Framework per la simulazione e la valutazione di modelli CTMC utilizzando Maude e Approximate Probabilistic Model Checking

Bellavista Daniele

8 marzo 2012

1 Introduzione

Nella prima parte di questo progetto ho implementato un simulatore per modelli CTMC (*Continuos time Markov Chain*) generici, cioè senza assunzione sulla struttura degli stati del sistema. Questo obiettivo è stato raggiunto in primis utilizzando Maude e il matching simbolico nativo, ma anche grazie all'utilizzo del meta-livello di Maude [3], che permette di trasformare moduli Maude in dati e di eseguire operazioni dinamiche su di essi.

Nella seconda parte, ho implementato un verificatore di proprietà PCTL monotone utilizzando la tecnica di *Approximate Probabilistic Model Checking* [4][2].

2 Il meta-livello di Maude

La logica di *rewriting* di *Maude* è riflessiva [3], ovvero descritta in termini di se stessa tramite una teoria universale. Rendendo disponibile all'utente l'accesso alla teoria universale di Maude, è possibile ragionare ponendosi al meta-livello sia della *"struttura"* dei moduli e termini *Maude*, sia delle strategie risolutive dei procedimenti di inferenza, riduzioni, matching e ricerche.

La *riflessione* della logica di rewriting può essere descritta attraverso la relazione :

$$\mathcal{R} \vdash t \rightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \bar{\mathcal{R}}, \bar{t} \rangle \rightarrow \langle \bar{\mathcal{R}}', \bar{t}' \rangle$$

Dove $\mathcal{R}$ è una teoria di *rewrite*, $\mathcal{U}$ è la teoria di rewrite universale, in grado di rappresentare qualunque teoria di rewrite finita in termini di se stessa. Il simbolo $\rightarrow$ indica che il termine a sinistra può essere riscritto come il termine di destra; t, t' sono termini, $\langle \bar{\mathcal{R}}, \bar{t} \rangle$ è un termine che rappresenta secondo la teoria $\mathcal{U}$ la coppia $(\mathcal{R}, t)$.

La teoria $\mathcal{U}$ è universale e può essere espressa anche in termini di se stessa. Dunque la relazione precedente crea una "reflective tower" nella quale è possibile

muoversi sia alto che in basso. Il livello più basso nella torre è chiamato forma canonica.

La teoria universale di rewrite $\mathcal{U}$ è implementata in Maude nel modulo `META-LEVEL`, il quale, oltre a contenere operazioni per muoversi lungo la reflective tower, contiene sort per descrivere al meta-livello moduli e termini.

La maggior parte delle operazioni del meta-livello di Maude sono parziali, poiché possono esistere argomenti del Sort corretto che tuttavia non sono una corretta meta-rappresentazione o non esiste una valido risultato.

2.1 Accesso dinamico a moduli e termini

Termini e moduli Maude possono essere rappresentati come altri termini Maude, sruttando la teoria di rewriting universale, permettendo di caricare moduli dinamicamente, lavorare e costruire termini di Sort anche sconosciuti ed eseguire su tali meta-moduli e meta-termini delle operazioni di reduce e rewrite.

2.1.1 upTerm e downTerm

Dato un qualunque termine Maude, è possibile salire e scendere la *reflectio tower* utilizzando le operazioni *upTerm* e *downTerm*:

```
op upTerm : Universal -> Term [poly (1) special (...)] .
op downTerm : Term Universal -> Universal [poly (2 0) special (...)] .
```

upTerm prende in ingresso un qualunque termine t e restituisce sempre la sua meta-rappresentazione $\bar{t}$ di sort *Term*. *downTerm* prende in ingresso $\bar{t}$, meta-rappresentazione di un termine t, ed un secondo argomento di un certo kind. Se t è dello stesso kind del secondo argomento, la *downTerm* restituisce t, altrimenti restituisce il secondo argomento.

```
Maude> red in META-LEVEL : upTerm ( 'a ) .
result Constant: ''a.Sort

Maude> red in META-LEVEL : upTerm( 'a 'b ) .
result GroundTerm: '__['' a.Sort,''b.Sort]

Maude> red in META-LEVEL : downTerm ( ''a.Sort , ('prova).Qid ) .
result Sort: 'a

Maude> red in META-LEVEL : downTerm ( ''a.Sort , ("Fail").String ) .
result String: "Fail"
```

2.1.2 upModule

I comandi *mod* e *fmod* permettono di definire moduli che Maude caricherà nel proprio database interno. Quando si eseguono istruzioni come

```
Maude> reduce in MIO-MODULO : prova( 1 , 2 , 3 ) .
```

Maude accede al database, cerca il modulo *MIO-MODULO* ed effettua la *reduce* dell'espressione *prova(1,2,3)*: l'accesso al database è statico.

Il meta-livello di Maude permette di ottenere la meta-rappresentazione di un modulo precaricato nel database, partendo dal suo nome espresso con un quoted identifier, tramite l'operazione:

```
op upModule : Qid Bool ~> Module [special (...)] .
```

Il primo argomento è il nome del modulo, il secondo argomento è true se si desidera ottenere anche la meta-rappresentazione di tutte le operazioni derivate dai moduli importati, false altrimenti. Il valore di ritorno Module è un particolare sort che meta-rappresenta un modulo Maude; ha una struttura definita, con un ordine preciso, dunque è semplice da analizzare ed alterare, ma non è possibile caricare un meta-modulo nel database di Maude.

2.1.3 metaRewrite e metaApply

Dato un oggetto di tipo Module è possibile eseguire dinamicamente operazioni di reduce, rewrite e apply (applicazione di una rule data una regola) partendo dalla meta-rappresentazione del termine da ridurre o riscrivere. L'operazione *metaRewrite* permette di eseguire su un meta-modulo *Module* la rewrite di un meta-termine *Term* fino ad un numero di volte specificato da *Bound*.

```
sort Bound .
subsort Nat < Bound .
op unbounded :-> Bound [ctor] .
op metaRewrite : Module Term Bound ~> ResultPair [special (...)] .
```

La metaRewrite ritorna valore di sort ResultPair:

```
sort ResultPair .
op {_,_} : Term Type -> ResultPair [ctor] .

op getTerm : ResultPair -> Term .
op getType : ResultPair -> Type .
```

contenente la meta-rappresentazione del risultato e la meta-rappresentazione del suo kind o sort. La funzione *metaRewrite* è parziale, quindi se Module o Term sono sintatticamente corretti, ma non sono corrette meta-rappresentazioni, è restituito un risultato indefinito di kind [ResultPair] .

