

1. Introduction: testing and verification

1.1 Introduzione

testing

- analisi dinamica → **runno** il programma

verifica

- analisi statica → **senza runnare** il programma

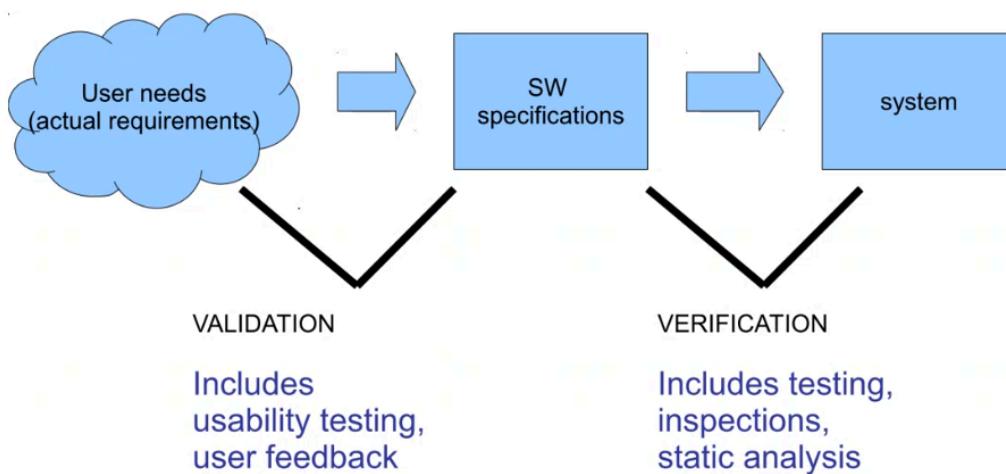
	TESTING	VERIFICATION
PROGRAMS	2. PROGRAM-BASED TESTING	3. PROGRAM-BASED VERIFICATION
MODELS	4. Model-based testing	4. MODELING Model checking

Validazione

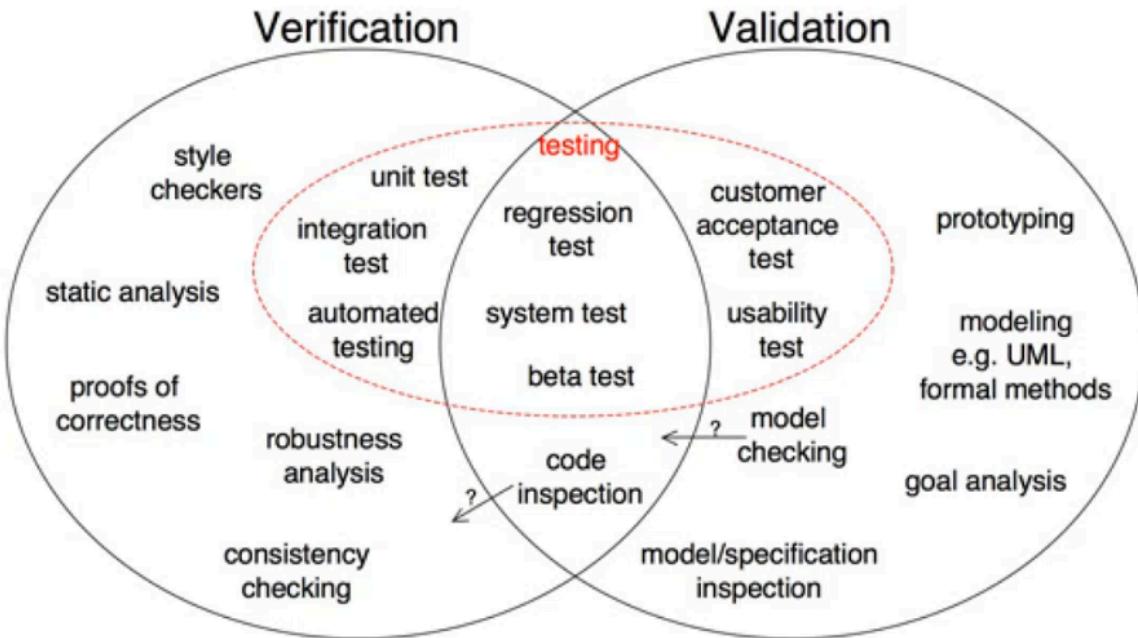
- sto facendo le cose che mi **servono?**
- più difficile perché richiede interazione con l'utente
- user acceptance testing
- richiede **intervento umano**

