Quelli che s'innamoran di pratica sanza scientia
son come 'l nochiere ch'entra in navilio sanza timone o bussola
che mai a' certezza dove si vada;
sempre la pratica debbe essere edificata sopra la bona teorica.

– Leonardo da Vinci.

Linguaggi Concorrenti e Mobili

Prof. Massimo Merro

Anno Accademico 2006/2007

Verona, 3 Aprile 2007

L'arte di progettare sistemi complessi...

- Oggigiorno i sistemi informatici sono utilizzati per gestire aspetti nevralgici della nostra società. Si pensi, ad esempio, alla gestione

  – del trasporto aereo o ferroviario;

  – di banche e borse valori;

  – di centrali per la produzione di energia elettrica (siano esse ad idrocarburi, nucleari, solari, etc);

  – di sistemi satellitari;

  – di sistemi militari;

  – etc....

  L'*affidabilità* di tali sistemi è senza dubbio di primaria importanza. È facile immaginare il disagio, ed in alcuni casi la catastrofe, che il malfunzionamento di uno solo di questi sistemi potrebbe causare.

- Garantire l'affidabilità di un sistema informatico costituito da qualche centinaio di moduli che evolvono in maniera indipendente (cioé *concorrentemente*) è una cosa tutt'altro che facile.

- In generale, un sistema dicesi *corretto* se soddisfa un certo numero di proprietà di programmazione.

- Nel caso di *sistemi sequenziali*, ovvero sistemi in cui ad ogni istante il controllo si trova in un solo punto del programma, le proprietà desiderabili potrebbero, ad esempio, essere le seguenti:

  - *Correttezza Parziale*: se il programma fornisce un risultato allora tale risultato è corretto. "Parziale" sta a significare che il programma non necessariamente termina (e quindi fornisce un risultato).

  - *Terminazione*: il programma termina sempre.

  - *Assenza di fallimenti*, come ad esempio l'assenza di divisioni per zero.

- Quando però si ha a che fare con *sistemi concorrenti*, in cui il controllo, in certi istanti, può trovarsi in diversi punti del programma, allora tali proprietà sono più difficili da verificare.

- Inoltre nei sistemi concorrenti altre proprietà diventano interessanti:

  - *Assenza di interferenze*: ovvero non vi è un uso indesiderato di variabili condivise tra due o più moduli concorrenti.
  - *Assenza di stalli*: non vi sono due o più moduli che entrano in stallo aspettandosi vicendevolmente.
  - *Correttezza in condizioni di fairness*: cioè correttezza nell'ipotesi in cui lo scheduler riservi a tutti i moduli un appropriato tempo di esecuzione.

Alcuni famosi malfunzionamenti

La storia moderna è piena di malfunzionamenti, anche molto gravi, in sistemi informatici complessi:

- 1962: la sonda spaziale USA Mariner 1 viene distrutta per via di un trattino mancante nel programma di guida, scritto in Fortran. Costo: 80 milioni di dollari.

- 1985-1987: il software di radioterapia Therac-25 sottopone pazienti malati di cancro a sovradosaggi letali di radiazioni, mietendo quattro vittime.

- 1990: la rete interurbana della AT&T collassa per nove ore; la causa del collasso fu poi attribuita a una singola riga di codice errata.

- 1991: i missili Patriot nella guerra del Golfo intercettano numerosi missili Scud iracheni, ma falliscono frequentemente. Il fallimento più grave consente a uno Scud di uccidere 28 soldati americani a Dhahran, in Arabia Saudita. Il sistema antimissile era concepito per funzionare ininterrottamente per un massimo di 14 ore. Durante l'attacco a Dhahran, la batteria difensiva di Patriot era rimasta in funzione per 100 ore senza interruzioni...

- 1994: Il processore Pentium sbaglia a fare la divisione! La Intel si vede costretta a offrire la sostituzione gratuita di tutti i processori difettori, al costo stimato di 450 milioni di dollari.

