www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Modellazione e analisi di sistemi

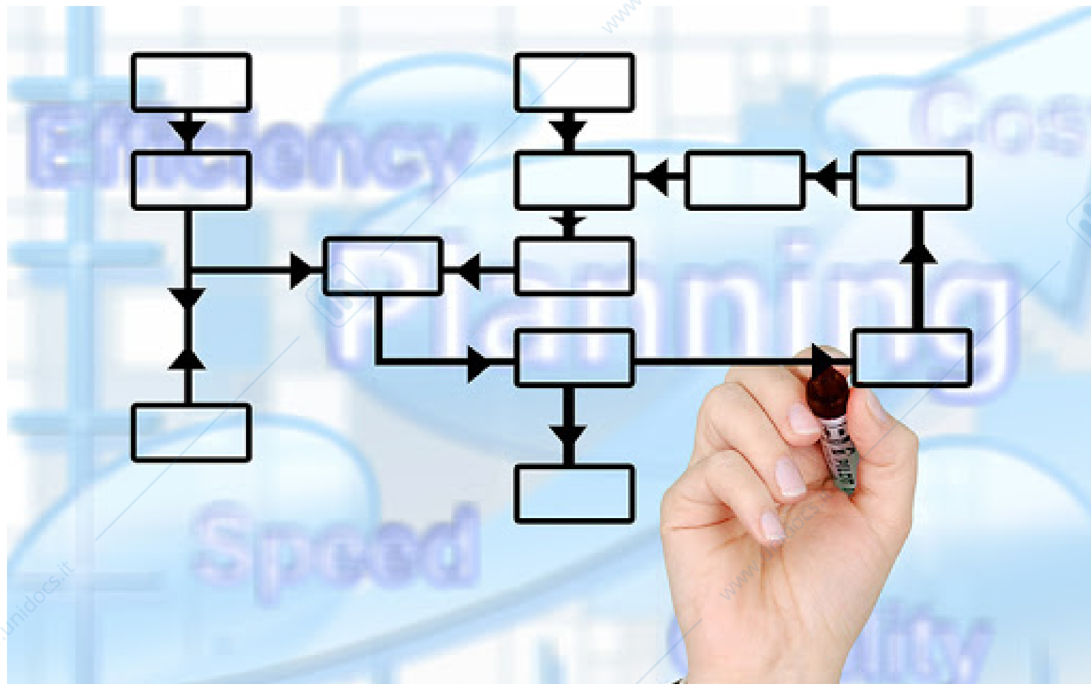
Modellazione E Analisi Di Sistemi (Università degli Studi di Milano)

Studocu non è sponsorizzato o supportato da nessuna università o ateneo.
Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Modellazione e analisi di sistemi

Ilaria Salbe



Appunti per il CdL Magistrale:
Sicurezza Informatica

Università degli Studi di Milano
Anno Accademico 2019/2020

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Indice

1 Introduzione **3**
- 1.1 Metodi formali . . . 3
- 1.2 Ripasso . . . 4

2 Introduzione alle Abstract State Machines **7**
- 2.1 Stato in ASM . . . 11
- 2.2 Regole di transizione . . . 13
- 2.3 Esempi . . . 16
  - 2.3.1 Orologio . . . 16
  - 2.3.2 Orologio 2 . . . 17
  - 2.3.3 Safety Injection System . . . 18
  - 2.3.4 Sistema di controllo del semaforo a senso unico . . . 19
  - 2.3.5 SluiceGate . . . 21
  - 2.3.6 Fattoriale . . . 23
  - 2.3.7 Fattoriale con stop . . . 23
  - 2.3.8 Ordinare un vettore . . . 24
  - 2.3.9 Operazioni sullo stack - Esempio di riserva . . . 24
  - 2.3.10 Esempio su aggiornamenti inconsistenti . . . 25

3 Asmeta **25**
- 3.1 AsmetaL . . . 27
  - 3.1.1 Linguaggio strutturale . . . 27
  - 3.1.2 Linguaggio delle definizioni . . . 29
  - 3.1.3 Linguaggio dei termini . . . 30
  - 3.1.4 Linguaggio delle regole . . . 32
- 3.2 Simulatore: AsmetaS . . . 34
- 3.3 Animator: AsmetaA . . . 34
- 3.4 Visualizer: AsmetaVis . . . 34

4 Prime tecniche di analisi **35**

5 Composizione di modelli: ASM multi agenti **38**

6 Metodologia di specifica: Ground Model e raffinamento di modelli **41**
- 6.1 Ground Model (GM) . . . 41
- 6.2 Raffinamento . . . 44

7 Model Checking **52**
- 7.1 Computation Tree Logic - CTL . . . 53
- 7.2 Algoritmi per Model Checking . . . 57
  - 7.2.1 Labelling Algortihm . . . 58
  - 7.2.2 Pseudo-codice dell'algoritmo di controllo del modello CTL . . . 59

8 Rappresentazione di un automa di Kripke e di una formula CTL con ROBDD (Reduced Ordered Binary Decision Diagrams) **61**
- 8.1 Idea base degli algoritmi per la manipolazione dei ROBDD . . . 66
- 8.2 Symbolic Model Checking = Model Checking + ROBDD . . . 67

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

9 Verifica Di Proprietà Temporali **68**
9.1 Reachability . . . 68
9.2 Safety . . . 69
9.3 Liveness . . . 69
9.4 Assensa di Deadlock . . . 70
9.5 Proprietà di Fairness . . . 70

10 Esercizi 3.3 pag 165 (Soluzioni) **72**

11 NuSMV **73**

12 AsmetaSMV: un model checker per modelli AsmetaL **79**
12.1 Domini e funzioni . . . 80
12.2 Regole . . . 86
12.3 Proprietà CTL . . . 91

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

1 Introduzione

Il corso presenta le fondamentali tecniche per la modellazione e la verifica formale di sistemi HD/SW. Gli strumenti che verranno utilizzati sono:

- I linguaggi di specifica, che ci permettono di realizzare modelli comprensibili a tutti in maniera oggettiva. Consentono di descrivere un sistema da analizzare e le proprietà da provare.
- I fondamenti teorici delle metodologie di analisi (validazione e verifica) basate su simulazione, testing e model checking. Nel corso ci soffermeremo principalmente sulle metodologie basate su model checking.
- Gli strumenti automatici (tool) che consentono la verifica ((semi-)automatica e/o interattiva).

1.1 Metodi formali

I metodi formali sono notazioni rigorose che consentono di descrivere un sistema in termini di oggetti matematici (modello o specifica). Il modello è una descrizione astratta del sistema che si vuole rappresentare, espressi in termini di oggetti matematici. Un metodo non è solo una notazione, ma è supportato da tecniche per analizzare sistemi, basate sulla matematica. Modellare implica astrarre, cioè semplificare la descrizione del sistema, conservando solo alcuni dei dettagli originali, e focalizzandosi sulle proprietà principali. Una volta che il modello soddisfa i requisiti minimi, viene via via raffinato, aggiungendo dettagli, al fine di catturare tutti i dettagli del sistema che si desidera rappresentare.

Le specifiche in linguaggio naturale sono: inconsistenti, ambigue e difficili da seguire. Mentre, le specifiche formali forzano a ragionare per estrapolare la logica di funzionamento di un sistema. Inoltre, possono essere usate per provare la correttezza del programma e le sue proprietà e per generare casi di test.

Esistono due tipi diversi di formalismi:

- Formalismi operazionali: forniscono una descrizione del comportamento del sistema in termini di operazioni di una macchina astratta. Il funzionamento della macchina astratta rappresenta il modello stesso.
- Formalismi dichiarativi: definiscono il sistema in termini di proprietà che devono essere soddisfatte. Le proprietà vengono espresse in linguaggio matematico.

Ad esempio: Il requisito da soddisfare è: Il valore di x sarà tra 1 e 5, finché ad un certo momento diventa 7. In ogni caso non sarà mai negativo. Vediamo come può essere espresso nelle diverse modalità di specifica:

- Dichiarativo (logica temporale): $A(1 \leq x \geq 5 \vee x = 7) \wedge AG(x \geq 0)$
- Operazionale

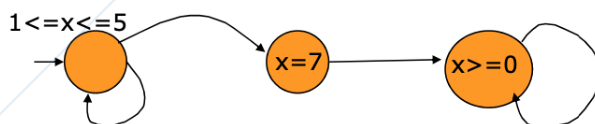


Figura 1: operazionale

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

I metodi formali presentano essi stessi dei limiti:

- Scrivere una specifica formale è come scrivere un programma: è possibile introdurre errori e non si è comunque sicuri di catturare il comportamento "inteso" del programma.
- É vero che è possibile fare una prova di correttezza, per testare se il modello è corretto, ma è un processo lungo e difficile: su carta, è facile introdurre errori, mentre sul calcolatore, chi assicura che (il programma usato) sia corretto?

L'analisi è suddivisa in validazione e verifica. La validazione è il processo di valutare un sistema o un componente durante o alla fine del processo di sviluppo per determinare se esso soddisfa i requisiti specificati. La verifica, invece, è il processo di valutare un sistema o un componente per determinare se i prodotti di una data fase di sviluppo soddisfano le condizioni (proprietà) imposte allo stato di detta fare. La validazione è necessaria per controllare che il sistema soddisfa i requisiti richiesti, attraverso simulazione e testing. La verifica, invece, è necessaria per garantire proprietà, attraverso dimostrazioni di proprietà (con thorem prover e model checker (tool automatico)). Non determina errori causati da specifiche incorrette. Gli errori possono essere propagati nelle fasi successive. La validazione è una simulazione che diventa più semplice tramite l'utilizzo dei formalismi operazionali, che descrivono graficamente il funzionamento del sistema. Alcuni esempi di formalismi operazionali: Abstract State Machines, B method, Z method, SCR (Software Cost Reduction) e Reti di Petri. I formalismi dichiarativi, che esprimono il sistema in termini di proprietà/condizioni, facilitano la verifica che è appunto il processo che controlla che la proprietà siano soddisfatte. Alcuni esempi di formalismi dichiarativi sono: logica temporale, Trio e algebre dei processi. Le tecniche di V&V (Validazione & Verifica) sono:

- Testing: Controlla le esecuzioni del software in accordo a qualche schema di copertura;
- Model checking: Usa un qualche software (tool) per controllare in automatico che la specifica soddisfa certe proprietà;
- Verifica deduttiva: Usando un qualche formalismo logico, dimostra formalmente che la specifica è corretta.

Esistono alcuni cattivi preconcetti sui metodi formali e sui processi di verifica:

- I metodi formali possono essere usati solo dai matematici. *Errato! Hanno un fondamento matematico e quindi rigoroso, ma l'utente dovrebbe non preoccuparsene.*
- Il processo di verifica è esso stesso soggetto ad errori, quindi perché preoccuparsene? *Lo adottiamo per ridurre gli errori, non per eliminarli completamente.*
- L'uso di metodi formali rallenta la realizzazione del progetto. *É probabile che invece lo velocizzi, dal momento che gli errori sono scoperti nelle fasi iniziali.*

Ma dall'altro canto esistono anche alcune esagerazioni su di essi:

- La verifica automatica può sempre trovare errori
- La verifica deduttiva può dimostrare che il software è completamente sicuro
- Il Testing è il solo metodo utilizzato nella pratica dalle industrie

1.2 Ripasso

Supponiamo di voler modellare un luchetto a combinazione, con sole tre chiavi A, B, C. Il luchetto si apre solo quando la combinazione ABA viene inserita. Questo sistema può essere rappresentato con un automa con 4 stati e 9 transizioni.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

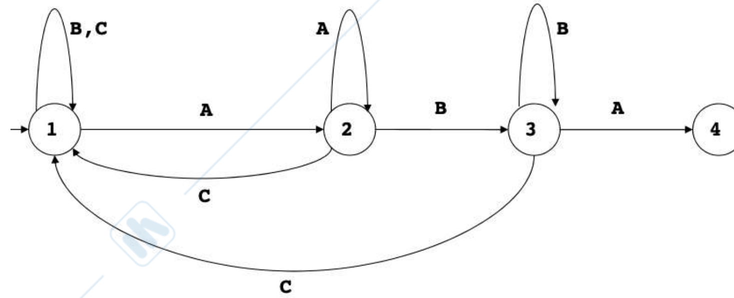


Figura 2: Sistema di esempio

Un'esecuzione è una sequenza di stati che descrive una possibile evoluzione del sistema. Le possibili esecuzioni del digicode sono:

- 1
- 11, 12
- 111, 112, 121, 122, 123
- 1111, 1112, 1121, 1122, 1123, 1211, 1212, 1221, 1222, 1223, 1231, 1234, ...

Queste possono essere organizzate sotto forma di albero.

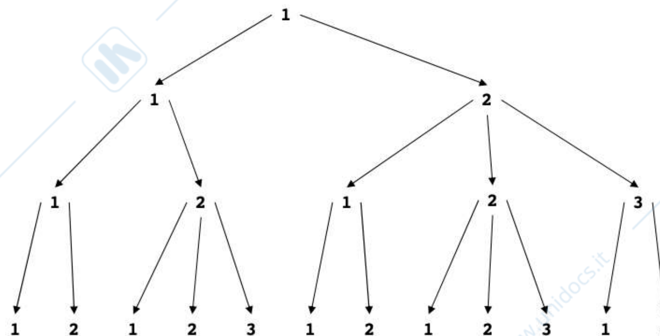


Figura 3: Albero delle esecuzioni

Un automa (o macchina) a stati finiti, abbreviata FSM (Finite State Machine), è una notazione formale che permette la rappresentazione astratta del comportamento di un sistema. Le FSM hanno: una rigorosa definizione matematica e una intuitiva rappresentazione grafica tramite diagrammi di stato. Un FSM è una tupla (S, I, δ), dove:

- S: insieme finito di stati
- I: insieme finito di eventi di input
- $\delta : S \times I \rightarrow S$: Funzione di transizione

Un diagramma di stato è un grafo direzionato i cui nodi rappresentano gli stati ed i cui archi, etichettati dagli input, rappresentano le transizioni di stato.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Esempio:
S = {s1, s2, s3}
I = {a, b}
δ = {(s1,a,s1), (s1,b,s2),
(s2,a,s2), (s2,b,s3),
(s3,a,s3), (s3,b,s1)}

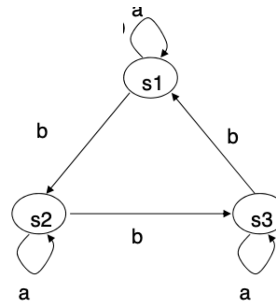


Figura 4: Transizioni di stato

Sono utilizzate per modellare con FSMS:

- GUIs
- Protocolli di rete
- Pacemakers e teller machines
- Applicazioni WEB
- Software di sicurezza
- Sistemi embedded

Non tutti i requisiti possono essere specificati con una FSM. Tra i requisiti non specificabili in FSM: requisiti real time, requisiti riguardanti performance e requisiti riguardanti tipi di computazioni. È possibile rappresentare solo un numero finito di stati. Le FSM non sono composizionali. Il modello FSM di un sistema non è unico. Infatti, si possono ottenere modelli diversi in base al livello di astrazione/dettaglio considerato. Lo stato è non strutturato: non consente di modellare informazioni. Le operazioni sono elementari: input esterni e output della macchina (automi di Mealy e di Moore).

Dal momento che i sistemi si sono evoluti (ad esempio sistemi distribuiti), le rappresentazioni sono state estese per poter soddisfare lo sviluppo dei sistemi:

- Per modellare l'output → Automi di Mealy e di Moore. La transizione diventa una tupla (S, I, O, S')
- Per modellare la memoria della macchina → FSM estese. La transizione è una tupla (S, I, O, G, A, S')
- Per modellare la comunicazione → FSM di comunicazione. La coppia (C, P), dove C rappresenta i canali e P i processi (FSM). Le transizioni $(s, null, s')$, $(s, c?i, s')$, $(s, c!o, s')$, dove $c?i$ rappresenta un dato di input e $c!o$ rappresenta un dato di output.
- Per modellare il tempo → FSM temporizzate. La transizione è una tupla $(s, g, a|I/O, s', [t_1, t_2])$, dove t_1 e t_2 rappresenta l'intervallo di tempo in cui la transizione è valida.
- Per modellare stato e transizioni strutturate, sottomacchina e computazione parallela → FSM di Harel (macchine di stato di UML). In uno stato posso descrivere una macchina sequenziale.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

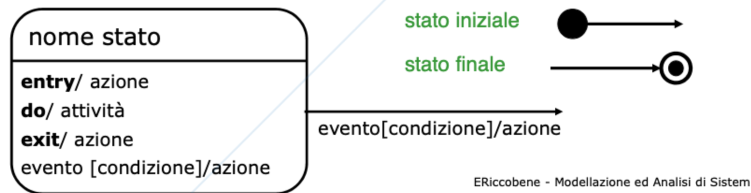


Figura 5: FSM di Harel

- Abstract State Machines → ASM = FSM con stati generalizzati.
- Automi di Kripke → lo stato porta informazione di una proprietà che è valida in esso.

2 Introduzione alle Abstract State Machines

Le ASM (Abastract State Machine) sono macchine a stati finiti con stati generalizzati. Rappresentano la forma matematica di Macchine Virtuali che estendono la nozione di Finite State Machine. Gli stati di controllo non strutturati delle FSM sono sostituiti nelle ASM da stati strutturati che modellano:

- Dati complessi arbitrari, con domini di base e funzioni per la struttura
- Operazioni per la manipolazione di dati

Lo stato rappresenta, nelle ASM, una struttura del primo ordine (perché usiamo variabili) multi sorted (perche possiamo usare domini differenti); vengono chiamati algebre. In ASM, le transizioni sono regole che descrivono il cambiamento di funzioni da uno stato al successivo. La regola base (basic transition rule) è "if Condition then Updates". Altri costruttori di regole sono:

- parallellismo (par) ed azioni sequenziali (seq)
- iterazioni (while) ed invocazione di sottomacchine
- non-determinismo (choose) e parallellismo sincrono non limitato (forall)
- multi-agenti sincroni/asincroni

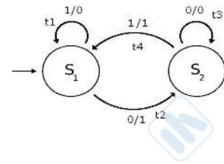
Le ASM sono dotate di un ambiente tool per:

- Editing
- Simulazione
- Validazione
- Verifica (via model checking)
- Generazione di casi di test

Un modello ASM può essere visto come pseudocodice su ADT (strutture dati astratte). Le ADT sono strutture dati ad alto livello. Asmeta (ASM mETAmodeling) mette a disposizione tool e altro materiale sulle ASM.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari



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 = **if** ctl_State = "s1" and input = "0" **then** ...
r_s2_1 = ...
r_s2_0 = ...

Figura 6: Da FSM a ASM

Le ASM sono composte da diverse parti: header, body, main rule, initialization.

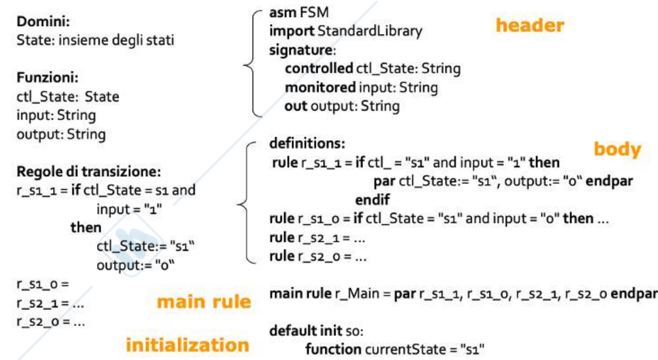


Figura 7: Parti di un modello ASM

In ASM, una FSM può essere definita da programmi della forma (uno per ogni stato).

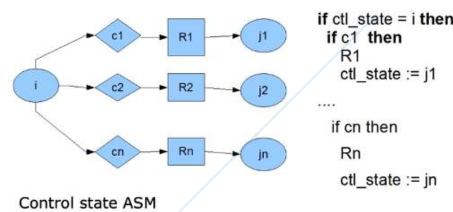


Figura 8: Da FSM a ASM

Dove:

- ctl_state è una variabile che rappresenta lo stato corrente e può prendere come valore uno stato di controllo (in un insieme finito)
- i, $j_1, j_2, ..., j_n$ sono stati interni di controllo (i valori di ctl_state)
- $ck(k = 1, 2, ..., n)$ rappresentano le condizioni di input
- R_k le azioni della macchina

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Le ASM sono analoghe alle FSM. Le differenze riguardano:

- La concezione degli stati: nelle FSM esiste un unico stato di controllo (ctl_state), che può assumere valori in un insieme finito di un certo tipo. Nelle ASM lo stato è più complesso
- Le condizioni di input e le azioni di output: nelle FSM è presente un alfabeto finito. Nelle ASM: input qualsiasi espressione, azioni generiche.

Le ASM hanno una notazione matematica in termini di algebra. Useremo AsmetaL come notazione concreta per l'editing di modelli ASM.

Esempio: Una porta girevole opera in modalità singolo bottone. Se la porta è chiusa, premendo il bottone la porta inizia ad aprirsi. Se si preme ancora una volta il bottone prima che la porta si apre del tutto, la porta inizia a chiudersi. Se, invece, permettiamo alla porta di aprirsi del tutto, una volta aperta la porta inizia a chiudersi automaticamente dopo un timeout di 2-secondi. Per impedire la chiusura automatica e mantenere la porta aperta occorre premere il bottone; a questo punto, un'ulteriore pressione del bottone, fa si che la porta inizi a chiudersi. Se si preme ancora una volta il bottone prima che la porta si chiuda del tutto, la porta inizia ad aprirsi.

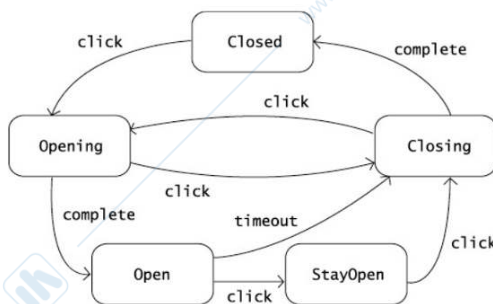


Figura 9: Modello FSM dell'esempio

Il modello ASM che deriva dall'esempio è rappresentato nella Figura sottostante.

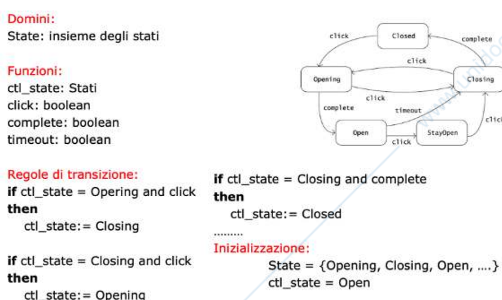


Figura 10: ASM relativo all'esempio

Un altro esempio: Una macchinetta automatica dispensa caffè o latte. La macchinetta accetta solo monete da 50 centesimi e da 1 euro. Se viene inserita una moneta da 50 centesimi, la macchinetta dispensa latte (se disponibile); se viene inserita una moneta da 1 euro, la macchinetta dispensa caffè (se disponibile). Se viene dispensata una bevanda, la sua disponibilità viene decrementata e la moneta viene conservata nella macchinetta. La macchina all'inizio contiene 10 unità per ogni bevanda; l'atto di erogazione di una bevanda corrisponde alla diminuzione di un'unità della disponibilità della stessa e alla conservazione della moneta (nelle monete conservate, non bisogna distinguere tra monete da 50 centesimi ed 1 euro). Se la bevanda non è disponibile, non viene erogata e la moneta non viene conservata. La macchina può contenere al massimo 25 monete; quando la macchina è piena di monete, non accetta altre monete e, quindi,

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

non eroga più alcuna bevanda. All'inizio la macchinetta non contiene alcuna moneta. L'utente del sistema decide ad ogni passo di simulazione il tipo di moneta da inserire.

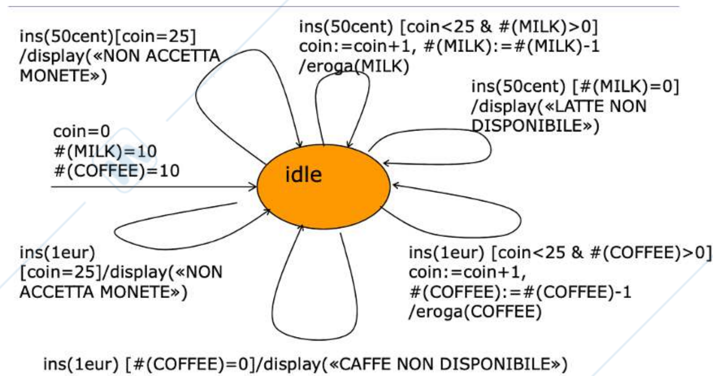


Figura 11: Modello FSM relativo all'esempio

Proviamo adesso a modellare con insiemi, funzioni e regole in ASM.
Segnatura:

- domain CoinType = HALF | ONE
- domain Product = COFFEE | MILK
- domain QuantityDomain subsetof Integer domain CoinDomain subsetof Integer
- available: Product → QuantityDomain
- coins: CoinDomain
- insertedCoin: CoinType

Definitions:

- domain QuantityDomain = 0 .. 10
- domain CoinDomain = 0 .. 25

Stato iniziale s0:

- coins = 0
- available($p in Product) = 10

La dicitura $p in Product significa per tutti i prodotti. La regola principale in ASM è la seguente:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
main rule r_Main =
    if(coins < 25) then
        if(insertedCoin = HALF) then
            if(available(MILK) > 0) then
                r_serveProduct[MILK]
                endif
            else
                r_serveProduct[COFFEE]
            endif
        endif
```

Figura 12: Main rule dell'esempio

Le istruzioni che iniziano con r_ fanno riferimento a regole secondarie. In questo caso, la regola serveProduct è la seguente:

