Introduzione ai Metodi Formali

- **Sistemi software** anche molto complessi regolano la vita quotidiana, anche in situazioni **life-critical** (e.g. avionica) e **business-critical** (e.g. operazioni bancarie). Esempi di **disastri digitali**: Pentium bug, razzo Arianne, missile Scud, microprocessore viper, . . .

- È perciò essenziale garantire la **correttezza** del software.

- A tal fine è necessario sviluppare **metodi formali** per la **verifica** della **correttezza** dei programmi. Esempi di certificazioni: impianti stereo della Philips; nell'avionica.

- Nella gran parte dei sistemi sw sono coinvolte varie componenti che lavorano in modo **concorrente**.

- Perciò è necessario **verificare** la **correttezza** di **programmi concorrenti**.

Programmi sequenziali e concorrenti

In un **programma sequenziale** il **controllo** risiede in ogni istante in esattamente un punto.

In un **programma concorrente** il **controllo** può risiedere in ogni istante in più punti.

Considereremo **programmi concorrenti** di due tipi:

- **paralleli**: le componenti comunicano attraverso una **memoria condivisa**;
- **distribuiti**: le componenti hanno una **memoria locale** e comunicano solo attraverso **scambio di messaggi**.

È molto facile commettere errori durante la progettazione di programmi concorrenti (esplosione degli stati, race condition, ...)

Verifica Formale di Programmi

La **verifica formale** di **programmi** è un approccio sistematico alla dimostrazione della **correttezza** di un **programma** rispetto ad una **specifica**, basato su tecniche matematiche rigorose.

Che cosa si intende per **correttezza** di un programma?

- per **programmi sequenziali**: **correttezza** del **risultato** (rispetto alla specifica), **terminazione**;

- per **programmi concorrenti**: oltre a **correttezza** e **terminazione**, **fairness**, **no deadlock**, **no interferenza**.

Esempio: un programma concorrente parallelo

Problema (Specifica): Sia $f : \mathbf{Z} \to \mathbf{Z}$ tale che esiste $z \in \mathbf{Z}$. $f(z) = 0$ (z è uno **zero** di f). Scrivere un **programma concorrente ZERO** che trova z.

Idea: Scrivere due programmi sequenziali S_1 e S_2 che cercano gli **zeri positivi** ($z > 0$) e gli **zeri negativi** ($z \leq 0$), rispettivamente, e comporli in parallelo $[S_1||S_2]$.

Il programma $[S_1||S_2]$ termina quando sia S_1 che S_2 terminano.

Soluzione 1: **ZERO-1**$\equiv [S_1||S_2]$

$S_1 \equiv$ *found*:= **false**; $x := 0$;
while $\neg$*found* **do**
  $x := x + 1$;
  found $:= f(x) = 0$
od

$S_2 \equiv$ *found*:= **false**; $y := 1$;
while $\neg$*found* **do**
  $y := y - 1$;
  found:= $f(y) = 0$
od

La Soluzione 1 non è corretta

Il programma **ZERO-1** può **non** terminare.

Se f ha esattamente uno zero positivo e inizialmente è attiva la componente S_1 e rimane attiva finchè non trova lo zero, allora, se in quel momento la componente S_2 viene attivata, la variabile *found* viene ripristinata a **false**. Poiché non ci sono altri zeri, *found* non verrà più aggiornata a **true** e il programma **ZERO-1 non** termina.

L'errore è dovuto al fatto che *found* è inizializzata due volte, una in ciascuna componente.

Soluzione 2

ZERO-2$\equiv$ *found* $:=$ **false**; $[S_1 || S_2]$

$S_1 \equiv$ $x := 0$;
while $\neg$*found* **do**
$x := x + 1$;
found$:= f(x) = 0$
od

$S_2 \equiv$ $y := 1$;
while $\neg$*found* **do**
$y := y - 1$;
found$:= f(y) = 0$
od

La Soluzione 2 non è corretta

Il programma **ZERO-2** può **non** terminare.

