Capitolo 1

Teoria

1. **Definizione FSM**:

   Una FSM è una tripla (S, I, δ), dove S è l'insieme degli stati, I l'insieme finito degli input e $\delta : S \times I \to S$ funzione di transizione

2. **Dare uno schema ASM per una FSM**:

   Una FSM può essere descritta come una FSM tramite il seguente schema: Dove:

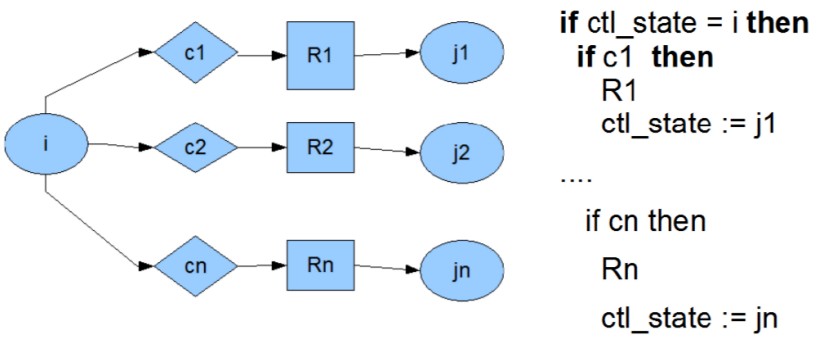


Figura 1.1: FSM in ASM

   - ctl_state: variabile che rappresenta lo stato corrente
   - R_k: azioni della macchina
   - c_k: condizioni di input
   - $i, j_1, \ldots, j_n$: stati interni di controllo, ovvero i valori assunti da ctl_state

   Le ASM sono analoghe alle FSM con differenze che riguardano gli stati, che nelle FSM è dall'unico stato ctl_state, e dalle condizioni di input e azioni di output che nelle FSM è un alfabeto finito e nelle ASM sono infinite.

3. **Stato ASM**:

   Fissato un vocabolario Σ contenente nomi di funzioni, uno stato A del vocabolario è un insieme non vuoto X, chiamato superuniverso di A, contenente

le interpretazioni dei nomi di funzioni. Se f è un nome di funzione n-aria di Σ, allora la sua interpretazione f^A è una funzione $X^n \to X$

4. **Locazione e aggiornamento ASM**:

   In ASM una locazione è definita dalla coppia $(f, (v_1, \ldots, v_n)) = loc$, dove f è il nome di funzione e gli $v_1, \ldots, v_n$ i suoi argomenti, e rappresenta l'interpretazione della funzione in un determinato stato (in memoria).

   Un aggiornamento (update) è dato da $(loc, b) = ((f, (v_1, \ldots, v_n)), b)$, ovvero l'interpretazione di f cambia attraverso la modifica dei valori $v_1, \ldots, v_n$ al valore b.

5. **Update Set e consistenza**:

   Un update set è un insieme di aggiornamenti, ognuno della forma $(loc, b) = ((f, (v_1, \ldots, v_n)), b)$

   Dato un update set U, diciamo che questo insieme di aggiornamenti è consistente quando per ogni locazione $(f, (v_1, \ldots, v_n))$, se $((f, (v_1, \ldots, v_n)), b) \in U$ e $((f, (v_1, \ldots, v_n)), c) \in U$ allora si avrà che $b = c$. Se non vale che i valori di b e c sono uguali all'ora l'update si dice inconsistente.

6. **Classificazioni funzioni**:

   Sia M una ASM e sia *env* l'ambiente di M, definiamo le funzioni in base a due principali categorie:

   - Basic
   - Derivate

   a loro volta queste due tipologie si dividono in

   - Statiche: la loro interpretazione non cambia nel corso dell'esecuzione
   - Dinamiche: I loro valori dipendono dagli stati di M. Si dividono a loro volta in
     - Monitorate (in): Scritte da env e lette ma non aggiornate da M
     - Out: Scritte da M e lette da env
     - Controllate: Scritte e lette da M
     - Condivise (Shared): Scritte e lette da M e da env

7. **Definizione ASM**:

   Una Abstract State Machine M è una terna (Σ, A, R), dove:

   - Σ: vocabolario
   - A: stato iniziale per Σ
   - R: insieme di nomi di regole, con un nome di regola di arietà zero chiamata "main rule" che rappresenta il punto di inizio per l'esecuzione della macchina

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

8. **Non determinismo**:

   In ASM il non determinismo è implementato attraverso la choose-rule. La sintassi del costrutto è la seguente: choose x with ϕ do $R[x]$, ovvero scegli casualmente un x che soddisfa la condizione ϕ ed esegui $R[x]$.