```
rule r_serveProduct($p in Product) =
    par
        available($p) := available($p) - 1
        coins := coins + 1
        endpar
```

Figura 13: Regola serveProduct

2.1 Stato in ASM

Uno stato è un'algebra a più sorti. Nelle ASM gli stati sono associati a un insieme di valori di qualsiasi tipo, memorizzate in apposite locazioni. Uno stato è un insieme di valori, memorizzata in un'apposita locazione. Matematicamente una locazione è definita come la coppia $(f, (v_1, ..., v_n))$ con f nome di funzione e $(v_1, ..., v_n)$ i suoi argomenti. La coppia funzione e argomenti è una locazione, la quale ha un valore che corrisponde al valore della funzione. Con una transizione cambiamo il valore contenuto nella locazione. In uno stato S, una locazione ha un valore: l'interpretazione della funzione nello stato. Una funzione può essere: 0-arie (Variabili) o n-arie (Funzioni). Le transizioni di stato delle FSM corrispondono alle transizioni di stato delle ASM con aggiornamenti dei valori contenuti nelle locazioni. Se loc è $(f, (v_1, ..., v_n))$ ed ha valore a, l'assegnamento ha la forma $f(v_1, ..., v_n) := newval$. L'assegnamento cambia il valore di loc, e quindi di $f(v_1, ..., v_n)$, da a a $newval$.

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 cambi o no da uno stato al successivo. Le funzioni sono intese in senso matematico, non informatico come "procedure". Le funzioni statiche di arietà zero sono dette costanti. Ogni vocabolario ASM contiene sempre le constanti statiche $undef, True, False$. I numeri sono costanti numeriche $(1, 2, ...)$. L'utente può aggiungerne di sue (costante $minimoVoto = 18$). Le funzioni statiche (di arietà > 0) sono definite tramite una legge fissa. Un esempio di funzioni statiche sono le usuali operazioni:

- tra numeri: $+$, -, ...
- tra booleani: AND, OR, ...

Sono "standard", ma l'utente può definirne di sue, ad es.: $max(n, m)$. Le funzioni dinamiche di arietà zero sono le comuni variabili dei linguaggi di programmazione.

Un esempio di vocabolario è il vocabolario Σ_{bool} dell'algebra booleana contiene:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- Due costanti 0 e 1,
- Una funzione unaria di nome '-' (per la negazione) e
- Due funzioni binarie di nomi
  - '+' (per l'OR) e
  - '*' (per l'AND)
- Tutte funzioni statiche

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.

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.

Vediamo adesso due esempi di stato su Σ_{bool}. Entrambi gli stati sono esempi di algebre booleane. La Figura sottostante presenta lo stato A per il vocabolario presentato precedentemente, inteso come un superuniverso booleano 0,1

Stato A per il vocabolario Σ_{bool} :

Il superuniverso dello stato A è X={0, 1}.

Le funzioni sono interpretate come:

0^A := 0 (zero)

1^A := 1 (uno)

$-^A a$:= 1 - a (complemento logico)

a $+^A$ b := max(a, b) (or logico)

a $*^A$ b := min(a, b) (and logico)

(dove a, b sono 0 oppure 1)

Figura 14: Stato A

La Figura sottostante presenta lo stato B per il vocabolario presentato precedentemente, inteso come un superuniverso che rappresenta l'insieme potenza degli interi non-negativi N.

Stato B per il vocabolario Σ_{bool} :

Il superuniverso dello stato B è l' *insieme potenza* degli interi non-negativi N.

le funzioni sono interpretate come: (where a, b subsets of N):

0^B := (empty set)

1^B := N (full set)

$-^B a$:= N \ a (set of all $n \in$ N such that $n \notin a$)

a $+^B$ b := $a \cup b$ (set of all $n \in$ N such that $n \in a$ or $n \in b$)

a $*^B$ b := $a \cap b$ (set of all $n \in$ N such that $n \in a$ and $n \in b$)

Figura 15: Stato B

Nelle applicazioni pratiche, il superuniverso di uno stato ASM è "suddiviso" in universi, rappresentati dalle loro funzioni caratteristiche. Se X è il superuniverso dello stato A, l'universo G di A è rappresentato dalla funzione $G : X \rightarrow Bool$ tale che $G(t) = true$ per tutti gli elementi t del superuniverso X di A che formano la partizione (o sotto insieme) G; $G(t) = false$, altrimenti. Ad esempio: $X = 1, 2, 3, a, b, mario, pippo$ ripartito in domini: $Interi = 1, 2, 3$, $Char = a, b$, $String = mario, pippo$. Ogni universo rappresenta un DOMINIO (set). In base a questa rappresentazione degli insiemi in termini di funzioni caratteristiche, uno stato di una

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

ASM consente di modellare domini eterogeni. I soliti domini predefiniti sono disponibili (Interi, String). L'utente può definirne altri da niente, come tipi astratti, come enumerativi o a partire da altri domini (strutturati). Presentiamo un esempio di stato ASM, relativo ad un orologio:

- Modello ASM: CLOCK real time
- Vocabolario Σ
  - Funzioni dinamiche
    - CurrTime: Real (viene incrementato dall'esterno)
    - DisplayTime: Real
  - Funzioni statiche
    - Delta: Real (intervallo di tempo)
    - + : Real x Real -> Real (addizione)

Figura 16: Esempio Modello ASM

I termini di Σ sono espressioni sintattiche così costruite: 1. Variabili $v_0, v_1, v_2, ...$ sono termini 2. Costanti c of Σ sono termini 3. Se f è un nome di funzione n-aria di Σ e $t_1, ..., t_n$ sono termini, allora $f(t_1, ..., t_n)$ è un termine. Tipicamente i termini sono denotati dalle lettere r, s, t; le variabili dalle lettere x, y, z. Un termine che non contiene variabili è detto chiuso. Consideriamo il vocabolario Σ_{bool}. Alcuni esempi di termini sono: $+(v_0, v_1)$ e $+(1, *(v_7, 0))$. Solitamente, vengono scritti in notazione infissa come: $v_0 + v_1$ e $1 + (v_7 * 0)$. I termini sono oggetti sintattici. Assumono significato (o semantica) nello stato. Il suo valore è l'interpretazione del termine in A. Il termine $v_0 + v_1$ di Σ_{bool} assume significato diverso se interpretato in A con $X = 0, 1$ o in B con X insieme potenza di N.

2.2 Regole di transizione

Le ASM sono metodi formalo per modellare i sistemi. Formalizzano la configurazione istananea di un sistema tramite algebre. Fissare un superuniverso per attribuire un significato ai simboli matematici, una sorta di dizionario. È necessario il superuniverso per dare un'interpretazione all'insieme dei simboli.

In matematica le algebre sono statiche: non cambiano col passare del tempo. Le algebre corrispondo agli stati, ma in Informatica, gli stati sono dinamici: evolvono essendo aggiornati durante le computazioni. Aggiornare stati astratti (abstract states) significa cambiare l'interpretazione delle (o solo di alcune) funzioni della segnatura della macchina. Quindi significa cambiare interpretazione all'algebra. Il modo in cui una macchina ASM aggiorna il proprio stato è descritto da regole di transizione (transitions rules) di una certa "forma". L'insieme delle regole di transizione di una ASM definiscono la sintassi di un programma ASM. Sia Σ un vocabolario, le regole di transizione di una ASM sono espressioni sintattiche su Σ generate attraverso l'uso di costruttori di regole.

Il costruttore base è l'update rule, che consiste in $f(t_1, ..., t_n) := s$, dove:

- f è un nome di funzione dinamica n-aria di Σ
- $t_1, ..., t_n$ e s sono termini di $\sum$

Il significato che assume questa regola è che a partire dallo stato successivo, il valore di f per gli argomenti $t_1, ..., t_n$ è aggiornato a s. Se f è 0-aria, cioè una variabile, l'aggiornamento ha la forma $c := s$. Un esempio è dato da $DispleyTime := CurrTime$, con $DisplayTime$: Real variabile (funzione 0-aria) reale.

Un altro esempio di costruttore è la Conditional Rule: **if** φ **then** R **else** S. Il significato di questa regola è che se φ è vera, allora esegui R, altrimenti esegui S. Un esempio consiste

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

nel seguente: **if** $CurrTime = DisplayTime + Delta$ **then** $DisplayTime := CurrTime$, con $DisplayTime, CurrTime$ e $Delta$ variabili (funzioni 0-arie) reali.

Sia M un ASM (Abstract State Machine) e *env* l'ambiente di M. La Figura sottostante rappresenta la classificazione delle funzioni ASM.

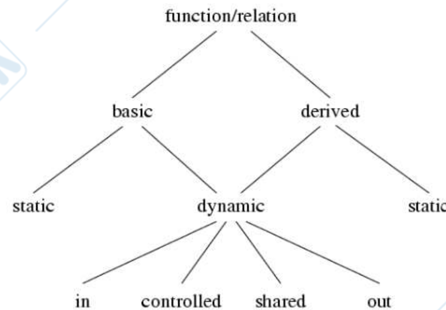


Figura 17: Classificazione delle funzioni ASM

Sia M un ASM e env l'ambiente di M. Esistono due tipologie di funzioni:

- Basic: sono le funzioni elementari, che costituiscono l'alfabeto di una ASM
- Derived: sono funzioni il cui valore in ogni stato è calcolato a partire dalle funzioni di base. I valori computati da funzioni monitorate e funzioni statiche per mezzo di una "legge" o "schema" fissati a priori.

A loro volta le funzioni Basic e Derived possono essere:

- Static: non cambiano mai valore durante l'esecuzione: il loro valore non dipende dallo stato corrente. Le funzioni static di arietà 0 sono le costanti.
- Dynamic: sono funzioni il cui valore dipende dallo stato corrente della ASM. Il loro valore dipende dagli update della ASM o dell'ambiente esterno. Le funzioni dynamic di arietà 0 corrispondono alle variabili dei tradizionali linguaggi di programmazione. Le funzioni Dynamic possono essere:
  - Monitored (o in): possono essere lette, ma non modificate dalla ASM. Sono modificate solo dall'ambiente esterno o da altre ASM, nel caso di sistemi multiagente. In ogni stato deve essere specificato il valore di tutte le funzioni monitored.
  - Controlled: sono modificate solo dalla ASM, mediante l'esecuzione delle regole di transizione, non dall'ambiente né da altre ASM.
  - Shared (o interaction): possono essere modificate da più ASM e dall'ambiente
  - Out: possono essere modificate ma non lette dalla ASM. In genere, sono monitored per l'ambiente e per le altre ASM.

La Tabella 1 mostra tutti i costruttori di regole.

La Skip Rule, la quale ha sintassi **skip**, ha come significato quello di non eseguire nessuna azione. Viene, spesso, utilizzata per completare la regola condizionale.

La Let Rule assegna il valore di t a x e esegue R, se R non ha parametri, realizza il passaggio per valore. **let** $x = t$ **in** R, assegna alla "variabile" x il valore di t. Un esempio è dato da **let** $x = f_1(f_2(f_3(y)))$ **in** $g(x) := 7$.

Il costruttore **par** per la block rule fa sì che R e S siano eseguite in parallelo. Due aggiornamenti paralleli (ipotesi $f(5) = 1$.

$f(5) := 2$

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Tabella 1: 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	**sei** 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 $t_1, ..., t_n$	$r(t_1, ..., t_n)$

$g(3) := f(5)$ – allo stato successivo si avrà $g(3) = 1$, perché $f(5)$ si aggiornerà a partire dallo stato successivo.

La (Macro) Call Rule ($r[t_1, ..., t_n]$) chiama r (regola con parametri) con argomenti $t_1, ..., t_n$. Una definizione di regola per un nome di regola r di arietà n è un'espressione $r(x_1, ..., x_n) = R$, dove R è una regola di transizione. In una call rule $r[t_1, ..., t_n]$ le variabili x_i che occorrono nel corpo R della definizione di r vengono sostituite dai parametri t_i (modularità).

Il costruttore Seq (**seq** R S **endseq**) corrisponde alla composizione sequenziale: 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).

Il costruttore forall corrisponde all'esecuzione di R in parallelo per ogni x che soddisfa φ. Implementa il concetto di parallelismo sincrono (bounded). Il costruttore è indicato nel seguente modo: **forall** x **with** φ **do** R.

Il costruttore choose esegue R in parallelo per un x che soddisfa φ. Implementa il concetto di non-determinismo. Il costruttore è indicato nel seguente modo: **choose** x **with** φ **do** R.

Riassumendo, una abstract state machine M è una terna (Σ, A, R), formata da:

- un vocabolario Σ,
- uno stato iniziale A per Σ,
- un insieme R di nomi di regole con:
  - un nome di regola di arietà zero, la cosidetta main rule (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 di tutti gli aggiornamenti.

Per estendere un sub-universo del superuniverso con nuovi elementi si usa la notazione: **import** x **do** R. Il significato di tale costrutto è: scegli un elemento x da Reserve (dominio predefinito di simboli "freschi"), cancellalo da Reserve, aggiungi x al superuniverso e esegui R. Spesso si preferisce la forma seguente: **extend** U **with** x [**do**] R, come abbreviazione per **import**x **do** $U(x) := true$ **seq** R.

Data la funzione $f(a_1, ..., a_n)$ in uno stato S della macchina:

- La coppia $loc = (f, (a_1, ..., a_n))$ è detta locazione e rappresenta matematicamente il valore di $f(a_1, ..., a_n)$ in memoria.
- Un aggiornamento (o update) è la coppia $(loc, b) = ((f, (a_1, .., a_n)), b)$. Il significato dell'aggiornamento è che l'interpretazione della funzione f in S viene modificata per gli argomenti $a_1, ..., a_n$ con il valore b.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- Un update set è un insieme di aggiornamenti.

A causa del parallelismo (la regola Block e Forall), una regola di transizione può richiedere più volte l'aggiornamento di una stessa funzione per gli stessi argomenti. Si richiede in tal caso che tali aggiornamenti siano consistenti. Un update set U è consistente, se vale la condizione: se $((f,(a_1,...,a_n)),b) \in U$ ed $((f,(a_1,...,a_n)),c) \in U$, allora $b = c$. Se l'update set U è consistente, allora i suoi aggiornamenti possono essere effettivamente eseguiti (fired) in un dato stato. Il risultato è un nuovo stato (di arrivo) dove le interpretazioni dei nomi delle funzioni dinamiche sono cambiati secondo U. Le interpretazioni dei nomi delle funzioni statiche sono gli stessi dello stato precedente (di partenza). Le interpretazioni dei nomi delle funzioni monitorate sono date dall'ambiente esterno e possono dunque cambiare in maniera arbitraria.

2.3 Esempi

2.3.1 Orologio

Prendiamo come esempio un orologio che ad ogni passo incrementa i secondi. La signature sarà la seguente:

- domain Second
- domain Minute
- domain Hour
- controlled seconds : Second
- controlled minutes : Minute
- controlled hours : Hour

Le regole sono le seguenti:

- macro rule IncMinHours

```
IncMinHours
1  par
2  if minutes = 59 then
3       hours := (hours + 1) mod 24
4  else
5       minutes := (minutes + 1) mod 60
```

- main rule Main

```
Main
1  par
2  if seconds = 59 then
3       IncMinHours()
4  else
5       seconds := (seconds + 1) mod 60
```

Lo stato iniziale è il seguente:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

default init s_0:

function seconds = 0

function minutes = 0

function hours = 0

Passare dalla notazione matematica alla notazione concreta può richiedere alcune precisazioni:

- Signature:

  domain Second subsetof Integer
  domain Second = {0..59}
  domain Minute subsetof Integer
  domain Minute= {0..59}
  domain Hour subsetof Integer
  domain Hour = {0..23}
  controlled seconds :Second
  controlled minutes :Minute
  controlled hours :Hour

2.3.2 Orologio 2

Incrementa l'ora (mod 3) all'arrivo di un segnale. Stessa segnatura di prima, ma:

domain Second = {0..3}

domain Minute = {0..3}

domain Hour = {0..3}

Inoltre, monitored signal: Boolean è inizializzata a false:

deafult init s_1

function signal = false

Regole:

- macro rule IncMinHours

```
IncMinHours
1  par
2  if minutes = 2 then
3       hours := (hours + 1) mod 3
4  else
5       minutes := (minutes + 1) mod 3
```

- main rule AdvancedClock

```
Main
1  if signal then
2       par
3       if seconds = 2 then
4            IncMinHours()
5       else
6            seconds := (seconds + 1) mod 3
```


Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

2.3.3 Safety Injection System

Si tratta di un Sistema di controllo per safety injection [P.J.Coutois and D.L.Parnas 1993]. Monitora la pressione dell'acqua e l'iniezione di liquido refrigerante nel reattore quando la pressione scende al di sotto di una certa soglia minima. L'operatore di sistema può bloccare questo processo premendo l'interruttore Block. Il sistema viene resettato tramite l'interruttore Reset.

Segnatura:

- Monitored Variables

  Block : {off,on}

  Reset : {off,on}

  WaterPressure: {0..2000}

- Controlled Variables

  Pressure : {TooLow, Normal, High}. Necessaria per specificare i modi in cui può trovarsi il sistema in base ai valori di WaterPressure.

  Overridden : {true, false}. true se safety injection è bloccato, false se non bloccato.

  SafetyInjection : {on,off};

- Static Variables

  Low = 900

  Permit = 1000

Regole:

- R1: WaterPressure diventa maggiore di Low, quindi Pressure da TooLow a Normal

```
1 if WaterPressure ≥ Low  &  Pressure = TooLow then
2       Pressure := Normal
```

- R2: Pressure da Normal ad High

```
1 if WaterPressure ≥ Permit  &  Pressure = Normal then
2       Pressure := High
3       Overriden := false
```

- R3: Pressure da Normal a TooLow

```
1 if WaterPressure < Low  &  Pressure = Normal then
2       Pressure := TooLow
```

- R4: Pressure da High a Normal

```
1 if WaterPressure < Permit  &  Pressure = High then
2       Pressure := Normal
3       Overriden := false
```


Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- R5: Il controller resetta il sistema

```
1 if Reset = on & (Pressure = TooLow|Pressure = Normal) then
2     Overriden := false
```

- R6: Il controller sovrascrive il sistema

```
1 if Block = on & Pressure = TooLow & Reset = off then
2     Overriden := true
```

- R7 + R8: Quando Pressure è TooLow, SafetyInjection è On, a meno che Overriden sia true

```
1 if Pressure = TooLow then
2     if Overriden then
3         SafetyInjection := off
4     else
5         SafetyInjection := on
```

- R9: Quando Pressure è Normal o High, SafetyInjection è sempre off

```
1 if Pressure != TooLow then
2     SafetyInjection := off
```

2.3.4 Sistema di controllo del semaforo a senso unico

Un sistema costituito da una coppia di semafori (LIGHTUNIT1 e LIGHTUNIT2) posti alle estremità di una via a senso unico alternato, collegati a un computer che li controlla. Ogni semaforo è dotato di una luce di Stop (luce rossa) ed una luce di Go (luce verde). Il computer accende/spegne le luci inviando ai semafori i segnali Rpulses e Gpulses. Lo stato delle luci dei due semafori varia seguendo un ciclo in quattro fasi:

- per 50 secondi tutte e due i semafori mostrano il segnale di Stop
- per 120 secondi il semaforo LIGHTUNIT1 mostra il segnale di Stop e LIGHTUNIT2 il segnale di Go
- per 50 secondi tutti e due i semafori mostrano nuovamente il segnale di Stop
- per 120 secondi il semaforo LIGHTUNIT2 mostra il segnale di Stop e LIGHTUNIT1 il segnale di Go

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

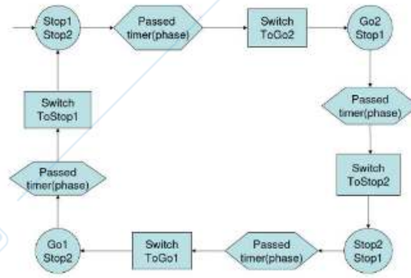


Figura 18: Control State ASM

Segnatura:

- enum domain LightUnit = {LIGHTUNIT1 | LIGHTUNIT2}
- enum domain PhaseDomain = {STOP1STOP2 | GO2STOP1 | STOP2STOP1 | GO1STOP2}
- domain Intervals subsetof Integer
  Rappresenta i periodi di durata dei segnali di stop e di go: domain Intervals = {50 , 120}
- dynamic controlled phase : PhaseDomain
- dynamic controlled stopLight : LightUnit → Boolean
- dynamic controlled goLight : LightUnit → Boolean
  Rappresentano le luci di stop e go del semaforo
- dynamic monitored passed : Intervals → Boolean

Regole:

- Rule stop1stop2_to_go2stop1

STOP1STOP2_TO_GO2STOP1
1 **if** phase=STOP1STOP2 & *passed*(50)) **then**
2 **par**
3 SWITCHLIGHTUNIT(LIFHTUNIT2)
4 phase := GOSTOP1
5 **endpar**

- Rule switchLightUnit(I in LightUnit)

SWITCHLIGHTUNIT(I IN LIGHTUNIT)
1 **par**
2 switch (goLight (l))
3 switch (stopLight (l))
4 **endpar**

- Macro rule switch (I in Boolean)

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
SWITCH(I IN BOOLEAN)
1  I := not(I)
```

- rule stop1stop2_to_go2stop1

```
STOP1STOP2_TO_GO2STOP1
1  if phase = STOP1STOP2 & passed(50) then
2        par
3        SWITCHLIGHTUNIT(LIGHTUNIT2)
4        phase := GO2STOP1
5        endpar
```

- Rule go2stop1_to_stop2stop1

```
GO2STOP1_TO_STOP2STOP1
1  if phase = GO2STOP1 & passed(120) then
         par
2        SWITCHLIGHTUNIT(LIGHTUNIT2)
3        phase := STOP2STOP1
4        endpar
```

- Rule stop2stop1_to_go1stop2

```
STOP2STOP1_TO_GO1STOP2
1  if phase = GO1STOP2 & passed(120) then
2        par
3        SWITCHLIGHTUNIT(LIGHTUNIT1)
4        phase := STOP1STOP2
5        endpar
```

- Main rule main

```
MAIN
1  par
2  STOP1STOP2_TO_GO2STOP1()
3  GO2STOP1_TO_STOP2STOP1()
4  STOP2STOP1_TO_GO1STOP2()
5  GO1STOP2_TO_STOP1STOP2()
6  endpar
```

2.3.5 SluiceGate

Sistema di irrigazione che controlla il flusso dell'acqua tramite una chiusa. La chiusa è sbarrata da una saracinesca che si può alzare ed abbassare. La saracinesca deve essere totalmente aperta per 10 minuti ogni tre ore; nel resto del tempo deve essere totalmente chiusa. Dei sensori segnalano se è completamente chiusa o completamente aperta. La saracinesca viene aperta/chiusa ruotando delle viti che sono manovrate da un motore. Al motore vengono inviati 4 segnali:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- per ruotare le viti in senso orario,
- per ruotarle in senso antiorario,
- per accenderlo e
- per spegnerlo

Il computer controlla il sistema emettendo i quattro segnali per guidare il motore; il computer è collegato ai due sensori posti alla estremità del percorso della saracinesca per conoscerne la posizione.

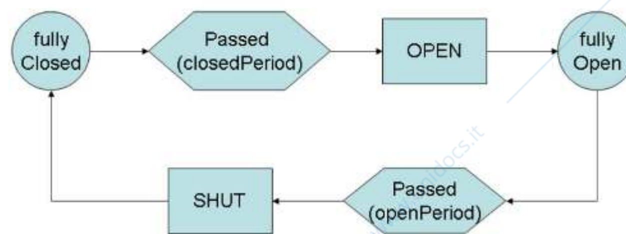


Figura 19: Control State ASM

Segnatura:

- domain Minutes subsetof Integer
- enum domain PhaseDomain = {FULLYCLOSED | FULLYOPENED}
- domain Minutes = {10 , 170}
- dynamic controlled phase : PhaseDomain
- dynamic monitored passed : Minutes → Bool
- domain Minutes = {10 , 170}. passed(10) e passed(170) indicano se è trascorso l'intervallo di tempo in cui la saracinesca deve essere completamente aperta e (risp.) completamente chiusa.

Regole:

- main rule main

```
1  par
2  if phase = FULLYCLOSED then
3      if passed(170) then
4          par
5          OPEN()
6          phase := FULLYOPENED
7          endpar
8  if phase = FULLYOPENED then
9      if passed(10) then
10         par
11         SHUT()
12         phase := FULLYCLOSED
13         endpar
```


Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

2.3.6 Fattoriale

Segnatura:

- dynamic monitored valore: Integer
- dynamic controlled indice: Integer
- dynamic controlled fattoriale: Integer

Regole:

- macro rule fattoriale

```
FATTORIALE
1  if indice > 1 then
2      seq
3      fattoriale := fattoriale*indice
4      indice := indice - 1
5      endseq
```

- main rule main

```
MAIN
1  seq
2  if indice = 1 then
3      if valore > 0 then
4          par
5          indice := valore
6          fattoriale := 1 FATTORIALE() endseq
```

2.3.7 Fattoriale con stop

Segnatura:

- dynamic monitored valore: Integer
- dynamic controlled indice: Integer
- dynamic controlled fattoriale: Integer
- dynamic controlled stop: Boolean

Inizializzazione: Default init s_0:

function indice = 1

function stop = false

Regole:

- macro rule fattoriale

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
FATTORIALE
1  if indice > 1 then
2      seq
3      fattoriale := fattoriale * indice
4      indice := indice - 1
5      if indice = 1 then
           stop := true
6          endseq
```

- main rule main

```
MAIN
1  if stop = false then
2      seq
3      if indice = 1 then
4          if valore > 0 then
5              par
6              indice := valore
7              fattoriale := 1
8      FATTORIALE()
9      endseq
```

2.3.8 Ordinare un vettore

```
SWAP($X IN INTEGER, $Y IN INTEGER)
1  par
2  $x := $y
3  $y := $x
4  endpar
```

```
SWAPSORT
1  choose $i in {0..9}, $j in {0..9} with $i < $j and vect($i) > vect($j) do
2  SWAP(VECT($I), VECT($J))
```

Non occorrono variabili di appoggio per lo swap, perché i nuovi valori degli aggiornamenti saranno disponibili solo nello stato successivo.

2.3.9 Operazioni sullo stack - Esempio di riserva

Segnatura:

- domain POS
- domain ELEM
- controlled next: POS → POS
- controlled head: POS
- controlled stack: POS → ELEM

Regole

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
ADD(ELEM)
•1  extend POS with pos do {
 2  next(pos):= head
 3  head:= pos
 4  stack(pos):= Elem }
```

```
REMOVE
•1  if head ≠ undef then
 2  else
 3       errorMessage
```

2.3.10 Esempio su aggiornamenti inconsistenti

Segnatura:

abstract domain Orders

enum domain Status = {PENDING | INVOICED | CANCEL}

static o1: Orders

static o2: Orders

static o3: Orders

controlled orderStatus: Orders → Status

Default init s_0:
function orderStatus($o in Orders) = PENDING
Regole:

```
MAIN
1  par
2  choose $o in Orders with true do orderStatus($o) := INVOICED
3  choose $oo in Orders with true do orderStatus($oo) := CANCEL
4  endpar
```

3 Asmeta

Asmeta è un framework per il metodo formale ASM (Abstract State Machines). É composto da diversi strumenti per eseguire diverse attività di convalida e verifica. Sono stati sviluppati i seguenti strumenti (stabili):

- Asmee: un editor per ASM scritto in lingua AsmetaL. Asmeta fornisce un editor integrato in Eclipse, per effettuare simulazioni e type checking.
- AsmetaLc: un compilatore/parser per i modelli AsmetaL
- AsmetaS: un simulatore
- AsmetaV: un validatore basato sullo scenario
- AsmetaA: un animatore di esecuzione
- AsmetaSMV: un modello di controllo basato su NuSMV, che permette di effettuare operazioni di verifica.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- AsmetaMA: permette la revisione del modello
- AsmetaVis: un visualizzatore grafico di modelli AsmetaL
- AsmetaRefProver: un prover della correttezza della raffinatezza
- Asm2SMT: un traduttore dai modelli AsmetaL ai contesti logici di Yices
- Asm2C ++: un generatore di codice

Alcuni strumenti sono forniti sia come plug-in eclipse che come barattoli autonomi che possono essere eseguiti dalla riga di comando. Alcuni sono disponibili solo come file jar.

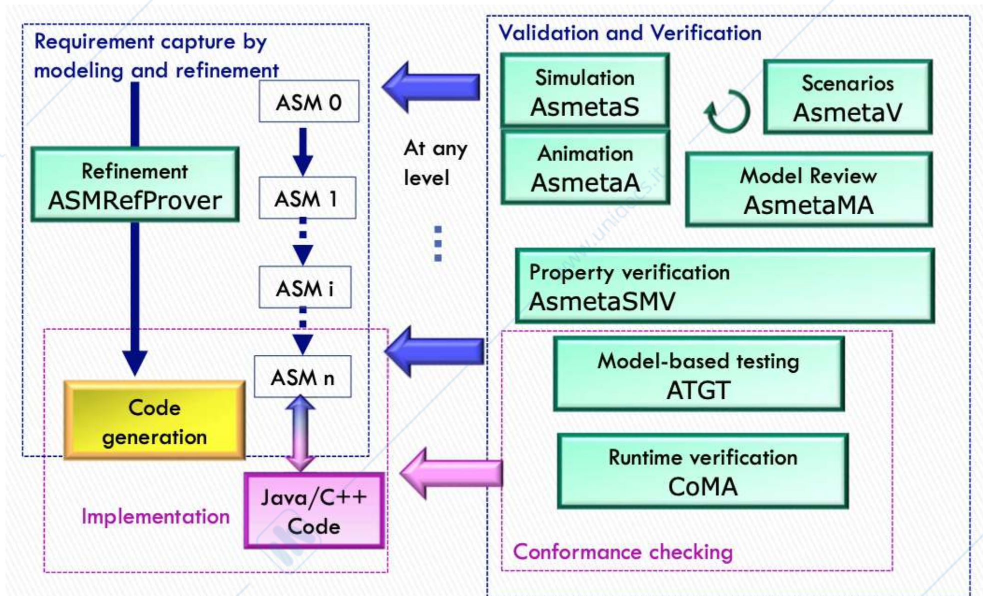


Figura 20: ASMETA-based Design Process

Riassumendo, il set di tool ASMETA è stato sviluppato a partire dalla definizione di AsmM, un metamodello per ASM. Il set di strumenti ASMETA include una sintassi testuale AsmetaL, per scrivere modelli ASM; AsmetaLc, per analizzare i modelli AsmetaL e verificarne la coerenza rispetto ai vincoli AsmM OCL; un simulatore, AsmetaS, per eseguire modelli ASM; il linguaggio Avalla per la validazione basata su scenari di modelli ASM, con il suo strumento di supporto, il validatore AsmetaV; il correttore modello SPIN; un front-end grafico chiamato ASMEE (ASM Eclipse Environment) che funge da IDE ed è un plug-in eclipse. Asmeta è un insieme di tool basati sulla piattaforma Eclipse, composto principalmente da tre fasi:

- Model editing e formato di interscambio
- Validation (Convalida):
  - Simulazione
  - Costruzione di scenario
  - Testing model-base
- Verifica:
  - Analisi statica
  - Model checking

Infine, si può definire Asmeta come un insieme di tool sviluppati con approccio meta-modeling della Model-driven Engineering (MDE).

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

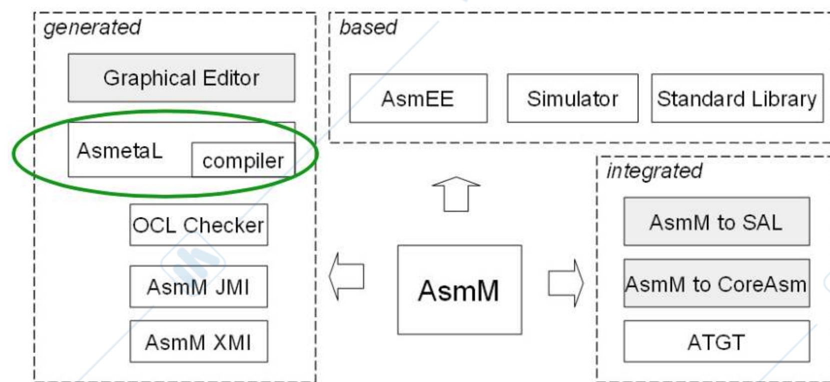


Figura 21: Architettura ASMETA

Tutti gli strumenti sopra elencati sono classificati in: generati, basati e integrati. Gli artefatti/strumenti generati sono derivati ottenuti (semi-)automaticamente, applicando le proiezioni MOF appropriate agli spazi tecnici Javaware, XMLware e grammarware. Gli artefatti/strumenti basati sono quelli sviluppati esplorando l'ambiente ASMETA e i relativi derivati. Gli artefatti/strumenti integrati sono strumenti esterni ed esistenti collegati all'ambiente Asmeta.

3.1 AsmetaL

Un modello ASM può essere letto come pseudocodice su strutture dati astratte. AsmetaL è un linguaggio formale che permette di descrivere un sistema che si vuole modellare, a tal fine AsmetaL include:

- Linguaggio Strutturale, che consente di descrivere la struttura di una macchina ASM. Il linguaggio strutturale consiste in costrutti per definire la struttura (scheletro) di una ASM mono-agente o sinc./asinc. multi-agente.
- Linguaggio delle Definizioni, il quale permette di definire domini (che rappresentano i tipi dei dati introdotti), funzioni (che posseggono domini e codomini, f: Integer → Integer), regole di transizione e invarianti.
- inguaggio dei Termini, che consente di specificare dei termini detti di base (variabili, costanti, termini di funzione) e termini speciali i quali possono essere associati a tuple o collezioni di dati (insiemi, sequenze, bag, mappe, eccetera);
- Linguaggio delle regole, il quale permette di specificare le regole che permettono alla macchina ASM di cambiare stato. Anche in questo caso si distinguono Regole di Base (skip, update, par, etc.) e Turbo Regole (seq, iterate, etc.).

