Testing & Verifica del Software

Appunti

Galizzi Francesco

30 maggio 2014

Indice

1 Introduzione

Obiettivi del testing e della verifica del software:

- valutare le **qualità** del programma;
- rendere possibile il miglioramento del software trovandone i **difetti.**

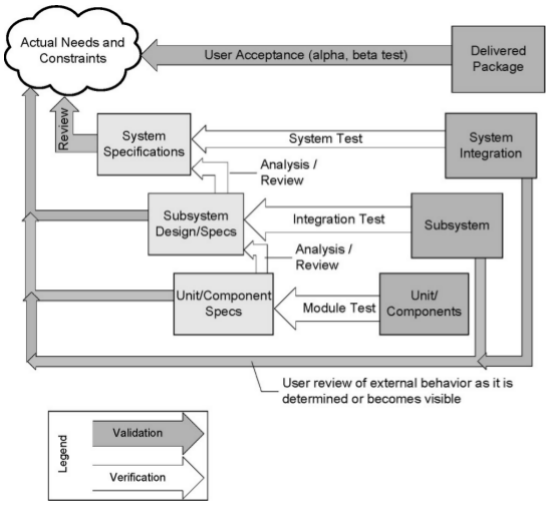
1.1 Concetti generali

1.1.1 Validazione e verifica

specifica una dichiarazione (documento) riguardo a una particolare proposta di soluzione ad un problema;

validazione verificare che il software che si sta costruendo risponda alle reali richieste dell'utente; effettuata comparando le richieste dell'utente con la specifica dei requisiti;

verifica verificare che il software rispetti il documento di specifica dei requisiti; effettuata comparando la specifica dei requisiti con il software.



Entrambi sono un confronto tra due o più **artefatti**, cioè prodotti del processo di sviluppo del software.

Va sempre comunque controllata anche la consistenza interna dei documenti (cioè che facciano affermazioni coerenti tra loro) e che non contengano ambiguità, insieme ad altri eventuali vincoli imposti dall'aderenza ad un determinato standard o simili.

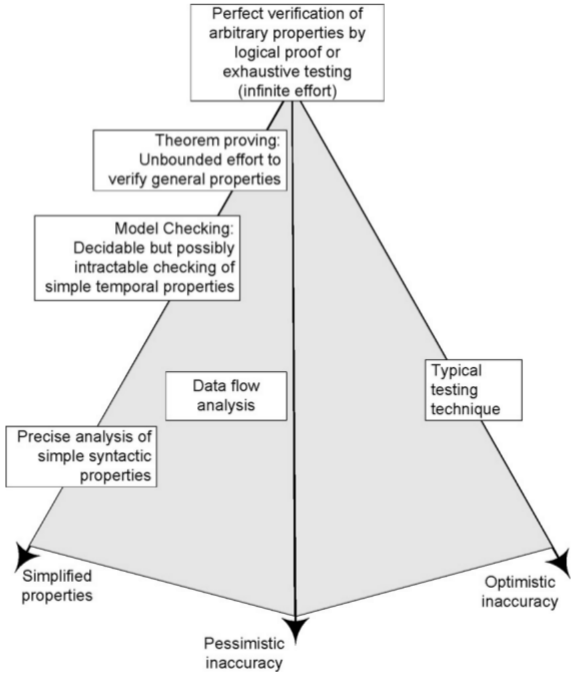
1.1.2 Gradi di libertà

Un ***problema indecidibile*** è un problema decisionale (ciò la cui risposta può essere solo "sì" o "no") per il quale è impossibile costruire un singolo algoritmo che porti sempre alla risposta corretta.

Il ***problema della terminazione*** (*halting problem*, tradotto anche con *problema dell'arresto* o *problema della fermata*) chiede se sia sempre possibile, descritto un algoritmo e un determinato input finito, stabilire se l'algoritmo in questione termini o continui la sua esecuzione all'infinito.
Si può dimostrare che il problema della terminazione è un problema indecidibile, e che ogni proprietà interessante riguardo il comportamento di un programma lo "contiene". Di conseguenza non può essere costruito nessun algoritmo che sia in grado di dare informazioni sulle proprietà di qualsiasi programma gli venga sottoposto.

L'alternativa a questo problema potrebbe essere il ***testing esaustivo***, cioè eseguire e controllare ogni possibile comportamento del programma ("dimostrazione per casi" che il programma si comporta nella maniera corretta). Il problema di questo approccio è ovviamente il tempo necessario a portare a termine il test. Esso è quindi *impraticabile*. Inoltre è anche *impossibile*: applicando un test, non riesco a distinguere se un programma P si è bloccato o devo aspettare ancora.

Approssimazioni



inaccuratezza ottimistica potrebbe accettare programmi che non possiedono tutte le proprietà desiderate (es.: testing);

inaccuratezza pessimistica non è garantita l'accettazione di un programma che possiede tutte le proprietà desiderate (es.: tecniche di analisi automatiche di programmi);

semplificazione/astrazione vogliamo verificare una proprietà S, ma non possiamo accettare inaccuratezza ottimistica né farne un'analisi precisa perché troppo difficile. Al posto di verificare S verifichiamo S', proprietà più semplice che è condizione sufficiente ma noon necessaria per S, e richiediamo che sia soddisfatta.

Terminologia

safe (sicuro) un'analisi sicura non prevede inaccuratezze ottimistiche (accetta solo programmi corretti);

sound (corretto) l'analisi di un programma P rispetto ad una formula F è corretta se accetta P solo se il programma soddisfa la formula (potrebbero esserci programmi corretti che non vengono accettati; il testing non è "sound");

complete (completo) l'analisi di un programma P rispetto ad una formula F è completa se il accetta sempre P quando questo soddisfa la formula (potrebbe accettare anche programmi non corretti).

Tipi di software Generalmente tutte le tecniche di testing e analisi sono applicabili a qualsiasi tipo di programma. Ci sono però delle categorie di software che richiedono che vengano verificate anche particolari proprietà, o per cui l'importanza relativa di una proprietà con l'altra può cambiare, o che richiedono particolari vincoli sulle tecniche applicabili:

- software real-time e safety-critical;
- applicazioni distribuite e/o concorrenti;
- interfacce grafiche;
- ...

1.2 Processo di test e di verifica

1.2.1 Cos'è il testing

Definizione dell'IEE SE Body of knowledge Il ***testing*** è un'attività svolta al fine di valutare le qualità di un prodotto, e di migliorarlo, identificandone difetti e problemi.

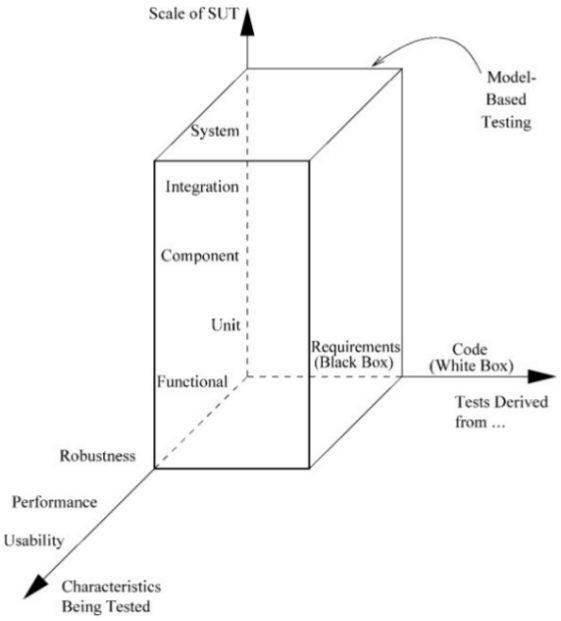
Il testing del software consiste nella verifica ***dinamica*** del comportamento di un programma per un set ***finito*** *di casi di test*, opportunamente ***selezionati*** dal solitamente infinito dominio di esecuzione, rispetto al comportamento ***previsto***.

dinamico richiede che il software venga eseguito;

finito solo un sottoinsieme di possibili input;

selezionato selezionato secondo qualche criterio;

previsto dev'esserci un modo per verificare la correttezza del software.



1.2.2 Processi di test classici (program-based)

Punti principali

progettazione dei casi di test: i casi di test devono essere progettati partendo dai requisiti di sistema e prendendo in considerazione gli obiettivi e le politiche d'alto livello . Ogni caso di test è definito da un contesto, uno scenario e qualche criterio di superamento/fallimento;

esecuzione del test e analisi dei risultati: i casi di test devono essere eseguiti sul SUT (System Under Test). I risultati del test sono poi analizzati per determinare la causa dell'eventuale fallimento di ogni test;

verifica della copertura dei requisiti: per gestire le qualità del processo di test (e quindi quelle del prodotto), occorre misurare la copertura dei requisiti da parte del test set. Solitamente ciò viene svolto grazie ad una matrice di tracciabilità tra i casi di test e i requisiti.

Testing manuale

- Progettazione dei test svolta a mano;
- esecuzione dei test fatta a mano dai tester;
- risultato del test da osservazione.

Molto semplice da implementare. Il costo è basso per pochi casi di test, ma cresce rapidamente con l'aumentare dei casi di test.

Testing capture and replay

- Progettazione ed esecuzione dei test fatte a mano;
- l'esecuzione viene registrata;
- quando un nuovo rilascio del SUT dev'essere testato, il tool capture/replay tenta la riesecuzione di tutti i test registrati e riporta quali hanno fallito.

Molto semplice rieseguire i casi di test. Non è robusto a cambiamenti del software.

Testing script-based

- I test sono script scritti dal tester (uomo);
- i test possono essere eseguiti automaticamente.

Molto semplice rieseguire casi di test. La manutenzione dei casi di test può essere costosa.

Tipici passi di uno script di test: inizializza SUT, lo porta nello stato richiesto, crea i valori di input e li passa al SUT, registra la risposta e la confronta con quella prevista, decreta il superamento/fallimento del test.

Testing program-based

- I test sono programmi scritti dal tester (uomo);
- l'esecuzione e la verifica del superamento sono fatti automaticamente eseguendo il test.

Molto semplice rieseguire casi di test. La scrittura e la manutenzione dei casi di test possono essere dispendiose in termini di tempo.

1.2.3 Processo di test model-based

Passi necessari:

- creare un modello del SUT e/o del suo ambiente;
- generare casi di test astratti dal modello;
- concretizzare i casi di test per renderne possibile l'esecuzione;
- eseguire i test sul SUT e decretarne l'esito;
- analizzare il risultato dei test.

1.2.4 Verifica formale

Tecnica che ***costruisce*** una prova ***matematica*** di ***consistenza*** tra una qualche rappresentazione formale del programma o progetto e una specifica formale (delle proprietà che il programma o il progetto devono avere).

costruisce è spesso un'attività umana

matematica basata su regole matematiche e logica

consistenza tra due artefatti

Verifica del programma Può provare che un programma è corretto (cosa che il testing non può fare); occorre dare una descrizione *formale* della specifica di un programma, dopodiché si cerca una prova che dimostri che il programma soddisfa le specifiche.

Verifica a runtime Controllo a runtime se il programma si comporta come previsto.

Verifica tramite modello formale Il progetto dev'essere convertito in un formato "verificabile" (es.: FSM). La proprietà da verificare dev'essere espressa in modo formale.

Il metodo più diffuso per la verifica formale è il **model checking**: dato il modello di un sistema, controlla esaustivamente e in modo automatico se il modello rispetta le specifiche.

2 Testing

2.1 Ruolo del testing

Il test del software è particolarmente arduo in quanto il software ha un comportamento molto discontinuo, quindi la selezione dei "punti" in cui effettuare il test è molto critica.

Il testing è efficace a trovare bug ma è inadeguato a provare l'assenza di bug; non può quindi sostituire una buona pratica di progettazione e implementazione. Occorre comunque ricordare che non tutti i software devono necessariamente essere corretti al 100%; lo dovrebbero essere almeno quelli usati in sistemi critici.

Il testing dovrebbe:

- essere automatizzato, in tutte le fasi possibili (scrittura, esecuzione, raccolta dell'output e analisi della correttezza);
- riguardare ogni fase dello sviluppo (requisiti, prototipi, codice, ...);
- essere esteso a tutti i componenti del sistema;
- essere pianificato (***test plan***);
- seguire determinati standard e metodologie ogniqualvolta ciò sia possibile.

Difetti trovabili col testing I difetti più critici sono di meno e più difficili da scoprire (anche se non sempre si ha questa distribuzione). Il testing tende a scoprire i difetti con probabilità proporzionale alla loro densità. Tecniche (più costose) di analisi statica invece cercano di scoprire difetti con probabilità proporzionale alla loro criticità.

Nell'eseguire il testing occorre considerare quanti questo costi, e confrontarlo con quanto costi avere un software con difetti. È "inutile" spendere troppo tempo a cercare difetti che non provocano danno; per questo motivo il sw non critico viene spesso rilasciato con ancora dei difetti.