   Un esempio:

```
choose i in {0,...,9}, j in {0,...,9} with i < j
and vect(i) > vect(j) do
swap[vect(i),vect(j)]
```

9. **Parallelismo sincrono**:

   In ASM il parallelismo sincrono è implementato attraverso la forall-rule. La sintassi del costrutto è la seguente: forall x with ϕ do $R[x]$, ovvero esegui R in parallelo per ogni x che soddisfa la condizione ϕ.

10. **Parallelismo limitato (sincrono)**:

    In ASM il parallelismo limitato è implementato attraverso la Block rule (composizione parallela). La sintassi del costrutto è la seguente: par R,S endpar, ovvero esegui R ed S in parallelo. Il parallelismo è implementato quindi sia dalla block rule che dalla forall, per la block si parla di parallelismo bounded, per la forall di parallelismo unbounded. Entrambe possono causare aggiornamenti inconsistenti.

```
signature
    [...]

definitions:
rule r_rule1 = skip
rule r_rule2 = skip

main rule r_Main
    par
        r_rule1[]
        r_rule2[]
    endpar
```

11. **Run ASM agente singolo**:

    Data una ASM M, una run (o esecuzione) è definita come una sequenza finita o infinita di stati di M $S_0, \ldots, S_n$ dove S_0 è lo stato iniziale e ciascuno S_{i+1} stato è ottenuto dal precedente S_i eseguendo simultaneamente tutte le regole di transizione che sono eseguibili in S_i

12. **Validazione e verifica sistema**:

    Validazione e verifica sono due approcci volti all'analisi di un modello. La validazione è necessaria a controllare che il sistema soddisfi i requisiti richiesti e questa attività dovrebbe essere svolta prima della verifica in modo da individuare errori e specifiche non corrette il prima possibile. La verifica, invece, è necessaria a garantire proprietà (es. safety, liveness, reachability,...). Forme di analisi sul ground model sono ad esempio le invarianti e la validazione tramite scenari.

13. **Invarianti**:

    Le invarianti in un modello ASM sono utilizzate per esprimere vincoli su funzioni e/o regole che devono essere garantiti in ogni stato. Le invarianti vanno dichiarate prima della main rule e sono dichiarate attraverso la parola chiave *invariant*

14. **ASM multiagente**:

    Le ASM multi-agent descrivono un modello distribuito di computazione attraverso l'esecuzione concorrente di Agenti sincroni o asincroni che hanno in comune stati globali condivisi. La classificazione è quindi tra

    - Sincrone: ogni agente esegue il suo programma parallelamente agli altri sulla base di un clock di sistema condiviso
    - Asincrone: ogni agente esegue il suo programma in parallelo su uno stato locale e sulla base di un clock autonomo indipendente dagli altri

15. **ASM sincrone: computazione**:

    Una ASM multi-agent con agenti sincroni ha una "quasi-sequential run", ovvero una sequenza di stati $S_0, \ldots, S_n$ dove ciascuno stato S_i è ottenuto dal precedente S_{i-1} eseguendo in parallelo le regole di tutti gli agenti

16. **ASM asincrone: computazione**:

    Una ASM multi-agent con agenti asincroni ha una "run parzialmente ordinata", ovvero un insieme parzialmente ordinato $(M, <)$ di mosse m che soddisfano le seguenti proprietà:

    (a) Storia finita: ad un certo istante t, la sequenza di mosse che ha condotto dallo stato S_0 allo stato S_t è finita, ovvero ciascuna mossa ha un numero finito di predecessori. Formalmente $\forall m \in M$ l'insieme $\{m' \mid m' < m\}$ è finito

    (b) Sequenzialità degli agenti: Ogni agente opera in modo sequenziale (le sue mosse sono sequenziali). Formalmente, dato l'agente $a \in Agent$ l'insieme di mosse $\{m \mid m \in M\}$, m eseguita da a, è linearmente ordinato per $<$

    (c) Coerenza: Ogni stato di M è il risultato delle mosse di agenti. Formalmente, dato un segmento iniziale finito X, sottoinsieme chiuso a sinistra

di $(M, <)$ e sia $\sigma(X)$ lo stato associato ad X, ovvero $\sigma(X)$ è il risultato di tutte le mosse di X. Ciascun segmento iniziale finito X di $(M, <)$ ha uno stato associato $\sigma(X)$ che è il risultato dell'applicazione della mossa m nello stato $\sigma(X - \{m\})$ per ogni elemento massimale $m \in X$

17. **Ground model e raffinamento**:

    Il ground model è un modello utilizzato in fase di specifica dei requisiti all'interno del processo di progettazione e sviluppo. Rappresenta il primo modello corretto ma non necessariamente completo dei requisiti e che permette quindi di specificare in forma di definizione matematica ciò che il sistema deve fare. Il GM deve rispettare alcune proprietà:

    - Preciso: rispetto al livello di astrazione prescelto
    - Flessibile: tale da essere modificato ed esteso
    - Semplice e conciso
    - Astratto ma corretto rispetto ai requisiti e completo sul livello di dettaglio desiderato
    - Validabile

18. **Raffinamento**:

    Il raffinamento è quel processo attraverso il quale, a partire da una GM, si passa da una versione astratta ad una più raffinata e concreta. Questo processo è applicato iterativamente fino al raggiungimento del livello desiderato. Il metodo di raffinamento per le ASM non è basato su alcun principio di sostituzione, ma sulle commutazioni algebriche.