3.1.1 Linguaggio strutturale

Una ASM è una tupla che si compone di: (name, header, body, main rule, initialization):

- name: è il nome della ASM, prima di iniziare a descrivere ogni funzione, regola della macchina ci dovrà essere questa dichiarazione asm Nome_macchina_ASM, che dovrà coincidere con il nome del file. Un modulo ASM è come una ASM senza main rule e senza inizializzazione. Nel definire un modulo ASM si usa la parola chiave module anzichè asm.
- header: si effettua la dichiarazione di domini e funzioni (e non la definizione). Quindi, contiene la dichiarazione della segnatura e l'import/export di moduli/componenti. Ad esempio, StandrdLibrary.asm è un modulo che contiene la definizione dei domini standard:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

  - Naturali, Integer, ...
  - Funzioni standard (+, -, ...)

  Quindi l'header è:

  import *StandardLibrary*

  signature:

  [dom_ declarations]

  [fun_ declarations]

- body: si effettuano le definizioni di domini (in particolare domini concreti statitici) e funzioni statiche. Inoltre, si specifica la dichiarazione delle regole.

  definitions:

  domain $D_1 = Dterm_1$

  ...

  function F_1 *[*$(p_{11}$ *in* d_{11}, ..., p_{1k1} *in* $d_{1k1} = Fterm_1$

  [rule_ declarations]

  [invariant declarations]

  Possono essere definiti solo domini concreti statici e solo funzioni statiche e derivate.

- main rule: è il punto di partenza della macchina, essa può essere considerata come una macro regola chiusa, ovvero una regola senza parametri;

  main rule $R = rule$, dove R è il nome della main rule. *rule* è proprio il corpo della regola di transizione. La main rule è sempre una (macro-)regola chiusa, cioè senza parametri.

- initialization: si esprimono una serie di stati iniziali, dove i domini definiti del body vengono inizializzati. Uno stato iniziale definisce un valore iniziale per domini e funzioni dichiarate nella segnatura.

  [default] init I_d *:*

  domain $D_{d1} = Dterm_{11}$

  ...

  function $F_{d1}[(p_{11}$ ***in*** d_{11}, ... , p_{1s1} ***in*** $d_{1s1})] = Fterm_{d1}$

  ...

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

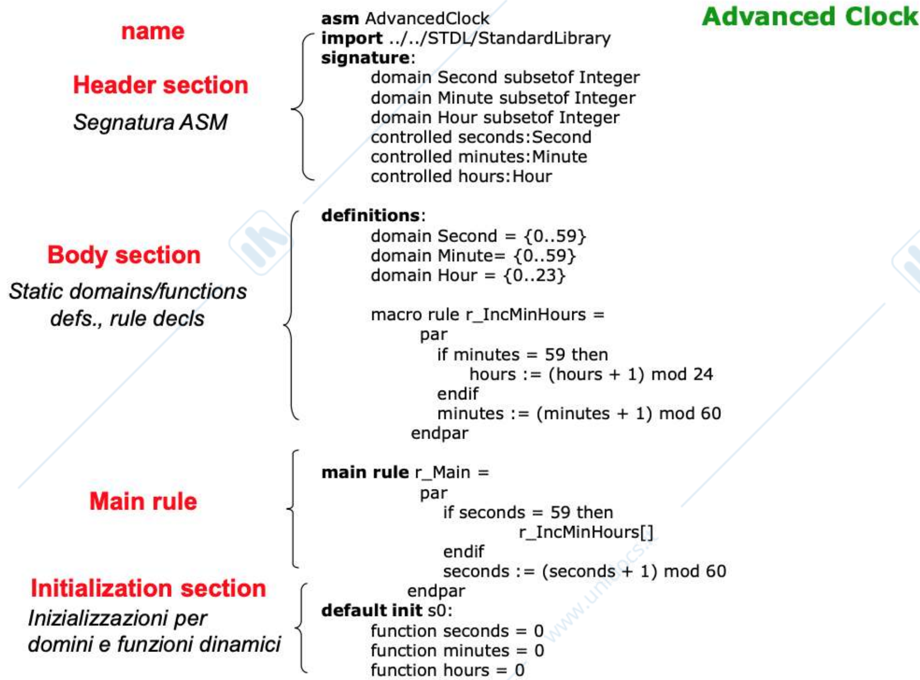


Figura 22: Un esempio di programma AsmetaL

3.1.2 Linguaggio delle definizioni

Specifica i costrutti per definire:

- Funzioni: La Tabella 2 descrive la dichiarazione di funzioni, dove D e C sono rispettivamente il dominio e il codominio della funzione. Il dominio è opzionale se la funzione è di arietà 0 (cioè una variabile).

Costrutto	Sintassi
Funzione Statica	static **f:** [**D** $\rightarrow$] **C**
Funzione Dinamica	*dynamic* monitored **f:** [**D** $\rightarrow$] **C** *dynamic* controlled **f:** [**D** $\rightarrow$] **C** *dynamic* shared **f:** [**D** $\rightarrow$] **C** *dynamic* out **f:** [**D** $\rightarrow$] **C** *dynamic* local **f:** [**D** $\rightarrow$] **C**
Funzione Derivata	derived **f:** [**D** $\rightarrow$] **C** height

Tabella 2: Dichiarazioni di funzioni

- Domini. Esiste una caratterizzazione dei domini:
  - type-domain: caratterizzano il super universo:
    * basic type-domains: Complex, Real, Integer, Natural, String, Char, Boolean, Rule, e Undef definiti nella standard library
    * structured: per costruire insiemi finiti, sequenze, bag, mappe, e tuple a partire da altri domini
    * enum: enumerazioni, ad esempio Color = RED, GREEN, BLUE. Forma standard: ***enum domain*** D = **{**EL_1 / ... / EL_n**}**

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

    * abstract: elementi di natura "astratta", non definiti se non attraverso funzioni definite su tale dominio. Agent e Reserve definiti nella standard library. Forma standard: ***abstract domain*** *D*, dove D è il nome del type domain.

  - concrete domain: user-defined e subset dei type- domain. La forma per la dichiarazione di domini concreti è data da: *[**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.

Inoltre nel linguaggio delle definizioni sono presenti operatori per costruire domini complessi:

  - ProductDomain: ***Prod*** ***(***d_1, d_2, ..., d_n***)***, dove d_1, ..., d_n sono i domini del prodotto cartesiano
  - SequenceDomain: ***Seq*** ***(*** *d* ***)***, dove *d* è il dominio base delle possibili sequenze
  - PowersetDomain: ***Powerset*** ***(*** *d* ***)***
  - BagDomain: ***Bag*** ***(*** *d* ***)***
  - MapDomain ***Map*** ***(*** d_1, d_2 ***)***

- Regole di transizione: la forma standard di una regola è *[**macro**]* ***rule*** *R* *[**(** x_1 **in** D_1, ..., x_n **in** D_n **)**]* = *rule*, dove:
  - R è il nome della regola
  - x_i sono i possibili parametri (opzionali)
  - D_i sono i domini che fanno da tipi per i parametri
  - rule è un costruttore di regola (vedi linguaggio delle regole)
  - la parola chiave macro è opzionale

- Invarianti: le convenzioni sugli ID sono le seguenti:
  - ID_VARIABILE: una stringa che inizia con "$" ($x, $abc, $pippo).
  - ID_ENUM: una stringa di lunghezza ≥ 2, fatta di sole lettere maiuscole (ON, OFF, RED).
  - ID_DOMAIN: una stringa che inizia con una lettera maiuscola (Integer, X, SetOfBags, Person, ...).
  - ID_RULE: una stringa che inizia con r_ (r_setMyPerson, r_update).
  - ID_FUNCTION: una stringa che inizia con una lettera minuscola diversa da "r_" e da "inv_" (plus, minus, continue, foo, ..).

- Commenti: esistono due formi per esprimere commenti nel testo, attraverso i simboli // oppure racchiudere il testo tra: /* e */.

3.1.3 Linguaggio dei termini

I termini si differenziano in:

- Termini di base: come nella logica del primo ordine (costanti, variabili, termini funzionali $f(t_1, t_2, .., t_n)$)

- Termini speciali come tuple, collezioni (insiemi, sequenze, bag, mappe), ecc.

La Tabella 3 rappresenta la modalità di rappresentazione delle costanti. Mentre, la tabella 4 rappresenta la modalità di rappresentazione delle variabili.

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari
Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Costanti	**Sintassi**
Complex Term	$x + iy$ $x - iy$ dove x e y sono numeri reali e i l'immagine
Real Term	Letterale numerico in notazione floating point
Integer Term	Letterale numerico con o senza segno
Natural Term	Letterale numerico seguito da "n"
Char Term	Un carattere delimitato da una coppia di apici
String Term	Una stringa racchiusa tra doppi apici
Boolean Term	**true false**
Undef Term	**undef**
Enum Term	**e** elemento di un dominio enumeration della segnatura

Tabella 3: Il linguaggio dei termini

Variabili e termini funzioni	Sintassi
Variable Term	**V** con v nome di variabile (preceduto da $)
Function Term	*[id .] f [*$(t_1, ..., t_n)$*]* dove: f è il nome della funzione da applicare $(t_1, ..., t_n)$ una tupla di termini *id* è il riferimento all'agente (se presente) che detiene la funzione f
Location Term	É come un Function Term, con f nome di funzione dinamica, ma non monitorata

Tabella 4: Il linguaggi dei termini - Definizione di varabili

La Figura 23 mostra il linguaggio dei termini, applicato alle applicazioni.

Collezioni	Sintassi
Sequence Term	$[t_1,...,t_n]$ con t_i termini omogenei $[]$: sequenza vuota
Set Term	$\{t_1,...,t_n\}$ con t_i termini omogenei $\{\}$: insieme vuoto
Bag Term	$<t_1,...,t_n>$ con t_i termini omogenei $<>$: bag vuoto
Map Term	$\{t_1->s_1,...,t_n->s_n\}$ con t_i termini omgenei, s_i termini omogenei $\{->\}$: mappa vuota
INTERVALLI per sequenze, insiemi e bag di numeri: $[t_{low}..t_{upp},s]$ $\{t_{low}..t_{upp},s\}$ $<t_{low}..t_{upp},s]>$ dove: t_{low} and t_{upp} sono inumeri iniziali e finali ; S opzionale ed indica il passo (se omesso, $s=1$)	

Figura 23: Linguaggio dei termini - Collezioni

La Figura 24 mostra il linguaggio dei termini, nei casi in cui si vogliano utilizzare termini speciali

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Termini speciali	Sintassi
Domain Term	D con D ID di un dominio concreto/type-domain, o termine rappresentante un dominio strutturato
RuleAsTerm	<<R>> con R regola
Tuple Term	$(t_1,\dots,t_n)$ dove t_i sono termini anche di natura diversa. La tupla vuota non esiste.
Conditional Term	if G then t_{then} [else t_{else}] endif dove G è un termine booleano, t_{then} e t_{else} sono termini della stessa natura
LetTerm	let(v_1=t_1,...,v_n=t_n)in $t_{v1,\dots,vn}$ endlet dove v_i sono variabili e $t_1,\dots,t_n,t_{v1,\dots,vn}$ sono termini
Case Term	switch t case t_1 : s_1 ... case t_n : s_n [otherwise s_{n+1}] endswitch dove t, t_i sono termini della stessa natura, e s_j sono termini della stessa natura pure.

Figura 24: Linguaggio dei termini - Termini Speciali

Infine, la Figura 25 mostra il linguaggio dei termini, applicato a termini quantitativi

Quantification Term	Sintassi
ExistTerm	(exist v_1 in D_1,...,v_n in D_n with $G_{v1,\dots,vn}$)
ExistUnique Term	(exist unique v_1 in D_1,...,v_n in D_n with $G_{v1,\dots,vn}$)
ForallTerm	(forall v_1 in D_1,...,v_n in D_n with $G_{v1,\dots,vn}$)
dove: – $v_1,\dots,v_n$ sono variabili, – D_i sono termini che rappresentano i domini dove variano le variabili, – $G_{v1,\dots,vn}$ **è un termine booleano che rappresenta la condizione**	

Figura 25: Linguaggio dei termini - Quantification Term

3.1.4 Linguaggio delle regole

Il linguaggio delle regole definisce la sintassi per:

- regole di base come skip, update, parallel block, ecc.
- turbo regole come seq, iterate, turbo submachine call, ecc.

La Figura 26 presentata sotto fornisce alcuni esempi di regole.
La Figura 27 rappresenta il linguaggio delle regole, per le regole derivate.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Regola	Sintassi
Skip Rule	`skip`
Update Rule	`L := t` dove t è un termine e L (detta locazione) è o un termine funzionale f(t1,..,tn) con f dinamica e non monitorata, o è una variabile
Conditional Rule	`if G then` R_{then}`[else` R_{else}`] endif` dove G è un termine che rappresenta la condizione booleana (detta guardia), e R_{then}, R_{else} sono regole

LetRule	`let (`v_1 `=` t_1`, ...,` v_n `=` t_n`) in`
	$R_{v1,...,vn}$
	`endlet`
	dove $v_1,\ldots,v_n$ sono variabili, $t_1,\ldots,t_n$ sono termini, e $R_{v1,\ldots,vn}$ è una regola

Block Rule	`par` R_1 R_2 `...` R_n `endpar` dove $R_1,R_2,\ldots,R_n$ sono regole
Seq Rule	`seq` R_1 R_2 `...` R_n `endseq` dove $R_1,R_2,\ldots,R_n$ sono regole
Forall Rule	`forall` v_1 `in` D_1`, ...,` v_n `in` D_n `with` $G_{v1,...,vn}$ `do` $R_{v1,...,vn}$ dove v_i sono variabili, D_i termini che rappresentano domini, $G_{v1,\ldots,vn}$ termine booleano che rappresenta la condizione, e $R_{v1,\ldots,vn}$ è una regola
Choose Rule	`choose` v_1 `in` D_1`, ...,` v_n `in` D_n `with` $G_{v1,...,vn}$ `do` $R_{v1,...,vn}$ `[ifnone R]` v_i sono variabili, D_i termini che rappresentano domini, $G_{v1,\ldots,vn}$ termine booleano che rappresenta la condizione, e $R_{v1,\ldots,vn}$ e R sono regole

Macro CallRule	`r[`t_1`,...,`t_n`]`
	dove r è il nome della regola e t_i sono termini che rappresentano gli effettivi argomenti passati
	`r[]`
	per chiamare una regola che è senza parametri

Extend Rule	`extend D with v1,...,vn do`
	`Rv1,...,vn`
	dove D è il nome di un abstract type-domain da estendere, v_i sono variabili legate ai nuovi elementi importati dalla riserva, e R è una regola

Figura 26: Linguaggio delle regole

Regola	Sintassi

Case Rule	`switch t`
	`case` t_1 `:` R_1 `... case` t_n `:` R_n
	`[otherwise` R_{n+1}`]`
	`endswitch`
	dove:
	$t,t_1,\ldots,t_n$ sono termini, e
	$R_1,\ldots,R_n,R_{n+1}$ sono regole

Figura 27: Linguaggio delle regole (derivate)

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

3.2 Simulatore: AsmetaS

Asmetas è un simulatore, le cui caratteristiche sono:

- Axiom checker: Se un assioma viene violato, AsmetaS lancia eccezione InvalidAxiomException
- Consistent Updates checking: In caso di update inconsistenti, AsmetaS lancia l'eccezione UpdateClashException che tiene traccia della coppia di locazioni oggetto dell'inconsistenza
- Random simulation: per mezzo di un ambiente random per le funzioni monitorate

Il simulatore si può utilizzare dopo il parser (spunta verde nella barra degli strumenti). Il simulatore corrisponde al simbolo del triangolo bianco su sfondo viola (quello a sinistra è il simulatore interattivo, mentre quello a destra genera valori randomici).

3.3 Animator: AsmetaA

L'animatore corrisponde al simbolo dell'A nera su sfondo bianco. Permette la simulazione del sistema, con una visone d'insieme agli stati, alle variabili e ai valori di funzione.

3.4 Visualizer: AsmetaVis

Il visualizer permette di visualizzare un modello in termini di una foresta navigabile di tree structures. Applica:

- Structural patterns: utile per visualizzare la struttura del modello in modo compatto
- Semantic patterns: utile quando altre informazioni sul workflow della. Le ASM possono essere inferiti dal modello. Consente la visualizzazione del modello come una finite state machine: è possibile definire (o inferire) una funzione mode che cambia valore nello stato

La Figura 28 presenta la notazione grafica del visualizer AsmetaVis.

Rule	Visual tree	AsmetaL notation
Skip rule do nothing	skip	skip
Update rule update f to v	f := v	f := v
Macro call rule	r_rule[]	r_rule[]
invoke rule *r_rule* with arguments $\overline{v}$ (if any)	r_rule[$\overline{v}$]	r_rule[$\overline{v}$]
Conditional rule execute *rule1* if *guard* holds, otherwise execute *rule2* (if given)	guard → vis_T (rule1)	if guard then rule1 endif
	guard —true→ vis_T (rule1); —false→ vis_T (rule2)	if guard then rule1 else rule2 endif
Block rule execute *rule1* ... *rulen* in parallel	par → vis_T (rule1), vis_T (rule2), ..., vis_T (rulen)	par rule1 rule2 ... rulen endpar
Forall rule execute *rule1* with all values $\overline{v} \in \overline{V}$ for which d($\overline{v}$) holds	forall —$\overline{v} \in \overline{V}$→ d($\overline{v}$) → vis_T (rule1[$\overline{v}$])	forall $\overline{v} \in \overline{V}$ with d($\overline{v}$) do rule1[$\overline{v}$]
Choose rule execute *rule1* with a $\overline{v} \in \overline{V}$ for which d($\overline{v}$) holds. If no such $\overline{v}$ exists, execute *rule2* (if given)	choose —$\overline{v} \in \overline{V}$→ d($\overline{v}$) → vis_T (rule1[$\overline{v}$])	choose $\overline{v} \in \overline{V}$ with d($\overline{v}$) do rule1[$\overline{v}$]
	choose —$\overline{v} \in \overline{V}$→ d($\overline{v}$) → vis_T (rule1[$\overline{v}$]); —ifnone→ vis_T (rule2)	choose $\overline{v} \in \overline{V}$ with d($\overline{v}$) do rule1[$\overline{v}$] ifnone rule2
Let rule execute *rule1* substituting $\overline{t}$ for $\overline{x}$	let —$\overline{x} = \overline{t}$→ vis_T (rule1[$\overline{x}$])	let($\overline{x} = \overline{t}$) in rule1[$\overline{x}$] endlet

Figura 28: Notazione grafica

Su Eclipse è rappresentato dai simboli BV (Basic Visualizer) e SV (Semantic Visualizer).

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

4 Prime tecniche di analisi

Quando si parla di analisi, esistono due approcci:

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

La validazione è meno human-intensive e impegnativa livello operazionale della verifica. La validazione dovrebbe precedere la verifica, per poter individuare errori il prima possibile ed evitare di provare proprietà corrette su specifiche incorrette.Sono possibili due prime forme di analisi sul ground model:

- Garanzia degli invarianti
- Validazione tramite scenari

In un modello ASM gli invarianti sono usati per esprimere vincoli (proprietà che devono sempre essere soddisfatte) su funzioni e/o regole che devono essere garantiti in ogni stato. L'invariante è una combinazione logica e ad ogni step l'algebra che ottengo deve soddisfare l'invariante. 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.

Gli invarianti devono essere dichiarati nelle definizioni e vanno dichiarati subito prima della main rule. Ogni invariante è dichiarato mediante la keyword ***invariant*** che precede il nome attributo dell'assioma. Le funzioni e regole su cui è espresso il vincolo vanno listate dopo la keyword ***over***: ***invariant over*** *id_function, ..., id_rule : term.*

Abstract domain è un insieme matematico astratto. Nella segnatura vanno dichiarati gli elementi dell'insieme astratto. Ad esempio:

abstract domain *Position*

static top: Position

static bottom: Position

Quindi l'insieme astratto contiene gli oggetti top e bottom.

Le funzioni statiche vanno definite nelle definizioni, e non nello stato iniziale, perché vanno dichiarate una sola volta. Le funzioni controllate, invece, possono essere inizializzate nello stato iniziale.

Le tecniche di validazione sono:

- Generazione di scenari,
- sviluppo di prototipi,
- animazione,
- simulazione,
- testing

Lo scenario è una descrizione di un possibile comportamento del sistema. É un'interazione osservabile tra il sistema ed il suo ambiente in specifiche situazioni. Gli scenari sono costruiti attraverso una notazione testuale (linguaggio Avalla). La semantica è chiara (definita in termini di ASM). Inoltre, uno scenario possiede la capacità di descrivere anche dettagli interni, non solo black box come per UML use cases, ma anche informazioni sullo stato. Avalla è un linguaggio

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

utile per validare modelli ASM scritti in AsmetaL, ed è integrato nell'ASMETA framework. Negli UML use cases, l'attore interagisce con il sistema: uno o più scenari possono essere generati per ogni caso d'uso (Black Box View), mentre l'attore ASM imposta le funzioni monitorate (ambiente) e controlla le locazioni (reazione della macchina).

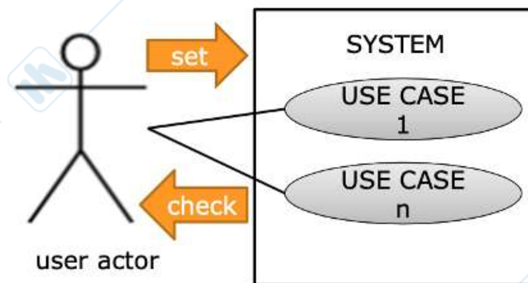


Figura 29: UML actor

L'ASM observer, invece controlla lo stato interno della macchina e gli invarianti e richiede l'esecuzione di regole arbitrarie (Gray Box View).

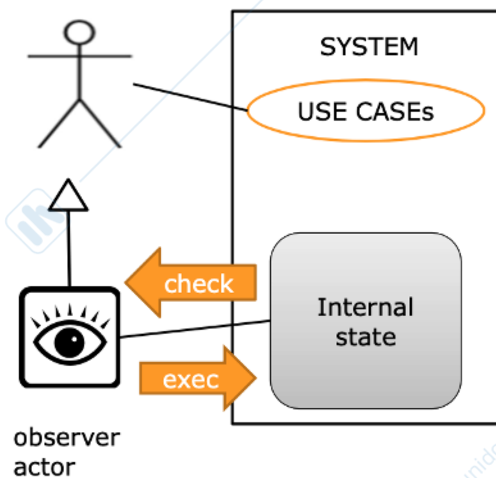


Figura 30: ASM observer

Quindi, esistono due tipi di attori esterni:

- User: ha una black box view del sistema;
- Observer: ha una gray box view del sistema.

Gli obiettivi per gli scenari sono due:

- Validazione classica: azione dell'utente e reazioni della macchina.
- Attività di testing: ispezione dell'observer dello stato interno della macchina.

Uno scenario ASM è una sequenza di interazioni, che consiste nelle seguenti azioni:

- User/Observer:
  1. impostare l'ambiente (ovvero i valori delle funzioni monitorate/condivise)
  2. verificare le uscite della macchina (ovvero i valori delle funzioni di uscita),

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

3. controllare lo stato della macchina e gli invarianti
4. chiedere l'esecuzione di determinate regole di transizione

- Macchina: fa un passo come reazione delle azioni dell'attore

Lo scenario è scritto nel linguaggio AValLa (Asm Validation Language). La Tabella 5 elenca le primitive del linguaggio AValLa.

Set	Un comando per impostare la posizione di una funzione (monitorata) su un valore specifico: simula l'ambiente
Check	Ispezionare i valori esterni e (solo per l'osservatore) ispezionare i valori interni nello stato corrente
Step	Per segnalare che l'ambiente ha terminato di aggiornare le posizioni monitorate, quindi la macchina può eseguire un passaggio
StepUntil	Per segnalare che la macchina può eseguire un passaggio in modo iterativo fino a quando una condizione specificata diventa vera
Invariant	Per indicare le proprietà delle specifiche critiche che devono sempre essere valide per uno scenario
Exec	Eseguire la regola di transizione quando richiesto dall'osservatore

Tabella 5: Primitive di AValLa

La Figura 31 elenca, invece, la sintassi dei comandi del linguaggio 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	

Figura 31: Sintassi dei comandi

La Figura 32 mostra un esempio di scenario sulla base del codice relativo all'esercizio AdvancedClock.

L'esempio Advanced Clock è un sistema chiuso, dove non c'è interazione con l'Observer, quindi non è possibile utilizzare il comando set.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Figura 32: Esempio di scenario

Internamente, il Validator di Asmeta genera un unico modello, con input lo scenario e l'ASM da validare. Quindi, ricevuti in input lo scenario e la specifica ASM:

- Integra la specifica con scenario
- Valida la specifica in AsmetaL
- Viene lanciato l'Asmeta Simulator per permettere la visione del risultato della validatura (PASS/FAIL, Coverage, Inspection Window).

Elenchiamo ora alcune osservazioni sull'uso del validatore:

1. Evitare di utilizzare cartelle di lavoro aventi nomi con spazio: in questo caso è possibile riportare un errore in fase di validazione dovuto al fatto che creando il file temp.asm dal merge tra la spec.asm e scenario.test, il comand import della spec.asm per importare la Standard Library, viene recreato in temp.asm seguendo i path assoluti; se tali path contengono nomi con spazio, il path dell'import in temp.asm non è riconosciuto come valido.
2. Nel load della specifica ASM, il path può essere dato con: load./spec.asm
3. Il validatore si ferma 2 passi (stati) dopo la fine della simulazione dello scenario. Questo è dovuto al fatto che, eseguito l'ultimo comando dello scenario, il simulatore fa:
    - Un passo per riportare l'esito del passo di esecuzione dello scenario (nella locazione result=1).
    - Un passo successivo per accorgersi che l'update set è vuoto e quindi la simulazione deve terminare.

5 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. Gli ASM multi agenti descrivono un modello distribuito della computazione. La computazione distribuita è modellata mediante un insieme di Agenti che operano in modo concorrente con movimenti concorrenti sincroni/asincroni e stati globali condivisi tra gli agenti. Gli agenti ASM

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

possono essere creati dinamicamente, hanno una visione parziale dello stato globale e hanno il proprio programma da eseguire.

Nella Figura 33 vediamo una rappresentazione grafica degli agenti. Ciascuno degli agenti avrà una propria algebra. Quindi ogni agente avrà una vista parziale dello stato globale (S), data da una funzione ASM "view". View è una funzione che ha per input un agente e lo stato globale e restituisce in output una vista sul sistema globale.

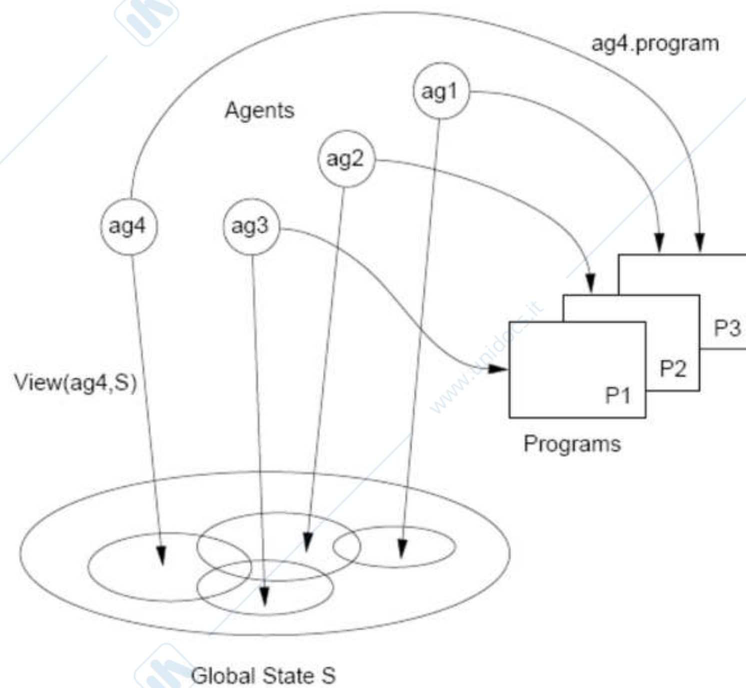


Figura 33: ASM multiagente

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. Le intersezioni rappresentano, invece, una vista condivisa con altri agenti. Per una $f : A \to B$ globale, la dichiarazione di f diventa $f : Agent \times A \to B$. $f(self, x)$ denota la versione privata di $f(x)$ dell'agente corrente self. self viene interpretata da a (agente) come se stesso. Se per ogni agente vale lo stesso modello, non andrò a scrivere un modello per ogni agente, ma scriverò un modello con la keyword self per indicare l'agente stesso.

Si distinguono modelli con agenti:

- Sincroni: gli agenti eseguono il loro programma in parallelo, sincronizzati su un implicito clock globale del sistema. Tutti partono nello stato iniziale, controllano le regole da calcolare e viene calcolato l'update set di tutti gli agenti, successivamente tutti passano allo stato S_1.
- Asincrone: gli agenti eseguono il loro programma in parallelo, ma in modo indipendente tra loro. Ciascuno ha il proprio clock che regola la durata di una mossa e ciascuno opera nel proprio stato locale. Ciascun agente va alla propria velocità.

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

- $a \in Agent$ (il dominio degli agenti)
- programmi $ASM(a)$ che sono ASM di base

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Se vogliamo specificare che si tratta di un ASM multi agente asincrona, bisogna specificarlo, tramite: **[asyncr] asm** ASM_name. La parola chiave asyncr è opzionale (perchè racchiusa in []): se presente, denota una ASM asincrona multi-agent, altrimenti, l'ASM è considerata sincrona multi-agent.

