \\ &3.A \to B : e(PK_B, N_B)) \end{aligned}$$

2.2.2 Cryptographic Protocols

The entire purpose of the Maude-PSL is to specify cryptographic protocols, which are then verified by Maude-NPA. A cryptographic protocol is a protocol that make certain security guarantees about the data sent and the principals involved. Typically, these protocols attempt to make two guarantees: the authenticity of the principals, and the secrecy of one or more terms sent during the protocol. If Alice wishes to communicate with Bob, then the first step is to make sure she is actually communicating with Bob, and not some unknown entity pretending to be Bob. Similarly, if Bob receives a message from Alice, he wants to make sure that he actually received a message from Alice, and not

a pretender. Therefore, suppose a protocol claims to successfully complete if and only if all principals involved are who they claim they are. Then we say that the protocol *authenticates* the principals. The protocol from Example 2.2.1 provides an example of an authentication protocol, i.e. a cryptographic protocol whose purpose is to authenticate the participants with each other.

Once Alice has confirmed that she is speaking with Bob, she usually wishes to send him sensitive information (perhaps a key for a different, more efficient encryption/decryption algorithm, or a meeting place). Then, she may encrypt the message with Bob's public key, or a symmetric key known only by the two of them, and send the message to Bob. A protocol guarantees the secrecy of a given message if and only if it is impossible for the intruder to obtain the message in plaintext.

When studying cryptographic protocols, we typically assume two things: perfect encryption, and that the intruder has complete control over the network. Perfect encryption says that the only way to decrypt an encrypted message is with the appropriate key. Giving the intruder complete control over the network, allows the intruder to receive every message sent, to send any message he/she can build to anyone on the network, and to destroy messages. The intruder is also capable of initiating separate protocol sessions with any other principal. This is called the *Dolev-Yao* model of the intruder [10]. The Dolev-Yao model allows us to focus solely on the protocol, and see if the guarantees it makes hold even under the most hostile environment imaginable. Attacks usually consist of the attacker either executing additional protocol sessions with the honest principals, or intercepting and replacing messages in such a way that the principals accidentally give the intruder access to secret information, or give the intruder the information he/she needs to impersonate one or more principals.

Example 2.2.2. Recall the Needham-Schroeder protocol from Example 2.2.1. Needham and Schroeder published the protocol in 1978. In 1995, Gavin Lowe found an attack that allows the intruder to impersonate A (in other words, a violation of the very authentication guarantee that the original protocol was designed to provide) [18]. The attack requires two sessions. In session 1, A initiates a valid session with I. In session 2, I begins a session with B in which I impersonates A. Following the convention from the original paper that reveals the attack, we write $I(A)$ to represent the intruder, and we use $x.y$ to indicate the sending of message y in session x. So 1.1 represents the sending of the first message in the first session.

The attack is as follows:

$$
\begin{aligned}
&1.1\ A \to I : (PK_I, A; N_A)\\
&2.1\ I(A) \to B : e(PK_I, A; N_A)\\
&2.2\ B \to I(A) : e(PK_A, N_A; N_B)\\
&1.2\ I \to A : e(PK_A, N_A; N_B)\\
&1.3\ A \to I : e(PK_I, N_B)\\
&2.3\ I(A) \to B : e(PK_B, N_B)
\end{aligned}
$$

In other words, the Intruder manages to successfully execute the protocol with B by passing the messages he/she receives from A in session 1 to B in session 2, and passing the messages he/she receives from B in session 2 to A in session 1. Note that the Intruder manages to accomplish this *without* ever learning the secret keys of A or B, or trying to break the encryption. Instead, A does all the work of authenticating the Intruder's identity as A to B. The fact that it took not quite two decades to find this attack is a testament to the difficulty in designing secure protocols and in verifying their security.

2.2.3 Protocol Strands

Due to the difficulty in proving cryptographic protocols secure, a lot of work has gone into developing tools that leverage ideas in formal methods (model checking, automated theorem proving, etc.) to prove cryptographic protocols (in)correct. However, in order to formally analyze a cryptographic protocol, we first need a mathematical model of cryptographic protocols. Two of the most popular such models are the π-calculus [22] (and various derivatives, such as the spi-calculus [1]), and the strand space model [15]. Since Maude-NPA uses strand spaces, we will focus exclusively on strands here.

Strand spaces model protocols by associating with each principal a strand. A strand is a sequence of signed messages (terms) $\pm m_1, \pm m_2, \ldots, \pm m_n$ that the associated principal sends and receives. A positive term $+m$ is a term that the owner of the strand sends, while a minus term $-m$ is a term that the owner of the strand receives. A strand is denoted $A : [\pm m_1, \pm m_2, \ldots, \pm m_n]$ where A is the principal who owns the strand, and each $\pm m_i$ represents either $+m_i$ or $-m_i$ depending on whether A sends or receives m_i as a part of the protocol.

Example 2.2.3. Recall the Needham-Schroeder protocol from Example 2.2.1:

$$
\begin{aligned}
&1.\ A \to B : e(PK_B, A; N_A)\\
&2.\ B \to A : e(PK_A, N_A; N_B)\\
&3.\ A \to B : e(PK_B, N_B)
\end{aligned}
$$

Then, the strand space that models this protocol is:

$$A : [+(e(PK_B, A; N_A)), -(e(PK_A, N_A; N_B)), +(e(PK_B, N_B))]$$
$$B : [-(e(PK_B, A; N_A)), +(e(PK_A, N_A; N_B)), -(e(PK_B, N_B))]$$

It should be noted that the strand space model has no notion of time. As a result, it cannot naturally track the state of a partial execution. Therefore, Maude-NPA augments the strands with a vertical bar "|" that separates past messages from future messages [13].

Example 2.2.4. Recall the strand space from Example 2.2.3. Before the protocol begins executing, our augmented strands are the following:

$$A : [nil| + (e(PK_B, A; N_A)), -(e(PK_A, N_A; N_B)), +(e(PK_B, N_B))]$$
$$B : [nil| - (e(PK_B, A; N_A)), +(e(PK_A, N_A; N_B)), -(e(PK_B, N_B))]$$

Note that *nil* represents an empty list of signed terms.

Now suppose Alice has sent $+(e(PK_B, A; N_A))$, and Bob has received it. Furthermore, Bob has sent $e(PK_A, N_A; N_B)$, but Alice has not yet received it. Then our strands would be the following:

$$A : [+(e(PK_B, A; N_A))| - (e(PK_A, N_A; N_B)), +(e(PK_B, N_B))]$$
$$B : [nil| - (e(PK_B, A; N_A)), +(e(PK_A, N_A; N_B))| - (e(PK_B, N_B))]$$

Note that bar after the first term in Alice's strand, and after the second in Bob's strand.

2.2.4 Maude-NPA

Maude-NPA is a tool for cryptographic protocol verification using the strand space model [13]. It is built in Maude, and its specifications are written using Maude. A specification consists of three modules:

- PROTOCOL-EXAMPLE-SYMBOLS
- PROTOCOL-EXAMPLE-ALGEBRAIC
- PROTOCOL-SPECIFICATION.

PROTOCOL-EXAMPLE-ALGEBRAIC defines the signature, Σ and the axioms B.PROTOCOL-EXAMPLE-ALGEBRAIC defines the equations (i.e. algebraic properties) of the operators defined in PROTOCOL-EXAMPLE-SYMBOLS. In PROTOCOL-SPECIFICATION we define at least three equations:

- STRANDS-DOLEVYAO
- STRANDS-PROTOCOL

- Some number of ATTACK-STATE(N) for N a natural number.

STRANDS-DOLEVYAO defines the capabilities of the intruder (while the Dolev-Yao model says that the intruder can do anything that honest principals can do, "anything an honest principal can do" will often vary from specification to specification). STRANDS-PROTOCOL defines the strands of the honest principals, while ATTACK-STATE(N) allows us to specify an attack we would like to prove our protocol (in)secure against.

Example 2.2.5. The following is a Maude-NPA specification of the NS protocol from Example 2.2.1 using the strands from Example 2.2.3. Three dashes (`---`) represents comments. The operators `pk` and `sk` are used to represent encryption with a role's public and secret key respectively. The modules DEFINITION-PROTOCOL-RULES and DEFINITION-CONSTRAINTS-INPUT are part of the implemenation of Maude-NPA, and provide the built-in sorts `Msg`, `Fresh`, and `Public` amongst other things.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
    protecting DEFINITION-PROTOCOL-RULES .
    sorts Name Nonce Key .
    subsort Name Nonce Key < Msg .
    subsort Name < Key .
    subsort Name < Public .

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

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

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

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

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
    protecting PROTOCOL-EXAMPLE-SYMBOLS .
    var X : Name . var Z : Msg .
    eq pk(X, sk(X, Z)) = Z [variant] .
    eq sk(X, pk(X, Z)) = Z [variant] .
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 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 ] &
        :: nil :: [ nil | +(A), nil ]
[nonexec] .

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

var S : StrandSet . var K : IntruderKnowledge .

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
   || nil
[nonexec] .

eq ATTACK-STATE(1)
  = :: r ::
  [ nil, -(pk(b,a ; N)), +(pk(a,  N ; n(b,r))),
      -(pk(b,n(b,r))) | nil ]
   ||  empty
   || nil
   || nil
   || never
   (:: r' ::
   [ nil, +(pk(b,a ; N)), -(pk(a,  N ; n(b,r)))
      | +(pk(b,n(b,r))), nil ]
   & S
   || K)
[nonexec] .
```

The sort `Fresh` is an implementation detail. Variables of sort `Fresh` are used to enforce the uniqueness of entities like nonces or timestamps. Therefore, by defining an operator with an argument of sort `Fresh` (i.e. `n : Name Fresh -> Nonce`) we are saying that the result of the said operator will be unique across sessions. In order to properly enforce this guarantee, the strands must explicitly track all fresh variables generated by that strand. A fresh variable is generated by a strand if the fresh variable first appears inside of a sent (i.e. positive) term. The variables generated by a strand are specified between the double colons at the beginning of each strand. So Alice's strand generates the fresh variable r.

Intruder capabilities are also modeled as strands. However each intruder strand must follow the following structure: A list of zero or more received (i.e. negative) messages followed by a single sent (positive) message. An intruder strand $:: nil :: [-m_1, -m_2, \ldots, -m_n, +m]$ says that if the intruder knows (i.e. intercepted) the terms $m_1, m_2, \ldots, m_n$, then the intruder can learn the term m (i.e. send the term to himself/herself). For example, the strand $:: nil :: [nil| -(X), -(Y), +(X;Y), nil]$ says that the intruder can concatenate any messages he/she knows to obtain a new message.

Maude-NPA verifies cryptographic protocols by performing a form of state space exploration. Therefore, attacks are defined as states that should be unreachable during protocol execution. Each fragment of the state is separated by double bars `||`. First, we specify one or more strands in some stage of execution

(usually fully executed). This tells us which principal(s) is expected to have executed the protocol. Then, we specify a list of terms in the intruder knowledge. The next two state fragments (both containing `nil`) are implementation details. They are only used when debugging Maude-NPA (but not when debugging the specification). The final stage fragment is called a *never* pattern. Never patterns are themselves state patterns that may not appear in a valid path from the start state to the attack state.

For example, attack 0 asks whether it is possible for Bob to execute the protocol successfully, and for the intruder to learn Bob's nonce. Meanwhile, attack 1 asks whether it is possible for Bob to successfully execute the protocol without Alice successfully executing the protocol. In other words, is it possible for the intruder to impersonate Alice.

Finally, observe that the strands for Alice and Bob are not exactly what one would expect. A straight translation of the NS protocol into strands would suggest the following two strands:

```
---Alice
:: r ::
[ nil | +(pk(B,A ; n(A,r))),
        -(pk(A,n(A,r) ; n(B, r'))),
        +(pk(B, n(B, r'))), nil ] &
---Bob
:: r' ::
[ nil | -(pk(B,A ; n(A, r))),
        +(pk(A, n(A, r) ; n(B,r'))),
        -(pk(B, n(B,r'))), nil ]
```

However, instead the specification has the following two strands:

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

Observe that Bob's nonce is represented as the variable `N` in Alice's strand, and `N'` is used to represent Alice's nonce in Bob's strand. This is because Maude-NPA makes use of an idea called *perspective* [13]. Perspective will be explored in greater detail in Chapter 3.2.3, however the basic idea is that each principal can only derive a limited amount of information from the messages they receive. For example, Alice expects to receive a nonce from Bob, but she has no way

of knowing for certain that the nonce she received was in fact generated by Bob (proving this is the whole purpose of the protocol). Maude-NPA requires the user to make these holes in each principal's knowledge explicit by replacing unknown terms with variables of the appropriate sort (or of sort `Msg` if it is assumed the principals cannot even type check a message).

In short, Maude-NPA is a tool that uses the strand-space model to model cryptographic protocols. The strand space model relies on specifying each principal's actions as a strand: a sequence of sent and received messages. Maude-NPA is implemented in Maude, a declarative language based on order-sorted rewriting with axioms, which is a model of computation based on one-way equational reasoning with subtyping.

Chapter 3

Language Description

A Maude-PSL specification consists of four sections. Each section contains a sequence of statements, with each statement terminated by a space and period. The four sections are: *Theory*, *Protocol*, *Intruder*, and *Attacks*. *Theory* defines the equational theory, *Protocol* specifies the protocol, *Intruder* contains the intruder capabilities, and *Attacks* contains the types of attacks (e.g. authentication, secrecy violations) to check for.

We will explain the language using a specification of the Diffie-Helman(DH) protocol as a running example [8].

Example 3.0.6. The top level of the specification is as follows:

```
spec DH is
    Theory
        ...
    Protocol
        ...
    Intruder
        ...
    Attacks
        ...
ends
```

Variables must be declared within a section. Variables inside of the *Theory* and *Intruder* sections only have scope within their respective sections. Variable declarations are not allowed in the *Attack* section. Instead, the *Attack* section relies on the variables declared in the *Protocol* section. This is because the attacks depend upon the variables used in the protocol specification. Therefore, attacks should only use variables that appear in the protocol.

Example 3.0.7. In the following two statements, we declare two variables `AName` and `BName` of sort `Name`, and one variable `M` of sort `Msg`.

```
vars AName BName : Name .
vars N M : Msg .
```

Observe the use of the keyword `vars` when declaring multiple variables of the same sort, and `var` when declaring a single variable.

Variable names must be single identifiers.

With the exception of the protocol section, variable meanings are not preserved between statements. For example, the following two equations (equations are explained in more detail in Section 3.1):

```
eq pk(AName, sk(AName, M)) = M .
eq sk(AName, pk(AName, M)) = M .
```

are equivalent to the following two equations:

```
eq pk(AName, sk(AName, M)) = M .
eq sk(BName, pk(BName, N)) = N .
```

Note the use of `BName` instead of `AName`, and `N` instead of `M` in the fourth equation.

The section headers (*Theory*, *Protocol*, *Intruder* and *Attacks*) do not end in a space and period. Each specification is begun with the keyword `spec` and terminated with the `ends` keyword. `DH` is the name chosen for the Diffie-Helman specification.

We will now look at each section in turn.

3.1 Theory

In this section, we define the algebraic theory of the cryptographic protocol. The theory is a Maude functional module (for details, see [7]) with a few small additions, and some restrictions on user-defined operators. We will only focus on those aspects of Maude functional modules that are important to algebraic theories typically used in cryptographic protocols.

A theory is meant to define two things: The "language" in which the specification will be written, and the properties of the operations (e.g. the commutativity of XOR) that the protocol depends upon. The language is defined using a user-defined sort structure and set of operator symbols. The properties are defined using equations and operator attributes.

3.1.1 Type Structure

The type structure is used to model the capabilities of each principal to perform type-checking. A richer or poorer type structure may be used to model stronger or weaker type checking capabilities, respectively.

There are three built-in sorts: `Msg`, `Public`, and `Fresh`, with `Public` a subsort of `Msg`. `Msg` represents the set of all messages that may be sent during protocol execution. `Public` represents those messages that are publicly known (e.g. public keys). `Fresh` is a special sort that is used to help model session-specific values like nonces and timestamps (see Page 21).

Sorts are defined using one of the `sort`, `sorts`, `type`, or `types` keywords[1], followed by the name(s) of the new sort(s). Sort names are single identifiers. Multiple sorts may be declared in a single line by separating their names with whitespace.

Example 3.1.1. The following statement declares the types used by our specification of DH:

```
types Name Nonce MultipliedNonces
      Generator Exp Key GeneratorOrExp
      Secret .
```

`Name` represents the set of all principal names. `Nonce` is obvious. We will be using nonces to represent the secret exponents used by DH to construct the symmetric key. `MultipliedNonces` represents the exponents that have been multiplied together as part of building the shared key. A `Generator` is the public base used for generating keys. An `Exp` is a generator (or a generator raised to one or more exponents) raised to one or more exponents. Terms of sort `Secret` represent the data that Alice and Bob wish to share securely.

Subsort relations are defined using one of the following keywords: `subtype`, `subtypes`, `subsort`, or `subsorts`.

Example 3.1.2. The following is the subtyping structure for DH.