L'operazione *metaApply* (e la sua generalizzazione *metaXapply*) permettono di applicare una rule basandosi sulla label ed ottenere un risultato dettagliato sulla riscrittura. La chiamata di $metaXapply(\bar{\mathcal{R}}, \bar{t}, \bar{l}, \sigma, n, b, m)$ riduce t con le equazioni di $\mathcal{R}$, dopodiché crea una sequenza di soluzioni applicando tutte le rule chiamate l al termine t e scarta i primi n risultati. b e m specificano il livello di nesting al quale applicare la riscrittura (per una riscrittura non limitata: 0 - *unbounded*).

```
sorts Result4Tuple Result4Tuple? .
subsort Result4Tuple < Result4Tuple? .
```

```
op {_,_,_,_} : Term Type Substitution Context -> Result4Tuple [ctor] .
op failure : -> Result4Tuple? [ctor] .

op metaXapply : Module Term Qid Substitution Nat Bound Nat ~>
                                                        Result4Tuple? .
```

Se l'operazione è andata a buon fine, il risultato *Result4Tuple* contiene:

Term: meta-termine riscritto.

Type: meta-tipo del termine riscritto.

Substitution: dettagli sulla sostituzione effettuata dalla apply.

Context: contesto nel quale è avvenuta la riscrittura. Contiene una lista di meta-termini non riscritti e un singolo "hole" dove è avvenuta la riscrittura.

2.1.4 metaMatch

Un'altra potente applicazione del meta-livello di Maude è la metaMatch (e la sua generalizzazione metaXmatch), che permette di eseguire un match fra due meta-termini all'interno di un meta-modulo permettendo di specificare condizioni booleane e di enumerare le soluzioni.

```
sorts MatchPair MatchPair? .
subsort MatchPair < MatchPair? .
op {_,_} : Substitution Context -> MatchPair [ctor] .
op noMatch : -> MatchPair? [ctor] .
op metaXmatch : Module Term Term Condition Nat Bound Nat ~>
                                              MatchPair? [special (...)] .
```

Il risultato *MatchPair* contiene le sostituzioni effettuate e il contesto nel quale è avvenuta la sostituzione.

3 Progetto CTMC

La prima parte del progetto che ho realizzato è un simulatore per modelli CTMC utilizzando il linguaggio Maude e le sue funzionalità di meta-livello.

3.1 Requisiti

Si vuole realizzare un simulatore per catene di Markov tempo continue in grado di:

- Gestire stati arbitrari.
- Gestire azioni di transizione da uno stato all'altro caratterizzate da un certo rate, permettendo di specificare condizioni e rate dinamici.
- Gestire delle overview, ovvero risultati parziali dell'esecuzione.

3.2 Realizzazione

Una transizione è descrivibile come $State \Longrightarrow [Rate]State\ if\ Condition$, ovvero come stati che possono essere trasformati in altri stati secondo un certo rate e se è avverata una data condizione. Questa definizione è naturalmente riconducibile alle *rewriting rule* di Maude; dunque, definita una sintassi per i membri delle rule, si può sfruttare il meccanismo di rewrite nativo.

L'ultimo punto dei requisiti riguarda la gestione delle overview. Definisco una Overview come una informazione che l'utente riceve ogni M passi di simulazione, contenente informazioni relative alla dinamica del sistema.

3.3 Organizzazione

Il sistema è suddiviso in due moduli. *CTMC* contiene per la sintassi di CTMC, che i moduli utente devono importare, mentre *CTMC-ENGINE* contiene le operazioni necessarie per la simulazione.

3.3.1 Modulo CTMC

All'interno del modulo *CTMC* ho definito i Sort e i costruttori del modello CTMC. Una catena di Markov è una coppia $\langle A, \rightarrow A\rangle$, dove A è un insieme di stati e $\rightarrow \subseteq A \times \mathbb{R}_0^+ \times A$ è una transizione fra due stati che avviene con un certo rate. Come detto precedentemente, è possibile sfruttare il meta-livello di Maude per gestire le transizioni come *rule* e definire unicamente la sintassi.

Per avere una definizione più generica possibile di stato del sistema, definisco uno *State* come un elemento polimorfico:

```
sort State  .
op <_> : Universal -> State [ ctor poly (1) ] .
```

Per modellare i rate, ho definito un sort che comprenda rate e stato finale e che possa essere riscritto dallo stato inziale: *Config* subsort di *State*. In questo modo *State* può essere riscritto come un *Config*.

```
sort Config Rate Var .
subsort Float < Rate .
subsort Config < State .
op [_]_ : Rate State -> Config [ ctor prec 80 ] .
```

Per semplificare ed ottimizzare le operazioni di simulazione, impongo che il modulo utente debba definire le regole di rewrite del modello con la label *model*. Ad esempio:

```
*** Esempio di modello CTMC in cui lo stato è un multi-set
*** di numeri naturali. Le regole che formano il modello devono
*** avere la label model.

sort NatMSet .
subsort Nat < NatMSet .
op nil : -> NatMSet .
```

```
op __ : NatMSet NatMSet -> NatMSet [ ctor comm assoc id: nil ] .

crl [model] : < 12 N:Nat 123 > =>
                  [0.2 * float(N:Nat)] < 1312 > if N:Nat > 14 .
crl [model] : < N:Nat > =>
                  [0.8 * float(N:Nat)] < 12 15 123 > if N:Nat > 1300 .
rl [model] : < N:Nat NL:NatList > =>
                  [12.0] < (N:Nat + 1) NL:NatList > .
```

Nel modulo *CTMC* ho anche definito la sintassi per l'overview, implementato come contenitore di una lista di *Entry*, coppie $(Tempo, Stato)$:

```
sort Overview Entry EntryList .
subsort Entry < EntryList .

op _:_ : Float State -> Entry [ ctor prec 80 ] .
op empty : -> EntryList [ ctor ] .
op _ - _ : EntryList EntryList -> EntryList [ ctor assoc id: empty prec 90 ] .
op [_] : EntryList -> Overview [ ctor ] .
```

Per i motivi che spiegherò in seguito, il modulo utente per la gestione dell'overview, deve implementare un sistema di rule per riscrivere l'Overview, tenendo conto che sarà effettuata un'unica rewrite . Per iterare l'overview si possono comunque usare sistemi di equazioni, poiché il risultato è sempre ridotto.