La parola chiave **import** permette di importare da altri modelli parte di segnatura che non è propria, così come l'**export** permette di rendere visibile parte della segnatura. Con export* è possibile rendere visibile all'esterno l'intera segnatura. In caso di ASM multi-agente è utile definire l'header 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]

Si tratta di funzioni di import/export di simboli (id) di domini, funzioni (e loro domini e codomini), e regole da/verso altre ASM m_i. La segnatura contiene dichiarazioni (non definizioni) di domini e funzioni, le quali vanno dichiarate nel modello in cui viene importato.

Nel caso di ASM sincrone tutte le ASM lavorano in parallelo. L'insieme opera negli stati globali, ottenuti dall'unione di tutti gli stati delle ASM componenti (dei singoli agenti). La sequenza di eventi che determina un'esecuzione è la sequenza degli stati che rappresentano l'esecuzione della sync_ASM. Un'esecuzione di multi-agent ASM con agenti sincroni è una quasi-sequential run, cioè 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.

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, perché non c'è un sistema di riferimento comune al tempo. Sorge la difficoltà di gestione della consistenza. Non è possibile mettere in relazione l'insieme di mosse tra agenti diversi, dal momento che hanno "orologi" non comparabili fra loro. É invece possibile ricostruire una relazione d'ordine tra le mosse del singolo agente.

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;
$m_1 < m_2$ se m_1 è eseguita prima di m_2)

dei suoi agenti che soddisfano le condizioni:

1. **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. Ad esempio, per ogni $m \in M$ l'insieme $\{m'|m' < m\}$ è finito.

2. **sequenzialità degli agenti**: ogni agente opera in modo sequenziale, ovvero l'insieme di mosse $\{m|m \in M$, a esegue m$\}$ di ogni agent $a \in Agent$ è linearmente ordinato per $<$.

3. **coerenza**: sia X un segmento iniziale finito (sottoinsieme chiuso a sinistra) di $(M, <)$, cioè un sottoinsieme finito di mosse. Sia $\sigma(X)$ lo stato associato ad X: $\sigma(X)$ è il risultato di tutte le mosse m in 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 $m \in X$. Prendiamo X sottoinsieme di M, che sia un segmento iniziale finito. Prendiamo l'elemento massimale di X, dal momento che non è possibile paragonare tutti gli argomenti del sottoinsieme, potremmo avere più di un massimo, per questo motivo si parla di massimale. Consideriamo il sottoinsieme X, senza considerare l'elemento massimale scelto. Applicando m allo stato risultante dal sottoinsieme senza m, ottengo lo stato $\sigma(X)$, che era associato al sottoinsieme finito di mosse iniziale.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

6 Metodologia di specifica: Ground Model e raffinamento di modelli

L'obiettivo nell'ambito di SE è definire un framework concettuale semplice e preciso per supportare e integrare le principali attività di sviluppo software e le principali tecniche di modellazione e analisi. Occorrono un metodo di specifica e un processo di sviluppo. Il metodo basato sulle Abastract State Machine (ASM) permette la specifica rigorosa e formale. Inoltre, permette la specifica di sistemi software, con flussi di computazione diversi:

- sequenziale
- parallela
- non deterministica
- distribuita

Il metodo ASM, quindi non solo lo sviluppo formale, si basa su tre concetti:

1. ASMs
2. Ground Model: è da intendersi come la prima ASM (primo modello) corretta, ma non necessariamente completa dei requisiti.
3. Raffinamento di modelli: metodo di progettazione che permette di formalizzare i requisiti tramite sviluppo incrementale della progettazione:
   - Raffinamento orizzontale: sviluppo per composizione (situazioni di gestione dell'errore, oppure di eventi che si scatenano a partire dall'ambiente)
   - Raffinamento verticale: dal ground model si sviluppano modelli più dettagliati a livello di codice.

La metodologia di sviluppo supporta tutte le attività di progettazione dai requisiti al codice:

- Specifica dei requisiti: costruzione del Ground Model (GM)
- Architettura e progetto delle componenti
- Validazione dei modelli, mediante simulazione
- Verifica delle proprietà del modello (dimostrazioni)
- Documentazione

6.1 Ground Model (GM)

Il GM descrive i requisiti del sistema in modo:

- evolutivo
- consistente e non ambiguo
- semplice e conciso
- astratto anche se non completo (rispetto ai requisiti)

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Il GM deve risultare comprensibile e verificabile sia per gli specialisti del dominio applicativo che del dominio tecnologico. Il Ground Model deve essere inteso come il contratto tra cliente e sviluppatore. Deve essere in grado di dimostrare al cliente che ciò che si sta sviluppando è il modello desiderato; inoltre, deve essere abbastanza dettagliato da permettere lo sviluppo da parte dello sviluppatore. É un accordo tra il cliente e lo sviluppatore. Per questo motivo, le ASM permettono di calibrare il livello di comprensibilità, in modo da rendere il modello iniziale comprensibile al cliente. Infatti, deve essere verificabile la correttezza e la consistenza del modello rispetto ai requisiti. Il Ground Model permette di rendere noto all'inizio dello sviluppo ciò che il sistema deve fare (behaviour) in forma di definizione matematica. Tutti gli elementi del GM (predicati, funzioni, trasformazioni) corrispondono a entità del mondo reale (proprietà, relazioni, processi). La corrispondenza desiderata è 1:1.

Il GM deve essere preciso rispetto al livello di astrazione prescelto e flessibile, in modo da poter essere facilmente modificato ed esteso. Quindi, deve soddisfare due principi fondamentali per lo sviluppo di modelli: riusabilità e adattabilità a diversi domini applicativi. Il GM deve essere semplice e conciso per soddisfare la comprensibilità sia da parte dei progettisti che dei conoscitori del dominio applicativo. Il GM deve essere sufficientemente astratto, ma deve essere:

- corretto rispetto ai requisiti
- completo in base al livello di dettaglio desiderato: ogni caratteristica semanticamente rilevante per il livello di astrazione fissato deve essere presente

Inoltre, il GM deve essere validabile.

La scelta del livello di astrazione è il problema principale della definizione del GM. Il metodo ASM affronta il problema, riducendolo a un problema di scelta del linguaggio opportuno per la comunicazione tra dominio applicativo e tecnologico. Per formulare il GM è necessario rispondere alle seguenti domande:

- Quali sono gli agenti del sistema e quali relazioni intercorrono tra di essi? In particolare, che relazione sussiste tra il sistema e il suo ambiente?
- Quali sono gli stati del sistema?
  - Quali sono i domini degli oggetti e quali sono le funzioni, predicati e relazioni definiti su di essi?
  - Quali sono le parti statiche e quali quelle dinamiche (inclusi input e output) degli stati?
- Quali sono gli stati del sistema coinvolti dalle transizioni?
  - Sotto quali condizioni per gli agenti si verificano le transizioni?
  - Quali effetti sugli agenti hanno le transizioni?
  - Che cosa si suppone accada quando le condizioni non sono soddisfatte?
  - Quali forme di uso erroneo devono essere previste e quali meccanismi di gestione delle eccezioni devono essere implementati?
  - Quali sono le caratteristiche di robustezza desiderate?
  - Come sono collegate le azioni interne agli agenti alle azioni esterne?
- Chi inizializza il sistema e in cosa consiste l'inizializzazione? Che relazione esiste tra l'inizializzazione e l'input?
- Esistono condizioni di terminazione?
  - Se sì, come sono determinate?

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

  - Che relazione esiste tra la terminazione e l'output?

- La descrizione del sistema è completa e consistente?
- Quali sono le assunzioni relativamente al sistema e quali sono le proprietà desiderate?

Prendiamo come esempio l'esercizio SluiceGate, un sistema di irrigazione che controlla il flusso dell'acqua tramite una chiusa. La chiusa è sbarrata da una saracinesca che si può alzare ed abbassare. La saracinesca deve essere totalmente aperta per 10 minuti ogni tre ore; nel resto del tempo deve essere totalmente chiusa. Dei sensori segnalano se è completamente chiusa o completamente aperta. La saracinesca viene aperta/chiusa ruotando delle viti che sono manovrate da un motore. Al motore vengono inviati 4 segnali:

- per ruotare le viti in senso orario,
- per ruotarle in senso antiorario,
- per accenderlo e
- per spegnerlo

Il computer controlla il sistema emettendo i quattro segnali per guidare il motore. Il computer è collegato ai due sensori posti alla estremità del percorso della saracinesca per conoscerne la posizione. Il modello viene sviluppato a 3 livelli di astrazione successivi:

- Livello 0: SluiceGateGround (ground model)
  - Saracinesca solo in posizione di totale apertura o totale chiusura
  - Non modellato il movimento di apertura/chiusura
  - Non modellato il motore
  - Si considera solo il trascorrere degli intervalli di tempo in cui la saracinesca deve rimanere aperta/chiusa

  Nel GM la saracinesca può trovarsi solo nella posizione di totale apertura o totale chiusura; a questo livello non modelliamo i movimenti del motore che portano da uno stato all'altro e non modelliamo il motore; consideriamo solo il trascorrere degli intervalli di tempo in cui la saracinesca deve rimanere aperta o chiusa.

- Livello 1: SluiceGateMotorCtl (mono agente): si introducono il motore ed i sensori di posizione della saracinesca
- Livello 2: SluiceGateCtl (multi-agente): si introduce un agente per il sistema di controllo ed un agente per l'ambiente.

Ad ogni livello di dettaglio sono verificate le proprietà di correttezza (la cui dimostrazione vedremo più oltre). A livello di Ground Model è la correttezza rispetto ai requisiti, mentre ai livelli successivi è la correttezza rispetto al livello più astratto. Grazie alla verifica di queste proprietà, il modello è consistente, oltre che preciso, flessibile, semplice, conciso, etc.

La correttezza del GM rispetto ai requisiti è garantita dalla prova di queste proprietà:

- se la saracinesca è FULLYCLOSED e sono passati 170 minuti (passed(170)), allora nello stato successivo deve diventare FULLYOPENED
- se la saracinesca è FULLYOPENED e sono passati 10 minuti (passed(10)), allora nello stato successivo deve diventare FULLYCLOSED

Le proprietà sono dimostrabili tramite formule temporali, ma possono essere validate tramite simulazione.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

6.2 Raffinamento

Il raffinamento è il secondo blocco concettuale usato nell'applicazione del metodo. Consiste nell'ottenere a partire da una macchina più astratta una sua forma più raffinata (concreta). É applicato iterativamente sino al punto di raffinamento considerato "sufficiente" dallo sviluppatore.

Nella maggior parte dei formalismi, il concetto di raffinamento si basa sul principio di sostituzione: è accettabile sostituire un programma con un altro a patto che sia impossibile per un utente rendersi conto della sostituzione. Spesso il principio di sostituzione è applicabile con varie restrizioni (forme del programma, operazioni sugli stati, interpretazioni sulle relazioni input-output, etc.). Il metodo di raffinamento per le ASM non è basato su alcun principio di sostituzione, ma sulle commutazioni algebriche.

Per raffinare una ASM A in una ASM R si devono definire:

- una nozione di stato raffinato (a livello R);
- una nozione di stati di interesse e di corrispondenze tra gli stati d'interesse S di A e quelli S̃ di R, ossia coppie di stati delle run che uno vuole relazionare, inclusa corrispondenza stati iniziali/finali;
- una nozione di segmenti di computazione astratta $\tau_1, \tau_2, ..., \tau_m$, in A e i corrispondenti segmenti di computazione raffinata $\sigma_1, \sigma_2, ..., \sigma_n$ in R;
- una nozione di locazioni di interesse (in A) e di locazioni corrispondenti (in R) per le quali si vuole stabilire la corrispondenza nel raffinare;
- la relazione di equivalenza (≡) dei dati tra le locazioni di interesse (a livello A ed R).

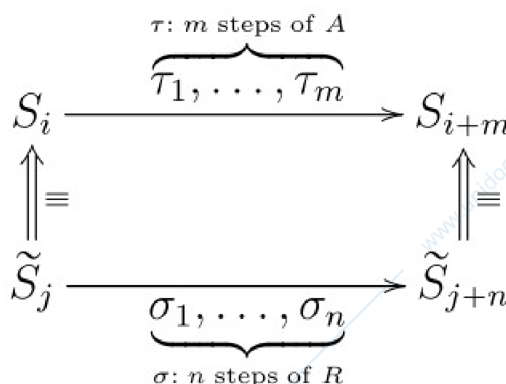


Figura 34: Schema per il raffinamento

Una volta stabilita la corrispondenza degli stati e la loro equivalenza, è possibile asserire che M* è un corretto raffinamento di M se e solo se ogni run raffinata (finita o infinita) simula una run astratta (finita o infinita) tra stati corrispondenti equivalenti.

Formalmente, fissata la nozione di equivalenza ≡ tra gli stati di interesse, fissati gli stati iniziali e finali, una ASM R è detta **corretto raffinamento** di A *se e solo se* ∀ R-run $\tilde{S}_0$, $\tilde{S}_1$, ..., esiste una A-run S_0, S_1 ... e sequenze $i_0 < i_1$... e $j_0 < j_1 <$ ... tali che $i_0 = j_0 = 0$ e $S_{i_k} \equiv \tilde{S}_{j_k}$ 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.

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari
Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Le sequenze di stati corrispondenti devono essere scelte in modo tale che siano minimali, ossia che all'interno di sequenze corrispondenti non ci siano stati equivalenti, ad esempio non esistano e tali che $i_k < i < i_{k+1}$, $j_k < j < j_{k+1}$ con $S_i \equiv \tilde{S}_j$.

La Figura 35 mostra graficamente questo concetto.

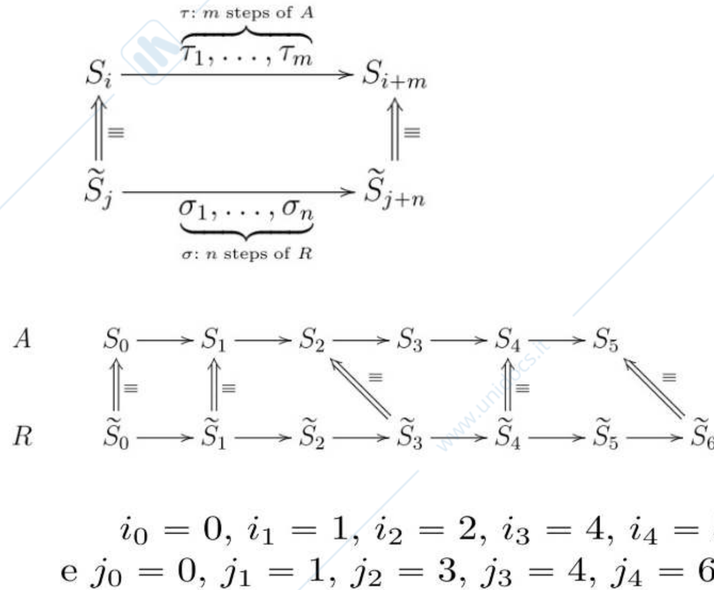


Figura 35: Rappresentazione grafica

Lo Stuffering refinement è una versione più semplice di raffinamento, la quale permette il controllo automatico della correttezza di raffinamento tra R raffinata ed A astratta. Con questa versione la relazione di equivalenza dei dati tra le locazione di interesse, si può ridurre all'uguaglianza. Formalmente, fissata la nozione di equivalenza ≡ tra gli stati di interesse, fissati tra gli stati iniziali e finali, una ASM R è detta **corretto raffinamento stuffering** di A *se e solo se* ogni R-run $\tilde{S}_0$, $\tilde{S}_1$, ..., può essere ripartita in sotto-run $\tilde{p}_0$, $\tilde{p}_1$, per ogni $\tilde{p}_i$ vale che $\forall \tilde{S} \in \tilde{p}_i : \tilde{S} \equiv S_i$.

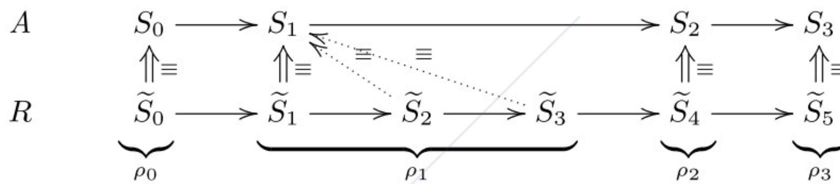


Figura 36: Rappresentazione grafica

Esistono due tipologie di raffinamento:

- Orizzontale (o Estensione conservativa): raffinamento incrementale usato per introdurre nuovi comportamenti (es. exception handling, error handling, ecc) o adattamenti a condizioni dell'ambiente. Supporta il principio del design for change. Un esempio di applicazione dell'estensione conservativa può essere il seguente: ad A viene aggiunto un nuovo comportamento modellato tramite A_{new} che esegue sotto una cond(new). Se A ed A_{new} modellano comportamenti mutuamente esclusivi, R ha la forma: *if not cond(new) then A* oppure if cond(new) then A_{new}. Se A_{new} viene richiamato da A, R è A estesa con la regola

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

if cond(new) then A_{new}. Questo può essere un esempio d'introduzione di una regola di errore.

- Verticale: permette di introdurre via via sempre più dettagli sugli elementi di un modello (domini, funzioni, regole). Supporta il principio del design for reuse

É possibile combinare le due tecniche mediante:

- Raffinamento della segnatura: raffinamento di domini e funzioni, stabilendo la nozione di stato raffinato, la relazione tra stati d'interesse e tra locazioni, e la relazione d'equivalenza. Un esempio è prendendo il modello SluiceGate è dato dall'introduzione dal seguente dominio domain PhaseDomain = { FULLYCLOSED | FULLYOPEN} raffinato in: domain PhaseDomain = {FULLYCLOSED | OPENING | FULLYOPEN | CLOSING}.
- Raffinamento procedurale: raffinamento di regole (raffinamento di sottomacchine): consiste nel sostituire una macchina (= complesso di regole) con una più complessa, stabilendo la corrispondenza tra segmenti di computazione in modo tale che l'effetto di un'operazione concreta su dati concreti sia equivalente all'effetto di operazioni astratte su dati astratti. Un esempio è il seguente: In A si usa un'operazione astratta *op* come regola vuota o comportamento non ulteriormente specificato, in R si avrà A + la rule r_op, vedi in Sluice le regole r_open ed r_shut.

Il raffinamento del SluiceGate consiste nel dettagliare le transizioni astratte, specificando le azioni del motore ed i sensori. Quindi, ogni operazione astratta di movimento della saracinesca è sostituita da un'operazione più dettagliata che coinvolge l'uso del motore. É un esempio di: raffinamento della segnatura, raffinamento procedurale e estensione conservativa.

Il raffinamento a livello 1 consiste nell'introduzione del motore e dei sensori di posizione della saracinesca. La parte della segnatura che viene mantenuta è:

domain Minutes subsetof Integer

dynamic controlled phase : PhaseDomain

dynamic monitored passed : Minutes → Boolean

domain Minutes = 10 , 170

Mentre la nuova segnatura introdotta è:

abstract domain Position

enum domain PhaseDomain = { FULLYCLOSED | OPENING | FULLYOPEN | CLOSING } //ridefinito

enum domain DirectionDomain = { CLOCKWISE | ANTICLOCKWISE } //direzione di movimento delle viti

enum domain MotorDomain = { ON | OFF } // stato del motore

dynamic controlled dir: DirectionDomain

dynamic controlled motor: MotorDomain

static openPeriod: Minutes

static closedPeriod: Minutes

static top: Position

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

static bottom: Position

dynamic monitored event: Position → Boolean

La definizione delle funzioni statiche consisiste in:

```
definitions:

    function openPeriod = 10
    function closedPeriod = 170 //(3*60 - 10)

default init s0

    function phase = FULLYCLOSED
    function motor = OFF
    function dir = ANTICLOCKWISE
```

Il raffinamento delle regole consiste nello sviluppo delle regole r_open e r_shut, con un maggior grado di definizione e nell'introduzione di nuove regole, ossia r_start_to_raise, r_start_to_lower e r_stop_motor. Le funzioni monitorate event(top)/event(bottom) segnalano se la saracinesca ha raggiunto il punto più alto/più basso del suo percorso. Il codice consiste nel:

```
rule r_start_to_raise =

    par
        dir := CLOCKWISE
        motor := ON
    endpar

rule r_start_to_lower =

    par
        dir := ANTICLOCKWISE
        motor := ON
    endpar

rule r_stop_motor =

    motor := OFF

rule r_open =

    par
        if (phase=FULLYCLOSED) then
            if (passed(closedPeriod)) then
                par
                    r_start_to_raise[]
                    phase := OPENING
                endpar
            endif
        endif
        if (phase=OPENING) then
```


Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
            if (event(top)) then
                par
                    r_stop_motor[]
                    phase := FULLYOPEN
                endpar
            endif
        endif
    endpar
```

La prova di correttezza di questo raffinamento fa uso delle seguenti Input Locations Assumption, che stabiliscono la relazione tra i valori dei sensori dell'ambiente ed il valore degli eventi del modello: quando la posizione di top (rispettivamente di bottom) della saracinesca è rilevata dal corrispondente sensore, allora event(Top) (rispettivamente event(Bottom)) diventa true in SluiceGateMotorCtl. La prova della correttezza è data da:

- Locazioni di interesse: phase, phase*, dove * indica la locazione nel modello raffinato.
- Equivalenza: ≡(phase, phase*) sse (phase=FULLYCLOSED ∧ (phase*=FULLYCLOSED ∨ phase*= OPENING) oppure (phase=FULLYOPEN ∧ (phase*=FULLYOPEN ∨ phase*= CLOSING))
- Stati corrispondenti: quelli in cui phase e phase* hanno valori equivalenti
- Segmenti di run corrispondenti:
  - A-run) FULLYCLOSED → FULLYOPEN → FULLYCLOSED → FULLYOPEN → ...
  - R-run) FULLYCLOSED → OPENING → FULLYOPEN → CLOSING → FULLYCLOSED → OPENING → ...

  É facile verificare che la definizione stuttering ref è soddisfatta.

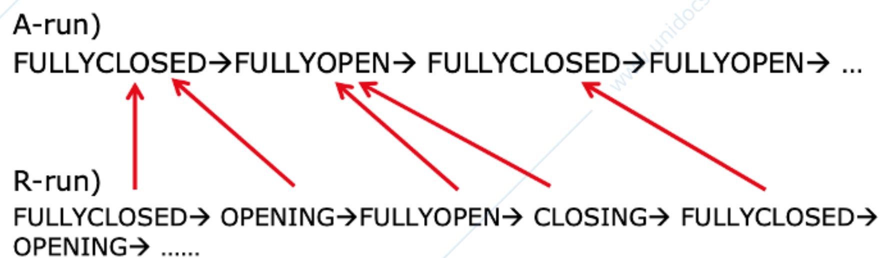


Figura 37: Prova della correttezza di SluiceGateMotorCtl rispetto a SluiceGateMotorGround

In ASM, la difficoltà del raffinamento non consiste tanto nella dimostrazione matematica della correttezza del passo di raffinamento, ma nel trovare la giusta granularità e nel formulare il raffinamento appropriato che rifletta una decisione di design. Il raffinamento è una tecnica usata per provare proprietà di sistemi che sono difficili da verificare a bassi livelli di astrazione. L'obiettivo è mostrare che una implementazione S* soddisfa una proprietà desiderata P*. I passi per fare questo sono:

- costruire un modello astratto S
- dimostrare che una forma astratta P della proprietà P* vale secondo un insieme I di assunzioni appropriate su S

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- mostrare che S è raffinato correttamente da S* e che le I valgono anche in S*

La proprietà di safety del sistema consiste nel fatto che la saracinesca è APERTA per 10 minuti ogni 3 ore e si elabora nei diversi livelli nel seguente modo:

- Proprietà di safety a livello 0:
  - se la saracinesca è FULLYCLOSED e sono passati 170 minuti (passed(170)), allora nello stato successivo deve diventare FULLYOPENED (i.e., non è in FULLYCLOSED)
  - se la saracinesca è FULLYOPENED e sono passati 10 minuti (passed(10)), allora nello stato successivo deve diventare FULLYCLOSED (i.e., non è in FULLYOPENED)
- Proprietà di safety a livello 1
  - se la saracinesca è FULLYCLOSED e sono passati 170 minuti, nello stato successivo non è in FULLYCLOSED.
  - ... analogamente per la seconda.

Vediamo ora come implementare il secondo livello di raffinamento di SluiceGate (SluiceGateCtl), nel quale si introduce un agente per il sistema di controllo ed un agente per la comunicazione tra sistema di controllo e dispositivi fisici:

- SLUICEGATECTLAgent : per modellare il SW di controllo del sistema fisico (motore, saracinesca,...)
- PULSESAgent : per modellare l'invio di impulsi dal sistema di controllo alle componenti fisiche (motori, rullo). Setta le componenti fisiche (motore e direzione di moto del rullo) sulla base dei valori degli impulsi

La parte nuova della signature che serve per modellare gli agenti è la seguente:

```
domain PULSESAgent subsetof Agent

domain SLUICEGATECTLAgent subsetof Agent // gli agenti effettivi

static pulses: PULSESAgent

static sluicegatectl: SLUICEGATECTLAgent

//funzioni in overload

dynamic shared pulse: DirectionDomain → Boolean

dynamic shared pulse: MotorDomain → Boolean
```

La definizione delle funzioni statiche rimane identica al livello di raffinamento precedente. La Main rule è la seguente:

```
main rule r_Main =
    seq
        program(sluicegatectl)
        program(pulses)
    endseq
```

L'inizializzazione delle funzioni è la seguente:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
default init s0:
    function phase = FULLYCLOSED
    function pulse($d in DirectionDomain) = false
    function pulse($d in MotorDomain) = false
    agent SLUICEGATECTLAgent: r_sluicegate[]
    agent PULSESAgent: r_pulses[]
```

Vediamo ora le regole:

```
rule r_pulses =
    par
        if (pulse(CLOCKWISE)) then
            par
                dir := CLOCKWISE
                pulse(CLOCKWISE) := false
            endpar
        endif
        if (pulse(ANTICLOCKWISE)) then
            par
                dir := ANTICLOCKWISE
                pulse(ANTICLOCKWISE) := false
            endpar
        endif
        if (pulse(MOTORON)) then
            par
                motor := MOTORON
                pulse(MOTORON) := false
            endpar
        endif
        if (pulse(MOTOROFF)) then
            par
                motor := MOTOROFF
                pulse(MOTOROFF) := false
            endpar
        endif

rule r_sluicegate =
    par
        if(phase=FULLYCLOSED) then
            if(passed(closedPeriod)) then
                par
                  r_start_to_raise[]
                  phase := OPENING
                endpar
        endif
```


Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
            if(phase=OPENING) then
                if(event(top)) then
                    par
                        r_stop_motor[]
                        phase := FULLYOPEN
                    endpar
            endif
            if(phase=FULLYOPEN) then
                if(passed(openPeriod)) then
                    par
                        r_start_to_lower[]
                        phase := CLOSING
                    endpar
            endif
            if(phase=CLOSING) then
                if(event(bottom)) then
                    par
                        r_stop_motor[]
                        phase := FULLYCLOSED
                    endpar
            endif
        endpar

rule r_start_to_lower =

        par
            r_emit[pulse(ANTICLOCKWISE)]
            r_emit[pulse(MOTORON)]
        endpar

rule r_stop_motor =

        r_emit[pulse(MOTOROFF)]

rule r_start_to_raise =

        par
            r_emit[pulse(CLOCKWISE)]
            r_emit[pulse(MOTORON)]
        endpar

macro rule r_emit($l in Boolean) =

        $l := true
```

La macro rule sluicegate è descritta come al livello di raffinamento precedente: il SW di controllo invia i segnali ai dispositivi fisici (motori, rullo), ma con la differenza che la comunicazione tra SW di controllo e dispositivi fisici è gestita dall'agente Pulse.

É un raffinamento (1,2): ogni segmento che consiste di 1 passo di SluiceGateMotorCtl (Livello 1) è raffinato da 2 passi di SluiceGateCtl (Livello 2). É possibile dimostrare che SluiceGateCtl è corretto raffinamento stuttering di SluiceGateMotorCtl:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- Stati corrispondenti: quelli in cui phase ha lo stesso valore
- Segmenti di run corrispondenti: Sui segmenti corrispondenti (in base al valore di phase). 1 passo di r_main di SluiceGateMotorCtl viene raffinata in 2 passi r_sluicegate → r_pulse in SluiceGateCtl.

7 Model Checking

Vi è un grande vantaggio nel poter verificare la correttezza dei sistemi informatici, siano essi hardware, software o una combinazione. Ciò è più evidente nel caso di sistemi critici per la sicurezza, ma si applica anche a quelli commercialmente critici, come chip prodotti in serie, mission-critical, ecc. Di recente i metodi di verifica formale sono diventati utilizzabili dall'industria e vi è una crescente domanda di professionisti in grado di applicarli. Le tecniche di verifica formale sono generalmente viste come la somma di tre componenti:

- Un framework in cui modellare il sistema che vogliamo analizzare
- Un linguaggio di specifica delle proprietà da verificare
- Un metodo per verificare che il sistema soddisfi le proprietà specificate.

Il model checking è un approccio di verifica delle proprietà automatico, basato sul modello. É stato progettato per essere utilizzato per sistemi concorrenti e reattivi e ha origine come metodologia post-sviluppo. Il controllo del modello (Model Checking) si basa sulla logica temporale. L'idea della logica temporale è che una formula non è staticamente vera o falsa in un modello, come nella logica proposizionale e predicata. Invece, i modelli della logica temporale contengono diversi stati e una formula può essere vera in alcuni stati e falsa in altri. Pertanto, la nozione statica di verità è sostituita da una dinamica, in cui le formule possono cambiare i loro valori di verità mentre il sistema evolve da stato a stato. Nella verifica dei modelli, i modelli $\mathcal{M}$ sono sistemi di transizione e le proprietà ϕ sono formule nella logica temporale. Per verificare che un sistema soddisfi una proprietà, dobbiamo fare tre cose:

- Si costruisce un modello $\mathcal{M}$ che descrive il comportamento del sistema;
- Si codifica la proprietà da verificare in una formula temporale ϕ
- Eseguire il controllo modello con gli ingressi $\mathcal{M}$ e ϕ. Quindi, si chiede al model checker di verificare che $\mathcal{M} \models \phi$. Il model checker emette la risposta "sì" se $\mathcal{M} \models \phi$ e "no" altrimenti; in quest'ultimo caso, la maggior parte dei model checker produce anche una traccia del comportamento del sistema che causa questo errore. Questa generazione automatica di tali "tracce" è uno strumento importante nella progettazione e nel debug dei sistemi.

Esistono diverse logiche temporali che possono essere divise in due classi fondamentali: le linear-time logics e le branching-time logics:

- LTL (Linear-time logics) considera il tempo come un insieme di path (cammini), dove un path è una sequenza di istanti di tempo.
- BTL (Branching-time logics) rappresenta il tempo come un albero, con radice l'istante corrente.
- Un'altra classificazione divide tra tempo continuo e discreto

Le logiche temporali hanno un aspetto dinamico per loro, poiché la verità di una formula non è fissa in un modello, come nella logica predicata o proposizionale, ma dipende dal punto temporale all'interno del modello. Noi studieremo una logica branching-time discreta, in cui il tempo si sta ramificando: Computing Tree Logic (CTL).

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

7.1 Computation Tree Logic - CTL

Computation Tree Logic, o CTL in breve, è una branching-time logic, nel senso che il suo modello di tempo è una struttura ad albero in cui il futuro non è determinato; ci sono diversi percorsi nel futuro, ognuno dei quali potrebbe essere il percorso "reale" che viene realizzato. La CTL è una logica con connettivi che ci permettono di specificare proprietà temporali. Essendo una logica branching-time, i suoi odelli sono rappresentabili mediante una struttura ad albero in cui il futuro non è deterministico: esistono differenti computazioni o paths nel futuro e uno di questi sarà il percorso realizzato. La logica è costruita su di un insieme di formule atomiche $AP\{p, q, r, ...\}$ che rappresentazione descrizioni atomiche del sistema. Definiamo in maniera induttiva le formule CTL:

$$\phi ::= \top|\bot|p \in AP|\neg\phi|\phi \wedge \phi|\phi \vee \phi|\phi \rightarrow \phi|AX\phi|EX\phi|A[\phi U \phi]|E[\phi U \phi]|AG\phi|EG\phi|AF\phi|EF\phi$$

Dove:

- $\top, \bot, \neg, \wedge, \cup, \rightarrow$ sono connettivi logici classici
- $AX, EX, AG, EG, AU, EU, AF$ e EF sono connettivi temporali

In particolare:

- A sta per "along All path" (inevitably)
- E sta per "along at least (there Exists) one path" (possibly)
- X, F, G e U sono gli operatori della logica temporale, dove:
  - X: next state
  - F: some future state
  - G: all future states (globally)
  - U: until

La coppia di simboli in $E[\phi U \phi]$, per esempio, è EU. In CTL, coppie di simboli come EU, sono indivisibili. AU e EU sono operatori binari e i simboli X, F, G e U non possono occorrere se non preceduti da A o E e viceversa.

La convenzione sull'ordinamento consiste in priorità vincolanti per i connettivi CTL:

- I connettivi unari (costituiti da $\neg$ e i connettivi temporali AG, EG, AF, EF, AX ed EX) si legano più strettamente. Quindi, si legano con priorità più elevata;
- Successivamente nell'ordine arrivano $\wedge$ e $\vee$;
- successivamente arrivano $\rightarrow$, AU e EU.

Naturalmente, possiamo usare le parentesi per ignorare queste priorità. Vediamo alcuni esempi di formule CTL ben formate e alcuni esempi che non sono ben formati, al fine di comprendere la sintassi. Supponiamo che p, q e r siano formule atomiche:

- Formule CTL ben-formate:
  - $AG(q \rightarrow EGr)$
  - $EF\ E(r\ U\ q)$
  - $A[p\ U\ EFr]$
  - $EFEGp \rightarrow AFr$
  - $AG(p \rightarrow A[p\ U\ (\neg p \wedge A[\neg p\ U\ q])])$

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- Formule CTL non ben-formate:
  - $EFGr$
  - $A\neg G\neg p$
  - $F[r\ U\ q]$
  - $EF(r\ U\ q)$
  - $AEFr$
  - $A[r\ U\ q] \wedge (p\ U\ r)]$

Vale soprattutto la pena capire perché le regole di sintassi non ci consentono di costruirle. Ad esempio, prendi $EF(r\ U\ q)$. Il problema con questa stringa è che U può verificarsi solo se accoppiato con una A o una E. La E che abbiamo è accoppiato con la F. Per trasformarla in una formula CTL ben formata, dovremmo scrivere $EFE[r\ U\ q]$ o $EFA[r\ U\ q]$. Si noti che utilizziamo le parentesi quadre dopo A o E, quando l'operatore associato è una U. Il motivo per cui $A[(r\ U\ q) \wedge (p\ U\ r)]$ non è una formula ben formata è che la sintassi non consente di mettere un connettivo booleano (come $\wedge$) direttamente all'interno di $A[]$ o $E[]$. Le occorrenze di A o E devono essere seguite da una di G, F, X o U; quando sono seguiti da U, deve essere nella forma $A[\phi U \phi]$. Ora, le ϕ possono contenere $\wedge$, poiché sono formule arbitrarie; quindi $A[(p \wedge q)\ U\ (\neg r \rightarrow q)]$ è una formula ben formata. Come con qualsiasi linguaggio formale è utile disegnare alberi di analisi per formule ben formate. L'albero di analisi per $A[AX\neg p\ U\ E[EX(p \wedge q)\ U\ \neg p]]$ è mostrato nella Figura 38.

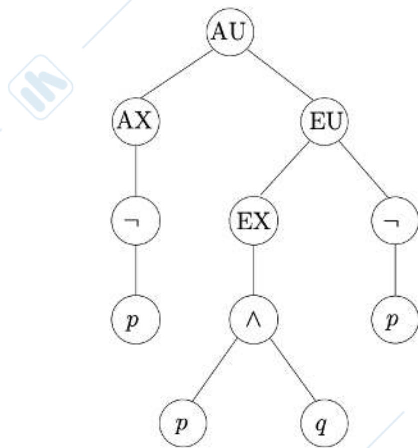


Figura 38: L'albero di analisi di una formula CTL senza notazione infissa

Un'interpretazione di CTL è una struttura di Kripke con relazione di serialità: $\mathcal{M} = (S, \triangle, I, L)$, dove:

- S: insieme di stati
- $\triangle$ (o $\rightarrow$): funzione di transizione, tale che $\forall s \in S \exists s' \in S$ con $s \rightarrow s'$
- I: insieme di stati, detti iniziali
- una funzione di etichettatura $L : S \rightarrow 2^{PA}$

$\forall s \in S$ si deve avere almeno una $s' \in S$ tale che $s \rightarrow s'$. Si impone l'assenza di deadlock (eventuale aggiunta di uno stato fittizio s_d). Un modo conveniente per rappresentare è l'utilizzo dei grafi diretti; i nodi rappresentano gli stati e contengono gli atomi proposizionali che sono veri in quello stato; gli archi rappresentano le transizioni.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

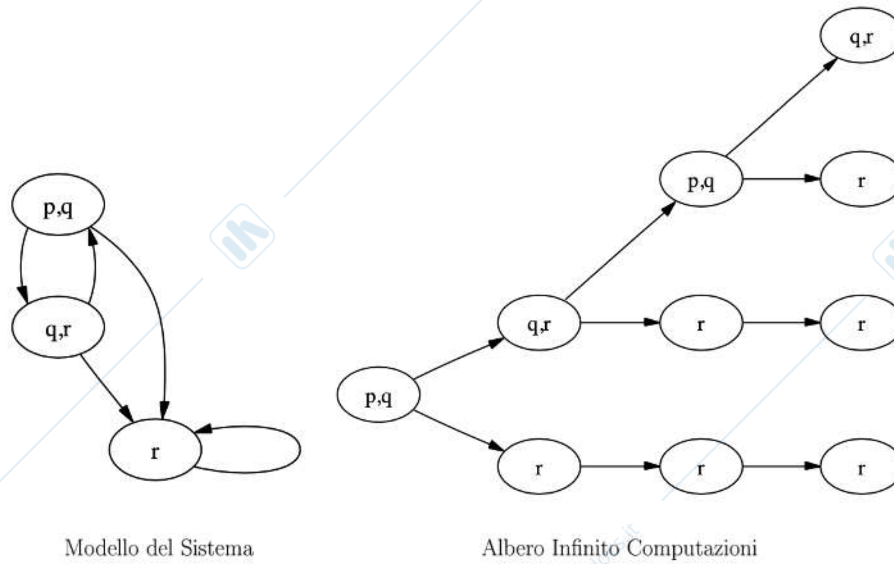


Figura 39: Unwinding di un sistema

Sia $\mathcal{M}$ un modello CTL. Dato uno stato $s \in S$, usiamo la seguente notazione per dire che una formula CTL ϕ è soddisfatta dal modello $\mathcal{M}$ nello stato s: $\mathcal{M}, s \models \phi$. La relazione $\models$ è definita per induzione strutturale sulle formule CTL:

1. $\mathcal{M}, s \models \top$ e $\mathcal{M}, s \not\models \bot$ per ogni $s \in S$.
2. $\mathcal{M}, s \models p$ sse $p \in L(s)$
3. $\mathcal{M}, s \models \neg\phi$ sse $\mathcal{M}, s \not\models \phi$
4. $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ sse $\mathcal{M}, s \models \phi_1$ e $\mathcal{M}, s \models \phi_2$
5. $\mathcal{M}, s \models \phi_1 \vee \phi_2$ sse $\mathcal{M}, s \models \phi_1$ oppure $\mathcal{M}, s \models \phi_2$
6. $\mathcal{M}, s \models \phi_1 \rightarrow \phi_2$ sse $\mathcal{M}, s \not\models \phi_1$ oppure $\mathcal{M}, s \models \phi_2$
7. $\mathcal{M}, s \models AX\phi$ sse per ogni s_1 tale che $s \rightarrow s_1$ è vero che $\mathcal{M}, s_1 \models \phi$. AX significa in tutti gli stati successivi.
8. $\mathcal{M}, s \models EX\phi$ sse per qualche s_1 tale che $s \rightarrow s_1$ è vero che $\mathcal{M}, s_1 \models \phi$. EX significa in qualche stato successivo (E è duale[1] di A).
9. $\mathcal{M}, s \models AG\phi$ sse per tutti i path $s_1 \rightarrow s_2 \rightarrow ...$, con s_1 uguale a s, è vero che $\mathcal{M}, s_i \models \phi$ per ogni i. In tutti i path di computazione che iniziano da s la proprietà ϕ è sempre verificata.
10. $\mathcal{M}, s \models EG\phi$ sse esiste un path $s_1 \rightarrow s_2 \rightarrow ...$, con s_1 uguale a s, in cui $\mathcal{M}, s_i \models \phi$ per ogni i. Esiste almeno un path di computazione che inizia da s in cui la proprietà ϕ è sempre verificata.
11. $\mathcal{M}, s \models AF\phi$ sse per tutti i path $s_1 \rightarrow s_2 \rightarrow ...$, con s_1 uguale a s, esiste almeno un s_i in cui $\mathcal{M}, s_i \models \phi$. In tutti i path di computazione che iniziano da s, esiste un qualche stato futuro in cui la proprietà ϕ è sempre verificata.

[1] In un'algebra di Boole la duale di una qualsiasi asserzione è l'asserzione che si ottiene scambiando fra loro il prodotto con la somma e lo 0 con l'1.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

12. $\mathcal{M}, s \models EF\phi$ sse esiste un path $s_1 \to s_2 \to ...$, con s_1 uguale a s, in cui esiste almeno un s_i in cui $\mathcal{M}, s_i \models \phi$. Esiste un path di computazione che inizia da s in cui esiste un qualche stato futuro in cui la proprietà ϕ è sempre verificata.

13. $\mathcal{M}, s \models A[\phi_1 U \phi_2]$ sse per tutti i path $s_1 \to s_2 \to ...$, con s_1 uguale a s, c'è qualche s_i sul path in cui $\mathcal{M}, s_i \models \phi_2$ e per ogni $j < i$ $\mathcal{M}, s_j \models \phi_1$. In tutti i path di computazione che iniziano con s, è vero che $\phi_1 U \phi_2$.

14. $\mathcal{M}, s \models E[\phi_1 U \phi_2]$ sse per tutti i path $s_1 \to s_2 \to ...$, con s_1 uguale a s, in cui c'è qualche s_i sul path in cui $\mathcal{M}, s_i \models \phi_2$ e per ogni $j < i$ $\mathcal{M}, s_j \models \phi_1$. Esiste un path di computazione che inizia con s in cui è vero che $\phi_1 U \phi_2$.

Quando parliamo di stati futuri includiamo anche il presente. Le precedenti clausole 9–14 si riferiscono ai percorsi di calcolo nei modelli. É quindi utile visualizzare tutti i possibili percorsi di calcolo da un dato stato s svolgendo il sistema di transizione per ottenere un albero di calcolo infinito, da cui "logica dell'albero di calcolo". I diagrammi nelle Figura 40 mostra schematicamente sistemi i cui stati iniziali soddisfano rispettivamente le formule $EF\phi, EG\phi, AG\phi e AF\phi$. Naturalmente, potremmo aggiungere più ϕ a ciascuno di questi diagrammi e preservare comunque la soddisfazione, anche se non c'è nulla da aggiungere per AG. I diagrammi illustrano un modo "minimo" di soddisfare le formule.

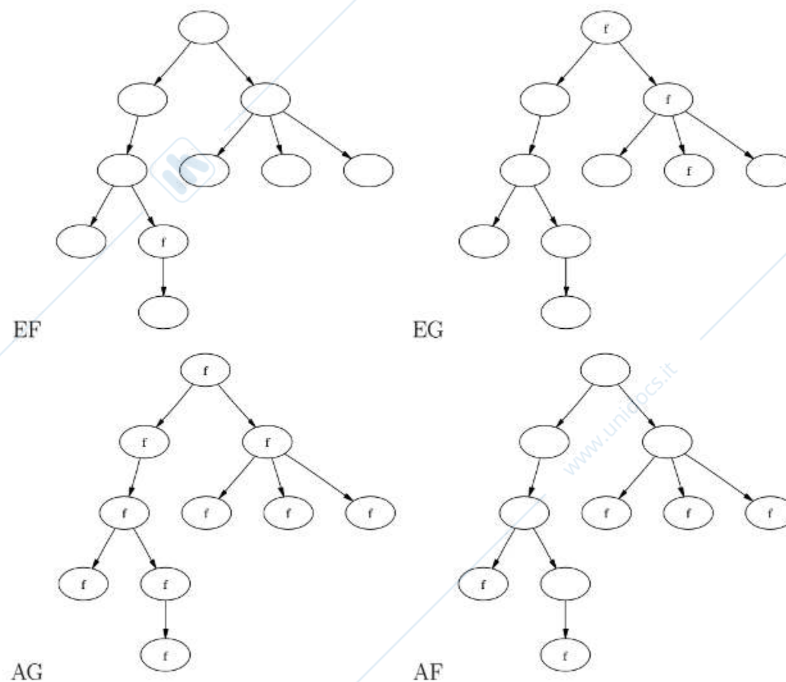


Figura 40: Unwinding di un sistema

É utile esaminare alcuni esempi tipici di formule. Supponiamo che le descrizioni atomiche includano alcune parole come busy e requested. Vediamo una serie di pattern comuni in CTL che ci permettono di codificare proprietà:

- Il sistema può raggiungere uno stato in cui è partito, ma non è pronto: $EF(started \land \neg ready)$
- In qualsiasi momento (in qualsiasi stato), se viene eseguita una richiesta allora seguirà una risposta (riconoscimento): $AG(requested \to AF acknowledged)$

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- In qualsiasi momento (in qualsiasi stato), il sistam raggiungerà sempre uno stato in cui è abilitato: $AG(AFenabled)$.
- In ogni computazione, esiste uno stato in cui il sistema entra in stato di deadlock permanente, quindi verrà bloccato in modo permanente: $AF(AGdeadlock)$.
- In qualsiasi momento, è sempre possibile raggiungere uno stato del sistema in cui viene eseguita una restart: $AG(EFrestart)$.
- In qualsiasi momento, se siamo al secondo piano, stiamo salendo e vogliamo andare al quinto, l'ascensore salirà fino a quando raggiungeremo il piano desiderato: $AG(floor = 2 \land direction = up \land ButtonPressed5 \to A[direction = upUfloor = 5$.
- In qualsiasi momento, se l'ascensore è al terzo piano, è libero e le porte sono chiuse è possibile che l'ascensore rimanga bloccato: $AG(floor = 2 \land door = closed \to EG(floor = 3 \land idle \land door = closed))$

Due formule CTL φ e ψ sono semanticamente equivalenti se ciascun stato in ciascun modello che soddisfa l'una soddisfa anche l'altra. Vediamo alcune equivalenze tra formule CTL:

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

Possiamo definire un insieme di simboli adeguato a descrivere le formule CTL: ogni formula CTL può essere riscritta in una equivalente contenente solo $\bot, \land, \neg, AF, EU$ e EX nel seguente modo:

1. $\top = \neg\bot$
2. $\phi_1 \lor \phi_2 = \neg(\neg\phi_1 \land \neg\phi_2)$
3. $\phi_1 \to \phi_2 = (\neg\phi_1 \lor \phi_2)$
4. $AX(\phi_1) = (\neg EX\neg\phi_1)$
5. $EF(\phi_1) = E(\top\ U\ \phi_1)$
6. $EG(\phi_1) = (\neg AF\neg\phi_1)$
7. $AG(\phi_1) = (\neg EF\neg\phi_1)$
8. $(A\phi_1\ U\ \phi_2) = \neg(E[\neg\phi_2\ U\ (\neg\phi_1 \land \neg\phi_1)] \lor \neg AF\phi_2)$

7.2 Algoritmi per Model Checking

Le definizioni semantiche per LTL e CTL ci consentono di verificare se gli stati iniziali di un determinato sistema soddisfano una formula LTL o CTL. Questa è la domanda base per il controllo del modello. In generale, i sistemi di transizione interessanti avranno un numero enorme di stati e la formula che ci interessa verificare potrebbe essere piuttosto lunga. Vale quindi la pena provare a trovare algoritmi efficienti.

Dati $\mathcal{M}, s \in Se\phi$ verifica se $\mathcal{M}, s \models \phi$. Possiamo considerare due tipi di problema:

- considerare uno stato $s_0 \in I$ in cui verificare che $\mathcal{M}, s_0 \models \phi$.
- considerare uno stato $s \in S$ qualsiasi, cioè estrarre l'insieme $x \subseteq S$ di stati in cui $\mathcal{M}, s \models \phi$. In questo ultimo caso possiamo anche verificare che $\mathcal{M}, s_0 \models \phi$, verificando che s_0 sia contenuto nell'insieme di stati che soddisfano ϕ, ottenuto come detto sopra.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

7.2.1 Labelling Algortihm

Presentiamo un algoritmo che, dato un modello e una formula CTL, genera l'insieme di stati del modello che soddisfano la formula. L'algoritmo non deve essere in grado di gestire esplicitamente tutte le connessioni CTL, poiché abbiamo già visto che i connettivi $\bot, \neg, \land$ formano un insieme adeguato per quanto riguarda i connettivi proposizionali; e AF, EU ed EX formano una serie adeguata di connettivi temporali. Data una formula CTL arbitraria ϕ, dovremmo semplicemente pre-elaborare ϕ per scriverlo in una forma equivalente in termini di un adeguato set di connettori, e quindi chiamare l'algoritmo di controllo del modello. L'algoritmo etichetta ciascuno stato $s \in S$ con l'insieme delle formule vere in s stesso. Quindi, ogni stato è etichettato da un insieme di formule. "Etichettare uno stato s con una formula ϕ" significa aggiungere ϕ all'etichetta di s. Ecco l'algoritmo:

INPUT: Un modello $\mathcal{M}$ e una formula ϕ

OUTPUT: Un insieme $X \subseteq S$ di stati s in cui ϕ è soddisfatta

Innanzitutto, modifica ϕ nell'output di TRANSLATE (ϕ), ovvero scriviamo ϕ in termini di connettività AF, EU, EX, $\bot, \neg$ e $\land$ utilizzando le equivalenze fornite in precedenza. Quindi, etichettare gli stati di $\mathcal{M}$ con le sottformule di ϕ che sono soddisfatte lì, iniziando con le più piccole sottformule e procedendo verso l'esterno verso ϕ. Supponiamo che ψ sia un subformula di ϕ e che gli stati che soddisfano tutte le subformule immediate di ψ siano già state etichettate. L'algoritmo etichetta gli stati valutando la struttura di ψ. Se ψ è:

- $\bot$: allora nessuno stato deve essere etichettato con $\bot$.
- p: allora etichetta uno stato s con p se $p \in L(s)$.
- $\psi_1 \land \psi_2$: etichetta uno stato s con $\psi_1 \land \psi_2$ se s è già etichettato con ψ_1 e ψ_2.
- $\neg\psi_1$: etichetta uno stato s con $\neg\psi_1$ se s non è già etichettato con ψ_1.
- $AF\psi_1$:
  - Se uno stato s è etichettato con ψ_1 allora s viene etichettato con $AF\psi_1$.
  - Uno stato è etichettato con $AF\psi_1$ se tutti gli stati successori sono etichettati con $AF\psi_1$, fin quando è possibile (Figura 41).

Figura 41: La fase di iterazione della procedura per l'etichettatura indica con le sottformule del modulo $AF\psi_1$.

- $E[\psi_1\ U\ \psi_2]$:
  - Se uno stato s è etichettato con ψ_2 allora viene etichettato con $E[\psi_1\ U\ \psi_2]$.
  - Uno stato è etichettato con $E[\psi_1\ U\ \psi_2]$ se è etichettato con ψ_1 e almeno uno dei suoi successori è etichettato con $E[\psi_1\ U\ \psi_2]$, fin quando possibile (Figura 42).

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

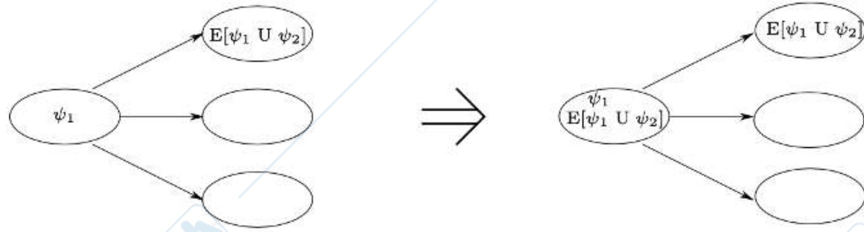


Figura 42: La fase di iterazione della procedura per l'etichettatura indica con le sottformule del modulo $E[\psi_1\ U\ \psi_2]$.

- $EX\psi_1$: Etichetta uno stato s con $EX\psi_1$ se uno dei suoi successori con ψ_1.

Dopo aver eseguito l'etichettatura per tutte le sottformule di ϕ (incluso ϕ stesso), abbiamo generato gli stati che sono etichettati ϕ. La complessità di questo algoritmo è $O(f \times V \times (V \times E))$, dove f è il numero di connettivi nella formula ϕ, V è il numero di stati ed E è il numero di transizioni; l'algoritmo è lineare nella dimensione della formula e quadratico nella dimensione del modello.

7.2.2 Pseudo-codice dell'algoritmo di controllo del modello CTL

Presentiamo lo pseudo-codice per l'algoritmo di etichettatura di base. La funzione principale SAT (soddisfabilità) assume come input una formula CTL. La Figura 43 presenta la funzione SAT. Prende una formula CTL come input e restituisce l'insieme di stati che soddisfano la formula. Chiama le funzioni SAT_{EX}, SAT_{EU} e SAT_{AF}, rispettivamente, se EX, EU o AF è la radice dell'albero di analisi dell'input.

```
function SAT (φ)
 /* determines the set of states satisfying φ */
begin
   case
      φ is ⊤ : return S
      φ is ⊥ : return ∅
      φ is atomic: return {s ∈ S | φ ∈ L(s)}
      φ is ¬φ1 : return S − SAT (φ1)
      φ is φ1 ∧ φ2 : return SAT (φ1) ∩ SAT (φ2)
      φ is φ1 ∨ φ2 : return SAT (φ1) ∪ SAT (φ2)
      φ is φ1 → φ2 : return SAT (¬φ1 ∨ φ2)
      φ is AX φ1 : return SAT (¬EX ¬φ1)
      φ is EX φ1 : return SAT_EX(φ1)
      φ is A[φ1 U φ2] : return SAT(¬(E[¬φ2 U (¬φ1 ∧ ¬φ2)] ∨ EG ¬φ2))
      φ is E[φ1 U φ2] : return SAT_EU(φ1, φ2)
      φ is EF φ1 : return SAT (E(⊤ U φ1))
      φ is EG φ1 : return SAT(¬AF ¬φ1)
      φ is AF φ1 : return SAT_AF (φ1)
      φ is AG φ1 : return SAT (¬EF ¬φ1)
   end case
end function
```

Figura 43: La funzione SAT

La Figura 41 presenta la funzione SAT_{AF}. Calcola gli stati che soddisfano ϕ chiamando SAT. Quindi, accumula stati che soddisfano $AF\phi$ nel modo descritto nell'algoritmo di etichettatura.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Si ripete il procedimento presentato nella Figura 41, finché non risulta nessun cambiamento dal passo precedente.

```
function SAT_AF (φ)
 /* determines the set of states satisfying AF φ */
local var X, Y
begin
    X := S;
    Y := SAT (φ);
    repeat until X = Y
    begin
        X := Y;
        Y := Y ∪ pre_∀(Y)
    end
    return Y
end
```

Figura 44: La funzione SAT_{AF}

La Figura 45 presenta la funzione SAT_{EU}. Calcola gli stati che soddisfano ϕ chiamando SAT. Quindi, accumula stati che soddisfano $E[\phi\ U\ \psi]$ nel modo descritto nell'algoritmo di etichettatura. Si ripete il procedimento presentato nella Figura 42, finché non risulta nessun cambiamento dal passo precedente.

```
function SAT_EU (φ, ψ)
 /* determines the set of states satisfying E[φ U ψ] */
local var W, X, Y
begin
    W := SAT (φ);
    X := S;
    Y := SAT (ψ);
    repeat until X = Y
    begin
        X := Y;
        Y := Y ∪ (W ∩ pre_∃(Y))
    end
    return Y
end
```

Figura 45: La funzione SAT_{EU}

La Figura 46 presenta la funzione SAT_{EX}. Calcola gli stati che soddisfano ϕ chiamando SAT. Quindi, guarda indietro $\rightarrow$ per trovare gli stati che soddisfano $EX\phi$.

L'algoritmo è presentato nella Figura 43 e le sue funzioni secondarie nelle Figure 44–46 usano le variabili di programma X, Y, V e W che sono insiemi di stati. Il programma per SAT gestisce direttamente i casi facili e passa casi più complicati a procedure speciali, che a loro volta potrebbero chiamare SAT ricorsivamente su sottoespressioni. Queste procedure speciali si basano sull'attuazione delle funzioni $pre_\exists(Y) = \{s \in S|$ esiste $s', (s \rightarrow s'$ and $s' \in Y)\}$ e $pre_\forall(Y) = \{s \in S|$ foralls $s', (s \rightarrow s'$ implies $s' \in Y)\}$. "Pre" indica un viaggio all'indietro lungo la relazione di transizione. Entrambe le funzioni calcolano una pre-immagine di un insieme di stati. La funzione $pre_\exists$ (strumento in SAT_{EX} e SAT_{EU}) accetta un sottoinsieme Y di stati e restituisce l'insieme di stati che può effettuare una transizione in Y. La funzione $pre_\forall$,

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
function SAT_EX (φ)
 /* determines the set of states satisfying EX φ */
local var X, Y
begin
    X := SAT (φ);
    Y := pre_∃(X);
    return Y
end
```

Figura 46: La funzione SAT_{EX}

utilizzata in SAT_{AF}, accetta un set Y e restituisce il set di stati che effettua transizioni solo in Y. Si noti che $pre_\forall$ può essere espresso in termini di complementazione e $pre_\exists$, come segue: $pre_\forall(Y) = S - pre_\exists(S - Y)$, dove scriviamo $S - Y$ per l'insieme di tutti gli $s \in S$ che non sono in Y.

La complessità dell'algoritmo è lineare rispetto alla dimensione del modello, ma esponenziale rispetto al numero di variabili e al numero di componenti del sistema. Aggiungere una variabile booleana significa raddoppiare la "complessità" del verificare una proprietà. Questa tendenza dello spazio degli stati di diventare enorme è noto come "State Explosion Problem". La ricerca ha cercato soluzioni a questo problema:

- Uso di strutture dati efficienti
- Uso di astrazione in fase di definizione del sistema in modo tale da eliminare parti irrilevanti ai fini della verifica delle proprietà.
- Partial Order Reduction: in sistemi asincroni, l'ordine di esecuzione di certe azioni può portare a stati equivalenti. Quindi, in fase di verifica è possibile valutare solo una parte delle possibili esecuzioni riducendo lo spazio di ricerca
- Certi sistemi possono essere rappresentativi di famiglie di sistemi reali. Ragionando su casi generici è possibile poi specializzare i risultati ottenuti.
- Divisione del problema in sotto-problemi più semplici (divide-et-impera).