Verifica (progettazione)

- sto facendo le cose nel modo **corretto** in relazione ai requisiti di una specifica?
- verifica di alcune proprietà da soddisfare
- può essere **automatizzata**



testing: utilizzato sia nella verifica che nella validazione



Problema indecidibile

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

Testing esaustivo

- controlla tutto, **infattibile**

Testing approssimato

- pessimistico = non garantisce di accettare un programma anche se va bene
- ottimistico = potrebbe accettare dei programmi anche se potrebbero portare degli errori

Proprietà semplificate = proprietà estese, ma diminuite di **gradi di libertà**

- **Inaccuratezza ottimistica** : potremmo non accettare un programma anche se ha la proprietà che vogliamo
- **Inaccuratezza pessimistica** : potremmo accettare alcuni programmi che non hanno la proprietà che vogliamo

Safe = **Sicuro**

- non ha inaccuratezza ottimistica
- accetta solo programmi corretti

Sound = **Corretto**

- se l'analisi passa allora il programma è corretto (è tipo un'implicazione forte, cioè se l'analisi passa allora il programma non può essere sbagliato)
- un programma potrebbe essere corretto ma anche non passare (conservativo)
- il testing non è sound

Complete = **Completo**

- passa l'analisi ogni programma corretto

1.2 Testing Process

Testing

- valutare la qualità di un prodotto per migliorarlo individuando i difetti
- set finito di casi di test rispetto al comportamento atteso
 - attingere da un dominio di test potenzialmente infinito
- valutazione del prodotto in esecuzione con un set finito di input basato su un criterio dal quale estraiamo una misura di correttezza
- tipologie
 - *Dynamic* = richiede l'esecuzione del sw
 - *Finite* = set finito di input
 - *Selected* = criterio selezionato
 - *Expected* = comportamento auspicabile dal quale estrarre la misura di correttezza

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

White-box = si controlla il codice

Passi principali

- Designing = **cosa testare**, in che contesto e con che criterio
- Executing = **come effettuare il test**, come analizzare i risultati e come catalogarli come fallimenti del test
- Verifying = **verifica della potenza del test** (traceability matrix tra test cases e requirements)

Tipi di testing:

- Manual testing
 - tutto a mano
 - conveniente se poca roba
- Capture and replay
 - registra i casi d'uso per avere dei test script
 - **re-run** dei casi di test vecchi su nuove versioni del sw
 - comodo ma poco robusto se i cambiamenti sono tanti
- Script-based
 - capture and replay tramite script, senza eseguirlo fisicamente la prima volta
 - lanciati in **automatico**, ma vanno "aggiornati" insieme al sw
 - costo di manutenzione (codice + script)
 - **TTCN molto usato nelle telecomunicazioni**
- Program-based
 - **stesso linguaggio del SUT** (System Under Testing)
 - i test sono dei **programmi**
 - **esecuzione del codice e verdetto inclusi nel test**
 - comodi da rilanciare, ma sono più onerosi da scrivere e manutenere

Testing Process	Solved Problems	Remaining Problems
Manual Testing	Functional testing	Imprecise coverage of SUT functionality No capabilities for regression testing Very costly process (every test execution is done manually) No effective measurement of test coverage
Capture/ Replay	Makes it possible to automatically reexecute captured test cases	Imprecise coverage of SUT functionality Weak capabilities for regression testing (very sensitive to GUI changes) Costly process (each change implies recapturing test cases manually)
Script- Based Testing	Makes it possible to automatically execute and reexecute test scripts	Imprecise coverage of SUT functionality Complex scripts are difficult to write and maintain Requirements traceability is developed manually (costly process)
Program-based Testing	No extra language is required	It may require additional effort during maintenance

