Testing & Verifica del Software

Appunti

Galizzi Francesco, Centurelli Valentina

14 febbraio 2017

Indice

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

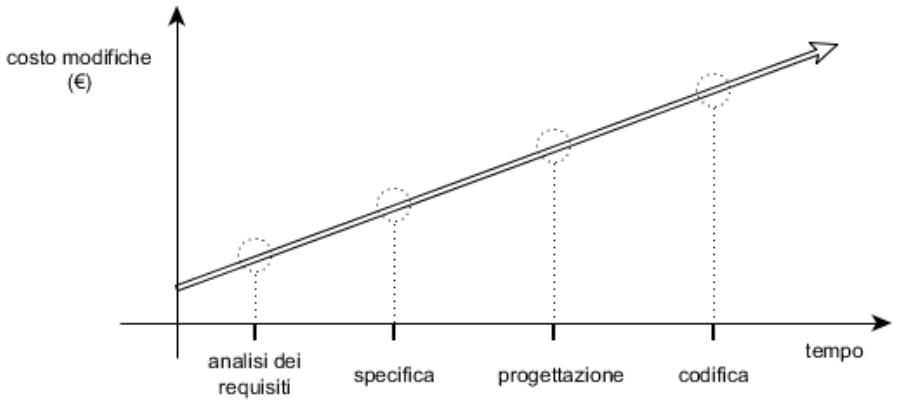
Revisioni

Data	Autore	Modifiche
10 giugno 2014	Galizzi Francesco	Stesura iniziale
12 giugno 2015	Centurelli Valentina	Arricchimento del documento

1 Introduzione

Obiettivi del testing e della verifica del software:

- valutare le **qualità** del programma: oltre a funzionare deve essere utile a qualcuno;
- rendere possibile il miglioramento del software trovandone i **difetti**: prima si trovano i difetti meno perdite economiche ci saranno.
  Per esprimere meglio questo concetto, si mostra l'aumento dei costi di modifica relativi a quando si trova il difetto:



1.1 Concetti generali

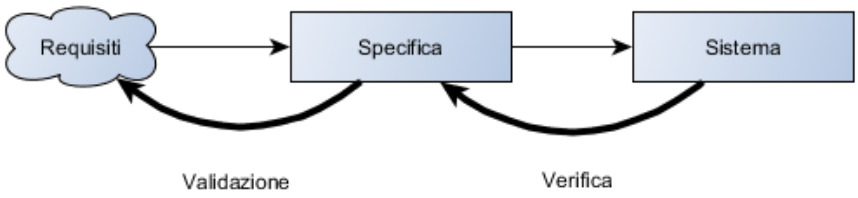
1.1.1 Validazione e verifica

requisiti cosa vuole l'utente;

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 (cioè 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.

Esempio: *l'impossibilità del testing esaustivo.*

Sia la seguente funzione:

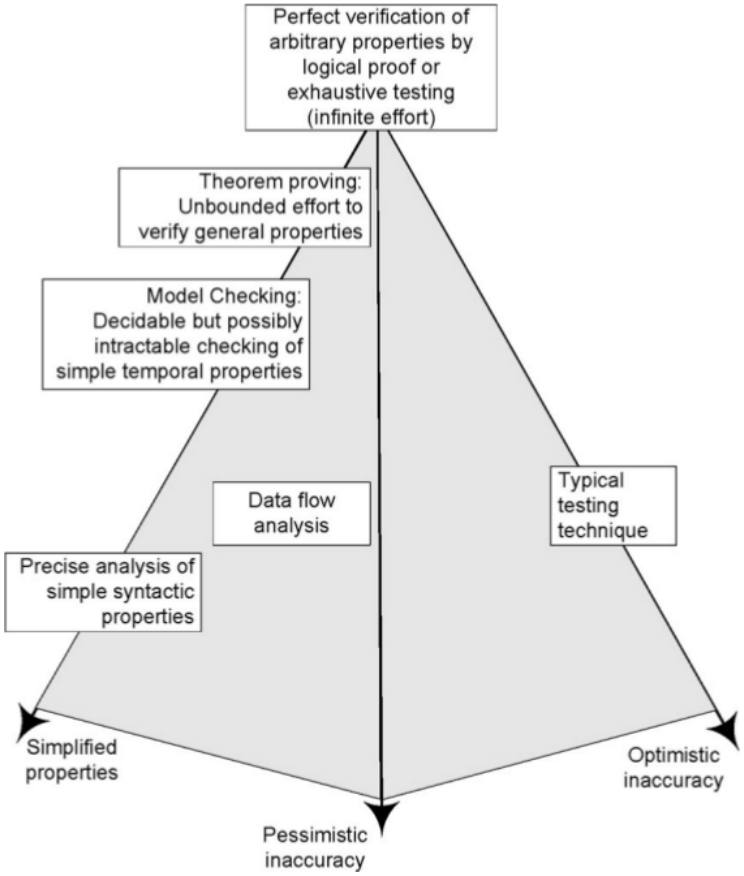
```
static int sum(int a, int b) { return a+b };
```

Questa funzione ha solo un'operazione molto semplice, però se si volessero testare tutti i casi possibili si dovrebbe sommare ogni singolo numero intero:

```
static void testSum() {
  int somma = 0;
  for (int i = MIN_VALUE; i <= MAX_VALUE; i++) {
    for (int j = MIN_VALUE;j <= MAX_VALUE;j++) {
      somma = sum(i, j);
      // confronto dei risultati per vedere se il programma è giusto
    }
  }
}
```

Gli integer in Java sono formati da 32 bit, quindi si avrebbero $2^{32} * 2^{32} = 2^{64}$ casi di test!

Approssimazioni



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

inaccuratezza pessimistica non è garantita l'accettazione di un programma che possiede tutte le proprietà desiderate (non accettare programmi corretti). 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 non necessaria per S, e richiediamo che sia soddisfatta.
Es.: Ho x numero naturale e ho la proprietà $S = x > 0 \; e \; x \; pari$. Non riesco a fare un'analisi precisa su S però riesco a verificare $S' = x > 0$.

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"), quindi P soddisfa $F \Rightarrow P$ accettato;

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

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.

Esempio: *caratteristiche del testing*

Sia la seguente funzione:

```
static int div(int a, int b) { return a/b; }
```

Essendo **dinamico**, il software viene eseguito, per esempio attraverso la funzione `testDiv`:

```
static void testDiv(int[] x, int[] y) {
  for (int i = 0; i < x.length ; i++) {
    for (int j = 0 ; j < y.length ; j++) {
      System.out.print( x[i] + " / " + y[j] + " = " + div(x[i], y[j]) );
    }
  }
}
```

Da notare che rispetto all'esempio precedente non si sono testati tutti gli interi ma solo determinati valori all'interno di due array: il testing è **finito**, quindi gli input sono pochi e ben definiti.

Se prendiamo come input $x = \{8\,;\,16\}$ $y = \{4\,;\,1\}$ otterremo come output:

```
8 / 4 = 2
8 / 1 = 8
16 / 4 = 4
16 / 1 = 16
```

Il testing serve per trovare errori, quindi non basta testare i casi più frequenti e semplici (come sopra, in cui si sono provati solo numeri positivi).
Servono dei **criteri** ben precisi per scegliere gli input, per esempio "si deve testare il programma in risposta all'inserimento di un valore nullo, negativo, uguale a zero o che non è un numero"$\Rightarrow$ $x = \{NULL\,;\,-16\,;\,0\}$ $y = \{4\,;\,0\,;\,'a'\}$
Infine, si deve avere la possibilità di "giudicare" il programma (il testing ha un comportamento **previsto**), quindi si modifica il codice come segue:

```
static int testDiv(int[] x, int[] y) {
  int temp = 0;
  for (int i = 0; i < x.length ; i++) {
    for (int j = 0 ; j < y.length ; j++) {
      temp = div(x[i], y[j]);
      System.out.print( x[i] + " / " + y[j] + " = " + div(x[i], y[j]) );
      System.out.println( "Confronto con " + (x[i]/y[j]) );
      if ( temp != (x[i]/y[j]) )
        return -1;
    }
  }
  return 0;
}
```

In pratica, se si trova anche solo un valore diverso da quello aspettato viene ritornato un codice di errore (-1 in questo caso).
Da notare che questo controllo è molto semplice dato che non ritorna il valore dell'input che ha rilasciato un valore diverso da quello attteso.

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 ed è costoso ripeterlo.

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. Una soluzione parziale al problema è associare ad ogni componente un identificativo così che il modulo di *capture* non si basi sulle coordinate/nome dell'oggetto.

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.

Lo script utilizza delle API per controllare ed osservare il sistema.

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.

Esempio: *testing program-based*

Sia la funzione `m` nella classe `Pippo`

```
class Pippo {
  ...
  m(···) {
    ...
  }
}
```

Si testerà con la classe `testPippo`:

```
static void testPippo() {
  test() {
    Pippo p = new p;
    p.m(···);
    }
  }
}
```

1.2.3 Processo di test model-based

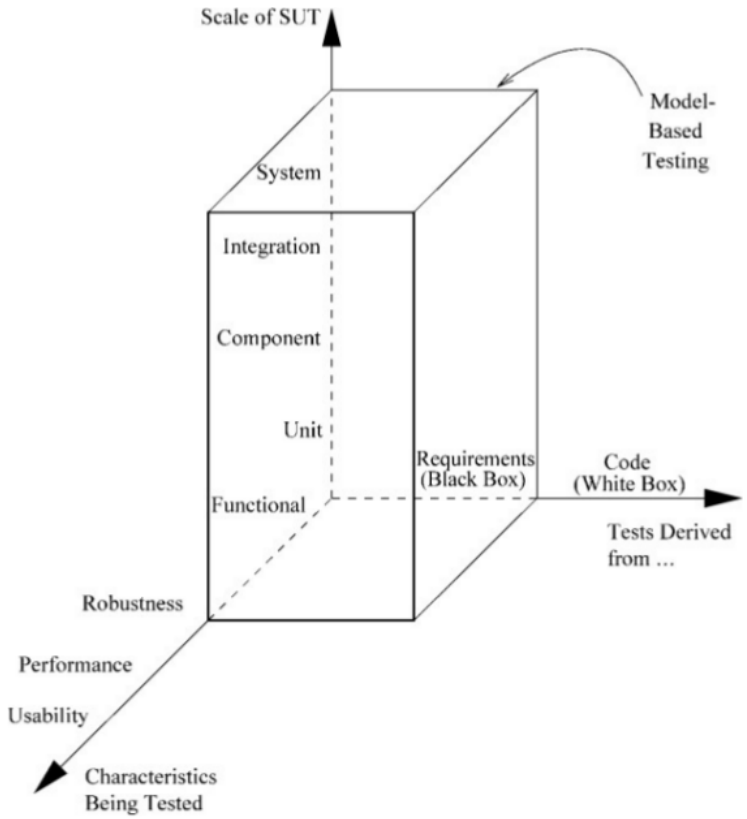
Il model-based testing parte dal presupposto di poter automatizzare la creazione di casi di test.

Si vedrà in seguito come classificare i processi di testing, per il momento si farà una breve descrizione delle caratteristiche di questo processo:

test blackbox i casi di test derivano dai requisiti del sistema, quindi non serve guardare il codice

test di robustezza è la caratteristica testata, cioè la "resistenza" del sistema quando si fanno azioni scorrette. Es: immissione di input errati.

system test si testa tutto il sistema.



Passi necessari:

- **creare un modello del SUT** e/o del suo ambiente (Es.: macchina a stati finiti);
  Il modello è più semplice del sistema e si concentra sugli aspetti chiave da testare. Scritto il modello si devono fare delle animazioni per verificare che il modello abbia un comportamento conforme al sistema.
- **generare casi di test astratti** dal modello;
  I test vengono scelti con un criterio specifico e non sono ancora eseguibili essendo basati su un modello astratto e non sul vero sistema.
- **concretizzare (conversione) dei casi di test** per renderne possibile l'esecuzione;
- **eseguire** i test sul SUT e decretarne l'esito;
  Ci sono due tipologie di esecuzione di test sul sistema:
  - test online: i test sono eseguiti mentre si procede con la conversione
  - test offline: viene creato uno script per eseguire i test che potrà essere richiamato in un secondo momento
- **analizzare il risultato** dei test e applicazione di azioni correttive;

È vantaggioso perché spesso l'utente deve solo progettare il modello.

1.2.4 Sintesi dei processi di testing

Si presenta una tabella riassuntiva dei processi di testing:

Processo	Cosa risolve	Problematiche
Testing manuale	//	Design dei test manuale. Esecuzione e riesecuzione manuale. Molto costoso.
Testing capture & reply	Riduce i costi di riesecuzione dei test (riesecuzione automatizzata)	Design dei test ed esecuzione manuale. Fragile alle modifiche del software. Costi di manutenzione alti.
Testing script based e program-based	Riduce i costi di esecuzione e riesecuzione dei test (esecuzione automatizzata)	Costi di manutenzione alti.
Testng model-based	Riduce i costi di design dei test (progettazione automatizzata).	Problemi nella conversione dei test astratti (dipende dal programma utilizato e non dal progettista).

1.2.5 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 logiche;

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. Ogni meccanismo che controlla un sistema in esecuzione è una verifica runtime.

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 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.

Discontinuità 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.

Es: se si testa la resistenza di un ponte in un punto particolare si è certi che anche intorno a quel punto il ponte resisterà allo stesso modo. Non si può dire lo stesso di un software, infatti basterebbe un input leggermente diverso per cambiare tutto (vedi esempio successivo).

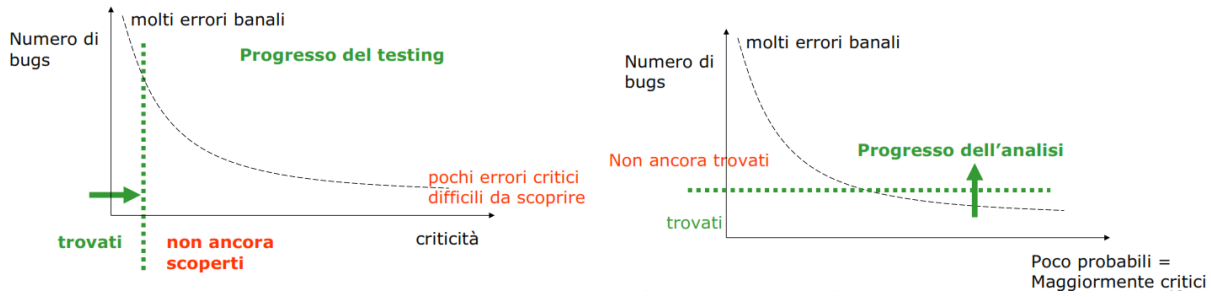
Esempio: *discontinuità del software*

Sia la seguente funzione:

```
static void provaDiscontinuita(float x) {
  float div;
  if (x == 3.99998)
    div = x / 0;
  else
    div = x / 1;
}
```

Nel punto $x = 3.99998$ verrà sollevata l'eccezione mentre nei punti vicini (Es: $x = 3.99997$; $x = 3.99999$) non si avranno problemi.

Difetti rilevabili 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 (grafico a sinistra).
Tecniche (più costose) di analisi statica invece cercano di scoprire difetti con probabilità proporzionale alla loro criticità (grafico a destra).



Nell'eseguire il testing occorre considerare quanto 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 software non critico viene spesso rilasciato con ancora dei difetti.

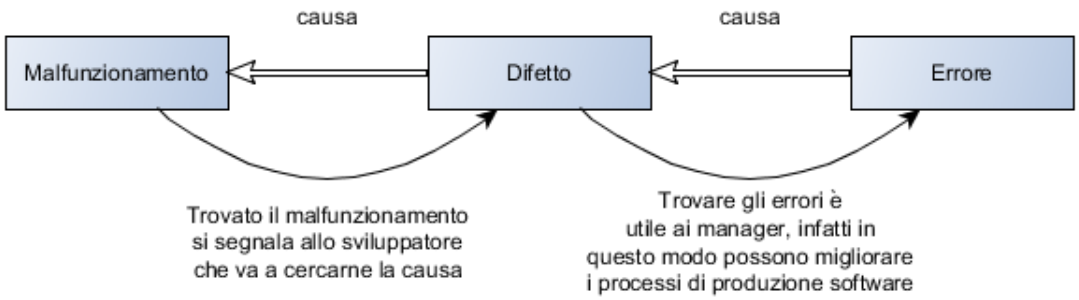
2.1.1 Terminologia

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

difetto (anomalia, fault o bug) elemento del programma sorgente non corrispondente alle aspettative (l'origine del comportamento errato); 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***.



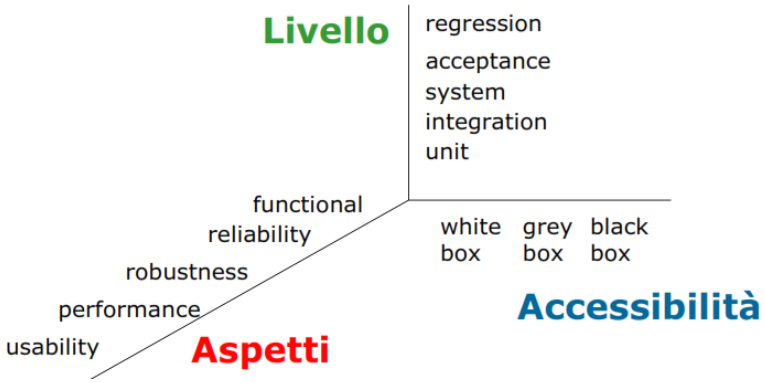
testing eseguire il programma con dei casi di test e analizzare i risultati per trovare i difetti (bug);

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*** (reliability) di un programma 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 Tipi di testing



2.1.3.1 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);
Es: controlla la compatibilità tra i dati scambiati tra i moduli e che non si facciano assunzioni (il modulo A passa solo interi positivi ma il modulo B "pensa" che siano interi negativi!)
Strategie:

top-down si parte ad integrare dai componenti ad alto livello e si testa il sistema ogni volta che si aggiunge un componente. Richiede lo sviluppo di stub (componenti software che simulano i moduli di basso livello);

bottom-up si parte ad integrare dai componenti di basso livello e si testa il sistema ogni volta che si aggiunge un componente. Richiede lo sviluppo di driver (componenti software che simulano i moduli di alto livello);

big-bang si unisce tutto e si testa il sistema globalmente (**sconsigliatissimo!!**);

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.

2.1.3.2 Accessibilità

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 ciò che è 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***[I] 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 attesi;
4. applica gli stessi input al programma e colleziona gli output osservati;
5. confronta gli output attesi con gli output osservati 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).

[I] ***Oracolo***: modo per stabilire se il test ha evidenziato un malfunzionamento oppure no.

Testing grey-box Mix tra i precedenti approcci.

2.1.3.3 Aspetti

test di funzionalità si cercano errori nelle funzionalità del sistema

test di affidabilità si controlla quanto il sistema va fuori uso

test di robustezza si cercano errori quando l'utente esegue comportamenti scorretti.
Es: inserire input errato

test di performance testa il sistema sotto condizioni di carico pesante

test di usabilità cerca i problemi nell'interfaccia utente

2.2 Fondamenti teorici

2.2.1 Definizioni base del testing

Un ***programma*** P è una funzione da un dominio D ad un codominio R, e può essere non definita per qualche $d \in D$: $P : D \to R$
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 $OK(P, d)\ \forall\, d \in D$ (corretto per ogni input inserito).

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;

caso di test elemento di D

test set (o test suite) sottoinsieme (finito) di D. (Es: T è una coppia di interi, D è l'insieme di tutte le coppie di interi quindi $T \subseteq D = INT \times INT$).

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 a S secondo il criterio C.
In generale $C : P \times S \times T \to \{true, false\}$ cioè dati programma, specifica e test set la funzione criterio dice se il test set è in grado di trovare tutti i difetti del programma ($true$) oppure no ($false$).

Un criterio di test può essere anche inteso come generatore di test set dato P e S, cioè $C : P \times S \to T$ tale che $C(P, S, T) = true$ (dati il programma e la specifica, creo un caso di test che rispetti il criterio).

Un criterio di test è ***affidabile*** se per ogni coppia di test set T_1 e T_2 adeguati secondo il criterio C ($C_A(T_i)$), se T_1 individua un malfunzionamento ($OK(P, T_1)$), allora ($\Leftrightarrow$) anche T_2 lo individua ($OK(P, T_2)$), e viceversa quindi $\forall T_1, T_2$

$$C_A(T_1) \wedge C_A(T_2)\, t.c.\, OK(P, T_1) \Leftrightarrow OK(P, T_2)$$

Da notare che non è detto che un criterio affidabile riesca a scoprire malfunzionamenti

Esempio: *criterio affidabile*

Sia la seguente funzione:

```
static int raddoppia(float x) { return x*x; }
```

Se ho $T_1 = 0$ e $T_2 = 2$ non trovo il malfunzionamento.

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

Si può anche scrivere $\exists t \in T\ t.c.\ C_{ps}(t) \wedge \neg OK(P, t)$.

Es: creazione causale di input per i casi di test.

Esempio: *criteri di un programma*

Un test set è adeguato secondo C_{ex} se contiene un numero positivo e uno negativo:
C_{ex} : *se* $\exists t_1 \in T\ t_1 > 0$ *and* $\exists t_2 \in T\ t_2 < 0$
Quindi un test set adeguato può essere: $\{-1; +1\}$ oppure $\{-1; -5; +6\}$.
Non importa il numero di test! Il criterio non si basa solo sul numero di argomenti.

Nel testing program-based C non dipende da S ($C : P \times T \rightarrow \{true, false\}$), mentre nel testing specification-based C non dipende da P ($C : S \times T \rightarrow \{true, false\}$).

Testing Program-based Il testing program-based verrà trattato nella sezione successiva, in questo paragrafo definiamo sinteticamente gli aspetti chiave.