```
subtypes Generator Exp < GeneratorOrExp .
subtype Exp < Key .
subtypes Name Generator < Public .
subtype Nonce < MultipliedNonces .
```

So every `Generator`, and every `Exp` is a `GeneratorOrExp`. Meanwhile, an `Exp` is treated as a key. All names and all generators are publicly known. Furthermore a single `Nonce` is viewed as a degenerate set of `MultipliedNonces`.

The Maude-PSL automatically makes all user-defined sorts subsorts of `Msg`.

3.1.2 Operators

In order to specify a protocol, we first need to define a language for the operations (e.g. exponentiation, encryption, and decryption) performed during protocol execution.

The language of the protocol is defined by declaring operators. These operators are then used to build terms, which provide a symbolic representation of the actions performed. For example, the term `e(K, M)` may be used to represent a message `M` encrypted with the key `K`.

[1] We consider the notion of *type* and *sort* to be the same; therefore, the keyword `type`, (resp `types`) is equivalent to the keyword `sort` (resp. `sorts`).

Example 3.1.3. The following statements declare some of the operators needed by DH.

```
ops sec n : Name Fresh -> Secret .
ops a b i : -> Name .
ops e d   : Key Msg -> Msg .
```

The operator `sec` encodes secret information: the term `sec(AName, r)` (for `AName` a variable of sort `Name`, and `r` a variable of sort `Fresh`) represents some secret information known only by Alice. The nonce operator `n` is very similar, except that it is used to represent nonces rather than arbitrary secret data.

The constants `a`, `b`, and `i` represent concrete names of distinct principals (Alice, Bob, and the Intruder respectively).

The operators `e` and `d` represent encryption and decryption respectively.

Operator declarations that declare a single operator begin with *op*, while operator declarations that declare multiple operators begin with *ops*, as done above for `sec` and `n`.

Following the *op* keyword is a string of one or more tokens defining the operator. The operator name is terminated by a space and colon. When defining multiple operators in a single declaration, each operator must be separated from the others by whitespace. If an operator containing whitespace is declared using the *ops* keyword, then the operator must be wrapped in parentheses.

Following the colon is a space separated list of argument sorts, then an ASCII arrow, then the result sort.

In Example 3.1.3, the non-constant operators `sec`, `n`, `e`, and `d` are declared in *prefix syntax*. In other words, to encrypt a message `M` with a key `K`, we write `e(K, M)`. However, the Maude-PSL also allows the user to define operators using *mixfix* syntax.

Example 3.1.4. The following is the mixfix operator for message concatenation.

```
op _;_ : Msg Msg -> Msg .
```

The underscores represent where the operator's arguments are placed. For example, to concatenate two messages `M` and `N` we write `M ; N` (note the blank spaces between `M` and `;`, and between `;` and `N`). When declaring a mixfix operator, there must be exactly as many underscores as there are argument sorts.

Finally, it is possible to assign certain attributes to each operator. In particular, we can declare binary operators to be *associative*, *commutative*, or having a *unit* element, using the keywords `assoc`, `comm`, or `id: c` (with `c` a constant of the appropriate sort), respectively. For *parsing* purposes (but not really as equational axioms) we can avoid parentheses in a binary operator by requiring the parser to parse terms as left- or right-associative using the attributes `gather(E e)` or `gather(e E)` respectively.

Example 3.1.5. Here we make the concatenation operator right-associative, and declare also an associative-commutative operator for multiplying nonces.

```
op _;_ : Msg Msg -> Msg [gather (e E)] .
op _*_ : MultipliedNonces MultipliedNonces
    -> MultipliedNonces [assoc comm] .
```

In order to use the `comm` axiom, both operator arguments must have the same sort. In order to use the `assoc` axiom, both arguments and the result must have the same sort. For technical reasons, the Maude-NPA does not allow `assoc` without `comm`, or `id` without `comm` and `assoc`.

The unit axiom `id:` should be used carefully, if at all. Treating the unit as a built-in axiom can lead to unexpected non-termination, because the unit axiom allows Maude to insert new terms (the unit constant) into other terms at will. This is particularly problematic with the Maude-PSL, because the equational theory is not executed directly by the user. Rather it is used behind the scenes by a symbolic model-checker attempting to solve a semi-decidable problem. Therefore, it may be difficult to realize that the protocol verification is non-terminating, not because the search space is infinite, but because the unit axiom has led to unexpected looping.

Any legal attribute in Maude is also a legal attribute in Maude-PSL. In particular, this includes parsing precedence attributes, which may be useful if the user is specifying a large number of mixfix operators. Chapter 3.9 of the Maude Manual describes the operator precedences, while Chapter 4.4 describes all other operator attributes ([7]). Note however that the only *axioms* that are allowed are the `comm`, `assoc` and `unit` axioms as explained above. Axioms are any attributes that influence the rewrite relation itself, rather than just the writing and printing of terms.

To ease implementation, we assume that the user-defined operators are disjoint from the tokens used by the built-in syntax. For a list of built-in tokens, see Appendix 6.3.

3.1.3 Algebraic Properties

Now that we know how to define the language of the protocol, the next step is to specify the properties of the operators. Certain built-in properties (i.e. associativity and commutativity) are already defined alongside the operators, but what about others, such as the relationship between multiplication and exponentiation? Such properties are defined as equations between terms.

Example 3.1.6. Mathematically, we define the relationship between exponentiation and multiplication as follows:

$$(g^y)^z = g^{y*z}$$

for g the base of the exponentiation, and y, z exponents. In Maude-PSL, using the signature defined in Section 3.1.2, we write the above equation as follows:

```
var G : Gen . vars Y Z : MultipledNonces .
eq exp(exp(G, Y), Z) = exp(G, Y * Z) .
```

Each equation begins with the `eq` keyword, and ends with a space and period. Contrary to equations in the mathematical sense, which are viewed in a symmetric way (i.e. $(g^y)^z = g^{y*z}$ and $g^{y*z} = (g^y)^z$ are the same equation), in the Maude-PSL, the orientation of an equation matters. This is because the equations are used from left to right as simplification rules. Furthermore, how the equation is oriented may have an impact on both its executability by simplification and whether or not the theory terminates. To ensure termination, equations should be written (from left to right) from "complex" to "simple," the idea being that every term, no matter how complex, will eventually become too "simple" to reduce further. What "complex" and "simple" mean is dependent upon the theory. Furthermore, the equational theory must also be confluent.

Equations are specified *modulo* the attributes commutativity, associativity, and unit that may have been declared as part of the operator declarations. So we could have just as easily written `eq exp(exp(G,Y),Z) = exp(G, Z * Y) .` (where we have swapped the order of `Z` and `Y` in the multiplication). The commutativity of `*` means that both equations are considered identical by the Maude-PSL.

Note that because the Maude-PSL is translated into the strand-based language used by Maude-NPA, the equational theory must meet the restrictions imposed by Maude-NPA [13]. The restrictions are as follows:

1. If an operator has axioms, then those axioms must be commutative, commutative-associative, or commutative-associative-identity. So an associative operator is not allowed unless it is also commutative. Maude-PSL will throw an error if an operator has an incompatible combination of axioms.

2. The equations are confluent modulo the axioms. This is not automatically checked.

3. The equations are terminating modulo the axioms. This is not automatically checked.

4. The equations are coherent modulo the axioms [16]. The Maude-PSL automatically makes equations coherent modulo the axioms, as discussed in Chapter 2.1.4.

5. The equations have the finite variant property [14]. This is not automatically checked.

3.2 Protocol

To fully specify a protocol, we need to specify three pieces of information (and an optional fourth):

1. The protocol's input.
2. The protocol itself.
3. The protocol's output.
4. (Optional) Shorthand for terms, referred to as definitions.

The protocol's input and output are specified for each principal, while the protocol itself is specified using an extension of the standard Alice-Bob syntax.

However, before we begin we must first draw a very important distinction between three related but subtly different ideas: *roles*, *role names*, and *role patterns*.

3.2.1 Roles

A *role* is one of the jobs that can be performed by a principal to execute the protocol. At the very minimum, each protocol has an initiator role and a responder role. Many protocols also have a neutral third party role, such as for example, a key server. Each role has a particular sequence of messages that it must send and receive, as dictated by the protocol.

Meanwhile, a *role name* is the name we use to represent each role. For example, *Alice* (*A* for short) is the name often used for the initiator role. Similarly, *Bob* is the name often used for the receiver role, and *Server* can sometimes be used as the name of a neutral third party role.

In the Maude-PSL we must explicitly name each role. The roles are named using a space-separated list of identifiers prepended by the keyword `roles`:

```
roles role1 role2 ... roleN .
```

Example 3.2.1. Below, we name the two roles for DH: *A* (initiator) and *B* (responder) roles.

```
roles A B .
```

Note that the order in which the names are declared does not matter.

Finally, we have *role patterns*. A *role pattern* is a principal's approximation of a role name based on the limited information available to that principal. Role patterns are what variables of sort *Name* actually represent in our DH specification.

In the Alice-Bob notation, all three notions are conflated. Consider the first step of DH: $A \rightarrow B \ : A, B, g^{p_a}$. Here the initiator role with role name A is sending a message containing the concatenation of two role patterns representing the initiator's and the responder's name and half of the Diffie-Helman key. The responder then receives an encrypted message containing two role patterns and half of a Diffie-Helman key.

This distinction between roles, role names, and role patterns is important to the Maude-PSL's syntax, which endeavors to make explicit the *knowledge* that each protocol may *infer* when receiving a message at each stage of protocol execution. Roles are explicitly declared using the `roles` keyword declared above. Role patterns are represented using variables (typically of type `Name`).

So in summary, we have two ways of identifying roles:

- *role names* - These are the true names of each role. They are explicitly declared in the Maude-PSL using the `roles` keyword.
- *role patterns* - Potentially incorrect names associated with each role by the principals based on their limited knowledge.

To ease exposition, when it is clear from context that a role pattern is an accurate representation of a role name (e.g. the role pattern that represents Alice's knowledge of her own name), then we will refer to both as role names.

3.2.2 Principal Input

Each protocol makes certain assumptions about the starting knowledge of each principal (for example, that Alice knows Bob's name, or that Alice and Bob already have a shared key).

These assumptions are codified by specifying input for each role.

Example 3.2.2. The following two statements specify the input of Alice and Bob in the DH protocol.

```
vars ANAME BNAME : Name .
In(A) = ANAME, BNAME .
In(B) = BNAME .
```

Here, we assume that Alice knows both her name and Bob's (she knows Bob's name, because she wants to speak with him). Bob only knows his own name, because he doesn't know for certain that it is Alice, as opposed to Claire, who wishes to communicate with him.

The token `In` takes a role name as argument. The input is a comma-separated list of variables, and must be defined for every role (since all roles must at the very least know their own name, the input is guaranteed to be non-empty). Note that although in the above example, all inputs are names, in general the input variables may be of any sort.

3.2.3 Protocol Specification

The protocol itself is specified using an extension of the Alice-Bob syntax that makes explicit the viewpoint of each principal, and what each principal can infer when receiving a message.

Example 3.2.3. Consider the *Diffie-Helman* public key authentication protocol, spelled out below [8]. Note that technically, the Diffie-Helman requires the user to perform an exponentiation modulo a large prime number. This is meant to help ensure that the exponentiation cannot be easily brute forced. However we are only interested in attacks in which the attacker does not use brute force, but rather manipulates the execution of the protocol to trick the honest principals into revealing sensitive information, or to trick one honest principal into thinking the intruder is a different honest principal. Therefore, to simplify exposition, we will ignore the modulus operation, and focus only on the exponentiation.

A represents the principal Alice, and B represents the principal Bob. $M_1; M_2$ is the concatenation of messages M_1 and M_2. $e(K_1, M_1)$ is the encryption of message M_1 with key K_1. PK_A (resp. pkB) is the public key of A (resp. B). SK_A and SK_B are the associated secret keys. N_A is a nonce generated by A, and N_B is a nonce generated by B.

$$
\begin{aligned}
&1.\ A \to B\ : A; B, g^{p_a} \\
&2.\ B \to A\ : A; B; g^{p_b} \\
&3.\ A \to B\ : e((g^{p_b})^{p_a}, s)
\end{aligned}
$$

where p_a and p_b are the secret, large numbers used as exponents by A and B, g is the shared base, and s is the secret information to be exchanged.

The following is the specification in the Maude-PSL.

```
vars AName BName A1Name : Name .
vars r1 r2 r3 : Fresh .
vars XEA XEB : Exp .
var S : Secret .

1 . A -> B :  ANAME  ; BNAME ;
                exp(g, n(ANAME, r1))
           |- A1NAME ; BNAME ; XEB .
2 . B -> A :  A1NAME ; BNAME ;
                exp(g, n(BNAME, r2))
           |- ANAME  ; BNAME ; XEA .
3 . A -> B :  e(exp(XEA, n(ANAME, r1)),
```

```
        sec(ANAME, r3))
    |- e(exp(XEB, n(BNAME, r2)),
        S) .
```

Recall that `A` and `B` are the role names declared in Example 3.2.1.

The most obvious difference from the standard Alice-Bob notation is that each message is split it into two different *perspectives*. The first term represents the sender's perspective of the message he/she is sending, while the second is the receiver's perspective of the message he/she is receiving.

Perspective

In the standard Alice-Bob notation, the protocol is described from an omniscient point of view in an ideal world: we know exactly what is sent by the sender, and we know that the receiver always receives that exact message. However, things are messier in practice. Whenever a principal receives a message, he/she can only derive a limited amount of information from the message. For example, consider the message sent by Alice in the first step of DH: `AName ; BName ; exp(g, n(AName, r1))`. When Bob receives the message, he does not know *a priori* that it was sent by Alice, since he does not know ahead of time that he will be speaking to Alice. Furthermore, he does not know what the value of Alice's nonce is, so he cannot break the exponentiated value into its component pieces (base and exponent). Therefore, from his point of view, he receives a message of the form `A1Name ; BName ; XEB .`, where `A1Name` is some role pattern that may or may not be Alice's name, and `XEB` is some exponentiated value that may or may not have been generated by Alice.

The sharp-eyed reader may observe that when Alice sends the name `BName` in the first step, Bob receives that exact same name, `BName`. Recall from Section 3.2.2 that we assumed that both Alice and Bob know Bob's name. Therefore, Alice knows to send Bob's name, and Bob expects (and can verify) that the second name is his (we assume that although Bob does not know who he will be communicating with, he does know that he will be using DH).

A perspective implicitly encodes the capabilities of a principal to decode a message. Therefore, when building each perspective, the user has to ask himself/herself "What types of information can this principal reasonably derive?" For example, we know that Bob can look inside the term `e(exp(XEB, n(B, r2)), S)` and extract `S`, because the term is encrypted using the shared key he has constructed with Alice. Furthermore, thanks to the properties of exponentiation and multiplication, we know that Bob will see the key as some base `XEB` raised to his exponent `n(B, r2)`. Alice meanwhile, sees the key as `e(exp(XEA, n(A, r1))` because she knows the value of her nonce, but has no computationally easy way to derive Bob's nonce.

3.2.4 Role Names vs. Role Patterns

Observe that although B receives the name `A1Name` in the first step, in the second step he sends his message to A, instead of some role A1. This is because of the distinction between role names and role patterns discussed at the beginning of the chapter. The character A is a *role name*, while *A1Name* is a *role pattern*.

Because of this distinction, the following two specifications are equivalent to that found in example 3.2.3:

```
vars AName BName A1Name : Name .
vars r1 r2 r3 : Fresh .
vars XEA XEB : Exp .
var S : Secret .

roles initiator responder .

1 . initiator -> responder :  ANAME  ; BNAME ;
                              exp(g, n(AName, r1))
                           |- A1NAME ; BNAME ;
                                XEB .
2 . responder -> initiator :  A1NAME ; BNAME ;
                              exp(g, n(BNAME, r2))
                           |- ANAME  ; BNAME ;
                                XEA .
3 . initiator -> responder :
    e(exp(XEA, n(ANAME, r1)), sec(ANAME, r3)) |-
    e(exp(XEB, n(BNAME, r2)), S)
```

and

```
vars A B A1 : Name .
vars r1 r2 r3 : Fresh .
vars XEA XEB : Exp .
var S : Secret .

roles A B .