8 Rappresentazione di un automa di Kripke e di una formula CTL con ROBDD (Reduced Ordered Binary Decision Diagrams)

Un diagramma Ordered Binary Decision (BDD) è una struttura di dati che viene utilizzata per rappresentare una funzione booleana. Rappresentare funzioni booleane ha assunto un ruolo importante in diversi ambiti, in quanto questo formalismo logico costituisce un metodo compatto per la descrizione formale. Durante la progettazione di circuiti, per esempio, rappresentare mediante formule logiche la struttura costituisce un modello formale utile per verificarne il comportamento prima della sua realizzazione fisica. Diventa importante avere una rappresentazione efficiente e compatta delle funzioni booleane. La Tabella 6 rappresenta le diverse rappresentazioni esistenti delle formule booleane.

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari
Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Rappresentazione formule booleane	Compact?	Satisfy	Validity	$\wedge$	$\vee$	$\neg$
Prop. formulas	Often	Hard	Hard	Easy	Easy	Easy
DNF Formulas	Sometimes	Easy	Hard	Hard	Easy	Hard
CNF Formulas	Sometimes	Hard	Easy	Easy	Hard	Hard
Tabelle di verità	Never	Hard	Hard	Hard	Hard	Hard
OBDD	Often	Easy	Easy	Medium	Medium	Easy

Tabella 6: Rappresentazioni delle formule boolean

Un albero di decisione è un grafo orientato nel quale:

- ad ogni nodo interno è associato un test su un attributo;
- ciascun ramo uscente da un nodo corrisponde a uno dei possibili risultati del test;
- ciascuna foglia corrisponde all'attribuzione di una classe.

Si tratta di un albero composti da nodi non-terminali, etichettati da variabili booleane, e nodi terminali etichettati con 0 o 1 (true o false).

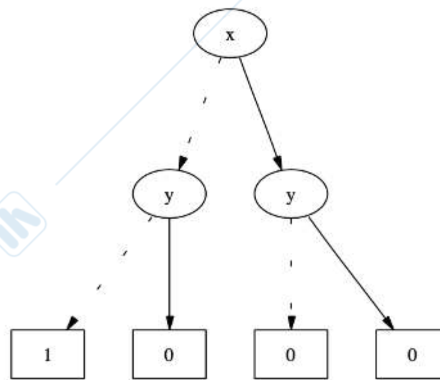


Figura 47: Alberi di decisione binari

Sia T un albero di decisione binario finito. T identifica un'unica funzione booleana delle variabili nei nodi non-terminali nel seguente modo: dato un assegnamento di 0 e 1 alle variabili booleane che occorrono in T, partendo dalla radice dell'albero, si segue la linea tratteggiata quando il valore della variabile del nodo corrente è 0, altrimenti si segue la linea continua. Il valore della funzione è il valore del nodo terminale che si raggiunge.

La rappresentazione mediante BDD è simile a quella tramite tabelle di verità. Infatti, se una funzione f dipende da n variabili booleane, il corrispondente BDD avrà almeno $2^{n+1} - 1$ nodi, mentre la tabella di verità avrebbe 2^n righe. In realtà, è possibile applicare delle riduzioni che compattano in maniera evidente i BDD e li rendono una struttura molto più efficiente. Le regole di riduzione sono tre:

- Rimozione di termini duplicati: se ci sono più nodi terminali con la stessa etichetta è possibile condividerne solo uno, eliminando gli altri.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

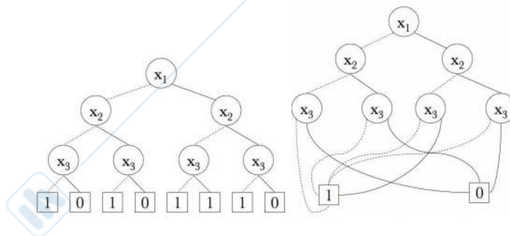


Figura 48: Rimozione di termini duplicati

- Rimozione di nodi con uscite rindondanti: se tutti gli archi uscenti da un nodo n puntano uno stesso nodo m, eliminiamo il nodo n e portiamo tutti i suoi archi entranti su m.

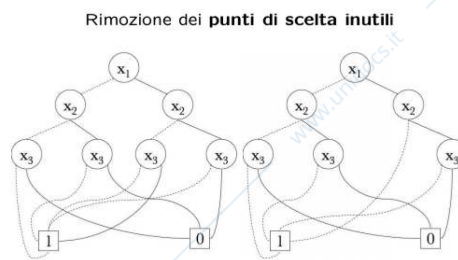


Figura 49: Rimozione di nodi con uscite rindondanti

- Rimozione di non-terminali duplicati: se due sottoalberi di T sono identici, possiamo condividere, come parte comune, uno di questi e eliminare l'altro.

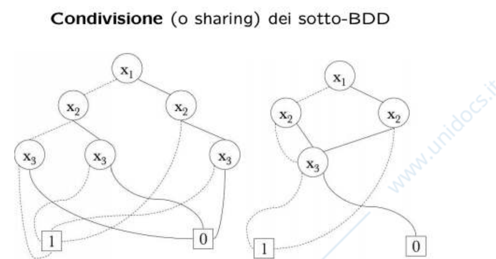


Figura 50: Rimozione di non-terminali duplicati

La Figura 51 mostra esempi di riduzione, applicando le regole C1 e C2.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

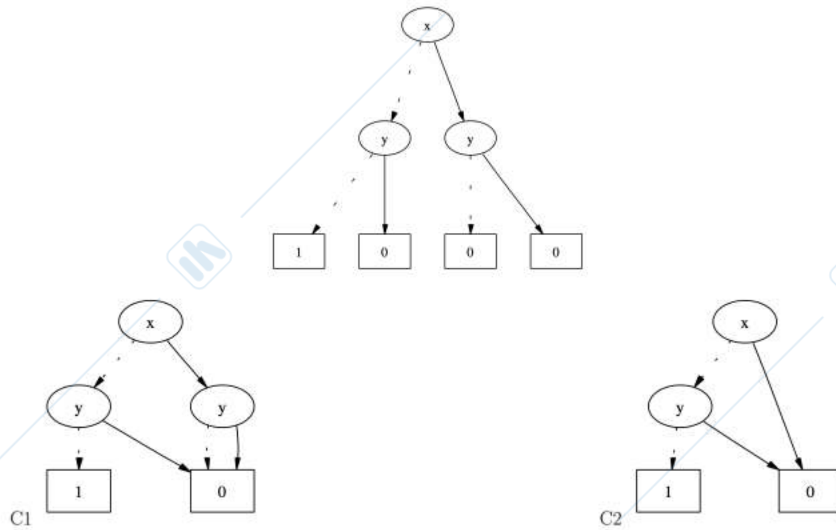


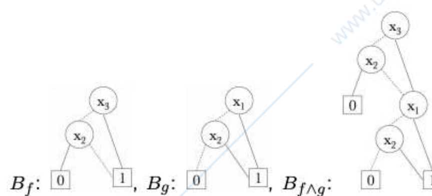
Figura 51: Alberi di decisione binari

Un grafo diretto aciclico (Directed Acyclic Graph - DAG) è un grafo orientato in cui sono assenti cicli. Un nodo di un DAG è iniziale se non ha archi entranti, mentre è terminale se non h archi uscenti.

Un Binary Decision Diagram è un DAG finito con un unico nodo iniziale, in cui i nodi terminali sono tutti etichettati con 0 o 1 e tutti i non-terminali, etichettati con variabili booleane, hanno esattamente due archi uscenti: uno etichetto con 0 e uno etichettato con 1. Un BDD è detto ridotto se nessuna delle riduzioni C1-C3 può essere applicata. B_0 è il BDD composoto dal solo nodo terminale etichettato con 0, mentre B_1 è il BDD composto dal solo nodo terminale etichettato con 1. Le operazioni logiche (AND, OR e NOT) possono essere esguite in maniera banale. Dati due BDD B_f e B_g:

- AND: Il BDD $B_f \wedge B_g$ si ottiene sostituendo i nodi terminali B_1 di B_f con B_g.

Funzioni booleane: $f = x_3 \vee \neg x_2$, $g = x_1 \vee x_2$



Compattezza **non preservata**

Figura 52: Operazione AND

- OR: Il BDD $B_f \vee B_g$ si ottiene sostituendo i nodi terminali B_0 di B_f con B_g.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

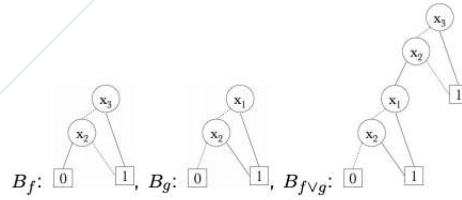


Figura 53: Operazione OR

- NOT: Il BDD $\neg B_f$ si ottiene invertendo i nodi terminali B_0 e B_1.

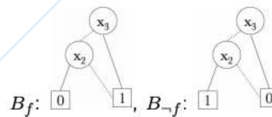


Figura 54: Operazione NOT

I problemi che sorgono dall'utilizzo di questo strumento sono:

- Occorrenza multipla di variabili lungo un path
- Difficoltà nel verificare l'equivalenza tra BDD
- Perdita di efficacia delle riduzioni C1-C3.

Imponendo un ordinamento delle occorrenze delle variabili è possibile ottenere un miglioramento dell'efficienza.

Sia $[x_1, ..., x_n]$ una lista ordinata di variabili senza ripetizioni e sia B un BDD le cui variabili occorrono nella lista. Diciamo che B segue l'ordinamento $[x_1, ..., x_n]$ se tutte le etichette dei nodi non-terminali di B occorrono nella lista e, per ogni occorrenza di x_i seguita da x_j lungo un path in B, vale $i < j$. Un ordered BDD (OBDD) è un BDD che ha un ordinamento definito da una lista di variabili.

Teorema: Esiste un unico reduced OBDD che rappresenta una funzione f. Cioè, siano B e B' due ROBDD con un ordinamento di variabili compatibile. Se B e B' rappresentano una stessa funzione f allora hanno una struttura identica.

Applicando ad un OBDD le regole di riduzione C1-C3 (in qualsiasi ordine), fin quando è possibile, otteniamo come risultato sempre lo stesso ROBDD: cioè, un ROBDD è una forma canonica, fissato un ordinamento, per le funzioni booleane. Le operazioni sugli OBDD non possono essere implementate come avviamo visto in precedenza per mantenere questa proprietà. L'ordine delle variabili scelto incide molto sulle dimensioni dei ROBDD, e quindi sull'efficienza della rappresentazione. Esistono diverse tecniche per calcolare ordinamenti che minimizzano il numero dei nodi. In generale, conviene costruire un OBDD con un ordinamento casuale.

I principali vantaggi dall'utilizzo dei ROBDD come forma canonica per le funzioni sono:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- Assenza di variabili ridondanti: se il valore di una funzione $f(x_1, ..., x_n)$ non dipende da una variabile x_i, allora un ROBDD che rappresenta questa funzione non conterrà nessun nodo etichettato con x_i.
- Test per l'equivalenza semantica: date due funzioni f e g rappresentate da due ROBDD B_f e B_g con ordinamento compatibile, è possibile verificare in maniera efficiente se le due funzioni sono semanticamente equivalenti controllando che i due ROBDD abbiano identica struttura.
- Test di validità: data una funzione f rappresentata da un ROBDD B_f è possibile verificre la validità controllando che B_f sia B_1
- Test di implicazione: date due funzioni f e g rappresentate da due ROBDD B_f e B_g con ordinamento compatibile, è possibile verificare la valitdità dell'implicazione $f \rightarrow g$ calcolando il ROBDD della funzione $B_f \wedge \neg B_g$. Questo è pari a B_0 se e solo se l'implicazione è vera.
- Test di soddisfacibilità: data una funzione f rappresentata da B_f, questa è soddisfacibile se e solo se B_f non è pari a B_0.

8.1 Idea base degli algoritmi per la manipolazione dei ROBDD

Esistono algoritmi efficienti per la composizione di ROBDD:

- **reduce** Riduzione di un OBDD: sia B un OBDD con ordinamento $[x_1, ..., x_n]$ con al più $l+1$ livelli. L'algoritmo **reduce** etichetta con un intero $id(n)$ ogni nodi di B, in modo tale che due sotto alberi con radice n e m rappresentano la stessa funzione se e solo se $id(n)$ è uguale a $id(m)$. L'algoritmo **reduce** analizza inizialmente i nodi terminali e viene assegnato inizialmento a B_0 id pari a $\#0$, e a B_1 id pari a $\#1$ (Regola C1). Successivamente, ipotizziamo che la **reduce** abbia già assegnato i valori di id per i nodi dei livelli $> i$ e concentriamo l'attenzione sui nodi del livello i.

  Definizione: Dato un nodo non-terminale n di un BDD, $lo(n)$ è il nodo puntato dalla linea tratteggiata da n e $hi(n)$ è il nodo puntato dalla linea continua da n. Descriviamo qui di seguito come un nodo n, che rappresenta la variabile x_i, può essere etichettato:

  - Se $id(lo(n)) = id(hi(n))$, allora $id(n)$ è settatto a questo valore (Regola C2).
  - Se esiste un altro nodo m, tale che n e m rappresentano la stessa variabile x_i e $id(lo(m)) = id(lo(n))$ e $id(hi(m)) = id(hi(n))$, allora $id(n)$ viene settato uguale a $id(m)$ (Regola C3).
  - Altrimenti, $id(n)$ viene settato al primo valore intero non ancora utilizzato.

  L'algoritmo termina applicando una ridirezione degli archi coerentemente con le regole C1-C3 eliminando i nodi ridondanti.

- **apply** Applica un operatore a una coppia di ROBDD: Dati due ROBDD B_f e B_g, **apply**(op, B_f, B_g) calcola il ROBDD che rappresenta la funzione $fopg$. Intuitivamente, l'algoritmo opera ricorsivamente sulla struttura dei due ROBDD:

  - Sia v la varibile più alta come ordine (la più sinistra nella lista) che occorre in B_f e B_g.
  - Si divide in due sottoproblemi, per v pari a 0 e v pari a 1, e si risolve ricorsivamente
  - Ai nodi terminali si applica l'operatore

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Per ottenere un ROBDD, in generale, si deve applicare la **reduce**.

Lemma(Shannon Expansion): Per tutte le funzioni booleane f e tutte le variabili booleane x abbiamo che $f \equiv \neg x \wedge f[0/x] \vee x \wedge f[1/x]$. La funzione apply è basata sulla Shannon Expansion della $fopg$. In particolare: $fopg \equiv \neg x_i \wedge (f[0/x_i]opg[0/x_i]) \vee x_i \wedge (f[1/x_i]opg[1/x_i])$.

Chiamiamo r_f ed r_g i nodi radice di B_f e B_g. $B_{(fopg)}$ è calcolato nel seguente modo:

1. Se entrambi r_f ed r_g sono nodi terminali, allora applichiamo l'operatore e restituiamo B_0 o B_1 a seconda del valore calcolato.
2. Negli altri casi, almeno uno tra r_f ed r_g è non-terminale. Assumiamo che entrambi rappresntino la variabile x_i e creiamo un nodo n per x_i con $lo(n) = apply(op, l(r_f), l(r_g))$ e $hi(n) = apply(op, h(r_f), h(r_g))$. Si tratta dell'espansione di Shannon applicata a $fopg$.
3. se, invece, r_f è un nodo x_i ed r_g è terminale o non-terminale x_j con $j > i$, allora è gestito in maniera simmetrica rispetto al punto 3.

La procedura si conclude con una chiamata a reduce.

- **restrict** Calcola il ROBDD di una funzione, fissato il valore di una variabile: Dato un ROBDD B_f, **restrict**$(0, x, B_f)$ calcola il ROBDD che rappresenta $f[0/x]$ utilizzando lo stesso ordinamento di B_f. Per ogni nodo n etichettato con x, gli archi entranti sono rediretti su $lo(n)$ e n è rimosso. Per ottenere un ROBDD bisogna applicare la reduce. In maniera analoga possiamo procedere per calcolare il ROBDD rappresentante $f[1/x]$.
- **exists** Calcola il ROBDD che rappresenta la funzione $\exists x.f = f[0/x] \vee f[1/x]$. Viene usato per calcolare la preimmagine. Possiamo, quindi, calcolare il ROBDD trame l'apply e la restric nel seguente modo: $apply(\vee, restrict(0, x, B_f), restrict(1, x, B_f))$. Possiamo incrementare l'efficienza dell'algoritmo se pensiamo che l'apply è applicata a due ROBDD identici fino al livello che contiene il nodo n etichettato con x. Quindi, l'apply calcola l'applicazione di $\vee$ ai due sottoalberi di n.
- **forall** Calcola il ROBDD che rappresenta la funzione $\forall x.f = f[0/x] \wedge f[1/x]$.

8.2 Symbolic Model Checking = Model Checking + ROBDD

É possibile sfruttare le potenzialità dei ROBDD per l'implementazione delle tecniche di verifica formale. In particolare per rappresentare gli insiemi e i modelli. Le operazioni vengono implementate tramite gli algoritmi visti per ROBDD. Il nome Symbolic Model Checking deriva dal fatto che si utilizza una rappresentazione simbolica, non esplicita.

Invece di enumerare gli stati raggiungibili uno alla volta, lo spazio degli stati a volte può essere attraversato in modo più efficiente considerando un gran numero di stati in una sola fase. Quando tale attraversamento dello spazio degli stati si basa su rappresentazioni di un insieme di stati e relazioni di transizione come formule logiche, diagrammi di decisione binari (BDD) o altre strutture di dati correlate, il metodo di verifica del modello è simbolico.

Ad esempio, l'algoritmo di Labeling può essere, e tipicamente è, realizzato efficientemente sfruttando i ROBDD.

Sia S un insieme finito. Dobbiamo rappresentare i vari sottoinsieme di S mediante ROBDD. Ma i ROBDD rappresentano funzioni booleane, quindi dobbiamo trovare un modo per codificare gli insiemi in formule booleane. Ad ogni elemento $s \in S$ associamo un vettore di valori booleani $(v_1, v_2, ..., v_n)$ tale che ogni $v_i \in \{0, 1\}$. Rappresentiamo un sottoinsieme T mediante la funzione booleana f_T che mappa i $(v_1, v_2, ..., v_n)$ in 1 se $s \in T$ o 0 altrimenti. Abbiamo 2^n vettori di lunghezza n (n deve essere tale che $2^{n-1} < |S| \leq 2^n$). Se $|S|$ non è una potenza esatta di 2, dei

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

vettori non hanno corrispondenza con elementi di S. La funzione $f_T : \{0,1\}^n \rightarrow \{0,1\}$ è quindi la funzione caratteristica di T.

Nel caso in cui S sia l'insieme degli stati di un modello CTL $\mathcal{M}$, possiamo sfruttare la funzione di etichettatura dei nodi L per definire i vettori booleani: fissato un ordinamento sull'insieme degli atomi, ad esempio: $x_1, x_2, ..., x_n$, rappresentiamo uno stato $s \in S$ tramite il vettore $(v_1, v_2, ..., v_n)$ dove, per ogni i, v_i è uguale a 1 se $x_i \in L(s)$ o 0 altrimenti. Deve valere $\forall s_1, s_2 \in S$, se $L(s_1) = L(s_2)$ allora $s_1 = s_2$ (condizione $2^{|Atoms|} \geq |S|$). Uno stato è rappresentato dal ROBDD dalla funzione booleana $l_1 \wedge l_2 \wedge ... \wedge l_n$, dove l_i è x_i se $x_i \in L(s)$ e $\neg x_i$ altrimenti. L'insieme di stati $\{s_1, s_2, ..., s_m\}$ è rappresentato dal ROBDD della funzione booleana $(l_{11} \wedge l_{12} \wedge ... \wedge l_{1n}) \vee (l_{21} \wedge l_{22} \wedge ... \wedge l_{2n}) \vee ... \vee (l_{m1} \wedge l_{m2} \wedge ... \wedge l_{mn})$, dove $l_{i1} \wedge l_{i2} \wedge ... \wedge l_{in}$ rappresenta lo stato s_i. I ROBDD permettono una rappresentazione compatta degli stati.

Implementare le operazioni elementari sugli insieme equivale a utilizzare per intersezione, unione e complementazione rispettivametne le operazioni booleane $\wedge, \vee$ e $\neg$. Defianiamo le altre funzioni che ci serviranno per l'implementazione dell'algoritmo di model checking:

$$pre_\exists(X) = \{s \in S | \exists s', (s \rightarrow s' \text{ e } s' \in X\}$$

$$pre_\forall(X) = \{s \in S | \forall s', (\text{ se } s \rightarrow s' \text{ allora } s' \in X\}$$

La relazione di transizione $\rightarrow$ di un modello CTL $\mathcal{M}$ è un sottoinsieme di $S \times S$. Un elemento sarà rappresentato da una coppia di vettori booleani. $s \rightarrow s'$ è rappresentato da una coppia di vettori booleani: $((v_1, v_2, ..., v_n), (v'_1, v'_2, ..., v'_n))$ con v_i è se $p_i \in L(s)$ e 0 altrimenti; analogalmente per i v'_i. Una transizione è rappresentata dall'OBDD $(l_1 \wedge l_2 \wedge ... \wedge l_n) \wedge (l'_1 \wedge l'_2 \wedge ... l'_n)$ e un insieme di transizione corrisponde all'OR delle formule delle singole transizioni: $f_\rightarrow = (\neg x_1 \wedge \neg x_2 \wedge \neg x'_1 \wedge \neg x'_2) \vee (\neg x_1 \wedge \neg x_2 \wedge x'_1 \wedge \neg x'_2) \vee (x_1 \wedge \neg x_2 \wedge \neg x'_1 \wedge \neg x'_2) \vee (\neg x_1 \wedge x_2 \wedge \neg x'_1 \wedge \neg x'_2)$. Questo equivale alle seguenti coppie di vettori: $(((0,0),(0,0)),((0,0),(1,0)),((1,0),(0,1)),((0,1),(0,0)))$.

Ricordiamo che $pre_\exists(X) = \{s \in S | \exists s', (s \rightarrow s' \text{ e } s' \in X\}$. Dato un ROBDD B_X per l'insieme X e $B_\rightarrow$ per la relazione di transizione $\rightarrow$:

1. Rinominiamo le variabili nella loro *versione primata* in B_X ottenendo $B_{X'}$.
2. Calcoliamo l'OBDD di exists($\hat{x}$, apply($\wedge, B_\rightarrow, B_{X'}$)).

Osservando che $pre_\forall = S - pre_\exists(S - X)$, possiamo sfruttare la costruzione di $pre_\exists$ per calcolare anche $pre_\forall$.

9 Verifica Di Proprietà Temporali

Il problema che ci poniamo è quello di come esprimere le proprietà che dovremo andare a verificare sul nostro sistema, utilizzando delle formule. Queste proprietà, dette per l'appunto proprietà temporali, si suddividono in quattro tipologie.

9.1 Reachability

Esprime il fatto che certe condizioni verranno raggiunte. In logica temporale si esprime con $EF\phi$ ed indica l'esistenza, a partire dallo stato attuale, di un cammino lungo il quale uno stato soddisferà una certa condizione ϕ. Vediamo un po' di esempi esplicativi:

- R1) $EF(n < 0)$: nel futuro esisterà almeno uno stato in cui vale $n < 0$;
- R2) $EFcrit_sec$: nel futuro esisterà almeno uno stato in cui entrerò in una sezione critica (quindi c'è possibilità di entrare in una sezione critica).

Si parla poi di **raggiungibilità condizionale** quando il raggiungimento di una determinata situazione è vincolata al verificarsi di una data condizione. In logica temporale la esprimiamo con EU. Di seguito un esempio:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- R5) $E(n \neq 0)Ucrit_sec$: indica il fatto che posso raggiungere la sezione critica solo dopo aver attraversato uno o più stati in cui vale $n \neq 0$. Letteralmente, nel futuro avrò almeno un cammino in cui $n \neq 0$, fino a quando inizierà la $crit_sec$.

Oltre che a condizioni generiche, la **raggiungibilità** è un concetto che può essere applicato anche **agli stati** e per esprimerlo abbiamo bisogno dell'annidamento $AG(EF)$. Di seguito due esempi che mettono in luce sfumature differenti.

- R6) $EF(init)$: dallo stato di partenza, esiste almeno un cammino futuro che mi riporterà a $init$;

- R7) $AG(EFinit)$: ovunque io sia, qualsiasi stato consideri come partenza, potrò sempre ricondurmi a $init$.

9.2 Safety

Questa *property* esprime invece il concetto "cose cattive non si verificheranno mai". Considerata ϕ come la nostra espressione cattiva negata (ϕ in realtà quindi è già la sua negazione), la esprimiamo con $AG(\phi)$, volendo dire che in nessuno stato globalmente questa si verificherà. Vediamo alcuni esempi:

- S1) $AG\neg(crit_sec1\&crit_sec2)$: esprime il fatto che in nessuno stato il sistema entrerà in due sezioni critiche contemporaneamente;

- S2) $AG\neg(overflow)$: ci dice che il sistema non andrà mai in overflow.

Come nel caso della precedente, anche qui viene definita la **safety condizionale**, che ci dice che il non verificarsi di una data situazione è vincolato dall'occorrenza di un'altra condizione. Si stratta di un rilassamento della proprietà di *Safety* e si esprime come mostrato di seguito.

- S3) $A(\neg startUkey)$: la macchina non partirà fino a quando la chiave non sarà inserita nel blocco d'accensione.

9.3 Liveness

Questa *property* esprime il concetto "cose buone si verificano". Vi sono due modi per esprimerla ed entrambi prevedono un annidamento: $AG + AF$ oppure $AG + EF$. La prima espressione è più forte in quanto da qualunque stato io parta, tutti i cammini mi ricondurranno sempre alla cosa buona che mi attendo. Vediamo alcuni esempi esplicativi.

- L1) $AG(req \rightarrow AFsat)$: in qualsiasi punto mi trovi, ad una richiesta farà sempre seguito, nel futuro, una risposta;

- L2) $AG(EFinit)$: da qualsiasi stato ci sarà sempre almeno un cammino in grado di ricondurmi a $init$;

- L3) $AG(proc1.state = entering \rightarrow AFproc1.state = critical)$: ad ogni richiesta di ingresso in sezione critica, questa verrà prima o poi sempre concessa;

- L4) $AG(AFs.st = sent)$: qualsiasi messaggio verrà prima o poi sempre inviato;

- L5) $AG(AFr.st = received)$: dovunque mi trovi, il messaggio verrà sempre ricevuto in futuro.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

9.4 Assensa di Deadlock

Esprime per l'appunto il fatto che il sistema non si troverà mai in uno stato di *deadlock* e che quindi, ovunque mi trovi, esisterà sempre uno stato successivo. Viene reso con $AG(EXtrue)$. Da notare la presenza di EX, in quanto si vuole che lo stato esistente sia immediatamente successivo a quello attuale. Questa proprietà è una delle più semplici da rendere in logica temporale in quanto ha un'interpretazione immediata e non prevede varianti.

9.5 Proprietà di Fairness

Questa proprietà è utilizzata quando si hanno più processi e più entità che computano in modalità mutualmente esclusiva. Il sitema richiede che solo un agente alla volta possa fare un passo di computazione e la scelta su chi dovrà andare in esecuzione non è deterministica. Sappiamo che in queste situazioni può accadere che un'entità prenda il sopravvento sulle altre, acquisendo sempre la precedenza e dando quindi luogo ad una situazione di *starvation*. La *property* di Fairness vuole evitare proprio questa situazione, garantendo che tutte gli agenti possano eseguire "infinitamente spesso". Purtroppo questa non può essere espressa in logica temporale in quanto non abbiamo un operatore $F\infty$ e dovremmo quindi ricorrere a SMW, andando ad esplicitare questo requisito come parte integrante del modello. Vediamo qualche esempio espresso in linguaggio naturale per avere un'idea.

- F1) Una porta all'interno di un sistema automatizzato dovrà sempre aprirsi infinitamente spesso;
- F2) Qualora una richiesta di ingresso in sessione critica verrà effettuata infinitamente spesso, infinitamente spesso questa dovrà essere soddisfatta.

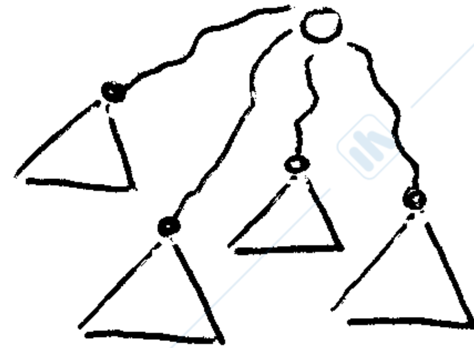
Dopo aver osservato queste proprietà, facciamo un passo indietro e torniamo ad una parte precedentemente tralasciata (slide n. 15 del file `ctl.pdf`). Vediamo insieme gli esempi riportati, con variabili che fanno riferimento a ipotetici modelli già stesi.

- $EF(started \wedge \neg ready)$: il sistema può raggiungere uno stato in cui è partito ma non è pronto;
- $AG(requested \rightarrow AFacknowledged)$: ovunque mi trovi, ad una richiestà farà sempre seguito nel futuro una risposta (proprietà di Liveness);
- $AG(AFenabled)$: in qualsiasi stato mi trovi, il sistema raggiungerà sempre stato in cui è abilitato (proprietà di Liveness);
- $AF(AGdeadlock)$: questa *property* può apparire anomala, in quanto ci dice che il sistema, ovunque ci si posizioni, dovrà in futuro sempre andare in *deadlock*. In realtà ci serve per esprimere il fatto che il sistema possa sempre raggiungere uno stato finale. Vediamo di seguito tre esempi diversi che ci aiutano a ragionare in vista anche di esercizi pratici.

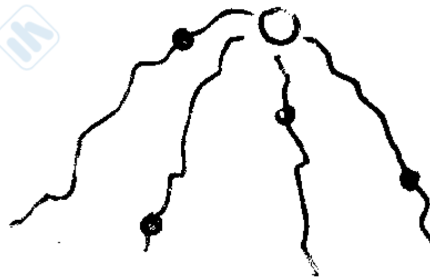
Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

(a) $AF(deadlock)$: ovunque mi posizioni, in futuro il sistema andrà prima o poi sempre in *deadlock*.