Passi principali:

- esamina la struttura del programma (quali i punti critici, le decisioni importanti,...);
- trova i casi di test (cioè input) che soddisfano un certo criterio di copertura;
- applica gli input (uno alla volta) e osserva il programma (output) ;
- controlla che non si verifichino errori e che gli output siano quelli attesi.

Vantaggi:

- il codice è una importante fonte di informazione disponibile e usabile.

Limitazioni:

- non riesce a trovare errori di omissione.
  Es: cosa succede se un programma dimentica di gestire un caso particolare? Guardando solo la struttura tale caso non verrà mai selezionato come caso di test.
- non fornisce "test oracles". Es: come faccio sapere se l'output ottenuto è quello atteso?

Testing specification-based Passi principali:

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

Vantaggi:

- la specifica funziona da oracolo

Limiti:

- le specifiche potrebbero non essere disponibili – potrebbe esserci solo il codice
- le specifiche devono essere "formali" – per generare i casi di test – per eseguire i casi di test selezionati per ottenere gli output
- maggiore sforzo rispetto al program-based test

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 che soddisfa 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, si possono definire:

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

stopping rules regole che 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. Si abbandona l'idea che il test dimostra la correttezza del programma e si iniziano ad usare criteri empirici.

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.

$$C_A \to C_B \text{ se } \forall P\ C_A(P, T) \land C_B(P, T)$$

2.3 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.

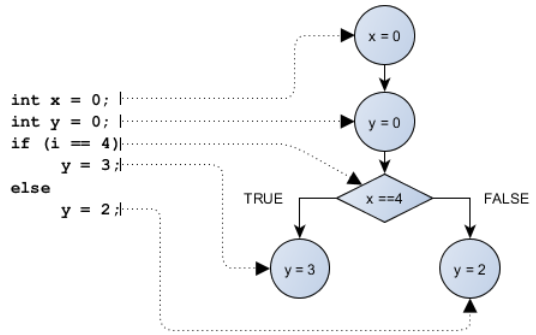
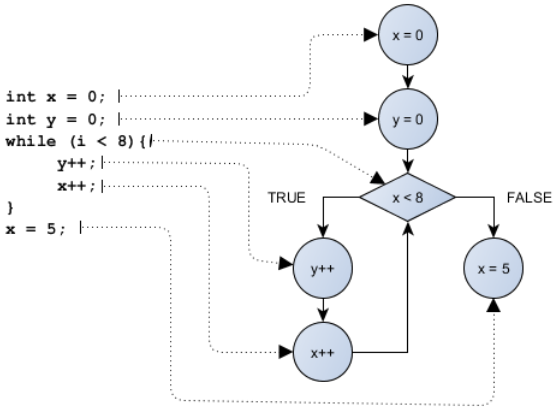
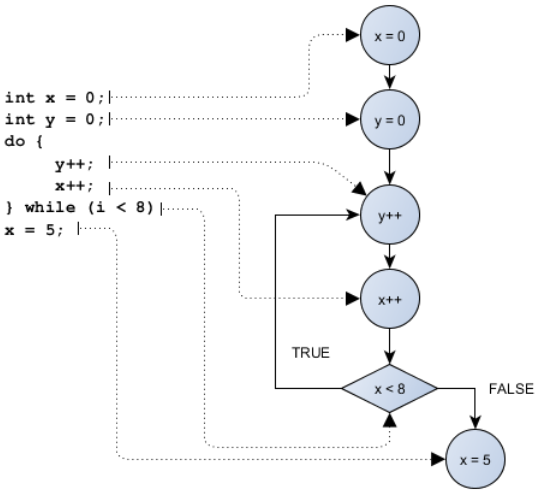
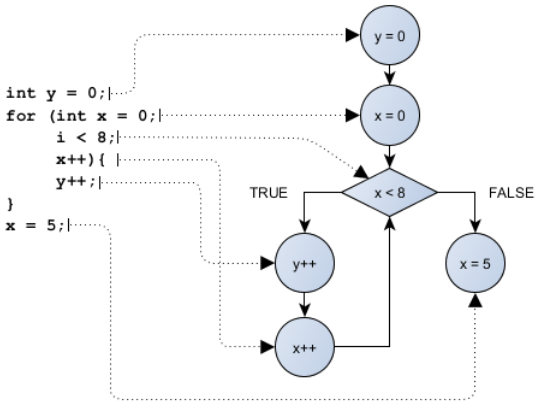
Il ***flusso di controllo*** di un programma può essere rappresentato tramite grafo:

- 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.

I cicli sono rappresentati nel modo seguente:

Istruzione if	Istruzione while
	
Istruzione do-while	**Istruzione for**
	

2.3.1 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.

Ci sono alcune istruzione che non possono essere eseguite contemporaneamente per un unico test (Es: risultati dell'istruzione IF), quindi si devono eseguire più test per coprirle tutte.

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):

$$C_{statement} = \frac{\#\, statement\, eseguiti}{\#\, statement\, eseguibili}$$

.

2.3.2 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 (ma non viceversa!).

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 "True" in una decisione aggiungi la decisione alla condizione;
  - se c'è un "False", aggiungi la negazione.

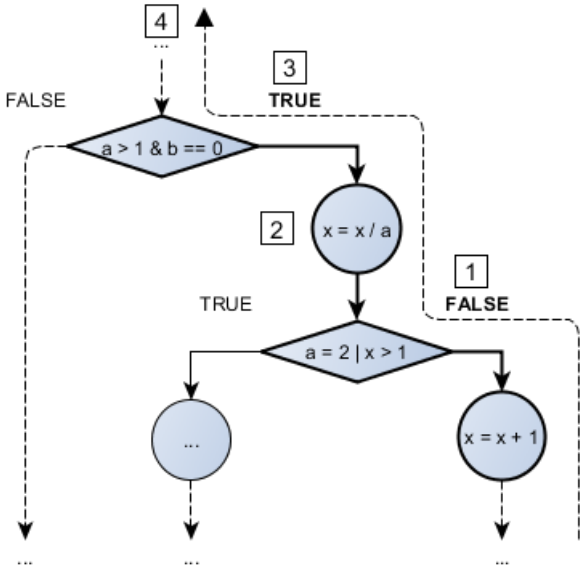
Potrebbe anche non essere risolvibile algoritmicamente, o addirittura non essere soddisfacibile (es.: $x > 0 \wedge x < 0$).

Esempio *copertura degli archi*

Ho il seguente codice, e vorrei eseguire l'istruzione `x = x + 1`

```
...
if (a > 1 & b == 0) {
  x = x / a;
  if (a = 2 / x > 1)
    ...
  else
    x = x + 1;
}
...
```

Si disegna il flusso di controllo del codice e si evidenziano le operazioni che si dovranno attraversare:



Si parte mettendo TRUE alla fine del cammino (`x = x + 1`) e si percorre il grafo al contrario. Si assegnano i valori che le variabili dovrebbero avere per rendere vera o falsa la condizione:

Passo [1] Alla prima condizione che incontriamo abbiamo $a = 2 \vee x > 1$ e per passare da quel ramo la condizione deve essere falsa, quindi si nega l'espressione

$$\neg(a = 2 \vee x > 1) \equiv \neg(a = 2) \wedge \neg(x > 1) \equiv a \neq 2 \wedge x \leq 1$$

Passo [2] C'è l'assegnamento $x = \frac{x}{a}$ quindi si sostituisce la x alla condizione del passo [1] ottenendo $a \neq 2 \wedge \frac{x}{a} \leq 1$

Passo [3] Si trova la condizione $a > 1 \wedge b = 0$ che deve essere vera, quindi non si applica nessuna negazione.

Passo [4] Si concatenando (con $\wedge$) le condizioni ottenute dai punti precedenti ottenendo

$$(a > 1 \wedge b = 0) \wedge \left(a \neq 2 \wedge \frac{x}{a} \leq 1\right)$$

2.3.3 Copertura delle decisioni (decision coverage)

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.

2.3.4 Copertura delle condizioni (condition coverage)

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.

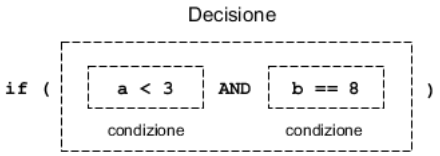
Una ***condizione*** è un'espressione booleana atomica (cioè non divisibile in altre espressioni più semplici) che appare in una decisione.

Copre quindi ogni condizione e la sua negazione.

2.3.5 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: una decisione è composta da condizioni, si possono coprire entrambi i rami di una decisione ma senza coprire tutti i casi delle condizioni e viceversa.



Se fissiamo `a = TRUE` e facciamo variare solo b otteniamo una copertura completa delle decisioni ma non delle condizioni!

a	b	a AND b
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE

Allo stesso modo, si può ottenere copertura della condizioni senza coprire le decisioni.

Se per esempio prendiamo `a < 3 OR b == 8`, possiamo prima porre `a = TRUE` e `b = FALSE` e poi `a = FALSE` e `b = TRUE`: si avrà una copertura completa delle condizioni, però si passerà sempre dal ramo di `TRUE` della decisione, quindi la copertura delle decisioni risulta incompleta.

a	b	a OR b
TRUE	FALSE	**TRUE**
FALSE	TRUE	**TRUE**

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 ||).

Un'altra soluzione è aggiungere più casi di test.

Esempio: *risoluzione del problema della short circuit evaluation*

Ho il seguente codice:

```
if (y == 0 || x > 0)
...
```

Se non ci fosse la short circuit evaluation basterebbe porre `y = TRUE` e `x = FALSE` e viceversa per avere una copertura completa delle condizioni.
Il problema è che `x = FALSE` non verrà valutata: dato da `y = TRUE` si sa già che il risultato della decisione è `TRUE`.
Quindi si aggiungerà `y = FALSE` e `x = FALSE` per poter valutare la seconda condizione:

y == 0	x > 0	y == 0 || x > 0
TRUE	non valutata	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

2.3.6 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.

Esempio: *Multiple Condition Coverage*

Ho il seguente codice:

```
if (a && b) || c
...
```

Con la MCC copro tutte le decisioni, quindi otterrò $2^n = 2^3 = 8$ casi di test.
Con la short circuit evaluation si possono "raggruppare" alcuni casi di test: quando si ha `a && b = TRUE` la condizione su `c` non verrà mai valutata, quindi è indifferente se `c = TRUE` oppure `c = FALSE`.

#	a	b	c
1	*TRUE*	*TRUE*	*TRUE*
	TRUE	*TRUE*	*FALSE*
2	TRUE	FALSE	TRUE
3	TRUE	FALSE	FALSE
4	FALSE	TRUE	TRUE
5	FALSE	TRUE	FALSE
6	FALSE	FALSE	TRUE
7	FALSE	FALSE	FALSE

2.3.7 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.

Esempio: *Modified Multiple Condition Coverage*

Ho il seguente codice:

```
if (peso > 250 || nPersone > 6)
...
```

Si "blocca" una condizione per testare i casi dell'altra: avendo una OR conviene porre una delle due condizioni a $FALSE$ in modo tale che sia l'altra a decidere del risultato finale.

peso > 250	nPersone > 6	peso > 250 || nPersone > 6
TRUE	**FALSE**	TRUE
FALSE	**FALSE**	FALSE
FALSE	TRUE	TRUE
~~**FALSE**~~	~~FALSE~~	~~FALSE~~

Nelle prime due righe si è bloccato `nPersone > 6` a falso per testare `peso > 250`.
Si nota che nella quarta riga si ha `peso > 250 = FALSE` e `nPersone > 6 = FALSE` che era già stato fatto nella seconda riga, quindi si può anche omettere.

2.3.8 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.

2.3.9 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.

2.3.10 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.

2.3.11 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_2 è il branch coverage).

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

2.3.12 Copertura ciclomatica

La ***complessità ciclomatica*** 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.

2.3.13 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).

2.3.14 Limiti della copertura

La copertura può essere pericolosa perché si può migliorare senza migliorare i test, e questo porta ad una falsa confidenza.

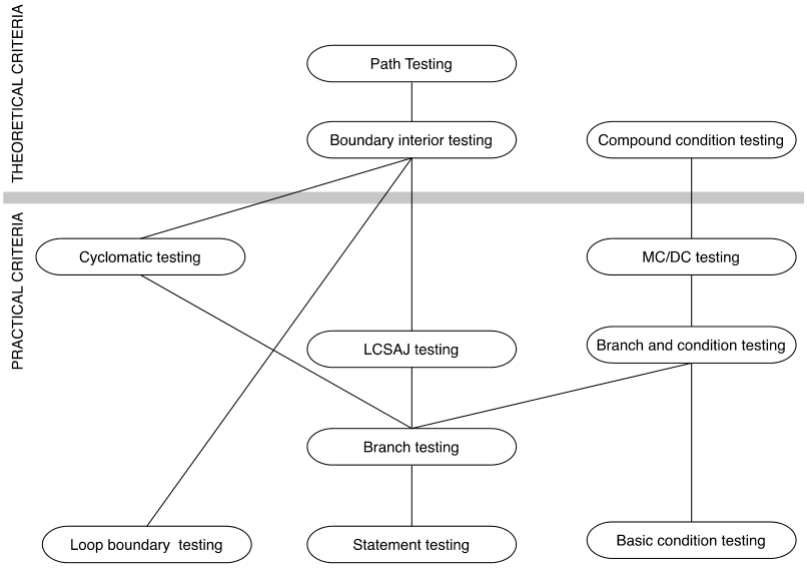
Esempio: *limiti della copertura*

Sia la seguente funzione:

```
static int abs(int x) {
  if (x == -2) return -2;
  else return x;
}
```

Ho la suite `abs(0) ; abs(2) ; abs(-2)` con copertura del 100% però il programma è sbagliato!

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 molte 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).

2.4 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 mantengono valido.

Non sempre il codice da testare ha buone caratteristiche di **controllabilità** (capacità di chiamare le funzionalità del programma) e **osservabilità** (capacità di catturare in modo semplice gli output. Es.: come catturo gli output quando testo l'interfaccia grafica di un programma?).
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.

2.5 Test di unità con JUnit e CodeCover

JUnit è un framework per l'automazione del testing di unità di programmi Java che si ispira all'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 (test driven development)

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 è un po' lento (si deve scrivere codice sia per l'applicazione che per i test), alla fine porta più vantaggi che svantaggi, infatti il codice è più affidabile anche se si aggiungono i costi di mantenimento dei casi di test.

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

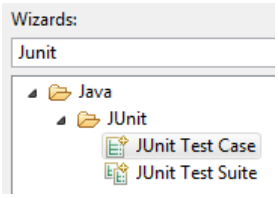
Per testare una classe X si crea una classe ausiliaria (in genere con nome XTest). XTest contiene dei metodi annotati con `@Test`, senza parametri e con void come tipo restituito, che rappresentano i casi di test. Per mantenere una buona separazione tra codice e test, è meglio avere tutti i test in un package separato.

Esempio: *test sulla classe* `Counter`

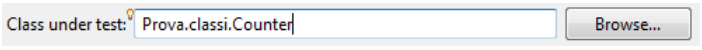
La classe ha questa struttura:

```
public class Counter {
  int conta;
  public Counter() { conta = 0; }
  public int inc() { return conta++; }
  public int dec() { return conta--; }
}
```

Si creerà un nuovo package (tasto destro su src → new → Package) chiamato `Test` in cui verrà creata la classe `CounterTest.`
Si può utilizzare JUnit per creare automaticamente lo scheletro della classe testante: tasto destro sul package `Test` → new → Other e nella finestra che comparirà scrivere nel filtro "Junit" e cliccare "JUnit Test Case".



Nella finestra successiva in basso si troverà la casella "Class Under Test", cliccare il bottone "Browse" per selezionare la classe che state testando.



Cliccare sul tasto "Next" e selezionare i metodi su cui fare il test, infine cliccare su "Finish".

Come detto prima, ogni test è di tipo `void` e preceduto da `@Test`.

```
package test;
import static org.junit.Assert.*;
import org.junit.Test;
public class CounterTest {
  @Test   public void testInc() { ... }
  @Test   public void testDec() { ... }
}
```

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`).

Esempio: *metodo assertEquals sul test* `testInc()`

Si testa il metodo visto nell'esempio precedente.
Come prima cosa si creerà l'oggetto della classe:

```
Counter c = new Counter();
```

Poi si andrà a chiamare il metodo `inc()` all'interno dell'asssert per controllarne il comportamento.

```
assertEquals(1, c.inc());
```

`c.inc()` restituirà l'incremento di conta (risultato attuale), dato che conta all'inizio è zero, se tutto va bene il metodo restituirà `1` (risultato atteso).

Da notare che è meglio creare un oggetto per ogni assert fatto, in modo da rendere ogni test indipendente: se ci fossero stati dei test in precedenza essi avrebbero potuto modificare il valore di conta, e quindi il risultato atteso non sarebbe più stato `1` ma si sarebbe dovuto contare l'effetto di altri test su esso, rendendo il design del test più difficile.
Il codice finale è quindi:

```
@Test public void testInc() {
  Counter c = new Counter();
  assertEquals(1, c.inc());
}
```

Esempio: *metodo* `assertTrue`

Il ragionamento è simile ad assertEquals, ma al posto di confrontare un risultato atteso e uno attuale si inserisce un'espressione booleana che deve risultare vera quando il test passa.

```
@Test public void testInc() {
  Counter c = new Counter();
  assertTrue(counter1.increment() == 1);
}
```

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 molti dati diversi; 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)`.

Esempio: *test parametrico*

Si vuole testare il metodo `incr` utilizzando i valori riportati in tabella.

```
static int incr(int x) { return x+1; }
```

Input	Valore atteso
0	1
2	3
5	6

Il concetto è quello di creare una lista di valori, in cui:

- le colonne rappresentano gli input e ouput dei test
- le righe i singoli casi di test

Si crea la lista di parametri:

```
@Parameters public static Collection creaParametri() {
  return Arrays.asList( new Object[][] { {0, 1}, {2, 3}, {5, 6} } } );
}
```

Dato che ad ogni riga è chiamato un nuovo caso di test, si "collegano" le colonne alle variabili utilizzate nei test:

```
private int input, outputAtteso;
public ParameterTest(int p1, int p2){
  input = p1;
  outputAtteso = p2;
}
```

Infine si scrive il caso di test:

```
@Test public void testParametrico(){
  int outputAttuale = incrementa(input);
  assertEquals(inputIncrementato,outputAttuale);
}
```

Dall'esempio si può vedere che il test parametrico fa i seguenti passi:

- chiama il metodo `getParameters` che deve essere statico dato che non sono state ancora create variabili
- cicla ogni riga della lista
  - chiama il costruttore del test e passa i valori di quella riga
  - chiama il meteodo del test e lo esegue

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.

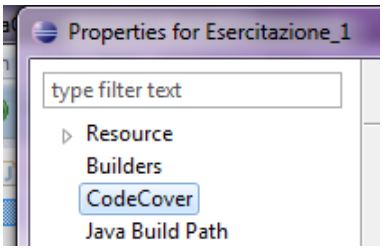
Utilizzo in Eclipse:

1. abilitare CodeCover nelle proprietà del progetto, ci sono due modi:

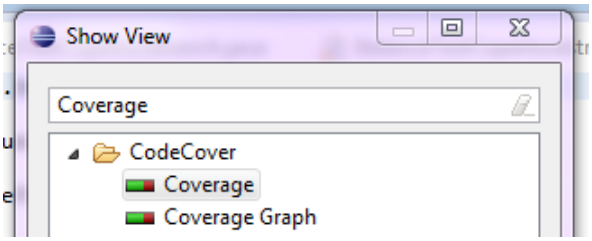
   a) Selezionare il progetto e premere ALT + ENTER

   b) Tasto destro sul Progetto → Proprieties

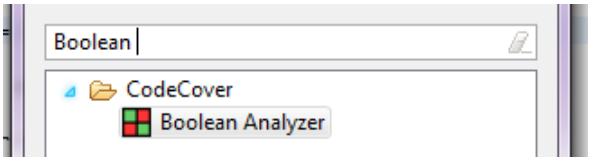
   In entrambi i casi si aprirà una finestra, selezionare nel menù a destra CodeCover e scegliere i criteri desiderati.



2. click destro sulla classe sotto test (nel package explorer di Eclipse) → selezionare "Use For Coverage Measurement";
3. click destro sulla "classe testante" → Run As → CodeCover Measurement For JUnit.
   Per vedere le percentuali di copertura di ogni test selezionare dal menù in alto Window → Show View → Other... e scegliere "Coverage".



   Mentre se si vuole vedere la copertura dei termini (utile per la copertura delle decisioni e delle condizioni) andare su Window → Show View → Other... e scegliere "Boolean Analyzer".



3 Verifica del codice

3.1 Concetti generali sulla logica

Lo scopo della logica nell'informatica è quello di creare un linguaggio per modellare situazioni con un linguaggio formale. *Argomentare delle situazioni* significa costruire argomenti su di loro: questi argomenti devono essere validi e rigorosi, così che una macchina possa eseguirli.

Esempio: *se il treno è in ritardo e non ci sono taxi alla stazione, allora John arriverà in ritardo al suo incontro.*

Come esseri umani sappiamo già come "analizzare" l'esempio riportato, quindi se troviamo l'affermazione *"John non è arrivato in ritardo all'incontro. I treni erano in ritardo"* possiamo fare la deduzione *"Quindi c'erano dei taxi alla stazione".*

Argomentando questa situazione si possono trovare argomenti che dati in input ad una macchina la farà arrivare a questa stessa conclusione.

Da notare che alla macchina non importa del significato delle frasi ma la loro *struttura logica*, quindi gli argomenti possono essere riassunti in lettere.
Es: `If p and not q, then r. Not r. p. Therefore, q`.
Dove p = treni in ritardo ; q = c'erano taxi ; r = John è in ritardo.

3.1.1 Linguaggio proposizionale

Per rendere gli argomenti rigorosi serve sviluppare un linguaggio che estrapoli la struttura logica dalle frasi: questo linguaggio prende il nome di ***linguaggio proposizionale logico***.

Proposizione frase dichiarativa che si può argomentare oggettivamente come vera o falsa.

