# Document on PSL Composition

Andrew Cholewa
archolewa@gmail.com

May 14, 2015

## 1 PSL Composition

In the following document, we describe a proposed syntax for protocol compositions in the Maude Protocol Specification Language (Maude-PSL).

We'd like our syntax to accomplish the following:

- The syntax should be suggestive: The tokens used should have some history of being used for similar operations.
- The syntax should balance minimality and legibility. The simpler and more compact the syntax is, the less prone it is to typos and similar hard-to-see errors. However, if the syntax is too compact it may become difficult to keep track of everything going on in each statement.
- The syntax should be expressive. Anything that can be done using the Maude-NPA protocol composition syntax (and that makes sense) should also be doable in the Maude-PSL syntax.
- The syntax should be modular. If we're composing protocols $A$ and $B$, then the code for composing $A$ with $B$ should only change if we modify the *output* of $A$ or the *input* of $B$. Any other modifications to the two protocols should not affect the specification of the composed protocol.

We will demonstrate the proposed syntax using the running examples from the *Sequential Protocol Composition in Maude-NPA* paper.

We have the *NSL Protocol(NSL)*:

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

We have the *Distance Bounding Protocol(DB)*, which assumes that $A$ and $B$ already share some nonce $N$:

1. $A \rightarrow B : N_A$
2. $B \rightarrow A : N \oplus N_A$

Finally, we have the *Key Distribution Protocol (KD)*, which assumes that $A$ and $B$ have a shared key $h(N_A, N_B)$:

1. $A \rightarrow B : \{Sk_A\}_{h(N_A,N_B)}$
2. $B \rightarrow A : \{Sk_A, N'_B\}_{h(N_A,N_B)}$
3. $A \rightarrow B : \{N'_B\}_{h(N_A,N_B)}$

We will be combining NSL and DB in a 1-1 composition, and combining NSL and KD in a 1-* composition.

First, we have the 1-1 composition: *NSL*; *DB*. For the NSL protocol we have the following output for each role (the full PSL specifications of each protocol have been included in the zip containing this document):

```
Out(A) = A, B, n(A, r), NB, h(n(A, r), NB) .
Out(B) = A1, B, NA1, n(B, r), h(NA1, n(B, r)) .
```

For the DB protocol, we have the following input:

```
In(A) = A, B, N .
In(B) = A, B, N .
```

We specify the composed protocol *NSL*; *DB* (note that the name does not have to be `NSL;DB`) as follows:

```
spec NSL;DB is
    composing NSL ; DB .

    NSL.A ;1 DB.B : DB.A |-> NSL.B,  DB.B |-> NSL.A,
        DB.N |-> n(NSL.A, NSL.r) .
    NSL.B ;1 DB.A : DB.A |-> NSL.B, DB.B |-> NSL.A1, DB.N |-> NSL.NA1 .

    Attacks
    ...
ends
```

The statement `composing NSL ; DB` tells us that this is a composition specification with NSL is the parent, and DB the child(we will discuss composing more than two protocols later).

The next two statements specify the relationships between the roles in the parent (NSL) and the roles in the child (DB).

Consider the first statement:

```
NSL.A ;1 DB.B : DB.A |-> NSL.B, DB.B |-> NSL.A,
    DB.N |-> n(NSL.A, NSL.r).
```

The first part of the statement: `NSL.A ;1 DB.B`, tells us that the principal in the NSL-Alice role will be in the DB-Bob role, and that the two roles are forming a 1-1 composition. Now, consider the second part of the statement:

```
DB.A |-> NSL.B, DB.B |-> NSL.A, DB.N |-> n(NSL.A, NSL.r).
```

This specifies how the input of DB-Bob role should be instantiated with the output of NSL-Alice.

Observe that not all of the output of the NSL-Alice role needs to be used (we are not using `NB`, or `h(n(A, r), NB)`). This allows us to make the parent protocol composable with a variety of different children protocol, without affecting the specification of any one composition. In short, we restrict the output of the parent protocol to only those terms that we need for the composition.

However, we do require the specifier to define a mapping for every input parameter of the child. Furthermore, we check to make sure that the sorts of the Input-Output mapping make sense (i.e. for each pair $x : s \mapsto t : s'$, $s'$ is a subsort of $s$ in the combined theory).

Note that each variable is prepended by the name of the specification from which it originates. This is primarily to minimize confusion on the part of the specifier. Here is the original syntax that José and I decided upon:

```
spec NSL;DB is
    composing NSL ; DB .

    A ;1 B : A |-> B, B |-> A, N |-> n(A, r) .
    B ;1 A : A |-> B, B |-> A, N |-> NA1 .
```

However, this syntax obscures the fact that we actually have two sets of different variables: `A, B, r` from the NSL protocol, and `A, B, N` from the DB protocol. This can lead to confusion with terms like `n(A, r)`. Should this be written as `n(A, r)` (the output of NSL) or should it be written as `n(B, r)`(the output after being instantiated with the mapping `A |-> B`)? The first case lines up most directly with our goal: specify the relationship between the input and output in an order-agnostic fashion. However, if we view the mapping as a substitution, then technically the first option is incorrect. Instead of being: `A |-> B, B |-> A, N |-> n(A, r)`, it should be: `A |-> B, B |-> A, N |-> n(B, r)`. This would suggest that we should manually replace `n(A, r)` with `n(B, r)`, but not only is this confusing in its own right (which `B` are we talking about?), but it is also very error prone. Meanwhile, prepending each variable name with the specification from which it originates allows us to use the Input and Output unchanged without any confusion about which principal the nonce belongs to.