3.3.2 Modulo CTMC-ENGINE

Il modulo engine è il simulatore dei modelli CTMC. Ho realizzato questo engine in modo che potesse simulare dinamicamente un modello, ricevendo il nome del modulo che lo implementa e la meta-rappresentazione dello stato iniziale. In questo modo non ha bisogno di conoscere operazioni ed sort del modulo utente.

Utilizzando l'operazione *upModule*, descritta nella sezione 2.1.2, il simulatore è in grado di ottenere il meta-modulo e cominciare la simulazione eseguendo ricorsivamente una sequenza di passi:

1. Trovare tutte le azioni concorrenti che possono attivarsi dallo stato corrente.
2. Scegliere una azione a_i con probabilità $\frac{r_i}{R}$, dove $R = \sum_i r_i$
3. Applicare l'azione, determinando il nuovo stato del sistema.
4. Incrementare il tempo di $\Delta t = \frac{1}{R} \log \frac{1}{\tau}$, dove $\tau = random(0, 1)$.
5. Appendere il nuovo risultato all'overview e controllare se occorre comunicarlo all'utente.

Tutti i passi sono realizzati utilizzando il meta-livello di Maude e i dati di stato ed overview sono trattati in forma non-canonica, quindi quando scrivo *stato* ed *overview* in realtà intendo le loro meta-rappresentazioni.

Il passo 1 è facilmente risolto imponendo al modulo utente di etichettare tutte le rule con la label *model*. Grazie a questo vincolo è possibile utilizzare ricorsivamente la *metaXapply*; incrementando di volta in volta l'ultimo parametro (vedi sezione 2.1.3), si possono iterare tutte le soluzioni possibili scartando quelle già ottenute. Le soluzioni sono tutte le riscritture valide a partire dallo stato corrette. Per avere la sicurezza di una riscrittura completa, impongo che il Context, ovvero la sequenza di termini non riscritti dalla rule, sia vuoto.

Come strutture di supporto, dichiaro tre sort:

```
*** Action -> possibile rewrite, memorizza rate e meta-stato finale
*** ActionList -> Lista di azioni
*** TotalActions -> composto dalla lista delle azioni e dalla
***                                              somma dei loro rate
sort Action ActionList TotalActions .
subsort Action < ActionList .

op ac : Float Term -> Action .
op nil : -> ActionList .
op _ ; _ : ActionList ActionList -> ActionList [ ctor assoc id: nil prec 110 ]  .
*** Definisce un TotalAction (somma dei rate e action list)
op acts : Float ActionList -> TotalActions [ ctor ] .
*** Operazioni di utilità per concatenare e aggiungere Azioni
*** a TotalAction
op _+_ : TotalActions TotalActions -> TotalActions .
op _#_ : TotalActions Action -> TotalActions [ prec 90 ] .
```

Action è una struttura di supporto che rappresenta un *Config*, ovvero un $[Rate]State$, definito nel modulo CTMC. *Action* memorizza il rate in forma canonica e il meta-stato come *Term*. *ActionList* è una lista di *Action* e *TotalAction* è un contenitore di una *ActionList* e della somma dei rate delle singole *Action*.

Per eseguire ricorsivamente la *metaXapply* e riempire un TotalAction, ho definito l'operazione *applyAll*, di cui riporto anche l'implementazione, poiché è significativa nell'uso del meta-livello.

```
*** Costruisce la TotalActions contenente tutte le azioni
*** possibili a partire dal dato stato iniziale. Funzione tail.
op applyAll : Module Term TotalActions Nat -> TotalActions .

*** meta-applico ricorsivamente la rule "model" incrementando
*** il numero di soluzioni da scartare: così ottenengo tutte
*** le soluzioni possibili.
*** Il risultato della metaXApply è composto da 4 termini.
*** Due non sono utilizzati, gli altri due sono:
***           * Term -> meta-Config risultato dall'applicazione della rule
***           * Context -> lista dei termini non riscritti.
***                  Deve essere vuota ( '[]' )
***
ceq applyAll(Mod, S, acts(SumR, AL), ResIdx) =
        *** Ricorsione appendendo le ActionList
```

```
        applyAll(Mod, S, acts(SumR, AL) # ac(R2, S2), ResIdx + 1)
        *** Eseguo la metaXapply
    if  result? := metaXapply(Mod, S, 'model , none, 0, unbounded, ResIdx) /\
        *** Controllo che sia andato tutto bene e che il context sia vuoto
        result? :: Result4Tuple /\ getContext(result?) == [] /\
        *** ''[_']_[RT, S2] è il meta-config riscritto ([RT] S2)
        ''[_']_[RT, S2] := getTerm(result?) /\
        R2 := downTerm(RT, 0.0) .
```

Ottenute tutte le possibili azioni, bisogna sceglierne una casualmente, sulla base del rate (passo 2). Questo può essere implementato generando un numero casuale fra 0 e la somma dei rate e scegliendo una *Action* trattando i corrispettivi rate come se fossero delle probabilità.

```
*** Dato l'indice della sequenza di numeri random, ritorna un
*** valore casuale compreso fra 0 e 1 estremi inclusi.
op rand : Nat -> Float .

*** Sceglie un'azione sulla base del rate
op pickOne : ActionList Float Float -> Action .
```

Ottenuta la action, l'estrazione dello stato finale è banale (passo 3):

```
*** Ricava dall'azione il meta-stato finale
op mineAction : Action -> Term .
eq mineAction(ac(R, S)) = S .
```

Terminato il passo di simulazione, occorre aggiornare l'overview ed eventualmente comunicarlo all'utente (passo 4). Ovviamente anche l'overview deve essere trattato al meta-livello, quindi l'append progressiva di *Entry* alla *EntryList* avviene così:

```
*** ELTmp è il nuovo meta-EntryList
*** ELI è il meta-EntryList (Term) dello stato precedente
*** TimeF è il nuovo tempo
*** StateF è il nuovo meta-stato

ELTmp := '_-_[ELI , '_:_[upTerm(TimeF), StateF]]
```

Per quando riguarda la comunicazione dell'Overview, ho implementato una rewrite con profondità 1 dell'overview sul modulo utente. L'utente può sfruttare quell'unica rewrite (si ricorda che la rewrite automaticamente riduce tutti i termini), per stampare i risultati a video. L'overview riscritta diventa l'overview iniziale per gli step successivi. Questa scelta motiva la struttura dell'overview come contenitore di entry-list e non solo come entry-list, poiché una rewrite può avvenire anche su sottosequenze di una lista.

