Testing e Verifica del Software

1.1 Introduzione

Validazione = sto facendo le cose che mi servono?
Verifica = sto facendo le cose nel modo corretto?

Problema indecidibile = problema di decisione per cui si sa che non esiste un algoritmo che ci dica con certezza se quel problema dara una risposta (halting problem = se termina o meno)

Testing esaustivo = controlla tutto, infattibile
Testing approssimato =

1) pessimistico : non garantisce di accettare un programma anche se va bene

2) ottimistico : potrebbe accettare dei programmi anche se potrebbero portare degli errori

Proprieta semplificate = proprietà estese ma diminuite di gradi di liberta

1) Inaccuratezza ottimistica : potremmo non accettare un programma anche se ha la proprietà che vogliamo
2) Inaccuratezza pessimistica : potremmo accettare alcuni programmi che non hanno la proprietà che vogliamo

Safe = Sicuro = non ha inaccuratezza ottimistica, accetta solo programmi validi

Sound = Corretto = se l'analisi passa allora il programma e' corretto (e' tipo un implicazione forte, cioe se l'analisi passa allora il programma non può non essere non corretto). un programma potrebbe essere corretto ma anche non passare (conservativo).

Complete = Completo = se passa l'analisi ogni programma corretto.

1.2 Testing Process

Testing = valutazione del prodotto in esecuzione con un set finito di input basato su un criteria dal quale estraiamo una misura di correttezza

1) Dynamic = in esecuzione
2) Finite = set finito di input
3) Selected = criterio selezionato
4) Expected = comportamento auspicabile dal quale estrarre la misura di correttezza

Black-box = si controlla da fuori, basandosi sulle specifiche, non sul codice
White-box = si controlla il codice

Passi principali:

1) Designing = cosa = cosa testare, in che contesto e con che criterio
2) Executing = come = effettuare il test, come analizzare i risultati e come catalogarli come fallimenti del test
3) Verifying = test = capire quanto e' potente il nostro test (traceability matrix tra test cases e requirements)

Tipi di testing:

a) Manual testing = tutto a mano, conveniente se poca roba
b) Capture and Replay = re-run dei casi di test vecchi su nuove versioni del software, comodo ma poco robusto se i cambiamenti sono tanti
c) Script-based = lanciati in automatico, ma vanno "aggiornati" insieme al software, costo di manutenzione
d) Program-based = esecuzione del codice e verdetto inclusi nel test, comodi da rilanciare ma sono piu onerosi da scrivere e manutenere

Testing Process	Solved Problems	Remaining Problems
Manual Testing	Functional testing	Imprecise coverage of SUT functionality No capabilities for regression testing Very costly process (every test execution is done manually) No effective measurement of test coverage
Capture/ Replay	Makes it possible to automatically reexecute captured test cases	Imprecise coverage of SUT functionality Weak capabilities for regression testing (very sensitive to GUI changes) Costly process (each change implies recapturing test cases manually)

Script- Based Testing	Makes it possible to automatically execute and reexecute test scripts	Imprecise coverage of SUT functionality Complex scripts are difficult to write and maintain Requirements traceability is developed manually (costly process)
Program-based Testing	No extra language is required	It may require additional effort during maintenance

e) Model-based = modella sia il software sia l'ambiente di test, generando test astratti rendendoli poi eseguibili come test che vengono valutati dopo l'esecuzione.

Formal Verification = tecniche che creano una prova matematica di consistenza tra una rappresentazione (design) e una specifica.

Program Verification = dice che un programma e' corretto rispettivamente ad una specifica, ed e' diverso dal testing esaustivo, perché quello non prova la correttezza (correttezza relativa alla specifica ≠ correttezza assoluta).

Runtime Verification = controlla a runtime che un programma fa quello che deve fare.

Formal Model Verification = e' la formal verification applicata a un modello effettivamente tangibile e verificabile (tipo FSM, Finite State Machine).

Model Checking = controllo esaustivo ed automatico del modello rispetto ad una specifica.

2.0 Code Testing

Test = collaudo del sw mediante prova = esecuzione specifica di casi di test.

Testing e' discontinuo, perché il software ha dei punti critici, quindi dei requisiti. E' efficace per trovare bug ma non può provarne l'assenza.

Testing : automatizzato, in ogni fase di sviluppo , esteso su tutto il sistema, pianificato, seguire standard.

