** Yahalom Protocol Tamarin Script (giving example)

The following are the descriptions of the Yahalom protocol, the Needham-Schroeder symmetric-key authentication protocol written in natural language, and the script for the Needham-Schroeder symmetric-key authentication protocol for Tamarin Prover . Write a script for the Tamarin Prover for the Yahalom protocol, using the Needham-Schroeder symmetric-key authentication protocol as an example of generating script from the natural language description.''

【Yahalom protocol in natural language】
Yahalom is an authentication and secure key-sharing protocol designed for use on an insecure network such as the Internet. Yahalom uses a trusted arbitrator to distribute a shared key between two people. This protocol can be considered as an improved version of Wide Mouth Frog protocol (with additional protection against man-in-the-middle attack), but less secure than the Needham–Schroeder protocol.

Protocol description
If Alice (A) initiates the communication to Bob (B) with S is a server trusted by both parties, the protocol can be specified as follows using security protocol notation:

* A and B are identities of Alice and Bob respectively
* Kas is a symmetric key known only to A and S
* Kbs is a symmetric key known only to B and S
* Na and Nb are are nonces generated by A and B respectively
* Kab is a symmetric, generated key, which will be the session key of the session between A and B

A->B: A, Na
  Alice sends a message to Bob requesting communication.
B->S: {A, Na, Nb}Kbs
  Bob sends a message to the Server encrypted under Kbs
S->A: {B, Kab, Na, Nb}Kas, {A, Kab}Kbs
  The Server sends to Alice a message containing the generated session key Kab and a message to be forwarded to Bob.
A->B: {A, Kab}Kbs, {Nb}Kab
  Alice forwards the message to Bob and verifies Na has not changed. Bob will verify Nb has not changed when he receives the message.

【Needham Schroeder protocol in natural language】
The Needham–Schroeder protocol is one of the two key transport protocols intended for use over an insecure network, both proposed by Roger Needham and Michael Schroeder.[1] These are:

The Needham–Schroeder Symmetric Key Protocol, based on a symmetric encryption algorithm. It forms the basis for the Kerberos protocol. This protocol aims to establish a session key between two parties on a network, typically to protect further communication.

Here, Alice (A) initiates the communication to Bob B. S is a server trusted by both parties. In the communication:
* A and B are identities of Alice and Bob respectively
* Kas is a symmetric key known only to A and S
* Kbs is a symmetric key known only to B and S
* Na and Nb are nonces generated by A and B respectively
* Kab is a symmetric, generated key, which will be the session key of the session between A and B

The protocol can be specified as follows in security protocol notation:
A->S: A, B, Na
  Alice sends a message to the server identifying herself and Bob, telling the server she wants to communicate with Bob.
S->A: {Na, Kab, B, {Kab, A}Kbs}Kas
  The server generates Kab and sends back to Alice a copy encrypted under Kbs for Alice to forward to Bob and also a copy for Alice. Since Alice may be requesting keys for several different people, the nonce assures Alice that the message is fresh and that the server is replying to that particular message and the inclusion of Bob's name tells Alice who she is to share this key with.
A->B: {Kab, A}Kbs
  Alice forwards the key to Bob who can decrypt it with the key he shares with the server, thus authenticating the data.
B->A: {Nb}Kab
  Bob sends Alice a nonce encrypted under Kab to show that he has the key.
A->B: {Nb-1}Kab
  Alice performs a simple operation on the nonce, re-encrypts it and sends it back verifying that she is still alive and that she holds the key.

