Formal analysis of cryptographic protocols

Cryptographic protocols

- mechanisms for securing computer systems
- use cryptographic primitives
  - (a)symmetric encryption
  - digital signatures
  - hash functions
- The most significant classes are protocols for
  - authentication - ensuring the identity of the participant
  - key establishment - creating and distributing secret keys

Možnosti automatických nástrojů

- Systémy analýzy formální bezpečnosti pracují s Dolev-Yao modelem útočníka
  - Kryptografická primitiva nedají útočníkovi žádnou informaci
  - Útočník vše monitoruje a má možnost jakýchkoliv zásahů do chodu protokolu (protokolů)
- První bod podcenění reality, druhý bod přecenění reality

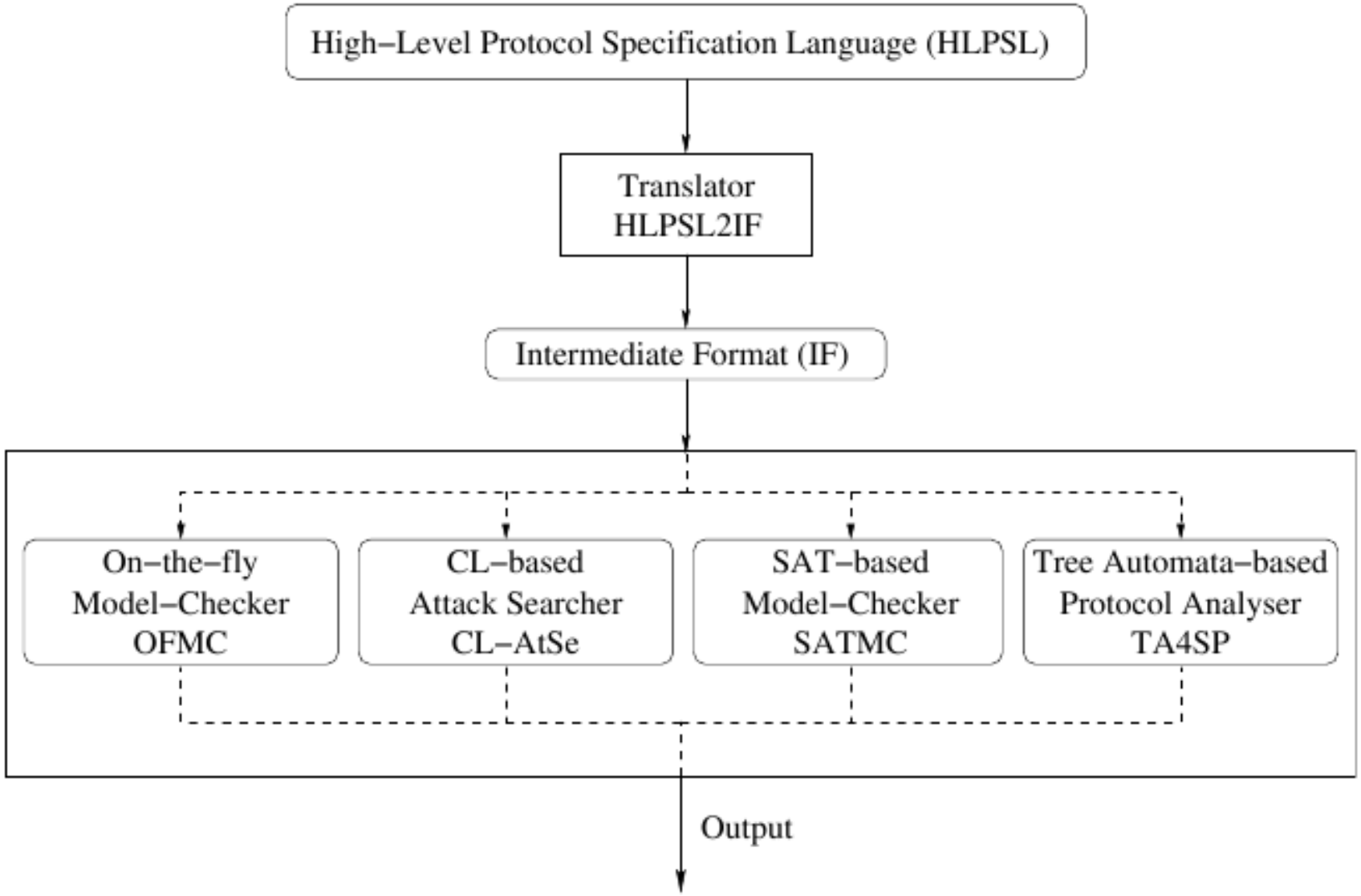
Úspěšné netriviální analýzy

- Lowe 1995 – Needham-Schroeder - útok
- Paulson 2002 – SET – verifikace (Isabelle)
- Armando 2005 – Siemens H.530 – útok a fix (AVISPA)
- Armando 2008 – SAML-based SSO pro GoogleApps – útok (AVANTSSAR – SATMC)
- seL4 microkernel – formálně verifikované jádro operačního systému (Isabelle)

AVISPA

- Automated Validation of Internet Security Protocols and Applications
- Obsahuje
  - Modulární silný formální jazyk HLPSL
  - Integruje několik usuzovacích strojů.
    - Falsifikace protokolu
    - Abstraktní verifikační metody

AVISPA základní schéma



Back-end OFMC

- On-the-Fly Model-Checker
- Provádí falsifikaci protokolu a ohraničenou verifikaci procházením přechodových stavů protokolu způsobem „demand-driven“
- Podporuje specifikaci algebraických vlastnosti kryptografických operátorů a typové i netypové protokolové modely

Back-end CL-AtSe

- Constraint-Logic-based Attack Searcher
- Provádí řešení podmínek s pomocí zjednodušovacích heuristik a technik na eliminaci renundancí
- Je stavěn modulární způsobem a lze dodat extenze.

Back-end SATMC

- SAT-based Model-Checker
- Staví logické předpoklady – formule na základě relací specifikovaných v IF
- Pak je řeší SAT řešitelem
- SAT je problém splnitelnosti logické formule – zda-li pro všechny možné vstupy má hodnotu NE. Obecně je to NP-úplný problém.

Back-end TA4SP

- Tree Automata based on Automatic Approximations for the Analysis of Security Protocols
- Aproximuje útočníkovu znalost používáním jazyku regulárních stromů a přepisováním.
- Může analyzovat libovolné množství seancí.

Jazyk HLPSL

- Specifikuje role pro účastníky protokolu.
- Složené role pro session – tedy kteří účastnící mezi sebou interagují
- Celkovou roli pro běh, sestává se s několika session, jedné pro regulérní běh a dalších v níž se vyskytuje místo jednotlivých účastníků útočník.