Testing vs analisi statica:
Testing = trova tanti errori banali, fa fatica a trovare pochi errori critici
Analisi statica = più costosa, ma mira a errori poco probabili e molto critici

Failure = Guasto = Malfunzionamento = esecuzione errata, parte dinamica
Fault = Bug = Difetto = elemento del programma non conforme che genera il malfunzionamento
Errore = fattore che causa il difetto (solitamente errore umano)

Il testing cerca di evidenziare i bug per scoprire gli errori e valutare un sw con una certa confidenza

Tipi di test:
1) accettazione = comportamento e requisiti dell'utente sono allineati (es. magari e' leggermente diverso dalle specifiche ma e' piu usabile)
2) conformità = comportamento e requisiti delle specifiche
3) sistema = controlla sia sw sia hw come monolitico
4) integrazione = controllo sulla cooperazione delle unità
5) unita = test della singola unità
6) regressione = test delle release successive

In java lo Unit Test e' a livello di class.
I vari metodi sono i Test Unit e si introducono:
a) Test Driver = metodo che chiama il Test Unit con i parametri
b) Test Stub = metodo che impersona eventuali metodi interni

I test di integrazione guardano i domini dei dati, la loro rappresentazione e la compatibilità. Si può sviluppare top-down (richiedono stubs), bottom-up (richiedono drivers) o big-bang.

I test di regressione mirano ad accertarsi di aver fixato i bug e di non averne creati di nuovi controllando la compatibilità con i vecchi casi di test per poterli riusare.

Metodologie di testing:

1) White-box = usa il codice, controlla la copertura, ma non garantisce che il sw effettivamente faccia cio che e' stato richiesto
    a) Program-based = esamina tutto il sw per trovare punti critici, crea casi di test che soddisfano le coperture, applica i vari input e controlla che l'output sia corretto senza il verificarsi di errori. Usa il codice, ma non trova "errori di omissione", cioè parti che non sono proprio state implementate perché non guarda la specifica. Non fornisce un oracolo.

2) Black-box = usa i requisiti per controllare il sw, creando dei casi di test e confrontandoli con l'elaborazione prevista da quella effettiva, lo fa solamente usando input e output del programma.
    a) Specification-based = esamina la specifica, seleziona dei casi per i test, li applica sia alla specifica sia al sw e poi ne confronta gli output. Funziona solamente da oracolo, e ha problemi se non ci sono le specifiche o non sono formali.

Test Oracle = modo per stabilire se il test ha evidenziato un malfunzionamento oppure no.

2.1.1 Concetti Base

Programma P = funzione da D a R, cioè, P: D → R
e può essere non definito per qualche d in D

Predicato OK(P,d) = se P e' corretto per l'input specifico d, cioè se produce il
risultato atteso P(d).

OK(P) = se P e' corretto, se per ogni d in D ho che OK(P,d)

Failure (malfunzionamento) = se P eseguito con d ≠ P(d), cioè o ottengo un valore P'(d)
oppure il programma si blocca

Fault (difetto o bug) = P e' implementato in P', P' ≠ da P

Errore = il motivo del difetto

Caso di test (test case) = elemento di D, chiamato t

Test set T(test suite) = insieme finito di D, chiamato T, con dentro i vari t

OK(P,T) = se per ogni t in T ho OK(P,t), cioè non ho scoperto errori per
tutti i casi di test.

Se non ho scoperto errori il test set e' negativo, se not OK il test e' positivo, cioè ho scoperto errori.
SE UN TEST HA SUCCESSO HA TROVATO UN BUG!

Test set Ideale = insieme finito di casi di test che basta per essere sicuri di non avere difetti, l'esecuzione senza errori sul test set ideale implica la correttezza di P, cioè OK(P,T') → OK(P)

Test set esaustivo = e' un tipo di test ideale, che prova tutte le combinazioni possibili, ma se D e' infinito non va bene

Specifica S.

Criterio C = C: P x S x T, oppure può essere un generatore di Test Set T partendo da P e S.

Test Criteria (adeguatezza) [Cps(T)] = funzione che e' vera solo se T e' adeguato a trovare ogni difetto in P rispetto a S secondo il criterio C.

a) Program-based Test Criteria :
    i) C non dipende da S, C: P x T
b) Specification-based Test Criteria :
    i) C non dipende da P, C: S x T

Teorema di Goodenough e Gerhart

Dato Criterio C, Programma P e Test Suite T, con C ideale, cioè affidabile per P e valido per P, allora se seleziono T con C e non trovo malfunzionamenti in P allora T e' ideale.

Se P passa un test set T ideale allora P e' corretto.

Teorema di Howden

Non esiste un algoritmo che dato un qualsiasi programma P generi un test ideale
finito.

Teorema di Dijkstra

Il test di un programma può rilevare la presenza di malfunzionamenti ma non
dimostrarne l'assenza. CIOÈ IL TESTING NON PUÒ DIMOSTRARE CORRETTEZZA DEL SOFTWARE

Adeguatezza del test = rilassamento dell' "idealità" dei test, praticamente impossibili da ottenere.

Test case specification = requisito minimo di uno o più casi di test
Test obligation = una proprietà obbligatoria da testare durante i casi di test
Test di adeguatezza = predicato che e' vero o falso se un programma passa o meno una test suite
Criterio di adeguatezza = gruppo di Test obligation.

Un test soddisfa un criterio di adeguatezza se passa tutti i test e tutti i test obligation sono verificati da almeno un test case.

