Formal Methods In Software Development

Needham-Schroeder Cryptography Protocol

Riccardo La Marca

May 7, 2022

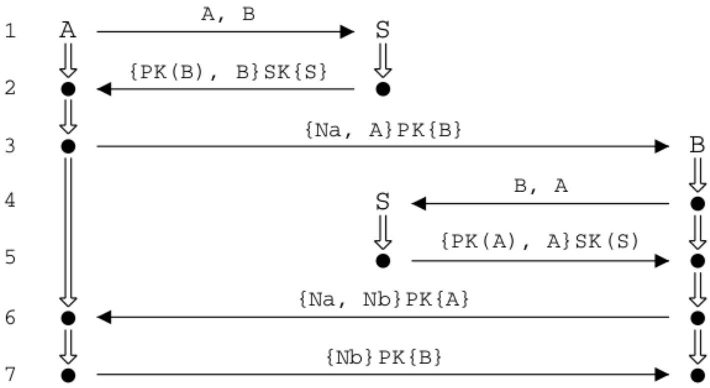
The Needham-Schroeder Protocol

Needham-Schroeder Protocol
Introduction

- Well known authentication protocol
- Aims to provide a secure communication
- Two main roles *initiator* A and a *responder* B
- Once authenticated A and B can exchange messages
- Uses *Public Key Cryptography*, i.e., any agent H has:
  - a public keys `PK(H)`
  - a secret keys `SK(H)`

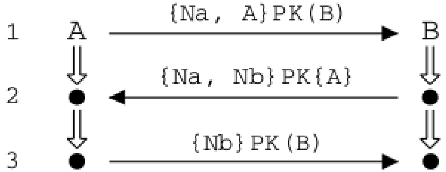
Needham-Schroeder Protocol

Complete model



- Assuming A and B know each other's public keys
- Four of the seven steps can be removed
- Steps 1, 2, 4 and 5 have been removed

Needham-Schroeder Protocol
Reduced model



The PROMELA Model

- Reveals flaws in the protocol not in the cryptosystem
- Processes for principals (A and B) and for Intruder
  - process `PIni` for the initiator
  - process `PRes` for the responder
  - process `PI` for the intruder
- Four boolean global variables:
  - `IniRunningAB` = 0;
  - `IniCommitAB` = 0;
  - `ResRunningAB` = 0;
  - `ResCommitAB` = 0;

- Messages exchanged using two handshake channels
- Each channels represent a type of message
- Messages like $\{x_1, x_2\}\texttt{PK}(x_3)$ are sent via

```
chan ca = [0] of {mtype, mtype, mtype, mtype}
```

- Messages like $\{x_1\}\texttt{PK}(x_2)$ are sent via

```
chan cb = [0] of {mtype, mtype, mtype}
```

- Where `mtype = {A, B, I, Na, Nb, gD, R}`

Needham-Schroeder Protocol

PROMELA Model - PIni

```
proctype PIni(mtype self; mtype party; mtype nonce){
    mtype g1;
    atomic {
        IniRunning(self, party);
        ca ! self, nonce, self, party;
    }
    atomic {
        ca ? eval(self), eval(nonce), g1, eval(self);
        IniCommit(self,party);
        cb ! self, g1, party;
    }
}
```

Needham-Schroeder Protocol

PROMELA Model - PIni (2)

- IniRunning and IniCommit are macros;

```
#define IniRunning(x, y) if
    :: ((x == A) && (y == B)) -> IniRunningAB = 1
    :: else skip
    fi

#define IniCommit(x, y) if
    :: ((x == A) && (y == B)) -> IniCommitAB = 1
    :: else skip
    fi
```

Needham-Schroeder Protocol

PROMELA Model - PRes

```
proctype PRes(mtype self; mtype nonce) {
    mtype g2, g3;
    atomic {
        ca ? eval(self), g2, g3, eval(self);
        ResRunning(g3, self);
        ca ! self, g2, nonce, g3;
    }
    atomic {
        cb ? eval(self), eval(nonce), eval(self);
        ResCommit(g3, self);
    }
}
```

Needham-Schroeder Protocol

PROMELA Model - PRes (2)

- ResRunning and ResCommit are macros;

```
#define ResRunning(x, y) if
    :: ((x == A) && (y == B)) -> ResRunningAB = 1
    :: else skip
    fi

#define ResCommit(x, y) if
    :: ((x == A) && (y == B)) -> ResCommitAB = 1
    :: else skip
    fi
```