HLPSL – Tvorba rolí

- Ilustrace na jednoduchém protokolu
  - Wide Mouth Frog
- Alice-Bob zápis protokolu
  - A -> S : {Kab}_Kas
  - S -> B : {Kab}_Kbs
- Notace:
  - {zpráva}_klíč
  - A,B,S – účastníci
  - Kab – symetrický klíč mezi účastníky A, B

HLPSL – Tvorba rolí 2

- Alice – Bob notace popisuje jen zprávy
- Nedokáže popsat sekvenci událostí pro rozsáhlé protokoly:
- Mnohdy takové protokoly mají řídící struktury jako
  - If-then-else větvení
  - cykly

HLPSL – Tvorba rolí 3

- Každý účastník popsán v modulu s názvem „basic role“
- Později budou pro jednotlivé role zřízeny instance.
- Basic role specifikuje
  - Co účastník zná na začátku (parametry)
  - Jeho počáteční stav
  - Způsob jakým se stav může měnit (transitions)

HLPSL – tvorba rolí 4

- Pro každého účastníka zvláštní role
- Pro WFM máme role `alice`, `bob` a `server`.
- Jména rolí začínají vždy malým písmenem
- Jméno agenta hrajícího roli začíná velkým písmenem
- Nejčastěji jsou jména odpovídající zápisu v Alice-Bob notaci tedy pro WFM: `A B S`

Fragment role alice

```
role alice(A,B,S : agent,
           Kas : symmetric_key,
           SND, RCV : channel (dy))
played_by A def=
   local State: nat, Kab: symmetric_key
   init State := 0
transition
...
end role
```

agents

komunikační kanály

hraje roli

State je přirozené číslo

Přechody stavu

```
step1. State = 0 /\ RCV({Kab'}_Kas) =|>
       State':= 2 /\ SND({Kab'}_Kbs)
```

definice přechodu sestává z předpokladu a z akce.

„priming“ - `X'` je nová hodnota proměnné `X`.

Do hodnoty `X` je přiřazena hodnota `X'` až na konci přechodu.

Složené role

- Složené role – „composed roles“ spojují základní role
- Typicky složená role popisuje session.
- Kromě role `alice` zadefinujeme role `bob` a `server` s příslušnými parametry.
- Vytvoříme roli, která reprezentuje jeden běh a vytvoří jednu instanci ke každé základní roli. Tradičně se role pojmenuje `session`

Session role

```
role session(A,B,S : agent,
     Kas, Kbs : symmetric_key) def=
local SA, RA, SB, RB SS, RS: channel (dy)
composition
        alice (A, B, S, Kas, SA, RA)
     /\ bob (B, A, S, Kbs, SB, RB)
     /\ server(S, A, B, Kas, Kbs, SS, RS)
end role
```

Hlavní role

- Program musí mít nějakou main() procedůru
- V systému AVISPA je jím role, která definuje:
  - Globání konstanty
  - Složení jedné či více session kde může útočník zaujímat některou z rolí jako legitimní účastník.
  - Útočníkovu znalost. Ta typicky obsahuje:
    - jména všech agentů
    - všechny veřejné klíče, jeho soukromý klíč, klíče, které sdílí s jinými,
    - všechny veřejně známé funkce

Hlavní role – environment

```
role environment()
def=
    const a, b, s          : agent,
          kas, kbs, kis : symmetric_key
intruder_knowledge = {a, b, s, kis}
composition
      session(a,b,s,kas,kbs)
   /\ session(a,i,s,kas,kis)
   /\ session(i,b,s,kis,kbs)
end role
```

Exekuční příkaz

- Zatím jsme měli jen deklarace
- Hlavní prováděcí příkaz je
  - `environment()`

Příklad

- Jednoduchý protokol na generování nového klíče
- A-B notace
  - `A -> B: {Na}_K`
  - `B -> A: {Nb}_K`
  - `A -> B: {Nb}_K1, where K1=Hash(Na.Nb)`

Modelování sdílených informací

- Alice a Bob se musí PŘEDEM domluvit na sdílených parametrech
- Toho se dosáhne předáním stejného parametru instancím těch rolí, které parametr sdílí.

Fragment role alice

```
role alice(..,
        K: symmetric_key,  % K a Hash musí být zaslán do každé
        Hash: hash_func,   % role, protože A a B se dohodli
        ..)
... def=
  local
    ...
    State : nat        % Proměnná typicky definovaná v každé roli
  init
    State := 0
  transition
  1. State = 0 /\ RCV(start) =|>
     State'= 2 /\ Na' := new() /\ SND({Na'}_K)
  2. State = 2 /\ RCV({Nb'}_K) =|>
     State'= 4 /\ SND({Nb'}_Hash(Na.Nb'))
```

Cíle protokolu

- Jednostranná autentifikace
  - A se autentifikuje k B
  - Požadujeme silnou autentifikaci
- Klíč K1 tajný

Zápis cílů – utajení

- Jednak specifikace v transition, kde proměnná vzniká. Klausule secret

```
1. State = 1 /\  RCV({Na'}_K) =|>
   State':= 3 /\ Nb' := new()
              /\ SND({Nb'}_K) /\ K1':=Hash(Na'.Nb')
              /\ secret(K1',k1,{A,B})
```

- Za druhé v sekci goal

```
goal
  secrecy_of k1
  authentication_on bob_alice_nb
end goal
```