【Needham Schroeder protocol Tamarin Prover script】
```
/*
 Example for the Tamarin Prover
 ==============================

 Authors:     Jannik Dreier
 Date:        March 2020
 Time:

 Description from SPORE:
 (http://www.lsv.fr/Software/spore/nssk.html)

 Needham Schroeder Symmetric Key

 A, B, S :        principal
 Na, Nb :         nonce
 Kas, Kbs, Kab :  key
 dec :            nonce -> nonce

 1.   A  ->  S  :   A, B, Na
 2.   S  ->  A  :   {Na, B, Kab, {Kab, A}Kbs}Kas
 3.   A  ->  B  :   {Kab,A}Kbs
 4.   B  ->  A  :   {Nb}Kab
 5.   A  ->  B  :   {dec(Nb)}Kab
```

```
  This protocol establishes the fresh shared symmetric key Kab.

  Messages 1-3 perform the distribution of the fresh shared symmetric key Kab
  and messages 4-5 are for mutual authentification of A and B.

  The operator dec is decrementation.

  The protocol must guaranty the secrecy of Kab: in every session,
  the value of Kab must be known only by the participants playing the roles of A, B and S in that
session.

  If the participant playing B accepts the last message 5,
  then Kab has been sent in message 3. by A (whose identity is included in the cipher of message
3).

  Authentication attack by Denning and Sacco.
  Assume that I has recorded the session i and that Kab is compromised.
  After the session ii, B is convinced that he shares the secret key Kab only with A.
  i.1.   A  ->  S    :      A, B, Na
  i.2.   S  ->  A    :      {Na, B, Kab, {Kab, A}Kbs }Kas
  i.3.   A  ->  I(B)  :      {Kab,A}Kbs
               assume that Kab is compromised
  ii.3.   I(A)  ->  B  :    {Kab,A}Kbs
  ii.4.   B  ->  I(A)  :    {Nb}Kab
  ii.5.   I(A)  ->  B  :    {dec(Nb)}Kab

  We use tags and only one server.

*/

theory Nssk
begin

builtins: symmetric-encryption

functions: dec/1, inc/1

equations: inc(dec(x))=x

// ====================
// == General rules ===
// ====================

// A and B already share a long-term symmetric key with S

rule Init:
  [ Fr(~kxs)]
  --[ KeyGen($X) ]->
  [ !LongtermKey(~kxs,$X) ]
```

```
rule Reveal_Longterm_Key:
   [ !LongtermKey(~sk,$A)]
  --[ Reveal($A)]->
   [ Out(~sk) ]

// ====================
// == Protocol rules ==
// ====================

// 1.   A  ->  S  :   A, B, Na
rule A_to_S:
  [
   Fr(~na),
  !LongtermKey(~kas,$A)
  ]
  --[ ]->
  [
   Out(<'1', $A, $B, ~na>)
   , StateA1($A, $B, ~kas, ~na)
  ]

// 2.   S  ->  A  :   {Na, B, Kab, {Kab, A}Kbs}Kas
rule S_to_A:
  let msg = senc(<'2', na, y, ~kab, senc(<'3',~kab,x>,~kbs)>,~kas) in
  [
     In(<'1', x, y, na>)
   , !LongtermKey(~kas,x)
   , !LongtermKey(~kbs,y)
   , Fr(~kab)
  ]
  --[
   ]->
  [
     Out(msg)
   , !ShorttermKey(~kab,x,y)
  ]

// 3.   A  ->  B  :   {Kab,A}Kbs
rule A_to_B_1:
  let msg = senc(<'2',~na, $B, kab, mb>,~kas) in
  [
     StateA1($A, $B, ~kas, ~na)
   , !LongtermKey(~kas,$A)
   , In(msg)
  ]
  --[
   ]->
  [
     Out(mb)
   , StateA2($A, $B, ~kas, ~na, kab, mb)
  ]
```

```
// 4.   B  ->  A  :   {Nb}Kab
rule B_from_A1:
 let msg1 = senc(<'3',kab,x>,~kbs)
     msg2 = senc(<'4',~nb>, kab)
 in
 [
    !LongtermKey(~kbs,$B)
   , In(msg1)
   , Fr(~nb)
 ]
 --[
   ]->
 [
    Out(msg2)
   , StateB1(x, $B, ~kbs, ~nb, kab)
 ]

// 5.   A  ->  B  :   {dec(Nb)}Kab  Send
rule A_to_B_2:
 [
    StateA2($A, $B, ~kas, ~na, kab, mb)
   , !LongtermKey(~kas,$A)
   , In(senc(<'4',nb>,kab))
 ]
 --[
   ]->
 [
   Out(senc(<'5',dec(nb)>,kab))
 ]

// 5.   A  ->  B  :   {dec(Nb)}Kab  Receive
rule B_from_A2:
 [
    StateB1(x, $B, ~kbs, ~nb, kab)
   , !LongtermKey(~kbs,$B)
   , In(senc(<'5',dec(~nb)>,kab))
 ]
 --[
   ]->
 []


end
```