    Esistono due tipologie di raffinamento:

    - Orizzontale (Estensione conservativa): raffinamento incrementale utilizzato per l'introduzione di nuovi comportamenti o adattamenti a condizioni dell'ambiente
    - Verticale: aumenta il dettaglio degli elementi del modello

19. **Corretto raffinamento**:

    Fissata la nozione di equivalenza $\equiv$ tra gli stati di interesse e fissati gli stati iniziali e finali, una ASM Mraff è detta corretta raffinamento di M se e solo se:

    ogni Mraff-run raffinata $Sr_0, Sr_1, \ldots$, esiste una M-run $S_0, S_1, \ldots$ e sequenze $i_0 < i_1 < \ldots$ e $j_0 < j_1 < \ldots$ tali che $i_0 = j_0 = 0$ e $S_{ik} \equiv Sr_{jk}$ per ogni k e si verifica una delle due condizioni:

    - Entrambe le run terminano e gli stati finali sono l'ultima coppia di stati equivalenti; oppure
    - Entrambe le run sono infinite

20. **Corretto raffinamento stuttering**:

    Fissata la nozione di equivalenza $\equiv$ tra gli stati di interesse e fissati gli stati iniziali e finali, una ASM Mraff è detta corretta raffinamento di M se e solo se:
    ogni Mraff-run raffinata $Sr_0, Sr_1, \ldots$ può essere ripartita in sotto-run $\rho_0, \rho_1, \ldots$ ed esiste una M-run $S_0, S_1, \ldots$ tale che $\forall \rho_i$ vale che $\forall Sr \in \rho_i : Sr \equiv S_i$

21. **Automa di Kripke**:

    Un modello per l'interpretazione di una formula CTL è data dall'automa di Kripke con relazione di serialità. Formalmente un automa di Kripke è definito dalla quadrupla $M = (S, \Delta, I, L)$, dove:

    - S: insieme degli stati
    - Δ (o anche $\rightarrow$): funzione di transizione, tale che $\forall s \in S \quad \exists s' \in S$ con $s \rightarrow s'$
    - I: insieme degli stati iniziali
    - L: funzione di etichettatura definita come $L : S \rightarrow 2^{PA}$, con PA insieme delle parti Si impone l'assenza di deadlock con un eventuale aggiunta di uno stato fittizio s_d

22. **Verifica di formule CTL ben formate**:

    Per verificare che una formula CTL sia ben formata è necessario costruire il relativo albero di parsing. Un albero di parsing è un albero costruito a partire da una formula CTL leggendola da sinistra a destra e associando ad ogni nodo un operatore. Gli operatori si dividono in due categorie:

    - Operatori unari: $\neg, AG, EG, AF, EF, AX, EX$
    - Operatori binari: $\wedge, \vee, \rightarrow, AU, EU$

    Vediamo il significato delle lettere:

    - A: lungo tutti i percorsi
    - E: lungo almeno un percorso
    - X: successivo
    - F: qualche stato futuro
    - G: tutti gli stati futuri (globalmente)

    A ed E danno indicazioni sui path a partire dallo stato corrente.E è il duale di A. Invece X,F,G riguardano gli stati presenti all'interno di quei path.

23. **Semantica dei connettivi CTL**:

    - $M, s \models AX\phi$: in tutti gli stati successivi a s
    - $M, s \models EX\phi$: in qualche stato successivo a s
    - $M, s \models AG\phi$: In tutti i path a partire da s vale sempre la condizione ϕ

- $M, s \models EG\phi$: Esiste almeno un path a partire dallo stato corrente s in cui è sempre verificata la condizione ϕ (in quel path per ogni stato vale ϕ). Formalmente $M, s \models EG\phi$ sse esiste un path $s_1 \to s_2 \to \ldots$, con $s_1 = s$, in cui $\forall i, M, s_i \models EG\phi$
- $M, s \models AF\phi$: In tutti i path a partire da s esiste uno stato futuro, compreso quello presente s, in cui la proprietà ϕ è soddisfatta
- $M, s \models EF\phi$: Esiste un path di computazione che inizia da s in cui esiste uno stato in cui la proprietà ϕ è soddisfatta
- $AU(\phi_1 U \phi_2)$: In tutti i path che iniziano con s è vero che $\phi_1 U \phi_2$
- $EU(\phi_1 U \phi_2)$: Esiste un path di computazione che inizia con s e in cui è vero $\phi_1 U \phi_2$