2.1.1 Terminologia

mafunzionamento (guasto o failure) funzionamento non corretto del programma, legato al comportamento che si osserva durante l'esecuzione;

difetto (anomalia, fault o bug) elemento del programma sorgente non corrispondente alle aspettative; riguarda quindi la parte statica;

errore fattore (umano) che causa una deviazione tra il software prodotto e il programma ideale;

Quando il programmatore commette un ***errore***, il programma avrà uno o più ***difetti*** che possono generare ***malfunzionamenti***.

testing eseguire il programma con dei casi di test e analizzare i risultati per trovare i difetti (bug);

debugging correggere i difetti ed eventualmente scoprire gli errori.

2.1.2 Scopi del testing

- mettere in evidenza i difetti (bug) mediante i malfunzionamenti, per scoprire eventuali errori;
- poter valutare l'***affidabilità*** di un sw (reliability) e fornire confidenza (test di accettazione):
  - dell'affidabilità del prodotto e della (probabile) correttezza;
  - dell'aver rilevato l'assenza di particolari tipi di malfunzionamenti;
  - che i comportamenti più critici o più frequenti non causino malfunzionamenti.

2.1.3 Livelli di granularità

Test di:

accettazione il comportamento del software è confrontato con i requisiti dell'utente finale;

conformità il comportamento del software (tutto) è confrontato con le specifiche dei requisiti;

unità test del comportamento delle singole unità; nella programmazione OO la singola unità è una classe, di cui vengono testati i metodi; per ogni metodo testato (***test unit***) possono essere introdotti:

test driver metodo che chiama il test unit con opportuni parametri;

test stub metodo che sostituisce eventuali metodi usati dal test unit (per testare in modo isolato e controllato, poco usato in quanto oneroso);

integrazione controllo sul modo di cooperazione delle unità (come previsto dal progetto); strategie:

top-down richiede lo sviluppo di stub;

bottom-up richiede lo sviluppo di driver;

big-bang ;

sistema controlla il comportamento dell'intero sistema (hw + sw) come monolitico;

regressione test del comportamento di release successive; viene eseguito per verificare di aver eliminato i difetti segnalati ed essere sicuri di non aver introdotto nuovi difetti, riusando quanto più possibile i vecchi casi di test, ed evitando di dover ritestare le parti non modificate.

2.2 Tipi di testing

Testing white-box (structural o program-based) È basato sulla struttura interna del programma.

- deriva i casi di test dal programma;
- controlla e osserva i programmi durante l'esecuzione.
- in genere analizza "quanto programma è stato eseguito" o coperto;
- non garantisce che il programma faccia quello effettivamente richiesto.

Passi principali per la costruzione dei test:

1. esamina la struttura del programma;
2. trova i casi di test (cioè input) che soddisfano un certo criterio di copertura;

3. applica gli input (uno alla volta) e osserva l'output del programma;
4. controlla che non si verifichino errori e che gli output siano quelli attesi.

Questo approccio non riesce a trovare errori di omissione, e non fornisce ***oracoli***[1] per i test.

Testing black-box (functional o specification-based) Ignora la struttura del programma e considera solamente i suoi requisiti.

- deriva i casi di test dai requisiti;
- controlla e osserva il programma solo attraverso la sua interfaccia esterna (input/output);
- in genere misura quanti input/output sono stati utilizzati;
- cerca di scoprire il maggior numero di difetti e di escludere quelli più critici.

Passi principali per la costruzione dei test:

1. esamina la specifica dei requisiti del programma;
2. seleziona un insieme di casi di test che soddisfano qualche criterio;
3. applica questi input alla specifica e colleziona gli output A (attesi);
4. applica gli stessi input al programma e colleziona gli output B (osservati);
5. confronta gli output A con gli output B e controlla che siano uguali.

In questo caso la specifica funziona automaticamente da oracolo.

Questo approccio richiede un maggiore sforzo rispetto a quello program-based, e non sempre è praticabile in quanto richiede che la specifica del programma sia disponibile e che sia scritta in modo formale (sia per generare i casi di test che per valutare gli output attesi).

Testing grey-box Mix tra i precedenti approcci.

2.3 Fondamenti teorici

2.3.1 Definizioni base del testing

Un ***programma*** P è una funzione da un dominio D ad un codominio R: $P : D \to R$ (può essere non definita per qualche $d \in D$.

Predicato OK:

- $OK(P, d)$ con $d \in D$ se P è corretto per l'input d, cioè se produce $P(d)$ corretto;
- $OK(P)$ se P è corretto, cioè se vale $OK(P, d)\ \forall\, d \in D$.

Ridefiniamo:

failure (malfunzionamento) quando eseguo P con d e non ottengo $P(d)$;

fault (difetto o bug) se l'implementazione P' di P è diversa da P, il difetto è "ciò che li differenzia";

errore motivo del difetto;

[1] ***oracolo***: modo per stabilire se il test ha evidenziato un malfunzionamento oppure no

caso di test elemento di D;

test set (o test suite) sottoinsieme (finito) di D;

test negativo se non evidenzia malfunzionamenti in P;

test positivo se evidenzia malfunzionamenti in P.

Un test set T è detto ***ideale*** se $OK(P,T) \Rightarrow OK(P)$, cioè se testando il programma con T non si osservano malfunzionamenti, allora P è corretto.
Un test ideale è quello esaustivo, cioè con $T = D$, ma come già discusso è impraticabile.

Un ***criterio di test*** o di ***adeguatezza*** C è una funzione che per un programma P e la sua specifica S, dato un test set T, $C_{P,S}(T)$ è vero se e solo se T è adeguato a trovare ogni difetto in P rispetto S secondo il criterio C.
In generale $C : P \times S \times T \rightarrow, \{true, false\}$. Nel testing program-based C non dipende da S, mentre nel testing specification-based C non dipende da P.

Un criterio di test può essere anche inteso come generatore di test set dato P e S: $C : P \times S \rightarrow T$ tale che $C(P,S,T) = true$.

Un criterio di test è ***affidabile*** se per ogni coppia di test set T_1 e T_2 adeguati secondo il criterio C, se T_1 individua un malfunzionamento, allora anche T_2 lo individua, e viceversa. Da notare che non è detto che un criterio affidabile riesca a scoprire malfunzionamenti.

Un criterio di test C è ***valido*** se, qualora il programma P non sia corretto, esiste almeno un test set T che soddisfa C che è in grado di evidenziarne il/i malfunzionamento/i.

Teorema di Goodenough e Gerhart Siano C un criterio di test e P un programma.
Se C è *affidabile* per P e C è *valido* per P allora C è *ideale*.
Sia ora T un test set chesoddisfa C; allora T è *ideale*, e se vale $OK(P,T)$ allora vale $OK(P)$.

Teorema di Howden Non esiste un algoritmo che dato un programma P generi un test ideale finito.

Enunciato di Djiskra Il test di un programma può rilevare la presenza di malfunzionamenti ma mai dimostrarne l'assenza.

Utilità dei criteri di test Sebbene non si possano definire criteri di test ideali, posso definire:

criteri empirici criteri che non sono né validi né affidabili ma che si sono dimostrati utili nella pratica;

stopping rule danno confidenza sul fatto che si sia "testato abbastanza";

test adequacy da una misura di copertura del programma (o delle specifiche).

Teorema di Weyuker Trovare un insieme di input che esegua una particolare istruzione, un particolare cammino o tutte le istruzioni è un ***problema non computabile***, cioè non esiste algoritmo per risolvere questo problema per ogni programma.

Esistono però algoritmi e tecniche che sono in grado di risolverlo per molti programmi.

Quindi l'adeguatezza di un test set non è dimostrabile; possiamo però usare delle ***regole di design*** per mettere in evidenza l'inadeguatezza di un test set. In pratica sostituiamo ai criteri d'adeguatezza dei vincoli più deboli, le regole di design. un test set che rispetti questi vincoli non è quindi *ideale*, ma è quantomeno utile.

Definiamo quindi:

caso di test insieme di input, condizioni d'esecuzione e criterio di superamento/fallimento;

specifica di test condizione che dev'essere rispettata da uno o più casi di test;

vincolo di test specifica di caso di test parziale;

test set set di casi di test;

test l'attività di esecuzione di un caso di test e conseguente valutazione dei risultati;

criterio di adeguatezza predicato che è vero o falso per una coppia $\langle programma, test\,set \rangle$.

Un test set T *soddisfa* un criterio di adeguatezza C se tutti i casi di test in T hanno successo e se ogni vincolo di test in C è soddisfatto da almeno un caso di test in T.
Esistono criteri che non possono essere soddisfatti per un dato programma; in questo caso posso escludere i vincoli insoddisfacibili dal criterio di test oppure posso misurare quanto quel vincolo è soddisfatto (es.: in %).

Implicazione Un criterio di test A ***implica*** un criterio di test B se, per ogni programma P, tutti i test set che soddisfano A rispetto a P soddisfano anche B rispetto a P.

2.4 Program-based testing

Nel testing basato sulla struttura dei programmi:

- i criteri di test sono definiti considerando il solo codice sorgente;
- il sorgente viene usato per:
  - generare i casi di test;
  - decidere quando si ha testato abbastanza.

I criteri di test strutturali sono definiti considerando la "copertura" del codice. Per ***copertura*** di un programma si intende la parte del programma che viene eseguita dai casi di test. In particolare, intendiamo la struttura del codice come flusso di controllo del programma; quindi la copertura sarà relativa al flusso di controllo.

Il flusso di controllo di un programma può essere rappresentato tramite grafo:

- rappresenta qualsiasi esecuzione possibile;
- ogni istruzione è un nodo del grafo;
- ogni istruzione è collegata alla sua successiva mediante una freccia.

Simbologia:

- nodo circolare: istruzione di assegnamento, lettura, scrittura o return;
- nodo romboidale: istruzione condizionale e/o decisionale; ha 2 uscite.

2.4.1 Copertura delle istruzioni (statement coverage)

Un test set T è adeguato per testare un programma P secondo il **criterio di copertura delle istruzioni** se per ogni istruzione (statement) s di P esiste un caso di test in T che esegue s.

In pratica ogni istruzione viene eseguita almeno una volta.

Capacità di rilevamento dei difetti:

- istruzioni (sempre) errate vengono individuate;
- gli errori nelle decisioni non è detto che vengano trovati.

È un criterio debole.

Utilizzo come misura di copertura (delle istruzioni): $C_{statement} = \frac{\#\, statement\, eseguiti}{\#\, statement\, eseguibili}$.

2.4.2 Copertura degli archi (branch coverage)

Un test set T soddisfa il ***criterio di copertura degli archi*** di P se e solo se ogni arco (branch) del grafo di P è percorso almeno una volta.

Il branch coverage implica lo statement coverage.

Utilizzo come misura di copertura (degli archi): $C_{branch} = \frac{\#\, branch\, eseguiti}{\#\, branch\, eseguibili}$.

Per trovare i valori per una determinata copertura:

- metti true alla fine del cammino che vuoi coprire;
- procedi all'indietro sul cammino, e:
  - se c'è un assegnamento, sostituisci la variabile assegnata con il valore che è stato assegnato;
  - se c'è una ramo "T" in una decisione aggiungi la decisione alla condizione;
  - se c'è un "F", aggiungi la negazione.

Potrebbe anche non essere risolvibile algoritmicamente, o addirittura non essere soddisfacibile (es.: $x > 0 \wedge x < 0$).

2.4.3 Copertura delle decisioni

Un test set T è adeguato per testare un programma P secondo il ***criterio di copertura delle decisioni*** se per ogni decisione di P esiste:

- un caso di test in T in cui la decisione è presa;
- un caso di test in T in cui la decisione non è presa.

Una ***decisione*** è un predicato (espressione booleana) guardia di un'istruzione condizionale (if) o di una iterativa (while, for, ...).

Copre quindi ogni decisione e la sua negazione. È equivalente al branch coverage.

2.4.4 Copertura delle condizioni

Un test set T è adeguato per testare un programma P secondo il ***criterio di copertura delle condizioni*** se per ogni condizione di P esiste:

- un caso di test in T in cui la condizione è vera;
- un caso di test in T in cui la condizione è falsa.

Una ***condizione*** è un'espressione booleana atomica (cioè non divisibile in altre espressioni più semplici) che appare in una decisione.

Copre quindi ogni condizione e la sua negazione.

2.4.5 Copertura delle decisioni e delle condizioni

Il ***criterio di copertura delle decisioni e delle condizioni*** richiede la copertura sia di tutte le decisioni che di tutte le condizioni.

Notare quindi che i due criteri non si implicano a vicenda.

Short circuit evaluation

I compilatori spesso usano per efficienza la valutazione a corto circuito per le espressioni booleane:

- $a \,\&\&\, b$: se a è falso non valuto b;
- $a \,||\, b$: se a è vero non valuto b.

Spesso esistono operatori che evitano ciò (& e | al posto di && e ||).

2.4.6 Multiple Condition Coverage (MCC)

Un test set soddisfa ***MCC*** se testa ogni possibile combinazione dei valori di verità delle condizioni atomiche in ogni decisione.

Quindi con n condizioni in una decisione si avranno 2^n combinazioni da soddisfare (T e F per ogni condizione). Se n è grande MCC non è fattibile.

Si rappresenta in genere con una tabella. Il numero di casi di test può essere ridotto sfruttando la short circuit evaluation.

2.4.7 Modified Condition/Decision Coverage (MCDC)

Il test set deve essere preso in modo che ogni condizione all'interno di una decisione deve far variare in modo indipendente il valore finale della decisione.