Here's the Tamarin Prover script for the Yahalom protocol based on the given description and structured similarly to the Needham-Schroeder symmetric-key protocol script:

```prolog
theory Yahalom
begin

builtins: symmetric-encryption

functions: dec/1, inc/1

equations: inc(dec(x))=x

// ====================
// == General rules ===
// ====================

// A and B already share a long-term symmetric key with S

rule Init:
  [ Fr(~kxs)]
  --[ KeyGen($X) ]->
  [ !LongtermKey(~kxs,$X) ]

rule Reveal_Longterm_Key:
    [ !LongtermKey(~sk,$A)]
  --[ Reveal($A)]->
    [ Out(~sk) ]

// ====================
// == Protocol rules ==
// ====================

// 1. A -> B : A, Na
rule A_to_B:
  [ Fr(~na) ]
  --[ ]->
  [ Out(<$A, ~na>), StateA1($A, $B, ~na) ]

// 2. B -> S : {A, Na, Nb}Kbs
rule B_to_S:
  let msg = senc(<$A, ~na, ~nb>, ~kbs) in
  [
      StateA1($A, $B, ~na)
    , Fr(~nb)
    , !LongtermKey(~kbs, $B)
  ]
  --[ ]->
  [ Out(msg), StateB1($B, ~na, ~nb, ~kbs) ]

// 3. S -> A : {B, Kab, Na, Nb}Kas, {A, Kab}Kbs
rule S_to_A:
  let msg1 = senc(<$B, ~kab, ~na, ~nb>, ~kas)
      msg2 = senc(<$A, ~kab>, ~kbs) in
  [
      In(senc(<$A, ~na, ~nb>, ~kbs))
    , Fr(~kab)
    , !LongtermKey(~kas, $A)
    , !LongtermKey(~kbs, $B)
  ]
  --[ ]->
  [ Out(msg1), Out(msg2), !ShorttermKey(~kab, $A, $B), StateS1($A, $B, ~na, ~nb, ~kab) ]

// 4. A -> B : {A, Kab}Kbs, {Nb}Kab
rule A_to_B_final:
  let msg1 = senc(<$A, ~kab>, ~kbs)
      msg2 = senc(~nb, ~kab) in
  [
      StateS1($A, $B, ~na, ~nb, ~kab)
    , In(msg1)
    , In(msg2)
  ]
  --[ ]->
  [ Out(msg1), Out(msg2), StateA2($A, $B, ~kab, ~nb) ]
```

```
// 5. Verification by B that Nb matches
rule B_Verify:
  [
      StateA2($A, $B, ~kab, ~nb)
    , In(senc(~nb, ~kab))
  ]
  --[ ]->
  []

end
```

Explanation of the Protocol Script:

1. **Init and Reveal Rules**: These rules generate and reveal long-term keys, setting up the initial shared keys for participants and the server.
2. **A_to_B**: Alice (A) sends her identity and nonce (`Na`) to Bob (B).
3. **B_to_S**: Bob encrypts Alice's identity, nonce `Na`, and his own nonce `Nb` under his long-term key `Kbs` and sends it to the server.
4. **S_to_A**: The server generates a session key `Kab`, then sends back two encrypted messages:
   - One for Alice, including `B`, `Kab`, `Na`, and `Nb` encrypted under `Kas`.
   - Another for Bob, containing `A` and `Kab` encrypted under `Kbs`.
5. **A_to_B_final**: Alice forwards the server's message to Bob and sends `Nb` encrypted under `Kab`, ensuring freshness.
6. **B_Verify**: Bob verifies that `Nb` matches, confirming the integrity of the key exchange.