Esempio: *proposizioni.*

"La somma dei numeri 3 e 5 è 8" è oggettivamente vera.
"*Mi piace la pizza*" è una frase dichiarativa ma non è oggettiva, quindi non è un proposizione.

Le proposizioni possono essere espresse in qualsiasi linguaggio naturale.

Siamo interessati a frasi dichiarative precise: dobbiamo controllare se un certo programma (o sistema) riesce a soddisfare le specifiche date, partendo dal presupposto che se tutte le assunzioni sono vere, allora la nostra conclusione è altrettante vero.

Frasi atomiche Sono proposizioni non scomponibili ulteriormente. Es: il numero cinque è dispari. Possiamo combinare le frasi atomiche più complesse attraverso queste regole:

- **Negazione:** si indica con $\neg$ e serve a negare una certa espressione, cioè ad invertirne il risultato. Es: dato $\neg p$ se p è $TRUE$ allora $\neg p = FALSE$
- **Unione:** date due proposizioni indica che almeno una è vera. Si indica con $p \vee r$.
- **Congiunzione:** date due proposioni indica che entrambe sono vere. Si indica con $p \wedge r$.
- **Implicazione:** data l'espressione $p \rightarrow q$ si dice che q è una conseguenza logica di p.
  La parte a sinistra della freccia di implicazione si chiama "ipotesi", mentre la parte a destra si chiama "tesi":

  $$\underbrace{p}_{ipotesi} \rightarrow \underbrace{q}_{tesi}$$

  L'implicazione può essere espressa come un'unione dell'ipotesi negata e della tesi: $p \rightarrow q \equiv \neg p \vee q$ (dove $\equiv$ è il simbolo di equivalenza semantica, cioè indica che le due espressioni producono due tavole di verità identiche). Allora:
  - Se l'ipotesi è falsa, l'implicazione è vera indipendente dalla tesi;
  - Se l'ipotesi è vera, l'implicazione è vera solo se la tesi è vera.

  Es: $p \wedge q \rightarrow \neg r \vee q$ indica che "se p e q sono vere, allora not(r) o q sono vere".

Deduzione naturale Abbiamo parlato in precedenza di come date delle assunzioni vere si può concludere che la conclusione sia vera, ma come possiamo dimostrarlo formalmente in modo che anche un programma lo capisca? Serve un insieme di regole che date le premesse ci permetteranno di arrivare ad una certa conclusione.

Supponiamo di avere una lista di formule $\varphi_1, \varphi_2, \ldots, \varphi_n$ che chiameremo ***premesse*** e una formula ψ che chiameremo ***conclusione***, applicando delle ***regole di prova*** alle premesse si otteranno delle "formule intermedie" che alla fine ci porteranno alla conclusione (e quindi dimostrare che date le premesse si può arrivare alla conclusione). Questo processo viene indicato dall'espressione $\varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi$, che prende il nome di ***sequente***.

Conseguenza semantica È un altro modo di vedere la definizione di sequente: una formula ψ è conseguenza semantica di un set di formule $\Gamma = \varphi_1, \varphi_2, \ldots, \varphi_n$ se e solo se quando Γ è vera (cioè tutte le sue formule sono vere) allora anche ψ è vera. Si indica con $\Gamma \models \psi$ oppure $\varphi_1, \varphi_2, \ldots, \varphi_n \models \psi$.

Esempio: *se il treno è in ritardo e non ci sono taxi alla stazione, allora John arriverà in ritardo al suo incontro.*

Poniamo p = treni in ritardo ; q = c'erano taxi ; r = John è in ritardo. Il sequente è

$$p \wedge \neg q \rightarrow r,\ \neg r,\ p \vdash q$$

che si traduce come "se è vero che i treni sono in ritardo (p) e ($\wedge$) non c'erano taxi ($\neg q$) allora ($\rightarrow$) John era in ritardo (r), se John non è in ritardo ($\neg r$) e i treni erano in ritardo (p) allora ($\vdash$) c'erano taxi (q)".

3.1.2 Regole di deduzione naturale

Le regole hanno il formato:

$$\frac{premesse\ della\ regola}{conclusione}nome\ della\ regola_{tipologia}$$

Il pedice "tipologia" indica se la regola introduce (pedice i) o elimina (pedice e) delle formule.

and introduction se abbiamo φ e ψ provate come vere separatamente, possiamo aggiungere la regola $\varphi \wedge \psi$. Si indica con

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi}\wedge_i$$

and elimination è l'operazione duale all'and introduction: se abbiamo la prova di $\varphi \wedge \psi$, allora possiamo ricavare ψ oppure φ. Si indicano con

$$\frac{\varphi \wedge \psi}{\varphi}\wedge_e \ ; \ \frac{\varphi \wedge \psi}{\psi}\wedge_e$$

Esempio: *data $p \wedge q$, $r \vdash q \wedge r$ prova che è valida.*

Come primo passo si elencano le premesse, si lascia una riga vuota e si scrive la conclusione:

$p \wedge q$
r
$\dots$
$q \wedge r$

Si cercherà di arrivare alla conclusione utilizzando le regole.

- si può applicare la and elimination per ottenere q da $p \wedge q$.
- ottenuta q, abbiamo r dalla premessa, quindi si può applicare la and-introduction per ottenere $q \wedge r$, cioè la nostra conclusione.

doppia negazione data una premessa possiamo ricavarne la sua doppia negazione e viceversa. Si indica con

$$\frac{\neg\neg\varphi}{\varphi}\neg\neg_i \ ; \ \frac{\varphi}{\neg\neg\varphi}\neg\neg_e$$

Esempio: Non é vero che non piove ($\neg\neg\varphi$) equivale a dire che piove (φ).

modus ponens consiste nell'eliminare l'implicazione, chiamata anche implies elimination. Si indica con:

$$\frac{\varphi \quad \varphi \to \psi}{\psi}\to_e$$

Esempio: Se φ =piove e $\varphi \to \psi$ indica che "se piove, le strande sono bagante" allora ψ è "le strade sono bagnate".
Se sappiamo che ha piovuto (φ) e sappiamo che le strade si bagnano quando piove ($\varphi \to \psi$) allora possiamo dire che le strade sono bagnate (ψ).

modus tollens se abbiamo un'implicazione $\varphi \to \psi$ e sappiamo che ψ è falsa, allora possiamo dire che anche φ è falsa. Si indica con:

$$\frac{\varphi \to \psi \quad \neg\psi}{\neg\varphi} MT$$

Esempio: Se Abramo Lincoln era etiope (φ), allora lui era africano ($\varphi \to \psi$).
Abramo Lincoln non era africano ($\neg\psi$), quindi non era etiope ($\neg\varphi$).

Dimostrare validità di una proposizione I passi per dimostrare che una proposizione è valida attraverso le regole sono:

- Individuare le proposizioni atomiche e rappresentarle attraverso lettere.
  Es: Data la proposizione *Se piove allora non esco. Piove.* poniamo p = "piove" e u = "sono uscita".
- Scomporre la frase principale in frasi più semplici e ricavare la loro struttura logica.
  Es: *Se piove allora non esco. Piove.* si scrivono $p \to \neg u$ e p.
- Ricavare il sequente.
  Es: *Se piove allora non esco. Piove. Quindi non sono uscita.*
  Il sequente è $p \to \neg u,\ p \vdash \neg u$
- Elencare i componenti (premesse e conclusione).
- Applicare le regole viste per arrivare alla conclusione.
  Es: dal sequente $p \to \neg u,\ p \vdash \neg u$ si applica modus tollens a $p \to \neg u$ e si ricava $\neg u$.

Esempio: *Individua le preposizioni, scrivi e dimostra i sequenti di "Se piove, allora se ho l'ombrello sto asciutto. Piove. Non sono asciutto. Non avevo l'ombrello."*

Poniamo p = "piove" ; o = "ho l'ombrello" ; a = "sto asciutto".
Scomponiamo il testo in frasi più semplice, evidenziando le parole chiave:

- "*Se piove, allora se ho l'ombrello sto asciutto*" diventa $p \to (o \to a)$
- "*Piove*" diventa p

- "***Non*** *sono asciutto*" diventa $\neg a$
- "***Non*** *avevo l'ombrello*" diventa $\neg o$

Il sequente è quindi $p \to (o \to a)$, p, $\neg a \vdash \neg o$, elenchiamo i vari componenti:

$p \to (o \to a)$
p
$\neg a$
$\dots$
$\neg o$

Applichiamo le seguenti regole:

- applichiamo modus ponens, data p vera e $p \to (o \to a)$ si ricava $o \to a$
- applichiamo modus tollens, data $o \to a$ e sapendo che a è falsa ($\neg a$), allora si può ottenere $\neg o$, che è la conclusione.

3.2 Concetti generali sulla verifica

Con il testing si può solo provare la presenza di difetti, mentre con la verifica si può dimostrare l'assenza di difetti.

Le proprietà che proveremo vere sono garantite per ogni esecuzione assumendo ad esempio che il computer non si rompa, che la memoria non finisca oppure che non ci sia overflow, ecc.

Criteri per la classificazione degli approcci alla verifica Per provare le proprietà esistono diversi metodi, divisi per:

classificazione dell'insieme 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 $\Gamma \vdash \phi$ (sequente)

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 automatico: il processo di verifica è completamente svolto da un programma (caso ottimale ma con complessità esponenziale).

semi automatico: la maggior parte delle azioni da eseguire sono automatiche, ma alcune sono particolarmente complesse dal punto di vista algoritmico e devono essere eseguite manualmente: esistono delle buone euristiche che aiutano il programmatore a completare questi step.

estensione della verifica 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", ...

sequentiale si assume che il programma sia eseguito su singolo processore e che non ci sia concorrenza

transformational il programma riceve un input e dopo un tempo di elaborazione finito termina con un output

fase di sviluppo 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.

3.2.1 Tripla di Hoare

Più formalmente, 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.

$\wedge$ per la AND Es: $\{x > 0 \wedge y = 3\}$

$\vee$ per OR

$\neg$ per NOT

$\rightarrow$ per implica, cioè se ho $A \rightarrow B$ signigica A implica B, cioè A è più forte di B
Es: se $A = \{x = 2\}$, $B = \{x > 0\}$ avere $x = 2 \rightarrow x > 0$ vuol dire "$x = 2$ implica che x è positivo"

Esempio *tripla di Hoare*

Sia SQRT una routine per il calcolo della radice quadrata, la correttezza di questa routine richiede che i valori inseriti siano positivi e che il quadrato di x sia approssimato per un errore pari a ε. La tripla di Hoare è

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

S : algoritmo per calcolare y come radice quadrata di x (con precisione ε) :

$$abs(y^2 - x) \leq 2 * \varepsilon * y$$

P : " l'input è positivo $x \geq 0$" deve essere vera prima di applicare l'algoritmo
Q : "y approssima esattamente il quadrato di x con un errore pari a ε" deve essere vera dopo l'applicazione dell'algoritmo.

Si può scrivere la tripla come

$$\{x \geq 0\}\, abs(y^2 - x) \leq 2 * \varepsilon * y\, \{y = \sqrt{x} \pm \varepsilon\}$$

Tripla di Hoare con aggiornamenti è una variante che consiste nell'integrare nella tripla standard (P; S; Q) degli updates (U) che rappresentano i cambiamenti del programma fino a quel momento.

Gli update sono contenuti tra parentesi quadre che restano vuote nel caso non ci siano stati cambi di stato. Gli operatori utilizzati nell'update sono:

:=	update atomico, cioè ad una variabile è stato cambiato il suo valore. Es. [x:= 5].
,	update in sequenza. Es. [x:= 5, y:= x+1]
|	update in parallelo, si assume che siano sempre consistenti. Es. [x:= 5 | y:= x+1]

Davanti ad un programma mettiamo gli update che aggiornano il suo stato a prima dell'esecuzione ($[U]S$), quindi la tripla di Hoare diventa:

$$\{P\}[U]S\{Q\}$$

Si ridefinirà la ***correttezza di S*** come: a partire dalle precondizioni (P), applica gli updates (U), esegui il programma S, e alla fine dell'esecuzione le postcondizioni (Q) devono essere valide.

Si fa notare che la tripla di Hoare strandard $\{P\}S\{Q\}$è un caso particolare della tripla con update (cioè quando non ci sono update per S) e si indicherà come$\{P\}[]S\{Q\}$.

Dynamic logic Una tripla di Hoare può essere espressa come una formula di logica dinamica:

- Correttezza parziale $P \Rightarrow [C]Q$
  - Se la precondizione P mantiene il corrente stato e l'esecuzione C raggiunge un altro stato, allora Q rimarrà in quello stato, ed è equivalente a $\{P\}C\{Q\}$
- Totale correttezza
  - Se la precondizione P mantiene il suo stato corrente allora esiste un altro stato che l'esecuzione C può raggungere in cui Q rimarrà in quello stato.
    Se C è deterministico allora esiste questo stato, quindi si ha l'equivalenza con $\{P\}C\{Q\}$

Anche qui esistono delle regole di deduzione naturale, per esempio la regola dell'assegnamento:

$$\frac{\{P\}[U, x := e]s\{Q\}}{\{P\}[U]x := e;\ s\{Q\}}$$

Indica che se il programma S è corretto rispetto a P e Q dati degli updates tra cui l'assegnamento $x := e$, allora si ha correttezza anche eseguendo in modo seriale l'aggiornamento e poi S.

3.3 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 che lega cliente e fornitore specificando obblighi e benefici delle parti (normalmente mappati in modo "duale"), deve essere esplicito (scritto) e non contenere clausole nascoste.

Esempio: *obblighi e vantaggi di cliente e fornitore*

	Obblighi	Vantaggi
Cliente	Mettere a disposizione un ettero di terreno edificabile e una certa quantità di soldi	Ricevere un edificio di tre piani entro un certo numero di mesi
Fornitore	Costruire un palazzo di 3 piani entro un certo numero di mesi	Non deve fare nulla se non riceve i soldi o se il terreno non è adatto

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.

- Precondizione forte: il metodo non può essere chiamato se non rispetta le precondizioni;

- Precondizione debole: la routine a gestire eventuali complicazioni.
  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.

Esempio: *precondizioni*

Ho un array di dieci elementi e ho il seguente metodo:

```
int incr(int a) { return a[x]+1; }
```

Le possibili precondizioni possono essere:

- precondizione debole: posso passare qualsiasi valore, dentro la routine farò in modo di non andare fuori dai limiti

  ```
  int incr(int a) if (x > 0 && x <= 9) { return a[x] + 1; }
  ```

- precondizione forte: impongo come precondizione

  ```
  // precondizione per x > 0 && x <= 9
  int incr(int a) { return a[x]+1; }
  ```

  La sintassi per scrivere le precondizioni verrà vista in seguito.

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. In questo 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).
Es: l'indice di un array deve essere sempre compreso tra 0 e la sua lunghezza - 1.
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.

3.4 JML

JML è un insieme di tool e di tecniche per il Design by Contract in Java e supporta invarianti di classe, precondizioni, postcondizioni, asserzioni locali e invarianti sui cicli.

3.4.1 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> @*/` oppure `//@ <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");

Esempio: *precondizioni e postcondizioni con JML*

Sia la seguente classe

```
public class Account{
  int balance;
  static final int minBalance = 1000;
```

```
  public Account (int initialAmount) { balance = initialAmount; }
}
```

Una precondizione forte può essere l'imporre che si crei un account con almeno l'importo minimo richiesto:

```
//@ requires initialAmount >= minBalance;
```

Mentre una postcondizione è imporre che sia stato aggiunto il giusto importo al bilancio:

```
//@ ensures balance == initialAmount;
```

Il codice JML è messo prima del metodo a cui fa rifermento, quindi si avrà:

```
//@ requires initialAmount >= minBalance;
//@ ensures balance == initialAmount;
public Account (int initialAmount) { ... }
```

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

\result per riferirsi al valore restituito;

Esempio: *\old e \result in JML*

Consideriamo l'esempio del conto in banca precedente aggiungendo due metodi:

```
public void deposit(int amount){ balance = balance + amount; }
public int withdraw(int amount) { balance -= amount; return balance; }
```

Per il metodo `deposit` si dovrà controllare che l'importo depositato sia positivo (precondizione) e che la somma finale sia effettivamente (vecchio)balance + amount (postcondizione).

```
//@ requires amount >= 0;
//@ ensures balance == \old(balance)+ amount;
```

Per il metodo `withdraw` le precondizioni saranno avere l'importo da ritirare maggiore di zero e che tale quantità non faccia scendere il bilancio sotto la soglia minima.
Le postcondizioni sono controllare che la sottrazione sia giusta e che il risultato sia uguale al nuovo bilancio.

```
/*@ requires amount >= 0 && amount <= balance - minBalance;
ensures balance == \old(balance) - amount && \result == balance; @*/
```

invariant per specificare invarianti, fanno il controllo all'inizio e alla fine del metodo;

Esempio: Per il conto in banca si può mettere `/*@ invariant balance >= minBalance; @*/`

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

Esempio: *assert in JML*

Si controlla che i valori delle variabili dentro l'`else` siano corrette:

```
if (i <= 0 || j < 0) {
  ...
} else if (j < 5 ) {
  //@ assert i > 0 && 0 <= j && j < 5;
  ...
}
```

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;

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

array

- 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;

Implicazioni

- 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).

Esempio *implicazioni*

Ho il metodo `minore(j, n)` e posso ritornare vero se e solo se `j < n`, quindi impongo nella postcondizione che se il risultato è vero allora `j < n` e viceversa:

```
//@ ensures \result <==> j < n;
boolean minore(int j, int n) {
  return j < n;
}
```

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: **<expression>** dev'essere booleana, e la valutazione del quantificatore restituisce un booleano;

\exists quantificatore esistenziale: **<expression>** dev'essere booleana, e la valutazione del quantificatore restituisce un booleano;

\sum, \product, \min, \max quantificatori generali: **<expression>** dev'essere un long (o un intero), e la valutazione del quantificatore restituisce un long;

\num_of quantificatore numerico: **<expression>** dev'essere booleana, e la valutazione del quantificatore restituisce un long.

Mentre `<type>` indica il tipo della variabile `<var>` i cui valori variano in `<range predicate>`.

Esempio *quantificatori*

Ho il metodo

```
public void resetArray(int[] a) {
  for (int i = 0 ; i < a.length ; i++)
    a[i] = 0;
}
```

Il seguente quantificatore significa: per ogni intero x compreso tra 0 e la lunghezza di a - 1, a[x] deve essere uguale zero.

```
//@ ensures ( \forall int x; 0 <= x && x < a.length - 1 ; a[x] == 0 );
```

Un'alternativa può essere:

```
//@ ensures ( \num_of int x; 0 <= x && x < a.length - 1 ; a[x] == 0 ) ==
a.length;
```

Che significa: il numero degli elementi di `a[x]` (dove `x` è un intero che va da 0 a `length` - 1) deve essere uguale alla lunghezza dell'array.

Un'altra alternativa è controllare che la somma sia uguale a zero:

```
//@ ensures (\sum int x; 0 <= x && x < 5; a[x]) == 0
```

In questo caso nel campo `expression` non si mette un booleano (Es: `a[x] == 0`) ma ciò che si vuole sommare.

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.

Esempio `private /*@ spec_public @*/ int weight;`

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`:

```
/*@ non_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.

Esempio `private /*@ non_null @*/ File[] files;`

Mentre per le variabili che devono essere nulle si ha:

```
/*@ nullable @*/ <type> <var_name>;
```

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>;
```

Esempio: *comportamento normale ed eccezionale*

Il metodo principale è l'algoritmo `peek`,che estrae un oggetto da una coda.

```
*@ pure @*/ Object peek() throws NoSuchElementException;
```

Il "comportamento normale" (`normal_behavior`) consiste nell'avere la coda non vuota (`isEmpty()` falso) e che l'elemento estratto appartenga alla lista `elementsInQueue`.

Vengono anche gestiti i comportamenti "anomali" (`exceptional_behavior`), cioè nel caso la coda sia vuota é lanciata l'eccezione `NoSuchElementException`, che indica cha l'elemento non esiste.

```
/*@ public normal_behavior
@ requires !  isEmpty();
@ ensures elementsInQueue.has(\result);
@ also
@ public exceptional_behavior
@ requires isEmpty();
@ signals (Exception e) e instanceof NoSuchElementException;
@*/
/*@ pure @*/ Object peek() throws NoSuchElementException;
```

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`.

Esempio: *sottotipazione*

Estendo `Account` con `ContoDeposito`

```
class ContoDeposito extends Account { ... }
```

Tutti gli invarianti di `Account` si applicano anche al conto deposito e ogni (overriding) metodo di `ContoDeposito` deve soddisfare le specifiche della superclasse.
Quindi invece che richiedere un nuovo contratto, JML usa il principio di specification inheritance in cui i contratti del metodo della super classe sono uniti con la parola chiave `also` ai contratti per la sottoclasse.

3.4.2 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`).

Quando risulta utile usare una dichiarazione assert:

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

3.4.3 Correttezza di una classe

Nel DbC una classe è corretta se sono corretti i costruttori e gli altri suoi metodi, quindi se sono rispettate precondizioni, varianti e postcondizioni:

- Per il costruttore deve valere:

  $$\{Precondizione_{costruttore}\}\ costruttore\ \{Invariane\ AND\ Postcondizione_{costruttore}\}$$

  Cioè si impone di avere la precondizione vera e dopo l'esecuzione del costruttore (cioè dopo che si è fatto `Nome_classe tempo = new Nome_classe( ··· );`) si vogliono avere invarianti e postcondizionli vere.

- Per ogni altro metodo se lo chiamo con precondizione vera e con un certo invariante, allora dopo l'esecuzione del metodo valgono le postcondizioni e continua a valere l'invariante,cioè:

$$\{Precondizione_{metodo}\ AND\ invariante\}\ metodo\ \{Invariane\ AND\ Postcondizione_{metodo}\}$$

Esempio: *correttazza di una classe*

Si crea la classe `counter`, imponendo come variante che $count \geq 0$

```
public class counter {
  int count;
  //@ invariant count =>0
  public counter() { count = -1; }
}
```

Questa classe non è giusta, infatti viola l'invariante!

