8. Verifica e validazione: prove statiche

Docente: Tullio Vardanega
tullio.vardanega@math.unipd.it

Premessa - 1

- Un numero sempre maggiore di sistemi software svolge funzioni ad elevato contenuto di criticità
  - Sicurezza come *safety*
    - Prevenzione di condizioni di pericolo alle persone o alle cose
      - Sistemi di trasporto pubblico
      - Sistemi per il supporto di transazioni finanziarie
  - Sicurezza come *security*
    - Impedimento di intrusione
      - Sistemi per il trattamento dei dati personali
      - Sistemi per lo scambio di informazioni riservate

Premessa - 2

- Il software presente in tali sistemi deve
  - Esibire *attributi* o *capacità funzionali*
    - Determinano cosa il sistema deve fare
  - Possedere *caratteristiche non funzionali*
    - Determinano come ciò deve essere fatto
  - Soddisfare *proprietà* o *requisiti*
    - Di costruzione
    - D'uso
    - Di funzionamento

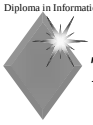
Premessa - 3

- Il pieno soddisfacimento di tutti i requisiti corrispondenti deve essere accertato *prima* dell'abilitazione all'uso del sistema
- Tale accertamento richiede verifica

> Verifica: conferma, mediante prova o presentazione di evidenza oggettiva, che determinati requisiti siano stati soddisfatti
>
> ISO 8402:1994 Quality management and quality assurance - vocabulary

Premessa - 4

- Nessun linguaggio di programmazione d'uso comune garantisce che ogni programma scritto in esso sia verificabile *per definizione*
- Per ogni scelta di linguaggio, occorre fare estrema attenzione al modo in cui esso possa essere utilizzato dall'applicazione
- La definizione di ciascun linguaggio di programmazione comporta scelte di bilanciamento tra funzionalità (potenza espressiva) ed integrità (idoneità alla verifica)



Tecniche di verifica - 1

- Tracciamento
  - La forma di verifica atta a dimostrare la *completezza* dell'implementazione
    - Identificando l'eventuale presenza di requisiti aggiuntivi (derivati) ed investigando il loro soddisfacimento
  - Ha luogo
    - Nel trattamento dei requisiti, tra requisiti elaborati (decomposti) ed originali (di alto livello)
    - Tra procedure di verifica e requisiti, disegno, codice
    - Tra codice sorgente e codice oggetto

Tecniche di verifica - 2

- Tracciamento (segue)
  - Certi stili di codifica (eventualmente facilitati dal linguaggio di programmazione in uso) si prestano maggiormente all'esecuzione di verifica mediante tracciamento
    - P.es.: assegnare singoli requisiti di basso livello a singoli moduli del programma, cosi da richiedere una sola procedura di test ed ottenere una più semplice corrispondenza tra essi
    - P.es.: maggiore é l'astrazione di un costrutto del linguaggio, maggiore é la quantità di codice oggetto generato per esso e maggiore é lo sforzo necessario per accertarne la corrispondenza

Tecniche di verifica - 3

- Revisioni
  - Strumento essenziale del processo di verifica
  - Possono essere condotte su
    - Requisiti, disegno, codice, procedure di verifica, risultati di verifica
  - Non sono automatizzabili
    - Richiedono la mediazione e l'interazione di individui
  - Possono essere formali od informali
    - Richiedono comunque l'indipendenza tra il verificato ed il verificatore
    - La semplicità e la leggibilità del codice sono particolarmente desiderabili quando non si possa ritenere che il verificatore sia un esperto del linguaggio

Tecniche di verifica - 4

- Analisi
  - *Statica*: applica a requisiti, progetto e codice
  - *Dinamica* (= test): applica a componenti del sistema od al sistema nella sua interezza
  - Gli standard di certificazione (vedi lezione 8) richiedono l'uso, in varie combinazioni, di 10 metodi di analisi statica

    1. **Flusso di controllo** 2. **Flusso dei dati** 3. **Flusso dell'informazione** 4. **Esecuzione simbolica** 5. **Verifica formale del codice** 6. **Verifica di limite** 7. **Uso dello stack** 8. **Comportamento temporale** 9. **Interferenza** 10. **Codice oggetto**

Analisi di flusso di controllo

- Ha l'obiettivo di
  - Assicurare che il codice sia eseguito nella sequenza attesa
  - Assicurare che il codice sia ben strutturato
  - Localizzare codice sintatticamente non raggiungibile
  - Identificare le parti del codice che possano porre problemi di terminazione (i.e.: chiamate ricorsive, iterazioni)
    - **Esempio**
      Analisi dell'albero delle chiamate (*call tree analysis*): mostra se l'ordine di chiamata specificato a progetto corrisponda a quello effettuato; rivela la presenza di ricorsione diretta o indiretta
    - **Esempio**
      Divieto/impedimento di modifica di variabili di controllo delle iterazioni

Analisi di flusso dei dati

- Accerta che *nessun* cammino d'esecuzione del programma acceda a variabili prive di valori significativi
- Usa i risultati dell'analisi di flusso di controllo insieme alle informazioni sulle modalità di accesso alle variabili (lettura, scrittura)
- Rileva possibili anomalie, come più scritture successive senza letture intermedie
- E` complicata dalla presenza e l'uso di dati globali che possano essere acceduti da ogni parte del programma

Analisi di flusso d'informazione

- Determina come l'esecuzione di una unità di codice crei dipendenze (e quali) tra i suoi ingressi e le uscite
- Le sole dipendenze consentite sono quelle previste dalla specifica
  - Consente l'identificazione di *effetti laterali* inattesi od indesiderati
- Può limitarsi ad un singolo modulo, a più moduli collegati, all'intero sistema

Esecuzione simbolica - 1

- Verifica proprietà del programma mediante manipolazione algebrica del codice sorgente senza bisogno di specifica formale
  - Si avvale di tecniche di analisi di flusso di controllo, di flusso di dati e di flusso di informazione
- Si esegue effettuando 'sostituzioni inverse'
  - Ad ogni (uso di) LHS di un assegnamento si sostituisce progressivamente il suo RHS
- Trasforma la logica sequenziale del programma in un insieme di *assegnamenti paralleli* nei quali i valori di uscita sono espressi come *funzione diretta* dei valori di ingresso

Esecuzione simbolica - 2

Assumendo assenza di *aliasing* e di effetti laterali di funzioni, dal seguente codice sorgente:

```
X := A+B;     -- X dipende da A e B
Y := D-C;     -- Y dipende da C e D
if X>0 then
   Z:=(Y+1); -- Z dipende da A, B, C e D
end if;
```

si ottiene:

```
A+B ≤ 0 ⇒
       X = A+B
       Y = D-C
       Z = conserva il valore precedente
A+B > 0 ⇒
       X = A+B
       Y = D-C
       Z = D-C+1
```


Verifica formale del codice

- Prova che il codice sorgente di un programma é corretto rispetto alla specifica formale dei suoi requisiti
  - Si pone l'obiettivo di esplorare *tutte* le esecuzioni possibili, ciò che non é ottenibile solo mediante prove dinamiche
- Correttezza parziale
  - Elaborazione e prova di teoremi (*condizioni di verifica*) la cui verità implica che il verificarsi di certe pre-condizioni e della terminazione del programma assicurano il verificarsi di certe post-condizioni
- Correttezza totale
  - Richiede prova di terminazione

Analisi di limite

- Verifica che i dati del programma restino entro i limiti del loro tipo e dell'accuratezza richiesta dall'applicazione
  - Analisi di *overflow* ed *underflow* nella rappresentazione di grandezze
  - Analisi di errori di arrotondamento
  - Verifica di valori di limite (*range checking*)
  - Analisi di limite di strutture
- Linguaggi evoluti assegnano limiti statici a tipi discreti consentendo verifiche automatiche sulle corrispondenti variabili
- Più problematico con tipi enumerati e reali

Analisi d'uso di stack

- Lo *stack* é l'area di memoria condivisa che i sottoprogrammi usano per immagazzinare dati locali, temporanei ed indirizzi di ritorno generati dal compilatore
- L'analisi determina la massima domanda di stack richiesta da un'esecuzione del programma e la relaziona con la dimensione dell'area fisica disponibile
- Verifica inoltre che non vi sia collisione tra stack (allocazioni statiche) ed *heap* (allocazioni dinamiche)
  - L'attuale Java non ha stack ma solo heap

Analisi temporale

- Concerne le proprietà temporali richieste ed esibite dalle dipendenze delle uscite dagli ingressi del programma
  - Produrre il valore giusto al momento giusto
- Carenze od eccessi del linguaggio di programmazione e delle tecniche di codifica possono complicare assai questa analisi
  - P.es.: iterazioni prive di limite statico, ricorso sistematico a strutture dati dinamiche

Analisi d'interferenza

- Mostra l'assenza di effetti di interferenza tra parti separate ('partizioni') del sistema
  - Non necessariamente limitate ai componenti software
- Veicoli tipici di interferenza
  - Memoria dinamica (*heap*) condivisa, dove parti separate di programma lasciano traccia di dati abbandonati ma non distrutti (*memory leak*)
  - Zone di I/O
  - Dispositivi condivisi programmabili con effetti a livello sistema (p.es.: watchdog)

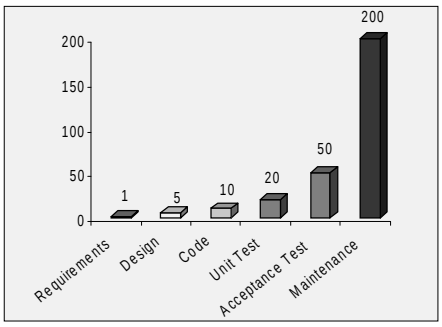
Analisi di codice oggetto

- Assicura che il codice oggetto da eseguire sia una traduzione corretta del codice sorgente corrispondente e che nessun errore (od omissione) sia stato/a introdotto dal compilatore
- Viene effettuata *manualmente*
- Viene facilitata dalle informazioni di corrispondenza prodotte dal compilatore

Scrivere programmi verificabili - 1

- Dalla scelta dei metodi di verifica richiesti discende la selezione di standard di codifica e di sottoinsiemi del linguaggio appropriati
  - L'uso di costrutti del linguaggio inadatti può compromettere la verificabilità del programma
- La verifica retrospettiva (a valle dello sviluppo) é *spesso* inadeguata
  - Sviluppo condotto senza tenere conto delle esigenze di verifica
  - Il costo di correzione di un errore é tanto più elevato quanto più avanzato é lo stadio di sviluppo

Costo di correzione di errori



Scrivere programmi verificabili - 3

- Eseguire ripetuti cicli di revisione-verifica *dopo* la rilevazione di ciascun errore é intollerabilmente laborioso
  - Approccio retrospettivo
- Assai più conveniente é effettuare analisi statiche *durante* la fase di codifica
  - Approccio costruttivo: correttezza per costruzione
    - Produce un miglior rapporto costo/qualità a fronte di un più elevato costo concettuale e progettuale

Scrivere programmi verificabili - 4

- Vi sono 4 ragioni fondamentali per richiedere o proibire l'uso di particolari costrutti del linguaggio di implementazione
  - Regole d'uso per assicurare predicibilità di comportamento
  - Regole d'uso per consentire modellazione del sistema
  - Regole d'uso per facilitare il test
  - Considerazioni pragmatiche

Regole d'uso per assicurare predicibilità di comportamento

- Il codice sorgente non deve presentare ambiguità
  - Casi tipici di ambiguità derivano da
    - Presenza di effetti laterali (p.es.: di funzioni): diverse invocazioni della stessa funzione producono risultati diversi
    - Effetti dell'ordine di elaborazione ed inizializzazione: l'esito di un programma può dipendere dall'ordine di elaborazione entro una unita e/o tra unita
    - Effetti dei meccanismi di passaggio dei parametri: la scelta di una modalità di passaggio (per copia, per riferimento), dove permessa, può influenzare l'esito dell'esecuzione

Regole d'uso per consentire modellazione del sistema

- I metodi di verifica statica comportano la costruzione di modelli del sistema da analizzare a partire dal codice del programma
- Le forme più semplici di analisi rappresentano il programma come un grafo diretto e studiano i percorsi possibili su di esso (storie di esecuzione)
  - Le transizioni tra stati (archi) hanno etichette che descrivono proprietà sintattiche o semantiche dell'istruzione corrispondente
    - La presenza di flussi di eccezione e di risoluzione dinamica di chiamata (*dispatching*) complica notevolmente la struttura del grafo
  - Ciascun flusso di controllo (*thread*) viene rappresentato ed analizzato separatamente

Regole d'uso per facilitare il test

- Due forme (o atteggiamenti) di test
  - Investigativo: informale (*debugging*)
  - Formale: aderente a standard richiesti
- Costrutti di linguaggio che sono di ostacolo (esempi):
  - La risoluzione dinamica di chiamata (*dispatching*) complica il test di copertura
  - La conversione forzata tra tipi (*casting*) complica l'analisi dell'identità dei dati
  - Le eccezioni predefinite complicano i test di copertura, presentando cammini di esecuzione difficili da raggiungere senza modificare il codice

Considerazioni pragmatiche

- L'efficacia dei metodi costruttivi di analisi statica e di test é funzione della qualità di strutturazione del codice
  - P.es.: ogni modulo con un solo punto di ingresso ed un solo punto di uscita
- La verifica di un programma mette in relazione frammenti di codice con frammenti di specifica
  - La verificabilità dipende dalla semplicità dell'informazione di contesto
    - Controllare e limitare gli ambiti (*scope*) e le visibilità
  - Conviene prestare attenzione all'architettura software, all'incapsulazione dello stato ed all'accesso ad esso