Needham-Schroeder Protocol

PROMELA Model - `PI`

- It models the Introduder process
- It can listen on the channel and intercepts messages
- It can also sends messages either to A (`PI`) or B (`PRes`)
- It has an initial knowledge that comprehends
  - Identity and public keys of both A and B
  - Its own identity, public and secret keys
  - The generic data gD
- It can gain different knowledge intercepting messages
  - Nonces: `Na` and `Nb`
  - Entire messages: `{Na, Nb}PK(A)`, `{Na, A}PK(B)` and `{Nb}PK(B)`
- This knowledge is expressed as boolean variables

Needham-Schroeder Protocol

PROMELA Model - PI (2)

```
proctype PI () {
  bit kNa = 0;
  bit kNb = 0;
  bit k_Na_Nb__A = 0;
  bit k_Na_A__B  = 0;
  bit k_Nb__B    = 0;
  mtype x1 = None;
  mtype x2 = None;
  mtype x3 = None;
do
  :: ca ! B, gD, A, B
  :: ca ! B, gD, B, B
  :: ca ! B, gD, I, B
```

```
:: ca ! B, A,  A, B
:: ca ! B, A,  B, B
:: ca ! B, A,  I, B
:: ca ! B, B,  A, B
:: ca ! B, B,  B, B
:: ca ! B, B,  I, B
:: ca ! B, I,  A, B
:: ca ! B, I,  B, B
:: ca ! B, I,  I, B
:: ca ! (kNa -> A : R), Na, Na, A
:: ca ! (((kNa && kNb) || k_Na_Nb__A)
     -> A : R), Na, Nb, A
:: ca ! (kNa -> A : R), Na, gD, A
:: ca ! (kNa -> A : R), Na, A, A
```

Needham-Schroeder Protocol

PROMELA Model - PI (3)

```
:: ca ! (kNa -> A : R), Na, B, A
:: ca ! (kNa -> A : R), Na, I, A
:: ca ! ((kNa || k_Na_A__B)
  -> B : R), Na, A, B
:: ca ! (kNa -> B : R), Na, B, B
:: ca ! (kNa -> B : R), Na, I, B
:: ca ! (kNb -> B : R), Nb, A, B
:: ca ! (kNb -> B : R), Nb, B, B
:: ca ! (kNb -> B : R), Nb, I, B
:: cb ! ((k_Nb__B || kNb)
  -> B : R), Nb, B
```

```
:: d_step {
    ca ? _, x1, x2, x3;
    if
    :: (x3 == I) -> k(x1);
                    k(x2)
    :: else k3(x1, x2, x3)
    fi;
    x1 = None;
    x2 = None;
    x3 = None;
}
```

Needham-Schroeder Protocol

PROMELA Model - PI (3)

```
    :: d_step {
         cb ? _, x1, x2;
         if
         :: (x2 == I) -> k(x1);
         :: else k2(x1, x2)
         fi;
         x1 = None;
         x2 = None;
    }
od;
}
```

Needham-Schroeder Protocol

PROMELA Model - PI (3)

- Where k, k2 and k3 are macros

```
#define k(x1) if                \
    :: (x1 == Na) -> kNa = 1 \
    :: (x1 == Nb) -> kNb = 1 \
    :: else skip             \
    fi
#define k2(x1, x2) if                          \
    :: (x1 == Nb && x2 == B) -> k_Nb__B = 1 \
    :: else skip                            \
    fi
#define k3(x1, x2, x3) if                                    \
    :: (x1 == Na && x2 == A  && x3 == B) -> k_Na_A__B  = 1 \
    :: (x1 == Na && x2 == Nb && x3 == A) -> k_Na_Nb__A = 1 \
    :: else skip                                          \
    fi
```

```
init {
    atomic {
        if
        :: run PIni(A, I, Na);
        :: run PIni(A, B, Na);
        fi;

        run PRes(B, Nb);

        run PI();
    }
}
```

From PROMELA To Murϕ Model

- Basically a Murϕ model is composed of
  - definition of constants
  - definition of types using the keyword `Type`
  - definition of global variables using `Var`
  - definition of `function` and `procedure`
  - definition of the `startstate`
  - list of `rules` or `ruleset`
  - definition of the `invariant`

