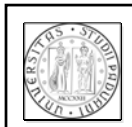


Verifica e validazione: analisi statica

Anno accademico 2013/14
Ingegneria del Software mod. A

Tullio Vardanega, tullio.vardanega@math.unipd.it

Premessa – 1

- **Un numero crescente di sistemi SW incorpora funzionalità critiche**
  - **Rispetto alla sicurezza intesa come *safety***
    - Prevenzione di condizioni di pericolo a persone o cose
      - Sistemi di trasporto pubblico
      - Sistemi per il supporto di transazioni finanziarie
      - Sistemi di controllo di impianti e di produzione
  - **Rispetto alla sicurezza intesa come *security***
    - Prevenzione di intrusioni
      - Sistemi per il trattamento dei dati personali
      - Sistemi per lo scambio di informazioni riservate

Premessa – 2

- **Il SW di tali sistemi deve esibire**
  - **Le capacità funzionali specificate nei requisiti**
    - Che determinano cosa il sistema deve fare
  - **Le caratteristiche non funzionali necessarie per garantire che il sistema lavori sempre come previsto**
- **Proprietà e requisiti dimostrabili**
  - **Di costruzione**
  - **D'uso**
  - **Di funzionamento**

Premessa – 3

- **Nessun linguaggio di programmazione garantisce a priori la completa verificabilità di ogni programma scritto in esso**
  - **La progettazione di un linguaggio di programmazione richiede bilanciamento tra funzionalità (potere espressivo) e integrità (onere di verifica)**
- **Scelto un linguaggio di sviluppo occorre valutare quali costrutti usare rispetto al loro impatto**
  - **Sulle caratteristiche di qualità e i costi di verifica**

Tecniche di verifica – 1

- **Tracciamento / I**
  - **Dimostrare completezza ed economicità della soluzione**
    - Soddisfacimento di tutti i requisiti
    - Nessuna funzionalità superflua o componente ingiustificato
  - **Ha luogo**
    - Tra requisiti *software* e requisiti utente
    - Tra procedure di verifica e requisiti, disegno, codice
    - Tra codice sorgente e codice oggetto (!)

Tecniche di verifica – 2

- **Tracciamento / II**
  - **Particolari stili di codifica facilitano la verifica mediante tracciamento**
    - Assegnare singoli requisiti di basso livello a singoli moduli del programma così da richiedere una sola procedura di prova e ottenere una più semplice corrispondenza tra essi
    - 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 (*joint review* / *audit*)**
  - **Strumento essenziale del processo di verifica**
  - **Possono essere condotte su**
    - Analisi, progettazione, codice, procedure e risultati di verifica
  - **Non sono automatizzabili**
    - Richiedono la mediazione e l'interazione di individui
  - **Possono essere formali (*audit*) o informali (*joint review*)**
    - Richiedono indipendenza tra verificato e verificatore
    - Richiedono semplicità e leggibilità di codice e diagrammi

Tipi 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**

1. Analisi di flusso di controllo

- Accertare che il codice esegua nella sequenza specificata
- Accertare che il codice sia ben strutturato
- Localizzare codice non raggiungibile
- Identificare segmenti d'esecuzione che possano non terminare
  - L'analisi dell'albero delle chiamate (*call-tree analysis*) mostra se l'ordine di chiamata corrisponda alla specifica e rileva la presenza di ricorsione diretta o indiretta
  - Divieto di modifica di variabili di controllo delle iterazioni

2. Analisi di flusso dei dati

- Accertare 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)
- Rilevare possibili anomalie
  - Esempio: più scritture successive senza letture intermedie
- Attività complicata dalla presenza e dall'uso di dati globali raggiungibili da ogni parte del programma

3. Analisi di flusso d'informazione

- Determinare quali dipendenze tra ingressi e uscite risultino dall'esecuzione di una unità di codice
- Le sole dipendenze consentite sono quelle previste dalla specifica
  - Consente l'identificazione di effetti laterali inattesi o indesiderati
- Può limitarsi a un singolo modulo oppure estendere a più moduli correlati oppure anche all'intero sistema

4. Esecuzione simbolica – 1

- Verificare proprietà del programma mediante manipolazione algebrica del codice sorgente
  - Combinando tecniche di analisi di flusso di controllo, di flusso di dati e di flusso di informazione
- Si esegue effettuando "sostituzioni inverse"
  - A ogni LHS di un assegnamento sostituendo il suo RHS
- Trasformare il flusso sequenziale del programma in un insieme di assegnamenti paralleli nei quali le uscite sono espresse come funzione degli ingressi