2.4.8 Copertura dei percorsi (path coverage)

Un test set T soddisfa il ***criterio di copertura dei percorsi*** di P se e solo se ogni percorso (path) del grafo di P è percorso almeno una volta.

Un ***percorso*** è una tra tutte le possibili combinazioni delle decisioni del programma.

Utilizzo come misura di copertura (dei percorsi): $C_{path} = \frac{\#\, path\, eseguiti}{\#\, path\, eseguibili}$.

Da notare che il numero di path in un programma che contiene cicli potrebbe essere infinito. Perchè il criterio sia fattibile si può dividere l'insieme infinito di path in un numero finito di "classi di path". A questo punto il criterio può essere applicato limitando il numero di cicli attraversati, la lunghezza del path che dev'essere attraversato o le dipendenze tra i path selezionati.

2.4.9 Boundary interior path testing

- Raggruppa tra loro tutti i path che differiscono solo per il sotto-path che seguono quando ripetono il corpo di un ciclo;
- segue ogni path nel flusso di controllo fino al primo nodo che si ripete (cioè per cui si è già passati).

L'insieme di path dalla radice dell'albero a ogni foglia è l'insieme richiesto di path per il ***criterio di test "boundary/interior"***.

È da tenere in considerazione che il numero di path potrebbe comunque crescere esponenzialmente.

2.4.10 Loop boundary adequacy

Un test set soddisfa il ***criterio di test "loop boundary"*** se per ogni ciclo:

- in almeno un caso di test il ciclo viene iterato 0 volte;
- in almeno un caso di test il ciclo viene iterato 1 volta;
- in almeno un caso di test il ciclo viene iterato più di 1 volta.

2.4.11 Linear Code Sequence And Jumps (LCSAJ)

Un ***LCSAJ*** è un sotto-path nel grafo del flusso di controllo (di un programma) che inizia e finisce con una decisione (da una "ramificazione" ad un'altra).

Definiamo quindi TER_{n+2} la copertura di n consecutivi LCSAJ (TER_1 è lo statement coverage, mentre TER_2è il branch coverage).

In pratica è una generalizzazione della copertura in base agli archi, in base a quante decisioni consecutive si vuole raggruppare (rendendole dipendenti una dall'altra).

2.4.12 Copertura ciclomatica

La ***complessità ciclomatic***a è il numero di path indipendenti nel CFG (Control Flow Graph, grafo di flusso di controllo).

Chiamiamo a il numero degli archi del CFG, n il numero dei nodi; per un CFG la complessità ciclomatica può essere calcolata come $a - n + 2$.

Il ***criterio di copertura ciclomatica*** conta il numero di path indipendenti che sono stati eseguiti, relativamente alla complessità ciclomatica.

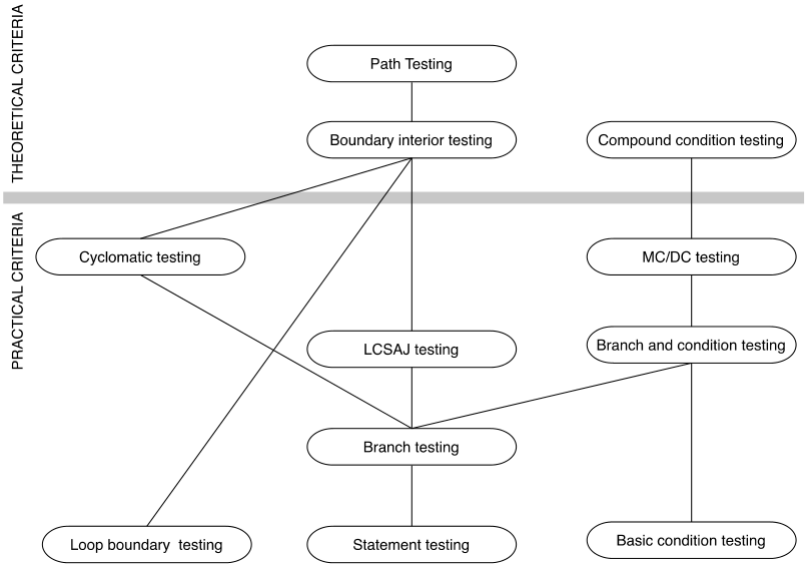
2.4.13 Procedure-call testing

Una volta che il test di unità (eseguito tramite una delle metodologie appena illustrate) è andato a buon fine, gli unici difetti che ancora potrebbero esserci nel codice sono da cercare nella comunicazione tra unità, e quindi nelle chiamate tra un/a prodedura/funzione/metodo ad un'altro/a.

Procedure entry and exit testing Le procedure possono avere più "punti" d'ingresso e d'uscita; occorre testarli tutti.

Call coverage Lo stesso punto d'entrata può essere usato in chiamate diverse (ignorando gli altri).

Relazioni di implicazione tra criteri di test program-based



A volte i criteri non sono soddisfacibili; potrebbero richiedere l'esecuzione di:

- istruzioni non eseguibili come risultato di una programmazione difensiva o di un riuso di codice (senza eliminare le parti non necessarie);
- condizioni non soddisfacibili come risultato di decisioni interdipendenti;
- path non percorribili come risultato di decisioni interdipendenti.

Una grande quantità di codice non eseguibile potrebbe indicare seri problemi di manutenibilità; una quantità contenuta è invece comune anche nei sistemi ben progettati e ben mantenuti.

Per poter utilizzare comunque criteri non soddisfacibili solitamente si imposta un obiettivo di copertura minore del 100%, e se si raggiunge o si supera quell'obiettivo il test viene accettato.

Tool per la generazione di casi di test

Ci sono molto tecniche per la generazione automatica dei casi di test. Hanno come vantaggio quello di non richiedere l'intervento umano, ma nessuna di queste tecniche può garantire coperture del 100% per qualsiasi programma, e non sempre è facile capire, al termine del test, se il programma si è comportato nel modo corretto (problema degli oracoli).

2.5 Esecuzione dei test

L'esecuzione dei test può essere resa automatica; quindi è conveniente che lo sia.

Normalmente è conveniente produrre codice di supporto alle attività di sviluppo ("scaffolding"); questo codice non è parte del prodotto finale come visto dall'utente, ma aiuta a rendere automatica la riesecuzione dei test ogni volta che il codice viene modificato. Esso comprende i cast di test, driver, stub, ed eventuali altri "sostituti" di parti dell'ambiente in cui il programma in sviluppo verrà lanciato (es.: emulazione di reti, ...). La generalizzazione o specializzazione di driver e stub è una questione di costi e riuso, e dovrebbe essere "calibrata" in base a quanto le unità di cui il programma è composto sono simili (o meno) tra loro.

Perché l'esecuzione automatica dei test sia davvero vantaggiosa occorre ovviamente che anche il controllo degli oracoli venga fatto automaticamente; ciò può essere fatto in 2 modi:

- oracoli basati su confronti: confronta gli output reali con quelli previsti, e riporta un fallimento se sono diversi; va bene se i casi di test sono pochi e vengono generati manualmente;
- codice self-checking come oracolo: il codice sotto test si auto-controlla, e notifica un fallimento nel caso l'esecuzione non proceda nel modo "corretto"; è utilizzabile anche con grandi quantità di casi di test, anche generati automaticamente, ma spesso effettua un controllo solo parziale.

A volte non c'è alternativa all'input e all'osservazione umana. In questi casi può essere automatizzata solo la riesecuzione dei test, tramite strumenti del tipo capture-and-replay. Il test così "catturato" è rieseguibile solo fintanto che i cambiamenti nel programma lo rendono non più valido.

Non sempre il codice da testare ha buone caratteristiche di controllabilità e osservabilità (es.: interfaccia grafica). Per questo la progettazione dei test automatici per queste parti di codice dovrebbe includere delle API per il controllo degli input e dei wrapper per l'osservazione degli output.

2.6 Test di unità con JUnit e CodeCover

JUnit è un framework per l'automazione del testing di unità di programmi Java che si ispira al eXtreme Programming. Ogni test contiene le asserzioni che controllano che il programma non contenga difetti. Esistono vari tool simili anche per altri linguaggi di programmazione.

Sviluppo guidato dai test

Iterazione dei seguenti passi:

1. scrivere i casi di test, e dalle specifiche (casi d'uso e storie dell'utente) scrivere un possibile scenario di chiamata di metodi e/o classi;
2. eseguire i test (che falliscono);
3. scrivere il codice fino a quando tutti i casi di test passano.

Anche se questo approccio inizialmente inizialmente è un po' lento (devo scrivere codice sia per l'applicazione che per i test), alla fine porta più vantaggi che svantaggi.

Supporta i programmatori nel:

- definire ed eseguire test e test set;
- formalizzare in codice i requisiti sulle unità;
- scrivere e debuggare il codice;
- integrare il codice e tenerlo sempre funzionante.

Test di una classe

Per testare una classe X si crea una classe ausiliaria (in genere con nome XTest). XTest contiene dei metodi annotati con "@Test", senza parametri e con void come tipo restituito, che rappresentano i casi di test.

Test di un metodo

Nel singolo metodo di test testiamo ogni metodo della classe sotto test. Si deve procedere così:

1. creare eventuali oggetti delle classe sotto test;
2. chiamare il metodo da testare e ottenere il risultato;
3. confrontare il risultato ottenuto con quello atteso:
   - per far questo usiamo dei metodi assert di JUnit che permettono di eseguire dei controlli;
   - se un assert fallisce, JUnit cattura il fallimento e lo comunica al tester.

N.B.: ogni metodo di test viene eseguito con degli oggetti diversi, in modo da non doversi preoccupare dell'ordine di esecuzione dei test (in pratica è come se il costruttore della classe di test venisse chiamato prima dell'esecuzione di ogni suo metodo).

Se un metodo deve essere eseguito una sola volta prima (o dopo) dei test si usa "@BeforeClass" (o "@AfterClass"), e deve essere static e public.

Per eseguire un metodo prima di ogni singolo test (o dopo) si usa "@Before" (o "@After").

Metodi assert

- assertEquals(expected, actual), assertEquals(String message, exp, act): per controllare l'uguaglianza di exp e act tramite il metodo equals (usa il metodo solo se è stato definito esplicitamente nella classe);
- assertSame(expected, actual), assertNotSame(exp, act), ...: uguale a sopra ma confronta tramite "==" al posto di usare il metodo equals;
- assertTrue(expression): per controllare che expression sia vera;
- assertNull(Object obj), assertNotNull(Object obj): per controllare che obj sia o non sia null;
- fail(), fail(String message): per terminare con un fallimento (lancia un errore di tipo AssertionFailedError).

assert di Java

Le prime versioni di JUnit avevano un metodo **assert** al posto del metodo **assertTrue**. Il nome è stato cambiato perché Java 1.4 ha introdotto "nativamente" la dichiarazione **assert**; essa può essere abilitata dalle preferenze di JUnit in eclipse o aggiungendo "-ea" agli argomenti dell'esecuzione. Due modalità:

- assert boolean_condition;
- assert boolean_condition: error_message.

Quando usare una dichiarazione assert:

- documentare condizioni che si sa essere vere;
- "assert false;" in codice irraggiungibile.

Non è da usare in sostituzione alle eccezioni.

JUnit in Eclipse

Per scrivere una caso di test:

1. scrivere la classe (o almeno il suo scheletro);
2. selezionare la classe per cui si vuole creare i casi di test e con il tasto destro → new → JUnit Test Case;
3. appare un dialogo: selezionare JUnit 4 e deseleziona tearDown, setUp, ...;
4. next → selezionare i metodi per cui si vuole creare i casi di test;
5. riempire i metodi di test con il codice che faccia i controlli opportuni;

Per eseguire un caso di test lanciarlo come "JUnit Test"

Eccezioni

È possibile verificare la presenza di eccezioni, quando un metodo deve generare un'eccezione; per farlo si deve annotare il metodo con "@Test(expected=Exception.class)".

Test parametrici con JUnit

Alcune volte si vuole chiamare lo stesso test con alcuni dati; si può fare ciò con i test parametrici:

1. si creano tante variabili d'istanza quanti sono i parametri di test;
2. si crea un costruttore del test che ha per parametri la n-upla identica alle variabili d'istanza, a cui vanno assegnati i valori passati al costruttore;
3. si crea un metodo statico annotato "@Parameters" che deve restituire una Collection (contiene le n-uple di parametri con i valori);
4. si annota la classe con "@RunWith(Parameterized.class)".

Altri suggerimenti per JUnit

Timer È possibile controllare se un test impiega troppo tempo annotandolo "@Test (timeout = x)" dove x indica i millisecondi;

@Ignore È possibile far ignorare dei test annotandoli con "@Ignore" (ovviamente seguito da "@Test");

CodeCover

Date delle classi e dei test che le testano, comunica quanto (e quale) codice di quelle classi viene coperto (cioè eseguito) durante l'esecuzione dei test.

Utilizzo in Eclipse:

1. abilitare CodeCover nelle proprietà del progetto e selezionare i criteri desiderati;
2. click destro sulla classe sotto test (nel package explorer di Eclipse) → selezionare "Use For Coverage Measurement";
3. click destro sulla "classe test" → Run As → CodeCover Measurement For JUnit.