- 1995: il software MacInTax della Intuit mette in piazza i segreti fiscali dei contribuenti americani. Il codice presente nei file di debug di MacInTax consente agli utenti Unix di accedere al computer centrale della Intuit, dove vengono conservate tutte le dichiarazioni redatte con MacInTax, e modificarle o cancellarle a piacimento.

- 1995: l'aeroporto avveniristico di Denver è messo in ginocchio dal sottosistema che si occupa della smistamento dei bagagli. Costo: 3.2 miliardi di dollari.

- 1996: il vettore Ariane 5 esplode al decollo. Il disastro avviene perché un programma del sistema di navigazione tenta di mettere un numero a 64 bit in una variabile a 16 bit. Il sistema, progettato per l'Ariane 4 è riciclato perché dimostratosi affidabile, tenta infatti di convertire la velocità laterale del missile dal formato a 64 bit al formato a 16 bit. Tuttavia l'Ariane 5 vola molto più velocemente dell'Ariane 4, per cui il valore della velocità laterale è più elevato di quanto possa essere gestito dalla routine di conversione, che a differenza di molte altre routine di conversione a bordo è priva dei normali controlli che evitano problemi di gestione.

Risultato: overflow, spegnimento del sistema di guida, e trasferimento del controllo al secondo sistema di guida, che però essendo progettato allo stesso modo va in tilt nella medesima maniera pochi millisecondi prima. Privo di guida, il vettore si autodistrugge.

Costo: 1 miliardo di euro.

- 1997: l'incrociatore lanciamissili USS Yorktown va in crash grazie a Windows NT per un *buffer overflow*...

- 2000: Millennium Bug!

Unteriori dettagli di possono essere facilmente rintracciare su Internet.

Tutti questi casi hanno hanno un minimo comune denominatore: i sistemi in questione sono molto complessi (ovviamente, relativamente al periodo in cui sono stati sviluppati) e composti da numerosi moduli concorrenti che interagiscono tra loro. Un banale errore non opportunamente gestito in uno di questi moduli si ripercuote disastrosamente su tutto il sistema.

- *Conclusione*: Urge una comprensione accurata dei sistemi complessi per poter finalmente progettare *correttamente* i sistemi di domani. Di conseguenza, metodologie basate sull'intuizione, o verifiche del tipo "*Questo modulo in genere funziona molto bene...*" non possono più essere applicate!

- Al contrario è necessario introdurre degli *strumenti matematici*, detti *modelli*, che consentano di rappresentare in maniera rigorosa il funzionamento dei sistemi complessi.

- Questi modelli, in linea di principio, dovrebbero stare ai sistemi infomatici come un sistema di equazioni differenziali sta ad un ponte o ad una casa.

- Lo scopo di questi modelli deve essere quello di consentire la rappresentazione della specifica del sistema in maniera chiara e rigorosa.

In modo da poter *verificare in maniera formale* che l'implementazione realizzata soddisfi la specifica in tutte le possibili situazioni (e senza dimenticare "casi sfortunati").

- L'utilizzo di tali sistemi è fondamentale già al momento della progettazione per prevedere le condizioni in cui un sistema non si comporta come previsto dalla sua specifica.

Modelli per sistemi trasformazionali

- Nel 1936 **Alan Turing** introdusse un modello di calcolo computazionale, oggi noto come la *Macchina di Turing*. Un modello semplice e geniale per rappresentare tutte le *funzioni calcolabili*.

- Nel 1941 **Alonzo Church** introdusse il λ-*calcolo* come strumento basilare per studiare la logica e la matematica. Il λ-calcolo modella la stessa classe di funzioni delle MdT ma cattura i concetti di *definizione ed applicazione di funzione* in maniera più chiara. La sintassi è la seguente:

$$M ::= x \mid \lambda x.M \mid M_1 M_2$$

e la regola di riduzione fondamentale è quella dell'applicazione di una funzione al suo argomento

$$\beta\text{-riduzione: } (\lambda x.M)N \ \rightarrow M\{x := N\}$$

Esempio: La funzione successore è definita come $\lambda x.x+1$, e quindi

$$(\lambda x.x+1)4 \ \rightarrow x+1\{x := 4\} = 5.$$

Si noti che il λ-calcolo consente di rappresentare non solo funzioni che prendono e ritornano semplici valori ma anche *funzionali* cioè funzioni che prendono come argomenti funzioni e ritornano funzioni.

