Modellazione e Analisi di Sistemi

Indice

Capitolo 1

Abstract State Machines

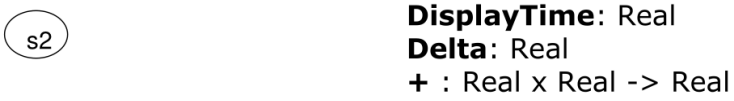
Le ASM sono delle FSM (*Final State Machines*) con stati generalizzati; rappresentano la forma matematica di macchine che estendono la nozione di FSM, **ampliando la definzione di stato** e **modificando la forma delle transizioni**.

Stati

Gli stati di controllo non strutturati vengono sostituiti da stati (strutturati) che modellano:

- **dati** complessi arbitrati (con domini di base e funzioni per la struttura)
- **operazioni** per la manipolazione di dati

Possiamo definire gli stati come delle *algebre*.



Transizioni

Le transizione sono *"regole"* che descrivono il cambiamento di funzioni da uno stato al successivo; permettono di modificare la struttura algebrica durante l'esecuzione della ASM.

if condition then Updates

Negli FSM le transizioni sono rappresentate con delle frecce.

Le ASM sono dotate di un ambiente di tool per:

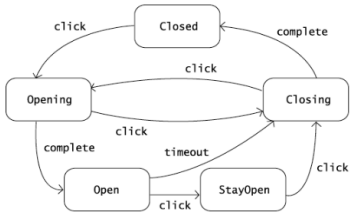
- editing
- simulazione
- validazione
- verifica
- generazioni di casi di test

Un modello ASM può essere visto come pseudocodice su strutture dati astratte.

Da FSM a ASM

Domini:
Stati: insieme degli stati

Funzioni:
ctl_state: Stati
click: boolean
complete: boolean
timeout: boolean



Regole di transizione:

```
if ctl_state = Opering and click
then
   ctl_state:= Closing

if ctl_state = Closing and click
then
   ctl_state:= Opening

if ctl_state = Closing and complete
then
   ctl_state:= Closed
.....
```

Inizializzazione:

State = {Opening, Closing, Open, ….}
ctl_state = Open

Possiamo definire ASM = (header, body, main rule, inizialization)

Domini:
State: insieme degli stati

Funzioni:
ctl_State: State
input: String
output: String

Regole di transizione:

```
r_s1_1 = if ctl_State = s1 and
                input = "1"
           then
                ctl_State:= "s1"
                output:= "0"
r_s1_0 =
r_s2_1 = ...
r_s2_0 = ...
```

main rule

initialization

```
asm FSM
import StandardLibrary
signature:
   controlled ctl_State: String
   monitored input: String
   out output: String
```

header

```
definitions:
 rule r_s1_1 = if ctl_ = "s1" and input = "1" then
                    par ctl_State:= "s1", output:= "0" endpar
               endif
rule r_s1_0 = if ctl_State = "s1" and input = "0" then ...
rule r_s2_1 = ...
rule r_s2_0 = ...

main rule r_Main = par r_s1_1, r_s1_0, r_s2_1, r_s2_0 endpar

default init so:
     function currentState = "s1"
```

body

1.1 Formalismo

1.1.1 Vocabolario

DEF: Un **vocabolario** Σ è una collezione finita di nomi di funzioni.
Le funzioni possono essere dinamiche o statiche, a seconda che l'interpretazione del nome della funzione cambia o no da uno stato al successivo (funzioni in senso matematico).

1.1.2 Costanti

Le funzioni statiche di arietà zero sono dette **costanti**. Ogni vocabolario contiene sempre le costanti *undef, true, false.*
Ad esempio:

- i numeri sono costanti numeriche
- `voto = 30`

1.1.3 Funzioni statiche

Le funzioni statiche (arietà > 0) sono definite tramite una legge fissa.
Ad esempio:

- operazioni tra numeri (+, -, ...)
- operazioni tra booleani (AND, OR, ...)
- `max(m, n)`

1.1.4 Fuzioni dinamiche

Le funzioni dinamiche di arietà zero sono le variabili dei linguaggi di programmazione.

