

Modellazione e Analisi di Sistemi

Indice

1	Abstract State Machines	2
1.1	Formalismo	4
1.1.1	Vocabolario	4
1.1.2	Costanti	4
1.1.3	Funzioni statiche	4
1.1.4	Fuzioni dinamiche	4
1.1.5	Stato ASM	4
1.1.6	Domini ASM	5
1.1.7	Termini ASM	5
1.2	Regole di transizione ASM	6
1.2.1	Update rule	6
1.2.2	Costruttore if-then-else	7
1.2.3	Classificazione delle funzioni	7
1.2.4	Costruttore skip	8
1.2.5	Costruttore let	8
1.2.6	Costruttore "par" per block rule	8
1.2.7	Costruttore seq	8
1.2.8	Costruttore forall	9
1.2.9	Costruttore chooose	9
1.2.10	Tutti i costruttori di regole	10
1.3	Aggiornamenti consistenti	10
1.4	Rule constructor	11
1.5	Riassumendo...	11
1.6	Esecuzione	12
1.7	La riserva di una ASM	12
2	Prime tecniche di analisi	13
2.1	Analisi di un modello	13
2.1.1	Garanzia degli invarianti	13
2.1.2	Validazione tramite scenari	14

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 definizione 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*.



CurrTime: Real
DisplayTime: Real
Delta: Real
+ : Real x Real -> Real

Transizioni

Le transizioni 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

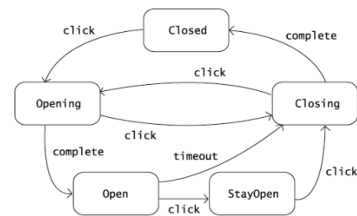
Funzioni:

ctl_state: Stati

click: boolean

complete: boolean

timeout: boolean



Regole di transizione:

if ctl_state = Opening 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, initialization)

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 := "o"

r_s1_0 =

r_s2_1 = ...

r_s2_0 = ...

main rule

initialization

asm FSM
import StandardLibrary

header

signature:
 controlled ctl_State: String
 monitored input: String
 out output: String

definitions:

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

body

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

default init so:

function currentState = "s1"

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

- $Interi = \{1, 2\}$
- $Char = \{a, b\}$
- $String = \{\text{mario}, \text{pippo}\}$

1.1.7 Termini ASM

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

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

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

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

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

Localizzazione ASM

Una localizzazione è definita matematicamente come una coppia $(f, (v_1, \dots, v_n))$ con f nome di funzione e (v_1, \dots, v_n) i suoi argomenti.

In uno stato S , una localizzazione ha un *valore*: l'interpretazione della funzione nello stato. Una funzione può essere:

- Variabile (0-aria): x, y, \dots
- Funzioni (n-arie): $f(1), \text{name}(2), \dots$

Aggiornamento di localizzazione

Le transizioni di stato delle ASM, tramite update rule, causano **aggiornamenti** dei valori contenuti nelle localizzazioni; significa che cambia l'interpretazione della funzione dinamica rispetto ai valori.

Se $loc = (f, (v_1, \dots, v_n))$ e $loc = a$, l'aggiornamento di loc ha la forma:

$$f(v_1, \dots, v_n) := newval$$

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

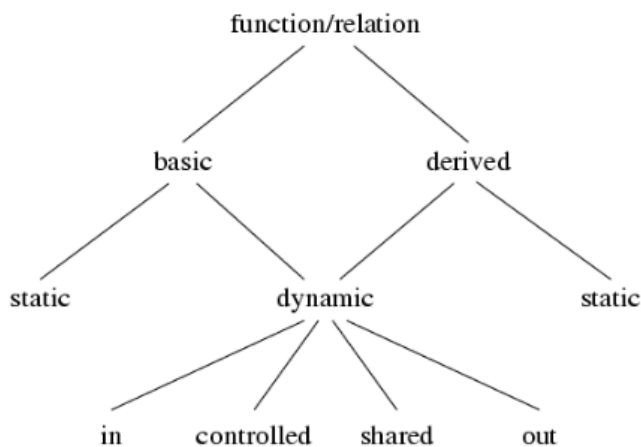
**if CurrTime = DisplayTime + Delta
then DisplayTime := CurrTime**

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 $x = f_1(f_2(f_3(y)))$ **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 $f: \text{Integer} \rightarrow \text{Integer}$

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 choose

- *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 $f: \text{Integer} \rightarrow \text{Integer}$

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

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 $f(a_1, \dots, a_n)$ in uno stato S_i della macchina:

- La coppia $loc = (f, (a_1, \dots, a_n))$ è detta *locazione* e rappresenta matematicamente il valore di $f(a_1, \dots, a_n)$ in memoria
- Un *aggiornamento* è la coppia $(loc, b) = ((f, (a_1, \dots, a_n)), b)$
→ significa che l'interpretazione della funzione f in S_i viene modificata per gli argomenti a_1, \dots, a_n con il valore b in S_{i+1}
- Un **update set** è un insieme di aggiornamenti

DEF: Un update set U è **consistente** se vale:

\forall locazione $(f, (a_1, \dots, a_n))$, se:

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

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 *funzioni 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:

$r[t_1, \dots, t_n]$

Significato: Chiama r (regola con parametri) con argomenti t_1, \dots, t_n

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

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

dove R è una regola di transizione.

In una call rule $r[t_1, \dots, t_n]$ 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:

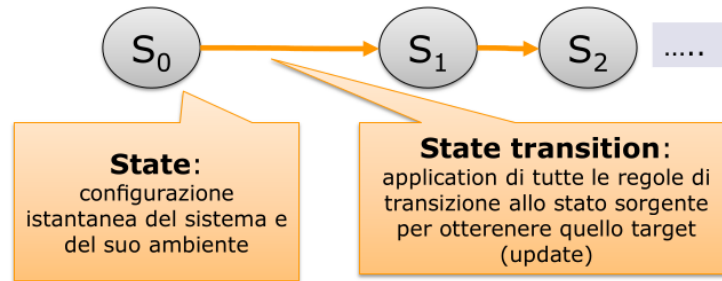
- un vocabolario Σ
- uno stato iniziale A per Σ
- un insieme R di nomi di regole, con:
 - un nome di regola di arietà zero, il *main* (l'*entry point* per l'esecuzione della macchina)
 - una definizione di regola per ogni nome di regola

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, \dots, 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 Garanzia 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 degli 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

- Le funzioni e regole su cui è espresso il vincolo vanno listate dopo la keyword *over*

```
asm axiom_example
import ../STDLib/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
```

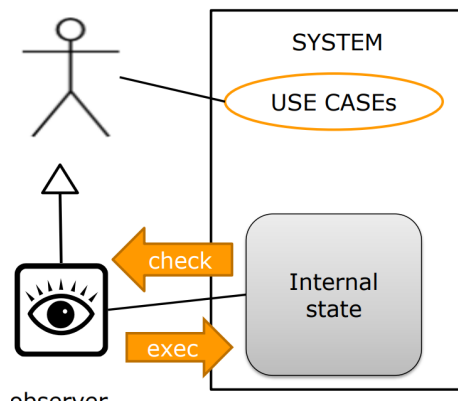
2.1.2 Validazione tramite scenari

- **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 (Avala) 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	