Satisfiability e Unsatisfiability = a volte non esistono test suite che possono soddisfare un determinato criterio di un programma. possiamo misurare la coverage dei test.

Subsumes relation = Test ad criterion A subsumes Tac B iff for every program P every test suite satisfying A also satisfies B

2.1.2 Testing basato sui programmi

Copertura = parte del programma che viene eseguita dai casi di test, relativa al flusso di controllo.

Flusso di controllo = grafo che accorpa tutti i cammini prendendo in considerazione tutte le possibilità. Viene considerato nel testing strutturale. Ad ogni esecuzione corrisponde un cammino

Criterio = $P \times T \rightarrow \{True, False\}$

Coperture:

a) Statement coverage = copertura istruzioni
b) Branch coverage = copertura archi
c) Decision coverage = espressione booleana = predicato, a guarda di una istruzione condizionale o di una iterativa
d) Condition coverage = espressione booleana atomica, non divisibile
e) MCC = Multiple Condition Coverage
f) MCDC = Modified Condition/Decision Coverage

Decisione = x > 0 and y < 10
Condizione 1 = x > 0
Condizione 2 = y < 10

a) Statement coverage
Un test set T e' adeguato per il criterio di Statement Coverage se per ogni istruzione S esiste almeno un caso di test t in T che la esegue.
E' un criterio debole, ma possiamo usarlo come misura come statement eseguiti / totali.

b) Branch coverage
Un test set T soddisfa il criterio di Branch Coverage se ogni arco e' percorso almeno una volta.

Lo Statement Coverage e' più debole, il branch coverage implica lo statement coverage.
(esempio : eseguire un if positivo una volta sola lo rende coperto (SC), ma si coprirà solamente il suo arco positivo(BC) ma non quello negativo(BC))

Il branch coverage può essere misurato come branch eseguiti / branch totali.

Generazione test:
procedo all'indietro dal basso verso l'alto, dopo aver generato il grafico di flusso
cercando di eliminare le coperture multiple. a volte possono esserci situazioni non
copribili (codice logicamente irraggiungibile aka un errore nel codice)

c) Decision coverage
Un test set T e' adeguato per il criterio di copertura delle decisioni se per ogni decisione di P esiste un caso t in cui la decisione e' presa e un caso t in cui la decisione non e' presa.
E' simile al branch coverage

d) Condition coverage
Un test set T e' adeguato per criterio di copertura delle condizioni se per ogni condizione di P esiste un caso t in cui la condizione e' vera e un caso t in cui e' falsa

Condition coverage non implica Decision coverage (esempio, con A or B, se copro A,B true e A,B false ho coperto tutte le decisioni ma non tutte le condizioni, per esempio A true e B false non l'avevo coperta).
Nemmeno l'opposto, cioè uno non implica l'altro, infatti:
Condition richiede che ogni condizione sia vera o falsa,
Decision che ogni decisione sia vera o falsa.
Ci sono combinazioni che vanno bene per uno ma non per l'altro in entrambi i sensi.

Alcuni compilatori usano la Short Circuit Evaluation in cui semplificano le espressioni (tipo se A && B e ho A falso skippo B, oppure se ho A || B se ho A vero skippo B). A volte possono essere evitati semplicemente mettendo & al posto di && o | al posto di ||

e) MCC
Un test soddisfa MCC se testa ogni combinazione dei valori di verità delle condizioni.

Con n condizioni si hanno 2^n combinazioni.
Di solito si costruisce una tabella

f) MCDC
Criterio più forte del condition coverage ma lineare in n e non merdosamente 2^n come MCC.
Un test soddisfa MCDC se ogni condizione all'interno di una decisione deve far variare in modo indipendente il valore finale della decisione.
Cioe' ogni condizione ha due casi di test dove il valore finale della decisione cambi in base solamente alla condizione in esame tenendo ferme tutte le altre.

Path = sequenza di branch
Path testing = criterio teorico che testa tutti i possibili path

Il numero di path potrebbe essere illimitato se ci sono cicli etc etc, quindi si limitano i loop o le lunghezze dei path.
Si raggruppano path simili, che si dividono internamente, con la tecnica del Boundary Interior Path Testing, perché tanto mi interessa percorrerli tutti e unicamente, quelli multipli possono essere ottimizzati.