Needham-Schroeder Protocol

From PROMELA To Murϕ Model - Types

- Some types has been rewritten like:
  - `mtype: enum{None, A, B, I, Na, Nb, gD, R}`
  - `bit: 0..1`
  - `byte: 0..255`
- Then I added a new type
  - `proc: enum{pi, pres, pini}`
- I defined a `record` type also for each process and channel

```
PIni_type : record
    pc       : byte;
    slef     : mtype;
    party    : mtype;
    nonce    : mtype;
    g1       : mtype;
    runable: boolean;
end;
```

```
ca_type : record
    x1    : mtype;
    x2    : mtype;
    PK    : mtype;
    x3    : mtype;
    size : bit;
    send : proc;
end;
```

Needham-Schroeder Protocol

From PROMELA To Murϕ Model - Types (2)

- For processes
  - `pc` is the program counter
    - rule out which rule of a process should be fired
  - `runable`
    - if False, none rule of a process can be fired
    - i.e., the process is blocked
- For channels
  - `size`, resemble the size of the channel
    - since we have rendez-vous ch, it's value is 0 or 1
  - `send`, who sent the message

- Each global variable of the PROMELA model has been defined
- Including also channels
- Here we have also a global variable for each process

```
Var
    IniRunningAB: boolean;
    IniCommitAB : boolean;
    ResRunningAB: boolean;
    ResCommitAB : boolean;

    ca : ca_type;
```

```
cb : cb_type;

PI   : PI_type;
PIni : PIni_type;
PRes : PRes_type;
init : initial_type;
```

- Each Macro in the PROMELA Model is now a `procedure`
- Also some `functions` have been defined
  - for instance, for sending a message
  - or receiving a message

```
procedure IniRunning(x: mtype;
                     y: mtype
);
begin
    if ((x = A) & (y = B)) then
        IniRunningAB := TRUE;
    endif;
end;
```

```
procedure retrieve_ca();
begin
    ca.x1 := None;
    ca.x2 := None;
    ca.PK := None;
    ca.x3 := None;
    ca.size := 0;
    ca.send := none;
end;
```

Needham-Schroeder Protocol

From PROMELA To Murϕ Model - Initial state

- Each global or local boolean variable has been set to FALSE
- Each processes' `pc` has been set to 0
- Each processes' `runable` has been set to FALSE
  - set to TRUE when initiated by the `init` process
- Each processes' `mtype` variable has been set to `None`

```
startstate
begin
    IniRunningAB := FALSE; IniCommitAB  := FALSE;
    ResRunningAB := FALSE; ResCommitAB  := FALSE;
    clear PI; clear PRes; clear PIni;
    clear ca; clear cb;
    init.runable := TRUE; init.pc := 0;
end
```

- A set of rules has been created for each processes
  - for each line of the PROMELA code of a process
  - a rule has been created, to simulate the interleaving
  - except `atomic` or `d_step` sections
  - in this case the entire code of the section is inside a single rule
- Every rule's guard is something like

```
rule "..."
    <process>.runable = TRUE
    & <process>.pc = x & ...      ==>
begin
    ...
    <process>.pc := y;
end;
```

Needham-Schroeder Protocol

From PROMELA To Murϕ Model - Rules (2)

- If we have to send/receive to/from a channel
  - insert in the guard the check if the channel is/isn't empty
- To simulate the handshaking
  - when a process sends then set `<process>.runable := FALSE`
  - when a process receives unblock the sender process
    - this is possible given the sender from the received message
- On the last rule of a process
  - it sets its `runable` to FALSE
- To simulate the `do` statement
  - we have a rule for each statement inside the `do`
  - since each statement inside the `do` is atomic (or just single)
  - after the execution of the rule, the `pc` is not changed
  - in this way, except for `runable = FALSE`, the guard is always TRUE

```
rule "PRes atomic 1"
    PRes.runable = TRUE & PRes.pc = 0 & isEmptyCa() = FALSE ==>
    var message: ca_type;
begin
    message := get_ca();
    if ((PRes.slef = message.x1) & (PRes.slef = message.x3)) then
        retrieve_ca();
        unblock_proc(message.send);
        PRes.g2 := message.x2;
        PRes.g3 := message.PK;
        ResRunning(PRes.g3, PRes.slef);
        send_ca(PRes.slef, PRes.g2, PRes.nonce, PRes.g3, pres);
        PRes.pc := 2;
        PRes.runable := FALSE;
    endif;
end;
```