Supponiamo che f abbia esattamente uno zero positivo e inizialmente sia attiva la componente S_2, finchè questa entra nel loop. A questo punto si attiva la componente S_1 finchè questa trova lo zero. Quindi la componente S_2 riparte e la variabile *found* è ripristinata a **false**. Poiché non ci sono altri zeri, *found* non verrà più aggiornata a **true** e il programma **ZERO-1 non** termina.

L'errore è dovuto al fatto che *found* può essere ripristinata *false* dopo essere stata aggiornata a *true*.

Soluzione 3

ZERO-3$\equiv$ *found* $:=$ **false**; $[S_1 || S_2]$

$S_1 \equiv$ $x := 0$;
while $\neg$*found* **do**
  $x := x + 1$;
  if $f(x) = 0$ **then** *found*:=**true** **fi**
od

$S_2 \equiv$ $y := 1$;
while $\neg$*found* **do**
  $y := y - 1$;
  if $f(y) = 0$ **then** *found*:=**true** **fi**
od

La Soluzione 3 è corretta?

Assumiamo che f abbia solo zeri positivi. Se la componente S_1 non viene mai attivata, allora il programma **ZERO-3 non** termina.

Ma questa esecuzione è legale? Dipende dalla definizione di composizione parallela.

Ci sono due possibilità:

1. L'esecuzione di $[S_1||S_2]$ consiste di un **interleaving** arbitrario di esecuzioni delle componenti S_1, S_2. E allora l'esecuzione sopra è legale e **ZERO-3 non** è corretto.

2. L'esecuzione di $[S_1||S_2]$ è **fair**, cioè ogni componente esegue prima o poi la sua prossima istruzione. In questo caso l'esecuzione sopra **non** è legale e **ZERO-3** è **corretto**.

Un costrutto per la fairness

Supponiamo che la fairness **non** sia garantita automaticamente.

Il programmatore può garantire la fairness del programma costruendo uno **scheduler** che forza ciascuna componente a eseguire **definitivamente** (prima o poi) la sua prossima istruzione.

A tal fine è necessario introdurre il costrutto

$$\textbf{await } B \textbf{ then } R \textbf{ end} \ ,$$

che sospende l'esecuzione di una componente.

Una componente esegue un'istruzione **await** se l'espressione booleana B vale **true**. Quindi l'istruzione R viene eseguita (come **azione atomica**), mentre tutte le altre componenti rimangono sospese. Se B vale **false**, la componente che esegue l'istruzione **await** è sospesa, mentre le altre possono procedere.

Problema: fornire una soluzione **fair** al problema dello zero.

Soluzione 4

ZERO-4$\equiv$ *found* := **false**;x:=0;y:=1; $[S_1||S_2]$

$S_1 \equiv$ **while** $\neg$*found* **do**
await $x \leq |y|$ **then** $x := x + 1$;
if $f(x) = 0$ **then** *found*:=**true** **fi**
od

$S_2 \equiv$ **while** $\neg$*found* **do**
await $|y| < x$ **then** $y := y - 1$;
if $f(y) = 0$ **then** *found*:=**true** **fi**
od

Soluzione 4 bis

ZERO-4$\equiv$ *turn* $:= 1$; *found* $:=$ **false**; $[S_1 || S_2]$

$S_1 \equiv$ $x := 0$;
while $\neg$*found* **do**
await *turn*=1 **then** *turn*:=2 **end**;
$x := x + 1$;
if $f(x) = 0$ **then** *found*:=**true** **fi**
od

$S_2 \equiv$ $y := 1$;
while $\neg$*found* **do**
await *turn*=2 **then** *turn*:=1 **end**;
$y := y - 1$;
if $f(y) = 0$ **then** *found*:=**true** **fi**
od

La Soluzione 4 è corretta?

L'istruzione **await** nelle due componenti garantisce l'alternanza di esecuzione tra S_1 e S_2. Quindi il programma è **fair**.

È **corretto**?

Assumiamo che f abbia esattamente uno zero positivo. Consideriamo un'esecuzione in cui S_1 ha appena trovato lo zero e si arresta prima di assegnare **true** alla variabile *found*. A questo punto S_2 viene eseguita per "un'iterazione e mezza" fino all'istruzione **await**. Poichè *turn* vale 1, S_2 si blocca. S_1 procede e termina, ma poichè *turn* resta uguale a 1, l'esecuzione di S_2 non potrà mai essere ripresa, cioè il programma è in **deadlock**.

