PCD Module 2.1

Modellare l'esecuzione di un programma concorrente

Creazione dei modelli di programmi concorrenti

Dai programmi ai modelli e ritorno

- Importanza dei modelli e dell'astrazione:
  - modello: descrizione rigorosa / rappresentazione della struttura del programma ad un certo livello di astrazione
  - rappresentazioni diagrammatiche per la progettazione di programmi
  - modelli formali per l'analisi e la verifica dei programmi

- Definire modelli appropriati per programmi concorrenti
  - astraendo dai dettagli di basso livello nell'attuale implementazione e realizzazione
  - abilitando la possibilità di ragionare sul comportamento dinamico dei programmi concorrenti

Un modello per l'esecuzione di programmi concorrenti

- Modellare ogni processo come una sequenza di azioni **atomiche**: 1 azione = 1 istruzione
- Presupposto di indipendenza della velocità: modellare esecuzione concorrente come sequenza di azioni ottenuta sovrapponendo arbitrariamente le azioni atomichedai processi.
  - Un singolo processore globale astratto che esegue tutte le azioni
  - Istruzioni atomiche => eseguite al completamento senza possibilità di sovrapposizione
  - Durante la computazione del control pointer o istruzione che indica la prossima istruzione che può essere eseguita da quel processo

- una **computazione/scenario** è un'esecuzione di una sequenza che può accadere come risultato della sovrapposizione.

Slide pg. 5: esempio di azioni atomiche

Diagramma a stati e scenari di esecuzione

Dato un modello, l'esecuzione di un programma concorrente può essere formalmente rappresentata da:

- **stati**: tupla (1 elemento di ogni processo che è etichetta di tale processo)
- **transizioni** tra stati (se l'esecuzione in s1 cambia lo stato in s2>

- in un **diagramma a stati**: grafo contenente tutti gli stati raggiungibili dal programma. Gli scenari sono rappresentati da percorsi direzionati nel diagramma a stati. I cicli rappresentano possibili computazioni infinite.

Esempio di diagramma a stati pg. 7-15

Numero di scenari prodotti da n processi, ognuno con m_i azioni: $ns = \frac{(\sum_{i=1}^{n} m_i)!}{(\prod_{i=1}^{n} m_i)!}$

L'importanza di essere atomici

Assegnazioni e incrementi a livello codice macchina

Strutture non atomiche

"Atomiche" possono essere anche le strutture dati, se esse possono essere in un numero finito di stati uguali al numero di valori che possono assumere, tipicamente tipi primitivi in linguaggi concorrenti (non lo è double in java, e.g.). ADT composti da oggetti dati semplici multipli sono tipicamente nonatomici (es. classi OO).

In tal caso, negli ADT si possono verificare due tipi di stati:

- **internal**: significativo per chi definisce l'oggetto (classe)
- **external**: significativo per chi usa i dati

La corrispondenza è parziale: esistono stati interni che non hanno corrispondenze esterne. Stati interni con corrispondenze esterne sono definiti **consistenti**.

L'esecuzione di una operazione su un ADT non atomico può passare tramite stati non consistenti. Ciò non è problema in caso di programmi sequenziali, ma lo è nei programmi concorrenti. È quindi necessario introdurre meccanismi adatti che garantiscano che i processi lavorino su oggetti già in stato consistente.

Sovrapposizione arbitraria

L'intreccio arbitrario significa che ignoriamo il tempo nella nostra analisi dei programmi concorrenti, concentrandoci solo su

- ordini parziali relativi alle sequenze di azioni a1, a2, ...
- l'atomismo dell'azione individuale: scegliere ciò che è atomico è fondamentale

Robustezza rispetto all'hardware (processore) e alle modifiche del software, indipendente dalle variazioni dei tempi / prestazioni. Questo rende i programmi concorrenti suscettibili di **analisi formale**, che è necessaria per garantire la correttezza dei programmi concorrenti. - dimostrare la correttezza oltre al tempo effettivo di esecuzione, che è tipicamente strettamente dipendente dalla velocità del processore e dai tempi di gestione del sistema

Correttezza

Correttezza dei programmi

- Verifica della correttezza dei programmi sequenziali: **unit tests** basati su input specificato e aspettando qualche output specificato. Diagnosticare, correggere, ripetere il ciclo... ma rieseguire un programma con lo stesso ingresso dà sempre lo stesso risultato
- Nuova prospettiva impegnativa nella Programmazione concorrente: lo stesso ingresso può fornire diverse uscite, a seconda dello scenario. Alcuni scenari possono dare un'uscita corretta mentre altri non possiedono un debug di un programma concorrente nel modo normale. Ogni volta che si esegue il programma probabilmente avremo un diverso scenario.
- Necessità di diversi approcci: analisi formale, controllo dei modelli. Basati su modelli astratti

Correttezza dei programmi concorrenti

La correttezza di programmi (eventualmente non terminanti) concorrenti è definiti in termini di proprietà dei calcoli - condizioni che devono essere verificate in tutti i possibili scenari. Due tipi di proprietà di correttezza:

Proprietà di sicurezza

La proprietà deve essere sempre vera, cioè per una proprietà di sicurezza P da tenere, deve essere vero in ogni stato di ogni calcolo. Espresse come invarianti di un calcolo. Usato tipicamente per specificare che le "cose cattive" non dovrebbero mai accadere:

- mutex
- no deadlock
- ...

Proprietà di vivacità

La proprietà deve eventualmente diventare vera, cioè per una proprietà di vivacità P da tenere, deve essere vero che in ogni calcolo esiste in uno stato in cui P è vero. Usato tipicamente per specificare che "cose positive" accadranno

- no starvation
- no dormancy
- comunicazione affidabile

Imparzialità

Una proprietà di vivacità che detiene che qualcosa di buono accade infinitamente spesso. Esempio

principale: un processo attivato infinitamente spesso durante un'esecuzione dell'applicazione, ogni processo ottenendo un giusto andamento. Cioè un'azione che può essere eseguita, alla fine verrà eseguita.

Quindi i programmi possono avere diversi comportamenti di abilità a seconda di come vengono interlevate le loro istruzioni. Come le istruzioni sono interlevate è il risultato di una politica di pianificazione: **unconditional / weak / strong fairness**.