From PROMELA To NuSMV Model

- Basically each NuSMV model is composed by `MODULE`s
- Each module can contains
  - `VAR` section to define local variables
  - `ASSIGN` section to define
    - initial value for variables with `init(...) := ...`
    - transitions with `next(...) := ...`
  - `TRANS` section to define transitions
    - as a set of current state/next state pairs
  - `DEFINE` section associate a symbol to an expression
- Finally, a NuSMV file must contains at least the `main` module
- Main module can contains also CTL/LTL spec to be checked

Needham-Schroeder Protocol

From PROMELA To NuSMV Model - Modules and Global variables - `VAR`

- Each process has been traslated into a NuSMV Module
- Now, in the main module we have the
  - definition of global variables
  - instantiation of modules for channels
  - instantiation of the module for the init process
  - all the other modules are instantiated by the init one
- To simulate the asynchronous behavior
  - PROMELA processes instantiated using `process` keyword
  - i.e., for instance `init: process init_process;`
  - where `init_process` is a module representing the init process

- Each state is an assignment of values to variables
- Initial state is a variable initialization for each module
  - using the `init` operator
- Each transition is a change of value of a variable
- To change a value we use the `next` operator
- Since from each state we can have multiple transition
- The `next` is usually combined with the case operator
  - a set of pair <condition_*i*>: <simple_expression*i*>;
  - usually the last pair is `TRUE: X;`
  - given the name of the updating variable being `X`
  - conditions for updating a variable are like

```
(pc = x & ...)  :  ...
```

```
MODULE PI(sup, run)
VAR
    x1: {None, A1, B, I, Na, Nb, gD, R};
    pc: 1..98;
    ...
ASSIGN
    init(x1) := None; init(pc) := 1;
    next(x1) := case
        pc = 52 & !(sup.ca_is_empty): sup.ca.x2;
        pc = 77: None;
        pc = 80 & !(sup.cb_is_empty): sup.cb.PK;
        pc = 96: None;
        TRUE: x1;
    esac;
    ...
```

- Also in this case, each module has a
  - program counter, `pc`
  - boolean variable `runable`
- In this case the `runable` is used only to
  - block the process until it will not be instantiated by the `init`
  - unlike the Murϕ model it does not regolate the handshaking
  - once the process starts `runable` will be always set to TRUE
- The program counter just
  - guide the process towards the correct next state (1 or more)

Needham-Schroeder Protocol

From PROMELA To NuSMV Model - States and Transitions - ASSIGN (4)

- To simulate the do operator starting from pc = x
  - we set the next to choose among multiple choices (let pc = y)
  - after executing the operation we reset pc = x

```
next(pc) := case
    pc = 1 & (      sup.ca_is_empty | sup.cb_is_empty
              | !(sup.ca_is_empty) | !(sup.cb_is_empty)):
                     { 2,  4,  6,  8, 10, 12, 14, 16, 18, 20,
                      22, 24, 26, 28, 30, 32, 34, 36, 38, 40,
                      42, 44, 46, 48, 50, 52, 80};
    pc = 2: 3;
    pc = 3 & sup.ca_is_empty: 1;
    ...
esac;
```

- In this section I define when a process
  - can start/continue its execution
  - have to be blocked
- In this way I also handle handshaking
  - i.e., if a process would like to send a message
  - this can be done iff `pc = x & sup.ca_is_empty`
  - after sending the message the `pc = y`
  - the process can go on iff `pc = y & sup.ca_is_empty`
  - if it is not then the process is blocked
- When a module is executed as a process
  - it has an implicit variable called `running`
  - if it is TRUE then the process can be executed
  - otherwise, it cannot.
  - using the TRANS section we can modify the value of it
  - according to some conditions

```
TRANS -- Module PRes
(
      pc = 1 & !(!(sup.ca_is_empty) & (  (slef = sup.ca.x1)
                                       & (slef = sup.ca.x3)))
    | pc = 7 & !(sup.ca_is_empty)
    | pc = 8 & !(sup.ca_is_empty)
    | pc = 9 & !(!(sup.cb_is_empty) & (  (slef = sup.cb.x1)
                                           & (nonce = sup.cb.PK)
                                           & (slef = sup.cb.x2)))
    | pc = 15
    | !runable
        -> !running
)
```

