Progetto di ASM

Michele Piffari

Matricola 1040658

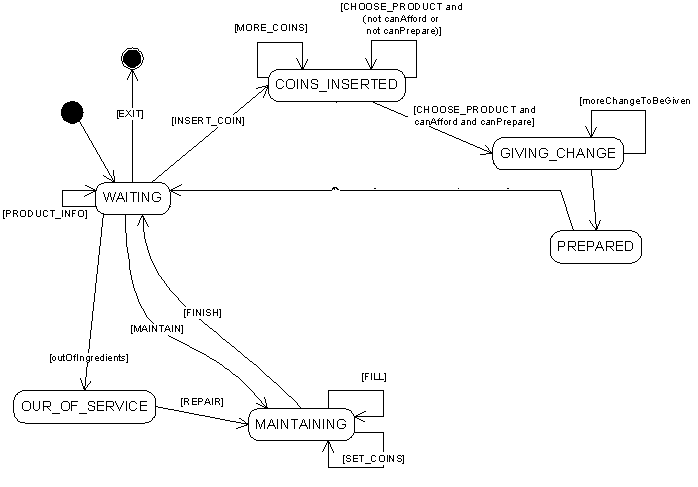
# TODO: dire che non riesco a diminuire il numero di monete quando do il resto. Non riesco a rifornire solo il numero di monete dalla quantità attualmente presente fino al massimo.

# Coffee Machine

L’*Abstract State Machine* progettata modella un distributore automatico di bevande.   
Il distributore può preparare diversi tipi di bevande (caffè, cappuccino, …), ognuna preparata con diverse quantità di ingredienti base (acqua, caffè, latte, …) i quali vengono consumati e devono essere reintegrati dal manutentore.   
Il distributore accetta pagamenti solamente in moneta. Il distributore è in grado di fornire il resto (anche se non sempre in modo esatto) .  
Quando tutte le bevande sono esaurite, il distributore va fuori servizio, in attesa che gli ingredienti vengano aggiunti dal manutentore; egli può inoltre prelevare o inserire monete dal distributore.

## FSM e ASM

La ASM è basata su una sottostante macchina a stati finiti, mostrata in figura, che definisce i principali stati e transizioni del distributore. La ASM permette di estendere questa FSM introducendo un concetto aumentato di “stato”, che comprende anche funzioni dinamiche, modificando le quali si possono memorizzare informazioni aggiuntive.



In particolare, è stato possibile memorizzare informazioni su:

* Quantità di ingredienti residue
* Monete possedute dal distributore
* Credito dell’utente attuale

## Eventi

La ASM sviluppata modella un sistema *event-driven*, ovvero un sistema in cui le transizioni da uno stato all’altro sono perlopiù scatenate da input dell’utente, mentre di solito la macchina si trova ferma in uno stato, in attesa di tali eventi.

Nel codice della ASM questi eventi sono denominati *action*; ad ogni stato corrispondono una o più azioni che l’utente può compiere quando la macchina è in quello stato. Ad esempio, quando il distributore è nello stato KEY\_INSERTED, le azioni disponibili sono di rimuovere la chiave, ricaricarla o selezionare un prodotto. Le azioni sono codificate come elementi di un dominio enumerativo, i quali essendo identificativi univoci, non possono essere duplicati :

enum domain CoinsInsAction = {ADD\_KEY | MORE\_COINS | CHOOSE\_PRODUCT}

enum domain Ingredient = {COFFEE | MILK | WATER | TEA | CHOCOLATE | PLASTIC\_GLASS}

...

La caratteristica event-driven del sistema si è riflessa nella ASM, infatti le regole (*rules*) implementate possono essere suddivise in due categorie: regole che attendono il verificarsi di un’azione (qui chiamate *event management rules*) e poi eseguono le corrette regole di transizione (*transition rules*); regole di transizione che verificano la guardia della transizione e, se verificata, eseguono gli update opportuni.   
In linea di massima, ad ogni stato corrisponde una event management rule, mentre ad ogni arco (transizione) corrisponde una transition rule.

Qui di seguito verrà esposta nel dettaglio la ASM sviluppata, a partire dai domini e dalle funzioni, per comprendere la macchina partendo dai blocchi più elementari. Le regole, a cui si è già accennato qui, verranno esposte successivamente.