Potrebbero generarsi comunque problemi di cicli infiniti, percio' e' stato introdotto il criterio di adeguatezza loop boundary sse per ogni loop:
- in almeno un t, il loop e' iterato 0 volte
- in almeno un t, il loop e' iterato esattamente 1 volta
- in almeno un t, il loop e' iterato almeno più di una volta

Unsatisfiability of criteria:
nel software e' comune trovare cose non raggiungibili, tipo statement illogici, condizioni mai matchate, percorsi non percorribili… e' sensato abbassare il goal di coverage < 100% pero' giustificando gli elementi non coperti (fossil code)

2.2 Test Execution

Scaffolding = Ponteggi = codice di supporto allo sviluppo, non visibile o addirittura non mantenuto alla fine. Include drivers (main temporanei), stubs (sostituti per funzioni), harnesses(imbragature, sostituti per hardware)…

Scrivere il codice in funzione dei test, per esempio non implementando tutto nella gui ma scrivendo API e WRAPPER (Controller e Observer)

Oracoli = ciò che ci dice se un test e' fallito o passato

Tipi di oracoli:

1) Comparison-based : controllano l'output predetto con quello effettivo
2) Self-Checking code : senza dover prevedere l'output, ma avviene tutto internamente
3) Human : a volte tocca metterci mano, però almeno possiamo ottimizzare la ripetizione dei test con una strategia Capture and Replay.

Selenium (browser capture-replay), con driver per browser specifico. Oppure Selendroid.

Mock object = oggetto simulato che imita gli stessi comportamenti di quello reale in maniera controllata.
Solitamente usata su DB o oggetti complessi o oggetti che esistono in parte (ai quali posso specificare solo ciò che mi interessa) oppure oggetti che ancora non esistono.
Mockito si usa per creare degli stubs

CI = Continuous Integration = il codice viene spesso sottoposto a controlli e a build di test, così si rilevano velocemente i problemi (regolarmente tipo più volte al giorno).

2.3.1 Unit Testing (JUnit e Codecover)

Codecover = copertura decisioni e condizioni

JUnit = framework fantastico e divertentissimo per Java che permette di vivere mille avventure spassose.
Si ispira all'eXtreme Programming (e no, non c'e' nulla che grida di piu' "fine anni 90' / inizio 2000" di un nome del genere, giusto per capire quanto e' vecchia sta roba)

Test-Driven Development Cycle:

1) scrivere i casi di test prima del codice partendo dalla specifica
2) esegui i test che falliranno sicuro
3) scrivi il codice fino a che non riesci a far passare tutti i test
4) riparti da 1

Good practices

- per testare una classe (Es. Videos.java) creo una classe con la X davanti al nome (Es. X…)
- la classe di test contiene metodi annotati con @Test che rappresentano i casi di test
- e' meglio tener separato codice e test, infatti e' norma fare cartella src e cartella test

Testare un metodo:

1) creare oggetti nella classe sotto test
2) chiamare il metodo e ottenere il risultato
3) confrontare il risultato ottenuto con quello atteso (con metodi di assert di JUnit)

NON COMPLETO!

2.3.2 Mutation Testing

aka Fault-Based testing, basati sull’iniezione di errori nel codice e sulla capacità di rilevarli.

Dopo aver mutato un sw (per esempio cambiando le condizioni degli if) viene sottoposto ad un controllo incrociato dei test, cioè se i casi di test passano con il sw base allora con la mutazione non devono passare (Mutant Killed), mentre se passano comunque non e’ un buon segno (Mutant Alive).

Possono esistere mutanti equivalenti, cioè sw mutati che sono sintatticamente diversi dall’sw originale ma semanticamente identici, e i test faranno sempre sopravvivere quei mutanti.

3.0 Logic for computer programming

Propositional Logic
analisi di relazioni tra proposizioni con connettori logici.

Declarative sentences
frasi basate sulle proposizioni che possono essere vere oppure false, non sono domande o ordini.

Atomic sentences
sono sentenze dichiarative minimali e non scomponibili

¬ negazione
∧ and
v or
→ conseguenza logica (≠ implicazione)
↔ iff
⊢ Natural deduction : insieme di regole che danno una conclusione

Le regole dicono che “se la parte sopra e’ verificata allora lo e’ quella sotto”, a dx c’e’ il nome della regola

And-Introduction [∧i]
$\frac{a\ b}{a \wedge b} \wedge i$
Se si ha una proposizione P e una proposizione Q, è possibile concludere P AND Q. Ecco un esempio:

P: Il sole è caldo.
Q: Il cielo è azzurro.
(P AND Q): Il sole è caldo e il cielo è azzurro.

And-Elimination [∧e]
$\frac{a \wedge b}{a} \wedge e1 \quad \frac{a \wedge b}{b} \wedge e2$
Se si ha una proposizione P AND Q, è possibile dedurre sia P che Q. Ecco un esempio:

(P AND Q): Il sole è caldo e il cielo è azzurro.
P: Il sole è caldo. (And-elimination)
Q: Il cielo è azzurro. (And-elimination)

Double Negation [¬¬]

$\frac{\neg\neg a}{a}\neg\neg e$

P: Marco non è scontento.

Non (non P): Non è vero che Marco non è scontento. (Double negation)

Implies-Elimination [$\rightarrow e$] (Modus Ponens)

$\frac{a \quad a\rightarrow b}{b}\rightarrow e$

P→Q: Se è venerdì, allora Marco va al cinema.

P: Oggi è venerdì.