Anche in questo caso è necessario l'utilizzo del meta-livello per effettuare una *metaRewrite* dinamicamente sul modulo utente.

```
***
*** Controlla se occorre riscrivere l'overview nel modulo utente.
```

```
*** Ritorna la meta-EntryList a cui appendere i risultati successivi.
***   Module -> meta-modulo utente .
***   Term -> meta-EntryList .
***   Nat -> step di simulazione corrente.
***   Nat -> Overview size
op checkOverview : Module Term Nat Nat -> Term .
```

3.4 Simulazione

I passi di simulazione elencati nella sezione precedente (3.3.2) sono svolti ricorsivamente da un insieme di operazioni *simulate* che ho implementato come un sistema di equazioni. La simulazione termina o al compimento dei passi richiesti o al raggiungimento di un deadlock, ritornando un SimulationResult, che contiene il tempo, lo stato corrente ed un booleano che vale true se è stato raggiunto un deadlock.

```
*** SimulationResult -> composto da tempo + meta-stato corrente + deadlock?
sort SimulationResult .
op sim : Float Term Bool -> SimulationResult [ ctor ] .
```

Per avviare la simulazione si deve ridurre una operazione di entry point *simulate* a 5 argomenti, la quale richiama la *simulate* a 8 argomenti che esegue la simulazione ricorsiva.

```
***
*** Wrapper per la simulazione
***     Qid -> Nome del modulo utente.
***     Term -> meta-stato iniziale.
***     Nat -> Numero di passi di simulazione desiderati.
***     Nat -> Ogni quanti passi ricevere l'overview
***            (0 se non si vuole gestire l'Overview).
***     Nat -> Indice per la generazione dei numeri random
***                                        da cui iniziare.
op simulate : Qid Term Nat Nat Nat -> SimulationResult .

***
*** Operazione di simulazione da usare ricorsivamente.
***     Module -> meta-modulo utente.
***     SimulationResult -> risultato parziale.
***         Nat -> Numero di step totali.
***                Nat -> Numero di step effettuati.
***     Bool -> true se l'utente vuole ricevere l'overview.
***     Nat -> Dimensione target dell'overview.
***     EntryList -> contenuto dell'Overview
***     Nat -> Indice per la generazione dei numeri random .
op simulate : Module SimulationResult Nat Nat
          Bool Nat Nat EntryList Nat -> SimulationResult .
```

Riporto l'implementazione dell'operazione di simulazione ricorsiva, mostrando i passi descritti nella sezione 3.3.2.

```
var Mod : Module .
var A : Action .
var StateI StateF : Term .
var AL : ActionList .
var RandIdx Step OvSize MaxStep : Nat .
var SumR TimeI TimeF Tau1 Tau2 : Float .
var ELI ELF : Term .
var memOv : Bool .

ceq simulate(Mod, sim(TimeI, StateI, false), MaxStep, Step, memOv,
                                              OvSize, ELI, RandIdx) =
    simulate(Mod, sim(TimeF, StateF, false), MaxStep, Step + 1, memOv,
                                           OvSize, ELF, RandIdx + 2)
        *** Passo 1
    if  acts(SumR, AL) := applyAll(Mod, StateI) /\ SumR > 0.0 /\
        *** Passo 2
        Tau1 := rand(RandIdx) /\ Tau2 := rand(RandIdx + 1) /\
        A := pickOne(AL, SumR * Tau1, 0.0) /\
        *** Passo 3
        StateF := mineAction(A) /\
        TimeF := TimeI + log(1.0 / Tau2) / SumR /\
        *** Passo 4
        ELF := ( if memOv
                 then checkOverview(Mod,
                    '_-_[ELI , '_:_[upTerm(TimeF), StateF]], Step + 1, OvSize)
                 else
                    'empty.EntryList
               fi) .
```

Nota: ho progettato il simulatore per essere il più generale possibile, perciò delego all'utente finale il compito di scrivere dei wrapper per non dover avviare la simulazione specificando i meta-stati anziché gli stati.

3.5 Modelli di esempio

Di seguito riporto tre modelli CTMC implementati utilizzando il framework. In tutti i modelli ho condiviso un Sort che definisce liste di oggetti *Var*, che rappresentano variabili identificate da un nome e a cui è associato un valore:

```
***
*** Modulo di utilità, definisce liste di variabili da poter
*** utilizzare come state.
***
fmod CTMC-EXTRA is

        pr CTMC .

        sort Var VarList .
        subsort Var < VarList .
```

```
        op null : -> VarList .
        op __ : VarList VarList -> VarList [ ctor assoc comm id: null ] .
        op v : String  Universal -> Var [ ctor poly (2) ] .

endfm
```

3.5.1 Modello Readers-Writers

Il modello readers and writers rappresenta un sistema concorrente in cui N processi accedono ad una risorsa condivisa in lettura o scrittura. La lettura è shared, la scrittura exclusive. In figura 3.5.1 è mostrata la rete di petri che lo implementa.