Zápis cílů - autentifikace

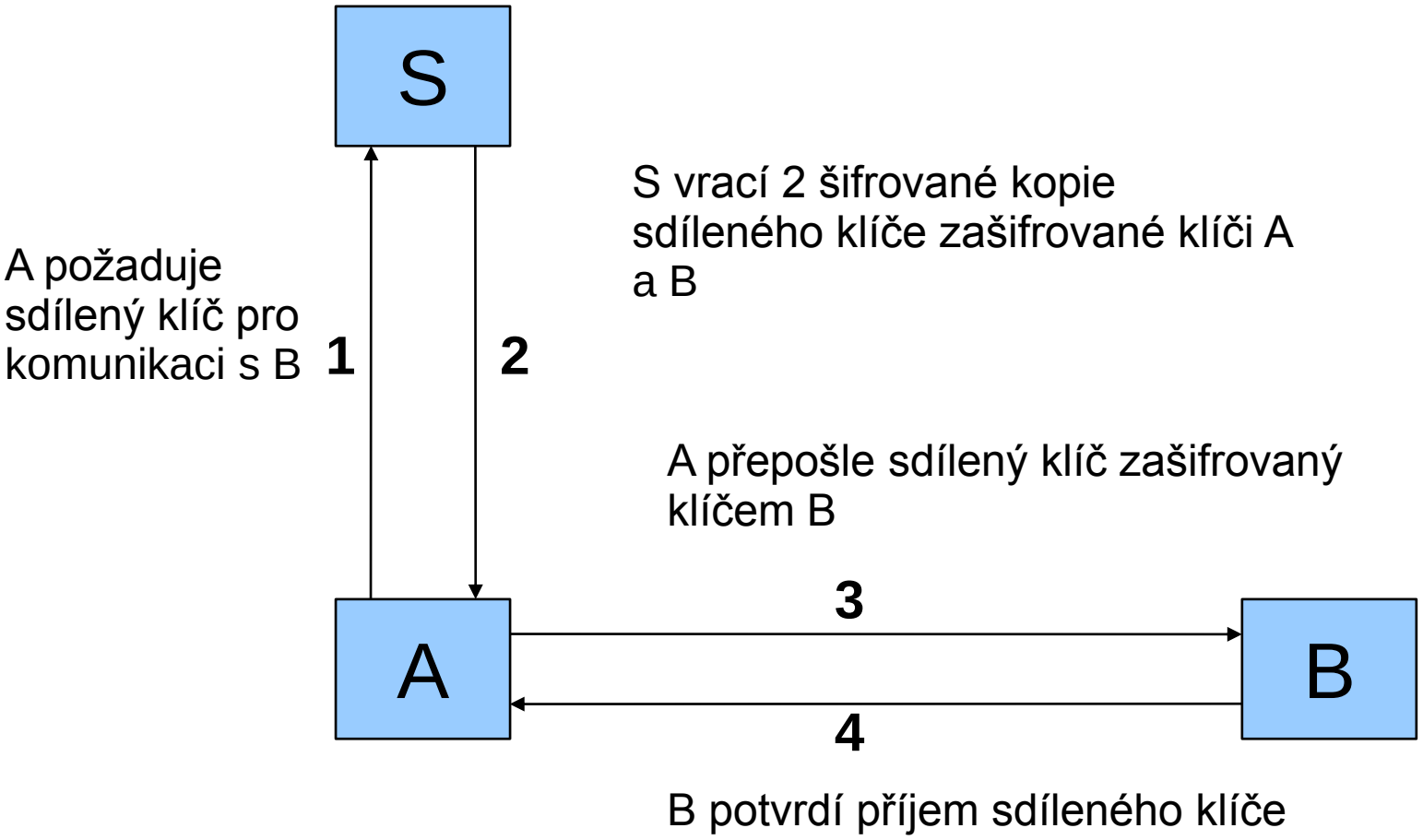
- Za prvé kdo požaduje autentikci. Sekce transition – klausule request
- `2. State = 3 /\ RCV({Nb}_K1) =|>`
- `   State’:= 5 /\ request(B,A,bob_alice_nb,Nb)`
- Za druhé kdo potvrzuje. Klausule witness
- `2. State = 2 /\ RCV({Nb’}_K =|>`
- `State’:= 4 /\ K1’:=Hash(Na’.Nb’) /\ SND({Nb’}_K1’)`
- `/\ witness(A,B,bob_alice_nb,Nb’)`
- Za třetí sekce goal

Příklad 2 - Kerberos

- Alice-Bob notace

```
A -> S: (A.B.{Na}_Ka)          % Ka mezi A a S
A <- S: (A.B.{K.Na.Ns}_Ka.     % S generuje K
                {K.Na.Ns}_Kb)  % pro oba
                 % A přepošle 2 část B)
A -> B: (A.B.{K.Na.Ns}_Kb.
                {Na.Ns}_K)     % Potvrzení
                               % B zná K
A <- B: (A.B.{Ns.Na}_K)
```

Reprezentace příkladu



Report útoku

```
ATTACK TRACE
i -> (a,3): start
(a,3) -> i: a.b.{Na(1)}_ka
i -> (s,7): a.i.{Na(1)}_ka
(s,7) -> i:
a.i.{K(2).Na(1).Ns(2)}_ka.{K(2).Na(1).Ns(2
)}_ki
i -> (a,3): a.b.{K(2).Na(1).Ns(2)}_ka.x253
(a,3) -> i: a.b.x253.{Na(1).Ns(2)}_K(2)
i -> (a,3): a.b.{Ns(2).Na(1)}_K(2)
```

Příklad 3

- Andrew Secure RPC protokol
- Alice-Bob notace
  - `A -> B : A.{Na}_Kab`
  - `B -> A : {Na+1.Nb}_Kab`
  - `A -> B : {Nb+1}_Kab`
  - `B -> A : {K1ab.N1b}_Kab`

Operátory

- HLPSL nepodporuje aritmetické operace.
- Hodnota s přičtenou 1 se přenáší pouze zašifrovaná a nikde nepotřebujeme inverzní operaci
- V modelu nahradíme sčítání operací hash funkce.

Bezpečnostní cíle

- Andrew Secure RPC má zajistit utajení a vzájemnou autentifikaci
- Klíč K1ab a nonce N1b jsou tajné
- Pravidlo, přidat podmínku secret k poslednímu transakčnímu pravidlu role, která vytváří proměnnou
  - `/\ secret(K1ab',k1ab,{A,B})`
  - `/\ secret(N1b' ,n1b, {A,B})`

Cíl utajení

- Specifikuje se (utajovaná proměnná, identifikátor, mezi kým je tajná informace sdílena)
  - `/\ secret(K1ab',k1ab,{A,B})`
- Je to specifikace události při trasování modelu.
- Interpretace útoku je v sekci goal
  - `secrecy_of k1ab, n1b`

Cíl autentifikace

- Klausule `request`
  - `/\ request(A,B,alice_bob_k1ab,K1ab')`
- `A` přijímá `K1ab` a spoléhá se na to, že `B` existuje a souhlasí s domluvou s `A` na této hodnotě. (a chce ji použít pro protokol id `alice_bob_k1ab`
- Klausule `wrequest`
  - Dostačuje, aby `B` existovalo v minulosti a v tom čase se dohodlo na `K1ab`

