Lezione 24 MSC
Concurrency Workbench

Edinburgh:

http://homepages.inf.ed.ac.uk/perdita/cwb/

Stony Brook:

http://www3.cs.stonybrook.edu/~cwb/

Roberto Gorrieri

What is CWB?

The Edinburgh Concurrency Workbench (CWB) is an automated tool which allows for the manipulation and analysis of concurrent systems.

For example, with the CWB it is possible to:

- define behaviours given in CCS, and perform various analyses on these behaviours, such as analysing the state space of a given process, or checking various semantic equivalences and preorders;
- define propositions in a powerful modal logic and check whether a given process satisfies a specification formulated in this logic;
- derive automatically logical formulae which distinguish nonequivalent processes;
- interactively simulate the behaviour of an agent, thus guiding it through its state space in a controlled fashion

CCS Syntax in CWB

The CCS syntax adopted in CWB is very similar to the one presented in Chapter 3:

- the empty process $\mathbf{0}$ is written as `0`;
- process constants are represented simply by their name; for example A is written as `A`, while A_1, A_2, as `A1`, `A2` and A', A'' as `A'`, `A''`;
- action prefixing, choice and parallel composition operators are represented by the same characters (for example, the process $a.\mathbf{0} \mid (b.\mathbf{0} + c.\mathbf{0})$ is written as `a.0 | (b.0 + c.0)`);
- output actions are expressed by an apex preceding the name of the channel (e.g. $\bar{a}.\mathbf{0}$ is written as `'a.0`): this is not to be confused with the apex used conventionally to distinguish different constant names, like $A, A', A'', \ldots$ that are written as `A`, `A'`, `A''`, `...`;
- the silent action τ is written as `tau`;
- the syntax of restriction is closer to the one adopted in [Mil80], where the set of restricted names closed in curly brackets follows the restricted process, separated by a backslash character: for example, $(\nu a)(a.\mathbf{0} \mid \bar{a}.\mathbf{0})$ is written as `(a.0 | 'a.0)\{a}`.

Il comando agent

- Prompt Command:
- Comando agent: permette di definire costanti e di aggiungerle all'ambiente corrente, che potrà essere salvato in un file con save (e ripreso da un file con input).

```
agent VMS1    = coin.VMS1c;
agent VMS1c   = askam.'amcoffee.VMS1 + coin.VMS1cc;
agent VMS1cc = askesp.'espcoffee.VMS1 +
               askam.'amcoffee.VMS1c;
```

Il comando set

- Permette di definire insiemi di azioni

```
set L = {flag1rf,flag1rt,flag1wf,flag1wt,flag2rf,flag2rt,flag2wf,
flag2wt,turnr1,turnr2,turnw1,turnw2};

agent HYMAN = (PROCESS1 | PROCESS2 | FLAG1 | FLAG2 | TURN)\L;
```

Comandi per equivalence checking

- maypre(A,B): verifica se A è minore o uguale a B nel weak trace preorder.
- mayeq(A,B): verifica se A e B sono weak trace equivalent.
- dftrace(A,B): verifica se A e B non sono weak trace equivalent, restituisce una traccia che li distingue.
- testeq(A,B): verifica se A e B sono testing/failure equivalent (appena più fine di completed trace equivalence)
- pre(A|Div, B): verifica se A è weak simulato da B (dove Div = tau.Div)
- eq(A,B): verifica se A e B sono weak bisimili.

Sintassi HML con ricorsione

- T per true
- F per false
- P | Q per P $\vee$ Q
- P & Q per P $\wedge$ Q
- <a>P e [a]P <<a>>P e [[a]]P <<eps>>P
- <->P per <Act>P e [-]P per [Act]P
- min(X.P) per X =min P
- max(X.P) per X =max P

Definire formule logiche

- Comandi prop e checkprop:

```
prop ME = [exitcs1]F | [exitcs2]F;
prop Inv(P) = max(X.P & [-]X);
checkprop(ALGORITHM,Inv(ME));
```

```
prop Livelock = max(X.<tau>X);
prop Poss(P) = min(X.P | <->X);
checkprop(ALGORITHM,Poss(Livelock));
```

In realtà è solo “divergent”

```
prop Deadlock = [-]F;
prop Poss(P) = min(X.P | <->X);
checkprop(ALGORITHM,Poss(Deadlock));
```

Altri comandi utili

- strongeq: verifica se i due processi sono strong bisimili
- deadlocks: identifica gli stati di deadlock raggiungibili da un processo
- dfstrong: trova una formula HML in grado di distinguere due processi non strong bisimili
- dfweak: trova una formula HML (con weak modalities) in grado di distinguere due processi non weak bisimili.

Algoritmo di mutua esclusione con semaforo

```
agent User = 'p.enter.exit.'v.User;
agent Sem = p.v.Sem;
agent Mutex2 = ( User[enter1/enter, exit1/exit] |
                 User[enter2/enter, exit2/exit] | Sem ) \ {p, v};
agent Spec1 = (enter1.exit1.Spec1 +
               enter2.exit2.Spec1);
```

- mayeq(Mutex2, Spec1) = true
- checkprop(Mutex2, Inv(ME)) = true
- checkprop(Mutex2, Poss(Deadlock)) = false
- checkprop(Mutex2, Poss(Livelock)) = false

Algoritmo di mutua esclusione con semaforo

```
agent User = 'p.enter.exit.'v.User;
agent Sem = p.v.Sem;
agent Mutex2 = ( User[enter1/enter, exit1/exit] |
                 User[enter2/enter, exit2/exit] | Sem ) \ {p, v};
agent Spec1 = (enter1.exit1.Spec1 +
               enter2.exit2.Spec1);
```

- eq(Mutex2,Spec1) = false
- dfweak(Mutex2,Spec1) = <<eps>>[[enter1]]F

```
agent Spec2 = (tau.enter1.exit1.Spec2 + tau.enter2.exit2.Spec2);
```

- eq(Mutex2,Spec2) = true

Algoritmo di mutua esclusione di Peterson

- checkprop(Peterson, Inv(ME)) = true
- checkprop(Peterson, Poss(Deadlock)) = false
- checkprop(Peterson, Poss(Livelock)) = true

- mayeq(Peterson, Spec2) = true
- eq(Peterson, Spec2) = false
- e ovviamente eq(Peterson, Mutex2) = false
- Ma perché?