(b) $AF(AGdeadlock)$: ovunque mi posizioni, esisterà sempre in futuro un punto a partire dal quale il sistema rimarrà permanentemente in *deadlock*.



(c) $AG(AFdeadlock)$: da qualunque stato io parta, in futuro esisterà sempre uno stato in cui il sistema va in *deadlock*.

Volendo stilare una classifica crescente in relazione alla forza di queste proprietà, otteniamo: Figure 55a, Figure 55c, Figure 55b.

- $AG(EFrestart)$: in qualsiasi momento, è sempre possibile raggiungere uno stato del sistema in cui viene eseguita una *restart*;
- $AG(floor = 2 \wedge direction = up \wedge ButtonPressed5 \rightarrow A[direction = upU floor = 5])$: in qualsiasi momento, se siamo al secondo piano, stiamo salendo e vogliamo andare al piano 5, l'ascensore raggiungerà il piano desiderato senza cambiare direzione di marcia;

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- $AG(floor = 3 \wedge idle \wedge door = closed \rightarrow EF(floor = 3 \wedge idle \wedge door = closed))$: in qualsiasi momento, se l'ascensore è al terzo piano, è libero e le porte sono chiuse, è possibile che l'ascensore rimanga bloccato.

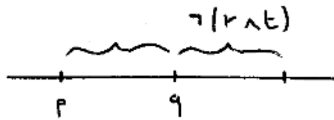
10 Esercizi 3.3 pag 165 (Soluzioni)

ESERCIZI 3.3 PAG. 105

1. Se p è seguito da q (dopo un numero finito di passi), allora il sistema entra in un "intervallo" in cui r e t non occorrono.

SVOLGIMENTO

Aiutiamoci con un' intuizione.



intervallo in cui r e t non valgono fino a quando tornano ad occorrere

$A[p \; U \; EF q] \rightarrow A[\neg(r \wedge t) \; U \; (r \vee t)]$

da notare che qui potremmo mettere anche E a seconda di quanto forte interpretiamo le specifiche => N.B.: se nel linguaggio naturale emergono ambiguità, vanno bene sia A che E

* nel compito, se possibile, spiegare la interpretazione!

(a) Esercizio 1

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

2. Considerate la formula AG(p -> AF(s ∧ AX(AF t))). Cosa esprime in termini di occorrenze degli eventi p, s, t.

SVOLGIMENTO

In ogni momento, se vale p, allora in futuro varrà s e poi, successivamente, varrà t. Quindi

(b) Esercizio 2

3. Scrivere una formula CTL per descrivere il fatto che p precede s e t in tutti i path. Suggerimento: provate a pensare alla negazione di questa specifica.

SVOLGIMENTO

Idea iniziale

(N.B.: non c'è scritto che s e t devono accadere insieme)

AG(p -> (AX AF s ∧ AX AF t)

da qualsiasi parte inizi: se non lo avessi, varrebbe solo da stato in cui mi trovo!

forziamo il fatto che debba fare un passo: se non lo avessi potrei avere s e t nello stesso stato di p!

(c) Esercizio 3

11 NuSMV

NuSMV è una reimplementazione e un'estensione del controllore di modelli simbolici SMV, il primo strumento di controllo dei modelli basato sui diagrammi di decisione binari (BDD). Lo strumento è stato progettato come un'architettura aperta per il controllo del modello. É finalizzato alla verifica affidabile di progetti di dimensioni industriali, da utilizzare come back-end per altri strumenti di verifica e come strumento di ricerca per tecniche di verifica formale. NuSMV supporta l'analisi delle specifiche espresse in CTL e LTL. L'interazione dell'utente viene eseguita con un'interfaccia testuale, nonché in modalità batch. NuSMV può essere utilizzato per

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

verificare se i vincoli LTL o CTL predefiniti sono validi per il modello definito. Ad esempio, abbiamo una specifica CTL che vogliamo controllare: CTLSPEC EF (proc5.state = critico). Se la specifica è vera, NuSMV ti informerà come nella Figura 57.

```
-- specification EF proc5.state = critical  is true
>NuSMV
```

Figura 57: Output del controllo di NuSMV

Qualsiasi modello di controllo fornisce almeno:

- Un linguaggio di input per descrivere il modello M_s e la proprietà da verificare (SMV). Consente la descrizione di:
  - Sistemi sincroni e asincroni
  - Descrizioni modulari e gerarchiche
  - Tipi di dati finiti: booleani ed enumerati
  - Non determinismo
- Un meccanismo automatizzato per il controllo di $M_s \models \phi$

Il linguaggio nuSMV prevede un modulo "main". Tre parti di codice, identificate da VAR, ASSIGN, SPEC:

- VAR identifica una parte del codice in cui sono definite le variabili. I possibili tipi sono:
  - Booleani: 1 se vero, 0 se falso
  - Enumerabili:
    - a: rosso, blu, verde;
    - b: 1, 2, 3;
    - c: 1, 5, 7;

    Le operazioni numeriche devono essere adeguatamente protette
- ASSIGN identifica una porzione di codice in cui le variabili sono inizializzate e viene descritta l'evoluzione. L'assegnazione avviene a due livelli:
  - Assegnazione allo stato iniziale: init (valore): = 0;
  - Assegnazione allo stato successivo (relazione di transizione): next (valore): = valore + carry_in mod 2;
- SPEC definisce le proprietà da verificare.

La Figura 58 mostra un esempio di automa di Kripke trasformato in un modello nuSMV. Osserviamo ora la differenza tra DEFINE e ASSIGN:

- ASSIGN $a := b|c$, dove a è un valore booleano, dichiara una nuova variabile di stato a e diventa parte della relazione invariante.
- DEFINE $d := b|c$ è effettivamente una definizione macro, ogni occorrenza di d è sostituita da $b|c$. Non viene generata alcuna variabile BDD aggiuntiva per d. Infatti, DEFINE non consuma memoria, al contrario di ASSIGN. Il BDD per $b|c$ diventa parte di ogni espressione usando d.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

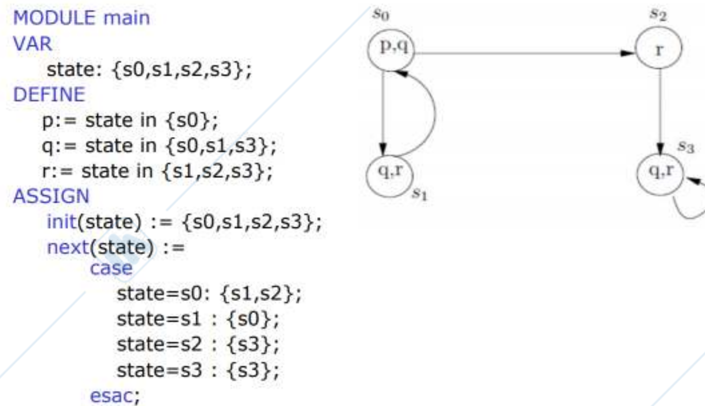


Figura 58: Automa di Kripke in NuSMV

Le espressioni possono fare riferimento al valore di una variabile nello stato successivo. La Figura 59 mostra un esempio in questo senso.

```
VAR a,b : boolean;
ASSIGN
        next(b) := !b;
        a := next(b);
```

Figura 59: *next* in NuSMV

Nell'espressione "case" le guardie vengono valutate in sequenza, ossia la prima guardia soddisfatta determina il valore risultante. Se nessuna delle guardie è vera, viene restituito un valore arbitrario valido: TRUE: else expr.

La variabile completamente non assegnata può modellare input non vincolati. $\{val_1$, ..., $val_n\}$ è un'espressione che assume uno qualsiasi dei valori dati in modo non deterministico. La scelta non deterministica può essere utilizzata per modellare un'implementazione che non è stata ancora perfezionata o un comportamento astratto. Lasciare molto non determinismo è oneroso per il model checker. Infatti, per quanto riguarda l'esempio in Figura 60, senza altre variabili, il sistema ha 4x2x24 = 128 possibili stati.

```
MODULE main
VAR
 cabin: 0..3;
 dir: {up, down};
 request: array 0..3 of boolean;
```

Figura 60: *next* in NuSMV

Il problema del ferryman è composto da: un traghettatore, una capra, un cavolo e un lupo, i quali sono su un lato di un fiume. Il traghettatore può attraversare il fiume con al massimo un passeggero nella barca. Esiste un conflitto di comportamento tra:

- La capra e il cavolo
- La capra e il lupo

Il traghettatore può trasportare tutte le merci dall'altra parte senza alcun conflitto? Vengono utilizzati quattro agenti: traghettatore, capra, lupo, cavolo. La posizione di ogni variabile è modellata da un valore booleano: 0 indica che l'agente è sul lato iniziale, mentre 1 indica che l'agente è sul lato finale. La variabile carry indica quale bene viene trasportato dal traghettatore. La Figura 61 mostra l'inizializzazione delle variabili.

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari
Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
MODULE main
VAR
   ferryman : boolean;
   goat : boolean;
   cabbage : boolean;
   wolf : boolean;
   carry : {g,c,w,0};
ASSIGN
   init(ferryman) := 0;
   init(goat) := 0;
   init(cabbage) := 0;
   init(wolf) := 0;
   init(carry) := 0;
```

Figura 61: The Ferryman problem

Attraverso $next(ferryman) := \{0, 1\}$; il traghettatore può decidere di attraversare il fiume, oppure no. Il valore di carry non è deterministico, ma determinato dal valore di ferryman, capra, lupo, cavolo.

La Figura 62 mostra come viene modellata la funzione next(carry). g è un membro dell'insieme dal quale viene scelto il next(carry).

```
next(carry) := case
    ferryman=goat : g;
    1 : 0;
    esac union
    case ferryman=cabbage : c;
    1 : 0;
    esac union
    case ferryman=wolf : w;
    1 : 0;
    esac union 0;
```

Figura 62: The Ferryman problem

Il valore successivo di capra, cavolo, lupo è deterministico, poiché il fatto che siano trasportati o meno è determinato dalla scelta del traghettatore rappresentata da carry.

```
next(goat) := case
   ferryman=goat & next(carry)=g : next(ferryman);
   1 : goat;
   esac;
next(cabbage) := case
   ferryman=cabbage & next(carry)=c : next(ferryman);
   1 : cabbage;
   esac;
next(wolf) := case
   ferryman=wolf & next(carry)=w : next(ferryman);
   1 : wolf;
   esac;
```

Figura 63: The Ferryman problem

Vogliamo trovare un percorso soddisfacente:

- per la condizione di raggiungibilità finale: (cavolo & capra & lupo & ferryman)
- nelle condizioni di sicurezza: (capra = cavolo | capra = lupo) → capra = ferryman

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

La specifica finale è SPEC E[((capra = cavolo | capra = lupo) → capra = ferryman) ∪ (cavolo & capra & lupo & ferryman)). Eseguiamo SMV con la negazione della proprietà sperando di trovare un contro esempio.

SMV trova un percorso infinito che si snoda attorno ai 15 stati. Lungo il percorso infinito, il farryman porta ripetutamente i suoi beni verso la destinazione (in modo sicuro), e poi di nuovo indietro (in modo non sicuro). La struttura afferma la sicurezza del viaggio di andata, ma non dice nulla su ciò che accade dopo.

La proprietà corretta è SPEC E[((capra = cavolo | capra = lupo) → capra = traghettatore) ∪ (cavolo & capra & lupo & ferryman) & AG (capra → AG capra)]. La capra compie almeno tre viaggi e, una volta attraversato il fiume per la terza volta, rimane alla destinazione.

Un programma SMV può essere composto da più moduli, come mostra la Figura 64.

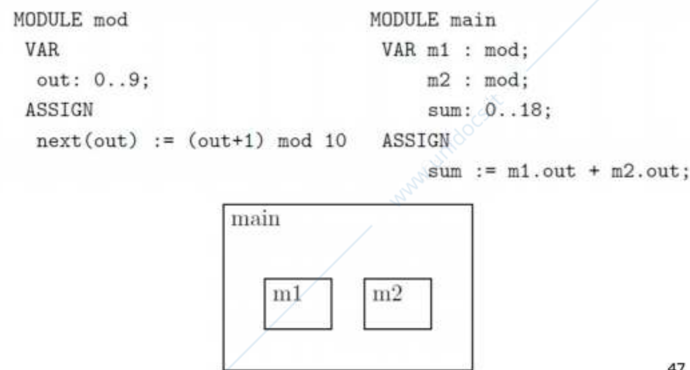


Figura 64: Moduli NuSMV

I moduli possono essere istanziati più volte, ogni istanza crea una copia delle variabili locali. Ogni programma ha un modulo principale, chiamato main. La Figura 65 mostra l'istanziazione doppia di un modulo.

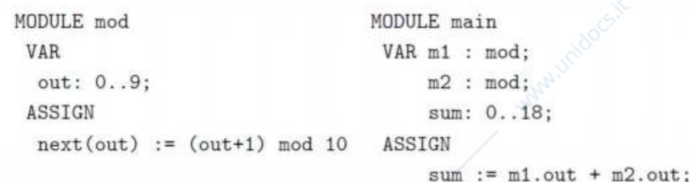


Figura 65: Moduli NuSMV

Le variabili dichiarate all'esterno di un modulo possono essere passate come parametri. Le variabili interne di un modulo possono essere utilizzate in moduli racchiusi (submodel.varname). I parametri vengono passati per riferimento.

Figura 66: Moduli NuSMV

La composizione di moduli avviene in due modalità:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- Composizione sincrona: Tutte le assegnazioni vengono eseguite in parallelo e in modo sincrono (un orologio globale). Un singolo passaggio del modello risultante corrisponde a un passaggio in ciascuno dei componenti.

- Composizione asincrona: Un passaggio della composizione è un passaggio di esattamente un processo. Ad ogni ticchettio dell'orologio viene scelto in modo non deterministico un processo ed eseguito per un ciclo. Le variabili, non assegnate in tale processo, rimangono invariate.

Per forzare l'esecuzione di un processo all'infinito spesso, possiamo usare un vincolo di equità (fairness). Un vincolo di equità limita l'attenzione del model checker solo a quei percorsi di esecuzione lungo i quali una data formula è vera all'infinito spesso. Ogni processo ha una variabile speciale chiamata RUNNING che è 1 se e solo se quel processo è attualmente in esecuzione. Aggiungendo la dichiarazione:

FAIRNESS

RUNNING

possiamo forzare efficacemente ogni istanza di inverter per l'esecuzione all'infinito spesso. Le FAIRNESS ctl_formulae consistono nel presupporre che sia vera infinitamente spesso. Il model checker esplora solo percorsi che soddisfano il vincolo di equità. Ogni vincolo di equità deve essere vero infinitamente spesso.

Un altro esempio di modello asincrono: utilizza una variabile semaforo per implementare la mutua esclusione tra due processi asincroni. Ogni processo ha quattro stati: inattivo, in entrata, critico ed in uscita. Lo stato di entrata indica che il processo vuole entrare nella sua area critica. Se la variabile semaforo è 0, passa allo stato critico e imposta il semaforo su 1. All'uscita dalla sua area critica, il processo imposta nuovamente il semaforo su 0. La Figura 67 mostra il funzionamento di questo semaforo.

```
MODULE main
  VAR
     semaphore : boolean;
     proc1 : process user(semaphore);
     proc2 : process user(semaphore);
  ASSIGN
     init(semaphore) := 0;
MODULE user(semaphore)
  VAR
     state : {idle, entering, critical, exiting};
  ASSIGN
     init(state) := idle;

  next(state) := case
     state = idle : {idle, entering};
     state = entering & !semaphore : critical;
     state = critical : {critical, exiting};
     state = exiting : idle;
     TRUE : state;
     esac;
  next(semaphore) := case
     state = entering : 1;
     state = exiting : 0;
     TRUE : semaphore;
     esac;
FAIRNESS
  running
```

Figura 67: Mutua esclusione

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Per la mutua esclusione, si richiedono le proprietà di:

- Safety: un solo processo è nella sua sezione critica in qualsiasi momento. AG ! (proc1.state = critical & proc2.state = critical)
- Liveness: ogni volta che un processo richiede di entrare nella sua sezione critica, alla fine sarà autorizzato a farlo.
  - AG (proc1.state = entering -> EF proc1.state = critical)
  - AG (proc2.state = entering -> EF proc2.state = critical)

12 AsmetaSMV: un model checker per modelli AsmetaL

Gli obiettivi che si prefigge AsmetaSMV sono:

- Sfruttare le potenzialità del model checker NuSMV nel framework Asmeta;
- evitare di scrivere due modelli, uno ad alto livello per la simulazione (AsmetaL) ed uno a basso livello per la verifica formale (NuSMV), per evitare perdite di tempo e il rischio di scrivere due modelli non equivalenti.
- permettere all'utente di definire proprietà CTL e LTL direttamente nel modello AsmetaL: l'utente può anche non conoscere la sintassi di NuSMV, ma deve solo conoscere gli operatori temporali.

Per poter utilizzare il tool AsmetaSMV devo seguire i seguenti passi:

1. Scrittura del codice AsmetaL;
2. Traduzione del codice AsmetaL in un codice NuSMV;
3. Esecuzione del codice NuSMV con il model checker.

La Figura 68 mostra l'architettura di AsmetaSMV.

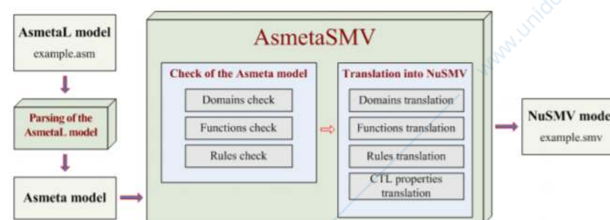


Figura 68: Architettura AsmetaSMV

Il passaggio di check all'interno del blocco AsmetaSMV è necessario perché AsmetaSMV non può tradurre qualsiasi costrutto di AsmetaL in NuSMV; diciamo che un costrutto è supportato se può essere tradotto, altrimenti che non lo è. Un costrutto AsmetaL non è supportato perché richiede condizioni che non possono essere soddisfatte in NuSMV. Per esempio, i type domains Real, Integer e Char non hanno un tipo corrispettivo in NuSMV e, quindi, non possono essere utilizzati come domini o codomini di funzioni in modelli AsmetaL che devono essere tradotti. La traduzione sarebbe troppo complicata.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

12.1 Domini e funzioni

In AsmetaL sono supportati i seguenti domini:

- type domains: basic type domains: Complex, Real, Integer, Natural, String, Char, Boolean, Rule and Undef;
- structured type domains: ProductDomain, SequenceDomain, PowersetDomain, BagDomain and MapDomain;
- abstract type-domains: sono domini generici;
- enum domains.
- concrete domains: sottoinsiemi di concrete domains

La mappatura è possibile solo per i domini supportati, ossia il cui tipo sia presente in NuSMV, di cardinalità finita:

- Boolean;
- enum domain;
- concrete domains di Integer e Natural.

Il Boolean domain è pienamente supportato; può essere usato sia nel dominio, sia nel codominio di una funzione AsmetaL. Il tipo corrispondente in NuSMV è boolean (con la prima lettera minuscola). I letterali di AsmetaL $true$ e $false$ diventano le costanti simboliche $TRUE$ e $FALSE$ in NuSMV (non esiste il corrispettivo dell'$undef$). La Figura 69 mostra un esempio di mappatura di booleani.

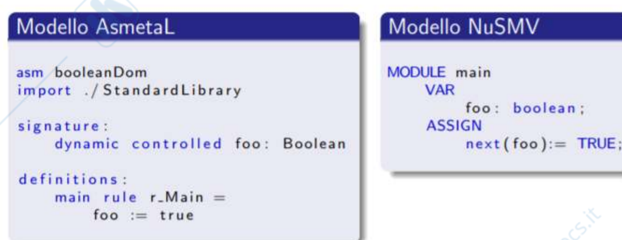


Figura 69: Mappatura booleani

In AsmetaL, gli enum domains sono dichiarati nell'header nel seguente modo: *enum domain* $Dom = \{EL_1|...|EL_n\}$, dove Dom è il nome dell'enum domain e $EL_1, ..., EL_n$ sono i suoi elementi; l'identificativo dell'elemento deve essere una stringa di almeno due lettere maiuscole. Il tipo enum in NuSMV ha una sintassi simile: $\{EL_1, ..., EL_n\}$. Nella traduzione, i nomi degli enumerativi non cambiano. Viene aggiunto un elemento EL_{undef} per gestire il valore $undef$. La Figura 70 mostra un esempio di mappatura di domini enumerati.

Sono supportati solo i concrete domains i cui type domains sono Integer o Natural. In AsmetaL, gli elementi di un concrete domain possono essere dichiarati in due modi: $\{a_1 : a_n\}$ se il dominio contiene tutti i numeri compresi nell'intervallo $[a_1, a_n]$, oppure $\{a_i, a_j, ..., a_k\}$ se il dominio contiene un insieme di numeri che non sono necessariamente contigui. I concrete domains di AsmetaL diventano enum types in NuSMV. Sia i concrete domains di Integer, sia quelli di Natural diventano enum types di interi in NuSMV. NuSMV, infatti, non ha il tipo Natural. Viene aggiunto il valore -2147483647 per gestire il valore $undef$: la mappatura è corretta solo se -2147483647 non è un valore del dominio di partenza. La Figura 71 mostra un esempio di mappatura di domini concreti.

Per ogni funzione AsmetaL, AsmetaSMV crea tante variabili in NuSMV quante sono le locazioni della funzione:

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

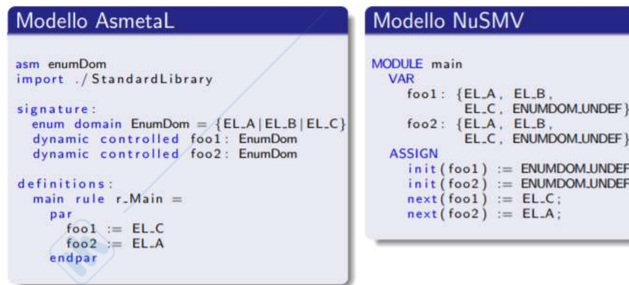


Figura 70: Mappatura enum

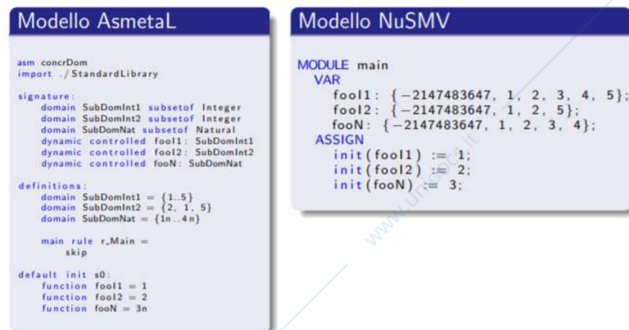


Figura 71: Mappatura concrete

- Una funzione AsmetaL di arietà zero viene mappata in una variabile NuSMV con lo stesso nome;
- una funzione AsmetaL di arietà uno viene mappata in k variabili, dove k è la cardinalità del dominio della funzione. Il nome di tali variabili è: $idFunc_elDom$, dove $idFunc$ è il nome della funzione ed $elDom$ è un elemento del dominio della funzione.
- una funzione AsmetaL di arietà maggiore di uno viene mappata in $m = \Pi_{i=1}^{n}|D_i|$ variabili, dove D_i è un elemento del Product domain usato come dominio della funzione. I nomi delle variabili sono: $idFunc_elDom_1_..._elDom_n$, dove $elDom_1 \in D_1, ..., elDom_n \in D_n$.

La cardinalità del dominio di una funzione deve essere finito, perché NuSMV può avere un numero finito di variabili. Il tipo di ogni variabile è ottenuto tramite la mappatura del codominio della funzione AsmetaL. Le locazioni inutilizzate in AsmetaL (locazioni che non sono né aggiornate né lette) non sono esportate in NuSMV.

La Figura 72 mostra un esempio di funzione di arietà 1.

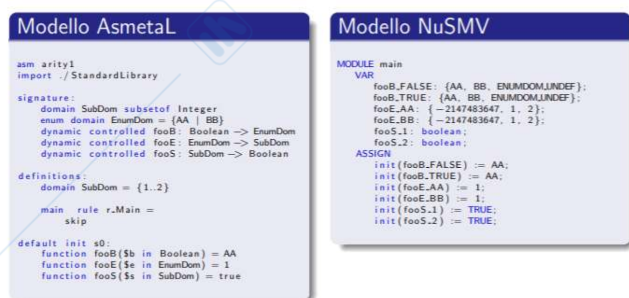


Figura 72: Mappatura funzione di arietà 1

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

La Tabella 7 mostra il processo di mappatura, in riferimento alla Figura 72 in dettaglio:

Funzione AsmetaL	Locazioni AsmetaL	Variabili NuSMV
fooB($b in Boolean)	fooB(false)	fooB_FALSE
	fooB(true)	fooB_TRUE
fooE($e in EnumDom)	fooE(AA)	fooE_AA
	fooE(BB)	fooE_BB
fooS($s in SubDom)	fooS(1)	fooS_1
	fooS(2)	fooS_2

Tabella 7: Processo di mappatura della Figura 72

La Figura 73 mostra un esempio di funzione di arietà maggiore di 1. La Tabella 8 mostra il

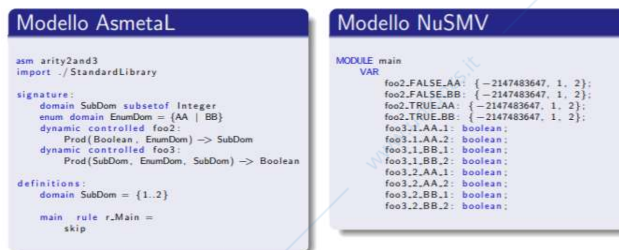


Figura 73: Mappatura funzione di arietà maggiore di 1

processo di mappatura, in riferimento alla Figura 73 in dettaglio:

Funzioni AsmetaL	Locazioni AsmetaL	Variabili NuSMV
foo2($b in Boolean, $e in EnumDom)	foo2(false, AA)	foo2_FALSE_AA
	foo2(false, BB)	foo2_FALSE_BB
	foo2(true, AA)	foo2_TRUE_AA
	foo2(true, BB)	foo2_TRUE_BB
foo3($i in SubDom, $e in EnumDom $j in SubDom)	foo3(1, AA, 1)	foo3_1_AA_1
	foo3(1, AA, 2)	foo3_1_AA_2
	foo3(2, AA, 1)	foo3_2_AA_1
	foo3(2, AA, 2)	foo3_2_AA_2
	foo3(1, BB, 1)	foo3_1_BB_1
	foo3(1, BB, 2)	foo3_1_BB_2
	foo3(2, BB, 1)	foo3_2_BB_1
	foo3(2, BB, 2)	foo3_2_BB_2

Tabella 8: Processo di mappatura della Figura 73

Le funzioni ASM sono classificate in:

- basic: funzioni che costituiscono la segnatura base dell'ASM. Le funzioni basic sono classificate in:
  - Static: non cambiano mai il valore durante ogni esecuzione della macchina; il loro valore, infatti, non dipende dallo stato della macchina.
  - Dynamic: possono cambiare il proprio valore come conseguenza delle azioni di alcuni agenti (o updates) o dell'environment. Sono classificate in:
    * Controlled: sono aggiornabili in modo diretto solo da istruzioni della macchina (transition rules).

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

  * Monitored (o in): sono lette ma non sono aggiornate dalla macchina; sono aggiornate solo dall'environment.
  * Shared: possono essere aggiornate e lette da pi'u di un agente.
  * Output: sono aggiornate, ma non lette dalla macchina; sono solo monitorate dagli altri agenti o dall'environment.

- derived: funzioni ausiliarie che sono calcolate in base al valore di altre funzioni basiche.

Il tool AsmetaSMV supporta solo funzioni controllate e monitorate. Chiamiamo variabili controllate le variabili ottenute dalla mappatura di una funzione controllata e variabili monitorate le variabili ottenute dalla mappatura di una funzione monitorata.

In AsmetaL è possibile che una locazione sia identificata usando come argomento di una funzione un'altra funzione. Tali modelli non possono essere tradotti da AsmetaSMV. La Figura 74 mostra una possibile soluzione a questa mancanza di supporto. Per risolvere il problema si può sostituire l'argomento usando la let rule.

Esempio di modello AsmetaL non supportato

```
asm functionAsArg
import ./StandardLibrary

signature:
        enum domain EnumDom = {AA | BB | CC}
        dynamic monitored monArg: EnumDom
        dynamic controlled contrArg: EnumDom
        dynamic controlled foo: EnumDom -> Boolean
        dynamic controlled foo2: EnumDom -> Boolean

definitions:
        main  rule r_Main =
                par
                        contrArg := AA
                        foo(monArg) := true //Not supported by AsmetaSMV
                        foo2(contrArg) := true //Not supported by AsmetaSMV
                endpar

default init s0:
        function contrArg = BB
```

Possibile soluzione

```
asm functionAsArgSupported
import ./StandardLibrary
signature:
        enum domain EnumDom = {AA | BB | CC}
        dynamic monitored monArg: EnumDom
        dynamic controlled contrArg: EnumDom
        dynamic controlled foo: EnumDom -> Boolean
        dynamic controlled foo2: EnumDom -> Boolean
definitions:
        main  rule r_Main =
                par
                        contrArg := AA
                        let ($x = monArg, $y = contrArg) in
                                par
                                        foo($x) := true //Supported by AsmetaSMV
                                        foo2($y) := true //Supported by AsmetaSMV
                                endpar
                        endlet
                endpar
default init s0:
        function contrArg = BB
```

Figura 74: Funzioni di funzioni

Le funzioni controllate sono le uniche funzioni le cui locazioni possono essere aggiornate in una transition rule. Durante l'esecuzione di una macchina ASM, in ogni stato, viene calcolato l'update set, cioè l'insieme di update che possono essere eseguiti. Un update set viene definito consistente se non ci sono locazioni che sono aggiornate contemporaneamente a due valori differenti.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Update consistente