- Model-based
 - **modella sia il sw sia l'ambiente di test** generando test astratti
 - modifica il modello modifica in automatico anche i casi di test che ne derivano
 - poi li rende eseguibili come test che vengono valutati dopo l'esecuzione

Formal Verification

- tecniche che creano una prova matematica di consistenza tra una rappresentazione (design) e una specifica

Program Verification

- un programma è corretto rispettivamente ad una specifica, assumendo che sia corretta e completa
- cerca di dimostrare la correttezza del codice in modo formale
- diverso dal testing esaustivo, perché quello non prova la correttezza
- **correttezza relativa alla specifica \neq correttezza assoluta**

Runtime Verification

- controlla a runtime che un programma faccia quello che deve
- istruzioni extra che controllano come si comporta il codice, in fase di produzione vengono poi eliminate

Formal Model Verification

- formal verification applicata a un modello effettivamente tangibile e verificabile (tipo FSM = Finite State Machine)

Model Checking

- controllo esaustivo ed automatico del modello rispetto ad una specifica

2. Program-based testing

2.1 Test case and selection adequacy

Code testing

Test

- collaudo del sw mediante prova
- esecuzione specifica di casi di test

Testing

- discontinuo, quindi la posizione in cui fare i test è molto critica
 - qualcosa funziona nella maggior parte dei casi, ma non sempre
 - occorre definire con chiarezza i requisiti
- efficace per trovare bug, ma non può provarne l'assenza
- automatizzato, in ogni fase di sviluppo, esteso su tutto il sistema, pianificato
- segue degli standard

Testing VS analisi statica:

- Testing
 - trova tanti errori banali
 - fa fatica a trovare **pochi errori critici**
- Analisi statica
 - più costosa
 - mira a errori poco probabili e **molto critici**

Failure = Guasto = Malfunzionamento

- **esecuzione errata**, parte dinamica
- riguarda il comportamento → problema **dinamico**

Fault = Bug = Difetto

- **elemento** del programma non conforme che genera il malfunzionamento
- problema **statico**
- potrebbe esistere anche senza osservare un malfunzionamento
più bug si compensano, non raggiungo sempre quella porzione di codice, ..

Errore

- fattore che **causa** il difetto (solitamente errore umano)
- delta fra sw prodotto e programma ideale da specifica

→ errore del programmatore → bug nel programma → malfunzionamento

scopi del testing

- cerca di evidenziare i bug tramite malfunzionamenti
- valutare un sw (reliability) con una certa confidenza (test di accettazione)

Tipi di test

- accettazione = comportamento e requisiti dell'utente sono allineati (es. magari è leggermente diverso dalle specifiche ma è più usabile)
- conformità = comportamento e requisiti delle specifiche
- sistema = controlla sia sw sia hw come monolitico
- integrazione = controllo sulla cooperazione delle unità
- unità = test della singola unità
- regressione = test delle release successive

Unit Testing

- In java è a livello di classe → si testano i vari metodi
- metodi = Test Unit, si introducono:
 - Test Driver = metodo che **chiama il Test Unit** con i parametri
 - Test Stub = (opzionale) metodo che **impersonifica eventuali metodi interni** per isolare il caso di test dal resto del sistema
test che includono accesso a DB

test di integrazione

- integrazione fra le diverse unità
func1 chiama func2 con una parametro intero
- guardano i domini dei dati, la loro rappresentazione e la compatibilità
- Si può sviluppare:
 - top-down = parto dal main, richiedono stubs
 - bottom-up, richiedono drivers (= insieme di due metodi chiamati da un metodo di più alto livello)
 - big-bang = testo tutto insieme