1.1.5 Stato ASM

DEF: Fissato un vocabolario Σ, uno **stato** A del vocabolario Σ è un insieme non vuoto X, detto *superuniverso di* A, con le interpretazioni dei nomi delle funzioni di Σ.
Da questa definizione, segue che:

- se f è un nome di funzione *n-aria* di Σ, allora la sua interpretazione f^A è una funzione da X^n a X
- Se c è un nome di costante di Σ, allora la sua interpretazione c^A è un elemento di X

Possiamo definire il superuniverso come un *"dominio di interpretazione"*; i simboli del vocabolario, presi singoralmente, sono soltanto simboli.

1.1.6 Domini ASM

Il superuniverso di uno stato ASM è suddiviso in *universi*, rappresentati dalle loro funzioni caratteristiche.

Se A è un sottoinsieme dell'insieme X, la funzione caratteristica di A è quella funzione da X all'insieme $\{0,1\}$ che sull'elemento $x \in X$ vale 1 se x appartiene ad A, e vale 0 in caso contrario.

Ogni universo rappresenta un dominio. In base a questa rappresentazione degli insiemi in termini di funzioni caratteristiche, uno stato di una ASM consente di modellare **domini eterogenei**.
Alcuni esempi di domini:

- predefiniti, come `Interi, String, ...`
- definiti dall'utente, come tipi astratti o a partire da altri domini

Esempio

Dominio $X = \{1, 2, a, b, mario, pippo\}$, ripartito in domini:

- $Interi = \{1.2\}$
- $Char = \{a, b\}$
- $String = \{mario, pippo\}$

1.1.7 Termini ASM

DEF: i termini di Σ sono espressioni sintattiche così costruite:

1. Variabili $v_0, v_1, v_2, \ldots$ sono termini
2. Costanti c di Σ sono termini
3. Se f è un nome di funzione *n-aria* di Σ e $t_1, \ldots, t_n$ sono termini
   $\Rightarrow f(t1, \ldots, t_n)$ è un termine

Ad esempio:

- $v_0 + v_1$
- $1 + (v_2 * 0)$

Un termine che non contiene variabili è detto chiuso. I termini sono *oggetti sintattici*. **Assumono significato (o semantica) nello stato**; il suo valore è l'*interpretazione del termine* in A.

1.2 Regole di transizione ASM

Aggiornare stati astratti significa cambiare interpretazione delle (o solo di alcune) funzioni della segnatura della macchina. Il modo in cui la macchina aggiorna il proprio stato è descritto da **regole di transizione**; l'insieme delle regole di transizione definiscono la sintassi di un *programma ASM.* e ne determinano la computazione.
DEF: Sia Σ un vocabolario; le regole di transizione di una ASM sono *espressioni sintattiche su* Σ *generate* attraverso l'uso di **costruttori di regole**.

1.2.1 Update rule

$f(t_1, \ldots, t_n) := s$, dove:

- f è un nome di funzione **dinamica** *n-aria* di Σ
- $(t_1, \ldots, t_n)$ e s sono termini di Σ

Significa che nello stato successivo, il valore di f per gli argomenti $(t_1, \ldots, t_n)$ è aggiornato ad s. Nel caso di una funzione 0-aria, cioè una variabile x, l'aggiornamento ha la forma $x := s$.

Locazione ASM

Una locazione è definita matematicamente come una coppia $(f, (v_1, \ldots, v_n))$ con f nome di funzione e $(v_1, \ldots, v_2)$ i suoi argomenti.
In uno stato S, una locazione ha un *valore*: l'interpretazione del della funzione nello stato. Una funzione può essere:

- Variabile (0-arie): x, y, ...
- Funzioni (n-arie): f(1), name(2), ...