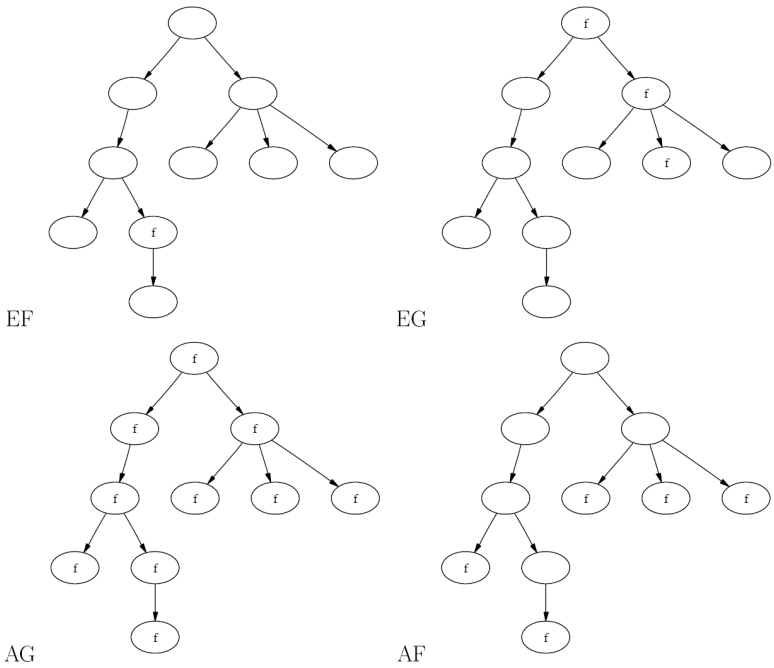


Figura 1.2: Tree CTL

24. **Equivalenza formule**

- $\neg AF\phi \leftrightarrow EG\neg\phi$
- $\neg EF\phi \leftrightarrow AG\neg\phi$
- $\neg AX\phi \leftrightarrow EX\neg\phi$
- $AF\phi \leftrightarrow A(\top\ U\ \phi)$
- $EF\phi \leftrightarrow E(\top\ U\ \phi)$

Possiamo dimostrare l'equivalenza di alcune di queste formule:

Dimostrazione. $\neg AF\phi \leftrightarrow EG\neg\phi$.

Per fare ciò è necessario dimostrare le due direzioni di equivalenza.

(a) $\neg AF\phi \rightarrow EG\neg\phi$
Per ipotesi $\forall M, \forall s$ supponiamo valga $M, s \models \neg AF\phi$. Negare $AF\phi$ significa dire che esisterà un cammino in cui la *phi* non varrà mai. Ovvero $\exists$ un path $s_1 \rightarrow s_2 \rightarrow \ldots$ tale che $s_1 = s$ e $\forall i, M, s_i \models \neg\phi$. Questo per definizione equivale a dire $M, s \models EG\neg\phi$

(b) $EG\neg\phi \rightarrow \neg AF\phi$
Ragioniamo per assurdo (negando la tesi). Supponiamo che $\forall M, \forall s.$ $M, s \models AF\phi$. Se ciò è vero, stando alla definizione si avrà che per ogni cammino a partire da s esisterà uno stato in cui vale ϕ. Ma questa affermazione contraddice l'ipotesi $EG\neg\phi$.

□

Dimostrazione. $\neg AX\phi \leftrightarrow EX\neg\phi$.

Per fare ciò è necessario dimostrare le due direzioni di equivalenza.

(a) $\neg AX\phi \rightarrow EX\neg\phi$
Per ipotesi $\forall M, \forall s$ supponiamo valga $M, s \models \neg AX\phi$. Negare $AX\phi$ significa dire che esisterà uno stato successivo $s', s \rightarrow s'$ in cui la ϕ non varrà. Questo per definizione equivale a dire $M, s \models EX\neg\phi$

(b) $EX\neg\phi \rightarrow \neg AX\phi$
Ragioniamo per assurdo (negando la tesi). Supponiamo che $\forall M, \forall s.$ $M, s \models AX\phi$. Ovvero esiste uno stato successivo allo stato presente in cui vale ϕ. Questa affermazione contraddice l'ipotesi per cui rimuoviamo l'assurdo e affermiamo quanto detto.

□

Normalmente nella dimostrazione dell'equivalenza di due formule ci si riconduce sempre a dimostrare che la prima equivalenza è vera per definizione e la seconda (quella di negazione) va fatta per assurdo.