This script should provide the structure needed for Tamarin Prover to simulate the Yahalom protocol based on the descriptions and the Needham-Schroeder example.

In a rule where a long term key is used, please insert a premise with LongtermKey predicate declaring the owner of the key.

Certainly! I've updated the script to include a `LongtermKey` predicate in the premises of rules where a long-term key is used to declare the owner of the key.

```prolog
theory Yahalom
begin

builtins: symmetric-encryption

functions: dec/1, inc/1

equations: inc(dec(x))=x

// ====================
// == General rules ===
// ====================

// A and B already share a long-term symmetric key with S

rule Init:
  [ Fr(~kxs)]
```

```
  --[ KeyGen($X) ]->
  [ !LongtermKey(~kxs, $X) ]

rule Reveal_Longterm_Key:
    [ !LongtermKey(~sk, $A) ]
  --[ Reveal($A) ]->
    [ Out(~sk) ]

// ====================
// == Protocol rules ==
// ====================

// 1. A -> B : A, Na
rule A_to_B:
  [ Fr(~na) ]
  --[ ]->
  [ Out(<$A, ~na>), StateA1($A, $B, ~na) ]

// 2. B -> S : {A, Na, Nb}Kbs
rule B_to_S:
  let msg = senc(<$A, ~na, ~nb>, ~kbs) in
  [
      StateA1($A, $B, ~na)
    , Fr(~nb)
    , !LongtermKey(~kbs, $B)  // Declaring the owner of the long-term key
  ]
  --[ ]->
  [ Out(msg), StateB1($B, ~na, ~nb, ~kbs) ]

// 3. S -> A : {B, Kab, Na, Nb}Kas, {A, Kab}Kbs
rule S_to_A:
  let msg1 = senc(<$B, ~kab, ~na, ~nb>, ~kas)
      msg2 = senc(<$A, ~kab>, ~kbs) in
  [
      In(senc(<$A, ~na, ~nb>, ~kbs))
    , Fr(~kab)
    , !LongtermKey(~kas, $A)   // Declaring the owner of the long-term key for A
    , !LongtermKey(~kbs, $B)   // Declaring the owner of the long-term key for B
  ]
  --[ ]->
  [ Out(msg1), Out(msg2), !ShorttermKey(~kab, $A, $B), StateS1($A, $B, ~na, ~nb, ~kab) ]

// 4. A -> B : {A, Kab}Kbs, {Nb}Kab
rule A_to_B_final:
  let msg1 = senc(<$A, ~kab>, ~kbs)
      msg2 = senc(~nb, ~kab) in
  [
      StateS1($A, $B, ~na, ~nb, ~kab)
    , !LongtermKey(~kbs, $B)   // Declaring the owner of the long-term key for B
    , In(msg1)
    , In(msg2)
  ]
  --[ ]->
  [ Out(msg1), Out(msg2), StateA2($A, $B, ~kab, ~nb) ]

// 5. Verification by B that Nb matches
rule B_Verify:
  [
      StateA2($A, $B, ~kab, ~nb)
    , In(senc(~nb, ~kab))
  ]
  --[ ]->
  []

end
```

Explanation of Changes:

1. **B_to_S Rule**: Added `` `!LongtermKey(~kbs, $B)` `` to declare that the key `` `Kbs` `` is owned by `` `B` ``.

2. **S_to_A Rule**: Added `` `!LongtermKey(~kas, $A)` `` and `` `!LongtermKey(~kbs, $B)` `` to declare the ownership of `` `Kas` `` by `` `A` `` and `` `Kbs` `` by `` `B` ``.
3. **A_to_B_final Rule**: Added `` `!LongtermKey(~kbs, $B)` `` to declare that the key `` `Kbs` `` is owned by `` `B` ``.