Algoritmo di mutua esclusione di Peterson

- eq(Peterson, Spec2) = false
- dfweak(Peterson, Spec2) =

  P = <<eps>><<enter2>>[[exit2]]<<enter1>>T
- checkprop(Peterson, P) = true
- checkprop(Spec2, P) = false
- Cosa vuol dire? Istanza di una proprietà più generale: “Attesa limitata o, meglio, garanzia di non perdere il turno a favore di chi è già stato servito.”
- Questa “proprietà” dovrebbe valere per Peterson e non per Mutex2. Come definirla?

Modificare le specifiche

- Aggiungere le azioni try1 e try2 che denotano la volontà del processo P1 e P2 di entrare in sezione critica:

Mutex2 = (User1 | User2 | Sem) \{p,v}

- User1 = try1.'p.enter1.exit1.'v.User1
- User2 = try2.'p.enter2.exit2.'v.User2

Osservazione: se Mutex2 esegue prima try1 e poi try2, non è detto che la contesa sia stata vinta da User1 (solo chi esegui p per primo vince!)

Modificare le specifiche (2)

- Aggiungere le azioni try1 e try2 che denotano la volontà del processo P1 e P2 di entrare in sezione critica:

Peterson = (P1 | P2 | B1f | B2f | K1) \ L

P1 = ‘b1wt.’kw2.try1.P11 …

P2 = ‘b2wt.’kw1.try2.P21 …

Osservazione: se Peterson esegue prima try1 e poi try2, non è detto che la contesa sia stata vinta da P1 (vince chi esegue per primo l’assegnamento a K!)

Attesa Limitata

- prop Even(P) = min(X.P | (<->T & [-]X))
- prop Inv(P) = max(X.P & [-]X)
- prop AL = Inv([try1](Even(<exit1>T))) &
  Inv([try2](Even(<exit2>T)))

AL = “garanzia che se provo ad entrare allora in un tempo finito entrerò (ed uscirò)”

- Mutex2 non soddisfa AL (posso per sempre servire lo stesso User)

Peterson non soddisfa AL (!!!)

- Osserva che Peterson ==>* Q dove Q può sia fare exit1 sia andare in loop su Q: ovvero lo stato Q è uno stato in cui P1 è entrato e non è uscito dalla sezione critica, mentre P2 sta in loop di test.
- Poiché Even è un minimo punto fisso, non è vero che per tutte le computazioni in tempo finito riuscirò a servire l'altro: se il primo non esce dalla sezione critica … non potrò mai servire l'altro! Allora AL è troppo forte!
- Ma è anche troppo generica: "prima o poi servirò l'altro" dovrebbe invece essere "se il secondo ha chiesto di entrare, allora servirò il secondo prima di servire di nuovo il primo". Alternanza Garantita!

Proprietà AG – alternanza garantita

- **AG = Inv([try1]W1) & Inv([try2]W2)**
- W1 = R1 | R2 | R3 W2 = Q1 | Q2 | Q3
- R1 = (<<enter1>>T & [[enter2]]F) & (<<enter1>>T & [[try2]][[enter2]]F) *(P1 ha vinto la race ed entrerà!)*
- R2 = R21 & [[try2]]R21 *(P2 ha vinto la race ma non è ancora dentro la sezione critica: quando esce, non potrà essere servito di nuovo!)*
- R21 = [[enter2]][[exit2]]
  (<<enter1>>T & [[try2]][[enter2]]F)
- R3 = [[exit2]](<<enter1>>T & [[try2]][[enter2]]F) *(P2 è già dentro la sezione critica, ma non può essere servito di nuovo)*
- Q1, Q2 e Q3 simmetriche scambiando 1 con 2.

Chi soddisfa AG?

- Mutex2 non soddisfa AG (non garantisce alternanza)
- Peterson soddisfa AG
- Spec3 soddisfa AG
- mayeq(Peterson, Spec3) ma non eq(Peterson, Spec3))
- **Esercizio**: trovare Spec4 tale che eq(Peterson, Spec4)

```
agent Spec3 = try1.Spec3t1 + try2.Spec3t2;
agent Spec3t1 = try2.Spec3t1t2 + enter1.Spec3e1;
agent Spec3t2 = try1.Spec3t1t2 + enter2.Spec3e2;
agent Spec3t1t2 = enter1.Spec3t2e1 + enter2.Spec3t1e2;
agent Spec3e1 = exit1.Spec3 + try2.Spec3t2e1;
agent Spec3e2 = exit2.Spec3 + try1.Spec3t1e2;
agent Spec3t2e1 = exit1.(enter2.Spec3e2 + try1.enter2.Spec3t1e2);
agent Spec3t1e2 = exit2.(enter1.Spec3e1 + try2.enter1.Spec3t2e1);
```