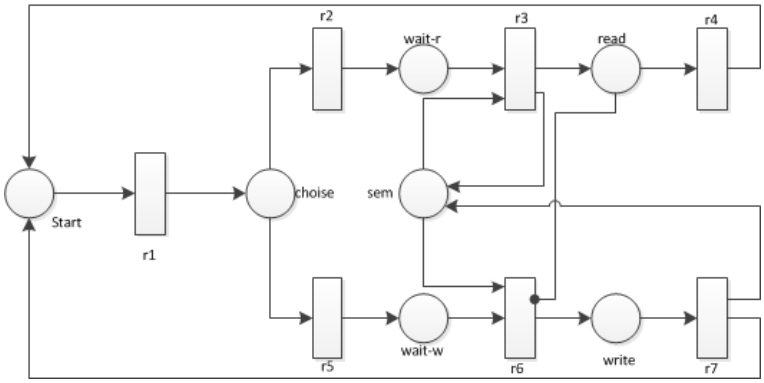

Figura 1: Rete di petri simulata da CTMC-READER_WRITER

Il modello si può esprimere in modo stocastico, facendo dipendere dal numero di token i rate da un place all'altro. Implementativamente, per rapprentare gli stati si utilizza un set di variabili, una per place, che hanno come valore il numero di token.

```
var VL : VarList .
var N, M : Nat .

*** Start
crl [model] : < v("start" , N) v("choose" , M) VL > =>
              [10.0 * float(N)]
                < v("start" , N + -1) v("choose" , M + 1) VL >
              if N > 0 .
*** Read
crl [model] : < v("choose" , N) v("waitr" , M) VL > =>
              [5.0 * float(N)]
                < v("choose" , N + -1) v("waitr" , M + 1) VL >
              if N > 0 .
crl [model] : < v("waitr" , N) v("read" , M) v("sem" , 1) VL > =>
```

```
             [20.0 * float(N)]
               < v("waitr" , N + -1) v("read" , M + 1) v("sem" , 1) VL >
             if N > 0 .
crl [model] : < v("read" , N) v("start" , M) VL > =>
             [50.0 * float(N)]
               < v("read" , N + -1) v("start" , M + 1) VL >
             if N > 0 .
*** Write
crl [model] : < v("choose" , N) v("waitw" , M) VL > =>
             [5.0 * float(N)]
               < v("choose" , N + -1) v("waitw" , M + 1) VL >
             if N > 0 .
crl [model] : < v("waitw" , N) v("write" , M) v("sem" , 1)
                                          v("read" , 0) VL > =>
             [40.0 * float(N)]
               < v("waitw" , N + -1) v("write" , M + 1)
                                          v("sem" , 0) v("read" ,0) VL >

             if N > 0 .
crl [model] : < v("write" , N) v("start" , M) v("sem" , 0) VL > =>
             [50.0 * float(N)]
               < v("write" , N + -1) v("start" , M + 1) v("sem" , 1) VL >
             if N > 0 .
```

3.5.2 Modello M/M/C/K per la simulazione delle code

M/M/C/K è un modello utilizzato nella teoria delle code per analizzare e simulare il comportamento di un sistema di C server che elaborano un task alla volta. I task nascono con un certo rate τ e sono memorizzati in un buffer di dimensione K. Se il buffer è pieno sono perduti. I task muoiono (sono svolti dai server) con un rate che vale $\mu * N$, dove N è il numero dei task, se $N \leq C$, altrimenti il rate vale $\mu * C$.

Gli stati del modello sono enumerabili da un numero naturale, tuttavia nel modello sono inclusi anche altri elementi di configurazione ed utilità come la dimensione del buffer, il numero di server e il numero di task perduti.

```
var N K L C : Nat .
var VL : VarList .

*** Accodamento nel Buffer
crl [model] : < v("buffersize" , K) v("buffer", N) VL > =>
              [ 0.535 ] < v("buffersize" , K) v("buffer", N + 1) VL >
              if N < K .
*** Buffer pieno
crl [model] : < v("buffersize" , K) v("buffer", N) v("losts", L) VL > =>
              [ 0.535 ] < v("buffersize" , K) v("buffer", N)
                                          v("losts", L + 1) VL >
              if N >= K .
```

```
*** Svolgimento di task (caso N < C)
crl [model] : < v("nserver" , C) v("buffer", N) VL > =>
              [0.120 * float(N)] < v("buffer", N + -1) v("nserver" , C) VL >
              if N > 0 /\ N < C .
*** Svolgimento di task (caso N >= C)
crl [model] : < v("nserver" , C) v("buffer", N) VL > =>
              [0.120 * float(C)] < v("buffer", N + -1) v("nserver" , C) VL >
              if N > 0 /\ N >= C .
```

3.6 Modello di test della Bromosulftaleina

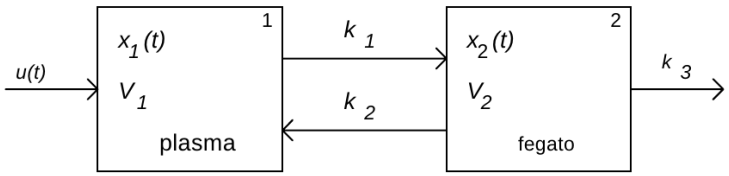

Figura 2: Modello BSF ([1])

Il test BSF [1] è semplificabile considerandolo un sistema a due compartimenti (uno per il fegato, l'altro per il plasma), con un certo rate di trasferimento. Si suppone un inserimento del colorante nel plasma istantaneo e si può semplificarlo imponendo la condizione inziale del plasma pari alla quantità di colorante iniettato. Il colorante, quando entra nel fegato, scompare con un certo rate e è riassorbito nel plasma con un altro rate. Per completezza si considera anche il volume dei due compartimenti (vedi figura 3.6).

Il modello è semplificato al caso di trasferimenti di colorante discreti.

```
var N K L : Nat .
var VP VF : Nat .
var VL : VarList .


*** Passaggio colorante dal compartimento plasma a fegato
crl [model] : < v("volplasma", VP) v("volfegato", VF)
                        v("plasma", N) v("fegato", L) > =>
              [ 0.454 * float(N) / float(VP) ]
              < v("volplasma", VP) v("volfegato", VF)
                v("plasma", N + -1)  v("fegato", L + 1) >
              if N > 0 /\ L < VF .

*** Passaggio colorante dal compartimento fegato a plasma
crl [model] : < v("volfegato", VF) v("volplasma", VP)
                        v("plasma", N) v("fegato", L) > =>
             [ 0.698 * float(L) / float(VF) ]
```

```
            < v("volfegato", VF) v("volplasma", VP)
              v("plasma", N + 1) v("fegato", L + -1) >
            if L > 0 /\ N < VP .

*** Scomparsa del colorante dal compartimento fegato
crl [model] : < v("volfegato", VF) v("fegato", L) VL > =>
            [ 0.432 * float(L) / float(VF) ]
            < v("volfegato", VF) v("fegato", L + -1) VL >
            if L > 0 .
```

4 Approximate Probabilistic Model Checking

La PCTL (*Probabilistic Computation Tree Logic*), permette di esprimere proprietà che si avverano con una probabilità $\rho \in J$ su di un particolare modello:

$$\phi ::= p|FALSE|\phi \rightarrow \phi'|P_J[\psi]$$

$$\psi ::= X\phi|\phi U\phi'|\phi U^I\phi'$$

Dato un modello CTMC, la verifica di proprietà PCTL è un poblema computazionalmente difficile a causa dell'esplosione degli stati da verificare. Una prima semplificazione si può ottenere considerando solo formule *monotone*, in grado di ridurre lo spazio degli stati per particolari proprietà. Una formula ψ è monotona se e solo se

$$\exists j \in \mathbb{N}, \forall \pi[j..], \pi[j..] \models \phi \wedge \forall i \in \mathbb{N}, i > j, \pi[i..] \models \phi$$