Aggiornamento di locazione

Le transizioni di stato delle ASM, tramite update rule, causano **aggiornamenti** dei valori contenuti nelle locazioni; significa che cambia l'interpretazione della funzione dinamica rispetto ai valori.
Se $loc = (f, (v_1, \ldots, v_n))$ e $loc = a$, l'aggiornamento di loc ha la forma:
$f(v_1, \ldots, v_2) := newval$

1.2.2 Costruttore if-then-else

`if` ϕ `then R else S`, ovvero: *se* ϕ *è vera, allora esegui* R*, altrimenti esegui* S.

Esempio

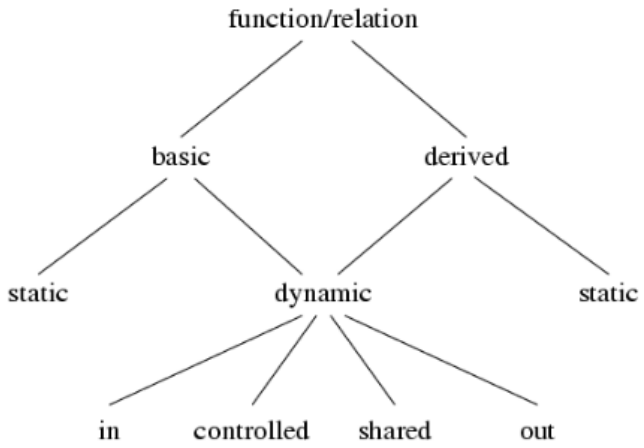
if CurrTime = DisplayTime + Delta
then DisplayTime := CurrTime

con DisplayTime, CurrTime, e Delta variabili (funzione 0-arie)
reali

1.2.3 Classificazione delle funzioni

Sia M una ASM e *env* l'ambiente di M:

- **Dynamic**
  - *monitored:* lette (non aggiornate) da M, scritte da *env*
    `CurrTime` viene incrementato dall'esterno
  - *out:* scritte (non lette) da M, lette da *env*
  - *controlled:* lette e scritte da M
    `DisplayTime` viene incrementato dalla regola
  - *shared:* lette e scritte da M e *env*
- **Derived:** valori computati da funzioni monitorate e funzioni statiche per mezzo di una "legge" fissata a priori



1.2.4 Costruttore skip

- *Skip Rule:*

  skip

  Significato: non fare niente

  if x>0 **then** x:=x+1

  else skip

1.2.5 Costruttore let

Serve per abbreviare ad esempio un termine molto complesso o lungo.

- *Let Rule (se R ha parametri, realizza il passaggio per valore):*

  let *x* = *t* **in** *R(x)*

  Significato: Assegna il valore di *t* a *x* ed esegui *R*

  let x = f1(f2(f3(y))) **in**

  g(x):= 7

1.2.6 Costruttore "par" per block rule

Block Rule (composizione parallela):

par
R
S
endpar

Modella **synchronous bounded parallelism**
- **bounded parallelism** = simultaneous application of different functions

Significato: *R* e *S sono eseguite* in parallelo

1.2.7 Costruttore seq

- *Seq Rule (composizione sequenziale):*

  seq *R S* ***endseq***

  Significato: *R* e *S sono eseguite* in sequenza (gli update causati in R sono già visibili in S; lo stato tra R ed S non è visibile)

Esempio: due aggiornamenti in sequenza dello stesso valore della funzione, ipotesi f(5)=1

```
seq
  f(5) := 2
  f(3) := f(5)   -- allo stato successivo f(3)=2
```

con f:Integer->Integer

1.2.8 Costruttore forall

- *Forall Rule:*

  forall *x* **with** φ **do** *R[x]*

  Significato: Esegui *R* in parallelo per ogni *x* che soddisfa φ

  Implementa il concetto di parallelismo sincrono (*potentially unbounded*)

1.2.9 Costruttore chooose

- *Choose Rule:*

  choose *x* **with** φ **do** *R[x]*

  Significato: Esegui *R* per un *x scelto random* che soddisfi φ

  Implementa il concetto di non-determinismo

Non deterministico perché scelgo una x a caso tra quelle che soddisfano la condizione.