25. **Algoritmo di labelling**:

Quando si svolgono esercizi per il model checking bisogna ricondurre ogni formula CTL ad una ad esse equivalente contenente solamente i simboli: $\bot$, $\wedge, \neg, AF, EU, EX$. Alcune riscritture possono essere:

- $\top = \neg \bot$
- $\phi_1 \vee \phi_2 = \neg(\neg\phi_1 \wedge \neg\phi_2)$
- $\phi_1 \rightarrow \phi_2 = (\neg\phi_1 \vee \phi_2)$
- $AX(\phi) = (\neg EX\neg\phi)$
- $EF(\phi) = E(\top U\phi)$
- $EG(\phi) = (\neg AF\neg\phi)$
- $AG(\phi) = (\neg EF\neg\phi)$
- $A(\phi_1 U\phi_2) = \neg(E\left[\neg\phi_2 U(\neg\phi_1 \wedge \neg\phi_2)\right] \vee \neg AF\phi_2)$

Algoritmo di model checking utilizzato per verificare per quali stati vale una determinata proprietà dato un automa in figura.

26. **Model checking - SAT**:

function SAT(ϕ)
begin
case
ϕ è $\top$: **return** S
ϕ è $\bot$: **return** $\emptyset$
ϕ è un atomo : **return** $\{s \in S \mid \phi \in L(s)\}$
ϕ è $\neg\phi$: **return** $S-$SAT(ϕ)
ϕ è $\phi_1 \wedge \phi 2$: **return** SAT(ϕ_1) $\cap$ SAT(ϕ_2)
ϕ è $\phi_1 \vee \phi 2$: **return** SAT(ϕ_1) $\cup$ SAT(ϕ_2)
ϕ è $\phi_1 \rightarrow \phi 2$: **return** SAT($\neg\phi_1 \vee \phi_2$)
ϕ è AX(ϕ_1) : **return** SAT($\neg$EX$\neg\phi_1$)
ϕ è EX(ϕ_1) : **return** SAT$_{EX}$(ϕ_1)
ϕ è A(ϕ_1 U ϕ_2) : **return** SAT($\neg$(E[$\neg\phi_2$ U ($\neg\phi_1 \wedge \neg\phi 2$)] $\vee$ EG$\neg\phi_2$))
ϕ è E(ϕ_1 U ϕ_2) : **return** SAT$_{EU}$(ϕ_1, ϕ_2)
ϕ è EF(ϕ_1) : **return** SAT(E($\top$ U ϕ_1)
ϕ è EG(ϕ_1) : **return** SAT($\neg$AF$\neg\phi_1$)
ϕ è AF(ϕ_1) : **return** SAT$_{AF}$(ϕ_1)
ϕ è AG(ϕ_1) : **return** SAT($\neg$EF$\neg\phi_1$)
end case
end function

Figura 1.3: Algoritmo di SAT

27. **Model checking -** $SAT_{AF}(\phi)$:

Vengono utilizzati gli insiemi X e Y come variabili. X è inizializzato con tutti gli stati della macchina. Y viene invece inizializzato con gli stati che soddisfano il SAT di ϕ. L'algoritmo inizia e si ripete finché X e Y non saranno uguali.

Quando il repeat inizia X viene inizializzato a Y e Y è uguale a se stesso unito agli stati successori di quello presente.

Alla fine viene restituito Y

28. **Model checking -** $SAT_{EU}(\phi, \psi)$:

Vengono utilizzati gli insiemi W,X ed Y come variabili. All'inizio W è inizializzato con il $SAT(\phi)$, X con l'insieme degli stati e Y con il $SAT(\psi)$.

function $\mathrm{SAT}_{AF}(\phi)$
local var X, Y
begin
  $X := S$;
  $Y := \mathrm{SAT}(\phi)$;
  repeat until $X = Y$
  begin
    $X := Y$;
    $Y := Y \cup \{s \mid \forall s' \text{ con } s \to s' \text{ e } s' \in Y\}$
  end
  return Y
end function

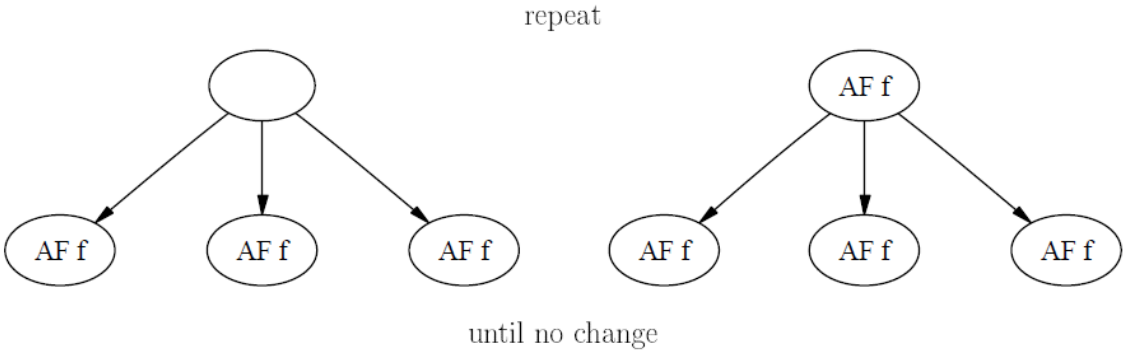


Figura 1.4: SAT AF

Quando il repeat inizia X viene inizializzato a Y e Y è uguale a se stesso unito a W intersecato con tutti gli stati per cui i loro successori appartengono all'insieme Y che soddisfano ψ.