3 Verifica del codice

3.1 Concetti generali sulla verifica

Verificare un programma significa provare che un programma è corretto, cioè che vale:

$$\{P\}\, S\, \{Q\}$$

detta **tripla di Hoare**; se P è vero prima di eseguire il programma S, allora dopo varrà Q, dove P e Q sono asserzioni (rispettivamente precondizioni e postcondizioni) scritte in logica proposizionale.

Criteri per la classificazione degli approcci alla verifica

proof-based la descrizione del sistema è un insieme Γ di formule (logiche) e la specifica è un'altra formula ϕ; il metodo di verifica consiste nel provare che $\Gamma \vdash \phi$ [1];

model-based la descrizione del sistema è un modello $\mathcal{M}$ (logico) e la specifica è una formula ϕ; il metodo di verifica consiste nel calcolare se $\mathcal{M}$ soddisfa ϕ (si scrive $\mathcal{M} \models \phi$); questo calcolo è solitamente automatico per modelli finiti, e come approccio è potenzialmente più semplice rispetto al precedente;

grado di automazione da completamente manuale a completamente automatico (la maggior parte delle tecniche sono una via di mezzo);

property-verification la specifica (da verificare) descrive una singola proprietà del sistema;

full-verification la specifica descrive completamente il comportamento del sistema;

dominio d'applicazione hardware o software, sequenziale o concorrente, interattivo o "batch", ...

pre-sviluppo modellazione e verifica vengono iniziate insieme allo sviluppo del programma, o prima;

post-sviluppo modellazione e verifica vengono svolte dopo che il programma è stato sviluppato.

Vantaggi della verifica

documentazione la struttura logica della specifica formale, scritta in una logica appropriata, è tipicamente usata come linea guida nella scrittura del codice vero e proprio;

time-to-market il debug fatto in collaborazione al testing è costoso, oneroso in termini di tempo e spesso nel tentativo di risolvere dei malfunzionamenti se ne creano altri. La verifica dei programmi può ridurre significativamente il tempo di sviluppo e quello di manutenzione eliminando molti errori già dalle prime fasi del processo di sviluppo;

riuso il software correttamente specificato e verificato è più facilmente riutilizzabile (è più chiaro il contesto nel quale può effettivamente tornare utile);

certificazioni i software per sistemi critici devono fornire una garanzia di corretto funzionamento in condizioni appropriate di utilizzo.

[1] Sia Γ un set di formule e ϕ un'altra formula. $\Gamma \vdash \phi$ significa che dal set di formule Γ posso ricavare una prova che ϕ vale.

3.2 Design by Contract

Il Design by Contract è un approccio alla verifica (del codice) proof-based, property-verification e pre-sviluppo. Segue l'idea (valida soprattutto per l'OO programming) che l'interfaccia di un modulo definisce un contratto. Un ***contratto*** è un accordo tra cliente e fornitore che:

- lega cliente e fornitore;
- è esplicito (scritto);
- specifica obblighi e benefici delle parti (normalmente mappati in modo "duale");
- non contiene clausole nascoste.

Oggetto del contratto è il software (in OO un insieme di classi), il cliente è chi utilizzerà il programma, o chi l'ha commissionato, e il fornitore è il programmatore, o chi ha ricevuto la commissione.

I contratti definiti possono essere verificati sia staticamente che dinamicamente. La verifica statica è un metodo di verifica vero e proprio, mentre la verifica dinamica (cioè in esecuzione) è più un metodo di testing white-box.

In un contratto si possono definire:

precondizioni cosa il metodo richiede (obblighi per il cliente);

postcondizioni cosa il metodo fornisce (obblighi per il fornitore).

Precondizioni e postcondizioni sono predicati logici, ma per i programmatori è conveniente usare la sintassi del linguaggio di programmazione o del tool.

Precondizioni In generale un metodo non deve gestire ogni possibile input. La scelta delle precondizioni è una scelta di progetto; non c'è una regola assoluta. Però è meglio scrivere metodi semplici che soddisfino un contratto ben definito che un metodo che "cerca" di gestire tutte le situazioni possibili.

Il cliente deve garantire la precondizioni prima di invocare il metodo.
Sia *pre* la precondizione per un metodo m di un oggetto x, il cliente dovrà controllare l'invocazione del metodo m di x in questo modo:

```
if(pre) x.m();
else {/* special treatment */}
```

oppure essere sicuro che *pre* valga prima della chiamata di m, in base al ragionamento sul programma.

Postcondizioni Se il metodo invocato rispetta le postcondizioni il cliente potrà assumere che esse siano vere dopo l'invocazione del metodo (l'invocante può anche non sapere "come è fatto" il metodo invocato, ma ha delle informazioni su cosa fa o non fa).

Solitamente il cliente controlla le precondizioni prima di invocare un metodo ma assume (non controlla) che le postcondizioni valgano dopo, mentre il fornitore assume (non controlla) che le precondizioni valgano e garantisce le postcondizioni.
inquesto modo si evitano controlli ridondanti.

Precondizioni forti e postcondizioni deboli rendono un programma più facile da scrivere.

Invariante È una condizione vera dopo la creazione dell'oggetto e dopo ogni operazione (cioè sempre durante la vita di tale oggetto).

L'invariante definisce:

- una postcondizione ulteriore: l'implemetazione di ogni metodo non deve violare l'invariante;
- una precondizione ulteriore: chi implementa il metodo sa che l'invariante vale.

Contratti e documentazione I contratti possono essere inseriti anche prima di fornire una vera implementazione del metodo; dalle definizioni dei contratti si possono estrarre in modo automatico le precondizioni e le postcondizioni e gli invarianti, che documentano cosa fa la classe.

Vantaggi tecnici nell'uso del DbC

- Processo di sviluppo meno dispersivo;
- permette di scrivere codice corretto, riutilizzabile in altri programmi (è facile controllare *se* può essere utilizzato in un determinato contesto);
- gestione delle eccezioni guidata da una precisa definizione dei casi "eccezionali";
- documentazione dell'interfaccia sempre aggiornata e attendibile;
- generazione automatica della documentazione;
- gli errori si presentano più vicini alla loro causa, quindi possono essere trovati più facilmente e più velocemente;
- guida per la generazione dei casi di test black-box.

3.3 JML

JML è un insieme di tool e di tecniche per il Design by Contract in Java. Supporta:

- invarianti di classe;
- precondizioni;
- postcondizioni;
- asserzioni locali;
- invarianti sui cicli.

3.3.1 Sintassi

Per rendere JML facile da usare le annotazioni JML vengono aggiunte come commenti nel file ".java" (quindi Il ".java" contiene il codice e il suo modello formale secondo DbC):

```
/*@ <JML specification> @*/
```

o

```
//@ <JML specification>
```

Le condizioni sono scritte come espressioni boolean di java con alcuni operatori in più e alcune parole chiave.

Operatori e parole chiave

requires per specificare le precondizioni ("richiede");

ensures per specificare le postcondizioni ("garantisce");

\old(x) per riferirmi al valore di x *prima* dell'invocazione del metodo;

\result per riferirmi al valore restituito;

invariant per specificare invarianti;

assert per richiedere che una certa condizione sia verificata ad un certo punto all'interno del programma;

normal_behaviour per indicare che le seguenti precondizioni e postcondizioni sono relative ad un funzionamento in condizioni normali del metodo;

exceptional_behaviour per indicare che le seguenti precondizioni e postcondizioni sono relative ad un metodo che non termina normalmente, cioè che lancia un'eccezione;

also per combinare più specifiche e per indicare che un metodo eredita le specifiche dalla superclasse;

assignable x per specificare che ad un metodo è consentito modificare il valore di x;

a[i] i-esimo elemento dell'array a;

a[i..j] tutti gli elementi dell'array a dall'i-esimo al j-esimo;

a[*] tutti gli elementi dell'array a;

o.* tutti i campi dell'oggetto o;

\nothing nessuna variabile;

\everything tutte le variabili;

a ==> b a implica b;

a <== b a segue da b;

a <==> b a vale se e solo se vale b (e viceversa);

a <=!=> b a non vale se e solo se vale b (e viceversa).

Precondizioni e postcondizioni sono da mettere subito prima dei relativi metodi, mentre gli invarianti nel corpo della classe.

Quantificatori la sintassi è:

(<quant.> <type> <var>; <range predicate>; <expression>)

(ovviamente preceduta da "ensures", "requires", "invariant" o "assert"), dove "quant." può essere:

\forall quantificatore universale;

\exists quantificatore esistenziale;

\sum, \product, \min, \max quantificatori generali;

\num_of quantificatore numerico.

Information hiding Il livello di visibilità di una specifica JML è determinato dal livello di visibilità del metodo che specifica.

Se una specifica pubblica deve valutare una variabile non pubblica allora la variabile va specificata come segue:

```
private /*@ spec_public @*/ <type> <var_name>;
```

Per gli invarianti si può dichiarare la visibilità.

Variabili nulle Molti invarianti e pre e postcondizioni richiedono che un certo riferimento non sia mai `null`. Per specificare che una certa variabile non sarà mai `null`:

```
/*@ not_null @*/ <type> <var_name>
```

utilizzabile sia nella dichiarazione di variabili e/o campi che per i parametri presenti nella firma di un metodo che per il parametro ritornato da un metodo.

Espressioni utilizzabili Ogni espressione utilizzata all'interno di specifiche JML dev'essere "side-effect free", cioè non deve alterare i valori dei campi delle classi o dei parametri dei metodi.

Per poter invocare un metodo all'interno di una specifica JML occorre dichiararlo **`pure`**:

```
public /*@ pure @*/ <ret_type> <method_name>()
```

N.B.: un metodo dichiarato **`pure`** non può nemmeno lanciare eccezioni.

Eccezioni Si può specificare che una certa eccezione va lanciata da un metodo se è verificata una certa precondizione:

```
//@ public exceptional_behaviour
//@ requires <condition>;
//@ signals_only <ExceptionType1>,...,<ExceptionTypeN>;
```

oppure

```
//@ public exceptional_behaviour
//@ requires <condition>;
//@ signals (Exception e) e instanceof <ExceptionType1> || ... || <ExceptionTypeN>
```

Loop invariants Invarianti che devono essere veri ad ogni iterazione di un ciclo (prima della prima iterazione e dopo ognuna). Possono contenere riferimenti a variabili presenti solo all'interno del ciclo. Utilizzo:

```
//@ loop_invariant
//@ <condition to be verified at every iteration>;
```

Sottotipazione Tutte le istanze di un certo tipo T devono soddisfare le specifiche JML di tutti i suoi supertipi.

Le specifiche possono eventualmente essere estese tramite l'utilizzo della parola chiave **`also`**.

3.3.2 Java assert

Java supporta in modo nativo le asserzioni, tramite l'utilizzo della parola chiave **`assert`**. Può essere utilizzata in due modi:

```
assert <expression>
```

oppure

```
assert <expression>: <message>
```

L'istruzione controlla che `<expression>` sia vera; se è falsa lancia un `AssertionError`, e nel secondo caso passa all'oggetto lanciato `<message>` come messaggio.

Di default le asserzioni sono disabilitate. Per abilitarle occorre compilare le classi utilizzando l'argomento `-enableassertion` (o `-ea`).

3.3.3 Tools per JML

jmlc compilatore JML, compila i file ".java" per avere classi instrumentate;

jmlrac interprete JML/Java, esegue il controllo di tutte le asserzioni JML: se una condizione non è verificata solleva un'eccezione particolare;

escjava2 controllo statico del codice per la prova di correttezza;

jmlunit tool per test d'unità JML/JUnit;

jmldoc per la generazione automatica di pagine html contenenti la documentazione del codice;

...

3.3.3.1 Plugin per Eclipse

Sono disponibili i seguenti plugin:

JML2 (`http://pm.inf.ethz.ch/research/universes/tools/eclipse/`) un po' vecchio;

Modern Jass (`http://modernjass.sourceforge.net/`);

JML4c (`http://www.cs.utep.edu/cheon/download/jml4c/`);

OpenJML (`http://jmlspecs.sourceforge.net/`) richiede OpenJDK (Java 7); non supporta tutto ma il plugin per Eclipse funziona bene.

Key (`http://www.key-project.org/download/`);

Noi useremo OpenJML per l'analisi dinamica, Key per l'analisi statica.

4 Verifica tramite Model Checking

Il Model Checking è un approccio alla verifica model-based, property-verification, automatico. È pensato per essere usato per sistemi concorrenti, inerattivi, e all'origine era usato post-sviluppo.

Consta principalmente di 3 fasi:

1. scrittura del modello da parte dell'utente (programmatore o simili);
2. asserzione di proprietà relative al modello da parte dell'utente;
3. verifica della validità delle proprietà; in caso le proprietà non siano valide, può fornire controesempi.

Richiede quindi:

- un *framework per la modellazione dei sistemi* (tipicamente un linguaggio per la descrizione di modelli);
- un *linguaggio di specifica* per la descrizione delle proprietà che devono essere verificate;
- un *metodo di verifica* per stabilire se la descrizione di un sistema soddisda la specifica.