1.2.10 Tutti i costruttori di regole

Nome	Significato	Sintassi
Skip rule	Non fa nulla	**skip**
Update rule	Aggiorna il valore di f per gli argomenti $(s_1,...,s_n)$ a t Nota: f è un nome di funzione dinamica	$f(s_1,...,s_n) := t$
Block rule	P e Q sono eseguite in parallelo	**par** P Q **endpar**
Sequence rule	P e Q sono eseguite in sequenza	**seq** P Q **endseq**
Conditional rule	Se φ è vera, esegui P, altrimenti Q	**if** φ **then** P **else** Q
Let rule	Assegna il valore di t a x e quindi esegui P	**let** $x = t$ **in** P
Forall rule	Esegui P in parallelo per ogni x che soddisfa φ	**forall** x **with** φ **do** P
Choose rule	Scegli un x che soddisfa φ e quindi esegui P	**choose** x **with** φ **do** P
MacroCall rule	Esegui la regola di transizione r con parametri $t_1,...,t_n$	$r(t_1,...,t_n)$

1.3 Aggiornamenti consistenti

A causa del parallelismo, una regola di transizione può richiedere **più volte l'aggiornamento di una stessa funzione** per gli stessi argomenti. Si richiede che tali aggiornamenti siano **consistenti**.
Data la funzione $f(a_1, \ldots, a_n)$ in uno stato S_i della macchina:

- La coppia $loc = (f, (a_1, \ldots, a_n))$ è detta *locazione* e rappresenta matematicamente il valore di $f(a_1, \ldots, a_n)$ in memoria
- Un *aggiornamento* è la coppia $(loc, b) = ((f, (a_1, \ldots, a_n)), b)$

  $\rightarrow$ significa che l'interpretazione della funzione f in S_i viene modificata per gli argomenti $a_1, \ldots, a_n$ con il valore b in S_{i+1}
- Un **update set** è un insieme di aggiornamnti

DEF: Un update set U è **consistente** se vale:
$\forall$ locazione $(f, (a_1, \ldots, a_n))$, se:

- $(f, (a_1, \ldots, a_n)) \in U$
- $(f, (a_1, \ldots, a_n)) \in U$
- $\Rightarrow b = c$

Se $b \neq c$, allora l'update si dice **inconsistente** (stiamo scrivendo due valori diversi nella stessa locazione).

- Se l'update set è consistente, allora i suoi aggiornamenti possono essere eseguiti in un dato stato; il risultato è un nuovo stato dove le *interpretazioni* dei nomi di *funzioni dinamiche* sono cambiati secondo U
- Le *interpretazioni* dei nomi delle*funzoni statiche* sono gli stessi dello stato precedente
- le *interpretazioni* dei nomi delle *funzioni monitorate* sono date dall'ambiente esterno e possono dunque cambiare in modo arbitrario

1.4 Rule constructor

(Macro) Call Rule:

$r\ [t_1, \ldots, t_n]$

Significato: Chiama *r (regola con parametri) con* argomenti $t1, \ldots, tn$

Una *definizione di regola per un nome di regola r* di arietà *n è un'espressione*

$$r\ (x_1, \ldots, x_n) = R$$

dove R è una regola di transizione.

In una call rule $r\ [t_1, \ldots, t_n]$ le variabili x_i che occorrono nel corpo R della definizione di r vengono sostituite dai parametri t_i (*modularità*)

1.5 Riassumendo...

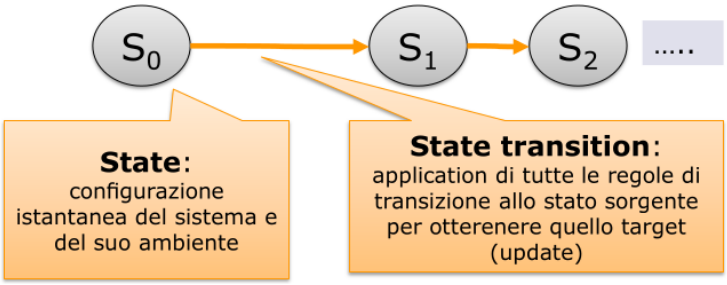
Una ASM è una terna (Σ, A, R), con:

- un vocabolario Σ
- uno stato iniziale A per Σ
- un insieme R di nomi di regole, con:
  - un nome di regola di arietà zero, il *main* (l'*entry point* per l'esecuzione della macchina)
  - una definizione di regola per ogni nome di regola

La semantica delle regole di transizione è data dall'insieme degli aggiornamenti.