Ovvero, una formula è monotona se la validità su di uno stato implica la validità su tutte le path che partono da esso. Grazie a questa proprietà, la verifica di $Prob_{\geq P}[\psi]$ può essere limitata alla verifica $Prob_{\geq P}[\psi]_k$, ovvero limitata a path con profondità pari a k, poiché la probabilità che la formula ψ sia vera, può solo aumentare. Formule composte unicamente da $FALSE$, proposizioni p, implicazioni $\phi \rightarrow \phi'$ e operatori $X\phi$, $\phi U\phi'$ comprese le versioni bounded, sono formule monotone. Una grammatica per le formule monotone è la seguente:

$$\phi ::= p|FALSE|\phi \rightarrow \phi'|P_{\geq \rho}[\psi]$$

$$\psi ::= X\phi|\phi U\phi'|\phi U^I\phi'$$

Le formule monotone non risolvono il problema della difficoltà computazionale, ma rendono possibile applicare un procedimento approssimato, ottenendo risultati appartenenti ad un intorno del vero risultato con una certa confidenza.

Il calcolo della probabilità si può approssimare con la tecnica di *fully polynomial randomized approximation scheme (FPRAS)* [4]. Un FPRAS è un algoritmo $A(x, \epsilon, \delta, k), \epsilon \in [0, 1], \delta \in [0, 1]$ il quale, dato in input lo stato iniziale del sistema, una approssimazione, un parametro di confidenza e la lunghezza delle path ritorna un valore di probabilità tale che:

$$Prob[\mid A(x, \epsilon, \delta, k) - \mu(x) \mid \leq \epsilon] \geq 1 - \delta$$

dove $\mu(x)$ è il risultato esatto della formula analizzata.

Una implementazione di A è la seguente [2]:

$\mathrm{A}(x, \epsilon, \delta)$:
$N := \log \frac{2}{\delta} / 2\epsilon^2$
$A := 0$
for $i \in [1; N]$ do
1. Genera una path casuale π, di lunghezza k.
2. Se la formula ϕ da valutare è vera, $A := A + 1$.
$ReturnA/N$

L'algoritmo ha complessità polinomiale in $\frac{1}{\epsilon}$ e $\log \frac{1}{\delta}$.

5 Progetto PCTL

La seconda parte del progetto riguarda l'implementazione di un verificatore di proprietà PCTL utilizzando APMC.

5.1 Strategia risolutiva

Il nuovo sistema, il verificatore, segue l'algoritmo descritto nella sezione 4. L'approssimazione ϵ, il fattore di confidenza δ e la profondità massima delle path sono forniti dall'utente insieme allo stato iniziale, al modello e alla formula da verificare.

Per la generazione della path casuale, sfrutto il simulatore CTMC realizzato nella prima parte del progetto, ma anziché generare interamente una path di profondità massima, simulo il sistema un passo alla volta, valutando la formula in ogni nuovo stato.

In generale, per risolvere una formula seguo questo procedimento:

1. Se la formula $f \in \phi$: risolvo la formula per lo stato corrente.
2. Se la formula $f \in \psi, f = X\phi$: nello stato successivo verifico ϕ.
3. Se la formula $f \in \psi, f = \phi U \phi'$: se vale ϕ', f è vera, altrimenti se vale ϕ, nello stato successivo verifico $\phi U \phi'$. Se non vale né ϕ né ϕ', f non è vera.
4. Se la formula $f \in \psi, f = \phi U^{\leq \tau} \phi'$: se vale ϕ', f è vera, alrimenti se vale ϕ, nello stato successivo verifico $\phi U^{\leq \tau} \phi$. Se non vale né ϕ né ϕ', f non è vera. Se arrivo in uno stato in cui il tempo $t > \tau$, la formula f non è vera.

5.2 Sintassi delle Formule

All'interno del modulo *PCTL*, ho definito i sort e i costruttori necessari a definire le formule PCTL. Come predicato p, utilizzo meta-stato *Term*, con semantica $p \equiv S$, dove S è il meta-stato corrente, oppure $p(State, Condition)$ in grado di definire predicati contenenti stati che effettuano un match con lo stato corrente

e che devono rispettare certe condizioni con sintassi definita dal sort *Condition* [3].

```
*** Formula -> phi ; PathFormula -> psi
sort Formula PathFormula .

*** Un predicato p è un meta-stato.
subsort Term < Formula .
*** p(Term, Condition) effettua un matching di Term con lo stato corrente,
*** controllando la Condition.
op p : Term Condition -> Formula [ ctor ] .
op FALSE : -> Formula [ ctor ] .
op TRUE : -> Formula [ ctor ] .
op _ implies _ : Formula Formula -> Formula [ ctor prec 90 ] .
op _ and _ : Formula Formula -> Formula [ ctor prec 80 ] .
op _ or _ : Formula Formula -> Formula [ ctor prec 80 ] .
op not _ : Formula -> Formula [ ctor prec 75 ] .
op P[_](_) : Float PathFormula -> Formula [ ctor prec 70] .

op (_)U[_](_) : Formula Float Formula -> PathFormula [ ctor prec 92] .
op (_)U(_) : Formula Formula -> PathFormula [ ctor prec 92] .
op X_ : Formula -> PathFormula [ ctor ] .
*** Per risolvere ricorsivamente X phi, creo un altro operatore C phi:
*** true se phi vale nello stato corrente
op C_ : Formula -> PathFormula [ ctor ] .
```

Il costruttore $C\_$ è un artefatto che ho introdotto per semplificare la risoluzione ricorsiva delle *PathFormula* (ψ). Il significato semantico di $C\phi$ è "nello stato corrente deve valere ϕ". L'operatore *P[_](_)* rappresenta l'operatore di probabilità e il suo significato è : $Prob_{\geq p}(\psi)$.