Alla fine viene restituito Y

29. **Model checking -** $SAT_{EX}(\phi)$:

    Vengono utilizzati gli insiemi X e Y come variabili. X è inizializzato con gli stati che soddisfano il SAT di ϕ, Y invece con qualsiasi stato appartenente all'insieme degli stati S i cui successori soddisfano ϕ. L'algoritmo non si ripete ma ha una sola esecuzione.

    Alla fine viene restituito Y

30. **Reduced Ordered Binary Decision Diagram - ROBDD**: g

31. **Proprietà di raggiungibilità**:

    Afferma che determinate situazioni particolari possono essere raggiunte. In CTL si esprime con $EF\phi$, ovvero esiste un cammino dallo stato corrente lungo cui un qualche stato soddisfa ϕ.

function $\mathrm{SAT}_{EU}(\phi, \psi)$
local var W, X, Y
begin
  $W := \mathrm{SAT}(\phi)$;
  $X := S$;
  $Y := \mathrm{SAT}(\psi)$;
  repeat until $X = Y$
  begin
    $X := Y$;
    $Y := Y \cup (\mathrm{W} \cap \{s \mid \exists s' \text{ con } s \to s' \text{ e } s' \in Y\}$
  end
  return Y
end function

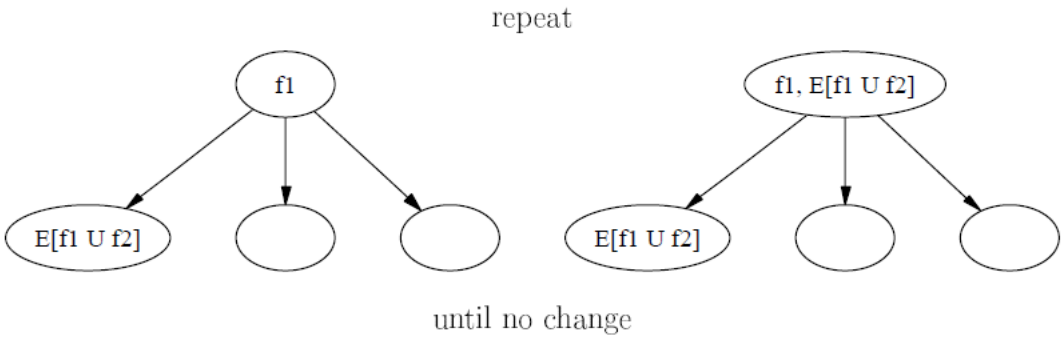


Figura 1.5: SAT EU

function $\mathrm{SAT}_{EX}(\phi)$
local var X, Y
begin
  $X := \mathrm{SAT}(\phi)$;
  $Y := \{s_0 \in S \mid s_0 \to s_1 \text{ per qualche } s_1 \in X\}$
  return Y
end function

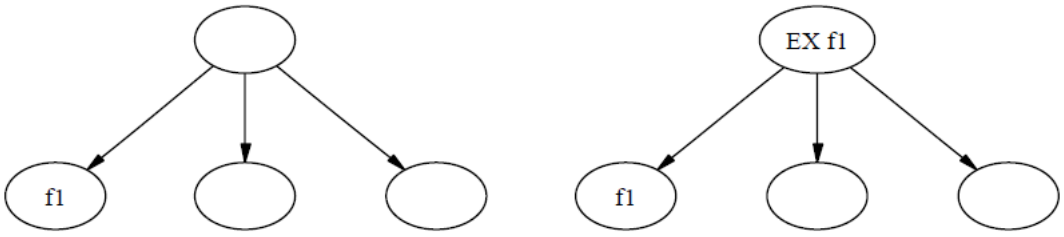
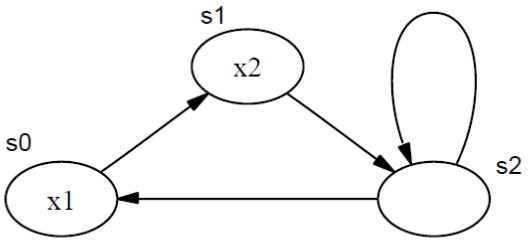