1.6 Esecuzione

La **run** (o **esecuzione**) di una ASM è definita come:
Una sequenza (finita o infinita) $S_0, S_1, \ldots, S_n$ di stati di M, dove S_0 è lo stato iniziale e ciascun stato S_{n+1} è ottenuto dallo stato precedente S_n eseguendo simultaneamente tutte le regole di transizione che sono eseguibili in S_n.



1.7 La riserva di una ASM

Capitolo 2

Prime tecniche di analisi

2.1 Analisi di un modello

Per l'analisi di un modello si usano due approcci:

- **Validazione:** necessaria per controllare che il sistema soddisfi i requisiti richiesti
- **Verifica:** necessaria per garantire proprietà (safety, leaveness, assenza deadlock, reachability, ecc.)

La validazione dovrebbe precedere la verifica per individuare errori il prima possibile e evitare di provare proprietà corrette su specifiche incorrette.
Sono possibili due prime forme di analisi sul ground model:

- Garanzia degli invarianti
- Validazione tramite scenari

2.1.1 Uso degli invarianti

- In un modello ASM gli invarianti sono usati per **esprimere vincoli** su funzioni e/o regole che devono essere garantiti in ogni stato
- In programmi AsmetaL usare gli invarianti è utile per **scoprire errori di modellazione**
  - In generale, l'assenza di violazione di invarianti non può essere considerata una prova della correttezza del modello, mentre la violazione di assiomi, è prova della incorrettezza del modello.

Dichiarazione di invarianti

Gli invarianti vanno dichiarati subito prima della *main rule*.
Ogni invariante è dichiarato mediante la keyword *invariant* che precede il nome attribuito all'assioma (invName).
Le funzioni e regole su cui è espresso il vincolo vanno listate dopo la keyword *over*.

```
invariant invName over id_function,...,id_rule:term
```

Esempio

Esempio

```
asm axiom_example
import ../STDL/StandardLibrary
signature:
    dynamic controlled fooA: Integer
    dynamic controlled fooB: Integer
    dynamic monitored monA: Boolean
    dynamic monitored monB: Boolean
definitions:
    macro rule r_a =
      if(monA) then  fooA := fooA + 1
      endif
    macro rule r_b =
      if(monB) then  fooB := fooB + 1
      endif

invariant over fooA, fooB: fooA!=fooB
```

```
main rule r_main =
   par
        r_a[]
        r_b[]
   endpar

default init s0:
   function fooA = 1
   function fooB = 0
```

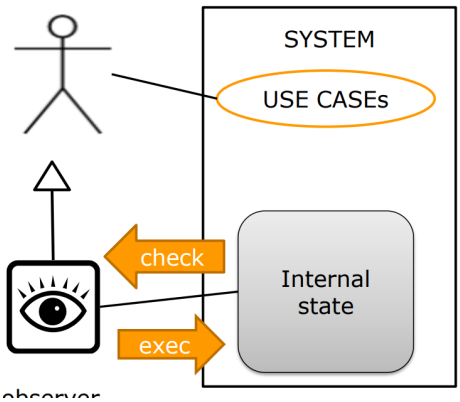
2.1.2 Validazione tramite scenari

- **Tecniche:** Generazione di scenari, sviluppo di prototipi, animazione, simulazione, testing
- **Scenario:** descrizione di un possibile comportamento del sistema (interazione osservabile tra il sistema ed il suo ambiente in specifiche situazioni)

Gli scenari sono costituiti attraverso una notazione testuale (Avalla) Semantica chiara (definita in termini di ASM) e capacità di descrivere anche dettagli interni (non solo black box come per UML use cases, ma anche informazioni sullo stato).