1 . A -> B :  A  ; B ; exp(g, n(A, r1))
           |- A1 ; B ; XEB .
2 . B -> A :  A1 ; B ; exp(g, n(B, r2))
           |- A  ; B ; XEA .
3 . A -> B :  e(exp(XEA, n(A, r1)), sec(A, r3))
           |- e(exp(XEB, n(B, r2)), S) .
```

Note in the third version that the same characters (`A` and `B`) are used to represent both the *role names* and the *role patterns*. This is perfectly legal,

and may make the specification more concise and closer to Alice-Bob notation. However, the user runs the risk of conflating role names and patterns if he/she uses this style.

Once a role name has been chosen, it must be used throughout the entire specification. For example, in the second version of the specification, where the initiator role is named *initiator*, the input must be specified as `In(initiator) = ANAME, BNAME ..`

The Maude-PSL enforces that all variables between roles are disjoint *with the exception of those declared in the role's inputs.* If the same variable shows up in both Alice's and Bob's inputs, then that same variable may be used in both Alice's and Bob's perspectives.

3.2.5 Principal Output

A principal's output is very similar to the input, and has a very similar syntax. A principal's output is a set of terms that the user deems to be "important," and that are (or can be) generated from terms in the protocol and input. For example, since the whole point of DH is to establish a shared key between Alice and Bob, the two perspectives on the shared key can be deemed important.

Example 3.2.4. The following are the outputs of Alice and Bob for our specification of DH

```
Out(A) = exp(XEA, n(A, r1)) .
Out(B) = exp(XEB, n(B, r2)) .
```

Note that the variables in the output terms of a particular role must have already appeared in the role's half of the protocol, or in the role's input. However, a term does *not* have to appear in the input or in the protocol specification in order to be considered output.

Output is meant to be used for protocol composition, and is included for future compatibility.

3.2.6 Definitions

In protocols of even moderate complexity (such as Otway-Rees [25]), the terms used in the specification quickly become painfully large. In complicated terms, it is often difficult to quickly separate out and mentally compartmentalize subterms into the purposes they are meant to serve (for example, recognizing the term `n(A, r1)` as the exponent in the term `exp(g, n(A, r1))`). Furthermore, it is not unusual in a cryptographic protocol for a term to be repeated many times, for example when encrypting multiple messages with the same key. As a result, any time that a term is modified, it needs to be manually changed everywhere in the protocol (and potentially in the attacks), which is highly error-prone.

To ease this burden we allow users to define what are essentially let statements for terms, referred to as *definitions*.

Example 3.2.5. The following are some definitions for our DH specification.

```
Def(A) = pa := n(AName, r1),
         secret := sec(AName, r') .
Def(B) = pb := n(BName, r2) .
```

Observe that the definitions are specific to each role. Furthermore, the definition names (`pa`, `s`, `pb`) of each role must be disjoint from the names used by other roles, and from variable and operator names.

Using these definitions gives us the following protocol specification:

```
1 . A -> B :  AName  ; BName ; exp(g, pa)
           |- A1Name ; BName ; XEB .
2 . B -> A :  A1Name ; BName ; exp(g, pb)
           |- AName  ; BName ; XEA .
3 . A -> B :  e(exp(XEA, pa), secret)
           |- e(exp(XEB, pb), S) .
```

This is both visually simpler than the specification in example 3.2.3 and also much closer to the original Alice-Bob description. Internally, each definition name is replaced by the corresponding term in a simple token replacement. So if we have a definition `xp := exp(g, M)`, and a term `exp(xp, M)`, then `exp(xp, M)` will be automatically replaced with `exp(exp(g, M), M)`.

Definition names may be any identifier and definitions may be used in other definitions (though be careful not to create circular dependencies). To illustrate, consider the following alternative set of definitions:

```
Def(A) = pa := n(A, r1),
            s := sec(A, r'),
         g^pa := exp(g, pa),
         xea^pa := exp(XEA, pa) .
Def(B) = g^pb := exp(g, pb),
             pb := n(B, r2),
         xeb^pb := exp(XEB, pb) .
```

Note that the order in which the definitions are defined does not matter. For example, we are defining `g^pb` using `pb` before we have defined `pb`. This is perfectly legal.

Using these definitions gives us the following specification:

```
1 . A -> B :  A  ; B ; g^pa
           |- A1 ; B ; XEB .
2 . B -> A :  A1 ; B ; g^pb
           |- A  ; B ; XEA .
```

```
3 . A -> B :  e(xea^pa, secret)
           |- e(xeb^pb, S) .
```

which is even closer to the original Alice-Bob description of the protocol. However, while on the face of it this may seem like an appropriate use of definitions, it is not. Definitions are meant to give names to complex terms. They are *not* meant to give the user a poor man's user-defined syntax. Especially since a very powerful system of user-defined syntax is already provided! If the user desires mixfix exponentiation, then the user should define a mixfix operator `_^_` and use that instead.

For an example of the proper use of non-trivial definitions, see Chapter 5, in which we specify the Otway-Rees protocol.

3.3 Intruder

The *Intruder* section defines the capabilities of the intruder. In the Maude-PSL it is assumed that the Intruder has complete control over the network: he/she can intercept messages, destroy messages, and inject his/her own messages into the communication at will. All that remains to be specified is what kinds of messages the Intruder can create, and what he needs to know to create them.

Therefore, an intruder capability takes the form of an implication: if the intruder knows terms $t_1, t_2, \ldots, t_n$, then he/she can learn $t'_1, t'_2, \ldots, t'_m$.

Example 3.3.1. The following are the intruder capabilities for the DH specification.

```
var r : Fresh .     var P : Name .
vars M1 M2 : Msg .  vars NS1 NS2 : Nonce .
var  K : Key .      var GE : GeneratorOrExp .

K, M1      => e(K, M1), d(K, M1) .
NS1, NS2   => NS1 * NS2 .
GE, NS1    => exp(GE, NS1) .
M1 ; M2   <=> M1, M2 .
           => n(i, r), g, P .
```

The statement
`K, M1 => e(K, M1), d(K, M1)` reads: "If the Intruder knows a key `K` and a message `M1`, then he/she can learn the terms `e(K, M1)` and `d(K, M1)`." In other words, if the intruder has a key, then he/she can encrypt an arbitrary message with said key, or attempt to decrypt an arbitrary message with said key.

The statement `M1 ; M2 <=> M1, M2 .` says that the Intruder can concatenate arbitrary messages, and split concatenated messages into their component pieces.

The statement `=> n(i, r), g, P.` says that the Intruder can generate his/her own nonces, he/she knows the generator being used by Alice and Bob, and he/she knows every role's name.

3.4 Attacks

The *Attacks* section contains a sequence of numbered attack patterns. Each attack pattern encodes an attack that the protocol needs to be proven secure against. Currently, the attack syntax is geared towards specifying two kinds of attacks: a violation of secrecy (in which the intruder learns a particular term that is supposed to be private), and a violation of authentication (in which Alice erroneously believes she has successfully executed the protocol with Bob).

Before we dive into the syntax for attack patterns, we first need to define the *variables of a role*. The variables of a role are represented as a set of all the variables that appear in the input and terms associated with a particular role.

Example 3.4.1. Recall the DH specification from example 3.2.3 with the input from example 3.2.2:

```
vars AName BName A1Name : Name .
vars r1 r2 r3 : Fresh .
vars XEA XEB : Exp .
var S : Secret .

In(A) = AName, BName .
In(B) = BName .

1 . A -> B :  AName  ; BName ;
                exp(g, n(AName, r1))
           |- A1Name ; BName ; XEB .
2 . B -> A :  A1Name ; BName ;
                exp(g, n(BName, r2))
           |- AName  ; BName ; XEA .
3 . A -> B :  e(exp(XEA, n(AName, r1)),
                sec(AName, r3))
           |- e(exp(XEB, n(BName, r2)),
                S) .
```

Then, the variables of the role A are `AName, BName, r1, XEA, r3`, while the variables for role B are `A1Name, BName, XEB, r2, S`.

When using definitions, the variables are computed *after* expanding the definitions. For example, consider a version of the above specification, where we use the definitions from example 3.2.5:

```
In(A) = AName, BName .
```

```
In(B) = BName .

1 . A -> B :  AName  ; BName ; exp(g, pa)
           |- A1Name ; BName ; XEB .
2 . B -> A :  A1Name ; BName ; exp(g, pb)
           |- AName  ; BName ; XEA .
3 . A -> B :  e(exp(XEA, n(AName, r1)),
                secret)
           |- e(exp(XEB, n(BName, r2)),
                S) .
```

Here, both A and B have the exact same variables as the first specification.

Each variable (with the exception of a variable of sort Fresh, which is solely an implementation artifact) represents a possible blind spot, i.e. a piece of data that could potentially be changed without affecting a "successful" execution of the protocol. For example, `AName` represents a blind spot in the sense that there is nothing forcing Alice to send her actual name to Bob, and Bob would never know: she could decide to send Claire's name instead. Meanwhile, the variable `XEB` encodes the fact that there is no guarantee that Bob actually receives the half-key generated by Alice: he might receive a half-key generated by the intruder. Determining whether or not these are *actual* blind spots is of course the entire purpose of the Maude-PSL and Maude-NPA.

Now, we can examine the syntax for specifying attacks.

Example 3.4.2. The following are four sample attack patterns from the DH specification.

```
0 .
    B executes protocol .
    Subst(B) = A1Name |-> a, BName |-> b,
        S |-> sec(a, r') .
    without:
        A executes protocol .
        Subst(A) = AName |-> a, BName |-> b .
1 .
    B executes protocol .
    Subst(B) = A1Name |-> a, BName |-> b,
        S |-> sec(a, r') .
    Intruder learns sec(a, r') .

2 .
    B executes protocol .
    Subst(B) = A1Name |-> a, S |-> sec(a, r') .
    With constraints BName != a .
```

```
Intruder learns sec(a, r') .
```

Note that the constants `a`, and `b` were declared in example 3.1.3 as constants of sort `Name`. The first attack represents an authentication attack, the second a secrecy attack. The third is similar to the second, except slightly more general (the role pattern `B` may be any term of sort `Name` except `a`).

Observe that no variables are declared. The Attack section uses all of the same variables as the Protocol section, because the semantics of the Attack and Protocol section are inextricably linked through variable instantiation. The vast majority of the time, the attack patterns and the protocol will use the same variables. Furthermore, most of the cases in which an attack pattern uses a new variable will be because the variable is misnamed. If the misnamed variable is not caught, then it will introduce a subtle error in which a protocol execution is not properly instantiated. Forcing the Attack section to have the same variables as the Protocol section allows us to easily catch these kinds of errors. Note that a variable does not have to be used in the Protocol section to be used in the Attack section, it merely needs to be declared in the Protocol section. Therefore, the user may still introduce new variables into an attack if so desired.

"`B executes protocol .`" tells us that Bob will fully execute his half of the protocol. It is also possible to specify a partial execution using "`B executes up to` N`.`", where N is the number of steps in the protocol Bob should execute. For example, suppose we wish to see if an attack is possible within the first two steps of the protocol. Then, we would write "`B executes up to 2 .`".

"`Subst(B) = A1 |-> a, B |-> b, S |-> sec(a, r') .`" defines a substitution instantiating a subset of the variables associated with Bob. Typically, this is used to fix the names in the protocol to particular names (represented by user-provided constants like `a`, `b`, or `i`). This allows us to define precisely which name Bob actually recieves. For example, in all three attacks, the fact that we are instantiating `A1` to `a` means that Bob is in fact receiving Alice's name. If we wished to consider an attack where Bob definitely receives someone else's name, we would instead map `A1` to either `i` or some other constant defined in the Equational Theory. For every statement of the form "`P executes protocol .`" there must be defined a corresponding substitution. However, substitutions may be provided for principals *without* a corresponding execution statement. Furthermore, the substitutions need to be well-formed order-sorted substitutions. If the user wishes to use the identity substitution, then the user must use the special keyword `id`. So the statement "`Subst(B) = id .`" defines Bob's substitution as the identity substitution. If the substitution is not the identity, then the domain of the substitution, and the variables contained in the range of the substitution should be disjoint.

The statement "`Intruder learns sec(a, r') .`" defines the intruder knowledge. If the intruder should know more than one term, then the terms

should be specified as a comma-separated list. Furthermore, the defined substitution will be applied to the intruder knowledge, so writing `Intruder learns sec(A1Name, r') .` is equivalent to writing `Intruder learns sec(a, r') .`. It should also be noted that terms in the intruder knowledge do not have to exactly match terms that appear in the protocol. In particular, the user is free to rename variables as he/she desires. For example, writing `Intruder learns sec(C, r') .` is equivalent to writing `Intruder learns sec(D, r') .` for variables `C`, `D` of sort name. Of course, in the above attacks, `Intruder learns sec(C, r') .` is not equivalent to writing `Intruder learns sec(A1Name, r') .` because `A1Name` will be instantiated to the constant `a`, but `C` will not.

The `without:` keyword allows us to specify protocol executions that *can't* happen during the specified attack. For example, attack 0 ask whether it is possible for Bob to execute the protocol without Alice executing the protocol as well. Intruder knowledge statements may not appear in without blocks. This is because of a subtlety in how Maude-NPA works. Maude-NPA works by performing backwards state space exploration. In other words, Maude-NPA begins at the attack state and goes backwards (i.e. into the past) until it reaches an initial state, or has explored all possible paths. There is another environment very similar to the `without:` blocks: the `state-space-reduction` blocks:

```
B executes protocol .
Subst(B) = A1NAME |-> a, B1NAME |-> b .
state-space-reduction:
    avoid:
        A executes protocol .
        Subst(A) = A1NAME |-> a .
        Intruder learns sec(a, r') .
    avoid:
        ...
```

The `avoid` blocks have no restrictions on which attack statements appear inside of them. The `state-space-reduction` environment is translated into the same thing as without blocks within Maude-NPA (never patterns). However, they are meant solely to optimize the search. They are *not* meant to contain any semantic information about the attack itself. Note that when one uses the state-space-reduction environment to optimize an attack, *one risks losing completeness of analysis.* In other words, if the specifier is not careful, he/she may optimize out a valid attack. Therefore, like all optimizations, the `state-space-reduction` environment is best avoided unless the specifier really understands how Maude-NPA works.

The statement `With constraints B != a .` allows us to impose certain restrictions on how the principal's terms may be instantiated during model checking. For example, the above statement says that the role pattern `B` may not be instantiated to the concrete name `a`. However, it may be instantiated

to any other term of sort `Name` (i.e. `b` or `i`) during the model checking process. Constraints are specified as a comma-separated list of disequalities of the form `T1 != T2` where `T1` and `T2` are arbitrary message terms. Multiple constraints are treated as a conjunction. Constraints are not used often, but they are for example invaluable when proving indistinguishability properties [27].

It is also possible to have multiple principals executing the protocol in a single attack pattern.

Example 3.4.3. The following is an attack pattern similar to pattern 1 in example 3.4.2, except with both Alice and Bob executing the protocol.

```
3 .
    B executes protocol .
    Subst(B) = A1Name |-> a, BName |-> b,
        S |-> sec(a, r') .

    A executes protocol .
    Subst(A) = AName |-> a, BName |-> b .

    Intruder learns sec(a, r') .
```

In this case, the union of all the defined substitutions is applied to the Intruder knowledge. So the attack in example 3.4.3 is equivalent to

```
4 .
    B executes protocol .
    Subst(B) = A1Name |-> a, BName |-> b,
        S |-> sec(a, r') .

    A executes protocol .
    Subst(A) = AName |-> a, BName |-> b .

    Intruder learns sec(AName, r') .
```

All substitutions must agree on shared variables. So in example 3.4.3, since Bob's substitution maps `B` to `b`, Alice's substitution must map `B` to `b`, and vice versa.

It is also possible to use the definitions in 3.2 in the attack patterns. However, if a definition is used, there must be a substitution defined for the associated role.

Example 3.4.4. The following is not a legal attack pattern, because the `secret` definition (which is associated with Alice) is being used in an attack pattern that only has a substitution defined for Bob's variables.

```
1 .
      B executes protocol .
```

```
Subst(B) = A1Name |-> a,
    BName |-> b, S |-> secret .
Intruder learns secret .
```

On the other hand, the following attack pattern is legal.