## Domains

Sono stati introdotti un dominio enumerativo per gli stati della FSM e un dominio enumerativo per ciascun insieme di eventi (ciascun insieme contiene gli eventi validi per uno stato). Sono stati introdotti anche un dominio statico concreto per i tagli di monete riconosciuti dal distributore (rappresentati in €cents), e un dominio astratto per i prodotti disponibili. I domini astratti possono essere estesi includendovi nuovi elementi, ma per l’ASM sviluppata tutti gli elementi utilizzati sono stati già specificati nella *signature*.   
Un ulteriore dominio enumerativo rappresenta gli ingredienti immagazzinati nel distributore.

abstract domain Product

domain CoinValue subsetof Integer

enum domain State = {WAITING | COINS\_INSERTED | GIVING\_CHANGE

| PREPARED | MAINTAINING | OUT\_OF\_SERVICE}

... *action domains* ...

enum domain Ingredient = {COFFEE | MILK | WATER | TEA | CHOCOLATE |

PLASTIC\_GLASS}

static coffee : Product

static tea : Product

static cappuccino : Product

static chocolate : Product

static glassOnly : Product

...

domain CoinValue = {5n, 10n, 20n, 50n, 100n, 200n}

Si vede quindi come il dominio CoinValue sia un subset del *basic domain*  Integer definito nella StandardLibrary: è stato comunque scelto un subset di Integer, nonstante il fatto che il taglio delle monete sia un valore NON negativo.

Invece, il dominio abstract Product rappresenta la corrispondente implementazione di una classe Java.

## Controlled, derived, static functions

Queste sono le funzioni *non-monitored* che sono state definite:

dynamic controlled state : State

dynamic controlled display : String

dynamic controlled credit : Integer

dynamic controlled coinsLeft : CoinValue -> Integer

dynamic controlled quantityLeft : Ingredient -> Real

Le funzioni *controlled* rappresentano funzioni in cui l’*interpretazione*, ovvero la mappatura dominio → codominio, può essere modificata dalla macchine tramite *updates*.   
Questo corrisponde al concetto di modifica di una variabile o di una struttura dati di tipo map all’interno di un linguaggio di programmazione.

Si vede come le prime tre funzioni rappresentino delle funzioni 0-arie, ovvero delle variabili: le ultime due funzioni invece sono n-arie, ovvero mappano dei valori da un dominio ad un codominio: nello specifico associano ad ogni CoinValue un valore Integer e ad ogni Ingrediente un avalore Real, che non è altro che la quantità rimasta.

I valori assunti dalle funzioni controlled rappresentano parte dello stato esteso delle ASM, quindi possono essere utilizzate per mantenere informazioni tra uno stato e l’altro della FSM sottostante.   
Le funzioni controlled definite sopra mantengono informazioni sul messaggio mostrato all’utente chiave e sulla quantità di monete e di ingredienti residui nel distributore.

static capacity : Ingredient -> Real

static quantityNeeded : Prod(Product, Ingredient) -> Real

static price : Product -> Integer

static currencyToString : Integer -> String

Le funzioni *static* sono funzioni la cui interpretazione viene fissata dalla definizione della ASM e non può essere modificata durante l’esecuzione. Possono essere paragonate alle costanti dei linguaggi di programmazione. Ad esempio, la funzione price è stata definita così:

function price($p in Product) =

switch($p)

case coffee : 30

case tea : 40

case cappuccino : 40

case chocolate : 40

case glassOnly : 10

endswitch

Essa rappresenta un legame costante tra un prodotto e il suo prezzo. La funzione currencyToString è invece una funzione di supporto per formattare numeri; anch’essa è statica perché il suo comportamento non può variare:

function currencyToString($n in Integer) =

concat(toString( $n / 100)," €")

Le funzioni *derived*, infine, sono funzioni definite a partire da funzioni dinamiche:

function canPrepare($p in Product) =

not (exist $i in Ingredient

with quantityLeft($i) < quantityNeeded($p,$i))

function outOfIngredients =

not (exist $p in Product with canPrepare($p) and $p != glassOnly)

## Monitored functions

Le funzioni *monitored* rappresentano degli input che l’utente fornisce alla macchina. Il valore di queste funzioni non è persistente, ma viene ad essere specificato dall’utente per ogni stato tramite tastiera (oppure lette anche da file esterno).   
Il simulatore ha però una caratteristica interessante, che è la *lazy evaluation*: una funzione monitored viene richiesta all’utente solo nel momento in cui la ASM deve “leggerne il valore”; in questo modo l’utente deve inserire il numero minimo indispensabile di input.