Da attore UML a attore ASM

Nell'UML use case l'attore interagisce col sistema, uno o più scenari possono essere generati per ogni caso d'uso, però visione BLACK BOX. L'ASM Observer può verificare lo stato interno della macchina e gli invarianti, e richiede l'esecuzione di regole arbitrarie.



Doppio uso degli scenari

- Due tipi di attori esterni:
- – **User:** ha una visione *black box* del sistema
  – **Observer:** ha una visione *grey box*
  – Due obiettivi per gli scenari:
  – ∗ **Validazione classica:** azione dell'utente e reazioni della macchina
    ∗ **Attività di testing:** ispezione dell'observer dello stato interno della macchina

Scenario ASM

Sequenza di interazione consentite delle azioni:

- da parte di user/observer:
  – **set** the environment (i.e. the values of monitored/shared functions)
  – **check** for the machine outputs (i.e. the values of out functions)
  – **check** the machine state and invariants
  – **ask** for the execution of given transition rules
- da parte della macchina: **makes** one *step* as reaction of the actor actions

Primitive di AvValLa - Asm Validation Language

Set	A command to set the location of a (monitored) function to a specific value: it simulates the **environment**
Check	To inspect external values and (only for the observer) to inspect internal values in the current state
Step	To signal that the environment has finished to update the monitored locations, hence the machine can perform a step
StepUntil	To signal that the machine can perform a step iteratively until a specified condition becomes true
Invariant	To state critical specification properties that should always hold for a scenario
Exec	To execute transition rule when required by the observer

Sintassi Avalla

Abstract syntax	Concrete syntax
Scenario	**scenario** name **load** spec_name Invariant* Command*
spec_name is the spec to load; invariants and commands are the script content	
Invariant	**invariant** name ':' expr ';'
expr is a boolean term made of function and domain symbols of the underlying ASM	
Command	(Set | Exec | Step | StepUntil | Check)
Set	**set** loc := value ';'
loc is a location term for a monitored function, and value is a term denoting a possible value for the underlying location	
Exec	**exec** rule ';'
rule is an ASM rule (e.g. a choose/forall rule, a conditional if, a macro call rule, ect.)	
Step	**step**
StepUntil	**step until** cond ';'
cond is a boolean-valued term made of function and domain symbols of the ASM	
Check	**check** expr ';'
expr is a boolean-valued term made of function and domain symbols of the ASM	

Capitolo 3

Composizione di modelli: ASM multi agenti

Grazie alle regole per la strutturazione, è possibile definire un sistema distribuito come una composizione di ASM.

- Ogni componente del sistema distribuito viene modellato mediante una opportuna ASM
- Risulta un sistema di ASM Distribuite, dette anche **ASM multi agenti**

3.1 ASM multi agenti

Descrivono un modello distribuito della computazione. La computazione distribuita è modellata mediante un insieme di Agenti che operano in modo concorrente

- *Movimenti* concorrenti sincroni/asincroni
- *Stati globali* condivisi tra gli agenti

Gli agenti ASM:

- Possono essere creati dinamicamente
- Hanno una visione parziale dello stato globale
- Hanno il proprio programma da eseguire

Le ASM multi agenti sono classificate in:

- **Sincrone:** gli agenti eseguono il loro programma in parallelo, sincronizzati su un implicito clock globale del sistema
- **Asincrone:** gli agenti eseguono il loro programma in parallelo, ma in modo indipendente tra loro

- Ciascuno ha il *proprio clock* che regola la durata du una mossa
- Ciascuno opera nel *proprio stato locale*

3.1.1 Definizione ASM multiagente:

Una sync/async ASM è un insieme di coppie (a, ASM(a)) dove

- *di agenti a ∈ Agent (il dominio degli agenti)*
- *e programmi ASM(a) che sono ASM di base*

3.1.2 Codifica AsmetaL di ASM Multiagenti

$$[asyncr]asmASM - name$$

a parola chiave asyncr è opzionale(perchè racchiusa nelle parentesi quadre)