```
1 .
   A executes protocol .
   Subst(A) = AName |-> a, BName |-> b .
   Intruder learns secret .
```

because there is a substitution defined for Alice, for whom `secret` is defined.

A sharp-eyed reader may also observe that `secret` is equivalent to the term `sec(A, r')`. So a literal reading of the substitution would seem to suggest that `S` is being mapped to `sec(A, r')`, violating our requirement that the domain of the substitution be disjoint from the variables in the range. In order to deal with this, internally the Maude-PSL applies each substitution to itself until it reaches an idempotent fixed point. The resulting idempotent substitution is then used instead of the substitution defined by the user. Of course, it is possible to define a substitution for which there is no idempotent fixed point, such as $\{A \mapsto B, B \mapsto A\}$. To handle this possibility, the translator throws an error if it fails to reach a fixed point after 100 applications. An important consequence of the computation of an idempotent fixed point substitution is the implicit assumption that the user desires a consistent instantiation across all the terms in the attacks. If the user desires an inconsistent instantiation between protocol roles and intruder knowledge (say, instantiating the variable `AName` to `a` in Alice's substitution, but mapping the secret `sec(AName, r3)` in the intruder knowledge to `sec(b, r3)`), then the user must manually instantiate the Intruder's knowledge.

3.5 Conclusion

The syntax and organization above makes the Maude-PSL an expressive, and simple language. It allows the user to keep the different parts of the specification separate, which allow for easier reuse. It allows the user to define non-trivial algebraic properties for his/her protocol to use. It allows the user to specify the protocol in a manner very close to the high-level Alice-Bob definition. Finally, it provides the user with great flexibility and clarity when defining attacks.

The next question is: Is this specification language sufficiently precise? That is, does this language *formally specify* a protocol in sufficient detail to perform formal verification? This will be answered in the affirmative in the next section, in which we define the formal semantics of the language in terms of strands.

Chapter 4

Rewriting Semantics of PSL

In this chapter we define the formal semantics of the Maude-PSL as a rewrite theory $(\Sigma, E \uplus B, R)$ that transforms Maude-PSL programs into sets of strands, which have a well-understood semantics with respect to protocol verification [29]. The set of rules R contains the translation rules themselves, while the set of equations E defines a variety of helper functions, syntax desugaring, and structural simplifications. The set of equations B contains equational properties that are not amenable to oriented equations, such as the associativity and commutativity of operators.

This semantics has been implemented directly in Maude, and is used to translate Maude-PSL-specifications into Maude-NPA specifications. As a result, the following rewrite rules use the flavor of strands used by Maude-NPA. See Chapter 2.2.4 for details about Maude-NPA strands.

4.1 Bird's Eye View of the Translation

The translation takes place inside of multiset (technically, an ACU soup), which is represented by the sort *TranslationData* (see Figure 4.1 for a visual representation). The following are included inside a single *TranslationData* term:

1. The PSL-specification as an associative-commutative soup of sections: *Theory*, *Protocol*, *Intruder*, and *Attacks*. Each section contains an associative list of statements with the identity symbol *pass*.

2. Intermediate data structures that the PSL specification will be translated into, which are then translated into Maude-NPA code.

The bulk of the rules extract data from the provided PSL-statements, and then insert that data into the appropriate intermediate data structures. Once the entirety of the PSL-specification has been translated, these intermediate data structures are then converted into the Maude-NPA module PROTOCOL-SPECIFICATION (see Figure 4.2). Note that these semantics assume that the user-defined equational theory has already been converted into the PROTOCOL-EXAMPLE-SYMBOLS, and PROTOCOL-EXAMPLE-ALGEBRAIC modules. This is a trivial process (barely more than an automated copy-paste), the details of which are omitted.

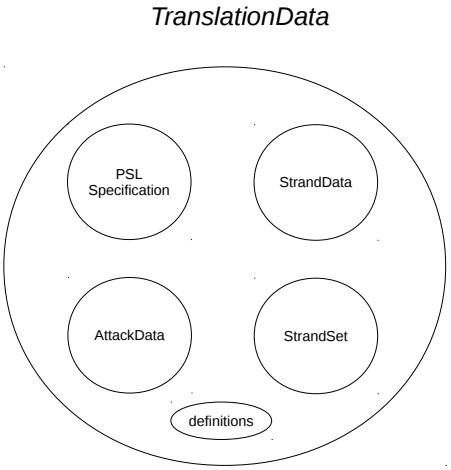

Figure 4.1: The largest oval represents the soup in which the other entities are floating. The smaller ovals represent distinct entities within that soup. *PSL Specification* is the PSL specification to be translated. *StrandData* is a mapping from roles to strands (used for building the protocol strands). *StrandSet* is a set of strands (used to build the Intruder capabilities). *AttackData* is a mapping from numbers to Maude-NPA states (used to build the Maude-NPA attack patterns). *Definitions* is the set of user-provided definitions.

4.2 Signature

The signature Σ decomposes into the disjoint union: $NPA \uplus P \uplus \Omega \uplus H$ where

1. *NPA* is the Maude-NPA signature [13].

2. P is all the user-defined operators, plus several enrichments. The enrichments are:

   (a) All user-declared sorts are made subsorts of `Msg` (a sort provided by the Maude-NPA signature) if they are not so already.

   (b) A free operator $\_\$;\_ : Msg\ Msg \rightarrow Msg$, which will be used internally, is added to the signature.

   (c) For every definition $T \coloneqq t$, with t a term of sort s, a constant $T :\rightarrow s$ is added to the signature.

   (d) All operators in P are made *frozen* if they are not so already. Essentially, if an operator is frozen, then Maude will not attempt to rewrite any subterms of the frozen operator. This allows Maude-NPA code to have complete control over how terms are rewritten. See the Maude Manual for more details about frozen operators [7].

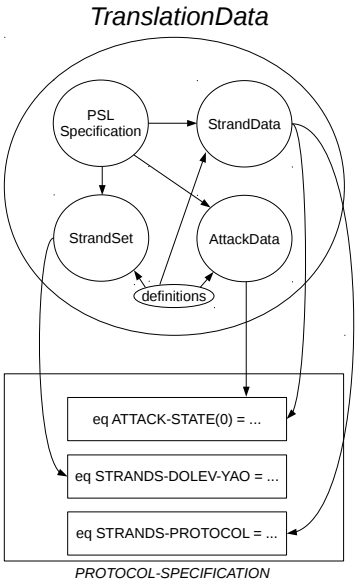

Figure 4.2: The square along the bottom of the diagram represents the Maude-NPA module *PROTOCOL-SPECIFICATION*. Each square contains the beginning of a (set of) statement(s) required by the *PROTOCOL-SPECIFICATION* module. *StrandData*, *AttackData*, and *StrandSet* are all populated using data extracted from *PSL Specification*, and *definitions*. The Maude-NPA attack states are built using information from *StrandData* and *AttackData*. The Dolev-Yao strands are built from *StrandSet*, while the protocol strands are built from *StrandData*.

3. Ω contains all of the operators and sorts used to encode the PSL-specification and data structures used in the translation.

4. H contains the symbols for internal helper functions.

4.2.1 Contents of Ω

Ω contains a sort `TranslationData` that consists of an ACU soup of data structures. The first data structure is a mapping from roles to triples. Each triple contains a strand, the strand's input, and the strand's output (see Chapter 2.2.4 for details about input and output of a strand):

$$
\begin{aligned}
mt &: \rightarrow \mathit{StrandData} \\
\_ \mapsto \{\_\}\_\{\_\} &: \mathit{Role}\ \mathit{MsgSet}\ \mathit{Strand}\ \mathit{MsgSet} \rightarrow \mathit{StrandDatum} \\
\_\&\_ &: \mathit{StrandData}\ \mathit{StrandData} \rightarrow \mathit{StrandData} \\
[\_] &: \mathit{StrandData} \rightarrow \mathit{TranslationData}
\end{aligned}
$$

where *StrandDatum* is a subsort of *StrandData*, the operator _&_ is ACU

with identity *mt*, and *Strand* is a sort defined by Maude-NPA.

Example 4.2.1. The following describes the initiator role of DH from Example 3.2.3 with the input from Example 3.2.2 and the output from Example 3.2.4 as a term of sort *StrandDatum*:

$$
\begin{aligned}
A \mapsto \{&AName, BName\} \\
&:: r1 :: [nil| \\
&\qquad + (AName; BName; exp(g, n(AName, r_1))), \\
&\qquad - (AName; BName; XEA), \\
&\qquad + (e(exp(XEB, n(BName, r_2)), S), \\
&\qquad nil] \\
\{&exp(XEA, n(A, r_1))\}.
\end{aligned}
$$

The subsignature Ω also contains an operator for associating numbers with attack patterns:

$$[\_ \mapsto \_] \ : \ Nat \ System \rightarrow AttackData$$

where *System* is a sort of Maude-NPA attack patterns.

Finally, Ω contains injection functions from *AttackData* into *TranslationData*, and from *StrandSet* (a sort provided by Maude-NPA) into *TranslationData*:

$$
\begin{aligned}
[\_] \ &: \ AttackData \rightarrow TranslationData \\
[\_] \ &: \ StrandSet \rightarrow TranslationData.
\end{aligned}
$$

4.3 Semantics

The following rules assume that some simple pre-processing has already been performed on the specification. In particular, it is assumed that when computation begins, the PSL-specification is floating in an AC soup with several other terms: a set of Definitions(see Chapter 3.2.6) that have already been extracted from the specification, an empty set of *StrandData* that will represent the protocol strands, and an empty set of strands that will represent the intruder capabilities. Note that both the empty set of *StrandData* and the empty *StrandSet* will be populated as computation progresses. Furthermore, we do *not* assume that the soup contains an empty set of attack patterns. The generation of the empty set of attack patterns is used to ensure that the *Protocol* section is translated before the *Attack* section. More details can be found in Section 4.3.3.

Example 4.3.1. Suppose the user has declared the following definitions as part of a larger PSL specification P:

```
Def(A) = nA := n(AName, r) .
Def(B) = nB := n(BName, r1) .
```

where `A` and `B` are roles, `n` is an operator for building nonces, `AName` and `BName` are variables of sort `Name` and `r` and `r1` are variables of sort `Fresh`. Then, we begin executing the semantics with the following term:

$$P\ [nA \coloneqq n(AName, r), nB \coloneqq n(BName, r1)]\ [empty.StrandSet]\ [mt.StrandData]$$

As computation proceeds, P will be eliminated one statement at a time, while the terms $[empty.StrandSet]$ and $[mt.StrandData]$ are populated. Once the set of *StrandData* has been fully populated, the term $[mt.AttackData]$ will be created and then populated.

4.3.1 Protocol

The *Protocol* semantics consist of two rules (not including error detection): one for processing the input and output, and one for processing each step in the protocol. The rules are shown in Figure 4.3.

The function *fresh* extracts the fresh variables from the passed term (if any), while *applyDefs* applies the user-provided definitions to the passed term. Both functions rely heavily on the reflective capabilities of rewriting logic, making them rather complex, and not particularly enlightening. Therefore, their definitions are omitted.

Rule 4.1 generates an empty strand for each role, while rule 4.2 populates the generated strands with the appropriate messages, following an algorithm inspired by work by Chevalier and Rusinowitch [6]. However, rule 4.1 does not fire unless a role has an input and an output associated with it. Furthermore, observe that the strands are actually generated with the time keeping bar (|) at the end of the strand, rather than at the beginning. However, this is immaterial. First, the notion of a present is a necessity of the implementation of the Maude-NPA, not an inherent property of strands. Furthermore, while the protocol strands require the bar to be at the beginning of the strand, the attack strands typically have the bar at the end. Therefore, when these strands are used to build a Maude-NPA specification, the bar will be shifted appropriately.

4.3.2 Intruder

The semantics for the *Intruder* section consists of a single rule, a helper function, and several equations that perform syntactic desugaring.

Figure 4.4 contains the desugaring equations, while Figure 4.5 contains the single rule.

The function *signedList* takes a list of messages, ms, and a single message, m. It returns a list of signed messages, where all the messages in ms are marked

$$
\begin{array}{l}
\textit{Protocol} \\
\quad p_1 \;\; \mathrm{In}(R) = i.\; p_2 \;\; \mathrm{Out}(R) = o.\; p_3 \\
s\ [d]\ [sd] \\
\longrightarrow \\
\textit{Protocol} \\
\quad p_1\ p_2\ p_3 \\
[d] \\
[R \mapsto \{i\} :: \mathit{nil} :: [\mathit{nil}|\mathit{nil}]\{\mathrm{applyDefs}(o, d)\}\ \&\ sd]
\end{array}
\tag{4.1}
$$

$$
\begin{array}{l}
\textit{Protocol} \\
\quad n.\ A \to B : m_A \vdash m_B. \\
\quad p_1 \\
[d] \\
[A \mapsto \{i_A\} :: f_A :: [l_A|\mathit{nil}]\{o_A\}\& \\
\ B \mapsto \{i_B\} :: f_B :: [l_B|\mathit{nil}]\{o_A\}\& \\
\ sd] \\
\longrightarrow \\
\textit{Protocol} \\
\quad p_1 \\
[d] \\
[A \mapsto \\
\quad \{i_A\} :: f_A, \mathrm{fresh}(\mathrm{applyDefs}(m_A, d)) :: \\
\quad [l_A, +(\mathrm{applyDefs}(m_A, d))|\mathit{nil}]\{o_A\}\& \\
\ B \mapsto \\
\quad \{i_B\} :: f_B :: \\
\quad [l_B, -(\mathrm{applyDefs}(m_B, d))|\mathit{nil}]\{o_A\}\& \\
\ sd]
\end{array}
\tag{4.2}
$$

Figure 4.3: Semantics for the *Protocol* Section. R is a role. p_1, p_2, p_3 are lists of statements in *Protocol*. d is the set of definitions (Section 3.2.6), sd is strand data (Section 4.2). Variables i and o are lists of messages. N is a natural number, A and B are both roles, m_A and m_B are both messages, and i_A, i_B, o_A, and o_B are lists of messages. Finally, l_A, and l_B are lists of signed messages.

with a minus sign, while m is marked with a plus sign. See Section 4.3.1 for an explanation of the functions *applyDefs* and *fresh*.

4.3.3 Attacks

The Attack semantics is the most complex, with two rules and three non-trivial helper functions. The first rule handles attacks that contain `without` blocks,

$$(\Rightarrow ms.) = (\emptyset \Rightarrow ms.)$$
$$(ms \Rightarrow m_1, m_2, ms_1.) = (ms \Rightarrow m_1.)(ms \Rightarrow m_2, ms_1.)$$
$$(ms_1 \Leftrightarrow ms_2.) = (ms_1 \Rightarrow ms_2.)(ms_2 \Rightarrow ms_1.)$$

Figure 4.4: Desugaring equations for intruder capabilities. The variables ms, ms_1, and ms_2 are lists of messages, while m, m_1 and m_2 are messages. The constant $\emptyset$ is an empty list of messages. Parentheses are solely to improve readability.

$$
\begin{array}{l}
\mathit{Intruder} \\
\quad ms \Rightarrow m. \\
\quad i \\
[s]\ [d] \\
\longrightarrow \\
\mathit{Intruder} \\
\quad i \\
[d] \\
[:: \mathrm{fresh}(m) :: [nil | \mathrm{signedList}(\mathrm{applyDefs}(ms, d), \mathrm{applyDefs}(m, d))]\ \&\ s]
\end{array}
$$

Figure 4.5: Translation semantics for the Intruder section. The variable ms is a list of messages, m is a message, and i is a sequence of PSL statements. The variable s is a set of strands, and d is the set of user provided definitions.

while the second rule handles attacks that do not contain `without` blocks.

Attack statements are split into two categories: the core block, and a set of `without` blocks. Core blocks are sets of execution statements (`A executes protocol .`), substitution statements (`Subst(A) = A |-> a .`), intruder knowledge statements (`Intruder learns a .`), and constraint statements (`With constraints a != B .`). A `without` block is a core block prefixed by the `without:` keyword. Observe that in the semantics, a `without` block has fewer restrictions than they do at the syntactic level (see Chapter 3.4). This is because `without:` and `avoid` blocks are turned into never patterns. By making the semantics more flexible than the syntax allows, we can reduce `avoid` blocks to `without` blocks.

Example 4.3.2. Consider the attack from Example 3.4.2:

```
0 .
    B executes protocol .
    Subst(B) = A1Name |-> a, BName |-> b,
        S |-> sec(a, r') .
    without:
        A executes protocol .
        Subst(A) = AName |-> a, BName |-> b .
```

Here, `B executes protocol .` and `Subst(B) = A1Name |-> a, BName |-> b, S |-> sec(a, r')` are part of a core block, while `A executes protocol .` and `Subst(A) = AName |-> a, BName |-> b .` form a `without` block.

The semantics for the *Attack* section can be found in Figure 4.6. K and S in the rules in Figure 4.6 are constants that Maude-NPA will interpret as variables of sort *IntruderKnowledge* and *StrandSet* respectively.

In order to guarantee that the strands have been fully processed before we use them to populate the attacks, we also have the following equation:

$$Protocol\ pass = [mt.AttackData]$$

where $mt.AttackData$ is the identity element for attack data. Since both rules in Figure 4.6 assume that a set of attack data already exists somewhere in the soup (represented by $[ad]$ in both rules), neither rule will fire until the *Protocol* section has been completely translated.

Unlike in the other sections, the helper (partial) functions used for translating the attacks are non-trivial, and each will be considered in turn. The first we will look at is *subst*, which is defined in Figure 4.7. This function accomplishes two tasks. First, it takes the disjoint union of the attack substitutions, to get a new substitution θ. Then, it attempts to build an idempotent substitution from θ as described in Chapter 3.4. To accomplish this, *subst* depends on three partial functions: *makeIdem*, *isValid* and *extractSubst*. The function *extractSubst* is a straightforward function that extracts and takes the union of all substitution statements defined in the passed core attack. *isValid* meanwhile checks that the substitution, θ, returned by *extractSubst* is a valid order-sorted substitution. If θ is an order-sorted substitution, then *isValid* returns the substitution unchanged. Otherwise, *isValid* is undefined. *makeIdem* attempts to make the passed substitution idempotent. The definitions of all three functions can also be found in Figure 4.7.

The function *containsSubst* returns *true* if the passed core attack has at least one substitution statement, and *false* otherwise. Given a substitution, θ, the partial function *checkSorts* returns θ iff θ is a well-formed order-sorted substitution. Much like *applyDefs*, *checkSorts* relies on the reflective capabilities of rewriting logic. As a result, despite being relatively straightforward (for each mapping $x \mapsto t$, check if the sort of t is a subsort of the sort of x), the definition is rather technical. Therefore, it is ommitted.

Next, consider the function *genAttStrands*, defined in Figure 4.8.

The function *prefix* takes a strand, $:: r :: [\pm m_1, \pm m_2, \ldots, \pm m_n, \ldots, \pm m_p]$ and a natural number n, and returns a strand of the form
$:: r :: [\pm m_1, \pm m_2, \ldots, \pm m_n | L]$ where L is a constant representing a list of signed messages, which will be treated as a variable by Maude-NPA. In other words, when a role R executes up to N, the role is actually executing the first N steps of the role, and then doing whatever he/she wants, be it the rest of the protocol,

$$
\begin{array}{l}
\mathit{Attacks} \\
\quad n.\ c\ w \\
\quad a \\
[d]\ [sd]\ [ad] \\
\longrightarrow \\
\mathit{Attacks} \\
\quad a \\
s\ [d]\ [sd] \\
[[n \mapsto \mathrm{genAttStrands}(c, \mathrm{subst}(c), sd, d)\ \&\ S \\
\quad ||\, \mathrm{genIntKnowledge}(c, \mathrm{subst}(c), d),\ K \\
\quad ||\mathit{nil} \\
\quad ||\mathit{nil} \\
\quad ||\, \mathrm{never}(\mathrm{genNeverPatterns}(w, sd, d))] \\
ad]
\end{array}
$$

$$
\begin{array}{l}
\mathit{Attacks} \\
\quad n.\ c \\
\quad a \\
s\ [sd]\ [d]\ [ad] \\
\longrightarrow \\
\mathit{Attacks} \\
\quad a \\
[d]\ [sd] \\
[[n \mapsto \mathrm{genAttStrands}(c, \mathrm{subst}(c), sd, d)\ \&\ S \\
\quad ||\, \mathrm{genIntKnowledge}(c, \mathrm{subst}(c), d),\ K \\
\quad ||\mathit{nil} \\
\quad ||\mathit{nil} \\
\quad ||\mathit{nil}]]
\end{array}
$$

Figure 4.6: The semantics for attacks. The first rule handles the case where an attack contains at least one `without` block. The second handles the case where there are no `without` blocks. The variable n is a number, c is a set of core attack statements, and w a set of without blocks. The variable a represents the rest of the attack section, while d is the set of definitions. sd is the strand data computed by the rules in Figure 4.3, and ad is the set of other attacks that have already been translated.

$$
\begin{aligned}
\text{subst}(c, d) &= \text{makeIdem}(\text{isValid}(\text{extractSubst}(c, d))) \\
\text{extractSubst}(\mathit{Subst}(r) = \theta.\ c, d) &= \text{applyDefs}(\theta, d), \text{extractSubst}(c, d) \\
\text{extractSubst}(c, d) &= \mathit{none}\ \mathit{if}\ \neg \text{containsSubst}(c) \\
\text{isValid}(\theta) &= \text{checkSorts}(\text{isFunction}(\theta)) \\
\text{isFunction}(m_1 \mapsto m_2, m_1 \mapsto m_2, \theta) &= \text{isFunction}(m_1 \mapsto m_2, \theta) \\
\text{isFunction}(m_1 \mapsto m_2, \theta) &= m_1 \mapsto m_2, \text{isFunction}(\theta)\ \mathit{if}\ m_1 \notin \text{Dom}(\theta) \\
\text{isFunction}(\emptyset) &= \emptyset \\
\text{makeIdem}(\theta) &= \text{makeIdem}(\theta, \theta, \mathit{false}, 0) \\
\text{makeIdem}(\theta_1, \theta_2, \mathit{false}, n) &= \text{makeIdem}(\theta_1, \theta_2\theta_1, \theta_2 == \theta_2\theta_1, s(n)) \\
&\quad \mathit{if}\ n \leq 100. \\
\text{makeIdem}(\theta_1, \theta_2, \mathit{true}, n) &= \theta_2
\end{aligned}
$$

Figure 4.7: Definition of the function *subst*, which returns the substitution used to instantiate attack strands. The variable c is a set of attack statements, d the user-defined definitions, and θ, θ_1, and θ_2 are (potential) substitutions. Variables m_1 and m_2 are of sort *Msg*, and n is a natural number.

$$
\begin{aligned}
\text{genAttStrands}(R\ \mathit{executes\ protocol}.\ c, \theta, R \mapsto \{i\}s\{o\}\ \&\ sd, d) &= \\
\text{applyDefs}(s, d)\theta\ \&\ \text{genAttStrands}(c, \theta, R \mapsto \{i\}s\{o\}\ \&\ sd, d) & \\
\text{genAttStrands}(R\ \mathit{executes\ up\ to}\ n.\ c, \theta, R \mapsto \{i\}s\{o\}\ \&\ sd, d) &= \\
\text{applyDefs}(\text{prefix}(s, n), d)\theta\ \&\ \text{genAttStrands}(c, \theta, R \mapsto \{i\}s\{o\}\ \&\ sd, d) & \\
\text{genAttStrands}(c, \theta, sd, d) &= \\
\emptyset\ \mathit{if}\ \neg \text{hasExecutionStmt}(c) &
\end{aligned}
$$

Figure 4.8: Definition of the partial function *genAttStrands*. R is a role, c a set of core attack statements, θ is an order-sorted substitution, i a set of variables, s is a strand, and o a set of terms. sd is a set of strand data, d is a set of definitions, and n is a natural number.

nothing, or some random set of messages. The predicate *hasExecutionStmt* checks whether or not there is an execution statement in a core attack c.

The partial function *genIntKnowledge* is defined in Figure 4.9. The constraint statements are handled by *genIntKnowledge* as well, because Maude-NPA uses the same part of the state for both the disequality constraints and the intruder knowledge (though conceptually, disequality constraints are *not* a part of the intruder knowledge). Furthermore, since both the Maude-PSL and Maude-NPA use the same syntax for disequality constraints, no special transformations need to be performed. Finally, *hasIntruderStmt* is a predicate on core attacks that returns `true` if there is at least one intruder knowledge or constraint statement in the passed core attack.

Finally, we have the function *genNeverPatterns*, defined in Figure 4.10. Ob-

$$\begin{aligned}
\text{genIntKnowledge}(\textit{Intruder learns } M.\ c, \theta, d) &= \text{inI}(\text{applyDefs}(M, d)\theta), \\
&\quad \text{genIntKnowledge}(c, \theta, d) \\
\text{genIntKnowledge}(\textit{ With constraints } E.\ c, \theta, d) &= \text{applyDefs}(E, d)\theta, \\
&\quad \text{genIntKnowledge}(c, \theta, d) \\
\text{genIntKnowledge}(c, \theta, d) &= \emptyset \textit{ if } \neg\, \text{hasIntruderStmt}(c) \\
\text{inI}(m, M) &= m \textit{ inI}, \text{inI}(M) \\
\text{inI}(\emptyset) &= \textit{nil}
\end{aligned}$$

Figure 4.9: Definition of *genIntKnowledge*. M is a set of messages, c is a set of core attack statements, θ is an order-sorted substitution, and d is a set of definitions. E is a set of disequalities between messages, and m is a message.

serve that each never pattern is computed using its own substitution, derived from the substitution statements defined as part of the without block. As such, the substitution for each `without` block is independent of the substitution computed for the core of the attack, and of substitutions for other `without` blocks.

$$\begin{aligned}
\text{genNeverPatterns}((\textit{without} : c)w, sd, d) &= \text{neverPattern}(c, sd, d) \\
&\quad \text{genNeverPatterns}(w, sd, d) \\
\text{genNeverPatterns}(\emptyset) &= \textit{nil} \\
\text{neverPattern}(c, sd, d) &= \text{genAttStrands}(c, \text{subst}(c), sd, d)\&S \\
&\quad || \, \text{genIntKnowledge}(c, \text{subst}(c, d)), K
\end{aligned}$$

Figure 4.10: Semantics for the partial function *genNeverPatterns*. c is a set of core attack statements, w is a set of without blocks, sd is a set of strand data, and d a set of definitions. S and K are constants of sort *StrandSet* and *IntruderKnowledge* respectively, which will be treated as variables by the generated Maude-NPA specification.

From here, it is a simple matter to extract the generated strands and Maude-NPA states, and wrap them in a Maude-NPA module. This part of the semantics is implemented in Maude, with some minor syntactic modification, and is used as the final stage of the translation from a Maude-PSL specification to Maude-NPA modules.

Chapter 5

Case Study

In this chapter we do an in-depth analysis of the Otway-Rees protocol [25]. The Otway-Rees protocol is meant to provide efficient, timely (i.e. no replay attacks possible) mutual authentication without the explicit use of clocks. Otway-Rees is interesting because it has a much more complicated term structure than the Diffie-Helman protocol, and it makes use of more than two principals. As a result, Otway-Rees allows us to see how the Maude-PSL handles a more complicated protocol. However, the Otway-Rees protocol is still simple enough to make a thorough study of the specification, translation, and verification of the protocol a worthwhile endeavor.

Following is the Otway-Rees protocol in Alice-Bob notation. C represents a conversation identifier generated by Alice. The conversation identifier is meant to be used as a nonce: a means of marking the conversation to protect against replay attacks. SK_A and SK_B are the secret keys of A and B respectively. R_1 is Alice's challenge, a unique value generated by Alice and used by her to ensure that the session key she receives from the server actually came from the server. K_C represents the session key to be used between A and B.

1. $A \rightarrow B : C; A; B; e(SK_A, R_A; C; A; B)$
2. $B \rightarrow S : C; A; B; e(SK_A, R_A; C; A; B); e(SK_B, R_B; C; A; B)$
3. $S \rightarrow B : C; e(SK_A, R_A; K_C); e(SK_B, R_B; K_C$
4. $B \rightarrow A : C; e(SK_A, R_A; K_C)$

5.1 Specification

The first step is to define the language (i.e. operators) we will use to express the protocol. The bare minimum that we need is a means of representing session-only values (i.e. nonces), a means of encrypting and decrypting messages with keys, and a means of concatenating messages. With that in mind, consider the following *Theory* section:

```
Theory
     types UName SName Name Key Nonce Masterkey
        Sessionkey .
     subtypes Masterkey Sessionkey < Key .
```

```
subtypes SName UName < Name < Public .
op n : Name Fresh -> Nonce .
ops a b i : -> UName .
op s : ->  SName .
op mkey : Name Name -> Masterkey .
op seskey : Name Name Nonce -> Sessionkey .
op e  : Key Msg -> Msg . //encryption
op d : Key Msg -> Msg . //decryption
op _;_ : Msg Msg -> Msg [gather (e E)] .

var K : Key .
var Z : Msg .
eq d(K, e(K, Z)) = Z .
eq e(K, d(K, Z)) = Z .
```

A master key corresponds to the SK_A and SK_B in the original specification. This is a key shared by a principal and the server, and only the principal and server.

Observe that our types are more precisely defined than may be strictly necessary. For example, distinguishing between master and session keys is not strictly necessary. Indeed, the encryption and decryption equations make no distinction between master and session keys. However, a more precise type system often makes Maude-NPA more efficient. Maude-NPA will not waste time exploring paths in which the protocol uses a master key like a session key. The downside is that any attacks that rely upon a confusion of types will not be caught. If the specifier is concerned about attacks that rely on confusing types, then a weaker type system must be used (at the extreme end, in which we assume no typing information at all, every term would be of sort `Msg`). Therefore, it is recommended a specifier starts with a more precise type system first, and correct any attacks found with the stronger type system. Then, gradually weaken the types until either the protocol has been proven secure at all levels of type sophistication desired, or Maude-NPA fails to terminate.

Next we need to specify the protocol. The terms in the Otway-Rees protocol are fairly complex, and can be difficult to process. Therefore, we will be making use of definitions to blackbox some of these terms, and give them a name that hints at their meanings, as well as matching them up more closely with names given them in the Alice-Bob notation. As a convention, definition names start with a lower case letter, in order to distinguish them from variables, which are (with the exception of variables of sort Fresh, since those are special) in all caps.

```
Protocol
    vars ANAME BNAME : UName .
    vars ANAME1 BNAME1 : UName .
    var SNAME : SName .
```

```
vars r r' r'' rM : Fresh .
vars RA CB CS RB : Nonce .
vars M1 MA : Msg .
var KCA KCB : Sessionkey .

roles A B S .

Def(A) = c   := n(ANAME, rM),
          ra  := n(ANAME, r),
         skA  := mkey(ANAME, SNAME) .
Def(B) = rb  := n(BNAME, r'),
         skB  := mkey(BNAME, SNAME) .
Def(S) = skA := mkey(ANAME, SNAME),
         skB := mkey(BNAME, SNAME),
          kc := seskey(ANAME, BNAME,
                    n(SNAME, r'')) .


In(A) = ANAME, BNAME, SNAME .
In(B) = BNAME, SNAME .
In(S) = ANAME, BNAME, SNAME .

1 . A -> B :  c  ; ANAME  ; BNAME ;
            e(skA, ra ; c ; ANAME ; BNAME)
           |- CB ; ANAME1 ; BNAME ; M1 .


2 . B -> S :  CB ; ANAME1 ; BNAME ;
                M1 ;
                e(skB, rb ; CB ; ANAME1 ; BNAME)
           |- CS ; ANAME ; BNAME ;
                e(skA, RA ; CS ; ANAME ; BNAME) ;
                e(skB, RB ; CS ; ANAME ; BNAME) .


3 . S -> B :  CS ; e(skA, RA ; kc) ;
                e(skB, RB ; kc)
           |- CB ; MA ;
                e(skB, rb ; KCB) .


4 . B -> A :  CB ; MA
           |- c  ; e(skA, ra ; KCA) .
```

```
    Out(A) = c, ra, KCA .
```

Recall that our role names are arbitrary, and do not need to conform to the Alice-Bob convention. The role identifiers used are purely a matter of taste. To get a feel for how different role identifiers look for a slightly more complex protocol, consider the following slight modification to the specification above.

```
Protocol
    vars I R : UName .
    var S    : SName .
    vars r r' r'' rM : Fresh .
    vars RI CR CS RR : Nonce .
    vars M1 MI : Msg .
    var KCI KCR : Sessionkey .

    roles init resp server .

    Def(init)   = c   := n(I, rM),
                  ri    := n(I, r),
                  ski   := mkey(I, S) .

    Def(resp)   = rr  := n(R, r'),
                  skr   := mkey(R, S) .

    Def(server) = ski := mkey(I, S),
                  skr   := mkey(R, S),
                kc  := seskey(I, R, n(S, r'')) .



    In(init) = I, R, S .
    In(resp) = R, S .
    In(server) = I, R, S .

    1 . init -> resp :  c ; I  ; R ;
                        e(ski, ri ; c ; I ; R)
                  |-       CB ; I1 ; R ;
                        M1 .


    2 . resp -> server :  CB ; I1 ; R ;
                        M1 ;
                        e(skB, rr ; CB ; I1 ; R)
```

```
            |-          CS ; I  ; R ;
                  e(ski, RI ; CS ; I ; R) ;
                  e(skr, RR ; CS ; I ; R) .


3 . server -> resp :  CS ;
                  e(ski, RI ; kc) ;
                  e(skr, RR ; kc)
            |-          CR ;
                        MI ;
                  e(skr, rb ; KCR) .


4 . resp -> init :  CR ;
                    MI
                 |- c  ;
                    e(ski, ri ; KCI) .