Here is an example of the one-to-many protocol composition $NSL; KD$.

Recall the output of NSL is the following:

```
Out(A) = A, B, n(A, r), NB, h(n(A, r), NB) .
Out(B) = A1, B, NA1, n(B, r), h(NA1, n(B, r)) .
```

while the input of KD is as follows:

```
In(A) = A, B, K .
In(B) = A, B, K .
```

This gives us the following composition specification:

```
spec NSL;KD is
    composing NSL ; KD .

    //NSL-Alice is the parent of KD-Alice.
    NSL.A ;* KD.A : KD.A |-> NSL.A, KD.B |-> NSL.B,
        KD.K |-> h(n(NSL.A, NSL.r), NSL.NB) .
    NSL.B ;* KD.B : KD.A |-> NSL.A, KD.B |-> NSL.B,
        KD.K |-> h(NSL.NB, n(NSL.B, NSL.r)) .

    //NSL-Alice is the parent of KD-Bob.
    NSL.A ;* KD.B : KD.A |-> NSL.B, KD.B |-> NSL.A,
        KD.K |-> h(n(NSL.A, NSL.r), NSL.NB) .
    NSL.B ;* KD.A : KD.A |-> NSL.B, KD.B |-> NSL.A,
        KD.K |-> h(NSL.NA1, n(NSL.B, NSL.r)) .
```

Note that the syntax is exactly the same, except rather than using `;1` for the role composition, we use `;*`, analogous to the $1-1$ vs. $1-*$ notation.

When composing more than two protocols together, we break the composition of any two protocols into different sections. At a high level, the structure of a composition specification would look like the following:

```
spec P1;P2;P3;P4 is
    composing P1 ; P2 ; P3 ; P4 .

    comp P1 ; P2
        A ;1 B : ...
        ...

    comp P2 ; P3
        ...

    comp P3 ; P4
        ...
```

Furthermore, composition specifications may also have an `Attacks` section, just like a normal PSL specification. The equational theory is taken to be

the union of the theories in the specifications being composed (using Maude's module operations behind the scenes), similarly with the intruder capabilities. The current thought is that attacks will not be imported.

However, attacks are going to be a little bit tricky, because they will very easily break modularity, since specifying an attack may require the specifier to instantiate a variable that is not part of the Input and Output. While it may be possible to define some attacks in a modular fashion (i.e. does composing NSL with KD make it possible to break NSL's guarantees of secrecy?), other attacks that directly invoke the protocol composition will be trickier. For example, I'm not sure what the best way is of handling the attack in section 8.1 of the *Sequential Protocol Composition* paper:

```
eq ATTACK-STATE(0) =
    :: r ::
    [nil , +(pk(C, n(a, r) ; a)) |
           -(pk(a, n(a, r) ; NC ; C)),
           +(pk(C, NC)),
           {NSL-init -> DB-resp ;; 1-1 ;; a ;; C ; n(a, r)}, nil] &
    :: r' ::
    [nil,  {NSL-resp -> DB-init ;; 1-1 ;; D ; b ; n(a, r)},
     +(n(b, r')),
     -(n(a, r) * n(b, r')) | nil]
    || (a != D), (C != b)
    || nil
    || nil
    || nil
    [nonexec] .
```

Finally, in the same folder as this document, you can also find the full PSL-specifications of the NSL, DB, KD, and the two compositions. I've also tried to manually translate the NSL;DB specification into a Maude-module, to give some sense of what the generated Maude-NPA code would look like. I have not included the translations of NSL, DB, or KD, because the details of the individual transformations are not important. What matters right now is how they are combined.

## 2 Current Status

The rewriting semantics for composition have been implemented for the simple case, and partially tested, under the module COMPOSITION, and COMP-TRANSLATION-TO-MAUDE-NPA in psl.maude. However a corner-case is not handled properly, and the semantics for attacks needs to be implemented. One may be able to directly invoke the attack semantics from the standalone translation. That depends on what changes (if any) need to be made to the attack syntax for attacks to make sense in terms of protocol composition. More details can be found in the comments for the above modules.

The python level parser must also be implemented. The job of the Python parser is to take the PSL specification and turn it into a term for use by Maude. In particular, the Python code will need to locate the PSL-specifications of the two sub-protocols, obtain the intermediate term for Maude to rewrite (obtained by calling `gen_intermediate` see the `psl.py` file), and wrap them in a `$translate` operator, along with an arbitrary (but unique number). See `composition_examples/debugging_maude/comp_nsl_db.maude` for an example of what the Python code should generate.