This update ensures that the ownership of each long-term key is explicitly declared in the premises of the rules where the key is used.

In a rule where two messaegs are sent, let the tuple of the messages to be sent in a single Out predicate.

Here's the updated script with a single `` `Out` `` predicate containing a tuple of messages for the rules where two messages are sent.

```prolog
theory Yahalom
begin

builtins: symmetric-encryption

functions: dec/1, inc/1

equations: inc(dec(x))=x

// ====================
// == General rules ===
// ====================

// A and B already share a long-term symmetric key with S

rule Init:
  [ Fr(~kxs)]
  --[ KeyGen($X) ]->
  [ !LongtermKey(~kxs, $X) ]

rule Reveal_Longterm_Key:
    [ !LongtermKey(~sk, $A) ]
  --[ Reveal($A) ]->
    [ Out(~sk) ]

// ====================
// == Protocol rules ==
// ====================

// 1. A -> B : A, Na
rule A_to_B:
  [ Fr(~na) ]
  --[ ]->
  [ Out(<$A, ~na>), StateA1($A, $B, ~na) ]

// 2. B -> S : {A, Na, Nb}Kbs
rule B_to_S:
  let msg = senc(<$A, ~na, ~nb>, ~kbs) in
  [
      StateA1($A, $B, ~na)
    , Fr(~nb)
    , !LongtermKey(~kbs, $B)  // Declaring the owner of the long-term key
  ]
  --[ ]->
```

```
    [ Out(msg), StateB1($B, ~na, ~nb, ~kbs) ]

  // 3. S -> A : {B, Kab, Na, Nb}Kas, {A, Kab}Kbs
  rule S_to_A:
    let msg1 = senc(<$B, ~kab, ~na, ~nb>, ~kas)
        msg2 = senc(<$A, ~kab>, ~kbs) in
    [
        In(senc(<$A, ~na, ~nb>, ~kbs))
      , Fr(~kab)
      , !LongtermKey(~kas, $A)   // Declaring the owner of the long-term key for A
      , !LongtermKey(~kbs, $B)   // Declaring the owner of the long-term key for B
    ]
    --[ ]->
    [ Out(<msg1, msg2>), !ShorttermKey(~kab, $A, $B), StateS1($A, $B, ~na, ~nb, ~kab) ]

  // 4. A -> B : {A, Kab}Kbs, {Nb}Kab
  rule A_to_B_final:
    let msg1 = senc(<$A, ~kab>, ~kbs)
        msg2 = senc(~nb, ~kab) in
    [
        StateS1($A, $B, ~na, ~nb, ~kab)
      , !LongtermKey(~kbs, $B)   // Declaring the owner of the long-term key for B
      , In(msg1)
      , In(msg2)
    ]
    --[ ]->
    [ Out(<msg1, msg2>), StateA2($A, $B, ~kab, ~nb) ]

  // 5. Verification by B that Nb matches
  rule B_Verify:
    [
        StateA2($A, $B, ~kab, ~nb)
      , In(senc(~nb, ~kab))
    ]
    --[ ]->
    []

  end
```

Explanation of Changes:

1. **S_to_A Rule**: Combined `msg1` and `msg2` into a tuple `<msg1, msg2>` and included it in a single `Out` predicate.
2. **A_to_B_final Rule**: Similarly, combined `msg1` and `msg2` into a tuple `<msg1, msg2>` and included it in a single `Out` predicate.

This approach ensures that when multiple messages are sent, they are combined into a single `Out` predicate as specified.

Show the pairs of rules such that each pair represents consecutive sending of messages from a participant.

Here are the pairs of rules in the Yahalom protocol script that represent consecutive sendings of messages from the same participant:

1. **Participant A (Alice)**:

- **Pair 1**: `A_to_B` → `A_to_B_final`
  - **Explanation**: The rule `A_to_B` represents Alice's initial message to Bob, and `A_to_B_final` represents her final message to Bob. These two rules are consecutive messages sent by Alice to Bob within the protocol sequence.