Out(init) = c, ri, KCI .
Out(resp) = rr, KCR, MI, M1, CR .
Out(server) = kc, RI, RR, CS .
```

The advantage here is that the role identifiers are even more distinct from the names that each principal associates with each role. However, the meaning is exactly the same as the original specification.

Recall that the Maude-PSL requires that variables associated with each principal be disjoint, with the possible exception of those variables declared as part of each principal's input. Similar, only those definitions declared for a specific principal may be used in that principal's terms. As a result, we are forced to use a separate variable `CB` and `CS` to represent Bob's and the server's perspectives on the conversation identifier `c` (`n(A, rM)`). We are also forced to declare the definition `skA` and `skB` twice. While variable sharing does not affect the semantics of the translation (since Maude-NPA automatically renames the variables of each strand to make them disjoint), the Maude-PSL requires (mostly) disjoint variables for two reasons. The first is to emphasize all the possible cracks in the protocol. For example, the fact that `CB` becomes `CS` in the second step emphasizes that the server does not know for certain that the conversation identifier it received is the same as the identifier that Bob sent. Second, disjoint variables increases the chances that we will catch a subtle semantic error arising from sloppy copy-paste. For example, observe that Bob's perspective on the term `e(skB, rb ; CB ; ANAME1 ; BNAME)` is very similar to the server's perspective on the same term `e(skB, RB ; CS ; ANAME ; BNAME)`. Therefore, the user would be tempted to just copy and paste Bob's term. However, if the user

does not properly modify the term to reflect the server's perspective, then a very subtle semantic error is introduced into the specification. In particular, if the server receives the term `e(skB, rb ; CS ; ANAME1 ; BNAME)`, then this suggests that the server should be able to recognize that the nonce it receives was generated by Bob. However, this is clearly incorrect. As a result, any attacks that rely on that uncertainty would not be caught, but Maude-NPA would not complain. Furthermore, it is highly unlikely that such a subtle mistake would be caught solely by eyeballing the terms. However, in this case, the Maude-PSL will throw an error because a definition `rb` appears in an S-term, but has not been declared for use by the server. Therefore, the user becomes aware of this mistake, and can correct it by replacing `rb` with `RB`.

Intruder capabilities are standard. The intruder needs to be able to know all names, including the server's name. The intruder should be able to generate a masterkey between himself/herself and any other principal. Similarly, the intruder should be able to encrypt/decrypt any message in his/her possession with any key in his/her possession, and should be able to concatenate messages, and split apart concatenated messages. This gives us the following intruder capabilities:

```
Intruder
    vars P : UName .
    vars K : Key .
    vars N M : Msg .

          => s, P, mkey(i, s) .
    M, N <=> M ; N .
    K, M  => d(K, M), e(K, M) .
    P     => mkey(P, i) .
```

Now, for the attacks. We are interested in two attacks: one in which the intruder learns the session key even though Alice successfully executes the protocol, and one in which the intruder successfully tricks Alice into thinking she has executed the protocol with Bob, when in fact Bob has not. In other words, an attack in which Alice executes the protocol, but Bob does not. However, to simplify the search as much as possible, we are going to assume that Bob does receive the first encrypted message that Alice sent. In other words, we shall instantiate `M1` in Bob's half of the protocol to `e(skA, ra ; c ; ANAME ; BNAME)`. This is a trade-off between coverage and speed. Instantiating `M1` immediately throws out any possible attacks that rely on the intruder replacing `e(skA, ra ; c ; ANAME ; BNAME)` with a different term while en route to Bob. At the same time, Maude-NPA will not bother exploring any paths that rely on the intruder intercepting `e(skA, ra ; c ; ANAME ; BNAME)` with something else, so the search space will be that much smaller.

Note that we also have a third attack (attack 0) that does not actually check for any violations of secrecy or authentication. This is an "empty" attack, and is meant to help the user debug the specification. If attack 0 *fails* to find an attack, then there is an error in the specification. This is because attack 0 simply says that one of the principals executes the protocol. There should always be at least one way to execute the protocol: a successful execution. Therefore, if Maude-NPA cannot find a way to successfully execute the protocol, then the protocol is improperly specified.

```
Attacks
   0 .
         A executes protocol .
         Subst(A) = ANAME |-> a, BNAME |-> b,
            SNAME |-> s .
   1 .
         A executes protocol .
         Subst(A) = ANAME |-> a, BNAME |-> b,
            SNAME |-> s .
         Intruder learns KCA .
   2 .
         A executes protocol .
         Subst(A) = ANAME |-> a, BNAME |-> b,
            SNAME |-> s .
         without:
             B executes protocol .
             Subst(A) = ANAME |-> a, BNAME |-> b,
                SNAME |-> s .
             Subst(B) = M1 |-> e(skA, c ; na ; ANAME ;
                                 BNAME),
                         ANAME1 |-> a, BNAME |-> b,
                         SNAME |-> s .
```

Observe that no variables are declared in the *Attack* section. Recall that the *Attack* section uses the same variables as the *Protocol* section. This ensures that we do not try to instantiate variables that do not actually appear in the *Protocol* section. Furthermore, observe that the second attack gives us an example of a non-idempotent substitution after definition expansion:

```
Subst(B) = M1 |-> e(mkey(ANAME, SNAME), n(ANAME, rM) ;
             n(ANAME, r) ; ANAME ; BNAME),
    ANAME1 |-> a, BNAME |-> b, SNAME |-> s .
Subst(A) = ANAME |-> a, BNAME |-> b, SNAME |-> s .
```

However the Maude-PSL will automatically generate the idempotent solution

```
    M1 |-> e(mkey(a, s), n(a, rM) ; n(a, r) ; a ; b),
ANAME  |-> a, BNAME |-> b, SNAME |-> s,
ANAME1 |-> a .
```

during the translation process. Note: If you do *not* wish to fully instantiate `e(keyAS, c ; ra ; ANAME ; BNAME)` (say you do not wish to instantiate the last two occurrences of `ANAME, BNAME`), then you need to rename those variables. For example, we could add the variable declaration `vars ANAME1 BNAME1 : UName` to the *Protocol* section, and then map `M1` to `e(keyAS, c ; ra ; ANAME1 ; BNAME1)`. Then, Maude-PSL will only instantiate those occurrences of `ANAME, BNAME` that appear in `keyAS, c, ra`.

The full specification is as follows:

```
spec Otway-Rees is
Theory
     types UName SName Name Key Nonce Masterkey
        Sessionkey .
     subtypes Masterkey Sessionkey < Key .
     subtypes SName UName < Name < Public .
     op n : Name Fresh -> Nonce .
     ops a b i : -> UName .
     op s : ->  SName .
     op mkey : Name Name -> Masterkey .
     op seskey : Name Name Nonce -> Sessionkey .
     op e  : Key Msg -> Msg . //encryption
     op d : Key Msg -> Msg . //decryption
     op _;_ : Msg Msg -> Msg [gather (e E)] .

     var K : Key .
     var Z : Msg .
     eq d(K, e(K, Z)) = Z .
     eq e(K, d(K, Z)) = Z .


Protocol
    vars ANAME BNAME : UName .
    vars ANAME1 BNAME1 : UName .
    var SNAME : SName .
    vars r r' r'' rM : Fresh .
    vars RA CB CS RB : Nonce .
    vars M1 MA : Msg .
    var KCA KCB : Sessionkey .
```

```
roles A B S .

Def(A) = c   := n(ANAME, rM),
          ra  := n(ANAME, r),
         skA  := mkey(ANAME, SNAME) .
Def(B) = rb  := n(BNAME, r'),
         skB  := mkey(BNAME, SNAME) .
Def(S) = skA := mkey(ANAME, SNAME),
         skB := mkey(BNAME, SNAME),
          kc := seskey(ANAME, BNAME,
                    n(SNAME, r'')) .


In(A) = ANAME, BNAME, SNAME .
In(B) = BNAME, SNAME .
In(S) = ANAME, BNAME, SNAME .

1 . A -> B :  c  ; ANAME  ; BNAME ;
            e(skA, ra ; c ; ANAME ; BNAME)
           |- CB ; ANAME1 ; BNAME ; M1 .


2 . B -> S :  CB ; ANAME1 ; BNAME ;
                M1 ;
                e(skB, rb ; CB ; ANAME1 ; BNAME)
           |- CS ; ANAME ; BNAME ;
                e(skA, RA ; CS ; ANAME ; BNAME) ;
                e(skB, RB ; CS ; ANAME ; BNAME) .


3 . S -> B :  CS ; e(skA, RA ; kc) ;
                e(skB, RB ; kc)
           |- CB ; MA ;
                e(skB, rb ; KCB) .


4 . B -> A :  CB ; MA
           |- c  ; e(skA, ra ; KCA) .

Out(A) = c, ra, KCA .
Out(B) = rb, KCB, MA, M1, CB .
Out(S) = kc, RA, RB, CS .
```

```
Intruder
    vars P : UName .
    vars K : Key .
    vars N M : Msg .

          => s, P, mkey(i, s) .
    M, N <=> M ; N .
    K, M  => d(K, M), e(K, M) .
    P     => mkey(P, i) .


Attacks
   0 .
         A executes protocol .
         Subst(A) = ANAME |-> a, BNAME |-> b,
            SNAME |-> s .
   1 .
         A executes protocol .
         Subst(A) = ANAME |-> a, BNAME |-> b,
            SNAME |-> s .
         Intruder learns KCA .
   2 .
         A executes protocol .
         Subst(A) = ANAME |-> a, BNAME |-> b,
            SNAME |-> s .
         without:
             B executes protocol .
             Subst(A) = ANAME |-> a, BNAME |-> b,
                SNAME |-> s .
             Subst(B) = M1 |-> e(skA, c ; na ; ANAME ;
                                 BNAME),
                        ANAME1 |-> a, BNAME |-> b,
                        SNAME |-> s .
ends
```

5.2 Translation

We have two methods for generating the Maude-NPA file from the above specification. In the first case, we can invoke the python script directly:

```
$ ./psl.py directory/to/Otway-Rees.psl
```

on the command line (assuming our file is named `Otway-Rees.psl`). This will generate the Maude-NPA file `directory/to/Otway-Rees.maude`, but will not

do anything else with it. Alternatively, we can run the bash script `psl.sh`:

```
$ ./psl.sh directory/to/Otway-Rees.psl
```

This will do the exact same thing as if we invoked the python file directly, except it will also load the `Otway-Rees.maude` file directly into Maude-NPA, allowing the user to immediately begin a Maude-NPA session.

The Maude-PSL generates the following Maude-NPA file. Note that the generated module has been reformatted for ease of readability. Furthermore terms of the form $X : SortName$ are inline declarations of variables. In Maude, inline declarations only have scope within the statement in which they appear. Inline variable declarations are not allowed in the Maude-PSL, due to subtleties in the relationship between a single statement in Maude-NPA, and a single statement in the Maude-PSL that make the semantics of inline declarations in the Maude-PSL unclear.

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
    protecting DEFINITION-PROTOCOL-RULES .
    sorts UName SName Name Key Nonce Masterkey
        Sessionkey .
    subsorts Masterkey Sessionkey < Key .
    subsorts SName UName < Name < Public .
    subsorts UName SName Name Key Nonce Masterkey
        Sessionkey < Msg .

    op n : Name  Fresh -> Nonce [ frozen ] .
    ops a  b  i : -> UName .
    op s : -> SName .
    op mkey : Name  Name -> Masterkey [ frozen ] .
    op seskey : Name  Name  Nonce ->
        Sessionkey [ frozen ] .
    op e : Key  Msg -> Msg [ frozen ] .
    op d : Key  Msg -> Msg [ frozen ] .
    op _;_ : Msg  Msg ->
        Msg [ gather  (  e  E  )  frozen ] .
    op _$;_ : Msg Msg ->
        Msg [ctor gather(e E) frozen].
endfm

fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
    protecting PROTOCOL-EXAMPLE-SYMBOLS .
    eq d ( K:Key , e ( K:Key , Z:Msg ) ) =
        Z:Msg [ variant ] .
    eq e ( K:Key , d ( K:Key , Z:Msg ) ) =
        Z:Msg [ variant ] .
```

```
endfm

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

    eq STRANDS-DOLEVYAO =
        :: nil ::
        [ nil | +(s), nil]  &
        :: nil ::
        [ nil | +(P:UName), nil]  &
        :: nil ::
        [ nil | +(mkey(i, s)), nil]  &
        :: nil ::
        [ nil | -(M:Msg), -(N:Msg), +(M:Msg ; N:Msg),
            nil]  &
        :: nil ::
        [ nil | -(P:UName), +(mkey(P:UName, i)), nil]
        &
        :: nil ::
        [ nil | -(K:Key), -(M:Msg),
              +(e(K:Key, M:Msg)), nil]  &
        :: nil ::
        [ nil | -(K:Key), -(M:Msg),
              +(d(K:Key, M:Msg)), nil]  &
        :: nil ::
        [ nil | -(M:Msg ; N:Msg), +(N:Msg), nil]  &
        :: nil ::
        [ nil | -(M:Msg ; N:Msg), +(M:Msg), nil]
        [nonexec].

    eq STRANDS-PROTOCOL =
        :: r':Fresh ::
        [ nil |
           -(CB:Nonce ; ANAME1:UName ; BNAME:UName ;
              M1:Msg),
           +(CB:Nonce ; ANAME1:UName ; BNAME:UName ;
              M1:Msg ;
              e(mkey(BNAME:UName, SNAME:SName),
                n(BNAME:UName, r':Fresh) ; CB:Nonce ;
                ANAME1:UName ; BNAME:UName)),
           -(CB:Nonce ; MA:Msg ;
```

```
            e(mkey(BNAME:UName, SNAME:SName),
              n(BNAME:UName, r':Fresh) ;
              KCB:Sessionkey)),
         +(CB:Nonce ; MA:Msg), nil]  &
      :: r'':Fresh ::
      [ nil |
         -(CS:Nonce ; ANAME:UName ; BNAME:UName ;
            e(mkey(ANAME:UName, SNAME:SName),
              RA:Nonce ; CS:Nonce ; ANAME:UName ;
              BNAME:UName) ;
            e(mkey(BNAME:UName, SNAME:SName),
              RB:Nonce ; CS:Nonce ; ANAME:UName ;
              BNAME:UName)),
         +(CS:Nonce ;
            e(mkey(ANAME:UName, SNAME:SName),
              RA:Nonce ;
              seskey(ANAME:UName, BNAME:UName,
                  n(SNAME:SName, r'':Fresh))) ;
            e(mkey(BNAME:UName, SNAME:SName),
              RB:Nonce ;
              seskey(ANAME:UName, BNAME:UName,
                  n(SNAME:SName, r'':Fresh)))), nil]
      &
      :: r:Fresh,rM:Fresh ::
      [ nil |
         +(n(ANAME:UName, rM:Fresh) ; ANAME:UName ;
            BNAME:UName ;
            e(mkey(ANAME:UName, SNAME:SName),
              n(ANAME:UName, r:Fresh) ;
              n(ANAME:UName, rM:Fresh) ;
              ANAME:UName ; BNAME:UName)),
         -(n(ANAME:UName, rM:Fresh) ;
            e(mkey(ANAME:UName, SNAME:SName),
               n(ANAME:UName, r:Fresh) ;
               KCA:Sessionkey)), nil] [nonexec].

var LIST : SMsgList-R .
var K : IntruderKnowledge .
var S : StrandSet .

eq ATTACK-STATE(0)=
      :: r:Fresh,rM:Fresh ::
      [ nil,
```

```
        +(n(a, rM:Fresh) ; a ; b ;
           e(mkey(a, s), n(a, r:Fresh) ;
               n(a, rM:Fresh) ; a ; b)),
        -(n(a, rM:Fresh) ;
           e(mkey(a, s), n(a, r:Fresh) ;
               KCA:Sessionkey)) | nil]
    || empty
    ||
    nil
    ||
    nil
    ||
    nil[nonexec].

eq ATTACK-STATE(1)=
    :: r:Fresh,rM:Fresh ::
    [ nil,
        +(n(a, rM:Fresh) ; a ; b ;
           e(mkey(a, s), n(a, r:Fresh) ;
               n(a, rM:Fresh) ; a ; b)),
        -(n(a, rM:Fresh) ;
           e(mkey(a, s), n(a, r:Fresh) ;
               KCA:Sessionkey)) | nil]
    ||
    KCA:Sessionkey inI
    ||
    nil
    ||
    nil
    ||
    nil[nonexec].

 eq ATTACK-STATE(2)=
    :: r:Fresh,rM:Fresh ::
    [ nil,
        +(n(a, rM:Fresh) ; a ; b ;
           e(mkey(a, s), n(a, r:Fresh) ;
               n(a, rM:Fresh) ; a ; b)),
        -(n(a, rM:Fresh) ;
           e(mkey(a, s), n(a, r:Fresh) ;
               KCA:Sessionkey)) | nil]
    || empty
    ||
```

```
        nil
        ||
        nil
        || never(
            (S &
            :: r':Fresh ::
            [ nil,
               -(CB:Nonce ; a ; b ;
                  e(mkey(a, s), n(a, rM:Fresh) ;
                      n(a, r:Fresh) ; a ; b)),
               +(CB:Nonce ; a ; b ;
                  e(mkey(a, s), n(a, rM:Fresh) ;
                  n(a, r:Fresh) ; a ; b) ;
                  e(mkey(b, s), n(b, r':Fresh) ;
                    CB:Nonce ; a ; b)),
               -(CB:Nonce ; MA:Msg ; e(mkey(b, s),
                  n(b, r':Fresh) ;
                  KCB:Sessionkey)),
               +(CB:Nonce ; MA:Msg) | nil] )
            || K)[nonexec].
endfm
select MAUDE-NPA .
```

From here, the specifier interacts with Maude-NPA in the exact same manner as if he/she had written a Maude-NPA file directly. See the Maude-NPA manual for details [13].