Q: Quindi, Marco va al cinema. (Modus ponens)

Modus Tollens [MT]

$\frac{a->b \quad \neg b}{\neg b}MT$

P→Q: Se è venerdì, allora Marco va al cinema.

¬Q: Marco non è andato al cinema.

¬P: Quindi, oggi non è venerdì. (Modus tollens)

3.1 Design by contract

Contratto : accordo tra cliente e fornitore che
-lega le parti
-e’ scritto esplicitamente
-specifica obblighi e benefici delle parti
-non ha clausole nascoste
-e’ utile nel testing, come oracolo
-e’ utile per la verifica runtime del codice

Precondizioni [P]: obbligo per il cliente, ciò che viene richiesto

Postcondizioni [Q]: obbligo per il fornitore, ciò che viene restituito

“E’ meglio scrivere un metodo semplice che soddisfa un contratto ben definito (con pre/post cond. stringenti) piuttosto che un metodo che cerca di gestire tutte le situazioni possibili”

Invariante : regola vera dopo la creazione di un oggetto e per ogni operazione da li in avanti

Eccezioni : si sollevano con la violazione del contratto, e’ la manifestazione di un bug.

JML : Java Modeling Language, insieme di tool per dbc in Java
Si usa con //@ oppure /*@ @*/

Sintassi di JML :
   a) informal spec : //@ requires (* comment *);
   b) formal spec : //@ requires x > 0;
   c) Precondizioni = requires
   d) Postcondizioni = ensures
   e) \old per riferirmi ad un valore prima dell’invocazione
   f) \result per riferirmi al valore restituito
   g) //@ invariant : (sulla classe!) controlla il valore dell’oggetto da un preciso momento (solitamente la creazione) in avanti.
   h) < = = > : se e solo se
   i) = = > : implica
   j) < = = : ne segue che (implica al contrario)

k) < = ! = > : non se e solo se
l) \forall : quantificatore per ogni elemento
m) \exists : esistenza di almeno un elemento
n) \sum , \product, \min, \max : operatori generali
o) \num_of : contatore di occorrenze

I vari quantificatori seguono questa sintassi:
(quantificatore tipo variabile; range; espressione)

esempi:
(\forall TipoGenerico s; p; q)
(\min TipoGenerico x; p; q)
(\forall int i; 0<i; \result < 30)

Information Hiding:
in jml il privato/pubblico segue quello standard di java, si usano solo oggetti pubblici nelle specifiche, se serve usarne qualcuno privato si mette spec_pubblic nella dichiarazione cosi:

private /*@ spec_public @*/ int peso;

Null: si puo specificare la non nullita con:
/*@ not_null @*/

Assert : condizione verificata in un certo punto del codice, non deve avere side-effects, puo richiamare metodi puri. Ci sono sia dentro JML sia dentro Java normalmente.

In Java si possono usare in zone che non dovrebbero essere raggiunte per lanciare eccezioni (tipo assert false : “errore”), oppure per forzare un controllo in un if (tipo if(x<0) {..} else{ assert $x \geqslant 0$;...}).
In Java di default gli assert sono disattivati a runtime.

Metodi puri = query : metodi che non usano = o +-=, che non hanno side-effects, indicati con /*@ pure @*/.

Assignable : metodi quasi puri che specificano esplicitamente cosa puo essere modificato.

Exceptional Behaviour : si applica in caso di post-state inaspettato, sia in signals_only (dice solo che tipo di eccezioni un metodo puo lanciare) sia in signals (ha altre post-condizioni da rispettare in caso inaspettato)

Loop Invariant : condizione che deve essere vera in ogni iterazione del loop (sia iniziale sia corrente)

Sottotipazione: ogni sottotipo deve rispettare le specifiche del suo supertipo, al massimo puo' "espanderle".

Un metodo che fa override ha le preconditions implicate da quello base e le sue postconditions implicano quelle di quello base. Cioe' le preconditions possono solo diventare piu precise mentre le post piu lascive.

Also : metodo per annotare con piu di un contratto un metodo.

Software Corretto: se per ogni esecuzione che parte soddisfando le precondizioni allora termina in uno stato che soddisfa le post (Hoare Triple {P} A {Q})

Classe Corretta: e' corretta se sono corrette:
1) creazione : {Pcost} costruttore {inv ^ Qcost}
2) metodi : {Pmet ^ inv} metodo {Qcost ^ inv}

Tool vari:
- JMLC : compiler
- JMLRAC : interpreter
- ESCJAVA2 : analisi statica
- JMLUNIT : unit test
- JMLDOC : generatore html
- JML4C : boh
- OPENJML : boh

3.2 Program Verification (OpenJML)

Verifica ≠ Testing
Testing = prova la presenza di difetti.
Verifica = ne dimostra l'assenza.