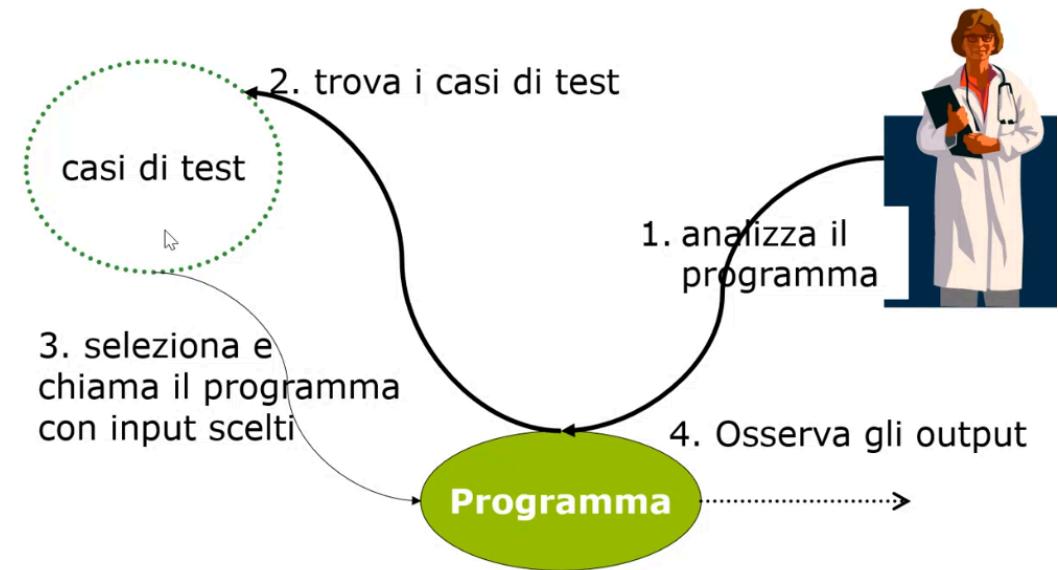
test di (non)regressione

- testing fatto durante la manutenzione
- mirano ad accertarsi di aver fixato i bug e di non averne creati di nuovi controllando la compatibilità con i vecchi casi di test per poterli riusare

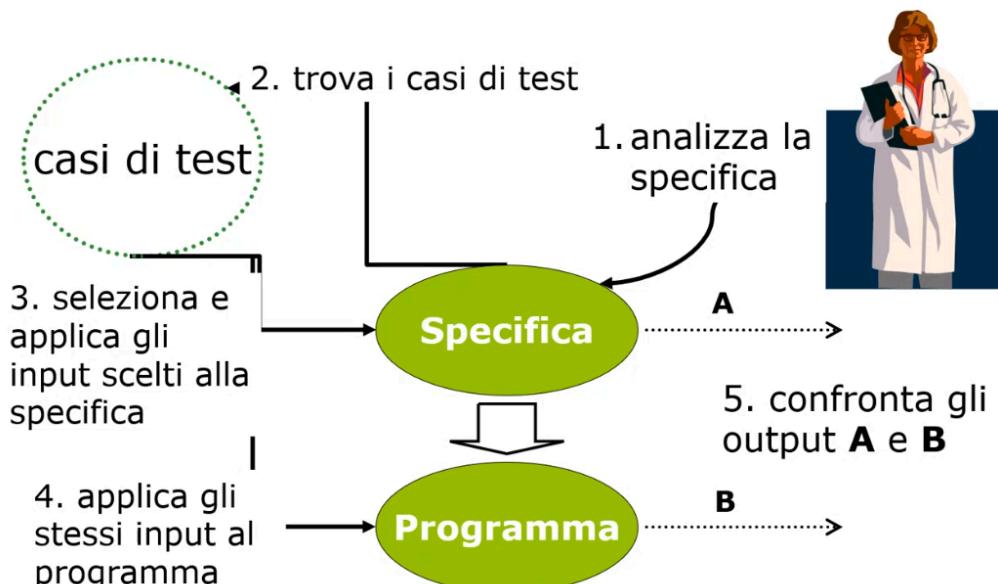
Metodologie di testing (in base al tipo di accesso all'unità testata)