Verification

- Let's define the technique used for property specification
  - say that X takes part in a protocol run with Y
  - if X has initiated a protocol session with Y
  - say that X commits to a session with Y
  - if X has correctly concluded a protocol session with Y
- Thus, we can say that
  - Authentication of B to A means
    - A commits to a session with B
    - B has indeed taken part in a run with A
  - and viceversa
- Finally
  - `IniRunningAB = TRUE` iff A takes part with B
  - `ResRunningAB = TRUE` iff B takes part with A
  - `IniCommitAB = TRUE` iff A commits to a session with B
  - `ResCommitAB = TRUE` iff B commits to a session with A

Needham-Schroeder Protocol

Verification - Properties definition - LTL specification (2)

- Now, we can rewrite the two properties
- Authentication of B to A (Property P1)

$$\mathbf{G}((\mathbf{G}\neg\texttt{IniCommitAB}) \vee (\neg\texttt{IniCommitAB}\ \mathbf{U}\ \texttt{ResRunningAB}))$$

- Authentication of A to B (Property P2)

$$\mathbf{G}((\mathbf{G}\neg\texttt{ResCommitAB}) \vee (\neg\texttt{ResCommitAB}\ \mathbf{U}\ \texttt{IniRunningAB}))$$

- I have added also a *deadlock* property (property P3)
  - in this case the model is in a deadlock whenever
  - we are in a state from which we cannot go further
  - and so we are unable to reach the authentication

Needham-Schroeder Protocol

Verification - Properties definition - LTL specification P1/P2 (3)

- PROMELA, `never claim` generated from

```
! ([] (([] !(IniCommitAB)) || (!(IniCommiAB) U (ResRunningAB))))
```

- Murϕ, we have the following invariant property

```
invariant "[] (([] !(p)) || (!(p) U (q)))"
    (!IniCommitAB) | (  (!IniCommitAB & !ResRunningAB)
                      | ( IniCommitAB &  ResRunningAB)
                      | (!IniCommitAB &  ResRunningAB)
                     )
```

- NuSMV, we have the following LTLSPEC in `main`

```
LTLSPEC G ((G !IniCommitAB) | (!IniCommitAB U ResRunningAB));
```

- PROMELA, I just added a `end` label to valid end states
  - in this case a valid end state is the last step of both PIni and PRes
- Murϕ, I have add two global variables resemble the `end` label
  - we have `end_PIni`, for PIni
  - we have `end_PRes`, for PRes

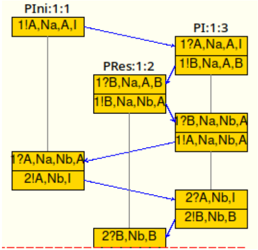
```
invariant "Deadlock"
    !(  PIni.runable | PRes.runable
      | PI.runable | init.runable) -> (end_PIni & end_PRes)
```

- NuSMV, we have the following `LTLSPEC` in `main`

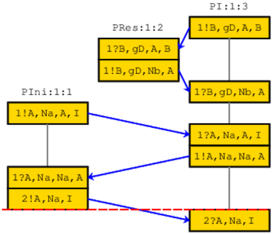
```
LTLSPEC G F (  IniCommitAB & IniRunningAB
             & ResCommitAB & ResRunningAB)
```

- Given commands for PROMELA
  - `./spin -a o_needham.pml`
  - `g++ pan.c -o pan.o`
  - `./pan.o -v`
- Given commands for Murϕ
  - `./mu o_needham.m -c`
  - `g++ o_needham.cpp -o o_needham.exe -I<lib_path>`
  - `./o_needham.exe -ndl -tv -d out`
- Given command for NuSMV
  - `./NuSMV o_needham.smv`

Property	Result	*PROMELA*		*Murϕ*		*NuSMV*	
		States	*Time*	*States*	*Time*	*States*	*Time*
P1	OK	1195	0,045s	8933	1,648s	3976997	1:01,93s
P2	FAIL	333	0,042s	6924	0,64s	3302921	1:31,93s
P3	FAIL	5	0,045s	2173	0,269s	3423932	6:44s



(a) Error Trace for P2



(b) Error Trace for P3