Figura 1.6: SAT EX



Consideriamo il modello CTL dato in figura, con ordinamento $[x_1,x_2]$:

$$\begin{aligned} S &= \{s_0, s_1, s_2\} \\ \rightarrow &= \{(s_0, s_1), (s_1, s_2), (s_2, s_0), (s_2, s_2)\} \\ L(s_0) &= \{x_1\} \\ L(s_1) &= \{x_2\} \\ L(s_2) &= \emptyset \end{aligned}$$

set of states	boolean values	boolean function
$\emptyset$		$\bot$
$\{s_0\}$	(1,0)	$x_1 \wedge \neg x_2$
$\{s_1\}$	(0,1)	$\neg x_1 \wedge x_2$
$\{s_2\}$	(0,0)	$\neg x_1 \wedge \neg x_2$
$\{s_0, s_1\}$	(1,0),(0,1)	$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_2$
$\{s_0, s_2\}$	(1,0),(0,0)	$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge \neg x_2$
$\{s_1, s_2\}$	(0,1),(0,0)	$\neg x_1 \wedge x_2 \vee \neg x_1 \wedge \neg x_2$
$\{s_0, s_1, s_2\}$	(1,0),(0,1),(0,0)	$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_2 \vee \neg x_1 \wedge \neg x_2$

Figura 1.7: ROBDD esercizio

Si parla di raggiungibilità condizionata quando una condizione restringe la forma dei cammini che raggiungono lo stato desiderato. In CTL si usa il connettivo EU.

La raggiungibilità inoltre può essere inoltre applicata a stati raggiungibili. La raggiungibilità da stati raggiungibili richiede l'annidamento di EF e di AG.

32. **Proprietà di safety**

    Afferma che, sotto certe condizioni, un evento non può mai accadere (cose cattive non si verificano mai). In CTL si esprime con $AG\phi$. Ogni stato di ogni cammino dallo stato corrente soddisfa ϕ e la ϕ esprime proprio l'evento o fatto che non si deve verificare.

    La safety condizionata utilizza il connettivo AU come: $AU\ [\phi U \psi]$: ϕ non deve verificarsi finché non si verifica ψ (per esprimere ciò la ϕ va messa col $\neg$).

    Safety = not-reachability. Nella maggior parte dei casi una proprietà di safety si esprime come la negazione di una proprietà di raggiungibilità: the system can not reach a state in which: $\neg EF(...)$ (oppure $AG())$

33. **Proprietà di liveness**

    Afferma che, sotto certe condizioni, un qualche evento alla fine accadrà, ovvero cose buone prima o poi se verificano. In CTL si esprime con: $AG + AF$ oppure $AG + EF$.

34. **Assenza di deadlock**

    Afferma che il sistema non si troverà in una condizione in cui non è possibile alcun progresso (situazione di stallo). In CTL si esprime con: $AG\ EX\ true$, ovvero qualunque sia lo stato raggiunto (AG), esiste uno stato immediatamente successore ($EX\ true$)

35. **Proprietà di fairness**

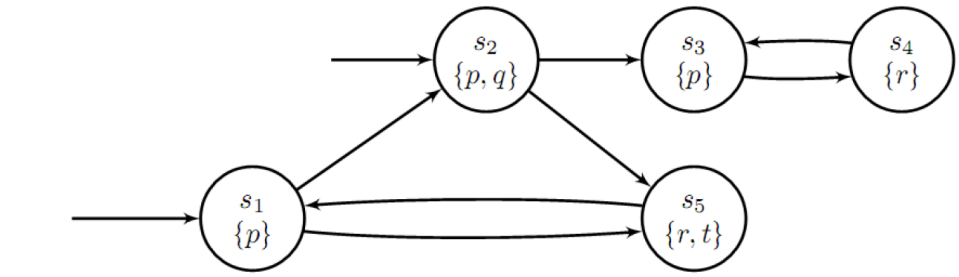
    Afferma che, sotto certe condizioni, un evento accadrà infinitamente spesso. Le proprietà di fairness non possono essere espresse in pura CTL perché manca l'operatore $F^{\propto}$, ovvero un numero infinito di volte o infinitamente spesso. In SMV le ipotesi di fairness vanno specificate come parte del modello piuttosto che ricorrere alla logica CTL+fairness.

36. **NuSMV**:

    Un programma in NuSMV è formato da un modulo main e tre parti principali:

    - VAR: parte di codice in cui vengono definite le variabili
    - ASSIGN: vengono inizializzate le variabili e vengono descritte le "evolution"

- CTLSPEC: definisce le proprietà da verificare In questo modo si mappa un automa di Kripke in un modello NuSMV:




```
MODULE main
    VAR
        state: {s1, s2, s3, s4, s5};
    DEFINE
        p := state in {s1, s2, s3};
        q := state = s2;
        r := state in {s4, s5};
        t := state = s5;
```

```
ASSIGN
    init(state) := {s1, s2};
    next(state) :=
        case
            state = s1: {s2, s5};
            state = s2: {s3, s5};
            state = s3: s4;
            state = s4: s3;
            state = s5: s1;
            TRUE: state;
        esac;

CTLSPEC AF r
```

Figura 1.8: Automa di Kripke modellato in NuSMV

Capitolo 2

PRATICA

Per ogni esercizio creo un project generico che mi va a creare una cartella all'interno della mia workspace. Una volta fatto ciò vado a creare all'interno della cartella appena creata un file contenente il nome del mio esercizio con ".asm".