Solitamente il Model Checking si concentra su proprietà temporali e sull'evoluzione temporale dei sistemi, quindi:

- i modelli descrivono sistemi "a stati";
- le proprietà sono scritte in logica temporale.

Vedremo 2 logiche temporali:

Linear-time Temporal Logic (LTL) è una logica temporale lineare, cioè che tratta il tempo come un insieme di percorsi, dove un percorso è una sequenza di stati del sistema;

Computation Tree Logic (CTL) è una logica temporale "ramificata", cioè che mostra ad ogni istante le possibili evoluzioni future dello stato del sistema.

4.1 Linear-time Temporal Logic (LTL)

Operatori

$\top$	true
$\bot$	false
$\neg$	negazione
$\vee$	or logico
$\wedge$	and lofgico

$\rightarrow$	implicazione ($a \rightarrow b$ equivale a scrivere $\neg a \vee b$)
X	("neXt state") Xa significa che a sarà vera nel prossimo stato
F	("some Future state") Fa significa che a sarà vera in almeno uno degli stati futuri
G	("all future state", Globally) Ga significa che a sarà vera in tutti gli stati futuri
U	("Until") spiegato più avanti
W	("Weak-until") spiegato più avanti
R	("Release") spiegato più avanti

Priorità degli operatori Dalla più alta alla più bassa:

- $\neg$, X, F, G
- U, R, W
- $\wedge$, $\vee$
- $\rightarrow$

Path Un ***path*** (o ***percorso***) in un modello $\mathcal{M} = (S, \rightarrow, L)$ [1] è una sequenza infinita di stati $s_1, s_2, s_3, \ldots$ in S tali che, per ogni $i \geq 1$, $s_i \rightarrow s_{i+1}$. Scriviamo il path come $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$

Sia $\pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ un path. Allora indichiamo con $\pi^i = s_i \rightarrow s_{i+1} \rightarrow s_{i+2} \rightarrow \ldots$ il suo sottopath che inizia dallo stato s_i.

Soddisfacimento delle regole: path Sia $\mathcal{M} = (S, \rightarrow, L)$ un modello e $\pi = s_1 \rightarrow \ldots$ un path per $\mathcal{M}$. π soddisfa una formula LTL se:

1. $\pi \models \top$
2. $\pi \not\models \bot$
3. $\pi \models p$ se e solo se $p \in L(s_1)$
4. $\pi \models \neg\phi$ se e solo se $\pi \not\models \phi$
5. $\pi \models \phi_1 \wedge \phi_2$ se e solo se $\pi \models \phi_1$ e $\pi \models \phi_2$
6. $\pi \models \phi_1 \vee \phi_2$ se e solo se $\pi \models \phi_1$ o $\pi \models \phi_2$
7. $\pi \models \phi_1 \rightarrow \phi_2$ se e solo se $\pi \models \phi_2$ ogniqualvolta $\pi \models \phi_1$
8. $\pi \models \mathrm{X}\phi$ se e solo se $\pi^2 \models \phi$
9. $\pi \models \mathrm{G}\phi$ se e solo se $\pi^i \models \phi$ per ogni $i \geq 1$
10. $\pi \models \mathrm{F}\phi$ se e solo se esiste $i \geq 1$ tale che $\pi^i \models \phi$
11. $\pi \models \phi \mathrm{U} \psi$ se e solo se esiste $i \geq 1$ tale che $\pi^i \models \psi$ e per tutti i $j = 1, \ldots, i-1$ vale $\pi^j \models \phi$ [2]

[1] S è l'insieme degli stati del modello, $\rightarrow$ è l'insieme delle transizioni e L è la funzione di labeling, cioè una funzione che indica per ogni stato quali proposizioni siano vere.

[2] Richiede che ψ si verifichi prima o poi.

12. $\pi \models \phi \mathrm{W} \psi$ se e solo se vale almeno una delle seguenti condizioni:
    a) esiste $i \geq 1$ tale che $\pi^i \models \psi$ e per tutti i $j = 1, \ldots, i-1$ vale $\pi^j \models \phi$
    b) $\pi^k \models \phi$ per ogni $k \geq 1$
13. $\pi \models \phi \mathrm{R} \psi$ [3] se e solo se vale almeno una delle seguenti condizioni:
    a) esiste $i \geq 1$ tale che $\pi^i \models \phi$ e per tutti i $j = 1, \ldots, i$ vale $\pi^j \models \psi$
    b) $\pi^k \models \psi$ per ogni $k \geq 1$

Soddisfacimento delle regole: stato Sia $\mathcal{M} = (S, \to, L)$ un modello, $s \in S$, ϕ una formula LTL. $\mathcal{M}, s \models \phi$ (o, più semplicemente, $s \models \phi$) se, per ogni possibile path π di $\mathcal{M}$ che inizia con s, $\pi \models \phi$.

Pattern pratici di specifica

- Proprietà di safety: qualcosa è sempre vero ($\mathrm{G}(\phi)$) oppure qualcosa di spiacevole non si verifica mai ($\mathrm{G}(\neg\phi)$);
- Proprietà di liveness: qualcosa accadrà ($\mathrm{F}(\phi)$) o qualcosa di buono si verifica ogni tanto, all'infinito ($\mathrm{GF}\psi \vee \mathrm{G}(\phi \to \mathrm{F}\psi)$)
- ϕ è falsa:
  - globalmente: $\mathrm{G}(\neg\phi)$
  - prima di ψ: $\mathrm{F}\psi \to (\neg\phi \mathrm{U} \psi)$
  - dopo ϑ: $\mathrm{G}(\vartheta \to \mathrm{G}(\neg\phi))$
  - tra ϑ e ψ: $\mathrm{G}((\vartheta \wedge \neg\psi \wedge \mathrm{F}\psi) \to (\neg\phi \mathrm{U} \psi))$
- ϕ può diventare vera:
  - globalmente: $\mathrm{F}\phi$
  - prima di ψ: $\neg\psi \mathrm{W}(\phi \wedge \neg\psi)$
  - dopo ϑ: $\mathrm{G}(\neg\vartheta) \vee \mathrm{F}(\vartheta \wedge \mathrm{F}\phi)$
  - tra ϑ e ψ: $\mathrm{G}((\vartheta \wedge \neg\psi) \to (\neg\psi \mathrm{W}(\phi \wedge \neg\psi)))$
  - dopo ϑ e fino a ψ: $\mathrm{G}((\vartheta \wedge \neg\psi) \to (\neg\psi \mathrm{U}(\phi \wedge \neg\psi)))$
- ϕ è sempre vera:
  - globalmente: $\mathrm{G}\phi$
  - prima di ψ: $\mathrm{F}\psi \to (\phi \mathrm{U} \psi)$
  - dopo ϑ: $\mathrm{G}(\vartheta \to \mathrm{G}\phi)$
  - tra ϑ e ψ: $\mathrm{G}((\vartheta \wedge \neg\psi \wedge \mathrm{F}\psi) \to (\phi \mathrm{U} \psi))$
  - dopo ϑ e fino a ψ: $\mathrm{G}((\vartheta \wedge \neg\psi) \to (\phi \mathrm{W} \psi))$

[3] È il duale di U; infatti $\phi \mathrm{R} \psi$ equivale a scrivere $\neg(\neg\phi \mathrm{U} \neg\psi)$.

Equivalenze Due formule LTL ϕ e ψ sono equivalenti ($\phi \equiv \psi$) se per ogni modello $\mathcal{M}$ e ogni percorso π in $\mathcal{M}$: $\pi \models \phi$ se e solo se $\pi \models \psi$.

1. $\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi$
   $\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi$
2. $\neg \mathrm{G}\phi \equiv \mathrm{F}\neg\phi$
   $\neg \mathrm{F}\phi \equiv \mathrm{G}\neg\phi$
   $\neg \mathrm{X}\phi \equiv \mathrm{X}\neg\phi$
3. $\neg(\phi \mathrm{U} \psi) \equiv \neg\phi \mathrm{R} \neg\psi$
   $\neg(\phi \mathrm{R} \psi) \equiv \neg\phi \mathrm{U} \neg\psi$
4. $\mathrm{F}(\phi \vee \psi) \equiv \mathrm{F}\phi \vee \mathrm{F}\psi$ [4]
   $\mathrm{G}(\phi \wedge \psi) \equiv \mathrm{G}\phi \wedge \mathrm{G}\psi$ [5]
5. $\mathrm{F}\phi \equiv \top \mathrm{U} \phi$
   $\mathrm{G}\phi \equiv \bot \mathrm{R} \phi$
6. $\phi \mathrm{U} \psi \equiv \phi \mathrm{W} \psi \wedge \mathrm{F}\psi$
   $\phi \mathrm{W} \psi \equiv \phi \mathrm{U} \psi \vee \mathrm{G}\phi$
7. $\phi \mathrm{W} \psi \equiv \psi \mathrm{R}(\phi \vee \psi)$
   $\phi \mathrm{R} \psi \equiv \psi \mathrm{W}(\phi \wedge \psi)$

Teorema Per qualsiasi formula LTL ϕ e ψ vale l'equivalenza: $\phi \mathrm{U} \psi \equiv \neg(\neg\psi \mathrm{U}(\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\psi$.

Limiti Esistono affermazioni impossibili da esprimere tramite LTL; ciò deriva dal fatto che la logica LTL non può affermare l'esistenza di path con determinate caratteristiche (es.: dallo stato attuale, esiste un path per cui vale ϕ).

Per controllare se esiste un path che parte da uno stato s che soddisfa la formula LTL ϕ, controlliamo invece che tutti i path soddisfino $\neg\phi$; una risposta positiva a questa condizione equivale ad una risposta negativa alla precedente, e viceversa.
Per le proprietà che usano sia quantificatori universali che esistenziali sui path questo approccio non può essere usato, perché la formula complementare contiene ancora entrambi.

4.2 Computation Tree Logic (CTL)

Operatori

$\top$	true
$\bot$	false
$\neg$	negazione
$\vee$	or logico
$\wedge$	and lofgico
$\rightarrow$	implicazione ($a \rightarrow b$ equivale a scrivere $\neg a \vee b$)

[4] F è distributivo rispetto a $\vee$, ma non rispetto a $\wedge$.
[5] G è distributivo rispetto a $\wedge$, ma non rispetto a $\vee$.

AX ("neXt state") AXa significa che a sarà vera in tutti i prossimi possibili stati

EX ("neXt state") EXa significa che a sarà vera per almeno uno dei prossimi stati

AF ("some Future state") AFa significa che a sarà vera in almeno uno degli stati futuri di ogni possibile path che parte dallo stato attuale

EF ("some Future state") EFa significa che a sarà vera in almeno uno degli stati futuri di almeno uno dei possibili path che partono dallo stato attuale

AG ("all future state", Globally) AGa significa che a sarà vera in tutti gli stati futuri di ogni possibile path che parte dallo stato attuale

EG ("all future state", Globally) EGa significa che a sarà vera in tutti gli stati futuri di almeno uno dei possibili path che partono dallo stato attuale

AU ("Until") A $[\phi \mathrm{U} \psi]$ [6] significa che ϕ è per forza vero finché non diventa vero ψ, per ogni possibile path che parte dallo stato attuale

EU ("Until") E $[\phi \mathrm{U} \psi]$ [7] significa che ϕ è per forza vero finché non diventa vero ψ, per almeno uno dei possibili path che partono dallo stato attuale

Priorità degli operatori Dalla più alta alla più bassa [8]:

- $\neg$, AX, EX, AF, EF, AG, EG
- AU, EU
- $\wedge$, $\vee$
- $\rightarrow$

Soddisfacimento delle regole: definizione approssimativa Sia $\mathcal{M} = (S, \rightarrow, L)$ un modello, $s \in S$, ϕ una formula CTL. La verifica di $\mathcal{M}, s \models \phi$ (o, più semplicemente, $s \models \phi$) va fatta ricorsivamente sulla struttura di ϕ, nel modo seguente:

- se ϕ è atomica, allora il soddisfacimento di ϕ da parte di s dipende da L;
- se la congiunzione "principale" di ϕ è binaria ($\wedge$, $\vee$, $\neg$, $\top$, ...) allora il soddisfacimento di ϕ sa parte di s dipende dall'usuale tabella della verità della congiunzione, e dall'ulteriore ricorsione su ϕ;
- se l'operatore "principale" inizia per A, si toglie tale A dalla formula e si verifica che ogni path da s verifichi tale formula;
- se l'operatore "principale" inizia per E, si toglie tale E dalla formula e si verifica che almeno un path da s verifichi tale formula.

[6] Si può scrivere anche AU (ϕ, ψ) o $\phi \mathrm{AU} \psi$.
[7] Si può scrivere anche EU (ϕ, ψ) o $\phi \mathrm{EU} \psi$.
[8] Nelle slides dice che assume le priorità simili alle precedenti, e poi mette AU e EU sullo stesso piano di $\rightarrow$; nel dubbio le ho scritte come le precedenti.

