PCD Module 2.3

Verifica di programmi concorrenti - intro

Verifica e testing

Terminologia

- **falla**: manifestazione di un errore (azione che produce un risultato errato)
- **guasto**: quando una falla si verifica, può causare un guasto. Un guasto può essere causato da molti errori, una falla può causare molti guasti. I guasti sono relativi alle specifiche.
- **testing**: attività volta alla ricerca di guasti (e falle rilevanti)
- **verifica**: controllare se il sistema soddisfa le specifiche
- **validazione**: controllare se il sistema soddisfa le aspettative del cliente

Tipologia di test

- **test di sicurezza**: verificano che un comportamento di una classe o del sistema sia conforme alla sua specifica, di solito assumendo la forma di invarianti di test, in genere utilizzando le assertion.
- **test di vitalità**: test di progresso / non-progresso, molto difficile da impostare

Problemi e limiti di test:

- in genere controlla solo un sottoinsieme degli scenari
- fenomeno "heisenbugs": il codice di prova può introdurre manufatti temporali o sincronizzazione che possono mascherare gli errori

Testing vs verifiche

- Il *testing* è volto a verificare che una proprietà (correttezza) vale per un certo scenario selezionato: il test rivela la presenza di errori, non la loro assenza
- La *verifica* è volta a verificare che una proprietà vale per tutti i possibili scenari (tracce). Necessita di tecniche formali (es. model checking)

Metodi formali

- **model checking**: la verifica è eseguita generando uno per uno gli stati del sistema e controllando le proprietà stato per stato. Possono essere automatizzati tramite appositi strumenti.
- **prove di invarianza induttive**: proprietà invarianti sono provate per induzione sugli stati del sistema. Possono essere automatizzate tramite strumenti chiamati sistemi deduttivi.

Entrambe le tecniche si appoggiano su un qualche tipo di linguaggio formale/calcolo per specificare la correttezza delle proprietà.

Proprietà di correttezza nel calcolo delle proposizioni

Con il calcolo proposizionale, le proprietà di correttezza sono espresse come *formule logiche* che devono essere vere per verificare la proprietà in un qualche stato del sistema. Tali formule sono asserzioni ottenute componendo proposizioni tramite connettori logici (and/or/not/implicazioni/equivalenze).

Nel nostro caso, le proposizioni riguardano i *valori delle variabili* e i *puntatori di controllo* durante l'esecuzione di un programma concorrente. Ad esempio, data una variabile booleana `wantp`, una proposizione atomica (*asserzione*) `wantp` è vera in un certo stato se e solo se il valore dell variabile `wantp` è vera in tale stato.

Ogni etichetta di un'istruzione del processo sarà usata come proposizione atomica la cui interpretazione è "il puntatore di controllo di tale processo è attualmente a quella etichetta".

Esempio: mutua esclusione

```
bool wantp = false
bool wantq = false

p                       | q
----------------------------------------
loop forever            loop forever
p1: non-CS          q1: non-CS
p1: wantp = true        q2: wantq = true
p3: await !wantq        q3: await !wantp
p4: CS                  q4: CS
p5: wantp = false   q5: wantq = false
```

La formula $p_4 \wedge q_4$ è vera se entrambi i puntatori di controllo del processo sono all'interno della CS. Se esiste qualche stato in cui questa formula è vera, allora significa che la proprietà di mutex non è soddisfatta. Allo stesso modo, un programma soddisfa la proprietà mutex se la formula $\neg(p_4 \wedge q_4)$ è vera per ogni possibile stato di ogni scenario.

Logica temporale

Processi e sistemi cambiano il loro stato nel tempo, e quindi anche l'interpretazione delle formule riguardanti i relativi stati possono cambiare nel tempo. C'è bisogno di un linguaggio formale/calcolo che può prendere in considerazione tale aspetto. La *logica temporale* è uno dei principali e popolari.

La **logica temporale** è una logica formale ottenuta aggiungendo operatori temporali alla logica proposizionale o dei predicati:

- **Linear Temporal Logic (LTL)**: per esprimere proprietà che devono essere vere in un certo stato per ogni possibile scenario.
- **Branching Temporal Logics**: per esprimere proprietà che devono essere vere in alcuni o tutti gli scenari partendo da uno stato, esempio **CTL (computational tree logic)**

LTL: operatori temporali

LTL è basato su due operatori temporali di base:

- $\Box$ **sempre**: la formula è vera in uno stato s_i di una computazione se e solo se è vera in *tutti* gli stati $s_j, j \geq i$. Sinonimo: G (globally). Può essere usato per specificare *proprietà di sicurezza* (sempre vere).
- $\Diamond$ **eventualmente**: la formula è vera in uno stato s_i di una computazione se e solo se è vera in *alcuni* stati $s_j, j \geq i$. Sinonimo: F (finally). È usato per specificare *proprietà di vitalità* (qualcosa che potrebbe essere vero).

Proprietà base

- **Riflessività**: $\Box A \rightarrow A, A \rightarrow \Diamond A$
- **Dualità**: $\neg\Box A = \Diamond\neg A, \neg\Diamond A = \Box\neg A$
- **Sequenze di operatori**: $\Diamond\Box A, \Box\Diamond A$

Deduzioni con logiche temporali

La LT è un sistema formale di logica deduttiva con suoi assiomi e regole di inferenza. Può essere usato per formalizzare la semantica di programmi concorrenti e usata per provare rigorosamente le proprietà di correttezza dei programmi.