Cíl autentifikace

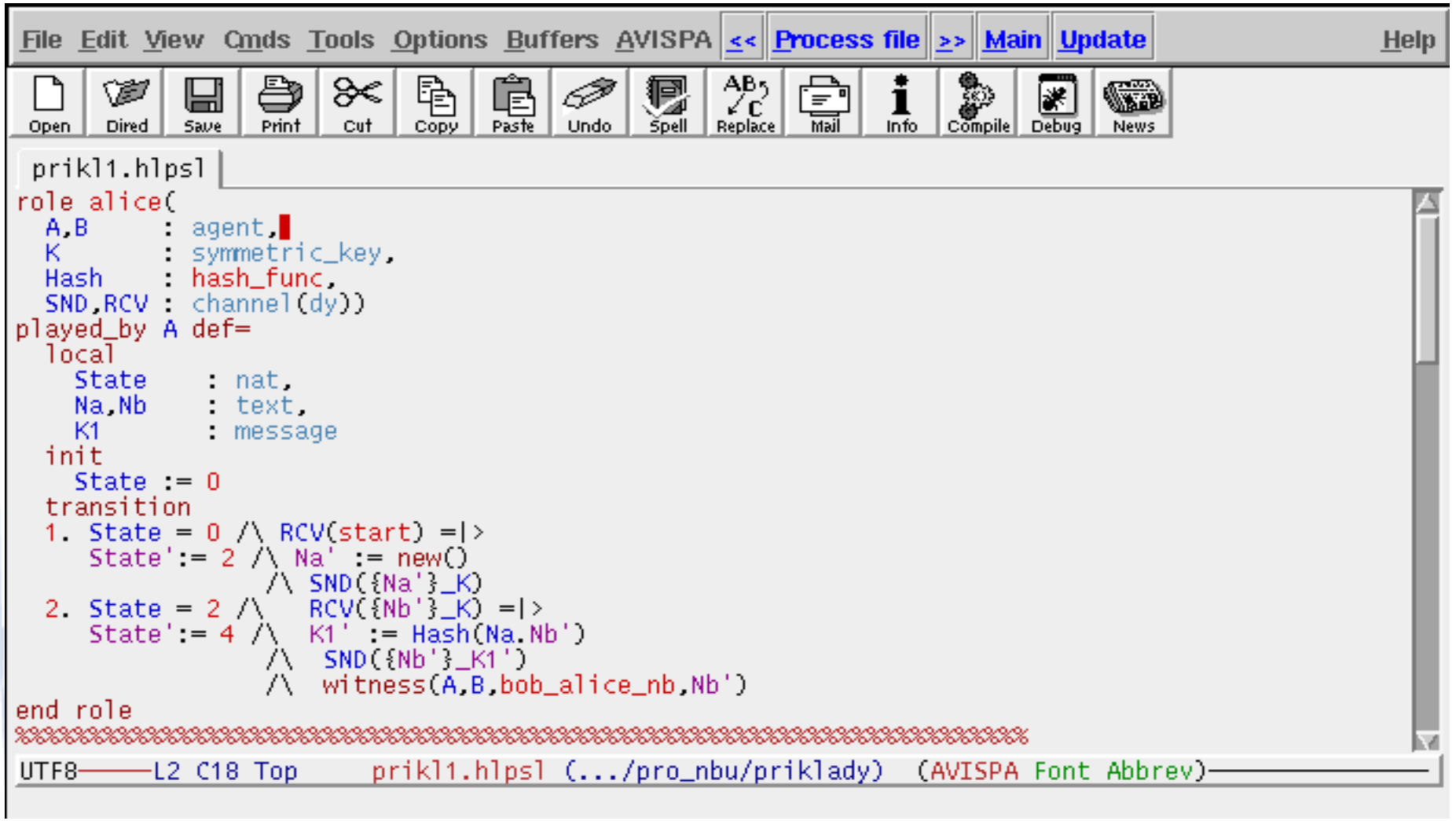
- Klausule witness
  - `/\ witness(B,A,alice_bob_k1ab,K1ab')`
- `B` navrhuje, že chce být partner s `A` a domluvit se na `K1ab`. Autentifikační požadavek je identifikován `alice_bob_k1ab`
- Sekce goal
  - `authentication_on bob_alice_nb`
  - `authentication_on alice_bob_na`
  - `authentication_on alice_bob_k1ab`

Útok

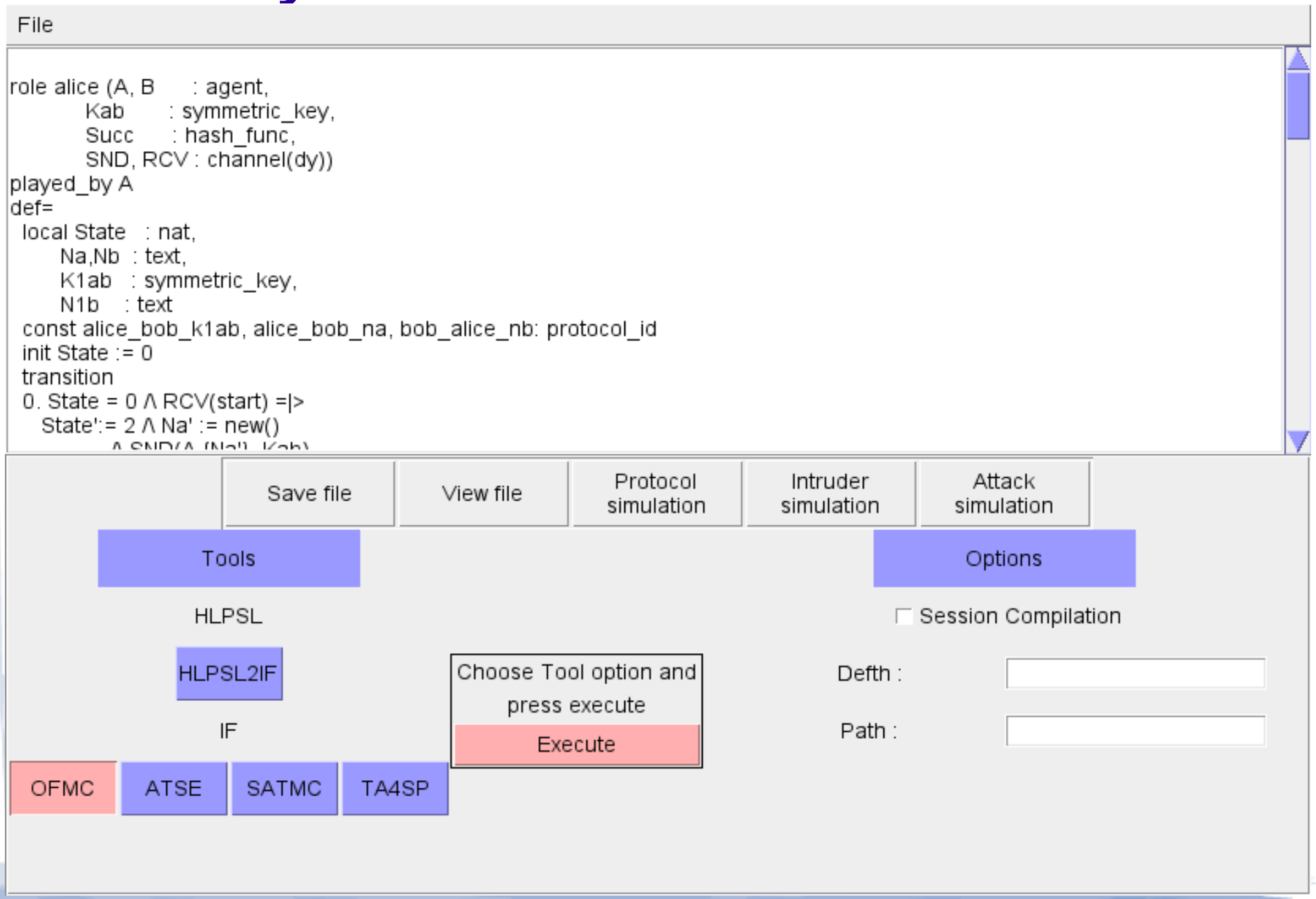
```
i -> (a,3): start
(a,3) -> i: a.{Na(1)}_kab
i -> (a,6): start
(a,6) -> i: a.{Na(2)}_kab
i -> (b,3): a.{Na(2)}_kab
(b,3) -> i: {succ(Na(2)).Nb(3)}_kab
i -> (b,6): a.{Na(1)}_kab
(b,6) -> i: {succ(Na(1)).Nb(4)}_kab
i -> (a,3): {succ(Na(1)).Nb(4)}_kab
(a,3) -> i: {succ(Nb(4))}_kab
i -> (a,6): {succ(Na(2)).Nb(3)}_kab
(a,6) -> i: {succ(Nb(3))}_kab
i -> (b,3): {succ(Nb(3))}_kab
(b,3) -> i: {K1ab(7).N1b(7)}_kab
i -> (a,3): {K1ab(7).N1b(7)}_kab
i -> (a,6): {K1ab(7).N1b(7)}_kab
```