4. Esecuzione simbolica – 2

Assumendo assenza di *aliasing* e di effetti laterali di funzioni

```
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
```

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


5. Verifica formale del codice

- **Provare la correttezza del codice sorgente rispetto alla specifica algebrica dei requisiti**
  - **Esplorando tutte le esecuzioni possibili**
  - **Non fattibile tramite analisi dinamica**
- **Correttezza parziale**
  - **Le condizioni di verifica sono espresse come teoremi la cui verità implica certe pre-condizioni in ingresso e certe post-condizioni in uscita**
    - La prova di correttezza vale sotto l'ipotesi di terminazione del programma
    - La prova di correttezza totale richiede prova di terminazione

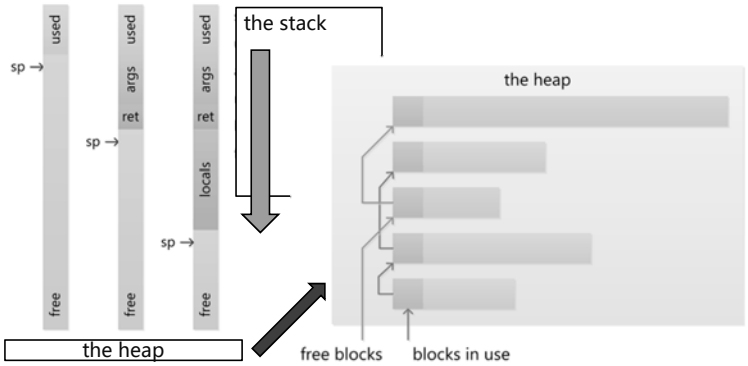
6. Analisi di limite

- **Verificare che i dati del programma restino entro i limiti del loro tipo e della precisione desiderata**
  - **Analisi di *overflow* e *underflow***
  - **Analisi di errori di arrotondamento**
  - **Rispetto dei limiti (*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**

7. Analisi d'uso di *stack* – 1

- **Lo *stack* è l'area di memoria che i sottoprogrammi usano per immagazzinare dati locali, temporanei e indirizzi di ritorno generati dal compilatore**
  - **Ogni flusso di controllo (*thread*) ha un suo proprio *stack***
  - **L'ampiezza dello *stack* cresce con l'annidamento di chiamate di procedura**
- **L'*heap* invece è l'area di memoria globale**
  - **L'ampiezza dell'*heap* è fissata a configurazione e poi consumata con la creazione dinamica di oggetti**

Stack & heap



7. Analisi d'uso di *stack* – 2

- **Determinare la massima domanda di *stack* richiesta da un'esecuzione in relazione con la dimensione dell'area di memoria assegnata al processo**
- **Verificare che non vi possa essere collisione tra *stack* e *heap* per qualche esecuzione**

8. Analisi temporale

- **Studiare le proprietà temporali richieste ed esibite dalle dipendenze delle uscite dagli ingressi del programma**
  - **Sapere di produrre il valore giusto al momento giusto**
- **Limiti espressivi dei linguaggi e delle tecniche di programmazione complicano questa analisi**
  - **Iterazioni prive di limite statico (*while*), ricorso sistematico a strutture dati dinamiche (*new*), ...**

9. Analisi d'interferenza

- **Mostrare l'assenza di effetti di interferenza tra parti separate ("partizioni") del sistema**
  - **Non necessariamente limitate a 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 (p.es. NT v5.x)
  - **I/O e altri dispositivi programmabili con effetti a livello sistema (esempio: *watchdog*)**

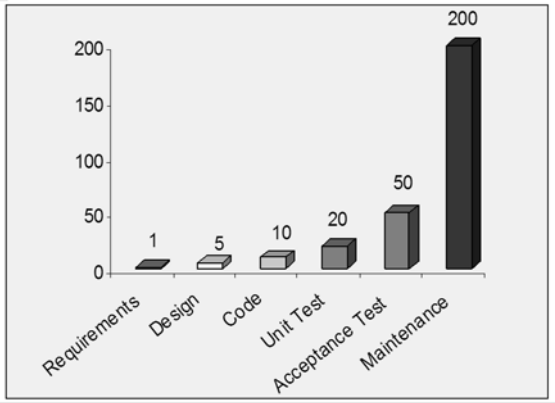
10. Analisi di codice oggetto

- **Assicurare che il codice oggetto da eseguire sia una traduzione corretta del codice sorgente corrispondente e che nessun errore od omissione siano stati introdotti dal compilatore**
  - **Viene ancora effettuata manualmente**
  - **Viene facilitata dalle informazioni di corrispondenza prodotte dal compilatore**