dynamic monitored selectedProduct : Product

dynamic monitored insertedCoin : CoinValue

dynamic monitored numberOfCoins : Integer

dynamic monitored filledIngredient : Ingredient

dynamic monitored waitingAction : WaitingAction

dynamic monitored coinsInsAction : CoinsInsAction

dynamic monitored preparedAction : PreparedAction

dynamic monitored outOfServiceAction : OutOfServiceAction

dynamic monitored maintainAction : MaintainAction

Tra le funzioni monitored figurano le funzioni che richiedono all’utente la scelta tra le azioni disponibili. Inoltre ci sono funzioni con cui l’utente specifica quale moneta, quale prodotto è stato selezionato e, per il manutentore, quale ingrediente è stato rifornito e quante monete ha lasciato nel distributore (quest’ultima è una scorciatoia molto pratica per evitare di inserire/estrarre una moneta alla volta).

## Event management rules e main rule

Come descritto inizialmente, le event management rules si occupano di ricevere le azioni dell’utente e di eseguire (*fire*) la regola di transizione opportuna. Qui c’è un elenco delle regole, la cui struttura è del tutto simile a quella dell’unica regola definita:

rule r\_waitingAction =

if(state = WAITING) then

switch(waitingAction)

case PRODUCT\_INFO: r\_productInfo[]

case INSERT\_COINS: r\_insertCoins[]

case MAINTAIN: r\_maintain[]

case EXIT: skip

endswitch

endif

rule r\_coinsInsAction = ...

rule r\_preparedAction = ...

rule r\_outOfServiceAction = ...

rule r\_maintainAction = ...

La *main rule*, che costituisce l’*entry point* del programma, controlla per prima cosa che il distributore possa operare, perché non ha finito gli ingredienti (r\_selfCheck); questo è possibile grazie al comportamento del blocco sequenziale, che effettua gli update dopo la valutazione di ciascun termine. Dopo il controllo, invece, vengono valutate in parallelo le regole di gestione degli eventi; per esse non vi è pericolo di update inconsistenti perché ciascuna regola ha una guardia che permette di valutare la regola solo quando la macchina si trova nello stato corretto (ogni regola si applica a un diverso stato, quindi sola una alla volta è “attiva”).

main rule r\_main =

seq

r\_selfCheck[]

par

r\_waitingAction[]

r\_keyInsAction[]

r\_coinsInsAction[]

r\_giveChange[]

r\_preparedAction[]

r\_maintainAction[]

r\_outOfServiceAction[]

endpar

endseq

La regola r\_giveChange è diversa dalle altre in quanto non deve reagire a un’azione dell’utente, ma deve essere iterata più volte (all’interno dello stesso stato GIVING\_CHANGE della FSM) finché il resto non è stato erogato completamente.

## Transition rules

Ad ogni transizione della FSM, anche se rientrante sullo stesso stato, è associata una regola di transizione, che si occupa di valutare la guardia della transizione e, se questa risulta vera, di eseguire gli update opportuni.

Qui sono riportati alcuni esempi di transition rules:

// check if some drink can be prepared with the ingredients left

rule r\_selfCheck =

if(state = WAITING and outOfIngredients) then

state := OUT\_OF\_SERVICE

endif

// accept an inserted coin and change state to COINS\_INSERTED

rule r\_insertCoins =

let($c = insertedCoin) in

seq

r\_accredit[$c]

coinsLeft($c) := coinsLeft($c) + 1n

r\_changeState[COINS\_INSERTED]

endseq

endlet

// accept more coins (no key inserted)

rule r\_moreCoins =

r\_insertCoins[]

// give the change back in a greedy way: give back coins starting from the // highest-value available coin; if some credit is left but there are no

// more coins (with value not greater than the residual credit), then the

// credit is lost

rule r\_giveChange =

if(state = GIVING\_CHANGE) then

/\* find the highest available coin value,

\* not greater than the residual credit \*/

choose $cv in CoinValue with coinsLeft($cv) > 0n and

$cv <= credit and

not(exist $cv2 in CoinValue with coinsLeft($cv2) > 0n and

$cv2 <= credit and $cv2 > $cv) do

par

credit := credit - $cv

coinsLeft($cv) := coinsLeft($cv) - 1n

display := "Pick up your change"

endpar

ifnone

seq

credit := 0n //Greedy: keep the residual credit

r\_changeState[PREPARED]

endseq

endif

// change state to MAINTAINING

rule r\_repair =

r\_changeState[MAINTAINING]

// restore the maximum amount of an ingredient

rule r\_fill =

quantityLeft(filledIngredient) := capacity(filledIngredient)