Cosa da fare in ordine:

1. import StandardLibrary
2. signature: dichiarazione variabili
3. definitions: definizioni variabili e funzioni
4. main rule r_Main=
5. default init s0: possibile definire un insieme di stati iniziali, solo domini concreti dinamici e solo funzioni controllate (no monitorate)

2.1 Dichiarazione dei domini

I domini rappresentano i tipi assunti dalle variabili nei normali linguaggi di programmazione.

ATTENZIONE: I NOMI DEI DOMINI INIZIANO CON LA LETTERE MAIUSCOLA PER CONVENZIONE

Attualmente i domini implementati in AsmetaL possono essere dei seguenti tipi:

- **Type-domain**: Tipi di domini
  - **basic type domains** : definiti nella standard library
    - basic domain Complex
    - basic domain Real
    - basic domain Integer
    - basic domain Natural
    - basic domain Char
    - basic domain String

    - basic domain Boolean
    - basic domain Undef
  - **enum**: enumerazione. Domini che assumono valori finiti di valori definiti da noi.

    enum domain <nomeDominio> = {VALORE1 | VALORE 2 | ...}
  - abstract: elementi di natura "astratta", non definiti se non attraverso funzioni definite su tale dominio. Agent e Reserve definiti nella standard library sono domini astratti.

    asbtract domain <nomeDominio>
  - concrete domain:

    domain <nomeDominio> subset of basicType

- **Concrete domain**: sottoinsiemi dei type domain definiti dall'utente

  [dynamic] domain D subsetof td

  dov D è il nome del dominio da dichiarare, td è il type-domain di cui D è sottoinsieme. La parola chiave dynamic è opzionale e denota che l'insieme è dinamico. Per default, un dominio è statico e va definito nella sezione definitions.

2.2 Funzioni

Le funzioni rappresentano le variabili di un normale linguaggio di programmazione

- Static: funzioni invariate durante la run
- Dynamic: aggiornate durante la run
- Monitored: modificato dall'ambiente (letto dalla macchina)
- Controlled: modificato dalle regole come azioni dell'agente
- Shared: modificato dall'ambiente e dalle regole
- Derived: definito da uno schema fisso in termini di altre funzioni (statiche o dinamiche).

2.3 Prod – Prodotto cartesiano

Prod (con la P maiuscola) rappresenta Il prodotto cartesiano dei domini d1,...,dn, cioè quell'insieme che contiene tutte le combinazioni tra gli elementi:

derived winner : Prod(Play, Play) -> Winner

Viene spesso utilizzato nelle funzioni derivate per fare dei confronti oppure per creare una matrice, ad esempio una scacchiera. Un esempio classico è quello della morra cinese, dei filosofi, o la scelta tra due giocatori che estraggono un valore e bisogna vedere chi vince.

2.4 CTL

Le CTL vanno definite all'interno della sezione **definitions:** e vanno dichiarate tramite keyword:

CTL(ctlSpecificata)

ctlSpecificata è una formula in logica CTL

Vediamo alcuni esempi per facilitare la scrittura:

- In ogni stato: ACLSPEC(ag(condizione))
- Esiste uno stato in cui vale che: ACLSPEC(ef(condizione))

2.5 AVALLA - Creazione scenari

Utilizzo per creare gli scenari per il nostro modello asm.

I file avalla hanno estensione ".avalla" e sono strutturati come segue:

1. scenario "nomeScenario"
2. load "nomeFile.asm"

Le azioni che possono essere specificate sono le seguenti:

- set funzione := valoreFunzione
- check funzione = valoreFunzione
- step -> effettua uno step della macchina

2.6 Agenti

In Asmeta il dominio di un agente va dichiarato come:

domain "nomeDominioAgente" subsetof Agent

In seguito un agente è definito statico come:

static "nomeAgente": "nomeDominioAgente"

Quando si lavora con gli agenti, cosa importante è andare a definire nello *default init s0* l'associazione tra un agente e la rule a cui esso deve riferirsi. L'associazione va fatta in questo modo:

agent "nomeDominioAgente": "nomeRule[ ]"

Questo perché poi all'interno del main, quasi sempre formato da un par in cui c'è una

forall per gli agenti, non vengono chiamate direttamente le rule ma si usa il costrutto:

program("nomeAgente)

L'agente avrà già associata la rule grazie a quanto definito nel default init. Nota che all'interno della rule nel caso bisogna riferirsi all'agent stesso si usa la parola chiave **self**

Capitolo 3

Domande esami

Le risposte alle domande sono i numeri assegnati alle definizioni teoriche del capitolo precedente :)

- CRISTO MORTO