- White-box
 - assume che il codice del programma sia disponibile
 - usa il codice, controlla la copertura, ma non garantisce che il sw effettivamente faccia ciò che è stato richiesto
 - Program-based testing
 - Esamina tutto il sw per trovare punti critici
 - crea casi di test che soddisfano le coperture
 - chiama il programma con input scelti e controlla che l'output sia corretto senza il verificarsi di errori
 - Usa il codice, ma non trova "errori di omissione", cioè parti che non sono proprio state implementate, perché non guarda la

specifica



- limitazioni
 - Non fornisce un oracolo → come faccio a sapere se l'output ottenuto è quello atteso? Serve quindi scriverli a mano
 - non gestisce eventuali omissioni (casi non gestiti dal codice)
- Black-box
 - non guarda il codice, considera solo i requisiti
 - usa i requisiti per controllare il sw creando dei casi di test e confrontandoli con l'elaborazione prevista da quella effettiva
 - controlla e osserva solo input e output del programma
 - Specification-based testing
 - esamina la specifica
 - seleziona dei casi per i test
 - li applica sia alla specifica sia al sw e poi ne confronta gli output
 - Funziona solamente da oracolo, e ha problemi se non ci sono le specifiche o non sono formali



- grey-box testing
 - conosco parte del codice

- conosco la specifica

Test Oracle

- modo per stabilire se il test ha evidenziato un malfunzionamento oppure no

Concetti base

come selezionare i casi di test

Programma $P = \text{funzione da } D \text{ a } R$, cioè, $P: D \rightarrow R$
e può essere non definito per qualche d in D

Predicato $\text{OK}(P, d) = \text{se } P \text{ è corretto per l'input specifico } d$, cioè se produce il risultato atteso $P(d)$.

$\text{OK}(P) = \text{se } P \text{ è corretto, se per ogni } d \text{ in } D \text{ ho che } \text{OK}(P, d)$

→ so che un programma è OK per qualche input, vorrei dimostrare che lo sia per ogni input

Failure (malfunzionamento)

- se P eseguito con $d \neq P(d)$, cioè o ottengo un valore $P'(d)$, oppure il programma si blocca

Fault (difetto o bug)

- P è implementato in P' , $P' \neq P$

Errore = motivo del difetto

Caso di test (test case) = elemento in $D = t = \text{punto nel dominio}$

Test set $T = \text{test suite} = \text{insieme finito di } D$, chiamato T , con dentro vari $t = \text{insieme di punti nel dominio}$

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

Se non ho scoperto errori il test set è negativo, se not OK il test è positivo, cioè ho scoperto errori.

SE UN TEST HA SUCCESSO HA TROVATO UN BUG!

Test set Ideale

- insieme finito di casi di test che basta per essere sicuri di non avere difetti
- l'esecuzione senza errori sul test set ideale implica la correttezza di P , cioè $\text{OK}(P, T') \rightarrow \text{OK}(P)$

Test set esaustivo

- tipo di test ideale che prova **tutte le combinazioni possibili**, ma se D è infinito non va bene

Specifica S

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

Test Criteria (adeguatezza) [Cps(T)] = funzione che è vera solo se T è adeguato a trovare ogni difetto in P, rispetto a S, secondo il criterio C

- Program-based Test Criteria
 - C non dipende da S
 - C: P x T
- Specification-based Test Criteria
 - C non dipende da P
 - C: S x T
- criterio **affidabile**
 - un criterio è affidabile per una coppia di test (T1,T2) se, quando T1 individua un malfunzionamento, allora anche T2 lo rileva
- criterio **valido**
 - se esiste un test set T, che soddisfa C, che è in grado di individuare il difetto