- Se presente, denota una ASM *asincrona* multi-agent
- Se omessa, l'ASM è considerata *sincrona* multi-agent
- Ogni agente ha una *visione parziale* dello stato globale
  - *View(a, M)* indica la vista dell'agente a dello stato globale di una ASM M
- Ogni agente può avere una *visione privata* non condivisa con altri agenti
  - per una $f : A- > B$ globale, la dichiarazione privata di f diventa $f : Agent$ x $A- > B$
    - f(self,x) denota la *verisone privata di f(x)* dell'agente corrente *self*
  - *self* viene interpretata da *a* (agente) come se stesso

Header di ASM multiagente

In caso di ASM multi-agente è utile definire l' **heade**
come:

[**import** m_1 [**(** id_{11},...,id_{1h1} **)**]
...
import m_k [**(** id_{k1},...,id_{khk} **)**]
]
[**export** id_1,...,id_e] or [**export ***]
signature :
[dom_declarations]
[fun_declarations]

• import/export di simboli (id) di domini, funzioni (e loro domini e codomini), e regole da/verso altre ASM m_i
• **export *** per esportare tutto
•Ricordare: la segnatura contiene dichiarazioni (non definizioni) di domini e funzioni!

3.1.3 Computazione di ASM multiagenti

ASM sincrone: computazione

Tutte le ASM lavorano in parallelo:

- L'insieme opera negli stati globali, ottenuti dall'unione di tutti gli stati delle ASM componenti
- La sequenza di eventi che determina un'esecuzione è la sequenza degli stati che rappresentano l'esecuzione della sync-ASM

una **multi-agent ASM con agenti sincroni** ha una *quasi-sequential run*

- una sequenza di stati S_0, S_1,... S_n,... dove ciascun stato S_i è ottenuto dal precedente S_{i-1} eseguendo in parallelo le regole di tutti gli agenti

ASM asincrone: computazione

Nelle async-ASM, gli agenti possono eseguire con clock diversi ed i loro step possono avere durate diverse

- Non c'è il concetto di controllo globale del sistema
- Possiamo dare delle *proprietà delle run*
- Difficoltà di gestione della consistenza

Una *multi-agent ASM con agenti asincroni* ha una run parzialmente ordinata, ossia un insieme parzialmente ordinato (M,¡) di mosse m

leggi: mossa = applicazioni di regole; m1 < m2 se m1 è eseguita prima di m2

dei suoi agenti che soddisfano le condizioni:

- **storia finita**: ad un certo istante t la sequenza di mosse che ha condotto dallo stato S_0 allo stato S_t è finita (ciascuna mossa ha solo un numero finito di predecessori)
  - per ogni $m \in M$ l'insieme $\{m' | m' < m\}$ è finito
- **sequenzialità degli agenti**: ogni agente opera in modo sequenziale
  - l'insieme di mosse $\{m | m \in M$, a esegue $m\}$ di ogni agent $a \in$ Agent è linearmente ordinato per $<$
- **coerenza: ogni stato di M è il risultato delle mosse di agenti**:
  - Sia X un segmento iniziale finito (sottinsieme chiuso a sinistra) di (M,<)
  - Sia σ(X) lo stato associato ad X: σ(X) è il risultato di tutte le mosse m in X
  - Ciascun segmento iniziale finito X di (M,¡) ha uno stato associato σ(X) che è il risultato dell'applicazione della mossa m nello stato $\sigma(\text{X} - \{m\})$, per ogni elemento massimale $m \in$ X

Capitolo 4

Specifica di protocolli di sicurezza: il protocollo di Needham Schroeder

4.1 Crittografia: concetti base

- **Algoritmi crittografici:** algoritmi che trasformano un testo (o una info) in un testo cifrato utilizzando chiavi
- **Protocolli crittografici:** sequenze di scambio di messaggi a cui gli agenti possono applicare algoritmi crittografici
  - **a chiave simmetrica:** la stessa K è utilizzata per criptare e decriptare
  - **a chiave pubblica:** K-pub per criptare e K-priv per decriptare
  - **a chiave condivisa:** una K-AB tra 2 agenti fornita da un terzo