3.4.4 Correttezza dei cicli

Introducendo i cicli può nascere il problema della **terminazione**, cioè il rischio che il programma vada in loop infinito (cioè non si sa se termina oppure no).

Molti programmi di controllo (compreso KeY) fanno fatica a provare che il ciclo termina (anche per istruzioni molto semplici), quindi per non costringere KeY a provare che un cilco termina si aggiunge il comando:

```
//@ diverges true;
```

Esempio: *terminazione di un ciclo*

```
//@ public normal_behavior
//@ requires timer >= 0;
//@ ensures \result==0;
//@ diverges true;
public static int countdown(int timer) {
while(timer>0){ timer --; }
return timer;
```

Mentre se si volesse fare un'analisi più accurata (cioè dimostrare che effettivamente il programma termina), ci sono diverse alternative che verranno mostrate in seguito.

3.5 Design by contract in Eclipse

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

OpenJML utilizzato per l'analisi dinamica: dopo aver scritto la classe java con le relative pre/post condizioni verranno testati gli input in un main.

Key utilizzato per l'analisi statica: attraverso regole della logica si cerca di provare che la classe ha un comportamento corretto se si rispettano le pre/post condizioni.

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/`);

3.5.1 Introduzione a OpenJML

OpenJML si occupa di fare controlli sintattici sul codice JML e di segnalare quando in un'esecuzione del programma qualche pre/post condizione non è stata rispettata.

Per controllare che il codice è stato scritto correttamente si preme il pulsante RAC, e nel caso ci siano errori comparirà la spunta vicino alla riga errata.

Per controllare se le pre/post condizioni sono rispettate, basterà creare una classe main dove si testeranno le istanze della classe e la loro risposta agli input: il ragionamento è identico rispetto a quello visto con CodeCover, con la differenza che non serve richiamare metodi assert, infatti una volta avviato il main ci penserà OpenJML a confrontare i valori con quelli richiesti dalle condizioni e a segnalare eventuali scorrettezze.

Esempio: *test di una classe con OpenJML*

Consideriamo la classe `counterFrom`

```
public class counterFrom {
  int count;
  //@ invariant count =>0
  //@ endures count == from;
  public counter(int from) { count = from; }
}
```

Si crea la classe Main e si prova ad inserire qualche input

```
public class Main {
  public static void main(String[] args) {
    counterFrom prova1 = new counterFrom(5);
    counterFrom prova2 = new counterFrom(-10);
}
```

Notare che l'ultimo input genererà errore, infatti anche se rispetta la postcondizone (`count == from`) non rispetta l'invariante (si è inserito un valore di `count` negativo).

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;

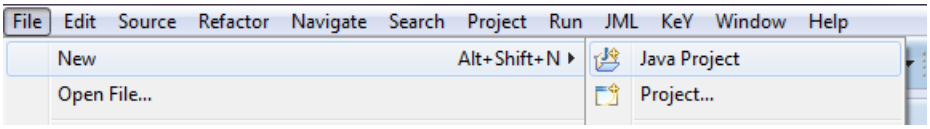
escjava2 controllo statico del codice per la prova di correttezza;

jmlunit tool per test d'unità JML/JUnit;

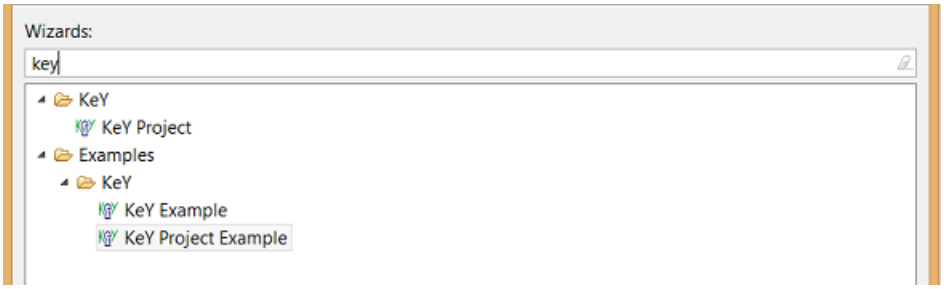
jmldoc per la generazione automatica di pagine html contenenti la documentazione del codice;

3.5.2 Introduzione a KeY

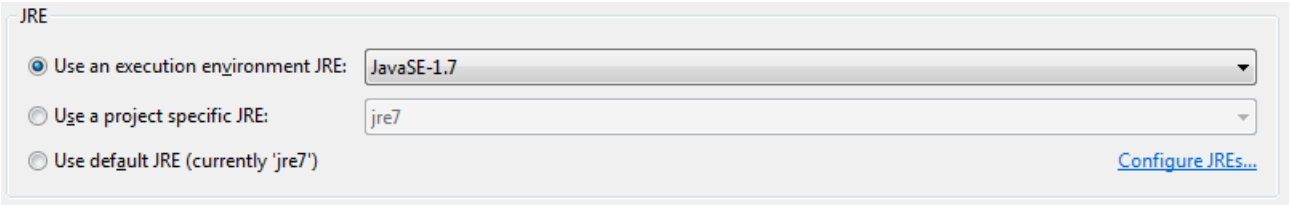
Dato un programma scritto in Java con le sue relative pre/post condizioni, per poterlo analizzare staticamente serve come prima cosa creare un progetto Key, andando su File → New → Project...



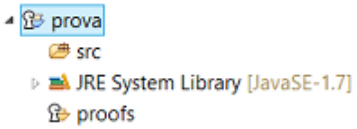
Aperta la schermata di creazione del progetto, cercare "key" e scegliere "KeY Project".



Nota: KeY funziona solo con Java 7, controllare quindi che nella sezione JRE di crea progetto sia specificato JavaSE-1.7 oppure JRE7.

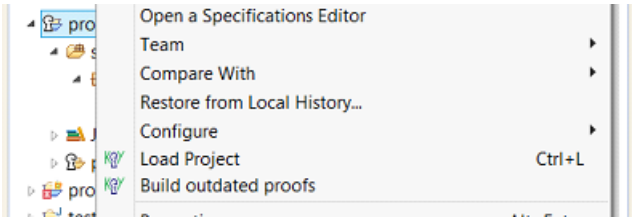


Se la creazione è andata a buon fine, verrà creato un progetto con una chiave come icona.

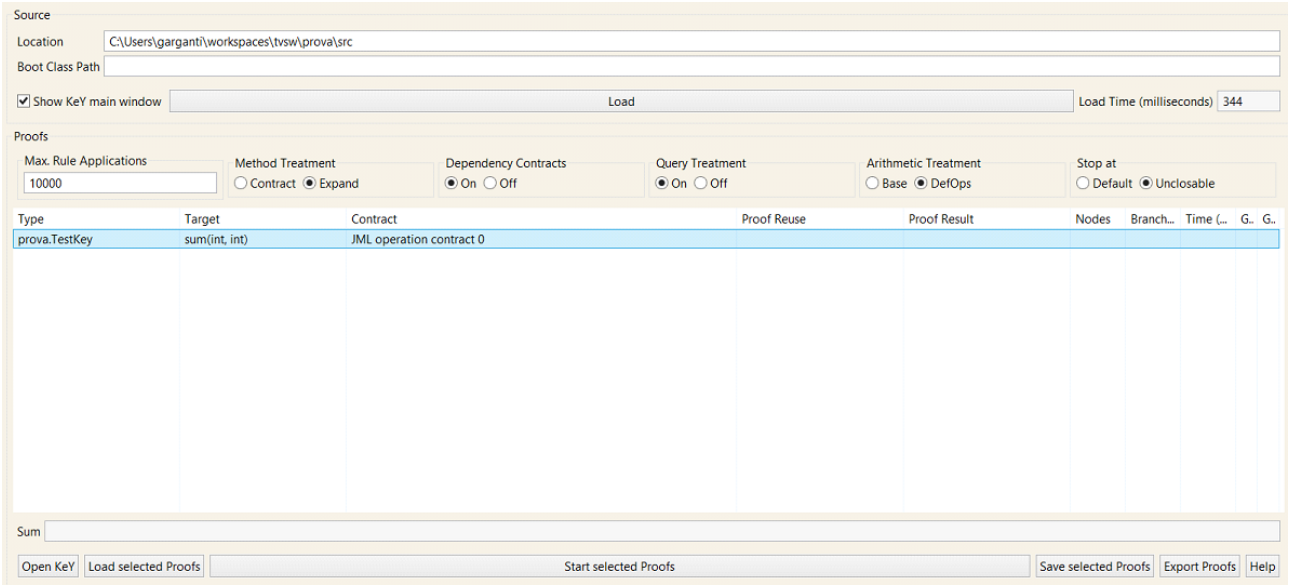


Si ricorda che KeY non funziona molto bene con System.Out, quindi prima di fare l'analisi è meglio commentare ogni tipo di output a schermo.

Per testare i progetti si caricano in KeY clicclando con il tasto destro sull'icona del progetto e scegliendo "Load Project" e nel caso venga chiesto con quale applicazione caricarlo scegliere "MonKeY".



Verrà aperta questa schermata, premere su “Load” per caricare le classi.



Premendo il bottone “Start selected proofs” il programma inizierà ad analizzare i progetti presenti. Nella colonna “Proof Result” verrà indicato il risultato dell’analisi:

Open l’analisi non è andata a buon fine (non si sono provate alcune regole), per trovare l’errore basterà aprire KeY (se non già aperto, premere Open KeY in basso a sinistra) e cercare la dicitura “OPEN GOAL” nella sezione “Proof”.

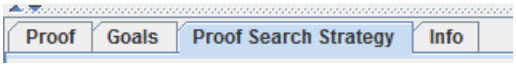
Closed l’analisi è andata a buon fine (tutte le regole sono state provate).

3.5.2.1 Correttezza dei cicli con KeY

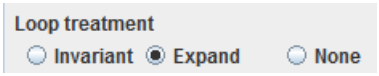
Si è parlato precedentemente di come KeY ha difficoltà a provare la correttezza dei cicli, analizziamo le tecniche per aggirare questo problema.

Unrolling Utilizzato quando si conosce il numero di iterazione ed esse sono piccole (Es: ciclo `for`), questa tecnica consiste nel trasformare il ciclo in una sequenza, così che l’analizzatore la tratti come una semplice lista di istruzioni.

- Caricare la classe come visto in precedenza (apertura di MonKeY)
- Aprire KeY premendo il bottone Open KeY in basso a sinistra
- In Key selezionare il menù “Proof Search Strategy”



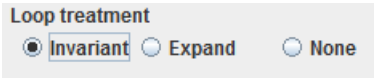
- Cercare nella lista “Loop treatement” e selezionare “Expand”



Invariante di ciclo Se non si conoscono il numero di iterazioni questo può essere particolarmente grande, quindi expand non funziona bene.

Si utilizza un invariante di ciclo, già visto in precedenza (vedi pagina 50).

Per abilitarlo in KeY si segue la stessa procedura vista per l'unrolling, con l'unica differenza che si deve selezionare “Invariant” al posto di “Expand”.



Un invariante di ciclo si trova attraverso le seguenti operazioni:

- Osservare la postcondizione e tentare di generalizzarla
- Ciclare il loop qualche volta per vedere se è presente un pattern
- Trovare un primo loop invariant e testarlo, cioè provare che sia inizialmente valido, vale per tutto il ciclo e permette di confermare la postcondizione
- Si possono incontrare ulteriori condizioni da aggiungere all'invariante, che a loro volta devono essere inizialmente valide e conservarsi nel ciclo
- Ripetere finché l'invariante riesce a dimostrare la postcondizione

Esempio *invariante di un ciclo*

Consideriamo il ciclo while dell'esempio precedente, si deve provare che la postcondizione `\result==0` (`timer` $= 0$) sia vera.
Consideriamo i valori di `timer` prima e dopo l'entrata nel ciclo:

- Prima del ciclo: la precondizione impone `timer >= 0`
- Dopo il ciclo: finché si rimane nel ciclo `timer` è positivo, quindi all'uscita varrà `timer <= 0`

Se la precondizione valesse sempre alla fine del ciclo si avrebbe sia `timer >= 0` che `timer <= 0` e quindi `timer = 0` (la postcondizione sarebbe vera).
Per imporre che la condizione `timer >= 0` valga sempre nel ciclo basta definirla in un loop invariant, quindi si aggiungerà la seguente porzione di codice:

```
//@ loop_invariant
//@ timer >= 0;
```

Il procedimento appena visto può essere riassunto nella seguente regola, chiamata ***loop rule***:

$$\frac{P \rightarrow U(I), \{I\ AND\ b\}S\{I\}, \{I\ AND\ \neg b\}[]s2\{Q\}}{\{P\}[U]while\ b\ do\ S\ od\ s_2\{Q\}}$$

Per spiegarla la scomponiamo in componenti:

- Al numeratore abbiamo:

  $P \rightarrow U(I)$ — Prima di entrare nel ciclo: l'invariante I deve valere all'inizio del ciclo, cioè I è una conseguenza della precondizione P.

  $\{I\ AND\ b\}S\{I\}$ — Esecuzione del ciclo:
  Precondizione: sia l'invariante che la condizione del ciclo (b) devono essere vere ($\{I\ AND\ b\}$).
  Programma: le istruzioni di un iterazione del ciclo (S).
  Postcondizione: l'invariante deve valore dopo un iterazione ($\{I\}$).

  $\{I\ AND\ \neg b\}[]s_2\{Q\}$ — All'uscita dal ciclo:
  Precondizione: si è usciti dal ciclo quindi la sua condizione è falsa ($\neg b$) e l'invariante vale ancora (ricavato per induzione: se vale per ogni iterazione, allora all'uscita del ciclo vale ancora).
  Programma: parte restante del programma dopo il ciclo (s_2).
  Postcondizione: Q.

- Al denominatore abbiamo la tripla risultatante dai ragionamenti appena descritti, composta da:
  - Precondizione: P
  - Programma: ciclo while per una certa condizione b e delle istruzioni S, e usciti dal ciclo ci saranno delle istruzioni s_2
  - Postcondizione: Q

Di conseguenza, date certe precondizioni, certi update, il programma rispetterà le postcondizioni Q anche con un ciclo while all'interno.

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, interattivi, e all'origine era usato post-sviluppo.

Da dei requisiti formali si ricava un modello su cui si verificano proprietà e si eseguono confronti in modo che il modello sia conforme con il sistema, quindi si può riassumere questo processo in 3 fasi:

1. scrittura del modello $\mathcal{M}$ da parte dell'utente (programmatore o simili);
2. asserzione di proprietà relative al modello da parte dell'utente (cioè si codificano le proprietà da verificare in una formula temporale ϕ);
3. verifica della validità delle proprietà, cioè verificare $\mathcal{M} \models \phi$ e quindi dimostrare che $\mathcal{M}$ soddisfa le proprietà ϕ; in caso le proprietà non siano valide, si possono 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.

I principali vantaggi nell'utilizzare un modello sono:

compatezza il modello è più semplice rispetto al sistema completo;

predittività il modello simula il comportamento tipico del sistema;

velocità il modello è pronto prima del sistema;

automatizzazione il modello è più facile da automatizzare;

viste possono esserci più modelli per diversi aspetti del sistema.
Es: un modello per l'analisi de rischi, un modello per le interazioni con l'utente, ecc.

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)

La logica è costruita su di un insieme di formule atomiche AP $\{p, q, r, \dots\}$ che rappresentano descrizioni atomiche del sistema.

Operatori

- Connettore logici classici:

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

- Connettori temporali

X	*neXt state*. Es: Xa significa che a sarà vera nel prossimo stato
F	*some Future state*. Es: Fa significa che a sarà vera in almeno uno degli stati futuri
G	*all future state*, anche detto Globally. Es: Ga significa che a sarà vera in tutti gli stati futuri
U	*Until* indica che una condizione a è vera fino a quando la condizione b diventa vera
W	*Weak-until* come until ma non obbliga b ad essere vera per forza
R	*Release* è il duale di until, cioè una condizione b è vera fino a quanto la condizione a diventa vera

Priorità degli operatori Dalla più alta alla più bassa:

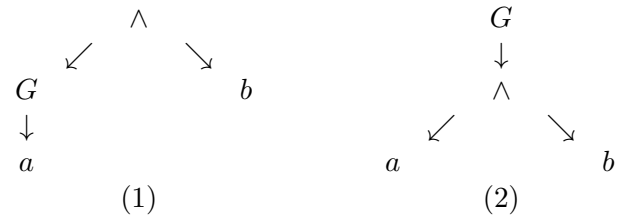
- $\neg$, X, F, G
- U, R, W
- $\wedge$, $\vee$
- $\rightarrow$

Esempio: *priorità degli operatori*

Siano le istruzioni $Ga \wedge b$ e $G(a \wedge b)$.

(1) Per $Ga \wedge b$ l'operatore G ha la priorità maggiore e si applica solo su a, quindi è come se fosse scritta $(Ga) \wedge b$

(2) Per $G(a \wedge b)$ le parentesi hanno priorità maggiore di G, quindi G sarà applicato su $a \wedge b$



Sistema di transizioni Il sistema che vogliamo verificare utilizzando LTL può essere modellato come una *final state machine*: si modella un sistema utilizzando stati (struttura statica) e transizioni (struttura dinamica). Un sistema di transizioni è rappresentato come:

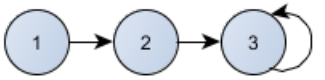
$$\mathcal{M} = (S, \rightarrow, L)$$

dove

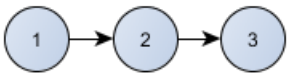
S rappresenta l'insieme degli stati del sistema

$\rightarrow$ è la relazione di transizione, per ogni stato esiste una transizione che i permette di andare nello stato successivo (cioè non ci sono "vicoli ciechi"), cioè per ogni $s \in S$ esiste una $s' \in S$ tale che $s \rightarrow s'$

$$\forall s \in S \; \exists s' \; t.c. \; s \rightarrow s'$$



È un sistema di transizioni



Non è un sistema di transizioni, lo stato (3) non ha transasioni verso altri stati

L rappresenta la funzione label

$$L: \; S \rightarrow \mathcal{P}(Atoms)$$

dove S è il dominio degli stati e $\mathcal{P}(Atoms)$ rappresenta tutti i sottoinsiemi dell'insieme $Atoms$ (insieme delle proposizioni del sistema).
Quindi L è una funzione che indica per ogni stato $s \in S$ quali proposizioni sono vere al suo interno.

Path Un ***path*** (o ***percorso***) in un modello $\mathcal{M} = (S, \rightarrow, L)$ 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

$$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$$

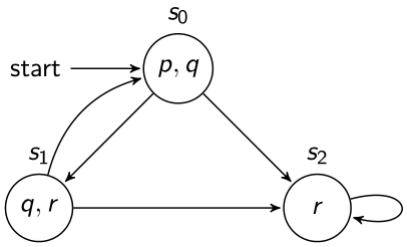
Sia $\pi = s_1 \to s_2 \to s_3 \to \dots$ un path. Allora indichiamo con

$$\pi^i = s_i \to s_{i+1} \to s_{i+2} \to \dots$$

il suo sottopath che inizia dallo stato s_i.

Esempio: *diagramma a stati*

Si analizza il modello $\mathcal{M}$ rappresentato nel seguente diagramma



S	I suoi stati sono $s_1,\ s_2$ e s_3.
$\to$	Le sue transazioni sono $s_0 \to s_1$, $s_0 \to s_2$, $s_1 \to s_0$, $s_1 \to s_2$ e $s_2 \to s_2$. Si fa notare che per ogni s_i c'è una freccia uscente (e quindi esiste lo stato s_j tale che $s_i \to s_j$).
L	Le proposizioni vere negli stati sono: $L(\,s_0\,) = \{p,\ q\}$, $L(\,s_1\,) = \{q,\ r\}$ e $L(\,s_2\,) = \{\,r\,\}$.

Soddisfacimento delle regole: path Sia $\mathcal{M} = (S, \to, L)$ un modello e $\pi = s_1 \to \dots$ un path per $\mathcal{M}$. π soddisfa le seguenti formule LTL:

1. $\pi \models \top$ (true vero per ogni path)
2. $\pi \not\models \bot$ (false falso per ogni path)
3. $\pi \models p$ se e solo se $p \in L\,(s_1)$ (la proposizione p è vera per il path π se e solo se p appartiene alle proposizioni vere dello stato s_1)
4. $\pi \models \neg\phi$ se e solo se $\pi \not\models \phi$ (la negazione della proposizione è vera per il path π se e solo se la proposizione non è vera per quel path)
5. $\pi \models \phi_1 \wedge \phi_2$ se e solo se $\pi \models \phi_1$ e $\pi \models \phi_2$ (l'AND delle proposizioni è vera per il path π se e solo se entrambe appartengono alle proposizioni vere di s_1)
6. $\pi \models \phi_1 \vee \phi_2$ se e solo se $\pi \models \phi_1$ o $\pi \models \phi_2$ (l'OR delle proposizioni è vera per il path π se e solo se almeno una delle due appartiene alle proposizioni vere di s_1)
7. $\pi \models (\phi_1 \to \phi_2)$ se e solo se $\pi \models \phi_2$ ogniqualvolta $\pi \models \phi_1$(l'implicazione è vera per il path π solo se quando si ha ϕ_2 vera allora anche ϕ_1 è vera nello stato s_1)
8. $\pi \models \mathrm{X}\phi$ se e solo se $\pi^2 \models \phi$ (è vera solo se allo stato successivo è vera la proposizione ϕ)
9. $\pi \models \mathrm{G}\phi$ se e solo se $\pi^i \models \phi$ per ogni $i \geq 1$ (notare quindi che il "futuro" comprende il "presente".)(la proposizione vale per ogni π)

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 $\pi^j \models \phi$ (richiede che ψ si verifichi prima o poi).

$$\begin{array}{cccccccc} \phi & \phi & \phi & \phi & \phi & ? & ? & ? \\ & & & & & \psi & & \\ \pi^1 & \pi^2 & \ldots & \pi^{i-2} & \pi^{i-1} & \pi^i & \pi^{i+1} & \ldots \end{array}$$

Il punto di domanda indica che da quando compare ψ non importa se ϕ vale oppure no.