2. **Participant S (Server)**:

- **Pair 2**: `S_to_A`
  - **Explanation**: This rule involves the server sending two messages in one step. Since this is a single rule with two messages sent in a tuple (combined in a single `Out` predicate), there is no further pair for consecutive messages from the server.

3. **Participant B (Bob)**:

- **Pair 3**: `B_to_S` → `B_Verify`
  - **Explanation**: The rule `B_to_S` represents Bob's message to the server, while `B_Verify` represents Bob's verification of the nonce `Nb`. These are consecutive steps, but they are not consecutive sendings to the same participant, as `B_Verify` is a reception rather than a sending action.

In summary, the main consecutive pair of rules with actual consecutive sendings of messages are:

- **Alice**: `A_to_B` → `A_to_B_final`
- **Server**: A single rule `S_to_A` sends two messages

Ensure that if the first rule of such a pair contains a predicate with its name beginning with "State", then the premise of the second rule in the pair contains the same predicate and no other "State" predicate. Remove "State" predicate from the premise of a rule if the rule has no subsequent sending rule from the same participant.

Here is the updated script with adjustments to the `State` predicates to ensure that:

1. In each pair of consecutive message-sending rules, if the first rule contains a `State` predicate in its conclusion, then the second rule includes that same `State` predicate in its premise, and no other `State` predicate.
2. `State` predicates are removed from the premises of rules where there is no subsequent sending rule from the same participant.

```prolog
theory Yahalom
begin

builtins: symmetric-encryption

functions: dec/1, inc/1

equations: inc(dec(x))=x

// ====================
```

```
// == General rules ===
// ====================

// A and B already share a long-term symmetric key with S

rule Init:
  [ Fr(~kxs) ]
  --[ KeyGen($X) ]->
  [ !LongtermKey(~kxs, $X) ]

rule Reveal_Longterm_Key:
    [ !LongtermKey(~sk, $A) ]
  --[ Reveal($A) ]->
    [ Out(~sk) ]

// ====================
// == Protocol rules ==
// ====================

// 1. A -> B : A, Na
rule A_to_B:
  [ Fr(~na) ]
  --[ ]->
  [ Out(<$A, ~na>), StateA1($A, $B, ~na) ]

// 2. B -> S : {A, Na, Nb}Kbs
rule B_to_S:
  let msg = senc(<$A, ~na, ~nb>, ~kbs) in
  [
      Fr(~nb)
    , !LongtermKey(~kbs, $B)  // Declaring the owner of the long-term key
  ]
  --[ ]->
  [ Out(msg), StateB1($B, ~na, ~nb, ~kbs) ]

// 3. S -> A : {B, Kab, Na, Nb}Kas, {A, Kab}Kbs
rule S_to_A:
  let msg1 = senc(<$B, ~kab, ~na, ~nb>, ~kas)
      msg2 = senc(<$A, ~kab>, ~kbs) in
  [
      In(senc(<$A, ~na, ~nb>, ~kbs))
    , Fr(~kab)
    , !LongtermKey(~kas, $A)   // Declaring the owner of the long-term key for A
    , !LongtermKey(~kbs, $B)   // Declaring the owner of the long-term key for B
  ]
  --[ ]->
  [ Out(<msg1, msg2>), !ShorttermKey(~kab, $A, $B), StateS1($A, $B, ~na, ~nb, ~kab) ]

// 4. A -> B : {A, Kab}Kbs, {Nb}Kab
rule A_to_B_final:
  let msg1 = senc(<$A, ~kab>, ~kbs)
      msg2 = senc(~nb, ~kab) in
  [
      StateA1($A, $B, ~na)    // Including the "StateA1" predicate from A_to_B
    , !LongtermKey(~kbs, $B)  // Declaring the owner of the long-term key for B
    , In(msg1)
    , In(msg2)
  ]
  --[ ]->
  [ Out(<msg1, msg2>), StateA2($A, $B, ~kab, ~nb) ]

// 5. Verification by B that Nb matches
rule B_Verify:
  [
      In(senc(~nb, ~kab))
  ]
  --[ ]->
  []

end
```