Soddisfacimento delle regole: definizione rigorosa Sia $\mathcal{M} = (S, \to, L)$ un modello, $s \in S$, ϕ una formula CTL. La relazione $\mathcal{M}, s \models \phi$ (o $s \models \phi$) è definita tramite induzione strutturale su ϕ:

1. $s \models \top$
2. $s \not\models \bot$
3. $s \models p$ se e solo se $p \in L(s)$
4. $s \models \neg\phi$ se e solo se $s \not\models \phi$
5. $s \models \phi_1 \wedge \phi_2$ se e solo se $s \models \phi_1$ e $s \models \phi_2$
6. $s \models \phi_1 \vee \phi_2$ se e solo se $s \models \phi_1$ o $s \models \phi_2$
7. $s \models \phi_1 \to \phi_2$ se e solo se $s \not\models \phi_1$ o $s \models \phi_2$
8. $s \models \mathrm{AX}\phi$ se e solo se per tutti gli s_1 tali che $s \to s_1$ abbiamo che $s_1 \models \phi$
9. $s \models \mathrm{EX}\phi$ se e solo se per almeno un s_1 tale che $s \to s_1$ abbiamo che $s_1 \models \phi$
10. $s \models \mathrm{AG}\phi$ se e solo se per tutti i path $s_1 \to s_2 \to s_3 \to \ldots$, con $s_1 = s$, e per tutti gli s_i lungo il path, vale che $s_i \models \phi$
11. $s \models \mathrm{EG}\phi$ se e solo se per almeno un path $s_1 \to s_2 \to s_3 \to \ldots$, con $s_1 = s$, e per tutti gli s_i lungo tale path, vale che $s_i \models \phi$
12. $s \models \mathrm{AF}\phi$ se e solo se per tutti i path $s_1 \to s_2 \to s_3 \to \ldots$, con $s_1 = s$, esiste un s_i lungo il path tale che $s_i \models \phi$
13. $s \models \mathrm{EF}\phi$ se e solo se per almeno un path $s_1 \to s_2 \to s_3 \to \ldots$, con $s_1 = s$, esiste un s_i lungo tale path tale che $s_i \models \phi$
14. $s \models \mathrm{A}[\phi_1 \mathrm{U} \phi_2]$ se e solo se per tutti i path $s_1 \to s_2 \to s_3 \to \ldots$, con $s_1 = s$, vale che $\phi_1 \mathrm{U} \phi_2$ [9]
15. $s \models \mathrm{E}[\phi_1 \mathrm{U} \phi_2]$ se e solo se per almeno un path $s_1 \to s_2 \to s_3 \to \ldots$, con $s_1 = s$, vale che $\phi_1 \mathrm{U} \phi_2$

Pattern pratici di specifica

- ϕ è falsa:
  - globalmente: $\mathrm{AG}(\neg\phi)$
  - prima di ψ: $\mathrm{A}[(\neg\phi \vee \mathrm{AG}(\neg\psi))\,\mathrm{W}\psi]$
  - dopo ϑ: $\mathrm{AG}(\vartheta \to \mathrm{AG}(\neg\phi))$
- ϕ può diventare vera:
  - globalmente: $\mathrm{AF}\phi$
  - prima di ψ: $\mathrm{A}[\neg\psi \mathrm{W}(\phi \wedge \neg\psi)]$
  - dopo ϑ: $\mathrm{A}[\neg\vartheta \mathrm{W}(\vartheta \wedge \mathrm{AF}\phi)]$
  - tra ϑ e ψ: $\mathrm{AG}((\vartheta \wedge \neg\psi) \to \mathrm{A}[\neg\psi \mathrm{W}(\phi \wedge \neg\psi)])$
  - dopo ϑ e fino a ψ: $\mathrm{AG}((\vartheta \wedge \neg\psi) \to \mathrm{A}[\neg\psi \mathrm{U}(\phi \wedge \neg\psi)])$

[9] $\phi_1 \mathrm{U} \phi_2$ intesa come per le formule LTL.

- ϕ è sempre vera:
  - globalmente: $\mathrm{AG}\phi$
  - prima di ψ: $\mathrm{A}\left[(\phi \vee \mathrm{AG}\left(\neg\psi\right))\,\mathrm{W}\psi\right]$
  - dopo ϑ: $\mathrm{AG}\left(\vartheta \rightarrow \mathrm{AG}\phi\right)$
  - tra ϑ e ψ: $\mathrm{AG}\left((\vartheta \wedge \neg\psi) \rightarrow \mathrm{A}\left[(\phi \vee \mathrm{AG}\left(\neg\psi\right))\,\mathrm{W}\psi\right]\right)$
  - dopo ϑ e fino a ψ: $\mathrm{AG}\left((\vartheta \wedge \neg\psi) \rightarrow \mathrm{A}\left[\phi\mathrm{W}\psi\right]\right)$

Equivalenze Due formule CTL ϕ e ψ sono equivalenti ($\phi \equiv \psi$) se ogni stato in ogni modello che soddisfa una delle due soddisfa anche l'altra.

1. $\neg\mathrm{AF}\phi \equiv \mathrm{EG}\neg\phi$
   $\neg\mathrm{EF}\phi \equiv \mathrm{AG}\neg\phi$
   $\neg\mathrm{AX}\phi \equiv \mathrm{EX}\neg\phi$
2. $\mathrm{AF}\phi \equiv \mathrm{A}\left[\top\mathrm{U}\phi\right]$
   $\mathrm{EF}\phi \equiv \mathrm{E}\left[\top\mathrm{U}\phi\right]$
3. $\phi\mathrm{U}\psi \equiv \phi\mathrm{W}\psi \wedge \mathrm{F}\psi$
   $\phi\mathrm{W}\psi \equiv \phi\mathrm{U}\psi \vee \mathrm{G}\phi$
4. $\phi\mathrm{W}\psi \equiv \psi\mathrm{R}\left(\phi \vee \psi\right)$
   $\phi\mathrm{R}\psi \equiv \psi\mathrm{W}\left(\phi \wedge \psi\right)$
5. $\mathrm{AG}\phi \equiv \phi \wedge \mathrm{AXAG}\phi$
   $\mathrm{EG}\phi \equiv \phi \wedge \mathrm{EXEG}\phi$
6. $\mathrm{AF}\phi \equiv \phi \vee \mathrm{AXAF}\phi$
   $\mathrm{EF}\phi \equiv \phi \vee \mathrm{EXEF}\phi$
7. $\mathrm{A}\left[\phi\mathrm{U}\psi\right] = \psi \vee \left(\phi \wedge \mathrm{AXA}\left[\phi\mathrm{U}\psi\right]\right)$
   $\mathrm{E}\left[\phi\mathrm{U}\psi\right] = \psi \vee \left(\phi \wedge \mathrm{EXE}\left[\phi\mathrm{U}\psi\right]\right)$

Teorema Per qualsiasi formula CTL ϕ e ψ vale l'equivalenza: $\mathrm{A}\left[\phi\mathrm{U}\psi\right] \equiv \neg\left(\mathrm{E}[\neg\psi\mathrm{U}\left(\neg\phi \wedge \neg\psi\right)] \vee \mathrm{EG}\psi\right)$.

Release e Weak-until Possono essere definiti come segue:

- $\mathrm{A}\left[\phi\mathrm{R}\psi\right] = \neg\mathrm{E}\left[\neg\phi\mathrm{U}\neg\psi\right]$
- $\mathrm{E}\left[\phi\mathrm{R}\psi\right] = \neg\mathrm{A}\left[\neg\phi\mathrm{U}\neg\psi\right]$
- $\mathrm{A}\left[\phi\mathrm{W}\psi\right] = \mathrm{A}\left[\psi\mathrm{R}\left(\phi \vee \psi\right)\right]$
- $\mathrm{E}\left[\phi\mathrm{W}\psi\right] = \mathrm{E}\left[\psi\mathrm{R}\left(\phi \vee \psi\right)\right]$

Limiti CTL (a differenza di LTL) non permette di selezionare un range di path descrivendoli tramite una formula (es.: "per tutti i path per cui vale ... allora vale ... ").

4.3 NuSMV (New Symbolic Model Verifier)

NuSMV è un programma per la verifica di modelli. Fornisce un linguaggio per la descrizione dei modelli, e controlla la validità di formule LTL e CTL su di essi. Per ogni formula restituisce vero se è verificata, altrimenti fornisce un controesempio.

Moduli I programmi SMV sono composti da moduli; uno dei moduli dev'essere chiamato `main`. Ogni modulo può dichiarare variabili e assegnare loro dei valori; gli assegnamenti solitamente forniscono il valore iniziale e il "prossimo" valore, dato come espressione avente come termini i valori attuali delle variabili; l'espressione può essere non deterministica. Ogni modulo può ricevere delle variabili in ingresso (tranne il `main`) e usare altri moduli come variabili.

Algoritmo 4.1 Struttura di un modulo NuSMV

```
MODULE <module_name>(<input_variable_1>,<input_variable_2>,...)
VAR
        <variable_name_1> : <variable_type>;
        <variable_name_2> : <variable_type>;
        .
        .
        .
ASSIGN
        init(<variable_name_1>) := <value>;
        assign(<variable_name_1>) := <expression|case|value|...>;
        .
        .
        .
DEFINE
        <variable_name_3> := <expression|case|value|...>;
        .
        .
        .
FAIRNESS <expression>;
.
.
.

LTLSPEC ...;
...
CTLSPEC ...;
...
```

I tipi delle variabili possono essere:

- `boolean`;
- `x..y` (con `x` e `y` interi, col significato di "da `x` a `y`");
- `{<valore1>, <valore2>, ...}` (dove i vari valori sono etichette significative per chi scrive il modulo);

ognuno dei quali può essere dichiarato come array, facendolo precedere dall'espressione "`array m..n of`".

Per calcolare il valore corrente delle variabili, in caso la relativa `assign` sia definita come espressione, vengono usati i valori delle variabili dello stato precedente; per le variabili definite nella sezione `DEFINE` invece vengono usati i valori attuali. Notare che per quest'ultime non è possibile usare espressioni non deterministiche.

Composizione sincrona e asincrona Normalmente in SMV c'è un clock globale che regola l'esecuzione di tutti i moduli.

È possibile creare moduli asincroni utilizzando la parola chiave **process**, da usare quando il modulo viene dichiarato come variabile all'interno di un altro modulo (es.: `<variable_name> : process <module_name>;`). In questo caso i moduli lavorano a velocità diverse, interfogliandosi arbitrariamente. Ad ogni colpo di clock, un modulo viene scelto casualmente ed eseguito per un ciclo.

FAIRNESS In SMV si possono introdurre vincoli di fairness: la dicitura "**FAIRNESS** ϕ" indica che SMV, controllando una qualsiasi specifca ψ, ignorerà i path lungo i quali ϕ non è soddisfatta infinite volte.

Esempio: se un modulo viene dichiarato **process**, potrebbe essere che non venga mai selezionato per essere eseguito; per far sì che le specifiche non vengano controllate lungo tali path (che si presume nella realtà non capiteranno mai, se lo schedulatore reale è ben fatto) aggiungiamo il vincolo seguente: **FAIRNESS running**. Verranno quindi presi in considerazione solo path in cui il modulo viene selezionato per l'esecuzione infinite volte.

La variabile **running** indica appunto se il modulo (nel quale ne viene controllato il valore e che è stato dichiarato **process**) è in esecuzione.

Bounded model checking L'esecuzione in modalità *bounded model checking* cerca i controesempi in ordine di lunghezza, dal più corto al più lungo fino ad una data soglia (di default è 10).

N.B.: un fallimento nel cercare un cotroesempio potrebbe voler dire che non esistono controesempi di lunghezza inferiore o uguale alla soglia, ma non che non ne esistono in assoluto.

4.4 CTL*

CTL* è una logica che combina i poteri espressivi di LTL e CTL.

È simile al CTL, ma elimina il vincolo per cui ogni operatore temporale (X, U, F, G) dev'essere associato biunivocamente ad un quantificatore per i path (A, E).
In pratica, mentre in CTL dobbiamo scrivere AXϕ o EXϕ, in CTL* possiamo scrivere anche Xϕ.

Operatori temporali e quantificatori per i path possono quindi essere composti arbitrariamente.

In CTL* si distinguono due classi di formule:

formule sugli stati che vengono valutate sugli stati:

$$\phi ::= \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid \mathrm{A}\,[\alpha] \mid \mathrm{E}\,[\alpha]$$

dove p è una formula atomica e α una formula su path;

formule sui path che vengono valutate lungo i path:

$$\alpha ::= \phi \mid (\neg\alpha) \mid (\alpha \wedge \alpha) \mid (\alpha \mathrm{U} \alpha) \mid (\mathrm{G}\alpha) \mid (\mathrm{F}\alpha) \mid (\mathrm{X}\alpha)$$

dove ϕ è una formula su stati.

LTL e CTL come sottoinsiemi di CTL* Qualsiasi formula LTL α è equivalente alla formula CTL* A $[\alpha]$.

CTL è una parte di CTL* ottenuta restringendo le possibili formule sui path a:

$$\alpha ::= (\phi \mathrm{U} \phi) \mid (\mathrm{G}\phi) \mid (\mathrm{F}\phi) \mid (\mathrm{X}\phi)$$

Operatori per il passato [10]

Y	"Yesterday", analogo di X
S	"Since", analogo di U
O	"Once", analogo di F
H	"Historically", analogo di G

Per l'LTL non aggiungono espressività (possono essere ricavati dagli altri operatori), per il CTL sì, in quanto permettono di esaminare stati che non sono successivi all'attuale.