- Esempi:
  - $(\Diamond\Box A1 \wedge \Diamond\Box A2) \rightarrow \Diamond\Box(A1 \wedge A2)$ è vera
  - $(\Box\Diamond A1 \wedge \Box\Diamond A2) \rightarrow \Box\Diamond(A1 \wedge A2)$ è falsa

Specificare proprietà di sicurezza

$\Box P, P = \neg$, Q è la descrizione di uno stato incorretto.

Esempio: proprietà di mutua esclusione ($\Box\neg(p_3 \wedge q_3)$)

```
int turn = 1

p                       q
---------------------------------
loop forever            loop forever
p1: non-CS          q1: non-CS
p2: await turn = 1  q2: await turn = 2
p3: CS                  q3: CS
p4: turn = 2            q4: turn = 1
```

Specificare proprietà di vitalità

$\Diamond P$, dove P è la descrizione di un caso buono.

Esempio: proprietà di progresso (assenza di *starvation*) -- codice uguale a prima

- Proprietà di progresso per lo stato attuale: $p_2 \rightarrow \Diamond p_3$
- Proprietà di progresso per tutti gli stati: $\Box(p_2 \rightarrow \Diamond p_3)$

Operatore prossimo (next)

$\circle\phi$ è vero se la proprietà ϕ rimane nel prossimo stato, in tutti i casi possibili. Sinonimo: X (neXt)

Operatori binari

- $A \union B$ **fino a** (until): la formula è vera in uno stato s_i se e solo se
  - B è vero in *qualche* stato $s_j, j \geq i$, e
  - A è vero in *tutti* gli stati $s_k, i \leq k < j$
  - ovvero: finchè B è vera, A deve essere vera fino a che ciò succede ??
- AWB **fino a-debole**: come Until, ma non è richiesto che la formula B diventi eventualmente vera. Se non lo fa, A rimane vera indefinitamente (finchè A è falso, B deve essere vero).

sempre ed **eventualmente** possono essere definiti in termini **fino a**:

- $\Diamond A = true \union A$
- $\Box A = \neg\Diamond(\neg A)$

Sorpasso (overtaking)

Esempio: `try-p,{try-q,CSq,try-q,CSq,...,CSq} (x1000),CSp`

- Non è un esempio di starvation, ma è vero che $\Diamond CSp$ ed è evidente che la libertà da starvation può essere una proprietà molto debole
- In alcuni casi abbiamo bisogno di assicurarci che un processo possa entrare nella sua CS in una quantità di tempo ragionevole.

Proprietà di sorpasso k-delimitata

Dal momento in cui un processo p prova ad entrare nella sua CS, un altro processo può entrare al massimo k volte prima che p lo faccia.

Esempio: 3-overtaking `try-p,try-q,CSq,try-q,CSq,try-q,CSq,CSp`

La proprietà può essere espressa dall'operatore **fino a-debole**, es. 1-bounded overtaking:

`try_cs(P) => not cs(Q) W cs(Q) W not cs(Q) W cs(P) (with pars: try_cs(P) => not cs(Q) W (cs(Q) W (not cs(Q) W cs(P)))`

Tecniche di verifica

Model-checking

- È la tecnica più importante ed usata per controllare automaticamente la correttezza delle proprietà di un sistema concorrente.
- Strategia basata sulla ricerca esaustiva nell'intero spazio degli stati di un sistema e verificare se una certa proprietà è soddisfatta

    - Proprietà come predicati di uno stato del sistema o di stati, espressi come specifiche logiche ad esempio le formule di logica temporale proposizionale
    - Se il sistema soddisfa le proprietà, il model checker genera una risposta di **conferma**, altrimenti produce una **traccia** (*trace*, controesempio)

- MC può essere applicato anche all'hardware, es. NASA

Affrontare il problema dell'esplosione dello stato degli spazi

Il grande problema con le tecniche di model checking è la dimensione dello spazio degli stati. Tecniche principali:

- applicando regole di riduzione del numero di stati
- costruzione incrementale del grafo
- model checking simbolico

Model checker: SPIN + linguaggio PROMELA

Java Path Finder

MC specializzato nella verifica di programmi scritti in Java. Scritto dalla Nasa. È una JVM che esegue programmi in tutti gli scenari possibili (percorsi di esecuzione), controllando deadlock, eccezioni non gestite, errori...

Prove induttive di invarianza

- **Invarianza**: formula che deve essere invariabilmente vera in un qualsiasi punto di una qualsiasi computazione (es. $\neg(p_4 \wedge q_4)$)
- Provate tramite **induzione** su tutti gli stati di tutte le computazioni. Per provare che A è un'invariante, si prova che A è vero nello stato iniziale e si assume che A sia vero in uno stato generico S e si prova che A è vera in tutti i possibili stati prossimi a S.
- Sistemi deduttivi

Verifica delle proprietà di sicurezza e vitalità

- Proprietà di sicurezza: facile da verificare, deve essere vera in tutti gli stati. È sufficiente trovare uno stato che non verifica la proprietà per completare la verifica.
- Proprietà di vitalità: non è sufficiente controllare gli stati uno per uno, ma tutti i possibili sceenari. è perciò piu complesso.

Logica temporale delle azioni (TLA/TLA+)

Usato per specificare e verificare sistemi complessi nel mondo reale (es. Amazon).

Una specifica TLA+ descrive l'insieme di tutti i possibili comportamenti legali di un sistema. Può essere usato per descrivere le proprietà di correttezza desiderate nel sistema e il design del sistema.

Obiettivo: rendere possibile dimostrare che la progettazione corretta di un sistema implementa le proprietà di correttezza desiderate, tramite strumenti come TLC model checker.

PlusCal, linguaggio algoritmico basato su TLA+ per scrivere algoritmi.