Explanation of Changes:

1. **A_to_B → A_to_B_final**:
   - `` `A_to_B` `` concludes with `` `StateA1($A, $B, ~na)` ``, which is included as the only `` `State` `` predicate in the premise of `` `A_to_B_final` ``.
2. **B_to_S**:
   - Since Bob has no subsequent sending rule, the `` `StateB1` `` predicate remains only in the conclusion of `` `B_to_S` `` without being included in the premise of any other rule.
3. **S_to_A**:
   - The server sends two messages in one rule, so the `` `StateS1` `` predicate remains in the conclusion without any premise dependency for consecutive sending.
4. **B_Verify**:
   - This is the final rule, and no `` `State` `` predicate is required in its premise.

This structure aligns with your preferences for handling consecutive sendings and managing `` `State` `` predicates in the Tamarin Prover script.

Remove the B_verify rule.

Here's the updated script with the `` `B_Verify` `` rule removed:

```prolog
theory Yahalom
begin

builtins: symmetric-encryption

functions: dec/1, inc/1

equations: inc(dec(x))=x

// ====================
// == General rules ===
// ====================

// A and B already share a long-term symmetric key with S

rule Init:
  [ Fr(~kxs) ]
  --[ KeyGen($X) ]->
  [ !LongtermKey(~kxs, $X) ]

rule Reveal_Longterm_Key:
    [ !LongtermKey(~sk, $A) ]
  --[ Reveal($A) ]->
    [ Out(~sk) ]

// ====================
// == Protocol rules ==
// ====================
```

```
// 1. A -> B : A, Na
rule A_to_B:
  [ Fr(~na) ]
  --[ ]->
  [ Out(<$A, ~na>), StateA1($A, $B, ~na) ]

// 2. B -> S : {A, Na, Nb}Kbs
rule B_to_S:
  let msg = senc(<$A, ~na, ~nb>, ~kbs) in
  [
      Fr(~nb)
    , !LongtermKey(~kbs, $B)  // Declaring the owner of the long-term key
  ]
  --[ ]->
  [ Out(msg), StateB1($B, ~na, ~nb, ~kbs) ]

// 3. S -> A : {B, Kab, Na, Nb}Kas, {A, Kab}Kbs
rule S_to_A:
  let msg1 = senc(<$B, ~kab, ~na, ~nb>, ~kas)
      msg2 = senc(<$A, ~kab>, ~kbs) in
  [
      In(senc(<$A, ~na, ~nb>, ~kbs))
    , Fr(~kab)
    , !LongtermKey(~kas, $A)   // Declaring the owner of the long-term key for A
    , !LongtermKey(~kbs, $B)   // Declaring the owner of the long-term key for B
  ]
  --[ ]->
  [ Out(<msg1, msg2>), !ShorttermKey(~kab, $A, $B), StateS1($A, $B, ~na, ~nb, ~kab) ]

// 4. A -> B : {A, Kab}Kbs, {Nb}Kab
rule A_to_B_final:
  let msg1 = senc(<$A, ~kab>, ~kbs)
      msg2 = senc(~nb, ~kab) in
  [
      StateA1($A, $B, ~na)    // Including the "StateA1" predicate from A_to_B
    , !LongtermKey(~kbs, $B)  // Declaring the owner of the long-term key for B
    , In(msg1)
    , In(msg2)
  ]
  --[ ]->
  [ Out(<msg1, msg2>), StateA2($A, $B, ~kab, ~nb) ]

end
```

Explanation of Changes:

- **Removed `B_Verify`**: The final verification rule for Bob has been removed, so the protocol script now ends with **`A_to_B_final`**.

This completes the adjustments as requested.