Metodi:
- ProofBased : tipo dimostrazione matematica
- SemiAuto : forse serve intervento uomo
- PropertyOriented : si provano singole proprieta
- Sequential : si ignora la concorrenza
- Development : durante lo sviluppo

Vantaggi:
- Documentazione : piu facile da scrivere
- Time to market : minor tempo di debug
- Refactoring : facilita di riuso
- Certificazione : piu facile ad essere certificato per applicazioni speciali (militare, medico…)

Tools:
- Java Path Finder
- KeY : usato con JML tramite JMLKey

Struttura di test: OPENJML e ESC con Z3 come solver

Arithmetic Modes: posso aver diversi scenari numerici in base a cosa sto considerando:
- Java Mode: overflow e underflow sono silenti
- Safe Mode: lancia errori di verifica se potrebbe verificarsi overflow
- Math (BigInt) Mode: niente under/overflow, unbounded

Si imposta con @CodeJavaMath, @CodeSafeMath, @codeBigintMath

Correttezza dei Loop:
1) trovare un loop invariant
2) accertarsi che sia inizialmente valido
3) accertarsi che sia preservato durante il loop
4) se valido dopo il loop allora anche le postcond. di un codice corretto saranno valide

Come trovo il loop invariant?
deve implicare all'uscita la post condizione, quindi risponde alla domanda "perche all'uscita vale?"

Consistenza = Soundness: ogni cosa che puo essere provata e' corretta

Completezza = un sistema di dimostrazioni e' detto completo se ogni asserzione vera puo essere dimostrato (la logica proposizionale lo e')

Hoare:

1) il calcolo aritmetico e' incompleto (vedi logica aritmetica overflow)
2) halting problem, non essendo decidibile {p}S{false} non possiamo provarlo sempre

Correttezza totale: un sw oltre che essere Corretto termina anche, da provare con la dimostrazione del fatto che termina sempre (es. ciclo che decrementa fino a zero arrivera sempre a zero anche se parte da molto grande etc etc…)

3.3 Analisi Statica

Valutazione del sw senza eseguirlo cercando violazioni di pratiche ragionevoli o consigliate.

Uno strumento di as S analizza il sorgente di un programma P per determinare se soddisfa una proprieta Fi, tipo “P non dereferenzia un puntatore nullo” oppure “P chiude tutti i file che ha aperto”.

Possono esserci molti falsi positivi e falsi negativi. I tool non sono in grado di capire se il funzionamento e’ quello atteso (es. magari in una funzione prodotto viene fatta la somma, ma non solleverebbe problemi all’analisi statica)

E’ un concetto piu ampio di “cercare i bugs”, si controlla la struttura dei file di risorse xml, json ,yaml etc etc, la compilazione, il controllo ortografico etc etc…

P e’ positivo se viola la proprieta Fi, negativo se la soddisfa.

Errori di AS:

1) Se S e’ sound (pessimistico): se accetta P e’ perche sicuramente soddisfa Fi
    a) Non ha falsi negativi, accetta solo chi soddisfa Fi
    b) Ha falsi positivi, rifiuta quelli ok
2) Se S e’ completo (ottimistico): se P soddisfa Fi allora accetta
    a) Ha falsi negativi, puo accettare anche chi non soddisfa Fi
    b) Non ha falsi positivi, se soddisfa accetta sempre

Meglio falsi negativi o falsi positivi?
Tanti falsi positivi = tanti errori da controllare a mano = piu sicuro = piu dispendioso di tempo = prima o poi si ignoreranno i warning

Tools:

a) FindBugs:
    loop infiniti, override di equals e hashcode incompatibili, dereferenziazione di null pointer, confronti con null,

operazioni binarie strane, equals non avverabili, bad naming, exceptions ignorate, uso di this, sincronizzazioni… classificati in corretti, bad practice, performance, correttezza multithread, malicious.

b) MyPy : tool python
c) Stan4J

3.4 Inspection

Esaminare un sw cercando costrutti conosciuti che portano problemi. Usato soprattutto quando non ci sono altre tecniche o le altre tecniche non hanno abbastanza coverage.

Attuate tramite le Formal Code Review, cioe un processo dove il codice viene ispezionato.

Perche farle?
Piu punti di vista, i tool potrebbero non vedere alcuni problemi, alcuni problemi rilevati potrebbero essere falsi positivi, bisogna scegliere se il problema vale il tempo che serve per fixarlo.

Code Inspection:
formale, di solito guidati da un moderatore che non ha scritto il codice, richiedono di conoscere le specifiche prima, viene prodotto un report dettagliato.
usate checklist, esaminazioni paritarie (io controllo il tuo codice, tu il mio).

Checklist tipo:
Errori di design, errori di computazione, errori di confronto, errori di control flow, errori di subroutine, input/output, memoria…

Inspection ≠ Testing
Inspection : valuta anche le proprietà strane, tipo riusabilità, manutenibilità, scalabilità, robustezza e richieste di design.

Solitamente non ci sono addetti standard alle ispezioni, ma sono ruoli temporanei, cosi da non far sentir tutti giudicati.