Teorema di Goodenough e Gerhart

- Dato Criterio C, Programma P e Test Suite T, con C ideale, cioè affidabile e valido per P, allora se seleziono T con C e non trovo malfunzionamenti in P, allora T è ideale
 - Se P passa un test set T ideale, allora P è corretto
 - riesco a definire un criterio ideale se però so già la natura del difetto
- metodo che restituisce il primo char di una stringa ricevuta in input
C = almeno una stringa con almeno due caratteri in cui il secondo è diverso dal primo
→ valido + affidabile = ideale

Teorema di Howden

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

Teorema di Dijkstra

- Il test di un programma può rilevare la presenza di malfunzionamenti ma non dimostrarne l'assenza
- cioè **IL TESTING NON PUÒ DIMOSTRARE CORRETTEZZA DEL SW**

Adeguatezza del test

- rilassamento di "idealità" dei test
- praticamente impossibili da ottenere

- Test case specification = requisito minimo di uno o più casi di test
- Test obligation = una proprietà obbligatoria da testare durante i casi di test
- Test di adeguatezza = predicato che è vero o falso se un programma passa o meno una test suite
- Criterio di adeguatezza = gruppo di Test obligation
- Un test soddisfa un criterio di adeguatezza se passa tutti i test e tutti i test obligation sono verificati da almeno un test case.

Satisfiability e Unsatisfiability

- a volte non esistono test suite che possono soddisfare un determinato criterio di un programma
- possiamo misurare la coverage dei test

Subsumes relation

- Test adequacy criterion A subsumes Test adequacy criterion B \leftrightarrow for every program P, every test suite satisfying A also satisfies B

2.2 Structural testing

Test definiti considerando la copertura del codice

- parte del programma che viene eseguita dai casi di test
- struttura del codice relativa al flusso di controllo del programma

Flusso di controllo

- grafo che accopra tutti i cammini prendendo in considerazione tutte le possibilità (non albero altrimenti non potrei tornare indietro)
- Viene considerato nel testing strutturale
- Ad ogni esecuzione corrisponde un cammino

Criterio di test basato sulla copertura dei programmi : $P \times T \rightarrow \{\text{True}, \text{False}\}$

Coperture (coverage)

- Statement coverage (SC) = copertura istruzioni
 - Un test set T è adeguato se per ogni istruzione S **esiste almeno un caso di test t in T che esegue S**
 - criterio debole, ma possiamo usarlo come misura:
statement eseguiti / totali
- Branch coverage (BC) = copertura archi
 - Un test set T soddisfa il criterio di Branch Coverage se **ogni arco è percorso almeno una volta**
 - **implica lo SC** \rightarrow SC è più debole
eseguire un if positivo una volta sola lo rende coperto (SC), ma si coprirà solamente il suo arco positivo(BC), non quello negativo(BC)
 - può essere misurato: branch eseguiti / branch totali

- Generazione test
 - procedo all'indietro dal basso verso l'alto, dopo aver generato il grafico di flusso cercando di eliminare le coperture multiple
 - cerco i valori di input che passano per i percorsi individuati
 - A volte possono esserci situazioni non copribili codice logicamente irraggiungibile aka un errore nel codice
- Decision coverage (DC)
 - espressione booleana = predicato, a guarda di una istruzione condizionale o di una iterativa
 - Un test set T è adeguato se per ogni decisione di P, esiste un caso t in cui la decisione è presa, e un caso t in cui la decisione non è presa
 - simile al BC
- Condition coverage (CC)
 - espressione booleana atomica, non divisibile
 - Un test set T è adeguato se per ogni condizione di P, esiste un caso t in cui la condizione è vera, e un caso t in cui è falsa
 - CC non implica DC
esempio, con A or B, se copro A,B true e A,B false ho coperto tutte le decisioni ma non tutte le condizioni, per esempio A true e B false non l'avevo coperta
 - DC non implica, infatti CC richiede che ogni condizione sia vera o falsa, DC che ogni decisione sia vera o falsa
 - Ci sono combinazioni che vanno bene per uno ma non per l'altro in entrambi i sensi
 - Alcuni compilatori usano la Short Circuit Evaluation in cui semplificano le espressioni
tipo se (A && B) e ho A falso skipo B, oppure se ho (A || B) se ho A vero skipo B
A volte possono essere evitati semplicemente mettendo & al posto di && o | al posto di ||
- Multiple Condition Coverage (MCC)
 - Un test soddisfa MCC se testa ogni combinazione dei valori di verità delle condizioni
 - Con n condizioni si hanno 2^n combinazioni
 - Di solito si costruisce una tabella
- Modified Condition/Decision Coverage (MCDC)
 - Criterio più forte del CC, ma lineare in n e non è 2^n come MCC
 - Un test soddisfa MCDC se ogni condizione all'interno di una decisione deve far variare in modo indipendente il valore finale della decisione
 - cioè ogni decisione è formata da più condizioni, e ogni caso di test deve far sì che la decisione finale cambi a seguito della