```
asm consistentUpdate
import ./StandardLibrary

signature:
    dynamic controlled foo: Boolean
    dynamic controlled foo1: Boolean

definitions:
    main rule r_Main =
        par
            foo := true
            foo := true
            foo1 := false
        endpar
```

Update inconsistente

```
asm notConsistentUpdate
import ./StandardLibrary

signature:
    dynamic controlled foo: Boolean

definitions:
    main rule r_Main =
        par
            foo := true
            foo := false
        endpar
```

Figura 75: Update consistente e update inconsistente

AsmetaSMV calcola tutti gli update set che possono essere eseguiti durante l'esecuzione di una macchina ASM; tali update set sono uniti in un unico global update set: per ogni locazione, i valori a cui può essere aggiornata sono elencati; ogni valore è associato alla condizione che deve essere soddisfatta affinché la locazione sia aggiornata a quel valore. Il global update set viene mappato nella sezione ASSIGN del modello NuSMV. Per ogni variabile (locazione in AsmetaL) il valore dello stato successivo viene determinato tramite l'espressione case, in cui sono riportate tutte le condizioni con i corrispondenti valori. Per ogni variabile, una condizione di default ($TRUE$) mantiene il valore immutato.

Modello AsmetaL

```
asm update
import ./StandardLibrary

signature:
    enum domain EnumDom = {AA | BB | CC}
    dynamic monitored mon: Boolean
    dynamic controlled foo: EnumDom
    dynamic controlled foo1: EnumDom

definitions:
    main rule r_Main =
        if(mon) then
            par
                foo := AA
                foo1 := CC
            endpar
        else
            par
                foo := BB
                foo1 := AA
            endpar
        endif
```

Update sets

mon = true

Locazione	Valore
foo	AA
foo1	CC

mon = false

Locazione	Valore
foo	BB
foo1	AA

Global update set

Locazione	Condizione	Valore
foo	mon	AA
	!mon	BB
foo1	mon	CC
	!mon	AA

Global update set

Locazione	Condizione	Valore
foo	mon	AA
	!mon	BB
foo1	mon	CC
	!mon	AA

Modello NuSMV

```
MODULE main
    VAR
        foo: {AA, BB, CC, ENUMDOM_UNDEF};
        foo1: {AA, BB, CC, ENUMDOM_UNDEF};
        mon: boolean;
    ASSIGN
        init(foo) := ENUMDOM_UNDEF;
        init(foo1) := ENUMDOM_UNDEF;
        next(foo) :=
            case
                mon: AA;
                !mon: BB;
                TRUE: foo;
            esac;
        next(foo1) :=
            case
                mon: CC;
                !mon: AA;
                TRUE: foo1;
            esac;
```

Figura 76: Funzioni controllate - Global update set

Se un modello AsmetaL contiene un update inconsistente, il parser AsmetaLc non può scoprirlo: gli update inconsistenti sono errori semantici, non sintattici. Il simulatore AsmetaS, invece, se incontra un update inconsistente durante la simulazione è in grado di segnalarlo. AsmetaSMV non può risolvere il problema degli update inconsistenti: se un modello AsmetaL contiene un update inconsistente, anche il modello NuSMV lo conterrà. Durante l'esecuzione del

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

codice NuSMV, inoltre, il model checker non segnala alcun errore: semplicemente aggiorna la variabile al primo valore la cui condizione viene soddisfatta (vedere l'esempio nella Figura 77).

Global update set

```
asm notConsistent
import ./StandardLibrary

signature:
    enum domain EnumDom = {AA | BB | CC}
    dynamic monitored mon: Boolean
    dynamic monitored mon2: Boolean
    dynamic controlled foo: EnumDom

definitions:
    main rule r_Main =
        par
            if(mon != mon2) then
                foo := AA
            endif
            if(mon2 != mon) then
                foo := BB
            endif
        endpar
```

Modello NuSMV

```
MODULE main
    VAR
        foo: {AA, BB, CC, ENUMDOM_UNDEF}; --controlled
        mon: boolean; --monitored
        mon2: boolean; --monitored
    ASSIGN
        init(foo) := ENUMDOM_UNDEF;
        next(foo) :=
            case
                (mon2 != mon): BB;
                (mon != mon2): AA;
                TRUE: foo;
            esac;
```

Nota

Nel modello NuSMV, se il valore di *mon* è diverso dal valore di *mon2*, la variabile *foo* viene aggiornata al valore *BB*, cioè il valore associato alla prima condizione soddisfatta.

Figura 77: Funzioni controllate - Update set inconsistente

In AsmetaL, le funzioni monitorate sono funzioni il cui valore viene aggiornato dall'environment. In NuSMV le variabili monitorate sono dichiarate, ma non sono né inizializzate né aggiornate. Quando NuSMV incontra una variabile monitorata, crea uno stato per ogni valore che la variabile può assumere; eseguiamo il seguente codice (Figura 78) con l'opzione '-r' che stampa il numero di stati raggiungibili: ci sono quattro stati raggiungibili che corrispondono ai quattro valori che la variabile *mon* può assumere.

Variabile monitorata con 4 valori

```
MODULE main
    VAR
        mon: 1..4;
```

NuSMV execution

```
[user@localhost asmetasmv]$ NuSMV -r numStatesMon.smv
*** This is NuSMV 2.5.2 (compiled on Sat Oct 30 12:18:33 UTC 2010)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>
system diameter: 1
reachable states: 4 (2^2) out of 4 (2^2)
```

Figura 78: Funzioni monitorate

In AsmetaL, le funzioni statiche e quelle derivate non possono essere aggiornate né in una update rule né dall'environment. Il loro valore (meccanismo di computazione) viene impostato nella sezione definitions e non cambia mai durante l'esecuzione della macchina. Le funzioni statiche non dipendono dallo stato della macchina, a differenza delle funzioni derivate. AsmetaSMV non fa differenza tra le funzioni statiche e quelle derivate: la loro mappatura è identica. In NuSMV le funzioni statiche e derivate sono espresse con la dichiarazione DEFINE. Nella Figuta 79 la funzione statica *stat* e la funzione derivata *der* sono state mappate in due definizioni nel codice NuSMV.

Modello AsmetaL

```
asm staticDerived
import ./StandardLibrary

signature:
    domain MyDomain subsetof Integer
    dynamic monitored mon1: Boolean
    dynamic monitored mon2: Boolean
    static stat: MyDomain
    derived der: Boolean

definitions:
    domain MyDomain = {1..4}

    function stat = 2
    function der = mon1 and mon2

    main rule r_Main =
        skip
```

Modello NuSMV

```
MODULE main
    VAR
        mon1: boolean;
        mon2: boolean;
    DEFINE
        stat:= 2;
        der:= (mon1 & mon2);
```

Figura 79: Funzioni statiche e derivate - Esempio

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari
Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Per ottenere un codice NuSMV corretto, le funzioni statiche e quelle derivate con codominio Boolean non possono assumere valore $undef$. Nel seguente codice (Figura 80) quando $mon1$ è false la funzione non è $undef$.

Modello AsmetaL

```
asm derivedNotExhaustive
import ./StandardLibrary
signature:
    dynamic controlled foo : Boolean
    dynamic monitored mon1: Boolean
    dynamic monitored mon2: Boolean
    derived der: Boolean

definitions:
    function der =
        if(mon1) then
            if(mon2) then
                true
            else
                false
            endif
        endif

    main rule r_Main =
        if(der) then
            foo := true
        endif
```

Modello NuSMV

```
MODULE main
    VAR
        foo: boolean;
        mon1: boolean;
        mon2: boolean;
    DEFINE
        der:=
            case
                mon1 & !mon2: FALSE;
                mon1 & mon2: TRUE;
            esac;
    ASSIGN
        next(foo):=
            case
                der: TRUE;
                TRUE: foo;
            esac;
```

Figura 80: Funzioni statiche e derivate - Esaustività delle condizioni

In questo caso, NuSMV segnala che le condizioni della definizione di *der* non sono esaustive, come mostrato in Figura 81.

Esecuzione del modello NuSMV

```
[user@localhost asmetasmv]$ derivedNotExhaustive.smv
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson


file derivedNotExhaustive.smv: line 11: case conditions are not exhaustive

NuSMV terminated by a signal
```

Figura 81: Funzioni statiche e derivate - Esaustività delle condizioni (2)

12.2 Regole

Descriviamo come funziona la mappatura delle regole:

- il tool inizia la traduzione dalla main rule e continua eseguendo un visita in profondità delle regole che incontra;
- il tool aggiunge le condizioni booleane che incontra (e.g. if, switch,...) su uno stack globale *Conds*; rimuove una condizione dallo stack quando lascia lo scope della condizione;
- quando il tool incontra l'aggiornamento di una locazione, lo memorizza nel global update set con una condizione opportuna: la condizione che deve essere soddisfatta, affinché l'aggiornamento sia eseguito, è il prodotto logico delle condizioni presenti sullo stack globale.

Ogni transition rule contribuisce in modo differente alla costruzione dello stack Conds e del global update set. La Figura 82 mostra come viene aggiornato lo stack con la mappatura delle regole.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

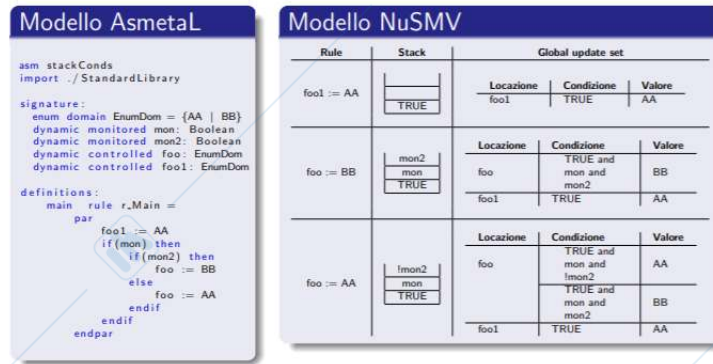


Figura 82: Mappatura delle regole

Le regole che AsmetaSMV può tradurre sono:

- update rule: La sintassi dell'update rule è: $l := t$, dove l è una locazione e t un termine. Tutti gli aggiornamenti di un modello AsmetaL sono memorizzati nel global update set che viene mappato nella sezione $ASSIGN$ del modello NuSMV. Vediamo la mappatura di una locazione numerica. L'esecuzione di NuSMV segnala che non è possibile aggiornare la variabile foo a 5. Infatti, poiché non c'è alcun controllo, la variabile foo viene incrementata fino a quando non raggiunge un valore che non appartiene al suo tipo. Per

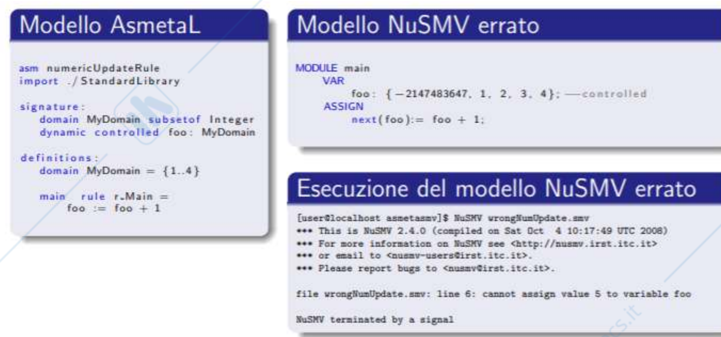


Figura 83: Update rule di una locazione numerica

risolvere il problema precedentemente descritto, per ogni aggiornamento di locazione numerica, aggiungiamo una condizione che afferma che il termine t deve essere contenuto nel codominio D_l della locazione l: t in D_l. La Figura 84 mostra la mappatura corretta dell'esempio precedente.

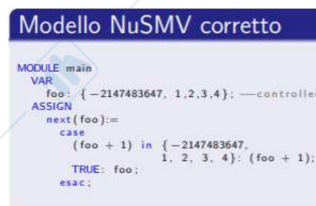


Figura 84: Update rule di una locazione numerica

Il tool, di default, aggiunge le condizioni. L'opzione d'esecuzione $-nc$ permette di non aggiungerle. Infatti, l'aggiunta delle condizioni potrebbe cambiare il comportamento del modello e violare l'equivalenza tra il modello AsmetaL ed il modello NuSMV.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- macrocall rule;
- block rule. La sintassi della block rule è:

  par

  R_1

  R_2

  ...

  R_n

  endpar

  dove $R_1, R_2, ..., R_n$ sono transition rules. In una block rule tutte le regole $R_1, R_2, ..., R_n$ sono eseguite in parallelo. AsmetaSMV traduce tutte le regole individualmente. Il contenuto dello stack Conds, all'inizio di ogni regola, è sempre lo stesso.

- conditional rule. La sintassi della conditional rule è:

  if cond then

  R_{then}

  else

  R_{else}

  endif

  dove cond è una condizione booleana e R_{then} e R_{else} sono transition rules. Se cond è true viene eseguita R_{then}, altrimenti viene eseguita R_{else}. La traduzione in NuSMV è la seguente:

  – cond viene aggiunta sullo stack Conds e viene visitata la regola R_{then}; in tal modo tutti gli aggiornamenti contenuti in R_{then} sono eseguiti solo se cond è true;
  – cond viene rimossa dallo stack Conds.
  – Se il ramo else non è nullo:
    * la condizione notCond (con notCond =!cond) viene aggiunta sullo stack Conds e viene visitata la regola R_{else}; in tal modo tutti gli aggiornamenti contenuti in R_{else} sono eseguiti solo se cond è false;
    * notCond viene rimossa dallo stack Conds.

Relativamente alla Figura 85, correttamente, la variabile foo viene aggiornata al valore AA solo se $guard = CC$ è true (ramo then); altrimenti la variabile viene aggiornata a BB (ramo else).

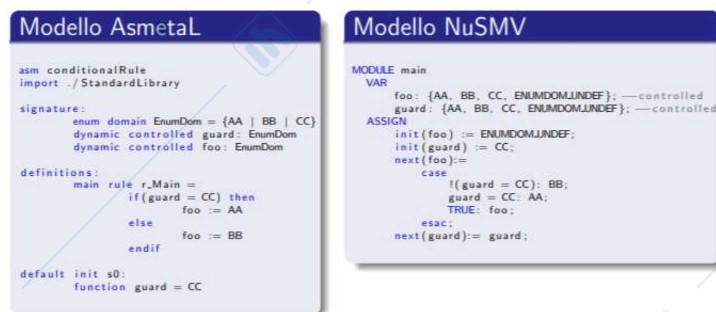


Figura 85: Conditional rule - Esempio

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- case rule. La sintassi della case rule è:

  switch t

  case $t_1 : R_1$

  ...

  case $t_n : R_n$

  otherwise R_{other}

  endswitch

  dove $t, t_1, ..., t_n$ sono termini e $R_1, ..., R_n, R_{other}$ sono transition rules. Per ogni ramo, la mappatura in NuSMV è:

  - la condizione $t = t_i$ viene aggiunta sullo stack Conds;
  - la regola R_i viene visitata;
  - la condizione $t = t_i$ viene rimossa dallo stack Conds.

  Se il ramo di default non è null:

  - la condizione $t! = t_1$ & ... & $t! = t_n$ viene aggiunta allo stack Conds e la regola R_{other} viene visitata;
  - la condizione precedente viene rimossa dallo stack Conds.

  Relativamente alla Figura 86, i due rami del case sono diventati due uguaglianze ($sw = AA$ *and* $sw = BB$). Il ramo otherwise è diventato il prodotto logico di due disuguaglianze ($sw! = AA$ & $sw! = BB$).

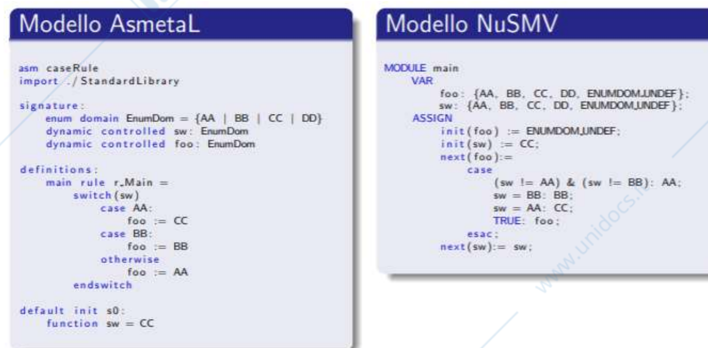


Figura 86: Case rule - Esempio

- let rule.

- forall rule. La sintassi della forall rule è: $forall$ v_1 in $D_1, ..., v_n$ in D_n with $G_{v_1}, ..., v_n$ do $R_{v_1}, ..., v_n$, dove $v_1, ..., v_n$ sono variabili logiche e $D_1, ..., D_n$ i loro domini. $G_{v_1}, ..., v_n$ è una condizione booleana su $v_1, ..., v_n$. $R_{v_1}, ..., v_n$ è una regola che contiene occorrenze di $v_1, ..., v_n$. L'obiettivo di una forall rule è quello di eseguire le regole $R_{v_1}, ..., v_n$ con tutti i valori delle variabili $v_1, ..., v_n$ che soddisfano la condizione $G_{v_1}, ..., v_n$. Il numero n_R di rami da valutare è uguale al prodotto delle cardinalità dei domini $D_1, ..., D_n$: $n_R = \Pi_{n=1}^{i} |Di|$. La traduzione in NuSMV, per ogni tupla di valori $d_1^{j_1}, ..., d_n^{j_n}$ con $d_1^{j_1} \in D_1, ... d_n^{j_n} in D_n$ esegue le seguenti operazioni:

  - le variabili $v_1, ..., v_n$ assumono i valori $d_1^{j_1}, ..., d_n^{j_n}$

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

- $G_{d_1^{j1},...,d_n^{jn}}$ e $R_{d_1^{j1},...,d_n^{jn}}$ sono la condizione e la regola dove le variabili sono sostituite con i loro valori correnti $d_1^{j1},...,d_n^{jn}$
- la condizione $G_{d_1^{j1},...,d_n^{jn}}$ viene aggiunta allo stack Conds;
- la regola $R_{d_1^{j1},...,d_n^{jn}}$ viene visitata;
- la condizione $G_{d_1^{j1},...,d_n^{jn}}$ viene rimossa dallo stack Conds.

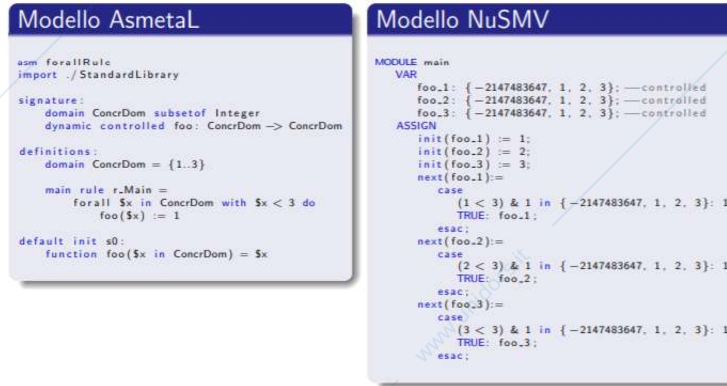


Figura 87: Forall rule - Esempio

- choose rule. La sintassi della choose rule è: choose v_1 in $D_1,...,v_n$ in D_n with $G_{v_1,...,v_n}$ do

  $R_{v_1,...,v_n}$

  ifnone R_{ifnone}

  dove $v_1,...,v_n$ sono variabili logiche e $D_1,...,D_n$ i loro domini. $G_{v_1,...,v_n}$ è una condizione booleana su $v_1,...,v_n$. $R_{v_1,...,v_n}$ è una regola che contiene occorrenze di $v_1,...,v_n$. L'obiettivo della choose rule è quello di eseguire una sola volta la regola $R_{v_1,...,v_n}$ con dei valori per $v_1,...,v_n$ che soddisfano $G_{v_1,...,v_n}$. Il numero n_R di rami da valutare è uguale al prodotto della cardinalità dei domini $D_1,...,D_n$: $n_R = \Pi_{n=1}^{n}|Di|$. Il ramo opzionale ifnone contiene la regola R_{ifnone} che deve essere eseguita se non ci sono valori per le variabili $v_1,...,v_n$ che soddisfano $G_{v_1,...,v_n}$. Nella mappatura, ogni choose rule viene identificata da un identificatore $chId$. Nel modello NuSMV, per ogni variabile logica v_i, viene creata una variabile $var\ v_i\ chId$. Il tipo di tale variabile è ottenuto dalla mappatura del dominio D_i. La mappatura in NuSMV, per ogni tupla di valori $d_1^{j1},...,d_n^{jn}$ con $d_1^{j1} \in D_1,...,d_n^{jn} \in D_n$, esegue le seguenti operazioni:

  - le variabili $v_1,...,v_n$ assumono i valori $d_1^{j1},...,d_n^{jn}$
  - $G_{d_1^{j1},...,d_n^{jn}}$ e $R_{d_1^{j1},...,d_n^{jn}}$ sono la condizione e la regola in cui le variabili sono state sostituite con i valori correnti $d_1^{j1},...,d_n^{jn}$
  - per ogni variabile v_i, viene aggiunta sullo stack Conds la condizione $var\ v_i\ chId = d_i^{ji}$
  - la condizione $G_{d_1^{j1},...,d_n^{jn}}$ viene aggiunta sullo stack Conds;
  - la regola $R_{d_1^{j1},...,d_n^{jn}}$ viene visitata;
  - le condizioni sulle variabili e $G_{d_1^{j1},...,d_n^{jn}}$ sono rimosse dallo stack Conds.
  - Se il ramo ifnone non è nullo: viene aggiunta sullo stack Conds la condizione

$$ifnonecond = \bigwedge_{d_1^{j1} \in D_1,...,d_n^{jn} \in D_n} !G_{d_1^{j1},...,d_n^{jn}}$$

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

dove il numero di termini del prodotto logico è n_R.

- la regola R_{ifnone} viene visitata;
- la condizione precedente viene rimossa dallo stack Conds.

Per essere certi che, in ogni stato, le variabili *var* v_i *chId* $(i = 1, ..., n)$ assumano dei valori che soddisfano $G_{v_1, ..., v_n}$, definiamo la seguente invariante nella sezione INVAR:

$$\bigwedge_{d_1^{j_1} \in D_1, ..., d_n^{j_n}} ((var_v_1_idCh = d_1^{j_1} \& ... \& var_v_n_idCh = d_n^{j_n}) \& G_{d_1^{j_1}, ..., d_n^{j_n}}) | ifNoneCond$$

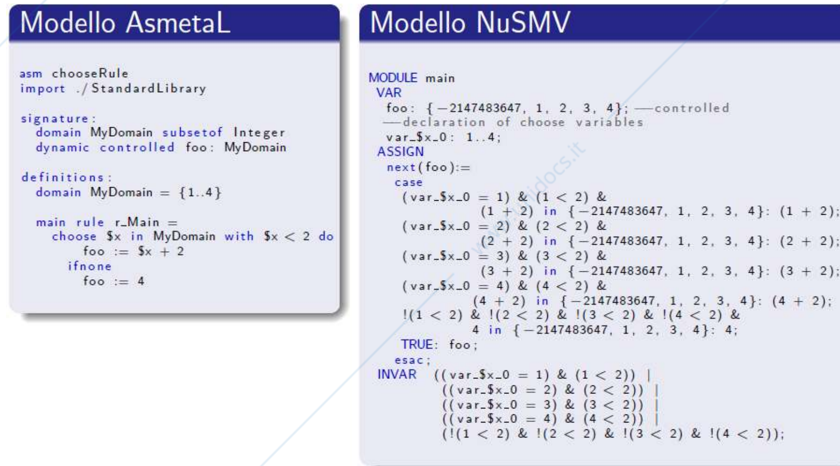


Figura 88: Choose rule - Esempio

12.3 Proprietà CTL

Le proprietà CTL devono essere dichiarate dopo la sezione degli invarianti. La sintassi di una proprietà CTL è CTLSPEC p, dove$p$ è un'espressione booleana. Le proprietà LTL è **LTLSPEC** q, dove q è un'espressione booleana. Le proprietà CTL in NuSMV sono rappresentate da **CTLSPEC** p_{CTL}, dove p_{CTL} è una formula CTL.

Per scrivere formule CTL in AsmetaL, abbiamo creato la libreria CTLlibrary.asm (esiste anche la libreria LTL, ma non la utilizzeremo) dove, per ogni operatore CTL e LTL, è stata dichiarata una funzione equivalente. Quasi tutte le funzioni sono unarie con dominio Boolean; solo le funzioni a ed e della CTLlibrary. Tutte le funzioni hanno codominio Boolean. Le funzioni della CTLlibrary possono essere utilizzate solo in una **CTLSPEC**. Per usare le funzioni CTL in un modello AsmetaL, dobbiamo importare la libreria CTLlibrary.asm. La Figura 88 mostra la mappatura degli operatori CTL in funzioni CTL nella libreria CTL.

Mappatura degli operatori CTL in funzioni CTL

Operatore CTL in NuSMV	Funzione CTL in AsmetaL
EG p	static eg: Boolean → Boolean
EX p	static ex: Boolean → Boolean
EF p	static ef: Boolean → Boolean
AG p	static ag: Boolean → Boolean
AX p	static ax: Boolean → Boolean
AF p	static af: Boolean → Boolean
E[p U q]	static e: Prod(Boolean, Boolean) → Boolean
A[p U q]	static a: Prod(Boolean, Boolean) → Boolean

Figura 89: CTL Library

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

La Figura 90 mostra un esempio di introduzione delle proprietà CTL.

Modello AsmetaL

```
asm ctlExample
import ./StandardLibrary
import ./CTLlibrary

signature:
   dynamic controlled fooA: Boolean
   dynamic controlled fooB: Boolean
   dynamic monitored mon: Boolean

definitions:
   CTLSPEC ag(fooA iff ax(not(fooA))) //true
   CTLSPEC ag(not(fooA) iff ax(fooA)) //true
   //false. Gives counterexample.
   CTLSPEC not(ef(fooA != fooB))

   main rule r_Main =
      par
         fooA := not(fooA)
         if(mon) then
            fooB := not(fooB)
         endif
      endpar

default init s0:
   function fooA = true
   function fooB = true
```

Modello NuSMV

```
MODULE main
   VAR
      fooA: boolean; --controlled
      fooB: boolean; --controlled
      mon: boolean; --monitored
   ASSIGN
      init(fooA) := TRUE;
      init(fooB) := TRUE;
      next(fooA) := !(fooA);
      next(fooB) :=
         case
            (mon): !(fooB);
            TRUE: fooB;
         esac;

--CTL properties
CTLSPEC   AG(fooA <-> AX(!(fooA)));
CTLSPEC   AG(!(fooA) <-> AX(fooA));
CTLSPEC   !(EF(fooA != fooB));
```

Figura 90: Proprietà CTL - Esempio

La Figura 91 mostra l'esecuzione del modello relativo alla Figura 90, in cui le prime due proprietà sono vere: la variabile $fooA$ inverte il suo valore ad ogni passo. L'ultima proprietà, invece, è falsa e NuSMV mostra un controesempio: esiste uno stato in cui le variabili $fooA$ e $fooB$ sono diverse.

Esecuzione del modello NuSMV

```
[user@localhost asmetasmv]$ NuSMV ctlExample.smv
*** This is NuSMV 2.5.2 (compiled on Sat Oct 30 12:18:33 UTC 2010)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>

-- specification AG (fooA <-> AX !fooA)  is true
-- specification AG (!fooA <-> AX fooA)  is true
-- specification !(EF fooA != fooB)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  fooA = TRUE
  fooB = TRUE
  mon = FALSE
-> State: 1.2 <-
  fooA = FALSE
```

Figura 91: Proprietà CTL - Esempio

In AsmetaL, un invariante è una proprietà che deve essere verificata in ogni stato della macchina; la sintassi è: **invariant over** $id_1, ..., id_n$: $ax_{id_1, ..., id_n}$, dove $id_1, ..., id_n$ sono i nomi di domini, funzioni o regole; $ax_{id_1}, ..._{,id_n}$ è un'espressione booleana che contiene occorrenze di $id_1, ..., id_n$. Per essere sicuri che un invariante sia sempre vero dovremmo simulare il modello lungo tutti gli stati possibili. Con la traduzione in NuSMV possiamo effettuare la verifica degli invarianti in modo esaustivo. Dato l'invariante visto sopra, se vogliamo verificare che $ax_{id_1}, ..._{,id_n}$ sia verificato in tutti gli stati del modello, potremm scrivere una proprietà CTL del genere **CTLSPEC** $ag(ax_{id_1}, ..._{,id_n})$. La funzione ag significa che la proprietà $ax_{id_1}, ..._{,id_n}$ deve essere verificata in tutti gli stati.

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

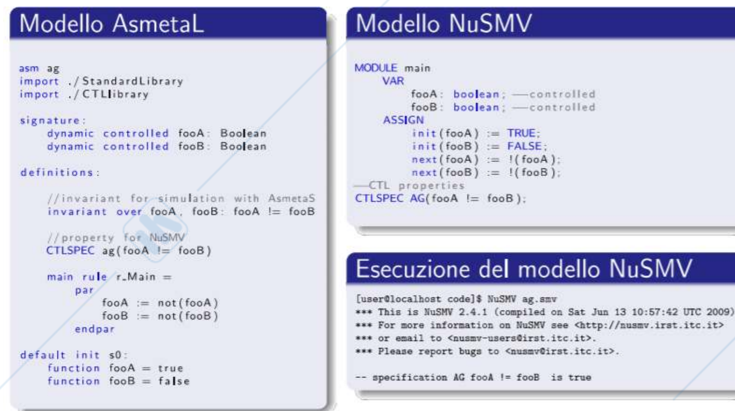


Figura 92: Invarianti

Scaricato da Stefano Giardina (giardina.stefano98@gmail.com)
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari