Ingegneria del Software mod. A
Verifica e validazione: analisi statiche

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

Premessa - 1

- Un numero crescente di sistemi *software* svolge funzioni con elevate caratteristiche di criticità
  - Sicurezza come *safety*
    - Prevenzione di condizioni di pericolo a persone o cose
      - P.es.: sistemi di trasporto pubblico
      - P.es.: sistemi per il supporto di transazioni finanziarie
  - Sicurezza come *security*
    - Prevenzione di intrusione
      - P.es.: sistemi per il trattamento dei dati personali
      - P.es.: 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 soddisfacimento di tutte le legittime aspettative deve essere accertato prima dell'abilitazione all'uso del sistema
- Accertamento mediante 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 qualunque programma scritto in esso sia, per definizione, verificabile
- Per ogni scelta di linguaggio, occorre fare estrema attenzione al modo in cui esso venga utilizzato dall'applicazione
- La progettazione di un linguaggio di programmazione comporta delicate scelte di bilanciamento tra funzionalità (potenza espressiva) ed integrità (idoneità alla verifica)

Tecniche di verifica - 1

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

Tecniche di verifica - 2

- Tracciamento (segue)
  - Certi stili di codifica (ancor più se facilitati dal linguaggio di programmazione) facilitano la verifica mediante tracciamento
    - P.es.: assegnare singoli requisiti di basso livello a singoli moduli del programma, così da richiedere una sola procedura di prova ed ottenere una più semplice corrispondenza tra essi
    - P.es.: maggiore l'astrazione di un costrutto del linguaggio, maggiore la quantità di codice oggetto generato per esso e maggiore l'onere di dimostrazione di 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 indipendenza tra verificato e 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 (prova, *test*): applica a componenti del sistema od al sistema nella sua interezza
  - Gli standard di certificazione 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
  - Accertare che il codice esegua nella sequenza attesa
  - Accertare che il codice sia ben strutturato
  - Localizzare codice non raggiungibile sintatticamente
  - Identificare parti del codice che possano porre problemi di terminazione (i.e.: chiamate ricorsive, iterazioni)
    - **Esempio**
      L'analisi dell'albero delle chiamate (*call tree analysis*) mostra se l'ordine di chiamata specificato a progetto corrisponda a quello effettivo, rivela la presenza di

Analisi di flusso dei dati

- Accerta che nessun cammino d'esecuzione del programma acceda a variabili prive di valore
- Usa i risultati dell'analisi di flusso di controllo insieme alle informazioni sulle modalità di accesso alle variabili (lettura, scrittura)
- Rileva possibili anomalie, p.es. più scritture successive senza letture intermedie
- È complicata dalla presenza e dall'uso di dati globali accedibili 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
  - Usa 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 il flusso 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, da:

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

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 la correttezza del codice sorgente rispetto alla specifica formale dei suoi requisiti
  - Esplora tutte le esecuzioni possibili, ciò che non è fattibile mediante prove dinamiche
- Correttezza parziale
  - Elaborazione e prova di teoremi (condizioni di verifica) la cui verità implica che il verificarsi di certe precondizioni assicura il verificarsi di determinate postcondizioni, sotto l'ipotesi di terminazione del programma

Analisi di limite

- Verifica che i dati del programma restino entro i limiti del loro tipo e della precisione desiderata
  - Analisi di *overflow* 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* è l'area di memoria 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 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* → implicazioni?

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 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*)
    - Azzeramento preventivo delle pagine di memoria riutilizzate
  - Zone di I/O
  - Dispositivi condivisi programmabili con effetti a livello sistema (p.es.: *watchdog*)

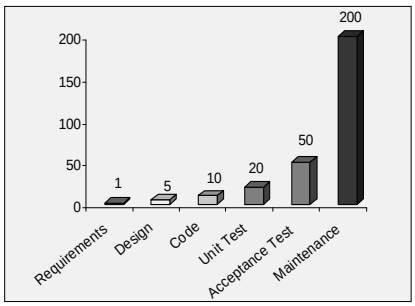
Analisi di codice oggetto

- 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

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 solo retrospettiva (ossia a valle dello sviluppo) è spesso inadeguata
  - Sviluppo condotto senza tenere conto delle esigenze di verifica è rischioso
  - Il costo di rilevazione e correzione di un errore è tanto maggiore quanto più avanzato è lo stadio di sviluppo

Costo di correzione di errori



Programmi verificabili - 3

- Eseguire ripetuti cicli di revisione-verifica dopo la rilevazione di ciascun errore è troppo oneroso
  - Approccio retrospettivo
- Conviene di più effettuare analisi statiche durante la fase di codifica
  - Approccio costruttivo: correttezza per costruzione
    - Produce un miglior rapporto costo/qualità a fronte di un maggior costo concettuale e progettuale

Programmi verificabili - 4

- 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 analisi del sistema
  - Regole d'uso per facilitare le prove
  - Considerazioni pragmatiche

Regole d'uso per assicurare predicibilità di comportamento

- Il codice sorgente non deve presentare ambiguità
  - 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 unità e/o tra unità
  - Effetti dei meccanismi di passaggio dei parametri: la scelta di una modalità di passaggio (per copia, per riferimento), ove permessa, può influenzare l'esito dell'esecuzione

Regole d'uso per consentire analisi del sistema

- La verifica statica richiede la costruzione di modelli del sistema da analizzare a partire dal codice del programma
- Rappresentano il programma come un grafo diretto e ne studiano i cammini possibili (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 le prove

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

Considerazioni pragmatiche

- L'efficacia dei metodi costruttivi di analisi statica e dinamica è 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 relaziona 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
  - Architettura *software*, incapsulazione dello stato ed accesso ad esso sono fattori decisivi