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 $\pi^j \models \phi$ (come in Until)

    b) $\pi^k \models \phi$ per ogni $k \geq 1$

$$\begin{array}{cccccccccccc} \phi & \phi & \phi & \phi & \phi & ? & ? & & & \phi & \phi & \ldots \\ & & & & & \psi & & & & & & \\ \pi^1 & \pi^2 & \ldots & \pi^{i-2} & \pi^{i-1} & \pi^i & \pi^{i+1} & \ldots & & \pi^1 & \pi^2 & \ldots \\ & & & (a) & & & & & & & (b) & \end{array}$$

13. $\pi \models \phi \mathrm{R} \psi$ (è il duale di U; infatti $\phi \mathrm{R} \psi$ equivale a scrivere $\neg(\neg\phi \mathrm{U} \neg\psi)$). se e solo se vale almeno una delle seguenti condizioni:

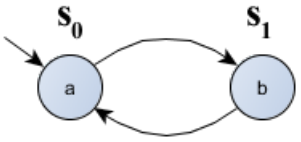
    a) esiste $i \geq 1$ tale che $\pi^i \models \phi$ e per tutti i $j = 1, \ldots, i$ vale $\pi^j \models \psi$

    b) $\pi^k \models \psi$ per ogni $k \geq 1$

$$\begin{array}{cccccccccccc} & & & & & \phi & & & & & & \ldots \\ \psi & \psi & \psi & \psi & \psi & \psi & ? & & & \psi & \psi & \\ \pi^1 & \pi^2 & \ldots & \pi^{i-2} & \pi^{i-1} & \pi^i & \pi^{i+1} & \ldots & & \pi^1 & \pi^2 & \ldots \\ & & & (a) & & & & & & & (b) & \end{array}$$

Esempio: *regole X, G e F.*

Sia il seguente modello



E il path π composto da

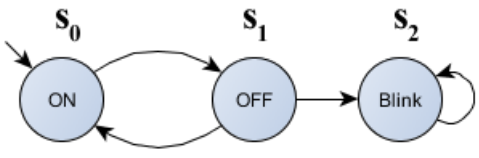
$$a \to b \to a \to \ldots \to b \to a \to \ldots$$

Si applicano le seguenti regole:

$\pi \models a$	È vero, infatti nello stato iniziale vale a.
$\pi \models a \wedge b$	È falso, infatti nello stato iniziale vale solo a e non a **e** b.
$\pi \models Xa$	Si sta chiedendo se nel prossimo stato (neXt state) vale a, ma nel secondo stato vale b, quindi è falso.
$\pi \models XXa$	Ci si chiede se tra due stati vale a quindi si procede di due passi: al passo successivo si ha la formula $\pi^2 \models Xa$ e andando ancora avanti si ottiene $\pi^3 \models a$: al terzo stato c'è a, quindi è vera.
$\pi \models Ga$	a vale sempre? No, quindi è falsa.
$\pi \models G(a \vee b)$	Si sta chiedendo se vale sempre a o b, quindi è vero dato che ogni stato assume come valore a oppure b.
$\pi \models G(a \vee Xb)$	Si sta chiedendo se vale a oppure lo stato dopo b, ma $Xb = a$ quindi si sta chiedendo se vale sempre a cioè $\pi \models Ga$, e si è già visto che è falso.
$\pi \models G(a \vee Xa)$	Si sta chiedendo se vale a oppure lo stato dopo a, ma $Xa = b$ quindi è come se stesse chiedendo se a o b valgono sempre cioè $\pi \models G(a \vee b)$, e si è già visto che è vero.
$\pi \models Fa$	Si sta chiedendo se in futuro la proprietà a è soddisfatta, quindi è vera.
$\pi \models F(Xb \wedge Xa)$	Si sta chiedendo se in futuro sono soddisfatte contemporaneamente le proprietà Xb e Xa, ma $Xb = a$ e $Xa = b$ quindi si sta chiedendo se in futuro esiste un singolo stato in cui a e b valgono contemporaneamente, e questo non è vero.

Esempio: *regole U, W e R.*

Sia la macchina a stati che rappresenta una luce che può essere accesa (`On`), spenta (`Off`) oppure che emana luce ad intermittenza senza fermarsi più (`Blink`).



E sia il suo percorso

$$\pi: \underset{\pi^1}{on} \rightarrow \underset{\pi^2}{off} \rightarrow \underset{\pi^3}{on} \rightarrow \underset{\pi^4}{off} \rightarrow \underset{\pi^5}{blink} \rightarrow \underset{\pi^6}{blink} \rightarrow \underset{\pi^7}{blink} \rightarrow \dots$$

Si mostrano brevemente le regole viste in precedenza:

$\pi \models on$	È vero, perché il primo stato è proprio on.
$\pi \models off$	È falso, si è visto che il primo stato è on.
$\pi \models X\, off$	Ci si sta chiedendo se nello stato successivo (al primo) c'è off, quindi è vero.
$\pi \models F\, blink$	Ci si sta chiedendo se in futuro varrà prima o poi blink, cioè esiste un i tale che $\pi^i \models blink$? Si, con $i = 5$ si ha $\pi^5 \models blink$.

$\pi \models G(on \vee blink)$ — Ci si sta chiedendo se vale sempre ($\forall i \geq 1$) la proprietà, ma non è vero perché non esiste uno stato con sia *on* che *blink*.

$\pi \models G\,(\,blink \rightarrow (\neg on \wedge \neg off)\,)$ — Prima di analizzare la regola completamente, si considera che:

- $\neg on \wedge \neg off$ è (semanticamente) equivalente a $on \vee off$ (legge di De Morgan).
- l'implicazione $blink \rightarrow (\neg on \wedge \neg off)$ è equivalente a $\neg blink \vee (\neg on \wedge \neg off)$ e utilizzando l'equivalenza vista al punto precedente si ottiene

Quindi si sta chiedendo se vale sempre $\neg blink \vee off$, ma si è visto che da $i = 5$ inizia ad essere vera *blink*, quindi è falso.

Ora passiamo a vedere esempi delle nuove regole:

$\pi \models on\,U\,blink$ — Ci si sta chiedendo se è vero *on* finché vale *blink*? No: c'è un alternanza di *on* e off, quindi *on* non vale sempre fino a *blink*, come invece è richiesto da U.
Sarebbe stata vera se il path π avesse avuto una struttura del genere:

$$\pi' : on \rightarrow on \rightarrow on \rightarrow on \rightarrow blink \rightarrow blink \rightarrow \ldots$$

$\pi \models (on \vee off)\,U\,blink$ — In questo caso è vero, infatti *on* oppure off valgono finché non subentra *blink*.

$\pi \models on\,W\,blink$ — È falsa, basta considerare i due casi del nel weak until:

- *on* deve valere sempre

oppure

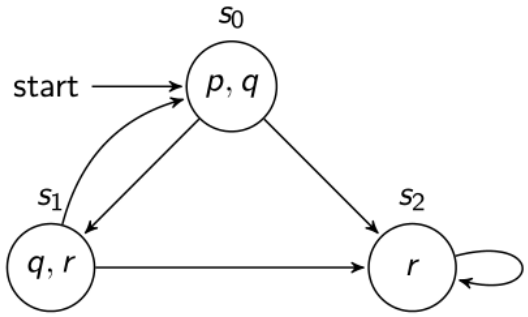
- *on* deve valere fino a quando compare *blink*

E in entrambi i casi non è vero

Soddisfacimento delle regole: stato Sia $\mathcal{M} = (S, \rightarrow, L)$ 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, $\pi \models \phi$.

Esempio: *validità del modello*

Riprendiamo il modello visto in precedenza



$\mathcal{M}, s_0 \models q\,U\,r$ Ci si sta chiedendo se vale q fino a quando vale r, questa volta la proprietà non deve essere vera su un singolo path ma su ogni path che inizia con s_0, si mostrano i due path principali:

- ho il path $s_0 \to s_1 \to s_0 \to s_1 \to s_2 \to s_2 \to \ldots$ in cui è vero (vale q fino a s_1, poi vale sempre r).
- ho il path $s_0 \to s_2 \to s_2 \to \ldots$ in cui è vero (vale q in s_0 e poi vale r in s_2)

Gli altri path sono una variazione dei due presentati, quindi si conclude che è $q\,U\,r$ è vera per il modello.

$\mathcal{M}, s_0 \models G\,(p \vee q)$ Non è vera, infatti in s_2 vale solo r (non vale $p \vee q$), come controesempio si mostra il path

$$\underbrace{s_0 \to s_1 \to s_0 \to s_1}_{\text{vale } p \vee q} \quad \to \quad \underbrace{s_2 \to s_2 \to s_2}_{\text{non vale } p \vee q} \to \ldots$$

Dato che la proprietà non vale per ogni possibile path che parte con s_0, allora non può valere per il modello $\mathcal{M}$.

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: $\mathrm{G}\,((\vartheta \wedge \neg\psi \wedge \mathrm{F}\psi) \to (\neg\phi \mathrm{U}\psi))$e

- ϕ può diventare vera:
  - globalmente: $\mathrm{F}\phi$
  - prima di ψ: $\neg\psi\mathrm{W}\left(\phi \wedge \neg\psi\right)$
  - dopo ϑ: $\mathrm{G}\left(\neg\vartheta\right) \vee \mathrm{F}\left(\vartheta \wedge \mathrm{F}\phi\right)$
  - tra ϑ e ψ: $\mathrm{G}\left(\left(\vartheta \wedge \neg\psi\right) \rightarrow \left(\neg\psi\mathrm{W}\left(\phi \wedge \neg\psi\right)\right)\right)$
  - dopo ϑ e fino a ψ: $\mathrm{G}\left(\left(\vartheta \wedge \neg\psi\right) \rightarrow \left(\neg\psi\mathrm{U}\left(\phi \wedge \neg\psi\right)\right)\right)$
- ϕ è sempre vera:
  - globalmente: $\mathrm{G}\phi$
  - prima di ψ: $\mathrm{F}\psi \rightarrow \left(\phi\mathrm{U}\psi\right)$
  - dopo ϑ: $\mathrm{G}\left(\vartheta \rightarrow \mathrm{G}\phi\right)$
  - tra ϑ e ψ: $\mathrm{G}\left(\left(\vartheta \wedge \neg\psi \wedge \mathrm{F}\psi\right) \rightarrow \left(\phi\mathrm{U}\psi\right)\right)$
  - dopo ϑ e fino a ψ: $\mathrm{G}\left(\left(\vartheta \wedge \neg\psi\right) \rightarrow \left(\phi\mathrm{W}\psi\right)\right)$

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. Legge di De Morgan
   $\neg\left(\phi \wedge \psi\right) \equiv \neg\phi \vee \neg\psi$
   $\neg\left(\phi \vee \psi\right) \equiv \neg\phi \wedge \neg\psi$
2. Dualità
   $\neg\mathrm{G}\phi \equiv \mathrm{F}\neg\phi$
   $\neg\mathrm{F}\phi \equiv \mathrm{G}\neg\phi$
   $\neg\mathrm{X}\phi \equiv \mathrm{X}\neg\phi$
3. $\neg\left(\phi\mathrm{U}\psi\right) \equiv \neg\phi\mathrm{R}\neg\psi$
   $\neg\left(\phi\mathrm{R}\psi\right) \equiv \neg\phi\mathrm{U}\neg\psi$
4. Distributività
   $\mathrm{F}\left(\phi \vee \psi\right) \equiv \mathrm{F}\phi \vee \mathrm{F}\psi$ (**nota:** F è distributivo rispetto a $\vee$, ma non rispetto a $\wedge$).
   $\mathrm{G}\left(\phi \wedge \psi\right) \equiv \mathrm{G}\phi \wedge \mathrm{G}\psi$ (**nota:** G è distributivo rispetto a $\wedge$, ma non rispetto a $\vee$).
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. $\phi\mathrm{W}\psi \equiv \psi\mathrm{R}\left(\phi \vee \psi\right)$
   $\phi\mathrm{R}\psi \equiv \psi\mathrm{W}\left(\phi \wedge \psi\right)$

Teorema Per qualsiasi formula LTL ϕ e ψ vale l'equivalenza:

$$\phi\mathrm{U}\psi \equiv \neg\left(\neg\psi\mathrm{U}\left(\neg\phi \wedge \neg\psi\right)\right) \wedge \mathrm{F}\psi$$

.

Esempio: *mutua esclusione*

Quando un processo concorrente condivide una risorsa (Es: file, cartelle, ecc.) a volte risulta necessario assicurarsi che nessun altro processo possa accerderci nello stesso momento: non è desiderabile avere più processi che modificano contemporameneamente lo stesso file.
Le proprietà a cui siamo interessati sono:

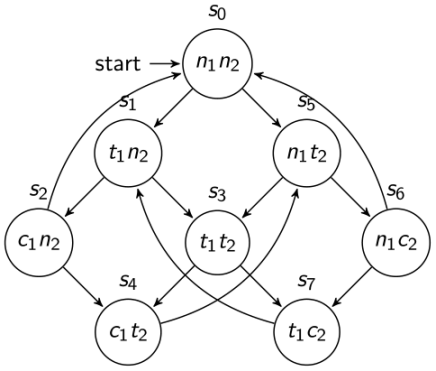
Safeness un solo processo alla volta nella sezione critica.

Liveness ogni processo che richiede di entrare nella sezione critica prima o poi avrà il permesso di accederci.

Non bloccante un processo può sempre richiedere di entrare nella sua sezione critica.

Non strettamente sequenziale i processi non devono entrare nella sezione critica in un modo sequenziale.

Si può modellare il processo di richiesta di una risorsa da parte di due processi come segue:



Dove

n_i il processo i non è nella sezione critica

t_i il processo i cerca di entare nella sezione criticasia

c_i il processo i è nella sezione critica

Verifichiamo le proprietà

Safeness si richiede che due processi non siano mai contemporanemente nella sezione critica, quindi $\neg G(c_1 \wedge c_2)$, che è vera (infatti non esiste uno stato che ha sia c_1 che c_2).

Liveness si rappresenta "un processo che richiede la sezione otterrà (nel futuro) il permesso di entrarci" come $t_1 \rightarrow F\,c_1$ e dato che deve valere sempre si utilizza l'operatore Globally: $G\,(t_1 \rightarrow F\,c_1)$. Si nota che è falsa, infatti non vale in tutti i percorsi:

$$s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_7 \rightarrow s_1 \rightarrow s_3 \rightarrow s_7 \rightarrow s_1 \ldots$$

questo a causa del loop composto da s_1, s_3 e s_7 dove continuerà a valere t_1 ma mai c_1.

Non bloccante non è rappresentabile in LTL, infatti richiede l'esistenza di un percorso, e in LTL questo è quasi impossibile da provare (vedi paragrafo successivo).

Non strettamente sequenziale basta trovare un path in cui non ci sia strict sequencing.

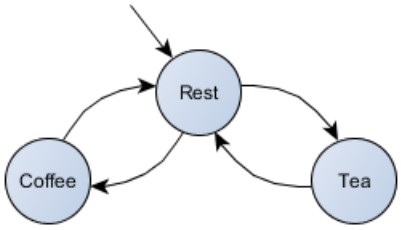
Limiti Esistono affermazioni impossibili da esprimere tramite LTL; ciò deriva dal fatto che la logica LTL non può affermare l'esistenza di path con determinate caratteristiche (es.: dallo stato attuale, esiste un path per cui vale ϕ).

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.

Esempio: *limiti del LTL*

Sia il seguente modello che rappresenta una macchina che distribuisce bevande:

- quando è nello stato `Tea` vuol dire che sta erogando thé
- quando è nello stato `Coffee` vuol dire che sta erogando caffé
- quando è nello stato `Rest` vuol dire che è in attesa di input



Vorrei applicare la regola "prima o poi arriva il the" ma è difficile da esprimere: l'espressione $F\ tea$ non va bene perché può esserci un percorso in cui si eroga solo caffè

$$coffe \rightarrow rest \rightarrow coffe \rightarrow \dots$$

Con $F\ tea$ si sta chiedendo che la proprietà tea valga in ogni percorso, mentre la richiesta era se esiste un percorso che soddisfa la proprietà.

4.2 Computation Tree Logic (CTL)

La CTL è una logica con connettivi che ci permette di specificare proprietà temporali: è una logica branching-time quindi i suoi modelli sono rappresentabili mediante una struttura ad albero in cui il futuro non è deterministico: esistono differenti computazioni o paths nel futuro e uno di questi sarà il percorso realizzato. La grande differenza tra CTL e LTL è la seguente:

- in LTL una proprietà specificata deve valere per ogni singolo percorso;
  Es: se scrivo $X\ a$ sto imponendo che per ogni percorso allo stato successivo valga a.
- in CTL è si può decidere se la proprietà deve valere per ogni singolo percorso oppure per almeno un percorso: si introducono le parole chiave A (All path) e E (Exists path) accostate agli operatori già visti per LTL (X, F, G, ecc).

Esempio: utilizzo di CTL

Riprendiamo la macchina che distribuisce bevande vista a pagina 68, la regola "prima o poi arriva il the" è espressa con

$$EF\ tea$$

che significa "esiste un percorso (E) per cui nel futuro (F) la proprietà tea è soddisfatta".

Operatori

- Connettori logici classi

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

- Connettori logici temporali
  - La proprietà deve valere per tutti i percorsi (All)

AX	("neXt state") AXa significa che a sarà vera in tutti i prossimi possibili 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
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
AU	("Until") A $[\phi \mathrm{U} \psi]$[I] significa che ϕ è per forza vero finché non diventa vero ψ, per ogni possibile path che parte dallo stato attuale

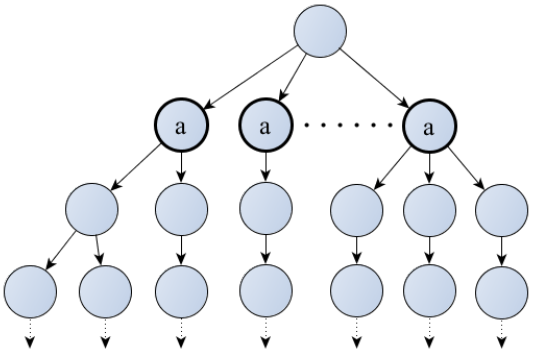
  - Esiste almeno un percorso che soddisfa la proprietà (Exists)

EX	("neXt state") EXa significa che a sarà vera per almeno uno dei prossimi stati
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
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
EU	("Until") E $[\phi \mathrm{U} \psi]$[II] significa che ϕ è per forza vero finché non diventa vero ψ, per almeno uno dei possibili path che partono dallo stato attuale

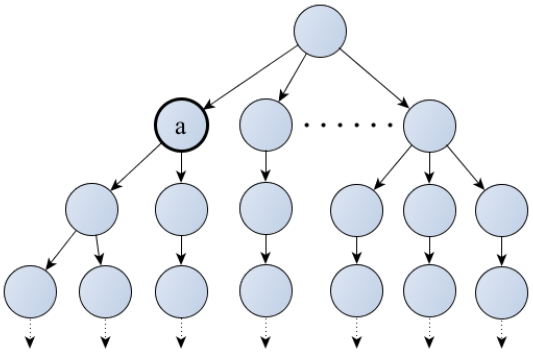
Da notare che i comandi AX, FX, ecc. sono uguali ai comandi X, F, ecc di LTL.

[I] Si può scrivere anche AU (ϕ, ψ) o ϕAUψ.
[II] Si può scrivere anche EU (ϕ, ψ) o ϕEUψ.