5.3 Maude-PSL vs. Maude-NPA

Now, we will highlight aspects of the two specifications that best illustrate the improvements the Maude-PSL makes over the strand-based language.

First, consider the specification of the protocol itself. Suppose we are debugging the specification, and we wish to make sure that we have correctly encoded the perspectives in the first step of the protocol. In the Maude-PSL, checking the perspectives is as simple as comparing the two messages in the first step:

```
1 . A -> B :  c  ; ANAME  ; BNAME ;
              e(skA, ra ; c ; ANAME ; BNAME)
           |- CB ; ANAME1 ; BNAME ;
              M1 .
```

Each subterm has been aligned along the concatenation operator, and the two perspectives are (vertically) as close as possible. Therefore, comparing the perspective is as simple as a few eye flicks, and a bit of thought.

Now, suppose we wish to compare perspectives using the Maude-NPA style strands. First, we need to isolate the strand for Alice, and the strand for Bob from the following strand set:

```
:: r':Fresh ::
[ nil |
   -(CB:Nonce ; ANAME1:UName ; BNAME:UName ;
      M1:Msg),
   +(CB:Nonce ; ANAME1:UName ; BNAME:UName ;
      M1:Msg ;
      e(mkey(BNAME:UName, SNAME:SName),
        n(BNAME:UName, r':Fresh) ; CB:Nonce ;
        ANAME1:UName ; BNAME:UName)),
   -(CB:Nonce ; MA:Msg ;
      e(mkey(BNAME:UName, SNAME:SName),
        n(BNAME:UName, r':Fresh) ;
        KCB:Sessionkey)),
   +(CB:Nonce ; MA:Msg), nil]  &
:: r'':Fresh ::
[ nil |
   -(CS:Nonce ; ANAME:UName ; BNAME:UName ;
      e(mkey(ANAME:UName, SNAME:SName),
        RA:Nonce ; CS:Nonce ; ANAME:UName ;
        BNAME:UName) ;
      e(mkey(BNAME:UName, SNAME:SName),
        RB:Nonce ; CS:Nonce ; ANAME:UName ;
        BNAME:UName)),
   +(CS:Nonce ;
      e(mkey(ANAME:UName, SNAME:SName),
        RA:Nonce ;
        seskey(ANAME:UName, BNAME:UName,
            n(SNAME:SName, r'':Fresh))) ;
      e(mkey(BNAME:UName, SNAME:SName),
        RB:Nonce ;
        seskey(ANAME:UName, BNAME:UName,
            n(SNAME:SName, r'':Fresh)))), nil]
&
:: r:Fresh,rM:Fresh ::
[ nil |
   +(n(ANAME:UName, rM:Fresh) ; ANAME:UName ;
      BNAME:UName ;
      e(mkey(ANAME:UName, SNAME:SName),
        n(ANAME:UName, r:Fresh) ;
```

```
        n(ANAME:UName, rM:Fresh) ;
        ANAME:UName ; BNAME:UName)),
    -(n(ANAME:UName, rM:Fresh) ;
      e(mkey(ANAME:UName, SNAME:SName),
        n(ANAME:UName, r:Fresh) ;
        KCA:Sessionkey)), nil] [nonexec].
```

If we're lucky, the person who wrote the specification was kind enough to leave comments indicating the strands for Alice, Bob, and the server. If we are not so lucky, then we need to determine which strand is which based on the pattern of messages. Fortunately, in Otway-Rees isolating Alice's strand is relatively easy: it is the strand whose first message is a sent message (i.e. the third strand in the set of strands above). Then, we need to take that first message, and compare it against the first messages in the other two strands in order to find the message that has the same basic structure as Alice's first message. For terms as complex as those used in Otway-Rees, this is not trivial. In this case, the first strand is Bob's strand. In other words, the two terms we care about are as far away from each other as they can be! The easiest approach to comparing the terms now is to copy and paste one of the terms next to the other, and compare them side-by-side.

Note that the Otway-Rees protocol can be written far more conveniently than it is above. For one thing, the strands are usually labeled in comments. For another, since the server and Alice never communicate directly, we can place Alice's and Bob's strand next to each other, and every principal will be next to every other principal with whom they communicate. However, even in that case, the first term in Alice's strand is still separated from the first term in Bob's strand by the entiretly of Alice's strand. Furthermore, there are other protocols for which the server does communicate with both Alice and Bob. In that case, there will always be two principals who communicate with each other, but whose strands are not next to each other.

Now, suppose we are approaching this specification for the first time, and we wish know to how the specification varies (if at all) from the protocol the specification is ostensibly describing. Or perhaps we are explaining the specification to a colleague. Either way, we need to be able to look at the specification, and reverse engineer the high-level description. Suppose we wish to construct the first step of the protocol. Then, in the Maude-PSL specification, we need only look at the following statement:

```
1 . A -> B :  c  ; ANAME  ; BNAME ;
              e(skA, ra ; c ; ANAME ; BNAME)
             |- CB ; ANAME1 ; BNAME ;
              M1 .
```

From here, it is a simple matter of converting the notation used in Maude-PSL to that used in the Alice-Bob notation. First, we replace `ANAME, BNAME,`

`ANAME1, BNAME1` with A, and B as appropriate (and obvious). Everything else simply differs from the high level specification by capitalization, giving us:

$$1.\ A \rightarrow B : C; A; B; e(SK_A, R_A; C; A; B)$$

Obviously, if we want to dig a bit deeper into precisely *how* Maude-PSL is representing the challenge (C), or the secret key of A, then we need to investigate the definitions. However, not only has this allowed us to very quickly reconstruct the first step, but it has also allowed us to more easily modularize our reverse engineering:

1. Determine the equivalent Alice-Bob step by converting the Maude-PSL notation into the standard Alice-Bob notation.
2. Expand the definitions as necessary, depending on how detailed one wishes to get.

For explaining the step in the protocol to a colleague, one may only need to convert the Maude-PSL notation into the Alice-Bob notation. For debugging the specification, or if the colleague asks a particularly subtle question, it may be necessary to expand definitions. However, that is a mechanical process. Most of the work involved in understanding the definitions lies in a knowledge of how the terms and equations are being modeled, not in parsing the specification itself.

Subsequent steps are slightly more complicated, because one also needs to instantiate perspectives, which may percolate through multiple steps. For example, suppose we now wish to determine the second step. Then, we look at the second step in the Maude-PSL specification:

```
2 . B -> S :  CB ; ANAME1 ; BNAME ;
                M1 ;
                e(skB, rb ; CB ; ANAME1 ; BNAME)
           |- CS ; ANAME  ; BNAME ;
                e(skA, RA ; CS ; ANAME ; BNAME) ;
                e(skB, RB ; CS ; ANAME ; BNAME) .
```

From the previous step (to say nothing of the variable name), we know that `CB` is Bob's perspective of Alice's challenge `c`. So that becomes C. Similarly, `M1` corresponds to the term `e(skA, ra ; c ; ANAME ; BNAME)`. A little bit of variable renaming gives us the step:

$$2.\ B \rightarrow S : C; A; B; e(SK_A, R_A; C; A; B); e(SK_B, R_B; C; A; B)$$

Similarly for the last two steps.

Determining the strand of a principal, P, is equally simple. Simply scan through the steps, and incrementally add messages. For each step in which P is sending a message, the message before the turnstile (`|-`) should show up in P's strand as a sent message. For each step in which P is receiving a message, the message after the turnstile should appear in the strand as a received message.

Now, suppose we wish to construct the same information from the Maude-NPA specification. First, we need to go through the same process discussed earlier for isolating the strands that we care about. Then, we need to extract the two terms we care about:

```
-(CB:Nonce ; ANAME:UName ; BNAME:UName ;
   M1:Msg)
 -----------
+(n(ANAME:UName, rM:Fresh) ; ANAME:UName ;
   BNAME:UName ;
   e(mkey(ANAME:UName, SNAME:SName),
     n(ANAME:UName, r:Fresh) ;
     n(ANAME:UName, rM:Fresh) ;
     ANAME:UName ; BNAME:UName))
```

Then we need to go through the same process as in Maude-PSL: compare the two perspectives and from there reconstruct the first high-level step. However, Maude-NPA does not have any sense of shorthand like the Maude-PSL. Therefore, we need to be able to recognize that `n(ANAME, rM)` is the challenge C, and `mkey(ANAME, SName)` is Alice's secret key. In other words, it becomes much harder with the Maude-NPA specification to control the level of detail at which you view the protocol. You always see all the details, all the time, whether you need to see them or not.

Now, suppose we wish to write a simple secrecy attack, in which the intruder manages to learn the session key that Alice and Bob are attempting to securely share, despite the fact that Alice successfully executes the protocol. Furthermore, suppose that Alice, Bob, the Server and the Intruder are all distinct entities. Then, we only need to write the following five lines in the Maude-PSL:

```
1 .
     A executes protocol .
     Subst(A) = ANAME |-> a, BNAME |-> b,
        SNAME |-> s .
     Intruder learns KCA .
```

Similarly, if we wish to understand what the attack is doing, we need to only read the above four lines. The only part of the attack that is at all subtle is the the substitution. However, much like the definitions, understanding all the subtleties may not be necessary, such as when explaining the attack to a

colleague. So again, the specifier is free to focus on the level of detail that is necessary for the job at hand.

However, in order to specify the exact same attack in Maude-NPA, we first need to find, and then copy and paste Alice's strand into an attack state. Then we need to manually instantiate the strand, add the intruder knowledge, and pepper the state with various `nil` constants to make the state fit the appropriate syntax. This gives us:

```
eq ATTACK-STATE(1)=
    :: r:Fresh,rM:Fresh ::
    [ nil,
       +(n(a, rM:Fresh) ; a ; b ;
          e(mkey(a, s), n(a, r:Fresh) ;
              n(a, rM:Fresh) ; a ; b)),
       -(n(a, rM:Fresh) ;
          e(mkey(a, s), n(a, r:Fresh) ;
              KCA:Sessionkey)) | nil]
    ||
    KCA:Sessionkey inI
    ||
    nil
    ||
    nil
    ||
    nil[nonexec].
```

First, this construction is very prone to error (i.e. it is very easy for the user to forget to replace a variable `ANAME` with the constant `a`). Second, not only is there additional information (in the form of the various `nil` keywords) that has no bearing whatsoever on the attack, but the attack state does not even make clear whose strand is being used, and how the strand is being instantiated. Therefore, if the specifier (or somebody else) approaches the specification later, and wishes to figure out what the attack is doing, he/she first has to determine which strand is being used, and then try to determine how it has been instantiated.

Finally, suppose we have edited the protocol, and now need to update the attacks to reflect the new protocol. In the case of the Maude-PSL, we only need to modify the attack state if the variables have been changed, which is unlikely if the protocol is only being tweaked. Even then, the only change that needs to be made is to the substitution. However, even the smallest change in the Maude-NPA specification requires the specifier to go through each attack that uses that strand (which may not be obvious), and make the exact same small change to every one of those attack states, while also preserving the intended instantiation. In short, the Maude-NPA specification, in addition to being harder to read, is

also much more brittle.

Chapter 6

Related Works and Conclusion

6.1 Related Works

ProVerif is one of the most popular protocol verification tools. ProVerif is based on the typed π-calculus, and as a result protocols are encoded in a variant of the applied π-calculus [4]. Much like the Maude-PSL, ProVerif allows the user to specify custom types and function symbols, and allows the user to define a custom equational theory using rewrite rules, but the theories allowed in ProVerif are considerably more restricted and cannot perform rewriting modulo AC. However, ProVerif also requires the user to explicitly formalize the types of checks that principals perform on the messages they receive (i.e. verifying digital signatures, extracting keys, etc.). These checks are implicitly formalized in the Maude-PSL through the use of perspectives. The notation of the applied π-calculus is quite distant from the standard Alice and Bob syntax typically used to informally describe protocols. Furthermore, since the applied π-calculus is a very general logic meant to model parallel processes, it is also relatively "low-level" for the purposes of verifying cryptographic protocols. As a result, it can be difficult to write and read ProVerif specifications, especially if the specifier lacks a deep understanding of applied π-calculus. On the other hand, a deep understanding of strand spaces is not necessary to read or write Maude-PSL specifications (though admittedly it is necessary to understand the attacks generated by Maude-NPA.

The High Level Protocol Specification Language (HLPSL) is another major protocol specification language, this one used by the AVISPA suite of tools [5]. HLPSL is based on Lamport's Temporal Logic of Actions [17]. HLPSL is similar to the original Maude-NPA strand-based input language, in that protocols are specified in terms of each role, rather than in terms of the messages sent and received. However, HLPSL allows the user more explicit control over the actions performed by each principal upon receipt of a message. For example, HLPSL allows the user to specify keyrings, and only request a key from a server if said key is not already in the principal's keyring. In other words the Maude-PSL requires the user to work at a much higher level of abstraction than HLPSL necessarily requires. For example, the Maude-PSL would abstract away keyrings by either assuming that the principals already know everyone else's public keys, or by

specifying the keys as input). However, HLSPL shares some of the same downsides as Maude-NPA: specifically the role-based specification makes it difficult to extract the universal sequence of message passes that serve as the intuitive formulation of protocols. The notation is also, like ProVerif, quite different from how protocol designers usually describe and think about protocols.

The Casper language, like the Maude-PSL, mixes a global specification of the protocol with role-specific information [19]. Both also allow for the specification of some algebraic properties (such as the commutativity of a given operator). In addition, the method of specifying the guarantees to be checked is very similar. For example, in Casper, the user claims that if the protocol is secure, then a given term will remain a secret after a successful run of the protocol. In the Maude-PSL the user claims that if a given term does not remain a secret after a successful run, then the protocol is not secure.

However, in Casper, in addition to specifying the sender and receiver viewpoint of a message, the specifier can also define tests that a principal performs upon receipt of a message. If these tests fail, then the user aborts the run. The Maude-PSL does not support such tests, though in many cases the information that would be gleaned is implicitly encoded in the points of view. Furthermore, Casper has separate syntax for each attack type, while in the Maude-PSL the attacks are defined using different combinations of more primitive statements. This provides the Maude-PSL greater flexibility in defining attacks, and reduces the syntax that needs to be learned. Also the Maude-PSL provides much greater flexibility in defining the term structure of the messages in the protocol.

The CAPSL language, developed at the Standford Research Institute, can be thought of as the spiritual ancestor of the Maude-PSL [21]. Some features, such as the Definitions, were inspired by similar features in CAPSL. Like the Maude-PSL, the CAPSL language attempts to surround an Alice and Bob specification with additional information that can then be translated into a given tool's potentially less intuitive input language. Furthermore, the basic structure of Maude-PSL specifications share many similarities to specifications in CAPSL. However, typically the only role-specific information provided in CAPSL are a list of assumptions. For example, in CAPSL, one may assume that Alice knows the name (Bob) of the principal he/she would like to communicate with. CAPSL specifications may also specify perspective, though it is not required like it is in the Maude-PSL. Furthermore, CAPSL was meant as a *lingua franca* between many different tools, much like HLPSL, while the Maude-PSL was designed explicitly for the Maude-NPA. As a result, CAPSL does not not have a natural means of handling the built-in axioms (i.e. commutativity and associativity) supported by Maude. Furthermore, CAPSL only supports user-defined prefix symbols, whereas the Maude-PSL supports more flexible mixfix operators, allowing the Maude-PSL specifications to more closely match the specifier's preferred syntax. Furthermore, the Maude-PSL and CAPSL differ in terms of how they specify the guarantees to be verified. CAPSL takes a positive approach:

the user specifies the goals directly. For example, `SECRET K` asserts that the key `K` must be kept secret. Meanwhile, the Maude-PSL takes a negative approach: the user specifies an attack pattern that, if successful, violates the goal. For example, an attack in which Alice executes the protocol, and the intruder learns the key `K`. Finally, the Maude-PSL allows the user to customize the capabilities of the Intruder in the *Intruder* section.

6.2 Future Work

As it stands, the Maude-PSL supports almost all the features of Maude-NPA. The only feature that the Maude-PSL currently lacks is support for protocol composition. Therefore, the next step is to implement protocol composition. After implementing protocol composition, the Maude-PSL will be able to express all types of protocol specifications that Maude-NPA currently supports. A means of importing sections (such as Theory and Intruder sections) would also be very valuable. For example, if a user wishes to use a theory of XOR, or a theory of asymmetric cryptography, then the user should be able to just import the requisite equational theories, rather than repeatedly rolling their own. In addition to saving the user time, this will also standardize specifications to some extent (since all specifications that import theories will have a core of shared syntax), and it will also drastically increase the chances that the user's equational theories meets all the assumptions made by Maude-NPA. Furthermore, if the user is importing one or more pre-defined theories, then the user should also be able to import the associated intruder capabilities. This would allow the user to focus on what is important: the protocol and attacks.