Programmi verificabili – 1

- **L'adozione di standard di codifica e di sottoinsiemi del linguaggio appropriati <u>discende</u> dalla scelta dei metodi di verifica richiesti**
  - **L'uso di costrutti del linguaggio inadatti può compromettere la verificabilità del programma**
- **La verifica solo retrospettiva (a valle dello sviluppo) è spesso inadeguata**
  - **Sviluppare tenendo sempre conto delle esigenze di verifica**
  - **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 – 2

- **Eseguire cicli «revisione – verifica» dopo ogni rilevazione di errore nel codice è inefficace e troppo oneroso**
  - **Approccio retrospettivo ⇨ *correctness by correction***
- **Molto meglio effettuare analisi statiche durante la codifica**
  - **Approccio costruttivo ⇨ *correctness by construction***

Programmi verificabili – 3

- **Prediligere o proibire l'uso di particolari costrutti del linguaggio di programmazione**
  - **Regole d'uso per assicurare comportamento predicibile**
  - **Regole d'uso per consentire analisi del sistema**
  - **Regole d'uso per facilitare le prove**
  - **Criteri di programmazione**
  - **Considerazioni pragmatiche**
- **Norme di progetto**

Comportamento predicibile

- **Codice sorgente senza ambiguità**
  - **Effetti laterali (p.es. di funzioni)**
    - Diverse invocazioni della stessa funzione producono risultati diversi
  - **Ordine di elaborazione e inizializzazione**
    - L'esito di un programma dipende dall'ordine di elaborazione entro e tra unità
    - Per esempio, l'attivazione dei *thread* in Java è fonte di impredicibilità
  - **Modalità di passaggio dei parametri**
    - La scelta di una modalità di passaggio (per valore, per riferimento) può influenzare l'esito dell'esecuzione

Analizzabilità del sistema

- **L'analisi statica costruisce modelli astratti del SW in esame**
  - **Questi modelli rappresentano ogni programma come un grafo diretto e ne studiano i cammini possibili**
  - **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 e analizzato separatamente**

Facilità di prova

- **Strategie di prova**
  - **Investigativa: informale e senza obiettivi prefissati di copertura**
  - **Formale: regolata da norme e grado di copertura fissato**
- **Alcuni costrutti di linguaggio sono di ostacolo**
  - **La risoluzione dinamica di chiamata (*dispatching*) complica le prove di copertura**
    - Vedi esempio: Per approfondire #28
  - **La conversione forzata tra tipi (*casting*) complica l'analisi dell'identità dei dati**
  - **Le eccezioni predefinite complicano le prove di copertura**
    - Cammini di esecuzione difficili da raggiungere senza modificare il codice

Criteri di programmazione

- **Riflettere l'architettura nel codice**
  - **Programmazione strutturata per esprimere componenti, moduli, unità e facilitare il riuso**
- **Separare le interfacce dall'implementazione**
  - **Fissare bene le interfacce già a partire dall'architettura logica**
- **Massimizzare l'incapsulazione (*information hiding*)**
  - **Usare membri privati e metodi pubblici per l'accesso**
- **Usare tipi specializzati per specificare dati**
  - **La composizione e la specializzazione aumentano il potere espressivo del sistema di tipi del programma**

Considerazioni pragmatiche

- **L'efficacia dei metodi di analisi è funzione della qualità di strutturazione del codice**
  - **Esempio: ogni modulo dovrebbe avere un solo punto di ingresso e un solo punto di uscita**
- **La verifica di un programma relaziona frammenti di codice con frammenti di specifica**
  - **La verificabilità è funzione della dimensione del contesto**
    - Controllare e limitare gli ambiti (*scope*) e la visibilità
  - **Una buona architettura facilita la verifica**
    - Esempio: incapsulazione dello stato e controllo di accesso