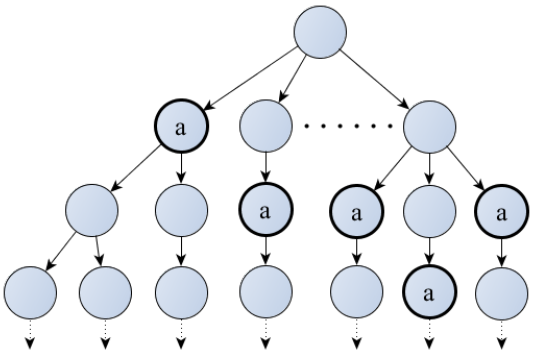
$\mathcal{M}, s \models AX\, a$



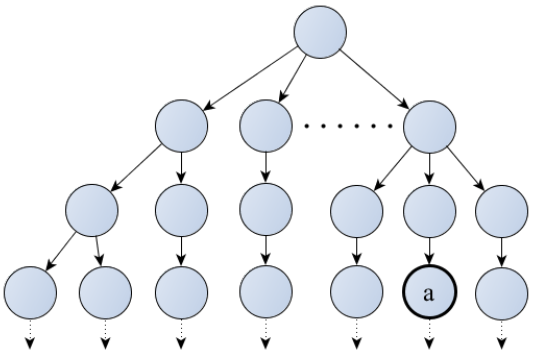
$\mathcal{M}, s \models EX\, a$



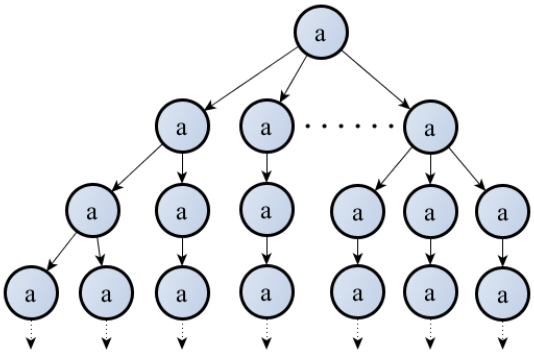
$\mathcal{M}, s \models AF\, a$



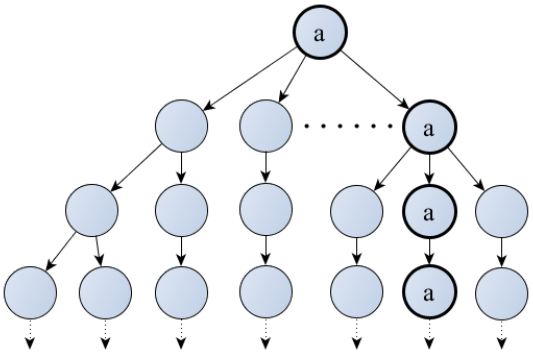
$\mathcal{M}, s \models EF\, a$



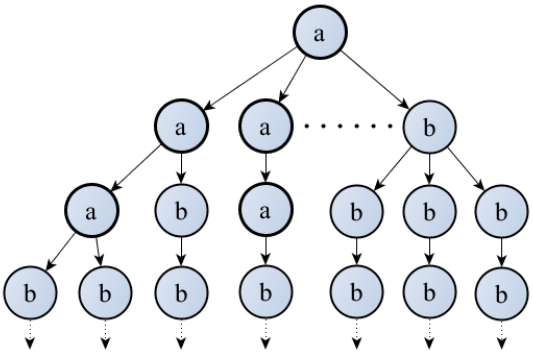
$\mathcal{M}, s \models AG\, a$



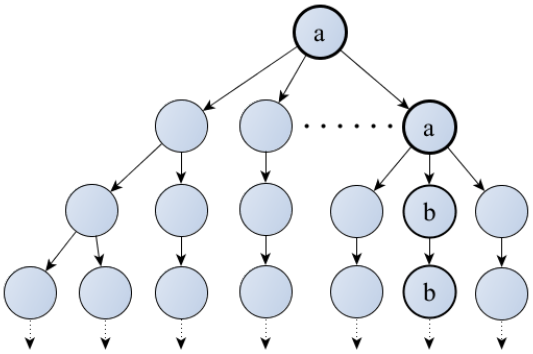
$\mathcal{M}, s \models EG\, a$



$\mathcal{M}, s \models a\, AU\, b$



$\mathcal{M}, s \models a\, EU\, b$



Priorità degli operatori Dalla più alta alla più bassa:

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

Esempio: *CTL*

Prendiamo un modello che gestisce le sbarre di un passaggio a livello, si chiede che sia impossibile raggiungere uno stato in cui il treno passa e le sbarre non sono chiuse.
Se si indicano le proprietà "il treno passa" con T e "le sbarre sono chiuse" con C allora una prima soluzione potrebbe essere

$$AG(T \wedge \neg C)$$

Però si costringe a far passare sempre il treno (T dovrà valere in ogni stato).
Una soluzione più corretta è

$$AG(T \rightarrow C)$$

Infatti l'implicazione può essere scritta come $\neg T \vee C$ quindi

- Se il treno non passa (T falso) l'implicazione è vera;
- Se il treno passa, l'implicazione è vera solo se C è vera (cioè le sbarre sono abbassate).

In alternativa si poteva usare EF nel seguente modo

$$\neg EF(T \wedge \neg C)$$

che significa "non può esistere uno stato nel futuro per cui il treno passa e la sbarre non sono abbassate".

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.

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 $\phi_1 \mathrm{U} \phi_2$[III]
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: $\mathrm{AG}((\vartheta \wedge \neg\psi) \to \mathrm{A}[\neg\psi \mathrm{U}(\phi \wedge \neg\psi)])$

[III] $\phi_1 \mathrm{U} \phi_2$ 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. $\mathrm{AG}\phi \equiv \phi \wedge \mathrm{AXAG}\phi$
   $\mathrm{EG}\phi \equiv \phi \wedge \mathrm{EXEG}\phi$
4. $\mathrm{AF}\phi \equiv \phi \vee \mathrm{AXAF}\phi$
   $\mathrm{EF}\phi \equiv \phi \vee \mathrm{EXEF}\phi$
5. $\mathrm{A}\left[\phi\mathrm{U}\psi\right] = \psi \vee (\phi \wedge \mathrm{AXA}\left[\phi\mathrm{U}\psi\right])$
   $\mathrm{E}\left[\phi\mathrm{U}\psi\right] = \psi \vee (\phi \wedge \mathrm{EXE}\left[\phi\mathrm{U}\psi\right])$

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

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.

Inoltre permette la descrizione di modelli asincroni e sincroni, con descrizioni modularizzate o gerarchiche.

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.

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.

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 ...;
...
```

Inizializzazione Si inizializzano le variabili scrivendo `VAR` seguito dalla notazione `nome_variabile: tipo;` 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);

Nota: `VAR` è scritto solo una volta all'inizio della porzione di codice dove si dichiareranno le variabili.

Dichiarazioni Esistono tre tipi di assegnamento

- `ASSIGN:` porzione di codice dove sono inizializzate le variabili e si descrive la loro evoluzione nel modello
  - Con `init(nome_variabile) := valore;` si dichiara il valore iniziale di una variabile;
  - Con `next(nome_variabile) := valore;` si definisce che valori assumerà quella variabile negli stati successivi (cioè si descrivono le transizioni);
  - Esistono anche assegnamenti allo stato attuale (invarianti) che si dichiarano con `nome_variabile := valore;`

    Nota: è consigliabile usare init-next oppure l'invariante, mai tutti e due insieme.

– Si può utilizzare l'espressione `case`, la cui sintassi è

```
case
espressione_logica1:  valore1;
espressione_logica2:  valore2;
...
TRUE: valore_deafult;
esac;
```

Si iniziano ad analizzare le espressioni partendo dalla prima: appena si trova un'espressione vera si assegna il valore corrispondente alla variabile.

Nota: se nessuna delle condizioni è vera, il risultato è scelto arbitrariamente.

- `DEFINE`: ogni valore della variaible è sostituito con l'espressione con cui è stata dichiarata. Si utilizza la notazione `DEFINE nome_variabile:= valore;`

  Nota: l'espressione può essere formata da altre variabili, ma si deve stare attenti a non creare riferimenti circolari.

Esempio: *riferimenti circolari*

Il seguente codice non verà eseguito perché ha creato dei riferimenti circolari:

```
DEFINE a:= c + 2;
DEFINE b:= a + 1;
DEFINE c:= b + 3;
```

Infatti quando `a` è dichiarata va a cercare il valore di `c` per sostituirlo al suo interno:

```
a:= (b + 3) + 2;
```

ma essendo anche `c` composta da `b` si espenda anche `b`,ma ha all'interno `a`, quindi non si riesce a fare la sostituzione:

```
a:= ( (a + 1) + 3 ) + 2;
```

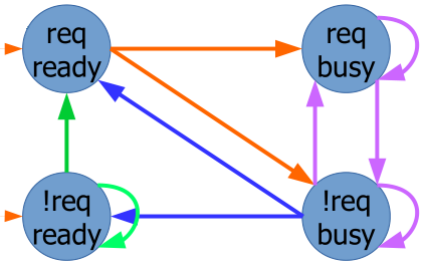
Nota:

- `VAR` e `DEFINE` sono legati alla staticità del modello
  - `VAR` si dichiarano stati e proprietà
  - `DEFINE` si indicano le proprietà valide in ogni stato);
- `ASSIGN` è legato alla dinamica (si dichiarano i valori iniziali e le transizioni)

Esempio: *modellazione di un singolo processo*

Sia il seguente modello, in cui

- il sistema inizialmente è ready
- se arriva una richiesta va a busy
- dopo un certo tempo t torna ready (se non c'è una ulteriore richiesta)



Si dichiara il modulo

```
MODULE main
```

Si dichiarano le variabili, in questo caso la descrizione dello stato (pronto o occupato) e una variabile che controlla quando arriva una richiesta.

```
VAR
request:  boolean;
state:  {ready, busy};
```

All'inizio il processo è pronto quindi lo stato iniziale è `ready`, mentre il prossimo stato dipende se è stata o meno fatta una richiesta:

- se lo stato è `ready` e `request` è vera, allora il processo dovrà accogliere la richiesta e quindi il prossimo stato sarà `busy`;
- se lo stato è `busy` e `request` è vera, allora il processo sta ancora analizzando la richiesta, quindi anche il prossimos stato sarà `busy`;
- se lo stato è `ready` e `request` è falsa, allora il processo non ha richiesta da analizzare quindi il prossimo stato sarà `ready`;
- Nel caso non ci si trovi in nessuno dei tre casi, allora verrà scelto arbitrariamente se il processo è occupato o pronto.

Rappresentiamo tutto in una tabella

Condizione		Prossimo stato
stato attuale	altre condizioni	
`ready`	`request` è vera	`busy`
`busy`	`request` è vera	`busy`
`ready`	`request` è falsa	`ready`
altro		scelta tra `busy` e `ready`

```
ASSIGN
  init(state) := ready;
  next(state) :=
    case
      state=ready & request:  busy;
      state=busy & request:  busy;
      state=ready & not request:  ready;
      TRUE: {ready, busy};
    esac;
```

Come regola generale per definire le transazioni si identifica in che stato ci si trova (Es: `state=ready`) seguito dalle condizioni (proposizioni vere) dello stato successivo.

Specifiche LTL e CTL Per specificare una formula LTL si utilizza il formato `LTLSPEC (exp)` dove `exp` sono le formule scritte in linguaggio LTL viste nella sezione precedente.
Analogalmente per CTL si utilizza `CTLSPEC (espressione_ctl)`.

Esempio: *espressioni LTL*

Riprendiamo la macchina che distribuisce bevande vista a pagina 68: è un modello molto semplice quindi le variabili sono solo i tre stati presenti nel modello.

Si parte dallo stato `rest`, mentre le transizioni si possono rappresenare come:

- se sono pronto, scegli arbitrariamente se preparare caffé o thé
- appena finisci torna a `ready`.

Quindi il codice fino ad ora sarà:

```
MODULE main
VAR
  status = {ready, coffe, tea};

case
  ready:  {coffe, tea};
  TRUE: ready;
esac
```

Se si volesse controllare se esiste un percorso in cui il *tea* è rispettata si scrive

```
CTL (EF tea);
```

Se si volesse controllare se in tutti i percorsi almeno una delle tre condizione è rispettata si scrive

```
LTLSPEC (F tea | coffe | rest);
```

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.

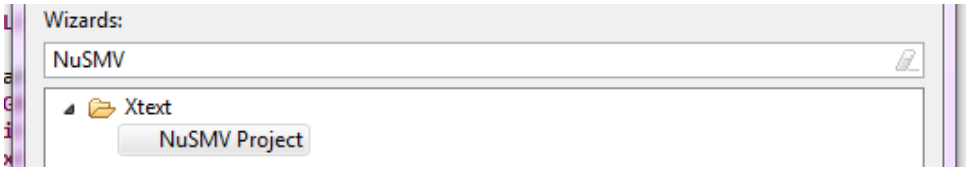
La variabile `running` indica appunto se il modulo (nel quale ne viene controllato il valore e che è stato dichiarato `process`) è in esecuzione.

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).

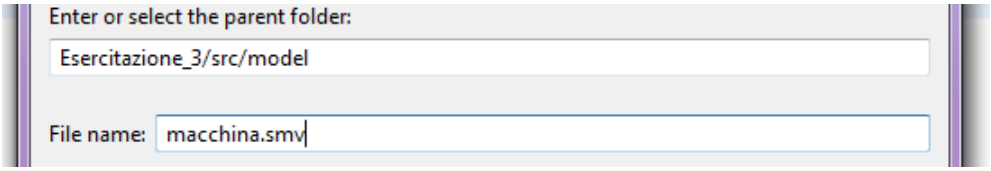
N.B.: un fallimento nel cercare un cotroesempio potrebbe voler dire che non esistono controesempi di lunghezza inferiore o uguale alla soglia, ma non che non ne esistono in assoluto.

4.3.1 NuSMV in Eclipse

Creare un nuovo progetto Andare su File→ New→Other... e scegliere "NuSMV Project" dalla finestra



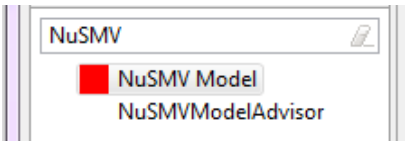
Creare un nuova macchina a stati Creare un package chiamato "model" in cui andranno tutte le macchine a stati, poi cliccare tasto destro sul progetto e andare su New → File, ricordarsi di mettere l'estensione ".smv" al file da creare e di selezionare la cartella in cui il file verrà creato. Infine cliccare su "Finish".



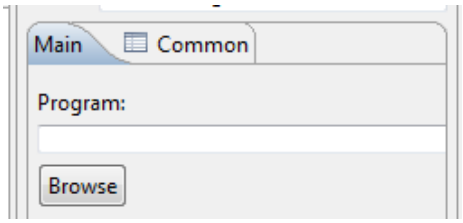
Eseguire la macchina a stati Cliccare tasto destro sul file .smv andare su Run as → NuSMV, nel caso vengenao segnalati degli errori guardare il paragrafo successivo.

Configurare NuSMV

- Scaricare il file NuSMV.exe
  - Andare sul sito sito `http://nusmv.fbk.eu/NuSMV/download/getting-v2.html` e cliccare su “NuSMV binaries”;
  - Verrà chiesto di registarsi, cliccare su “Do not registrer” per saltare la registrazione;
  - Nella pagina a cui si verrà indirizzati e scaricare il file “NuSMV-2.5.4-i386-pc-mingw32” o “NuSMV-2.5.4-i386-pc-mingw64” a seconda se si ha eclipse a 32 o 64 bit;
  - Cliccare sul file scaricato ed installarlo
    - Quando chiederà in che cartella installarlo evitate cartelle in cui servono privilegi di amministratore per scriverci (Es: C:\Programmi) perché darà errore.
- Per collegare il programma appena installato ad eclipse:
  - Cliccare con il tasto destro sul progetto e fare Run as → Run Configuration e cercare nel menù sulla sinistra “NuSMV Model”;



  - Cliccare sull’incona  e cancellare dalla textbox “NuSMV” (se no non comparirà la finestra di configurazione);
  - Nella schermata che apparirà cliccare su “Browse”, quindi selezionare il file NuSMV.exe (si troverà nella cartella “bin” all’indirizzo a cui avete installato NuSMV)



- Infine cliccare su “Run”.
- Si dovrebbe essere in grado di eseguire la macchina con tasto destro su file Run as → NuSMV.

Nota: , se no non comparirà la finestra di configurazione.

4.4 CTL*

CTL* è una logica che combina i poteri espressivi di LTL e CTL: ci sono operazione che può fare solo CLT e non LTL e viceversa.

Esempio: *limiti del CTL e LTL*

Per LTL si è già visto che non si può esprire "Esiste un percorso?" (vedi esempio a pagina 68), mentre per CTL è difficile esprimere "tutti i percorsi che avranni p hanno anche q", infatti in LTL si può scrivere

$$F\,p \to F\,q$$

Mentre in CLT non basta sostituire F con AF o EF, infatti:

$AF\,p \to AF\,q$	Obbliga p ad essere vera nel futuro di tutti i percorsi, mentre invece era solo richiesto se esisteva allora c'era anche q;
$AF\,p \to EF\,q$	Come sopra, inoltre inoltre $EF\,q$ non obbliga che q ad essere vero ogni volta che p è vero;
$EF\,p \to AF\,q$	p deve esistere in tutti i percorsi, quindi non va bene;
$EF\,p \to EF\,q$	Anche se non obbliga p ad essere sempre vera, la $EF\,q$ da gli stessi problemi che dava per $AF\,p \to EF\,q$.

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 $[\alpha]$ (quindi LTL è più espressivo).

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[IV]

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.

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

4.5 Algoritmi di Model Checking

Vogliamo risolvere il problema $\mathcal{M}, s \overset{?}{\models} \phi$ (cioè dato un modello, la formula ϕ vale?).

Vedremo solo il **Labeling Algorithm** per formule CTL.

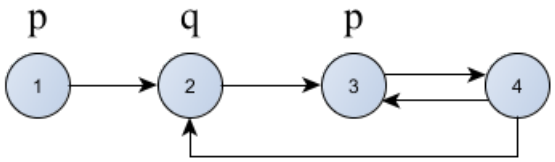
Tale algoritmo accetta in ingresso un modello $\mathcal{M} = (S, \to, L)$ e una formula CTL φ, e dà come output l'insieme degli stati di $\mathcal{M}$ che soddisfa φ.

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:

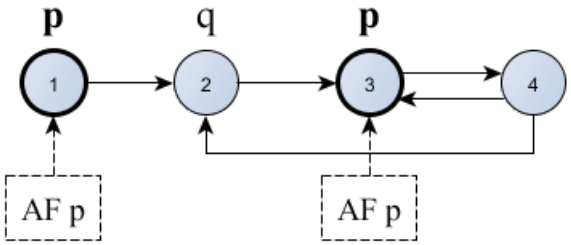
- se $\varphi = \bot$ allora nessuno stato viene etichettato con $\bot$;
- se $\varphi = p$ allora vanno etichettati con φ tutti gli stati s tali che $p \in L(s)$;
- se $\varphi = \neg\psi$:
  - eseguire l'algoritmo di labeling per ψ;
  - etichettare con $\neg\psi$ tutti gli stati s non etichettati con 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;

Esempio: *labeling AF*

Sia il seguente schema dove è già stato fatto il labeling per p e q:



Si farà il labeling per $AF\,p$: dove vale p si mette anche $AF\,p$.



Ora si guarda di stato in stato per vedere se etichettarli come $AF\,p$

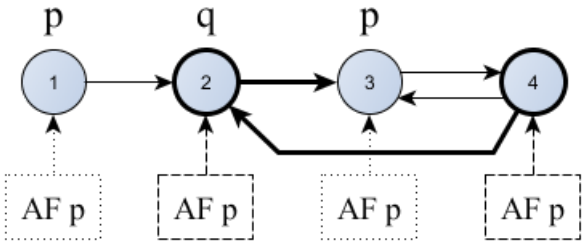
(1) Già etichettato con $AF\,p$

(2) Ho la transizione (2) $\rightarrow$ (3) e quindi avendo nello stato (3) $AF\,p$ allora anche (2) è etichettato.

(3) Già etichettato con $AF\,p$

(4) Come prima, ho (4) $\rightarrow$ (2) con (2) etichettato $AF\,p$, quindi anche (4)è etichettato.

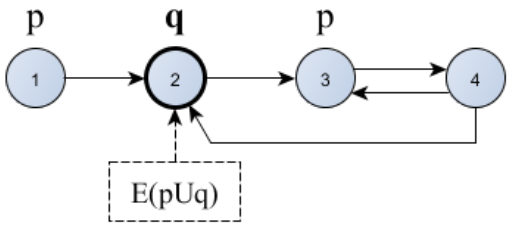
Si è ottenuto il seguente grafico:



- se $\varphi = \mathrm{E}\,[\psi_1 \mathrm{U} \psi_2]$:
  - 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.

Esempio: *labeling EU*

Riprendiamo lo stesso schema dell'esercizio precedente ed etichettiamo per $E(p\,U\,q)$: si inizia mettendo l'etichetta $E(p\,U\,q)$ a tutti gli stati con q.



Cerco negli stati etichettati con p se esiste uno stato successivo etichettato con $E(p\,U\,q)$: in caso affermativo, anche lo stato con p è etichettato.

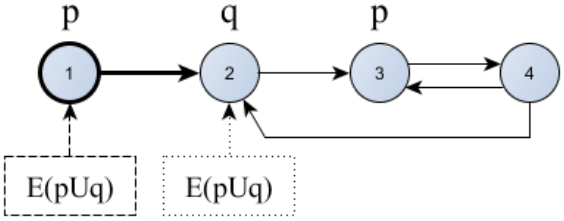
(1) Ho (1) → (2) con (2) etichettato, quindi si etichetta anche (1)

(2) Non ha p

(3) Non esiste uno stato successivo etichettato con $E(p\,U\,q)$

(4) Non ha p

Quindi si è ottenuto il seguente grafico:



Ovviamente l'algoritmo restituisce tutti gli stati etichettati con φ.
La complessità dell'algoritmo è

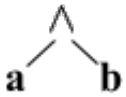
$$\mathcal{O}(f \cdot V \cdot (V + E))$$

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.

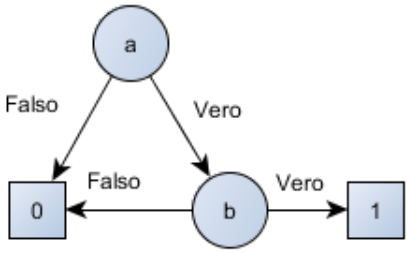
L'algoritmo è lineare rispetto alle dimensioni sia di φ che di $\mathcal{M}$, tuttavia la dimensione di quest'ultimo è molto spesso esponenziale rispetto al numero di variabili e al munero di componenti del sistema che eseguono in parallelo; questo problema è noto come il **problema di esplosione degli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

Una possibile soluzione è utilizzare le strutture dati BDD (Binary Decision Diagram) che utilizzano una versione grafica alternativa all'albero (utilizzato nella logica proposizionale) per rappresentare le proposizioni.

Espressione: $a \wedge b$



Albero della logica proposizionale



Binary Decision Diagram

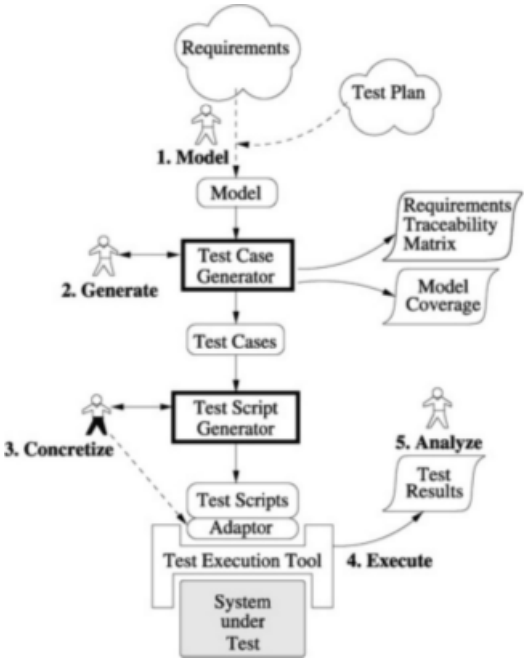
5 Testing Model-based

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;
   - verificare la 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 i 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 comunque 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 (chiamati blocchi b_i) 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, cioè nessuna parte vuota e

completezza l'unione dei blocchi deve dare come risultato tutto il dominio, cioè $b_1 \cup \ldots \cup b_n = D$

disgiunzione tutte le parti sono disgiunte a coppie: non si devono avere due elementi (input) uguali in due test diversi cioè $b_i \cap b_j = \emptyset$

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.

Esempio: *partition testing*

Abbiamo la seguente funzione di ricerca:

```
public int search (List<T> L, T element) { ··· }
```

Per scegliere gli input, un primo approccio può essere:

b_1 primo elemento della lista ⇒ `element = L.get(0)`

b_2 ultimo elemento della lista ⇒ `element = L.get(L.size - 1)`

b_3 elemento in mezzo ⇒ `element = L.get(i) AND i > 0 AND i < L.size - 1`

Ma non è ancora un partitional testing, infatti non rispetta le proprietà di

Completezza: mancano dei casi di test (Es: `element` non appartiene alla lista);

Disgiunzione: se la lunghezza della lista è 1 allora i test b_1 e b_3 hanno gli stessi input.

Detto questo, si creano i test prestando particolare attenzione alla lunghezza della lista:

- Lunghezza della lista diversa da 1

  b_1 primo elemento della lista ⇒ `L.size > 1 AND element = L.get(0) AND element != L.get(0)`

  b_2 ultimo elemento della lista ⇒ `L.size > 1 AND element = L.get(L.size - 1) AND element != L.get(L.size - 1)`

  b_3 elemento in mezzo ⇒ `L.size > 1 AND element = L.get(i) AND i > 0 AND i < L.size - 1`

  b_4 elemento non trovato ⇒ `L.size > 1` e non esiste un `i` tale che `element = L.get(i)`

- Lunghezza della lissa uguale a 1

  b_5 primo elemento della lista ⇒ `L.size > 1 AND element = L.get(0)`

  b_6 elemento non trovato ⇒ `L.size > 1 AND element != L.get(0)`

- Lista vuota

  b_7 elemento non trovato perché la lista è vuota ⇒ `L.size = 0` e e non esiste un `i` tale che `element = L.get(i)`

5.1.3 Dal partition testing al combinatorial testing[I]

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.

[I] 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).

Esempio: *differenza tra interface-based e functionality-based*

Sia la seguente funzione che ritorna il numero di soluzioni di un'equazione di seconda grado dati i coefficenti del polinomio:

```
public int numSolutions(int a, int b, int c){
  double delta = b - 4*a*c;
  if (delta < 0) return 0;
  if (delta == 0) return 1;
  return 2;
}
```

Vediamo come cambiano gli input a seconda dell'approccio:

Interface non si guarda cosa fa la funzione ma solo i valori di `a`, `b` e `c`, quindi un tipo di input può essere la combinazione tra valori maggiori, uguali o minore di zero (tranne per `a` che deve essere diverso da 0).

- b_1 maggiori di zero: `a = 4; b = 5; c = 7; numSolutions(4, 5, 7) = 0`
- b_2 uguali a zero: `a = 1; b = 0; c = 0; numSolutions(1, 0, 0) = 1`
- b_3 minori di zero: `a = -5; b = -3; c = -1; numSolutions(-5, -3, -1) = 0`

Functionality si guarda cosa fa la funzione e si cerca di coprire tutti i possibili output, in questo caso si cerca di coprire ogni `return`.

- b_1 ritorna 0 soluzini: `a = 4; b = 5; c = 7; numSolutions(4, 5, 7) = 0`
- b_2 ritorna 1 soluzini: `a = 1; b = 0; c = 0; numSolutions(1, 0, 0) = 1`
- b_3 ritorna 2 soluzini: `a = -5; b = -3; c = +2; numSolutions(1, 0, 0) = 2`

Si fa notare come con functionality-based si è ottenuta una copertura al 100% delle istruzioni.

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.

Esempio: *Pair-wise coverage*

Abbiamo tre variabili a, b, c ognuna delle quali può assumere i valori -1, 0 oppure 2.
Se si facesse un ACoC servirebbero $3^3 = 27$ combinazioni, mentre con PWC ne servono meno: ogni combinazioni di coppie ha $2^3 = 8$ elementi, ma combinando le tre coppie tra loro si ottengono meno di $2^3 \cdot 3 = 24$ casi di test.
Si vedrà in seguito con quale algoritmo si possono ricavare queste coppie.

T-Wise Coverage (TWC) Estende il concetto del PWC a tuple di valori di parametri. È più costoso del PWC ma apparentemente non ne migliora l'efficacia.
Se tutti i parametri hanno la stessa lunghezza, allora il numero di test da eseguire è $[\max(p_i)]^t$ con $i = 1, \ldots, P$.

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 (crescita orizzontale);
  3. se necessario, si aggiungono casi di test per coprire le tuple mancanti (crescita verticale).

  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.

Esempio: *Metodi search-based parameter-based*

Abbiamo i parametri con i seguenti valori:

$$A = \{ A_1, A_2 \} \quad B = \{ B_1, B_2 \} \quad C = \{ C_1, C_2, C_3 \}$$

Si parte combinando le coppie di A e B: $\Rightarrow$ Poi si aggiungono arbitrariamente i valori di C:

A	B	C
A_1	B_1	
A_1	B_2	
A_2	B_1	
A_2	B_2	

A	B	C
A_1	B_1	C_1
A_1	B_2	C_2
A_2	B_1	C_3
A_2	B_2	C_1

Adesso bisogna aggiungere i casi di test mancanti, quindi si elencano le coppie (A, C) e (B, C)

$$(A_1, C_1),\ (A_1, C_2),\ (A_1, C_3),\ (A_2, C_1),\ (A_2, C_2),\ (A_2, C_3)$$
$$(B_1, C_1),\ (B_1, C_2),\ (B_1, C_3),\ (B_2, C_1),\ (B_2, C_2),\ (B_2, C_3)$$

e si barrano quelle già inserite:

$$\cancel{(A_1, C_1)},\ \cancel{(A_1, C_2)},\ (A_1, C_3),\ \cancel{(A_2, C_1)},\ (A_2, C_2),\ \cancel{(A_2, C_3)}$$
$$\cancel{(B_1, C_1)},\ (B_1, C_2),\ \cancel{(B_1, C_3)},\ \cancel{(B_2, C_1)},\ \cancel{(B_2, C_2)},\ (B_2, C_3)$$

Quindi si aggiungono i casi di test mancanti:

A	B	C
A_1	B_1	C_1
A_1	B_2	C_2
A_2	B_1	C_3
A_2	B_2	C_1
$\boldsymbol{A_1}$	$\boldsymbol{B_2}$	$\boldsymbol{C_3}$
$\boldsymbol{A_2}$	$\boldsymbol{B_1}$	$\boldsymbol{C_2}$

5.2 CitLab

Si utilizza Citlab per fare testing combinatoriale in Eclipse.

- Per installarlo andare su `http://svn.code.sf.net/p/citlab/code/updatesite/`
- Esempi di sintassi: `http://sourceforge.net/p/citlab/wiki/CitLabLanguage/`

5.2.1 Introduzione al linguaggio

Parametri I tipi base di parametri in CitLab sono:

- `Enumerative`: come `enum` è una lista di valori simbolici;

  Esempio: `Enumerative display {16MC 8MC BW};`

- `Boolean`: come `boolean`.

- `Range`: rappresenta un intervallo di interi, è possibile specificare la distanza (`step`) tra un numero e l'altro. Lo `step` di default è 1;

  Esempio: per rappresentare l'intervallo di interi $\{10; 15; 20; 25; 30\}$ (vado da 10 a 30 ogni 5 numeri), si scrive:
  `Range textLines [ 10 ..  30 ] step 5;`

- `Numerbs`: rappresenta una lista di numeri interi (è come un `Enumerative` ma permette operazioni aritmetiche)

  Esempio: Numbers year {2012 2013};

Creare un tipo Per creare un tipo si "estende" un tipo base:

```
Types:
  tipo_esteso nome_nuovo_tipo { valori_tipo };
end
```

Inizializzazione Si inizializzano le variabili nell'ambiente `Parameters`. Ci sono due modi per inizializzare le variabili:

- Se il tipo è base la sintassi è uguale a quella di Java:

  ```
  tipo_padre nome_variabile
  ```

- Se il tipo è stato creato si utilizza la sintassi

  ```
  tipo_base nome_variabile :  tipo_creato
  ```

  dove `tipo_base` è il tipo che è stato esteso quando si è dichiarato il nuovo tipo (`tipo_creato`)

  ```
  Parameters:
  tipo_padre nome_variabile;
  tipo_base nome_variabile :  tipo_creato;
  end
  ```

Esempio: *tipi in CitLab*

Si crea il tipo `cameraType` che "estende" il tipo `Enumerative` e assume i valori 2 megapixel (`2MP`), un megapixel (`1MP`) oppure nessuna fotocamera (`NOC`).

```
Types:
  EnumerativeType cameraType { 2MP 1MP NOC };
end
```

Si dichiarano le variabili `rearCamera` (fotocamera posteriore) e `isReady` (stato della telecamera):

```
Parameters:
  Enumerative rearCamera :  cameraType;
  Boolean isReady;
end
```

Vincoli Tutti i vincoli sulle variabili sono contenuti nella sezione Constraints, indicata come

```
Constraints:  # vincolo # end
```

dove `vincolo` è un'espressione logica che deve essere sempre vera. Ogni vincolo deve essere compreso tra #.

Esempio: vincoli in CitLab

Modelliamo i vincoli: "se lo schermo è nero e bianco, allora il telefono non può avere una fotocamera" e "se c'è la telecamera allora è sempre disponibile"

```
Constraints:
  # display == display.BW => rearCamera == cameraType.NOC #
  # !cameraType.NOC => isReady #
end
```

Test Si possono aggiungere dei casi di test manualmente: CitLab presuppone che i casi di test siano già stati eseguiti quindi l'algoritmo di generazione di casi di test utilizzerà i test inseriti per ridurre la ridondanza. I casi di test sono aggiunti come lista di valori delle variabili nella sezione Seeds:

```
Seeds:# lista_valori # end
```

Come per i vincoli, ogni caso di test deve essere compreso tra #

Esempio: *test manuale in CitLab*

```
Seeds:
  # emailViewer=false , display=display.16MC, frontCamera=cameraType.NOC, year =
2012, rearCamera=cameraType.2MP, textLines=30 #
end
```

Goal Si possono aggiungere ulteriori vincoli ai test (utili per analizzare le situazioni critiche), indicati tra #

Esempio: *goal in CitLab*

Se si vuole essere sicuri che la test suite abbia almeno un test in cui il telefono non ha fotocamera (né anteriore né posterioe) si scrive:

```
Goal
  # rearCamera == cameraType.NOC or frontCamera == cameraType.NOC #
end
```

Commenti Si commenta con `//` e `/* ··· */`

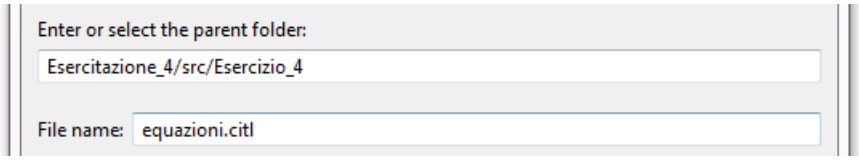
Nota: Goal e Seeds non sono sempre supportati.

5.2.2 CitLab in Eclipse

Creazione dei test La creazione dei test è molto simile a quella vista per Nusmv (vedi pagina 78):

- Creare il progetto CitLab andando su File → New → Other e scegliere "CitLProject" ;

- fare tasto destro sul package in cui si vuole creare il file e selezionare New → File;
- chiamare il file come si vuole e ricordarsi di aggiungere l'estensione ".citl";



- cliccare sul "Finish";

Eseguire i test Cliccare sull'icona  e scegliere il generatore di casi di test adeguato. Se viene chiesto se aprire il file corrente dire di sì.

Verrà aperta una finestra in cui si potranno gestire le impostazioni (ignora i vincoli, semplifica, ecc), per generare i test cliccare su "Generate".

Esempio: *equazioni di secondo grado*

Riprendiamo l'esempio sulle equazioni di pagina 89.

Come prima cosa si dichiara il modello:

```
Model equazioni
```

Come parametri si è deciso di utilizzare il valore della soluzione (cioè zero, uno o due) e i valori dei tre input:

- Gli tre input sono stati dichiarati come numeri. Es: `Numbers a { -10 -5 5 10};`
- La soluzione è stata dichiarata come `Enumerative` (si poteva anche dichiarare come `Numbers`)

```
Parameters:
  Enumerative soluzione {zero una due};
  Numbers a { -10 -5 5 10};
  Numbers b { -10 -5 0 5 10};
  Numbers c { -10 -5 0 5 10};
end
```

Abbiamo due tipi di vincoli:

- Vincoli sugli input: `a` sempre diverso da zero (per ipotesi aggiuntiva non matematica)
- Vincoli sugli output: i valori che assume soluzione a seconda del valore del delta che ha valore `b*b - 4*a*c`
  - è zero se delta < 0
  - è una se delta $== 0$
  - è due se delta > 0

  Purtroppo CitLab non permette di avere variabili definite direttamente senza dominio, non si può scrivere `delta = b*b -4*a*c`
  Come alternativa si può usare l'espressione di delta nei constraints.

```
Constraints:
  # a != 0 #
  # (soluzione == soluzione.zero) => b * b - 4 *a*c < 0 #
  # (soluzione == soluzione.una) => b * b - 4 *a*c == 0 #
  # (soluzione == soluzione.due) => b * b - 4 *a*c > 0 #
end
```

Eseguendo il test si otterrà una griglia fatta in questo modo:

Test	soluzione	a	b	c
1	due	-10	-10	-10
2	zero	-10	-5	10
3	due	-10	5	5
4	zero	-10	10	-5

5.3 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).

Esempio: *testing basato su specifiche*

Abbiamo una funzione `foo` che per ogni `x` mi restituisce il doppio
Sia la seguente funzione:

```
public int foo(int x) { return 2*x; }
```

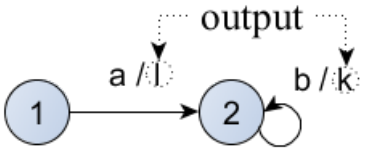
Si potrebbero provare gli input `x`= $\{2; 1; 10; \dots\}$ ma non si può essere sicuri che l'ouput ottenuto è quello atteso: a seconda del tipo di specifica ci sono approcci diversi

- Specifica formale: la specifica è molto probabilmente cartacea, quindi si deve provare formalmente che $\forall$`x foo(x)` = `2*x;`
- Specifica eseguibile: la specifica è a sua volta un programma o un modello, quindi basterà fare un conformance testing.

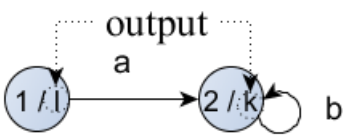
5.3.1 Macchine a stati finiti (FSM)

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.



Macchina di Mealy



Macchina di Moore

Macchina di Mealy 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.

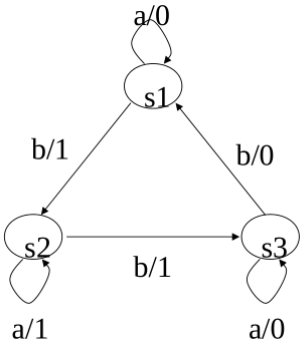
Gli output possono anche rappresentare azioni eseguite dalla macchina (Es: richiamano delle funzioni).

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 e modularità. D'ora in poi FSM = macchina di Mealy.

Esempio: *Macchina di Mealy*

Descriviamo la seguente macchina:



S	L'insieme dei suoi stati è $S = \{ s_1, s_2, s_3 \}$
I	L'insieme degli input possibili è $I = \{ a; b \}$ (sono le scritte a sinistra di / sulla transizione)
O	L'insieme degli output possibili è $O = \{ 0; 1 \}$ (sono le scritte a destra di / sulla transizione)
T	Le transizioni si rappresentano come (*stato iniziale*; *input*; *stato finale*) quindi $s_1 \to s_2$ il cui input sulla transazione è b è rappresentato come $(s_1; b; s_2)$. Analogalmente le altre transizioni sono (s_1, a, s_1), $(s_1; b; s_2)$, (s_2, a, s_2), (s_2, b, s_3), (s_3, a, s_3), (s_3, b, s_1).

G	È simile alle transizioni, con la differenza la tripletta è formata come segue: (*stato iniziale*; *input*; *output*). Quindi gli output sono (s_1, a, 0), (s_1, b, 1), (s_2, a, 1), (s_2, b, 1), (s_3, a, 0), (s_3, b, 0).

Semplificazione della FSM Si semplifica il set $\{S, S_0, I, O, T, G\}$ unendo le due triple T e G per ottenere una quadrupla { *stato iniziale*; *input*; *output*; *stato finale* }.

Esempio: *Semplificazioni della macchina di Mealy*

Riprendendo la macchina a stat di prima di possono unire le triplette di T e G ed ottenere (s_1, a, 0, s_1), (s_1, b, 1, s_2), (s_2, a, 1, s_2), (s_2, b, 1, s_3), (s_3, a, 0, s_3), (s_3, b, 0, s_1).

Esempio: *Progettare una FSM dalle specifiche*

Abbiamo la seguente specifica:

"*Per accedere al parcheggio, il semaforo deve essere verde. Mentre la sbarra è aperta, il semaforo è rosso. Quando l'auto è entrata, la sbarra si chiude ed il semaforo ritorna verde. Per uscire dal parcheggio, l'autista deve inserire il biglietto pre-pagato nel dispositivo che comanda la sbarra. Quando l'auto è uscita la sbarra si chiude*".

Come prima cosa si devono individuare gli elementi che compongono il sistema:

"*Per accedere al parcheggio, il* ***semaforo deve essere verde****. Mentre la* ***sbarra è aperta****, il* ***semaforo è rosso****. Quando l'****auto è entrata****, la* ***sbarra si chiude*** *ed il* ***semaforo ritorna verde****. Per uscire dal parcheggio, l'autista deve* ***inserire il biglietto pre-pagato*** *nel dispositivo che comanda la sbarra. Quando l'****auto è uscita*** *la* ***sbarra si chiude***"

Quindi abbiamo: semaforo verde, semaforo rosso, sbarra alzata, sbarra chiusa, auto entrata, auto uscita, inserire biglietto.

Per capire se un elemento è un input, output o stato bisogna prima di tutto individuare che cosa si vuole modelizzare con il sistema: si sta modellando l'entrata/uscita da un parcheggio, quindi è molto utile descrivere lo stato del sistema come barra alzata/abbassata. Chiameremo gli stati del sistema *closed* (barra abbassata) e *open* (barra alzata).
Gli input rappresentano le condizioni per andare da uno stato all'altro:

- l'entrata e l'uscita della macchina ("*Quando l'auto è entrata, la sbarra si chiude*"/ "*Quando l'auto è uscita la sbarra si chiude*"). Si indicano con m_{in} (macchina entrata nel parcheggio) e m_{out} (macchina uscita);
- l'inserimento del biglietto ("*Per uscire dal parcheggio, l'autista deve inserire il biglietto pre-pagato*"). Si indica con *ins_biglietto*;
- il semaforo verde ("*il semaforo deve essere verde*"). Si indica con *verde*

Gli output rappresentano le azioni fatte dal sistema appena si arriva nel nuovo stato:

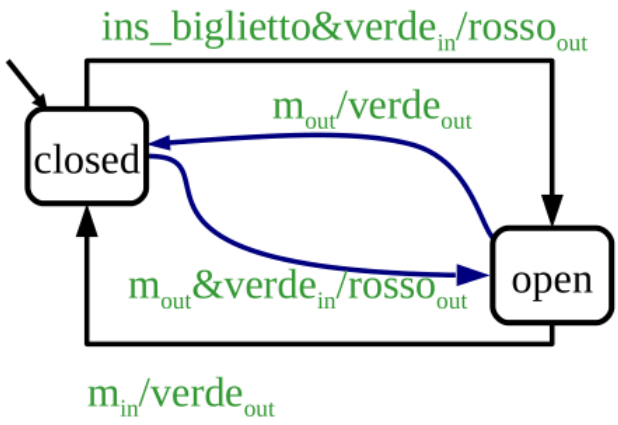
- semaforo rosso ("*Mentre la sbarra è aperta, il semaforo è rosso*"). Si indica con $rosso_{out}$.
- semaforo verde ("*la sbarra si chiude ed il semaforo ritorna verde*").

Per evitare confusione tra il semaforo verde quando si accede al parcheggio e il semaforo verde quando si esce dal parcheggio, si differenziano in $verde_{in}$ e $verde_{out}$.

Per le transizioni basta leggere il testo ed individuare le condizioni (input) per passare da *closed* a *open* (e viceversa) e le azioni che ne derivano quando si passa da uno stato all'altro (output). Si sono riassunte in queste tabella:

Stato iniziale	Input	Output	Stato finale
closed	*ins_biglietto* e $verde_{in}$	$rosso_{out}$	*open*
open	m_{out}	$verde_{out}$	*closed*
closed	m_{out} e $verde_{in}$	$rosso_{out}$	*open*
open	m_{in}	$verde_{out}$	*closed*

Infine, si è schematizzata la tabella creata:



5.3.2 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).

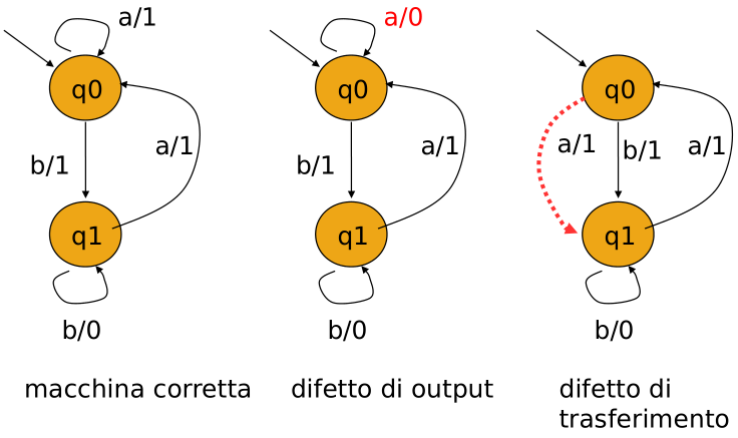
I metodi di test che vedremo sono ideali, cioè provano l'assenza di difetti. Sono validi però solo sotto ipotesi precise e abbastanza forti, anche se nella pratica hanno dimostrato di essere efficaci anche quando le ipotesi non valgono.

Assunzioni:

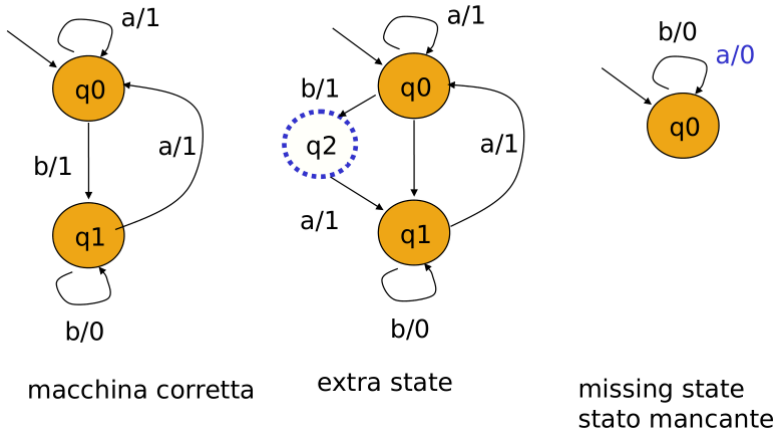
- I e S deterministiche, inizializzate e completamente specificate;
- dev'essere possibile raggiungere ogni stato (S fortemente connessa);
- non possono esistere stati equivalenti (S minimizzata);
- I non varia durante l'esecuzione;
- I ha lo stesso alfabeto di S;
- I non ha più stati di S (assumiamo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

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.



Fault model È un modello ipotetico su quali tipi di difetti possono accadere in una implementazione, senza modello dei difetti possono esserci un numero infinito di implementazioni errate quindi il testing non può garantire nulla. Notare che in un programma i modelli di difetti sono molto più numerosi e difficili da scoprire.
Es: difetti di uso di aritmetica,chiamata a funzioni sbagliate, difetto nella specifica del tipi di dato, difetto di inizializzazione, difetto nel numero di variabili, uso di operatori logici errati, ecc.

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.

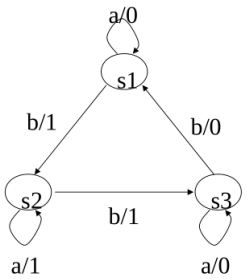
Permette di trasformare un test set in un'unica test sequence.

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).

Esempio: *Test sequence*

Riprendiamo in considerazione la macchina a stati di Mealy vista qualche esercizio prima



Se applichiamo l'input *ababab* si attraverserà la macchina partendo dallo stato s_1 (avendo come input a), poi s_2 (input b), ancora s_2 (input a), ecc fino ad ottenere questo elenco di transizioni

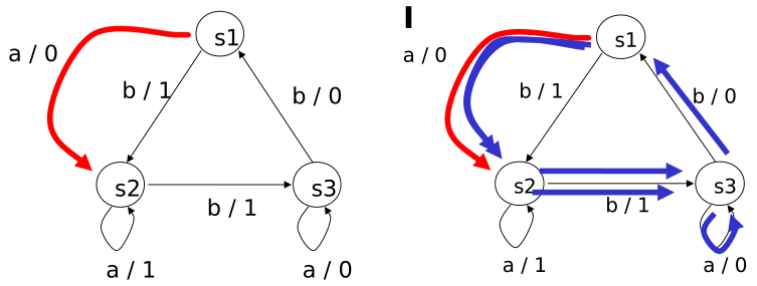
	s_1	$\rightarrow$	s_1	$\rightarrow$	s_2	$\rightarrow$	s_2	$\rightarrow$	s_3	$\rightarrow$	s_3	$\rightarrow$	s_1
input		a		b		a		b		a		b	
output		0		1		1		1		0		0	

Se si raccolgono gli output di ogni transizione, si ottiene 011100.

Nel caso l'implementazione avesse avuto un transfer error come segue, allora si sarebbero otteneute le transizioni

	s_1	$\rightarrow$	s_2	$\rightarrow$	s_3	$\rightarrow$	s_3	$\rightarrow$	s_1	$\rightarrow$	s_2	$\rightarrow$	s_3
input		a		b		a		b		a		b	
output		0		1		0		0		0		1	

e quindi l'ouput sarebbe stato 010001, che è diverso da quello ottenuto con la specifica!



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).

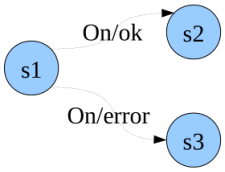
Tour Euleriano: è il TT più corto. Per macchine simmetriche (tanti archi uscenti quanti entranti) è semplice da trovare (tempo lineare). Per macchine non simmetriche è più complesso (tempo polinomiale).

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

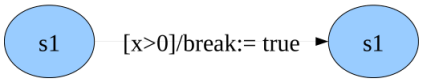
- con status message: applico status dopo ogni messaggio, verificando lo stato; permette di scoprire tutti i difetti (output e trasferimento);
- senza status message: garantisce la scoperta solo dei difetti di output; potrei comunque riuscire a capire in quale stato mi trovo applicando input aggiuntivi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

Estensioni

- Macchina non deterministica: una macchina è detta non deterministica quando da uno stato posso raggiungere due stati diversi senza un vero criterio (quindi non si sa quando passo da uno stato all'altro).
  Si può usare runtime testing (genero mentre eseguo i casi di test) oppure al posto di utilizzare delle sequenze utilizzo degli input fatti ad albero.



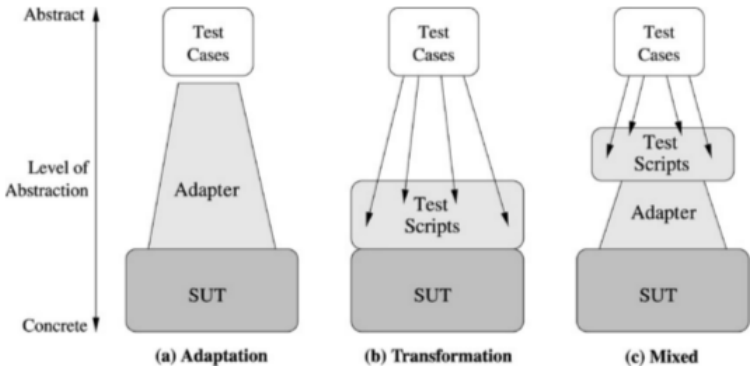
- Se la macchina ha della variabili per gestirle si aggiungono variabili agli stati, guardie e assegnamenti



5.4 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.

5.4.1 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 sempre 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.

5.5 Testing FSM con ModelJUnit

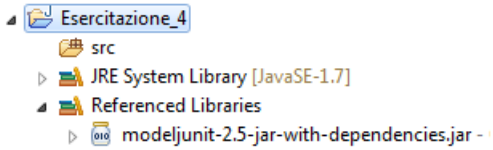
Permette di scrivere un modello FSM direttamente in Java.

Predisposizione del progetto

- Scaricare dal seguente link il jar di ModelJUnit da `http://sourceforge.net/projects/modeljunit`
- Includerlo nel path
  - Fare File → New → Java Project
  - Scrivere il nome del progetto e cliccare su "Next"
  - Cliccare sulla scheda "Libraries"

– Premere il bottone “Add External Jars”, quindi selezionare il jar scaricato

– Cliccare il pulsante “Finish” (il jar di ModelJUnit verrà aggiunto in “Referenced Libraries”)



Se si è già creato il progetto per arrivare alla scheda Libraries si può fare tasto destro sul progetto e selezionare Build Path → Configure Build Path

Scrivere modelli 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, è anche chiamata azione;
- `public boolean action_iGuard()`: viene chiamata per verificare se nello stato attuale può essere eseguita l’azione action_i .

Può ritornare:

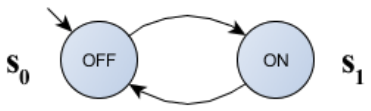
true l’azione può essere eseguita (quindi può essere chiamata), se non implementata l’azione può essere sempre eseguita, quindi ritorna true;

false l’azione non può essere eseguita (quindi non può essere chiamata)

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

Esempio: *creare un modello*

Sia il seguente modello che simula un interruttore:



Come prima cosa si creerà la classe implemtentando `FsmModel`

```
public class onOff implements FsmModel { ··· }
```

Premere `Ctrl + Shift + O` per importare direttamente la classe `FsmModel`, oppure scrivere sopra `public class` il seguente codice:

```
import nz.ac.waikato.modeljunit.FsmModel;
```

Si crea la variabile che descrive gli stati del sistema, nel nostro caso si chiamano `ON` e `OFF`:

```
enum State{ON, OFF};
```

Si dichiara la variabile che indica l'astrazione dello stato (da notare che il suo tipo è l'enumerativo `State` creato poco prima):

```
State state;
```

Nel costruttore si dichiara il valore delo stato iniziale:

```
public onOff(){ state = State.OFF; }
```

Si implementando i metodi `getState` (che ritorna la variabile `state`) e `reset` (che si occupa di "azzerare" il sistema, cioè lo fa ritornare allo stato iniziale)

```
@Override public Object getState() { return state; }
@Override public void reset(boolean arg0) { state = state.OFF; }
```

Si implementano le transizioni da uno stato all'altro attraverso la funzione `change()` , come sopra, se non riconosce `@Action` premere `Ctrl + Shift + O` per importare la classe (nel caso venga chiesto che classe importare scegliere `nz.ac.waikato.modeljunit.FsmModel`)

```
@Action public void change(){
  System.out.println("cambio stato");
  if( state == State.OFF ) state = State.ON;
  else state = State.OFF;
}
```

Si implementano anche due azioni di settaggio dello stato (cioè si impone di essere in un certo stato):

```
@Action public void on(){ state = State.ON; }
@Action public void off(){ state = State.OFF; }
```

Da notare che durante il cambio di stato si poteva anche richiamare `this.on()` al posto di `state = State.ON` (vale lo stesso per `off()`).

Esempio: *guardie*

Si possono utilizzare le guarda al posto degli if, quindi nella macchina dell'esercizio precedente al posto di utilizzare `change()` per controllare quando accendere o spegnere, si utilizzano due guardie:

- `onGuard()` dice quando è possibile chiamare l'azione `on()`, quindi si può chiamare solo quando lo stato è `OFF`
- `offGuarda()` dice quando è possibile chiamare l'azione `off()`, quindi si può chiamare solo quando lo stato è `ON`

Il codice sarà il seguente:

```
public class onOff implements FsmModel {
  ...
  @Action public void on(){ state = State.ON; }
  public boolean onGuarda(){ return state = State.OFF; }
  @Action public void off(){ state = State.OFF; }
  public boolean offGuarda(){ return state = State.ON; }
  ...
}
```

Offline testing Scrivere un caso di test Junit che testa la classe:

- Creare l'algoritmo di generazione dei test passando come variabile al costruttore la macchiana a stati creata da noi

  Es: `Tester tester = new GreedyTester( new FSM() );`

  Ci sono due tipi di tester:

  `GreedyTester` testa il sistema scegliendo i passi in maniera "greedy" (l'algoritmo di scelta fa ottimizzazioni locali);

  `RandomTester` testa il sistema scegliendo a caso i passi da fare.

- `buildGraph` costruisce il grafo della parte osservata del modello

  Es: `tester.buildGraph();`

- Crea il criterio di copertura utilizzando `CoverageMetric`, essendo è un interfaccia è implementata da

  `TransitionCoverage` misura il numero di transizioni testate

  `ActionCoverage` misura il numero di azioni testate

  `StateCoverage` conta quante volte uno stato è stato attraversato

  Es: `CoverageMetric trCoverage = new TransitionCoverage();`

- Si collega il criterio di copertura all'algoritmo di generazione dei test:

  Es: `tester.addCoverageMetric(trCoverage);`

- Genere una test suite composta da 20 passi

  Es: `tester.generate(20);`

- Per stampare informazioni aggiuntive si utilizza

  Es: `tester.getModel().printMessage(...);`

Esempio: *informazioni aggiuntive*

Per vedere il nome della copertura scelta si utilizza `getName()`, mentre per vedere quanto è stato coperto si utilizza `toString()`

```
tester.getModel().printMessage(
  trCoverage.getName() + " was " + trCoverage.toString()
);
```

Verrà stampato `transition coverage was 5/6`.

- Se si crea un `Listener` prima di generare i test è possibile seguire i passi nel modello.

Esempio: *aggiunta del Listener*

La classe `VerboseListener` stampa qualsiasi messaggio inviato dal modello, quindi se si aggiunge il listener poco prima di far partire i test "registerà" ogni cambio di stato:

```
tester.addListener(new VerboseListener());
tester.generate(5);
```

Quando viene fatta una transizione è chiamato il metodo `doneTransition` del `VerboseListener` che stamperà il messaggio "`done`" e la stringa ritornata dal `toString()` della classe `Transition`, ovvero una tripletta formata da (*stato iniziale*; *azione*; *stato finale*)
Riprendendo l'esempio dell'interruttore, una transizione verrà rappresentata come

```
done (ON, on, ON)
```

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()`**).

Esempio: *online testing*

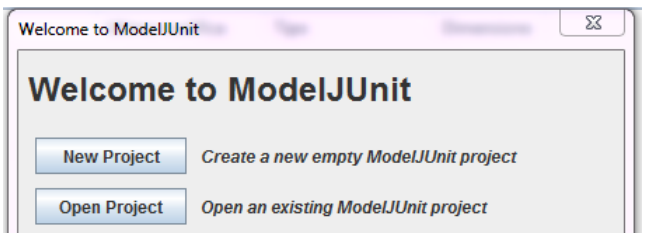
Riprendendo l'esercizio dell'interruttore, il testo online è (per motivi di spazio non sono stati aggiunte le importazioni):

```
public class test {
  public static void main(String[] args) {
    Tester tester = new GreedyTester(new onOff());
    tester.buildGraph();
    CoverageMetric trCoverage = new TransitionCoverage();
    tester.addCoverageMetric(trCoverage);
    tester.addListener(new VerboseListener());
    tester.generate(20);
    tester.getModel().printMessage(
      trCoverage.getName() + " was " + trCoverage.toString()
    );
  }
}
```

Online 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;
  - Andare su File → Export
  - Scrivere "jar" nella casella di testo e selezionare "Jar File", infine cliccare su "Next"
  - Nella finestra successiva selezionare il progetto/package che si vuole esportare e cliccare su "Finish"
- copiare il file modeljunit-2.5-jar-with-dependencies.jar all'interno della cartella del progetto;
- fare doppio click sul jar (quello appena spostato) oppure eseguire il comando `java -jar modeljunit-2.5-jar-with-dependencies.jar`, apparirà la seguente schermata:



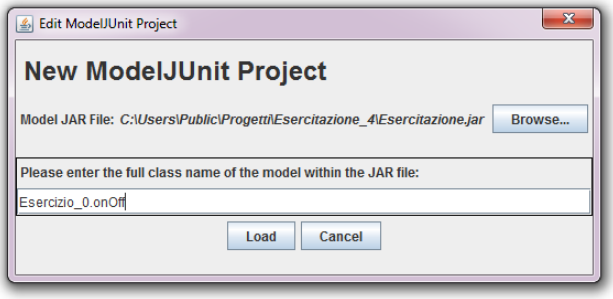
- creare un nuovo progetto cliccando su "New Project"
  - cliccare su "Browse" specificando come jar quello precedentemente esportato;
  - come nome della classe inserire il nome della classe da testare (comprsi i package di cui fa parte)

- se al passo precedente dà errore (probabilmente perché aprendo il jar prende il percorso assoluto, ma poi ragiona come se fosse relativo):
  - salvare il progetto (File → Save Project) 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)

    ```
    <className>PACKAGE.MIACLASSE</className>
    <packageLocation>file:NOMEJAR.jar</packageLocation>
    ```
  - aprire **modeljunit-2.5-jar-with-dependencies.jar** e aprire il file *.mju modificato.

Esempio: *apertura in ModeJUnit*

Se si volesse aprire onOff, allora si compilerebbero i campi in questo modo:

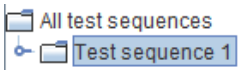


Da notare che sono stati anche inseriti i nomi dei package di cui fa parte, se no ModelJUnit non avrebbe potuto accedere alla classe in modo corretto.

Se si apre il file .mju con il blocco note si noterà il seguente

```
</project>
  ...
<className>Esercizio_0.onOff</className>
<packageLocation>
  file:/C:\Users\Public\Progetti\Esercitazione_4\Esercitazione.jar
</packageLocation>
</project>
```

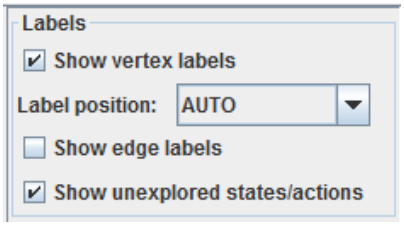
La prima volta che il modello è caricato andando su All test sequence → Test sequence 1 si possono già vedere i vari modelli di transizioni



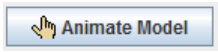
In particolare se le transizioni a linea continua (→) sono quelle che il modello copre, mentre quelle tratteggiate (⇢) sono quelle non presenti nel modello.

La possibilità di osservare le transazioni/stati è utile perché in questo modo si può scoprire se ci si è dimenticati di collegare degli stati tra loro.

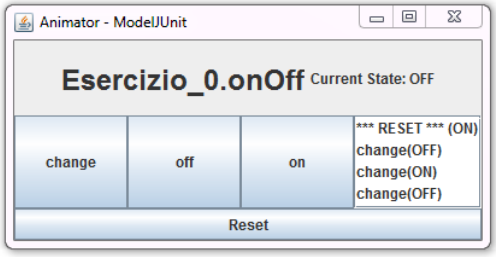
Per vedere solo le transizioni/stati coperti, andare nel menù a destra e togliere la spunta a "Show unexplored state/action"



Analizziamo le voci del menù:



viene aperta una finestra in cui si può navigare il modello semplicemente cliccando sulle sue azioni, è utile per controllare se ci sono errori nelle transizioni:



si gestiscono i parametri di esecuzione dei test, si può scegliere:

Test generation Algortihm: tipologia di generazione dei test (random, greedy, ecc)

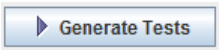
Total test lenght: il numero dei passi che farà il test

Avergage Test lenght:

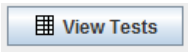
[nome algoritmo] Parameters: parametri specifici di ogni algoritmo di generazione di test

Reporting: che cosa mostrare (mostra test generati, mostra spiegazione dei fallimenti, scelta dei colori delle coperture, ecc)

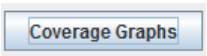
Nota: ingrando la finestra si può anche vedere lo script che verrà eseguito (notare che utilizza le stesse classi che usiano anche noi nell'offline testing)



Genera i test secondo le impostazioni fatti



Mostra i test eseguiti



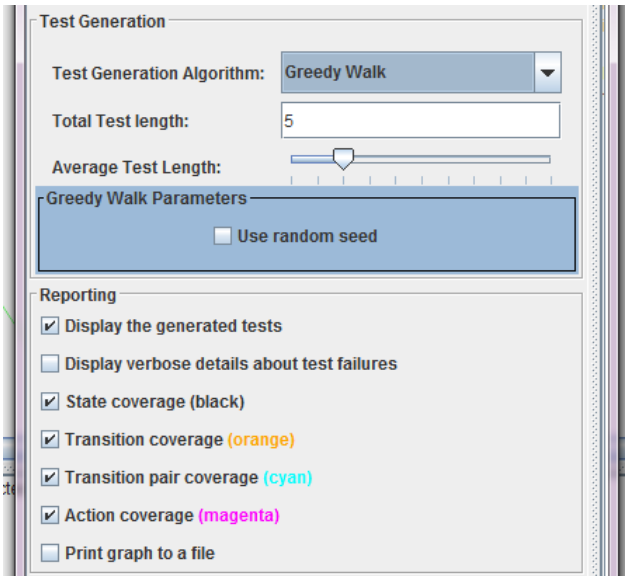
Mostra il grafico di copertura di stati (in nero), transizioni (in arancione), coppie di transizioni (azzurro) e azioni (magenta).

Esempio: *offline testing*

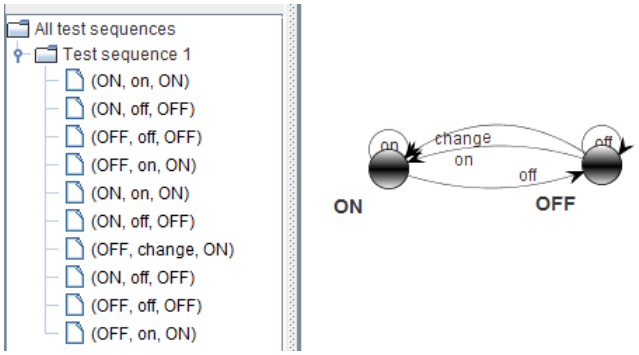
Si sono fatte le seguenti azioni:

(1) Modifica impostazioni: si è cliccato su "Test Configuration" e si sono modificate le impostazioni dei test

(2) Generazione test: con "Generate Tests" si sono generati ad eseguiti i test. Nell'immagine (2) sono mostrate le transazioni fatte



(1)



(2)