Podpůrné prostředí Xemacs

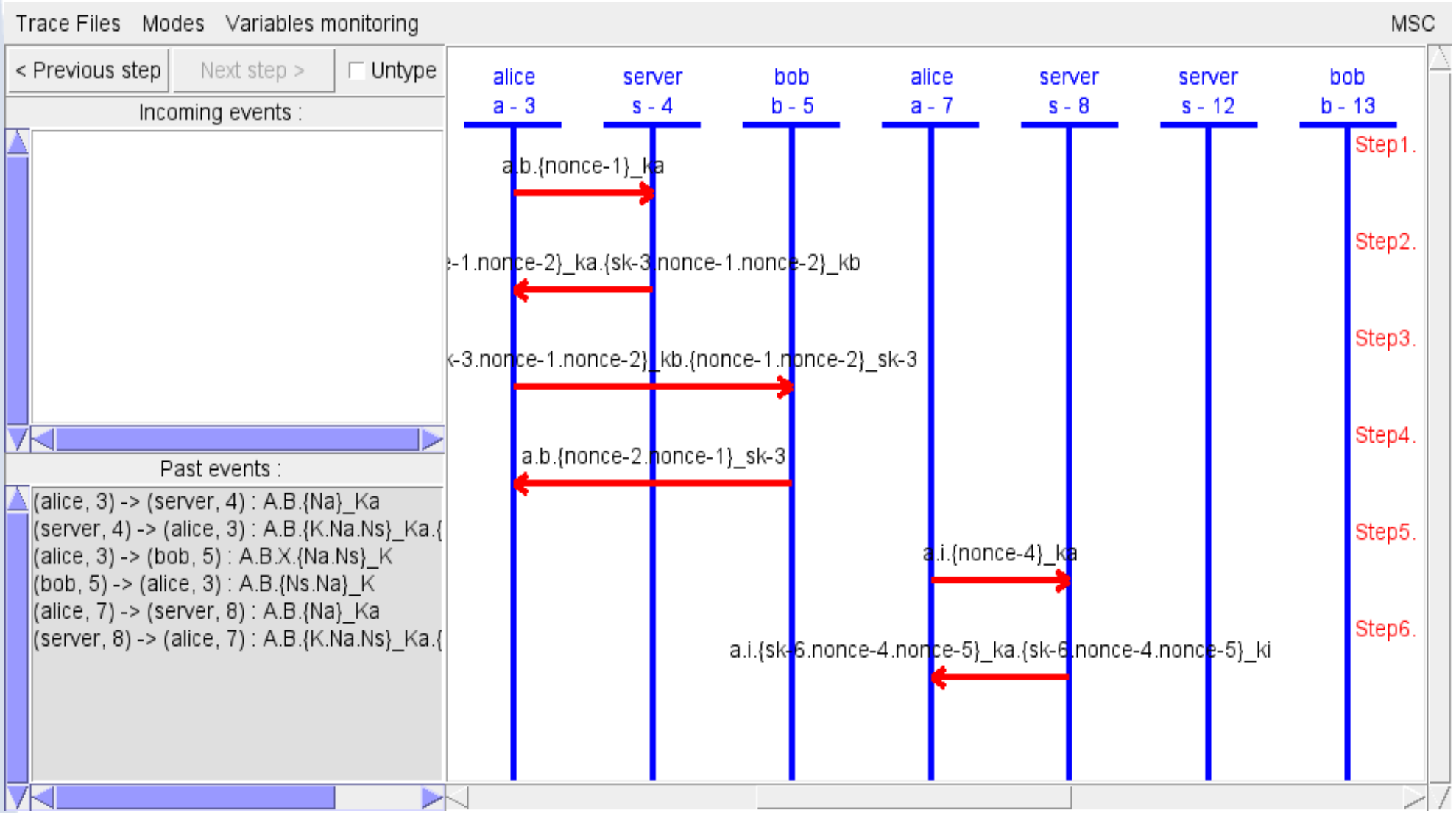
- Xemacs mód



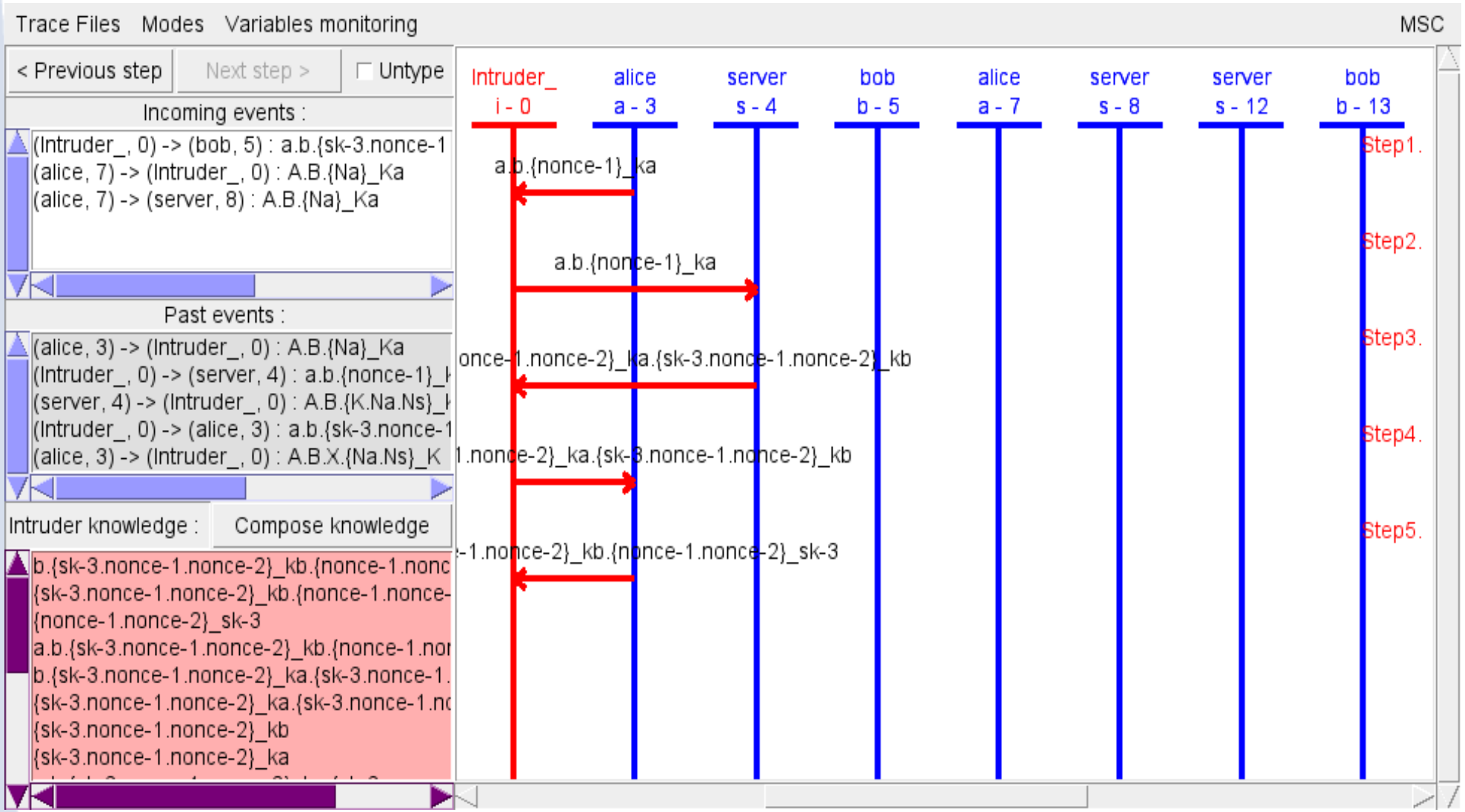
Security Protocol ANimator - SPAN



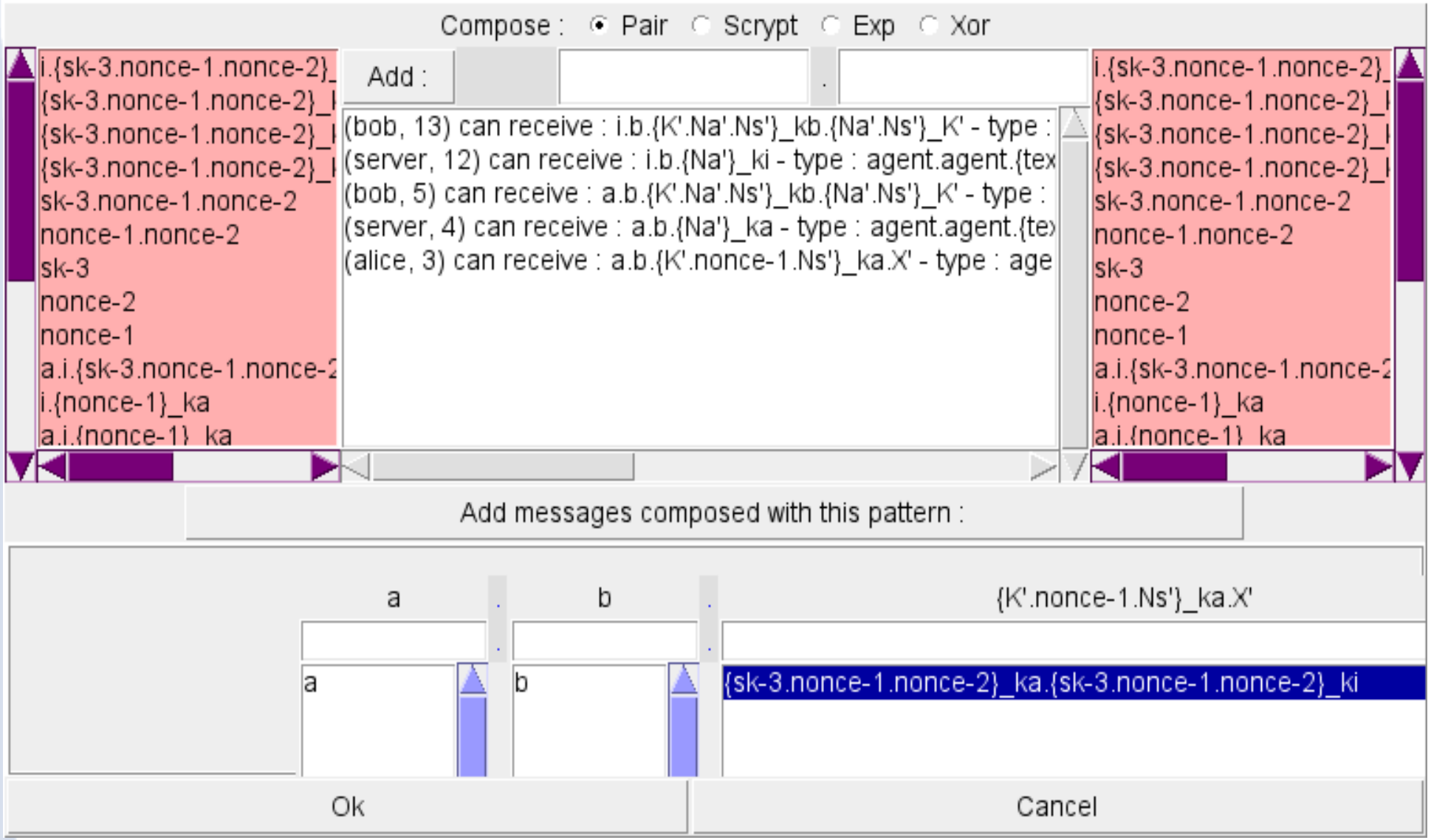
SPAN simulace protokolu



SPAN simulace útoku



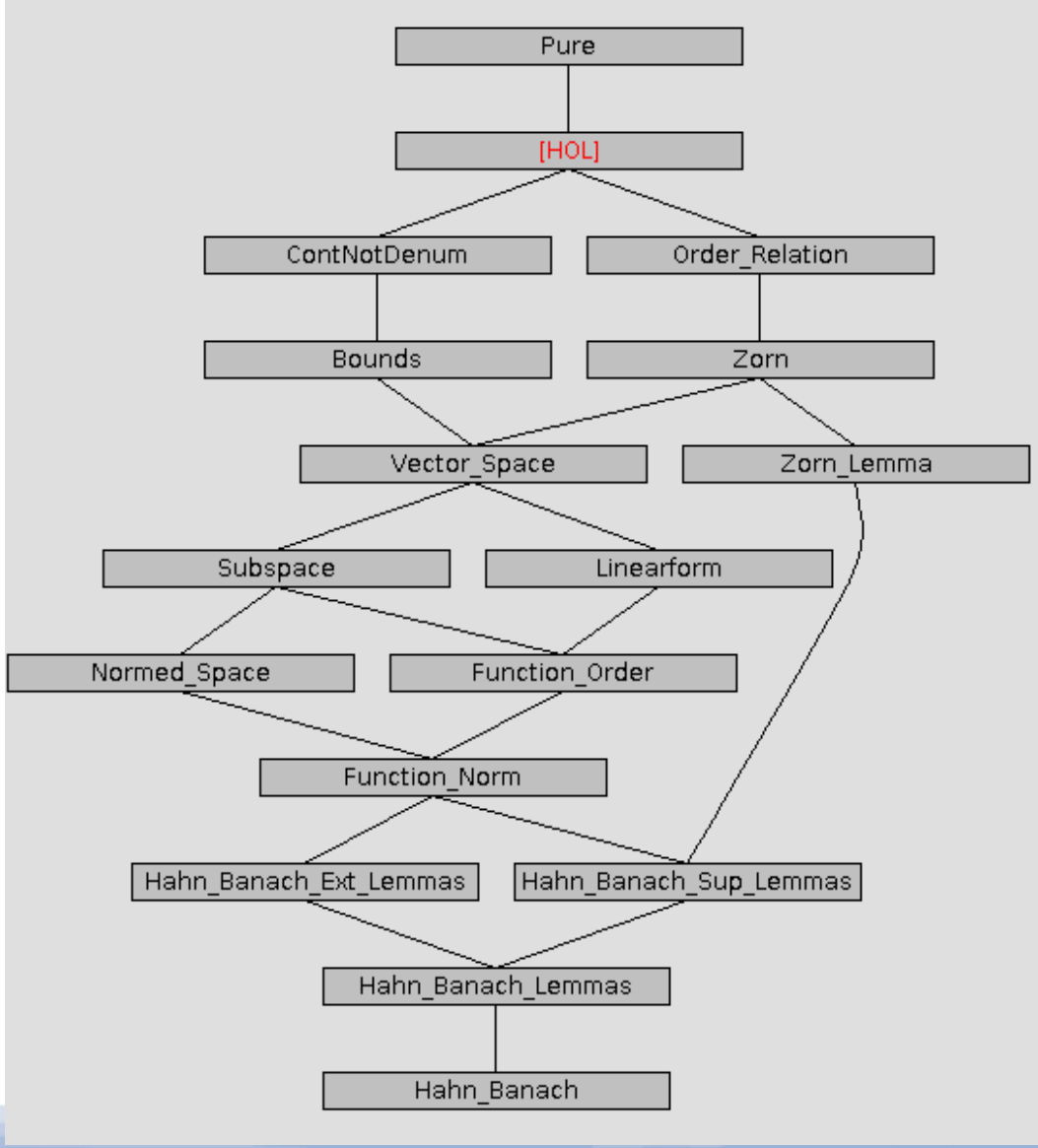
SPAN konstrukce zpráv



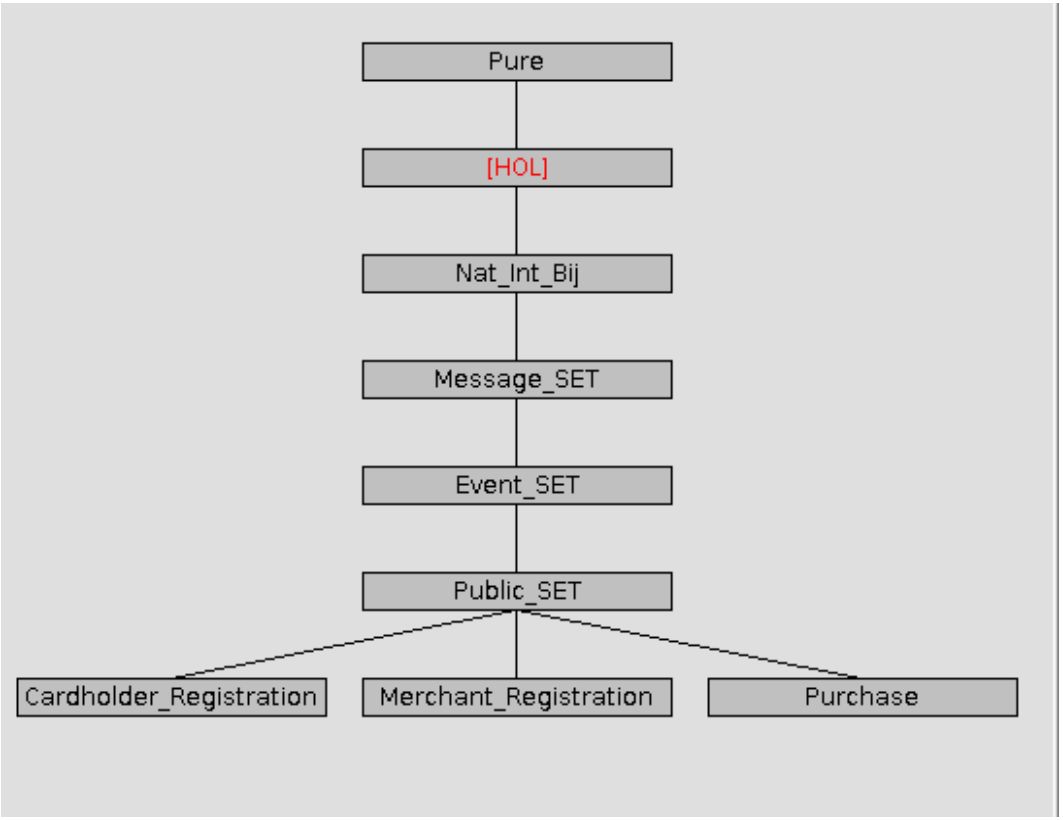
Isabelle/HOL

- Obecný interaktivní dokazovací stroj pro teorémy
- Interaktivní znamená, že tvorba důkazu je interaktivním způsobem.
- Stále se na něm pracuje. Začátky druhá polovina 80 let, poslední verze červen 2010.

Závislosti pro Hahn-Banach



Závislosti teorií pro SET



Isabelle – Příklady použití

- Formální matematika, logické formalismy, vývoj programů, verifikace.
- Důkaz DSA podpisového schématu
- Verifikace bankovního protokolu SET
- Verifikace JavaCard – podmnožiny jazyka Java pro smartcards a Java.
- Verifikace mikrojádra L4 operačního systému
  - 11 lidí, 200 000 řádků kódu v Isabelle/HOL, celkově 25-30 člověko-roků.