L'operatore $p(S1, C)$ è verificato se , con stato corrente S, esiste un matching fra $S1$ e S che rispetti la condizione C. Questa procedura è facilmente implementabile attraverso la *metaXmatch* imponendo che il Context sia vuoto, ovvero che non esistano termini senza matching.

```
***
*** match(M, t1, t2, tc) verifica che esista un matching fra il meta-stato.
*** t1 e l'intero meta-stato t2, rispettando la condizione tc.
***   - Module : meta-modulo utente.
***   - Term : meta-stato da verificare.
***   - Term : meta-stato corrente.
***   - Condition : condizione che il meta-stato da verificare deve rispettare.
op match : Module Term Term Condition -> Bool .
```

5.3 Verifica di Proprietà

Per verificare una formula su un modello a partire da un certo stato iniziale, si utilizza l'operazione *verify*, specificando il nome del modulo utente, il meta-stato iniziale, la formula e tutti i parametri necessari. L'operazione *verify* funge

da wrapper per una operazione *verify* con differenti argomenti, che verifica la formula.

```
***
*** Verfica la formula data (entry point)
***   - Qid : Nome del modulo utente su cui c'è il modello .
***   - Formula : formula PCTL fa verificare.
***   - Term : meta-stato iniziale
***   - Float : approximation.
***   - Float : Confidence.
***   - Nat : Max Depth.
***   - Nat : indice di partenza per la generazione di numeri casuali.
op verify : Qid Formula Term Float Float Nat Nat -> Bool .

***
*** Verfica la formula data
***   - Module : meta-modulo utente.
***   - Formula : formula PCTL fa verificare.
***   - SimulationResult : meta-stato corrente + tempo + deadlock?
***   - Float : approximation.
***   - Float : Confidence.
***   - Nat : Max Depth.
***   - Nat : indice di partenza per la generazione di numeri casuali.
op verify : Module Formula SimulationResult Float Float Nat Nat -> Bool .
```

L'operazione verify, verifica le formule ϕ. Quando compare una formula del tipo $P[\_](\psi)$, è computato il numero di sample N da effettuare e verify richiama l'operazione *approxChecking* che per N volte crea una path random verificando la formula ψ. Come ulteriore miglioramento delle prestazioni, interrompo il model checking se la probabilità incrementale ha già raggiunto la probabilità target (le formule sono monotone, dunque la probabilità può solo aumentare ulteriormente).

```
***
*** Esegue un model checking approssimato della PathFormula data
***   - Module : meta-modulo utente.
***   - PathFormula : path-formula PCTL fa verificare.
***   - SimulationResult : Stato corrente + tempo.
***   - Nat : Numero di campioni corretti.
***   - Nat : Numero di campioni testati.
***   - Nat : Numero di campioni massimi.
***   - Float : Probabilità da raggiungere.
***   - Float : approximation.
***   - Float : Confidence.
***   - Nat : Max Depth.
***   - Nat : indice di partenza per la generazione di numeri casuali.
op approxChecking : Module PathFormula SimulationResult
                    Nat Nat Nat Float Float Float Nat Nat -> Float  .
```

Per verificare una formula all'interno durante la generazione di una path, ho creato una operazione *checkOnRandomPath* che ricorsivamente incrementa di

un passo la simulazione, verificando di volta in volta ψ, seguendo le strategie descritte nella sezione 5.1:

$\phi_1 U \phi_2$: nello stato corrente verifico ϕ_2, se è vera, allora la formula è verificata, altrimenti verifico ϕ_1. Se ϕ_1 è false, allora la formula è falsa, altrimenti avanzo la simulazione di un passo con l'operazione *simNcheck*, verificando nello stato successivo la stessa formula $\phi_1 U \phi_2$.

$\phi_1 U^{\leq \tau} \phi_2$: si comporta esattamente come la versione non bounded, ma prima di tutto controllo se $\tau \geq t$, dove t è il tempo nello stato corrente. Se la relazione è falsa, la formula non è verificata.

$X\phi$: avanzo la simulazione di un passo con l'operazione *simNcheck*, verificando nello stato successivo la formula $C\phi$.

$C\phi$: controllo nello stato corrente se ϕ è verificata.

Ovviamente la verifica di ϕ avviene richiamando l'operazione *verify*.

```
***
*** Controlla una PathFormula su una path casuale
***   - Module : meta-modulo utente.
***   - PathFormula : path-formula PCTL da verificare.
***   - SimulationResult : Stato corrente + tempo.
***   - Nat : Profondità corrente.
***   - Nat : Profondità massima.
***   - Float : approximation.
***   - Float : Confidence.
***   - Nat : indice di partenza per la generazione di numeri casuali.
op checkOnRandomPath : Module PathFormula SimulationResult
                            Nat Nat Float Float Nat -> Bool .

***
*** Esegue una simulazione di un passo e controlla la PathFormula
*** sul nuovo stato.
***   - Module : meta-modulo utente
***   - PathFormula : path-formula PCTL da verificare nello
***                                                 stato successivo.
***   - SimulationResult : Stato corrente + tempo
***   - Nat : Profondità corrente
***   - Nat : Profondità massima
***   - Float : approximation
***   - Float : Confidence
***   - Nat : indice di partenza per la generazione di numeri casuali
op simNcheck : Module PathFormula SimulationResult
                             Nat Nat Float Float Nat -> Bool .
```

5.4 Confronto con PRISM

Se con il framework realizzato si può avere una maggiore espressività rispetto a PRISM, lo si paga enormemente nelle prestazioni. Di seguito riporto dei

test che ho effettuato su tre moduli diversi, descritti nella sezione 3.5. PRISM mette a disposizione l'operatore $P =?[\ldots]$, che ritorna la probabilità anziché un booleano. Per ottenere lo stesso risultato nel framework, basta valutare $P[1.0](\ldots)$ e abilitare le print con il comando

```
Maude> set print attribute on .
```

Al termine della valutazione è stampato il valore di probabilità finale.

5.4.1 Modello readers-writers

Il modello readers-writers è descritto nella sezione 3.5.1. Di seguito riporto anche la sua implementazione in PRISM:

```
ctmc

// Numero di processi
const int N = 100;

module RW
start : [0..N] init N;
choise : [0..N] init 0;
waitr : [0..N] init 0;
waitw : [0..N] init 0;
sem : [0..N] init 1;
read : [0..N] init 0;
write : [0..N] init 0;

[t1] start>0 & choise<N  -> start :
     (start'=start-1)&(choise'=choise+1);
[t2] choise>0 & waitr<N & waitw<N -> choise :
     (choise'=choise-1) & (waitr'=waitr+1);
[t3] choise>0 & waitr<N & waitw<N -> choise :
     (choise'=choise-1) & (waitw'=waitw+1);
[t4] waitr>0 & sem>0 & read<N -> waitr*sem :
     (waitr'=waitr-1) & (read'=read+1);
[t5] waitw>0 & sem>0 & read=0 & write<N -> waitw*sem :
     (waitw'=waitw-1) & (sem'=sem-1) & (write'=write+1);
[t6] read>0 & start<N -> read :
     (read'=read-1) & (start'=start+1);
[t7] write>0 & sem<N & start<N -> write :
     (write'=write-1) & (start'=start+1) & (sem'=sem+1);

endmodule

// Definizione dei rate

const double r1=10;
const double r2=5;
const double r3=5;
```

```
const double r4=20;
const double r5=40;
const double r6=50;
const double r7=50;

module base_rate
[t1] true  -> r1 : true;
[t2] true  -> r2 : true;
[t3] true  -> r3 : true;
[t4] true  -> r4 : true;
[t5] true  -> r5 : true;
[t6] true  -> r6 : true;
[t7] true  -> r7 : true;
endmodule
```

Il test è stato fatto con condizione iniziale 100 processi, 10000 profondità massima, $\epsilon = 0.01$ e $\delta = 1.0 \cdot 10^{-10}$.