Soluzione 5

ZERO-5$\equiv$ *turn* $:= 1$; *found* $:=$ **false**; $[S_1 || S_2]$

$S_1 \equiv$ $x := 0$;
while $\neg$*found* **do**
await *turn*=1 **then** *turn*:=2 **end**;
$x := x + 1$;
if $f(x) = 0$ **then** *found*:=**true** **fi**
od
turn:=2

$S_2 \equiv$ $y := 1$;
while $\neg$*found* **do**
await *turn*=2 **then** *turn*:=1 **end**;
$y := y - 1$;
if $f(y) = 0$ **then** *found*:=**true** **fi**
od
turn:=1

La Soluzione 5 è corretta?

Si può dimostrare **formalmente** che il programma **ZERO-5** è **corretto**.

Tuttavia **ZERO-5** può essere migliorato, riducendo le istruzioni all'interno del costrutto **await** e aumentando cosí il grado di **parallelismo** del programma.

Una soluzione ottimizzata

Soluzione 6: **ZERO-6**$\equiv$ *turn* $:= 1$; *found* $:=$ **false**; $[S_1||S_2]$

$S_1 \equiv$ $x := 0$;
while $\neg$*found* **do**
wait *turn*=1;
turn:=2;
$x := x + 1$;
if $f(x) = 0$ **then** *found*:=**true** **fi**
od
turn:=2

$S_2 \equiv$ $y := 1$;
while $\neg$*found* **do**
wait *turn*=2
turn:=1;
$y := y - 1$;
if $f(y) = 0$ **then** *found*:=**true** **fi**
od
turn:=1

dove **wait** B sospende la componente, se B vale *false*, non ha effetto, altrimenti.

La Soluzione 6 è corretta?

Rispetto al programma **ZERO-5**, dobbiamo garantire che S_2 **non interferisca** con il codice di S_1 compreso tra le istruzioni **wait** *turn*=1 e *turn*:=2, i.e. non porti ad un effetto indesiderato sulle variabili del programma (e vice versa S_1 non deve interferire con S_2).

Se un'istruzione di S_2 non contenente la variabile *turn* viene eseguita tra le due istruzioni di S_1, allora questa istruzione può essere scambiata con l'istruzione *turn*:=2 di S_1, ottenendo un'esecuzione di **ZERO-5**. Altrimenti, supponiamo che l'istruzione *turn*:=1 all'interno del loop di S_2 oppure quella fuori dal loop, sia eseguita subito dopo l'istruzione **wait** S_1. Il primo caso non si verifica, perchè vale *turn* =2, poichè S_1 ha appena eseguito la **wait**. Tuttavia il secondo caso potrebbe verificarsi, ma allora S_2 termina, quindi *found* è **true** e S_1 termina.

Si dimostra **formalmente** che il programma **ZERO-6** è **corretto**.

Conclusioni

- Anche programmi banali come **ZERO** possono nascondere errori difficili da scoprire.
- Una dimostrazione **informale** di **correttezza** non è sufficiente.
- Sono necessarie **tecniche formali** per stabilire la **correttezza** di programmi.

Correttezza Di Programmi

I programmi devono soddisfare alcune proprietà.

Per programmi **sequenziali**:

- **Correttezza parziale**: se il programma fornisce un output, esso è corretto rispetto alla specifica (non si richiede la terminazione).
- **Terminazione**.
- **Assenza di errori** (e.g. overflow, divisione per 0).

Per programmi **concorrenti**: oltre alle proprietà sopra,

- **Fairness**, i.e. ogni componente non terminata esegue la sua prossima istruzione definitivamente.
- **Assenza di deadlock**, i.e. non si verifica che componenti non terminate rimangano bloccate indefinitivamente nell'attesa che una condizione diventi vera.
- **Assenza di Interferenza (o race condition)**, i.e. nessuna componente parallela può manipolare le variabili, in modo da invalidare una certa proprietà di queste variabili stabilita in altre componenti.

Tecniche di Verifica Formale

- Approccio operazionale.

- Approccio assiomatico.