Základní struktury jazyka

- Teorie – kolekce typů, funkcí a teorémů
  - Jako modul v programovacím jazyce
  - `theory T`
  - `imports B1 . . . Bn`
  - `begin`
  - *`declarations, definitions, and proofs`*
  - `end`
- B1 … Bn jsou teorie na nichž je naše založena

Základní struktury jazyka

- Typy, výrazy
  - Silně typovaný jazyk. Typy mohou být
    - Základní typy
    - Konstruktory typů – list
    - Typy funkcí (i více argumentů)
    - Typové proměnné
  - Výrazy – jsou aplikace funkce na argumenty
    - Podmínkové výrazy if b then t1 else t2
    - Přiřazení let x = t in u
    - case e of c1 ⇒ e1 | ... | cn ⇒ en se vyhodnotí jako ei, pokud e=ci

Základní struktury jazyka

- Formule, proměnné
  - Jsou výrazy jejich výsledná hodnota je bool
  - Základní operace s logickými výrazy
  - Kvantifikátory
    - ∀ x. P – obecný kvantifikátor
    - ∃ x. P – existenční kvantifikátor
    - ∃! x. P – existuje právě jedno x splňující P
  - Typové podmínky např. x < (y::nat)
- Proměnné
  - Volné, vázané, neznámé, ?x = ?x

Příklad 1 – seznam

```
theory ToyList
imports Datatype
begin
datatype 'a list = Nil
   ("[]")
                  | Cons 'a "'a list"     (infixr "#" 65)
primrec app :: "'a list => 'a list => 'a list"
                                  (infixr "@" 65)
where
"[] @ ys = ys" |
"(x # xs) @ ys = x # (xs @ ys)"
primrec rev :: "'a list => 'a list" where
"rev [] = []" |
"rev (x # xs) = (rev xs) @ (x # [])"
```

Příklad 1 – rozbor

- Theory – deklarace teorie
- Imports – teorie na nichž je založena
- Datatype – deklarace proměnných včetně konstruktorů
- Primrec – deklarace jednoduché rekurzivní funkce
  - primrec není primitivum, je konstrukt vytvořený ve vyšší rovině v jazyce ML (něco jako standardní funkce)

Cíl důkazu

- Dokazujme formuli:
  - theorem rev_rev [simp]: "rev(rev xs) = xs"
- Specifikujeme způsob dokazování
  - apply(induct_tac xs)
- A pokusíme se dokázat automaticky
  - apply(auto)
- Důkaz selže – je nutné doplnit lemma.