4.0 Models and Modeling

Modello: utilita ingegneristica che permette di fare test e analisi su sistemi che ancora non esistono, tra cui analisi preliminari e soprattutto test automatizzati.

Un modello deve essere:

- Compatto = implica comprensibilita
- Predittivo = deve rappresentare in maniera accurata le caratteristiche cruciali del prodotto finale
- Multiplo = combinabile con altri modelli
- Semanticamente significativo = abbastanza espressivo per capire cosa succede negli scenari, e nei casi di errore
- Sufficientemente generale

Notazioni possibili:

- Pre/Post (state based) = best for data-oriented systems
- Transition-based = best for control-oriented system
- History-based
- Functional
- Statistical

Modello operazionale vs dichiarativo

- Operazionale = Imperativo = “Come faccio a far accadere X?”
- Dichiarativo = “Come mi accorgo che X e’ accaduto?”

4.1 Temporal Logic

Model Checking: modo completamente automatico di fare una verifica formale composto da 3 componenti all'interno di un Framework:

1) Modello = cio che descrive il sistema
2) Proprieta = cio che vogliamo verificare (in una formula temporale)
3) Verifica = cioe chiedere al model checker se il modello implica la proprieta

Logiche temporali:

- Linear Time logics (LTL) = il tempo e' un insieme di cammini, cioe una sequenza di istanti di tempo
- Branching Time Logics (CTL) = il tempo e' un albero con istante corrente come radice

Operatori temporali LTL:

- X = neXt state
- F = some Future state
- G = Globally in all future state
- U = Until
- W = Release
- R = Weak-Until

Ordine degli operatori

1) Prima il not, XFG
2) Poi URW
3) Poi $\wedge$ *e* $\vee$
4) Poi $\rightarrow$

Transition System: tipologia di modellazione che usiamo per i sistemi che interessano a noi, formato da:

- State Set S
- Stato iniziale S0
- Relazione di transizione $\rightarrow$ che lega un qualsiasi s ad un s' (impedisce i deadlock per design)
- Labelling function = $L:S \rightarrow P(AP)$

Formalmente indicati come $M = (S, S0, \rightarrow, L)$, possono essere facilmente rappresentati da grafi.

P(AP) e’ il powerset (insieme delle parti) di proposizioni atomiche AP.

Un path e’ un infinita sequenza di stati scritto come s1 → s2 → s3…

DALLA SLIDE 15 CONVIENE STUDIARE SULLE SLIDES

4.3 Model Checking

SLIDES

4.5 (4.2.1) Abstract State Machines ASMETA

5.1 Model Based Testing

Steps:

1) Modello astratto del sut: astratto = piccolo e semplice, ma che implica lo stesso funzionamento di quello reale.
2) Generazione test astratti: criteri per la copertura del modello
3) Concretizzazione dei testi astratti in test fattibili: spesso fatto a mano, o con dei wrapper automatici
4) Esecuzione dei test e collezione dei verdetti
5) Analisi dei risultati

Modellazione:

- focus sul sut
- inclusione operazioni da testare
- data field complessi vengono rimpiazzati da strutture semplici di supporto
- tipi di modellazione:
    - Solo input = Partition/Combinatorial Testing: solo gli input critici del sistema
    - Input e comportamento: basato su FSM

5.2.1 Partition / Combinatorial Testing

Input Testing:
testing basato sulle interfacce, tipo blackbox, senza nessuna informazione riguardante il sistema sottostante.

- semplice da applicare
- si puo usare a diversi livelli di testing

Partition Testing: (vecchio come la pangea)
si sceglie una partizione del dominio di input del programma, dividendo il dominio in regioni, cosi sostituendo un dominio (anche infinito) in un numero enumerabile di regioni.

IDM: Input Domain Model

Partitioning Domains:
Dominio D
Schema di partizioni q di D
lo schema definisce un insieme di blocchi Bq = B1, B2….
La partizione deve rispettare 2 proprieta:

1) no overlap (B disjoint)
2) completezza (la somma dei B da D)

ISP Coverage:

- All combination (ACoC) = piu forte = N-Wise con N #parametri
- T-Wise
- Pair-Wise : copre ogni possibile coppia
- Each Choice = piu debole = 1-Wise

Metodi greedy per generare casi combinatoriali:
Basati sui parametri → IPO = InParameterOrder
Basati sui testi

IPO costruisce in passi incrementali, partendo dal pairwise scegliendo due parametri che voglio (in genere i piu grandi per primo), poi aggiungo a dx gli altri parametri (horizontal growth, o a caso o cerco di minimizzare i pair inutili), controllo se ho coperto tutti i pair tra tutti i parametri, e se mancano li aggiungo sotto (vertical growth)

5.2 Testing basato su specifiche

seleziono i casi di test usando la specifica, e posso usarla come oracolo, perche' mi dice come si dovrebbe comportare un sistema se fosse corretto, cioe' se applico i casi di test alla specifica e al mio sw devo avere gli stessi risultati (se non lo sono e' il sw ad essere sbagliato, la specifica no perche e' oracolo e quindi verita) = conformance tesing

FSM, usiamo la macchina di Mealy che produce un output per ciascuna transizione, cosi formata (S,I,O,delta,lamda) con:
S insieme finito di stati, I insieme finito di eventi di input, O insieme finito di eventi di output, delta funzione di transizione (SxI → S) cioe funzione che dato uno stato e un input so in che stato andare, lambda funzione di output (SxI → O) cioe funzione che per ogni stato e input mi da l'output di quella transizione.