- Sia le MdT che il λ-calcolo possono essere visti come dei semplici (ma espressivi) *linguaggi di programmazione*, in cui i *programmi* non sono altro che *funzioni.*

- Il vantaggio di avere linguaggi semplici consiste nel poterli studiare adeguatamente fornendone una semantica rigorosa.

- Il λ-calcolo può essere visto come il più semplice linguaggio in grado di fornire funzioni parametriche o procedure. Esso è infatti considerato il *cuore dei linguaggi funzionali.* Gli studi sul λ-calcolo ci hanno permesso di capire aspetti fondamentali della programmazione. Come il *passaggio dei parametri*, lo *scope* delle variabili, i *tipi*, etc tutti ingredienti cruciali nei linguaggi attuali.

- Comunque, sia le MdT che il λ-calcolo hanno una sotanziale *limitazione*: descrivono solamente sistemi *non-interattivi* o *trasfor-*

mazionali: cioè sistemi che dato un input, dopo una elaborazione, tornano (se terminano) un output.

- Ma i sistemi complessi a cui siamo interessati sono in generale *sistemi reattivi*, ovvero sistemi che, una volta in esecuzione, reagiscono ininterrottamente con gli inputs provenienti dall'esterno producendo outputs che possono interagire nuovamente col sistema senza che l'esecuzione abbia termine.

Modelli per Sistemi Reattivi

- Ma come sono fatti i sistemi reattivi?
- Esempi di sistemi reattivi sono:
  - Il vostro orologio al quarzo
  - Il vostro cellulare
  - Un sistema operativo....

- Ad esempio, in un sistema operativo di nuova concezione varie componenti interagiscono tra loro ed evolvono in maniera autonoma.

- In questo corso ci interesseremo ad una classe particolari dei sistemi reattivi: i *sistemi concorrenti* in cui entità chiamate *processi* evolvono concorrentemente interagendo tra loro.

- **Domanda:** Se il λ-calcolo (o le MdT) può essere visto come il modello più semplice per rappresentare sistemi trasformazionali, esiste un modello analogo che per i sistemi reattivi?

- Questa è stata una grande sfida iniziata negli anni '80 e che ha coinvolto centinaia di ricercatori.

- In prima approssimazione possiamo pensare di modellare un sistema reattivo attraverso più automi che interagiscono tra loro ed evolvono concorrentemente.

- **Domanda:** Quando possiamo dire che due sistemi reattivi si comportano nello stesso modo?

- La risposta a quest'ultima domanda è vitale perchè non possiamo sperare di capire il comportamento di un sistema reattivo se non siamo in grado di dire quando due sistemi si comportano nello stesso modo!

- Secondo la teoria classica degli automi, due automi sono equivalenti se riconoscono lo stesso linguaggio. Tale approccio, però, mostreremo essere totalmente inadeguato per sistemi reattivi.

- Uno dei concetti più innovativi introdotto negli ultimi 50 anni, è quello di *bisimulazione* (**David Park**, 1981), che ha radicalmente rivoluzionato vari settori della logica e della matematica. In prima approssimazione possiamo dire che due sistemi sono bisimili se sono in grado di simularsi reciprocamente.

- Più o meno nello stesso periodo **Robin Milner** (1980) sviluppò un modello algebrico semplice e maneggevole per rappresentare sistemi che interagiscono tra loro in maniera concorrente. Tale modello si chiama *CCS* (*Calculus of Communitating Systems*). Ben presto la bisimulazione di Park divenne l'ingrediente fondamentale per fornire un'adeguata semantica ai processi concorrenti descritti in CCS.

- Sempre in quel periodo, ed in maniera del tutto indipendente, **Tony Hoare** sviluppò un linguaggio, il *CSP* (Communicating Sequential Processes) anch'esso basato sul concetto fondamentale di *interazione tra processi*.

- Questa convergenza di vedute non fu casuale. Al contrario rappresentò un chiaro segno che il concetto di *interazione tra processi*, fosse un concetto fondamentale per la rappresentazione di sistemi concorrenti.

- Sia Tony Hoare che Robin Milner sono stati insigniti, del *Premio Turing*, massimo riconoscimento in campo informatico.

- Negli anni '80 e '90 il CCS è stato studiato in maniera intensiva da una vasta schiera di ricercatori in tutto il mondo con una ricaduta notevole sui linguaggi concorrenti attuali.

- Comunque, come già visto per il λ-calcolo, anche il CCS ha delle *limitazioni*. In particolare, la struttura di interconnessione tra processi è *staticamente determinata*. Cioè non può essere modificata per tutta la l'esecuzione del sistema. Non vi è modo di creare nuovi *canali* di comunicazione tra processi o di trasmettere l'indirizzo di un canale ad un processo per consentirgli di leggere e scrivere su quel canale.

Modelli per Sistemi Mobili

- È un dato di fatto che in sistemi complessi la struttura di interconnessione (o di comunicazione) tra processi evolve a tempo d'esecuzione.

- Un esempio per tutti, quando cliccate su un link di una pagina web, create una connessione (virtuale) tra il vostro processo locale ed un processo remoto che gestisce la pagina web.

- Tale link viene creato a run-time (quando cliccate) e viene distrutto a run-time (quando lasciate la pagina).

- Altri esempi di link costruiti e distrutti dinamicamente sono:

  - quelli tra un aereo e le varie torri di controllo a terra
  - quelli tra un cellulare e le varie antenne diffuse sul territorio
  - quando andate in giro col vostro laptop, ect...

- Tutti questi sistemi vengono chiamati *mobili*, in quanto la loro struttura di interconnessione può variare a tempo di esecuzione.

- Nel 1989, **Robin Milner**, **Joachim Parrow** e **David Walker** svilupparono un nuovo modello per sistemi concorrenti che consentisse alla struttura di comunicazione tra processi di variare a tempo di esecuzione: il π*-calcolo*.

- Anche in questo caso, l'avvento del π-calcolo ha monopolizzato l'interesse della comunità scientifica della Teoria dei sistemi concorrenti.

- In questi anni è stata eseguita una grande mole di lavoro sul π-calcolo da parte dei ricercatori. Molti progressi sono stati compiuti, da un punto di vista logico, di teoria dei tipi, uguaglianza comportamentali, ect..

- Anche il π-calcolo ha le sue *limitazioni* però! Ad esempio, presuppone che i processi siano in esecuzione nello stesso sito. Non modella cioè il concetto di *locazione* (virtuale o fisica) all'interno della quale possono essere eseguiti uno o più processi.

- **Domanda:** Oltre alla mobilità modellata dal π-calcolo, esistono altre forme di mobilità? Si per lo meno due: (i) la *mobilità di codice*, cioè programmi che vengono trasferiti da un posto ad un altro (si pensi al codice caricato tramite applets) (ii) *mobilità di agenti*, cioè la mobilità da un luogo ad altro di processi in corso d'esecuzione.

- Lo studio di queste due forme di mobilità rappresenta una delle ultime sfide nella Teoria dei sistemi concorrenti e distribuiti ed è conosciuta sotto il nome di *Global Computing*.

- La mobilità di codice in sistemi distribuiti è stata studiata, tra gli altri, da **Hennessy** e **Riely**, che nel 1998 hanno sviluppato un modello chiamato $D\pi$ (Distributed π-calculus).

- La mobilità di agenti in sistemi distribuiti è stata studiata, tra gli altri, da **Cardelli** e **Gordon** che, anch'essi nel 1998, hanno sviluppato un modello chiamato *MA*, *Mobile Ambients*.