Důkaz

```
lemma app_Nil2 [simp]: "xs @ [] = xs"
apply(induct_tac xs)
apply(auto)
done
lemma app_assoc [simp]: "(xs @ ys) @ zs = xs @ (ys @ zs)"
apply(induct_tac xs)
apply(auto)
done
lemma rev_app [simp]: "rev(xs @ ys) = (rev ys) @ (rev xs)"
apply(induct_tac xs)
apply(auto)
done
theorem rev_rev [simp]: "rev(rev xs) = xs"
apply(induct_tac xs)
apply(auto)
done
end
```

Modelování protokolů

- Modelování agentů
  - `datatype agent = Server`
  - `| Friend nat`
  - `| Spy`
- Modelování klíčů
  - `types key = nat`
  - `consts invKey :: "key ⇒ key"`

Modelování protokolů

- Modelování zpráv
  - `datatype`
  - `msg = Agent agent`
  - `| Nonce nat`
  - `| Key key`
  - `| MPair msg msg`
  - `| Crypt key msg`
- Injektivita konstruktorů dává teorém
  - `Crypt K X = Crypt K′ X′ ⟹ K = K′ ∧ X = X′`

Induktivní množina

- Sudá čísla
  - `inductive set even :: "nat set" where`
  - `zero[intro!]: "0 ∈ even" |`
  - `step[intro!]: "n ∈ even ⟹ (Suc (Suc n)) ∈ even"`
- Dvě konstrukční pravidla
  - Zero: 0 je v množině
  - Step: Pokud n je v množině pak, je tam n+2.

Modelování útočníka

```
inductive set
  analz :: "msg set ⇒ msg set"
  for H :: "msg set"
  where
    Inj [intro,simp] : "X ∈ H ⟹ X ∈ analz H"
  | Fst:    "{|X,Y|} ∈ analz H ⟹ X ∈ analz H"
  | Snd:    "{|X,Y|} ∈ analz H ⟹ Y ∈ analz H"
  | Decrypt [dest]:
          " [[Crypt K X ∈ analz H; Key(invKey K) ∈ analz H
]]
          ⟹ X ∈ analz H"
```

Modelování útočníka

- Vlastnosti operace analz
- Je monotónní
  - $G \subseteq H \Longrightarrow \text{analz } G \subseteq \text{analz } H$
- Je idempotentní
  - $\text{analz } (\text{analz } H) = \text{analz } H$

Modelování útočníka

```
inductive set
  synth :: "msg set ⇒ msg set"
  for H :: "msg set"
  where
    Inj    [intro]: "X ∈ H ⟹ X ∈ synth H"
  | Agent  [intro]: "Agent agt ∈ synth H"
  | MPair  [intro]:
         "[[X∈ synth H;Y ∈ synth H]]⟹{|X,Y|}∈ synth H"
                                                    }
  | Crypt  [intro]:
          "[[X∈ synth H;Key K∈ H]]⟹Crypt K X ∈ synth H"
```

Modelování útočníka

- Vlastnosti operátoru synth
- Je monotónní a indempotentní
- Splňuje zajímavou rovnicí spolu se analz
  - analz (synth H) = analz H $\cup$ synth H

Modelování útočníka

- Operátor parts : parts H množina komponentů z prvků H
- Podobná definice jako analz se změnou konstruktoru Crypt
  - `Crypt K X ∈ parts H ⟹ X ∈ parts H`

Události

- Zprávy jsou modelovány událostí
  - Says A B X
- Stopa události je seznam zpráv
- Použité zprávy jsou modelovány
  - `used (Says A B X # evs)`
    `= parts {X} ∪ used evs`
- Funkce knows popisuje co se útočník naučí
  - `knows Spy (Says A B X # evs)`
    `= insert X (knows Spy evs)`

Události

- Vše co se útočník může naučit
  - analz (knows Spy evs)
- Vše co může vytvořit
  - synth (analz (knows Spy evs))

Kryptosystém s veřejným klíčem

- Funkce pubK mapuje agenty na jejich veřejné klíče
  - `consts pubK :: "agent ⇒ key"`
  - `abbreviation priK :: "agent ⇒ key"`
  - `where "priK x ≡ invKey(pubK x)"`
- Jsou zavedeny dva axiomy
  - `axioms`
  - `inj_pubK: "inj pubK"`
  - `priK_neq_pubK: "priK A <> pubK B"`

Needham-Schroeder + Lowe

- Needham-Schroeder protokol s veřejným klíčem s modifikací proti Lowe útoku
- 1. $A \rightarrow B : \{Na, A\}_{Kb}$
- 2. $B \rightarrow A : \{Na, Nb, B\}_{Ka}$
- 3. $A \rightarrow B : \{Nb\}_{Kb}$

```
inductive_set ns_public :: "event list set"
  where

    Nil:  "[] ∈ ns_public"

  | Fake: "⟦evsf ∈ ns_public;  X ∈ synth (analz (knows Spy evsf))⟧
          ⟹ Says Spy B X  # evsf ∈ ns_public"

  | NS1:  "⟦evs1 ∈ ns_public;  Nonce NA ∉ used evs1⟧
          ⟹ Says A B (Crypt (pubK B) {|Nonce NA, Agent A|})
                # evs1  ∈  ns_public"

  | NS2:  "⟦evs2 ∈ ns_public;  Nonce NB ∉ used evs2;
           Says A' B (Crypt (pubK B) {|Nonce NA, Agent A|}) ∈ set evs2⟧
          ⟹ Says B A (Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|})
               # evs2  ∈  ns_public"

  | NS3:  "⟦evs3 ∈ ns_public;
           Says A  B (Crypt (pubK B) {|Nonce NA, Agent A|}) ∈ set evs3;
           Says B' A (Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|})
              ∈ set evs3⟧
          ⟹ Says A B (Crypt (pubK B) (Nonce NB)) # evs3 ∈ ns_public"
```

Poznámka k modelu

- Sémantický model je teorie množin.
- Důkazy jsou prováděny indukci v této teorii
- Informační notace protokolu je převáděna do indukčních pravidel.
- Bezpečnostní cíle jsou formulovány v teorii množin
  - `X \<notin> analz (knows Spy evs)`

Bezpečnostní teorém

```
theorem Spy_not_see_NB [dest]:
 " [[Says B A (Crypt (pubK A)
      {Nonce NA, Nonce NB, Agent B} ) ∈ set evs;
     A \<notin> bad; B \<notin> bad;
         evs ∈ ns_public ]]
  ⟹ Nonce NB ∈ analz (knows Spy evs)"
```