4.5 Algoritmi di Model Checking

Vogliamo risolvere il problema $\mathcal{M}, s \overset{?}{\models} \phi$. Vedremo solo il **Labeling Algorithm** per formule CTL.

Tale algoritmo accetta in ingresso un modello $\mathcal{M} = (S, \to, L)$ e una formula CTL φ, e da come output l'insieme degli stati di $\mathcal{M}$ che soddisfa φ.

Il primo passo consiste nel modificare φ in modo tale che usi solo gli operatori AF, EU, EX, $\wedge$, $\neg$, e $\bot$ usando le equivalenze date precedentemente. Quindi si procede ad etichettare con φ tutti gli stati di $\mathcal{M}$ in cui vale φ, applicando le seguenti regole:

- se $\varphi = \bot$ allora nessuno stato viene etichettato con $\bot$;
- se $\varphi = p$ allora vanno etichettati con φ tutti gli stati s tali che $p \in L(s)$;
- se $\varphi = \neg\psi$:
  - eseguire l'algoritmo di labeling per ψ;
  - etichettare con $\neg\psi$ tutti gli stati s non etichettati con ψ;
- se $\varphi = \psi_1 \wedge \psi_2$:
  - eseguire l'algoritmo di labeling per ψ_1 e ψ_2;
  - etichettare con $\psi_1 \wedge \psi_2$ tutti gli stati s già etichettati con ψ_1 e ψ_2;
- se $\varphi = \mathrm{EX}\psi$:
  - eseguire l'algoritmo di labeling per ψ;
  - per ogni stato, se almeno uno degli stati che gli succedono è etichettato con ψ, allora lo stato va etichettato con $\mathrm{EX}\psi$;
- se $\varphi = \mathrm{AF}\psi$:
  - eseguire l'algoritmo di labeling per ψ;
  - ogni stato etichettato con ψ va etichettato anche con $\mathrm{AF}\psi$;
  - per ogni stato, se tutti gli stati che gli succedono sono etichettati con $\mathrm{AF}\psi$, allora anche lo stato stesso va etichettato con $\mathrm{AF}\psi$.
    Rieseguire questo passo finché una nuova iterata non aggiunge nuove etichette;
- se $\varphi = \mathrm{E}[\psi_1 \mathrm{U} \psi_2]$:

[10] NuSMV li supporta solo per le formule LTL.

- eseguire l'algoritmo di labeling per ψ_1 e ψ_2;
- etichettare con $\mathrm{E}[\psi_1 \mathrm{U} \psi_2]$ ogni stato etichettato con ψ_2;
- per ogni stato, se è etichettato con ψ_1 e almeno uno degli stati che gli succedono è etichettato con $\mathrm{E}[\psi_1 \mathrm{U} \psi_2]$, allora anche lo stato stesso va etichettato con $\mathrm{E}[\psi_1 \mathrm{U} \psi_2]$.
  Rieseguire questo passo finché una nuova iterata non aggiunge nuove etichette.

Ovviamente l'algoritmo restituisce tutti gli stati etichettati con φ.

La complessità dell'algoritmo è $\mathrm{O}(f \cdot V \cdot (V + E))$, dove f è il numero di operatori nella formula, V è il numero di stati del modello ed E il numero di transizioni del modello. Qualche miglioramento può essere ottenuto gestendo direttamente anche l'operatore EG.

L'algoritmo è lineare rispetto alle dimensioni sia di φ che di $\mathcal{M}$, tuttavia la dimensione di quest'ultimo è molto spesso esponenziale rispetto al numero di variabili e al munero di componenti del sistema che eseguono in parallelo; questo problema è noto come il **problema di esplosione degli stati**.

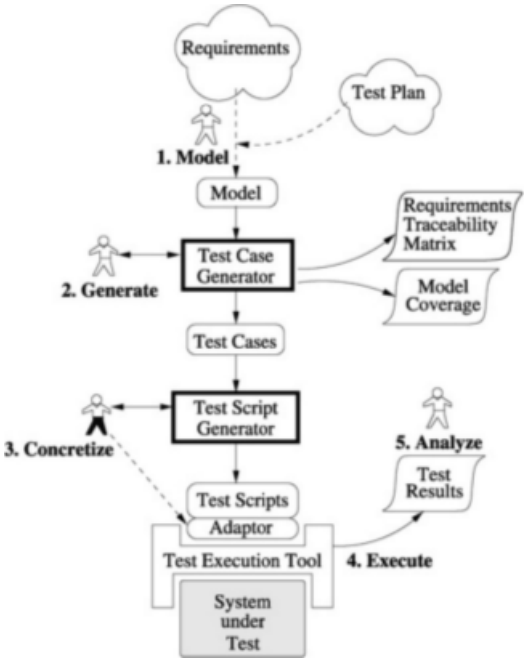
5 Testing Model-based

Quattro approcci (o campi applicativi) principali:

- generazione di dati di input per un test da un modello del dominio corrispondente;
- generazione di casi di test da un modello dell'ambiente;
- generazione di casi di test e di oracoli da un modello comportamentale;
- generazione di script di test (eseguibili) da test astratti.

Passi del processo di testing model-based

1. creazione del modello del SUT e/o del suo ambiente a partire dai suoi requisiti e dal piano di test:
   - il modello dev'essere astratto, non esaustivo ma finalizzato al testing;
   - verifica della consistenza del modello e che modelli davvero il comportamento desiderato;
2. generazione di casi di test astratti dal modello, in base a determinati criteri di copertura (output aggiuntivi di questo passo sono una matrice di tracciabilità dei requisiti e un indice di copertura del modello);
3. concretizzazione dei casi di test astratti al fine di renderli eseguibili; quest'operazione difficilmente può essere completamente automatizzata, richiede spesso la scrittura di codice di adattamento da parte di un programmatore;
4. esecuzione dei test sul SUT e assegnazione di verdetti di successo/fallimento:
   - testing model-based online: i casi di test vengono eseguiti subito dopo esser stati generati; il tool che li genera è strettamente collegato al SUT, e gestisce l'esecuzione e la registrazione dei risultati;
   - testing model-based offline: il tool che genera casi di test non ne gestisce l'esecuzione, quindi sarà necessario usarne un'altro per l'esecuzione; l'esecuzione dei test è completamente svincolata dal processo di generazione dei test.
5. analisi dei risultati dei test.



Modellazione del SUT Ovviamente può essere testato solo ciò che viene modellato. Per questo motivo per la creazione del modello del SUT è fondamentale la scelta del livello di astrazione. Spesso piuttosto che creare un unico modello per tutto il sistema risulta più utile creare modelli parziali di parti del sistema.

Semplificazioni (o astrazioni) tipiche:

- includere solo le operazioni che si vogliono testare;
- includere solo i dati che sono utili per modellare il comportamento delle operazioni da testare;
- sostituire semplici enumerazioni a campi di dati complessi o a classi.

Cosa modellare:

- solo gli input e/o l'ambiente del programma → testing combinatoriale;
- anche il comportamento → testing FSM-based.

5.1 Partition testing e testing combinatoriale

5.1.1 Testing combinatoriale

Si basa sulle interfacce. Non viene considerata nessuna informazione riguardo il SUT ma solo il suo spazio degli input.

Può essere sia model-based che program-based.

Vantaggi

- Può essere applicato a livelli diversi, cioè può essere usato come test di unità, integrazione e/o sistema;
- relativamente semplice da applicare, in quanto non considera la struttura del SUT;
- facilmente adattabile e modificabile al fine di ottenere più o meno test;

- non richiede di conoscere l'implementazione del SUT (lo si può utilizzare quindi anche quanto non è disponibile l'intero codice/modello del SUT).

Efficacia Riesce a trovare errori che le tecniche tradizionali di test possono non essere in grado di trovare.

Efficienza Un basso grado di interazione tra gli input permette di trovare la maggior parte degli errori. Solitamente li si considera a coppie; in ogni caso il grado di interazione è sempre minore di 6.

5.1.2 Partition testing

Spesso anche per piccoli programmi il dominio degli input (insieme di tutti i possibili input del sistema) è così esteso che lo si può considerare infinito. Nasce quindi il problema di scegliere set finiti (e significativi) di valori dal dominio degli input.

Il partition testing risolve questo problema suddividendo ogni dominio in parti (partizionandolo appunto) e sostituendo al dominio stesso un enumerazione formata da rappresentanti delle parti in cui il dominio è stato suddiviso.

Ovviamente lo schema di partizionamento del dominio deve avere le tipiche proprietà di un insieme delle parti (nessuna parte vuota, tutte le parti sono disgiunte a coppie e la loro unione deve dare come risultato tutto il dominio).

Da ogni partizione viene scelto un rappresentante che si assume essere ugualmente efficace per il testing, e si scelgono i casi di test combinando questi rappresentanti.

Problemi

- È possibile che vi siano molti sotto-domini in una singola partizione, il che fa crescere il numero di casi di test;
- non fornisce linee guida su come selezionare gli input dai sotto-domini all'interno delle partizioni.

5.1.3 Dal partition testing al combinatorial testing[1]

1. Identificazione delle funzioni da testare (intese come "cose che il SUT dovrà fare"): ciò implica che come input va considerato anche l'ambiente in cui il software lavora (hardware, software con cui collabora, ...);
2. indentificazione degli input: operazione quasi meccanica, ma è importante che tutti gli input vengano identificati (quelli scritti sopra per esempio, ma anche file, ...);
3. modellazione del dominio degli input: comprende la suddivisione in partizioni e l'identificazione di valori di test per ogni parametro (solitamente si scelgono valori validi, valori invalidi e valori particolari come i limiti del dominio), con particolare attenzione a che i test siano completi ma cercando di evitare che il numero di valori da testare cresca troppo e/o inutilmente (approcci a questo problema sono descritti più nel dettaglio più avanti);
4. applicazione di un criterio di test per la scelta di combinazioni di valori per cui eseguire i test.

[1] Cioè dal partizionamento del dominio degli input al testing combinatoriale.

5.1.4 Approcci all'Input Domain Modeling (IDM)

Interface-based Considera direttamente i valori individuali dei parametri. È semplice da applicare e in certe situazioni può essere parzialmente automatizzato.

Alcune informazioni su dominio e semantica degli input non vengono considerate; ciò può portare ad un modello incompleto. Ignora anche le relazioni che ci sono tra i parametri.

Functionality-based Modella i domini da una descrizione del comportamento del SUT. È più difficile da sviluppare e richiede un maggiore sforzo di progetto, ma dà come risultato test migliori, o un numero minore di test che ha però la stessa efficacia.

Può basarsi direttamente sui requisiti del SUT e non sulla sua implementazione.

Valori particolari di uno stesso parametro, o combinazioni di valori particolari di determinati parametri, possono influenzare molte caratteristiche del SUT, per cui è più difficile tradurre i valori in casi di test (es.: se un oggetto ha quattro parametri, una combinazione particolare di valori dei parametri può conferire una proprietà "aggiuntiva" all'oggetto).

5.1.5 Criteri di scelta delle combinazioni di valori

All Combination Coverage (ACoC) Vengono testate tutte le combinazioni di partizioni di tutti i parametri.

Numero di test da eseguire: $\prod_{i=1}^{P} p_i$ dove P indica il numero di parametri e p_i il numero di partizioni del dominio del parametro i-esimo.

Each Choice Coverage (ECC) Un valore da ogni partizione di ogni parametro dev'essere usato in almeno un caso di test.

Numero di test da eseguire: $\max(p_i)$ con $i = 1, \ldots, P$.

Pair-Wise Coverage (PWC) Un valore da ogni partizione di ogni parametro dev'essere combinato con un valore da ogni partizione di ogni altro parametro.

Numero di test da eseguire: $\max(p_i) \cdot \max(p_j)$ con $i, j = 1, \ldots, P$ e $i \neq j$.

È abbastanza efficace nel trovare difetti, pur riducendo notevolmente il numero di casi di test rispetto al criterio ACoC.

T-Wise Coverage (TWC) Estende il concetto del PWC a t-uple di valori di parametri. È più costoso del PWC ma apparentemente non ne migliora l'efficacia.

Base Choice Coverage (BCC) Una partizione di base viene scelta per ogni parametro. Un test base viene creato usando le partizioni base di ogni parametro. I test successivi vengono creati testando, per un parametro alla volta, tutte le partizioni diverse da quella base (combinata quindi con quelle base di tutti gli altri parametri).

Multiple Base Choices Coverage (MBCC) Identico al precedente con l'unica differenza che per ogni parametro può essere scelta più di una partizione base.

5.1.6 Tecniche di generazione di casi di test

Metodi algebrici Si basano su proprietà matematiche e/o algebriche.

Sono molto veloci e spesso producono risultati ottimali, ma hanno un'applicabilità limitata e difficilmente supportano vincoli sui parametri e relazioni tra essi.

Esempi: quadrati latini, costruzioni ricorsive, ...

Metodi logici Basati su verifiche di soddisfacibilità di proprietà e model checking.