Maude-NPA makes a large number of assumptions about the equational theory provided [13]. Some of them, particularly termination, are non-trivial to verify, and therefore often are not. However, these properties are absolutely essential if we want the output of Maude-NPA to make any sense. Fortunately, tools have already been developed for checking if an equational theory written in Maude is terminating and confluent: the Maude Termination Tool and Church-Rosser Checker respectively [11, 12]. There is also a very straightforward algorithm to semi-decide the finite variant property (see [2]). Therefore, in order to ensure that our protocol proofs are not "built on sand" so to speak, the translation should include an option to perform these heavy checks. When this option is activated, then in addition to performing the standard syntactic and simple semantic checks, the Maude-PSL translator would also automatically invoke the Maude Termination Tool, Church Rosser-Checker, and an implementation of the algorithm for checking the finite variant property on the equational theory. If the checks succeed, then the specification is translated like normal. If the checks fail, then the output of the tool is displayed to the user, and the translation process aborted. Then, the user may make the necessary changes, and attempt to translate the theory again, just like any other translation error. Of

course, checking termination, and the finite variant property are undecidable in general (although confluence *is* decidable if termination has already been proven), so they should not be required in order to perform a successful translation. However, if all it takes to perform the checks is a flag on the command line, then the user is much more likely to perform the checks at least once.

Currently when Maude-NPA finds an attack, the attack is printed as a Maude-NPA state. Therefore, the attack as printed is displayed in a manner quite distant from the message passing notation typically used to express attacks. As a result, in order to understand the output of the tool, the user does still need to understand the structure of a Maude-NPA state, which includes a set of strands. Therefore, a means of automatically translating Maude-NPA states into a message passing notation similar to that used by the Maude-PSL would drastically improve the usability of the tool, because it would allow the user to work at the same level both for the input and the output. Alternatively (or in addition) there is a graphical user interface built by Santiago and Talcott to visualize attacks (and the state space in general) of a Maude-NPA specification [28]. Integrating the Maude-PSL with this interface would greatly enhance the usability of Maude-NPA.

6.3 Conclusion

The specification of cryptographic protocols is a careful balancing act. On one end, we have the classic Alice and Bob notation. This notation is intuitive, clear, and expressive. However it is also relatively informal, and sacrifices precision simplicity and readability. On the other hand, we have notation like that used by Maude-NPA: a very precise notation that provides all the information needed by a computer to perform automatic reasoning, but at the cost of simplicity, and readability. Furthermore, the original Maude-NPA input language has a slew of other weaknesses, chief among them a variety of syntactic artifacts that are required by Maude, but don't contribute to the meaning of the specification, and a substantial amount of code repetition.

In an effort to fix some of these issues, and to bring these two extremes closer together, we have in this document proposed a new input language for Maude-NPA: the Maude Protocol Specification Language (Maude-PSL). The Maude-PSL begins with the Alice and Bob notation, and extends it slightly to provide additional information. It also provides a means, through the *Theory* and *Intruder* sections to make explicit information usually ignored by the standard Alice and Bob notation: the algebraic properties of the operations used in the protocol, and the capabilities of the intruder. All of this information is specified in a concise notation that minimizes repetition and improves readability. Finally, attacks are specified in a combination of mathematical notation and natural language that makes it possible to describe attacks succinctly, clearly and in a manner that makes them resilient to changes in the protocol. Further-

more, the Maude-PSL performs a variety of syntactic and semantic checks that endeavor to minimize the chances of the user committing subtle errors that are all but impossible to detect. The Maude-PSL also opens the door for a deeper integration of Maude-NPA with other tools developed to aid Maude developers, such as the Maude Termination Tool, and the Church-Rosser Checker. We believe that the Maude-PSL will highlight subtleties in protocols before verification even begins, simplify the specification, communication, and verification of said protocols, and increase user confidence of the results of said verification.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1 – 70, 1999.

[2] S. E. Andrew Cholewa, José Meseguer. Variants of variants and the finite variant property. Technical Report 47117, Department of Computer Science, University of Illinois at Urbana-Champaign, 201 North Goodwin Avenue, Urbana, IL, 61801, 2014.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998.

[4] B. Blanchet, B. Smyth, and V. Cheval. *ProVerif 1.89: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*, 2014.

[5] Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A high-level protocol specification language for industrial security-sensitive protocols. In *Proceedings of Workshop on Specification and Automated Processing of Security Requirements*, 2004.

[6] Y. Chevalier and M. Rusinowitch. Compiling and securing cryptographic protocols. *Information Processing Letters*, 110(3):116 – 122, 2010.

[7] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.6)*, 2011.

[8] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, Sept. 2006.

[9] S. F. Doghmi, J. D. Guttman, and F. J. Thayer. Searching for shapes in cryptographic protocols. In *Proceedings of Thirteenth International Conference on tools and*, pages 523–538, 2007.

[10] D. Dolev and A. C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, Mar 1983.

[11] F. Durn, S. Lucas, and J. Meseguer. Mtt: The maude termination tool (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer Berlin Heidelberg, 2008.

[12] F. Durn and J. Meseguer. A church-rosser checker tool for conditional order-sorted equational maude specifications. In P. lveczky, editor, *Rewriting Logic and Its Applications*, volume 6381 of *Lecture Notes in Computer Science*, pages 69–85. Springer Berlin Heidelberg, 2010.

[13] S. Escobar, C. Meadows, and J. Meseguer. *Maude-NPA, Version 2.0*, 2011.

[14] S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. In P. lveczky, editor, *Rewriting Logic and Its Applications*, volume 6381 of *Lecture Notes in Computer Science*, pages 52–68. Springer Berlin Heidelberg, 2010.

[15] F. J. T. Fábrega. Strand spaces: Proving security protocols correct. *J. Comput. Secur.*, 7(2-3):191–230, Mar. 1999.

[16] J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In J. Díaz, editor, *Lecture Notes in Computer Science*, volume 154, pages 361–373. Springer, 1983.

[17] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[18] G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, Nov. 1995.

[19] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, pages 53–84, 1998.

[20] J. Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81:721–781, 2012.

[21] J. K. Millen and G. Denker. Capsl and mucapsl. *Journal of Telecommunications and Information Technology*, pages 16–27, 2002.

[22] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, Sept. 1992.

[23] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.

[24] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, 175 Fifth Avenue, New York, NY 10010, USA, 2002.

[25] D. J. Otway and O. Rees. Efficient and timely mutual authentication. *Operating Systems Review*, 21(1):8–10, 1987.

[26] G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981.

[27] S. Santiago, S. Escobar, C. Meadows, and J. Meseguer. A formal definition of protocol indistinguishability and its verification using maude-npa. In *Security and Trust Management - 10th International Workshop, STM 2014, Wroclaw, Poland, September 10-11, 2014. Proceedings*, pages 162–177, 2014.

[28] S. Santiago, C. Talcott, S. Escobar, C. Meadows, and J. Meseguer. A graphical user interface for maude-npa. *Electronic Notes in Theoretical Computer Science*, 258(1):3 – 20, 2009. Proceedings of the Ninth Spanish Conference on Programming and Languages (PROLE 2009).

[29] F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security properties correct. *Journal of Computer Security*, 7:191–230, 1999.

[30] L. Viganó. Automated security protocol analysis with the avispa tool. *Electronic Notes in Theoretical Computer Science*, 155(0):61 – 86, 2006. Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI) Mathematical Foundations of Programming Semantics XXI.

Appendix: Formal Syntax

Here we provide a BNF specification of the Maude-PSL syntax. Note that this BNF grammar is necessarily incomplete: the full grammar of a specification cannot be known until the user has defined all custom operators. We use the non-terminal ⟨*term*⟩ to represent terms built using the user-defined operators. Meanwhile, ⟨*id*⟩ refers to sequences of characters that do not contain whitespace. Finally, the nonterminal ⟨*nat*⟩ refers to natural numbers, and ⟨*string*⟩ refers to any string. Furthermore, spaces in the grammar represent actual spaces.

Note that although the ⟨*spec*⟩ rule generates the ⟨*theory*⟩, ⟨*protocol*⟩, ⟨*intruder*⟩, and ⟨*attack*⟩ sections in a specific order, they can in fact appear in any order. However they all must appear exactly once. Similarly with the core attack statements.

⟨*spec*⟩ ::= '`spec`' ⟨*id*⟩ '`is`' ⟨*theory*⟩ ⟨*protocol*⟩ ⟨*intruder*⟩ ⟨*attacks*⟩ '`ends`'

⟨*theory*⟩ ::= '`Theory`' ⟨*theoryStmts*⟩

⟨*theoryStmts*⟩ ::= ⟨*theoryStmt*⟩ '`.`'
| ⟨*theoryStmt*⟩ '`.`' ⟨*theoryStmts*⟩

⟨*theoryStmt*⟩ ::= ⟨*typeStmt*⟩
| ⟨*subtypeStmt*⟩
| ⟨*opStmt*⟩
| ⟨*varStmt*⟩
| ⟨*eqStmt*⟩

⟨*typeStmt*⟩ ::= '`type`' ⟨*ids*⟩
| '`types`' ⟨*ids*⟩
| '`sort`' ⟨*ids*⟩
| '`sorts`' ⟨*ids*⟩

⟨*ids*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ ⟨*ids*⟩

⟨*subtypeStmt*⟩ ::= '`subtype`' ⟨*ids*⟩ '`<`' ⟨*subtypes*⟩
| '`subtypes`' ⟨*ids*⟩ '`<`' ⟨*subtypes*⟩
| '`subsort`' ⟨*ids*⟩ '`<`' ⟨*subtypes*⟩
| '`subsorts`' ⟨*ids*⟩ '`<`' ⟨*subtypes*⟩

⟨*subtypes*⟩ ::= ⟨*ids*⟩
| ⟨*ids*⟩ '`<`' ⟨*subtypes*⟩

⟨*opStmt*⟩ ::= '`op`' ⟨*idsUscore*⟩ '`:`' ⟨*ids*⟩ '`->`' ⟨*id*⟩
| '`ops`' ⟨*idsUscore*⟩ '`:`' ⟨*ids*⟩ '`->`' ⟨*id*⟩
| '`op`' ⟨*idsUscore*⟩ '`:`' ⟨*ids*⟩ '`->`' ⟨*id*⟩ '`[`' ⟨*opattrs*⟩ '`]`'
| '`ops`' ⟨*idsUscore*⟩ '`:`' ⟨*ids*⟩ '`->`' ⟨*id*⟩ '`[`' ⟨*opattrs*⟩ '`]`'

⟨*idsUscore*⟩ ::= ⟨*id*⟩
| '`_`'
| ⟨*id*⟩ ⟨*idsUscore*⟩
| ⟨*id*⟩'`_`' ⟨*idsUscore*⟩
| '`_`'⟨*idsUscore*⟩

⟨*opattrs*⟩ ::= ⟨*opattr*⟩
| ⟨*opattr*⟩ ⟨*opattrs*⟩

⟨*opattr*⟩ ::= '`comm`'
| '`assoc`'
| '`id:`' ⟨*term*⟩
| '`iter`'
| '`gather(e E)`'
| '`gather(E e)`'

⟨*varStmt*⟩ ::= '`var`' ⟨*ids*⟩ '`:`' ⟨*id*⟩
| '`vars`' ⟨*ids*⟩ '`:`' ⟨*id*⟩

⟨*eqStmt*⟩ ::= '`eq`' ⟨*term*⟩ '`=`' ⟨*term*⟩
| '`eq`' ⟨*term*⟩ '`=`' ⟨*term*⟩ '`[`' ⟨*eqAttrs*⟩ '`]`'

⟨*eqAttrs*⟩ ::= ⟨*eqAttr*⟩
| ⟨*eqAttr*⟩ ⟨*eqAttrs*⟩

⟨*eqAttr*⟩ ::= '`variant`' | '`metadata`' ⟨*string*⟩ | '`homomorphism`'

⟨*protocol*⟩ ::= '`Protocol`' ⟨*protocolStmts*⟩

⟨*protocolStmts*⟩ ::= ⟨*protocolStmt*⟩ '`.`'
| ⟨*protocolStmt*⟩ '`.`' ⟨*protocolStmts*⟩

⟨*protocolStmt*⟩ ::= ⟨*varStmt*⟩
| ⟨*roleStmt*⟩
| ⟨*inStmt*⟩
| ⟨*defStmt*⟩
| ⟨*stepStmt*⟩
| ⟨*outStmt*⟩

⟨*roleStmt*⟩ ::= '`roles`' ⟨*ids*⟩

⟨*inStmt*⟩ ::= '`In(`'⟨*id*⟩'`) =`' ⟨*idList*⟩

⟨*idList*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩'`,`' ⟨*idList*⟩

⟨*defStmt*⟩ ::= '`Def(`'⟨*id*⟩'`) =`' ⟨*defPairs*⟩

⟨*defPairs*⟩ ::= ⟨*defPair*⟩
| ⟨*defPair*⟩, ⟨*defPairs*⟩

⟨*defPair*⟩ ::= ⟨*id*⟩ '`:=`' ⟨*term*⟩

⟨*stepStmt*⟩ ::= ⟨*int*⟩ '`.`' ⟨*id*⟩ '`->`' ⟨*id*⟩ '`:`' ⟨*term*⟩ '`|-`' ⟨*term*⟩

⟨*outStmt*⟩ ::= '`Out(`'⟨*id*⟩'`) =`' ⟨*termList*⟩

⟨*termList*⟩ ::= ⟨*term*⟩
| ⟨*term*⟩'`,`' ⟨*termList*⟩

⟨*intruder*⟩ ::= '`Intruder`' ⟨*intruderStmts*⟩

⟨*intruderStmts*⟩ ::= ⟨*intruderStmt*⟩ '`.`'
| ⟨*intruderStmt*⟩ '`.`' ⟨*intruderStmts*⟩

⟨*intruderStmt*⟩ ::= ⟨*varStmt*⟩
| ⟨*capabilitiesStmt*⟩

⟨*capabilitiesStmt*⟩ ::= '`=>`' ⟨*termList*⟩
| ⟨*termList*⟩ '`=>`' ⟨*termList*⟩
| ⟨*termList*⟩ '`<=>`' ⟨*termList*⟩

⟨*attack*⟩ ::= '`Attacks`' ⟨*attackPatterns*⟩

⟨*attackPatterns*⟩ ::= ⟨*attackPattern*⟩
| ⟨*attackPattern*⟩ ⟨*attackPatterns*⟩

⟨*attackPattern*⟩ ::= ⟨*int*⟩ '`.`' ⟨*coreAttack*⟩
| ⟨*int*⟩ '`.`' ⟨*coreAttack*⟩ ⟨*withoutBlocks*⟩
| ⟨*int*⟩ '`.`' ⟨*coreAttack*⟩ ⟨*reduction*⟩
| ⟨*int*⟩ '`.`' ⟨*coreAttack*⟩ ⟨*withoutBlocks*⟩ ⟨*reduction*⟩

⟨*coreAttack*⟩ ::= ⟨*executions*⟩ '`.`' ⟨*learns*⟩ '`.`' ⟨*constraints*⟩
| ⟨*executions*⟩
| ⟨*executions*⟩ ⟨*learns*⟩
| ⟨*executions*⟩ '`.`' ⟨*constraints*⟩

⟨*executions*⟩ ::= ⟨*execution*⟩ '`.`'
| ⟨*execution*⟩ '`.`' ⟨*executions*⟩

⟨*execution*⟩ ::= ⟨*executes*⟩ '`.`' ⟨*subst*⟩

⟨*executes*⟩ ::= ⟨*id*⟩ '`executes protocol`'
| ⟨*id*⟩ '`executes up to`' ⟨*nat*⟩

⟨*subst*⟩ ::= '`Subst(`' ⟨*id*⟩ '`) =`' ⟨*substMappings*⟩

⟨*substMappings*⟩ ::= ⟨*substMapping*⟩
| ⟨*substMapping*⟩'`,`' ⟨*substMappings*⟩

⟨*substMapping*⟩ ::= ⟨*id*⟩ '`|->`' ⟨*term*⟩

⟨*learns*⟩ ::= '`Intruder learns`' ⟨*termList*⟩

⟨*constraints*⟩ ::= '`With constraints`' ⟨*neqList*⟩

⟨*neqList*⟩ ::= ⟨*neq*⟩
| ⟨*neq*⟩, ⟨*neqList*⟩

⟨*neq*⟩ ::= ⟨*id*⟩ '`!=`' ⟨*term*⟩

⟨*withoutBlocks*⟩ ::= ⟨*withoutBlock*⟩
| ⟨*withoutBlock*⟩ ⟨*withoutBlock*⟩

⟨*withoutBlock*⟩ ::= '`without:`' ⟨*executions*⟩

⟨*reduction*⟩ ::= '`state-space-reduction:`' ⟨*avoidBlocks*⟩

⟨*avoidBlocks*⟩ ::= ⟨*avoidBlock*⟩ '`.`'
| ⟨*avoidBlock*⟩ ⟨*avoidBlocks*⟩

⟨*avoidBlock*⟩ ::= '`avoid:`' ⟨*coreAttack*⟩