// show information about price/availability of a product

rule r\_productInfo =

let($p1 = selectedProduct) in

if(canPrepare($p1)) then

display := currencyToString(price($p1))

else

display := "not available"

endif

endlet

## Utility rules

Con utility rule si intendono qui quelle regole che servono come supporto ad altre regole, ma non rappresentano né regole di transizione né regole di gestione degli eventi. Alcuni esempi:

rule r\_accredit($q in CoinValue) =

credit := credit + $q

rule r\_consumeIngredients($p in Product) =

forall $i in Ingredient do

quantityLeft($i) := quantityLeft($i) - quantityNeeded($p,$i)

## Assiomi

Gli assiomi rappresentano degli invarianti, ovvero enunciati che devono sempre essere verificati dalla macchina, in qualsiasi stato. Sono utilizzati per garantire che la ASM rispetti certe proprietà, e sono quindi utili per evidenziare difetti di progettazione o di codifica della macchina.

Gli assiomi sono stati definiti utilizzando funzioni ausiliarie:

axiom inv\_integerGlasses over integerGlasses : integerGlasses

axiom inv\_nonNegIngredients over nonNegIngredients : nonNegIngredients

function integerGlasses =

quantityLeft(PLASTIC\_GLASS) = ntor(rton(quantityLeft(PLASTIC\_GLASS)))

function nonNegIngredients =

not (exist $i in Ingredient with quantityLeft($i) < 0.0)

## Inizializzazione

Di default, tutte le funzioni prendono valore undef per quei valori del dominio per cui non sono state esplicitamente definite.   
Perché la macchina inizi a operare in uno stato diverso, più significativo (anche perché raramente la macchina viene definita in modo da poter gestire valori undef), si deve inizializzare la macchina, ovvero definire uno stato iniziale:

default init initial\_state:

function state = WAITING

function display = "Waiting..."

function coinsLeft ($cv in CoinValue) =

switch($cv)

case 5n : 5n

case 10n : 20n

case 20n : 20n

case 50n : 20n

case 100n : 10n

case 200n : 5n

endswitch

function credit = 0n

function quantityLeft($i in Ingredient) =

switch($i)

case PLASTIC\_GLASS : 10.0

case WATER : 8.0

otherwise 2.0

endswitch

In questo caso, lo stato FSM iniziale è quello di attesa, il credito in monete è nullo e la macchina possiede un discreto quantitativo di monete e ingredienti, per poter operare per un certo periodo senza bisogno di manutenzione.

## Simulazione

La macchina è stata simulata con AsmetaS, sia in modalità interattiva che in modalità batch.  
In modalità interattiva è facile scoprire errori sia di transizione di stato FSM sia di update, perché ad ogni update viene mostrato lo stato completo. Tuttavia una simulazione esaustiva è piuttosto lunga, per cui è difficile trovare errori nelle parti di macchina eseguite più raramente.

La modalità random permette di trovare facilmente violazioni degli assiomi e update inconsistenti: eseguendo una simulazione random con un numero elevato di transizioni (es. 1000) è più probabile coprire anche situazioni poco frequenti o poco naturali per un utente umano. Ad esempio, con una simulazione random risulta evidente che il numero di monete nel distributore non ha un limite superiore, per scelta progettuale; ciò sarebbe difficile da verificare con una simulazione manuale, dove le monete vengono inserite una per volta, tranne da parte del manutentore. Con la simulazione random è però impossibile trovare errori di update o di transizione, poiché nessuno verifica che ogni nuovo stato raggiunto sia coerente con le aspettative.

Con la modalità batch è possibile fornire al simulatore degli input predeterminati, tramite un file, cosa molto più veloce rispetto al doverli inserire manualmente. Con questa modalità è facile trovare errori di transizione, anche dopo che la macchina ha eseguito una lunga sequenza di update; infatti se una transizione è sbagliata l’input successivo sarà inatteso per il simulatore e ciò porta ad una catena di errori facilmente individuabile. Inoltre, conoscendo gli input forniti tramite file e avendo la descrizione di ogni stato in output al simulatore, si può tracciare l’origine degli errori più facilmente rispetto alla modalità random.

Per la simulazione batch è stato creato un file di environment testCoffee.env che esegue un tour completo degli stati e delle transizioni (della FSM). Per facilitare questo tipo di simulazione è stato introdotta anche l’azione EXIT nel dominio WaitingAction, che è l’unica azione che produce un update set vuoto; è quindi possibile eseguire la simulazione batch con opzione –ne:

java -jar AsmetaS.jar -ne -env testCoffee.env CoffeeVendingMachine.asm