Protocolli per l'autenticazione

Lo scopo è **l'autenticazione**

- fornire ad una parte la certezza dell'identità del mittente del messaggio ricevuto
- eventualmente stabilire una chiave (o altra info) comune

Segretezza: in più possono garantire che l'informazione (o parte) scambiata è rimasta segreta

Assunzioni di base

- **Forza della spia:** la spia (man in the middle) può
  - intercettare (ed eventualmente analizzare) ogni messaggio che sia stato spedito da un agente
  - formare (sintetizzare) nuovi messaggi utilizzando la sua base di conoscenze
  - cambiare destinatario di un messaggio
- **Forza della crittografia:**
  - un messaggio può essere decriptato solo se si ha la chiave giusta
  - un agente può generare sempre nonces nuove
  - un agente autenticato è non compromesso se la sua chiave privata non è nota alla spia

4.2 Needham-Schroeder a chiavi pubbliche

Obiettivo: stabilire mutua autenticazione tra due agenti A e B (e una nonce segreta comune)

Protocollo

1. **A** che vuole iniziare la comunicazione con **B**, manda una nuova nonce a **B**:

$$A->B : \{\text{Na, A}\}\text{Kb}$$

2. **B** legge il messaggio, rimanda la nonce di **A** più un'altra nuova:

$$B->A : \{\text{Na, Nb}\}\text{Ka}$$

3. **A** legge il messaggio, riconosce la nonce rispedita da **B** (solo B può averla letta) e rimanda a **B**, Nb:

$$A->B : \{\text{Nb}\}\text{Kb}$$

4. **B** ricevendo indietro la nonce Nb sa che l'autenticazione (handshake) è avvenuta (solo **A** può averla letta)

Vulnerabilità

Il protocollo è suscettibile ad attacchi di tipo *man in the middle*:

- un impostore Spy riesce ad avere il controllo del traffico (modellato con Alice che inizia una sessione di comunicazione con Spy)
- inoltra i messaggi a Bob convincendolo di essere in comunicazione con Alice ed inganna Alice facendole credere di comunicare con Bob

Spy agisce da "uomo nel mezzo" e intercetta tutti i messaggi di Bob, che crede di essere in comunicazione sicura con Alice
L'attacco è stato descritto per la prima volta da Gavin Lowe nel 1995

4.2.1 Attacco di Lowe

A inizia una istanza del protocollo con la spia (o con un agente compromesso)

$$A->Spy : \{\text{Na, A}\}\text{Ks}$$
$$Spy->B : \{\text{Na, A}\}\text{Kb}$$
$$B->A : \{\text{Na, Nb}\}\text{Ks}$$
$$Spy->B : \{\text{Nb}\}\text{Kb}$$

La spia acquisisce una informazione segreta di B: **Nb è compromessa**
A sta usando una nonce compromessa
B ha una nonce Nb compromessa, eppure né A né B sono compromessi
Per garantire l'autenticazione il protocollo richiederebbe

- che gli agenti non siano compromessi ma
- anche che non abbiano iniziato comunicazioni con agenti compromessi

La soluzione di Lowe

1. **A** che vuole iniziare la comunicazione con **B**, manda una nuova nonce a **B**:

$$A->B : \{\text{Na, A}\}\text{Kb}$$

2. **B** legge il messaggio, rimanda la nonce di **A** più un'altra nuova e **la sua identità**:

$$B->A : \{\text{Na, Nb, B}\}\text{Ka}$$

3. **A** legge il messaggio, riconosce la nonce rispedita da **B** (solo B può averla letta) e rimanda a **B**, Nb:

$$A->B : \{\text{Nb}\}\text{Kb}$$

4. **B** ricevendo indietro la nonce Nb sa che l'autenticazione è avvenuta (solo **A** può averla letta)

Esercizio NS in ASM