Sugli archi del grafico si indicano le i/o cioe le coppie eventoInput e eventoOutput.

No due transizioni con lo stesso input sullo stesso stato (NON SAREBBE UNA MACCHINA DETERMINISTICA)

Come fare conformance testing con FSM?
-creazione di una FSM basata sulla specifica di cui conosco tutto, chiamata S
-implementazione I del mio sw sotto forma di FSM di tipo black box in modo da avere due FSM da confrontare direttamente con una sequenza di test

Come trovo i casi di test?
analizzando la fsm della specifica con un algoritmo, poi li applico sia sulla macchina S della specifica sia sulla macchina I del software e ne confronto i risultati

Sotto alcune ipotesi (forti) possiamo trovare dei test ideali:
-S e I sono deterministiche (non dipendono dal tempo) e inizializzate (hanno stato iniziale)
-S e I sono completamente specificate (per ogni stato uso tutti gli input e so dove vanno)

-S e’ fortemente connessa (posso raggiungere qualsiasi stato, anche con passi multipli, da qualsiasi altro stato)
-S e’ minimizzata (non esiste FSM piu piccola con lo stesso comportamento)
-I non cambia durante i test
-I ha lo stesso alfabeto di S (stessi input e output)
-I non ha piu stati di S (cioe i difetti non aumentano gli stati)
-Errori considerati:
    -Output error: vado nello stato giusto con l’input giusto ma sbaglio l’output
    -Transfer error: vado nello stato sbagliato con l’input e output giusto
-Errori non considerati:
    -stati extra o stati in meno

Messaggi speciali: Reset (mando a stato iniziale), Status (chiedo in cui stato e’).

Nei nostri test set T genereremo una sola test sequence ts (cioe una sequenza di input con i suoi output) ma molto lunga.

Come genero i casi?
-State cover method = copertura stati
-Transition tour TT

Copertura stati:
in ogni T ogni stato viene visitato almeno 1 volta
TT:
in ogni T ogni transizione viene visitata almeno 1 volta

Metodo 1: (stati + status message)
Copertura degli stati e lettura dello status message per controllare che il comportamento sia corretto
Ho una sequenza di input e una sequenza formata da output (delle transizioni) e degli stati (con status message)
puo trovare sia errori di output sia di transfer, ma non lo garantisce

Metodo 2: (transition tour)
sequenza di input che passa in tutte le transizioni almeno una volta e poi torna a casa.

il tour piu veloce e’ quello Euleriano.
garantisce scoperta di difetti di output ma non di transfer

Metodo 3: (TT + status message)
garantisce la scoperta sia dei difetti di output sia quelli di transfer

5.4 Making tests executable

Concretizzazione:
“tradurre” i test astratti nel linguaggio del sw effettivo

Modelliamo solo cio che serve, usiamo input e output necessari, semplifichiamo i dati complessi, il SUT e’ gia inizializzato, ci basta modellarne una parte del SUT

online testing:
applico immediatamente i casi di test al sistema che funziona, in tempo reale. (richiede di mettere insieme a funzionare sia il generatore di casi di test sia il sw)

offline testing:
generazione e esecuzione dei casi di test sono separati, completamente indipendenti

Colmare il gap semantico tra test cases e sut:
-non traduco ma uso un adapter (o wrapper) (online testing)
-traduco ogni cosa in dei test script (offline testing)
-traduco un po e poi adatto

ATGT: generatore automatico di scenari in avalla per asmeta

Specifica = asmeta
Validazione = model checker, avalla
Generazione = atgt o a mano

GraphWalker:
macchina a stati con label su transizioni. due tipi di elementi, stati(vertici) e transizioni(edge).
vertici=stati=verification=assertion
edge=transizioni=azioni

Yakindu:
modello come macchine di stato (statecharts) tipo uml. definisco interfacce dei moduli per cosa poso modificare o quali event posso fare
lo posso validare lanciando la simulazione.
posso convertire da yakindu a junit.

CTWedge:
plugin di eclipse che genera test suite per il combinatorial testing.
specifico un modello con dei parametri e delle constraints sui parametri.

Input Domain Modeling:
-Interface Based: divido in partizioni basate su valori significativi per i vari input senza considerare l’applicazione specifica (es ho interi scelgo a caso)
-Functionality Based: scelgo le partizioni in base al risultato che mi aspetto del metodo che devo testare