P=?[true U[0.1] waitw > 0 & read > 0] (la probabilità che entro 0.1 secondi, ci sia un processo in lettura ed uno in attesa di scrivere). La formula espressa nel framework è:

```
red in CTMC-RW_VERIFIER : verify(100,
 P[1.0] ( (TRUE) U[0.1] (
    p( upTerm(< v("read", N:Nat) v("waitw", M:Nat) V:VarList >),
                  upTerm(N:Nat > 0 and M:Nat > 0) = 'true.Bool )))
   , 0.01, 0.0000000001, 10000, 42793) .
```

Dove CTMC-RW_VERIFIER è un wrapper per facilitare la verifica delle operazioni di verifica sul modulo CTMC-READER_WRITER.

PRISM: tempo: 17.855 secondi . Probabilità: 0.7882288460727687

ENGINE: tempo: 2036.773 secondi . Probabilità: 0.77615413803280076

La differenza di probabilità è 0.012074708, quindi rientra nei limiti dell'approssimazione.

5.4.2 Modello BSF

Il modello del test della Bromosulftaleina è descritto nella sezione 3.6. Di seguito riporto anche la sua implementazione in PRISM:

```
ctmc
// Volume Plasma
const int VP = 1000;
// Volume Fegato
const int VF = 800;
// Colorante iniziale
const int IC = 500;
```

```
module BSF
Pl : [0..VP] init IC;
Fe : [0..VF] init 0;

[t1] Pl > 0 & Fe < VF -> Pl / VP :
      (Pl'=Pl-1) & (Fe'=Fe+1);
[t2] Fe > 0 & Pl < VP -> Fe / VF :
     (Pl'=Pl+1) & (Fe'=Fe-1);
[t3] Fe > 0 -> Fe / VF :
     (Fe'=Fe-1);

endmodule

//base rates

const double r1=0.454;
const double r2=0.698;
const double r3=0.432;

module base_rate
[t1] true  -> r1 : true;
[t2] true  -> r2 : true;
[t3] true  -> r3 : true;
endmodule
```

Il test è stato fatto con condizione iniziale 800 volume fegato, 1000 volume plasma, 500 colorante, 10000 profondità massima , $\epsilon = 0.05$ e $\delta = 1.0 \cdot 10^{-4}$. **P=?[true U[50000] Fe = 0 & Pl = 0]**. La formula espressa nel framework è:

```
red in CTMC-BSF_VERIFIER : verify(1000, 800, 500,
 P[1.0] ( (TRUE) U[50000.0] (
    upTerm(< v("plasma", 0) v("fegato", 0) V:VarList >)))
   , 0.05, 0.0001, 10000, 1351) .
```

Dove CTMC-BSF_VERIFIER è un wrapper per facilitare la verifica delle operazioni di verifica sul modulo CTMC-BSF.

PRISM: tempo: 6.874 secondi . Probabilità: 0.6431095406360424

ENGINE: tempo: 984.625 secondi . Probabilità: 0.69863705199394244

La differenza di probabilità è 0.055527511, quindi rientra nei limiti dell'approssimazione.

5.4.3 Modello MMCK

Il modello del test M/M/C/K è descritto nella sezione 3.5.2. Di seguito riporto anche la sua implementazione in PRISM:

```
ctmc
// Numero di server (C)
const int nServ = 5;
// Dimension del buffer (K)
const int buffSize = 10;

const int lostSize = 1000;

module MMCK
Buff : [0..buffSize] init 0;
Losts : [0..lostSize] init 0;

[t1] Buff < buffSize ->
     (Buff'=Buff+1);
[t2] Buff >= buffSize & Losts < lostSize ->
     (Losts'=Losts+1);
[t3] Buff > 0 & Buff < nServ -> Buff :
     (Buff'=Buff-1);
[t4] Buff >= nServ -> nServ :
     (Buff'=Buff-1);

endmodule

//base rates

const double r1=0.535;
const double r2=0.120;

module base_rate
[t1] true  -> r1 : true;
[t2] true  -> r1 : true;
[t3] true  -> r2 : true;
[t4] true  -> r2 : true;
endmodule
```

Il test è stato fatto con condizione iniziale 10 dimensione buffer, 5 server, profondità massima $K = 10000$, $\epsilon = 0.01$ e $\delta = 1.0 \cdot 10^{-10}$.

P=?[true U[70.0] losts > 0] (la probabilità che il buffer regga almeno 50 secondi). La formula espressa nel framework è:

```
red in CTMC-MMCK_VERIFIER : verify(10, 5,
 P[1.0] ( (TRUE) U[70.0] (
    upTerm(< v("losts", 1) V:VarList >)))
   , 0.01, 0.0000000001, 10000, 12523) .
```

Dove CTMC-MMCK_VERIFIER è un wrapper per facilitare la verifica delle operazioni di verifica sul modulo CTMC-MMCK.

PRISM: tempo: 6.689 secondi . Probabilità: 0.41349972595809265

ENGINE: tempo: 1227.478 secondi . Probabilità: 0.40806948016358197

La differenza di probabilità è 0.005430246, quindi rientra nei limiti dell'approssimazione.

Riferimenti bibliografici

[1] Gianni Gnudi. Dispense di bioingegneria - modelli matematici a compartimenti. AA 2010/2011.

[2] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 307–329. Springer, 2004.

[3] Steven Eker Patrick Lincoln Narciso Martí-Oliet José Meseguer Manuel Clavel, Francisco Durán and Carolyn Talcott. *Maude Manual (Version 2.6)*. Freely avaiable at http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf, January 2011.

[4] Bernhard Steffen and Giorgio Levi, editors. *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*. Springer, 2004.