Metodi search-based (greedy) Basati sull'euristica.
Sono molto flessibili e non impongono restrizioni sugli input, ma sono più lenti e i test set che danno come output non sono ottimali. Due tipologie:

- parameter-based: aggiungono un parametro alla volta.
  Fa parte di questa categoria l'IPO (In-Parameter-Order) (usa il criterio TWC), che consta dei seguenti passi:

  1. si costruiscono casi di test per i primi t parametri considerandone semplicemente le combinazioni;
  2. si aggiungono, una alla volta, gli altri parametri ai casi di test;
  3. se necessario, si aggiungono casi di test per coprire le t-uple mancanti.

  Non sono ancora stati decisi/trovati criteri convenienti per la scelta dei valori dei parametri quando si aggiungono colonne (parametri) e/o righe (casi di test).

- test-case-based: aggiunge un caso di test alla volta.

5.2 Testing basato su specifiche

1. I criteri di test sono definiti a partire dalla specifica:
   - dati di test derivati dalle specifiche;
   - si sceglie quali funzioni testare, e quindi quanto del SUT viene coperto;
2. la specifica viene usata come oracolo: "applico" a programma e specifica gli stessi input e verifico che diano lo stesso output:
   - **conformance testing**: nel caso di specifiche eseguibili, eseguo i casi di test sulle specifiche e sui programmi e confronto i risultati, testando così la conformità del programma alla sua specifica.

Notazioni di specifica del sistema: diagrammi UML (macchine a stati, diagrammi di interazione), ASM, Simulink, ... noi useremo le FSM (di Mealy).

5.2.1 Macchine a stati finiti (FSM)

Sia FSM (S, I, O) una FSM con output, dove S è l'insieme degli stati, I l'insieme degli input e O l'insieme degli output.

- Macchina di Mealy: FSM che produce un output per ciascuna transizione;
- macchina di Moore: FSM che produce un output per ciascuno stato.

Definizione formale: una macchina di Mealy è una sestupla, $\{S, S_0, I, O, T, G\}$ dove:

- S è un insieme finito di stati;
- S_0 è lo stato iniziale ($S_0 \in S$);
- I è l'insieme finito dei possibili input;

- O è l'insieme finito dei possibili output;
- T è una funzione di transizione, $T : S \times I \to S$ che fa corrispondere a coppie stato-input un altro stato (lo stato d'arrivo della transizione);
- G è una funzione di output, $G : S \times I \to O$ che fa corrispondere a coppie stato-input un output.

Graficamente si rappresenta come una FSM con etichette del tipo "i/o" sulle transizioni. Quindi ogni transizione è una tupla (s, i, o, s') dove s è lo stato di partenza, i l'input che scatena la transizione, o l'output generato dalla transizione e s' lo stato d'arrivo.

Gli output possono anche rappresentare azioni eseguite dalla macchina.

Limite delle FSM Numero finito di stati rappresentabili: la crescita del numero di stati di una FSM ottenuta come composizione di altre FSM è esponenziale nel numero delle FSM composte.

Per superare questo limite sono state definite delle varianti (tra cui Statecharts di UML) che introducono il concetto di sottomacchina e permettono:

- composizione sequenziale;
- composizione parallela;
- modularità.

D'ora in poi FSM = macchina di Mealy.

5.2.2 Conformance testing con FSM

Scenario:

- la specifica S del SUT è una FSM nota;
- il SUT implementato I da testare è una FSM non nota di cui si può solo osservare l'output applicando certi input.

Obiettivo: determinare se I è un'implementazione corretta di S.

Passaggi:

1. analizzare la FSM della specifica;
2. trovare i casi di test;
3. applicare i casi di test alla specifica;
4. applicare i casi di test al programma;
5. confrontare gli output.

Vedremo come poter ottenere i casi di test dalla FSM della specifica algoritmicamente.

Utilità

- Le FSMs sono spesso usate direttamente per specificare protocolli, sistemi di controllo embedded e circuiti digitali;
- molte altre notazione sono simili alle FSM;
- spesso è utilizzabile anche per l'astrazione soltanto di una parte del sistema (che magari si vuole testare, evitando di dover gestire la complessità di un modello di tutto il sistema).

I metodi di test che vedremo sono ideali, cioè provano l'assenza di difetti. Sono validi però solo sotto ipotesi precise e abbastanza forti, anche se nella pratica hanno dimostrato di essere efficaci anche quando le ipotesi non valgono.

Assunzioni:

- I e S deterministiche, inizializzate e completamente specificate;
- dev'essere possibile raggiungere ogni stato (S fortemente connessa);
- non possono esistere stati equivalenti (S minimizzata);
- I non varia durante l'esecuzione;
- I ha lo stesso alfabeto di S;
- I non ha più stati di S.

Difetti considerati:

- difetto di output (viene generato l'output sbagliato);
- difetto di trasferimento (una transizione non termina nello stato corretto).

Non vegono invece considerati: stato extra, stato mancante.

Messaggio di reset È un input particolare r che se applicato alla macchina la riporta al suo stato iniziale S_0 da qualsiasi altro stato.

Se non esiste un tale messaggio al suo posto si una una homing sequence, cioè una sequenza di transizioni che hanno come ultimo stato di destinazione S_0.

Permette di trasformare un test set in un'unica test sequence.

Messaggio di status È un messaggio che mostra in output lo stato corrente della macchina senza cambiarlo. In pratica per ogni stato s avrò una transizione $\{s, status, state, s\}$ (che in ordine sono: partenza, input, state, arrivo).

I metodi che vedremo generano un test set che spesso è formato da una sola sequenza di test (test sequence).

Metodi

Differiscono tra loro per le seguenti caratteristiche:

- applicabilità: alcuni richiedono reset, status, o altre sequenze;
- capacità a trovare i difetti;
- lunghezza dei casi di test e complessità di calcolo.

Copertura

- Degli stati: un test set T è adeguato secondo la copertura degli stati di una FSM M se l'esecuzione da parte di M di tutte le sequenze di T causa la visita di ogni stato di M.
  Non garantisce la scoperta di alcun difetto (soprattutto di quelli sulle transizioni).
- delle transizioni: un test set T è adeguato secondo la copertura delle transizioni di una FSM M se l'esecuzione da parte di M di tutte le sequenze di T causa la visita di ogni transizione di M.

Metodo Transition Tour Un **Transition Tour** (TT) è una sequenza di input che applicata a M nel suo stato iniziale attraversa tutte le transizioni di M almeno una volta (e ritorna allo stato iniziale).

Tour Euleriano: è il TT più corto. Per macchine simmetriche (tanti archi uscenti quanti entranti) è semplice da trovare (tempo lineare). Per macchine non simmetriche è più complesso (tempo polinomiale).

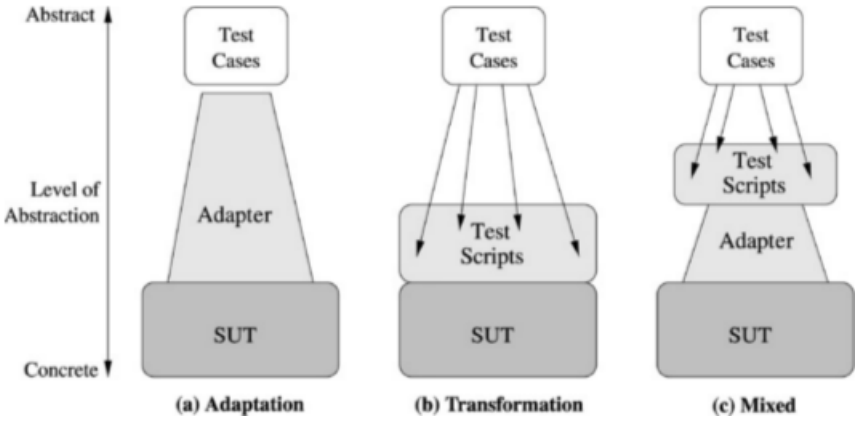
Il metodo Transition Tour prevede che il test set abbia una sola sequenza, che dev'essere un Transition Tour. Due varianti:

- con status message: applico status dopo ogni messaggio, verificando lo stato; permette di scoprire tutti i difetti (output e trasferimento);
- senza status message: garantisce la scoperta solo dei difetti di output; potrei comunque riuscire a capire in quale stato mi trovo applicando input aggiuntivi e osservando gli output (chiaramente non potrò quindi usare il Tour Euleriano)

5.3 Concretizzazione dei test

È una fase importante del processo di testing. Richiede spesso uno sforzo significativo e a volte lo svolgimento di attività da parte di un programmatore.

5.3.1 Approcci alla concretizzazione dei test



Adaptation Prevede la scrittura a mano di codice di adattamento che colmi il divario tra casi di test astratti e SUT. In pratica crea un wrapper attorno al SUT che ne fornisce una visione più astratta, che i casi di test astratti riescono a comprendere. Passi:

- setup: inizializzazione del SUT in modo che sia pronto per il testing. Porta il SUT nello scenario di applicazione del test;

- concretizzazione: traduzione di ogni operazione astratta a livello di modello in una o più operazioni concrete del SUT con appropriati valori di input;
- astrazione: estrazione e astrazione dei risultati dalle operazioni concrete svolte e comparazione coi risultati previsti;
- teardown: reset del SUT, da eseguire alla fine di ogni test sequence o di ogni insieme di test.

Trasformation Prevede la trasformazione dei test astratti in script di test eseguibili.

I test script risultanti possono essere scritti in un linguaggio di programmazione "standard", in un linguaggio di scripting (TCL, JavaScript, ...) o in una notazione di test (standard o proprietaria; una delle più usate è la TTCN-3).

Sono necessarie la scrittura di codice di setup e di teardown, di template delle operazioni (non semre banale a causa della mappattura non uno-a-uno delle operazioni del SUT con quelle del modello) e la concretizzazione dei valori dei parametri del modello in valori dei parametri del SUT.

In caso di SUT non deterministico, il test astratto potrebbe avere una struttura ad albero. In questo caso il tool di trasformazione dev'essere in grado di gestire tali tipi di test.

Oltre a tenere traccia della mappatura tra test astratti e test script eseguibili, spesso è desiderabile scrivere anche un piano di test che descriva la struttura del test set, le impostazioni usate per generarli ecc, in quanto i test script generati potrebbero essere usati anche per testare altro codice e/o altri SUT con uno sforzo di adattamento minimo.

Mixed Mix dei due approcci precedenti.

Solitamente per l'online testing è meglio l'approccio Adaptation, in quanto richiede un maggiore accoppiamento tra SUT e tool di testing.

Per l'offline testing vanno bene entrambi, e quindi spesso è preferibile utilizzare l'approccio Trasformation, in quanto presenta alcuni vantaggi (meno frammentazione dei test), oppure un mix dei due.

5.4 Testing FSM con ModelJUnit

Permette di scrivere un modello FSM direttamente in Java.

Per predisporre il progetto scaricare e includere nel build path del progetto il jar scaricabile dal seguente link: `http://sourceforge.net/projects/modeljunit/`.

La classe che fa da modello FSM deve implementare l'interfaccia `FsmModel`, e quindi implementare i seguenti metodi:

- `Object getState()`: ritorna il valore visibile (astrazione) dello stato corrente (tipicamente una stringa);
- `void reset(boolean)`: per resettare la FSM al suo stato iniziale.

Per modellare le transizioni:

- `@Action public void action_i()`: modifica lo stato della FSM;
- `public boolean action_iGuard()`: viene chiamata per verificare se nello stato attuale può essere eseguita l'azione action_i (se non implementata l'azione può essere sempre eseguita).

Attenzione: ricordarsi di dichiararli `public`, altrimenti il tool potrebbe non riconoscere i metodi.

Online testing Scrivere un caso di test Junit che testa la classe. Esempio:

```
Tester tester = new GreedyTester(new FSM());
tester.buildGraph();
CoverageMetric trCoverage = new TransitionCoverage();
tester.addCoverageMetric(trCoverage);
tester.generate(20);
tester.getModel().printMessage(...); // per stampare informazioni aggiuntive
```

Si possono quindi modificare la tecnica di generazione dei casi di test (greedy, casuale, ...) e i criteri di copertura.

Il tool mostra, per ogni passo, la tupla (`getState()`, `action_i`, `getState()`), permettendo di rilevare difetti di trasferimento (e anche di output, in base a cosa si fa mostrare al metodo `getState()`).

Offline testing Mostra graficamente la FSM e dà la possibilità di creare manualmente le sequenze di test oppure generarne automaticamente specificando il criterio e il numero di passi.

Passi da seguire:

- in eclipse, esportare il jar del progetto che contiene la classe da testare all'interno della cartella del progetto stesso;
- copiare il file modeljunit-2.5-jar-with-dependencies.jar all'interno della cartella del progetto;
- eseguire il comando `java -jar modeljunit-2.5-jar-with-dependencies.jar`;
- creare un nuovo progetto, specificando come jar quello precedentemente esportato e come nome della classe il nome della classe da testare;
- se al passo precedente da errore (probabilmente perché aprendo il jar prende il percorso assoluto, ma poi ragiona come se fosse relativo):
  - salvare il progetto nella cartella del progetto di eclipse;
  - aprire il file appena salvato (*.mju) con un editor di testo e per l'attributo packageLocation cancellare il percorso del file, lasciando solo "file:" e il nome del jar (con l'estensione).