variazione di una sola condizione, lasciando tutte le altre fisse

```
int check(int x, int b){  
    if (x > b & x % 2 == 0)  
        return 100;  
    return 200;  
}
```

	x>b	x%2==0	DEC
1	T	T	T
2	F	T	F
3	T	T	T
4	T	F	F

1. $x=2$ $b=1$ \rightarrow ~~val 100 ✓~~
2. $x=2$ $b=3$ \rightarrow ~~val 200~~
3. $x=3$ $b=2$ \rightarrow ~~val 200~~

- Prendo tutte le possibili combinazioni come se riempissi la tabella dei numeri in binario (1 = true, 0 = false)
- per ogni variabile considero due righe in cui tale valore è l'unico a cambiare e determina output finale diverso
 - il caso base 00.. è valido come caso in cui la variabile vale 0 \rightarrow diminuiscono i numeri di casi da considerare

TE

```

1 boolean check (int[] a, int b)
2   if (a != null && a.length > 0):
3     for (int : a):
4       if (x > b && x % 2 == 0):
5         return true;
6   return false

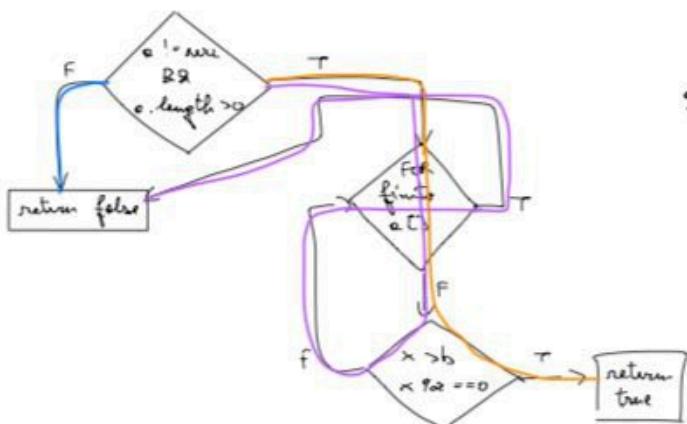
```

• SC

$$a = [2], b = 0 \quad \text{crossover } 2, 3, 4, 5$$

$$a = \text{null}, b = 0 \quad \text{crossover } 2, 6$$

• BC



$$\text{SC} \left\{ \begin{array}{l} a = [2], b = 0 \\ a = \text{null}, b = 0 \end{array} \right.$$

sc way better per BC

$$\Rightarrow a = [2], b = 5$$

• MCDC

$$\text{if } (x > b \& \& x \% 2 == 0)$$

$$a \neq \text{null} \& \& a.length > 0$$

$x > b$	$x \% 2 == 0$	x	b	a
T	T	2	0	[2]
F	F	2	4	[2]
T	F	3	0	[2]

$a \neq \text{null}$	$a.length > 0$	a
T	T	$a = [2] \quad \checkmark$
F	T	$a = \text{null} \quad \checkmark$
T	F	$a = []$

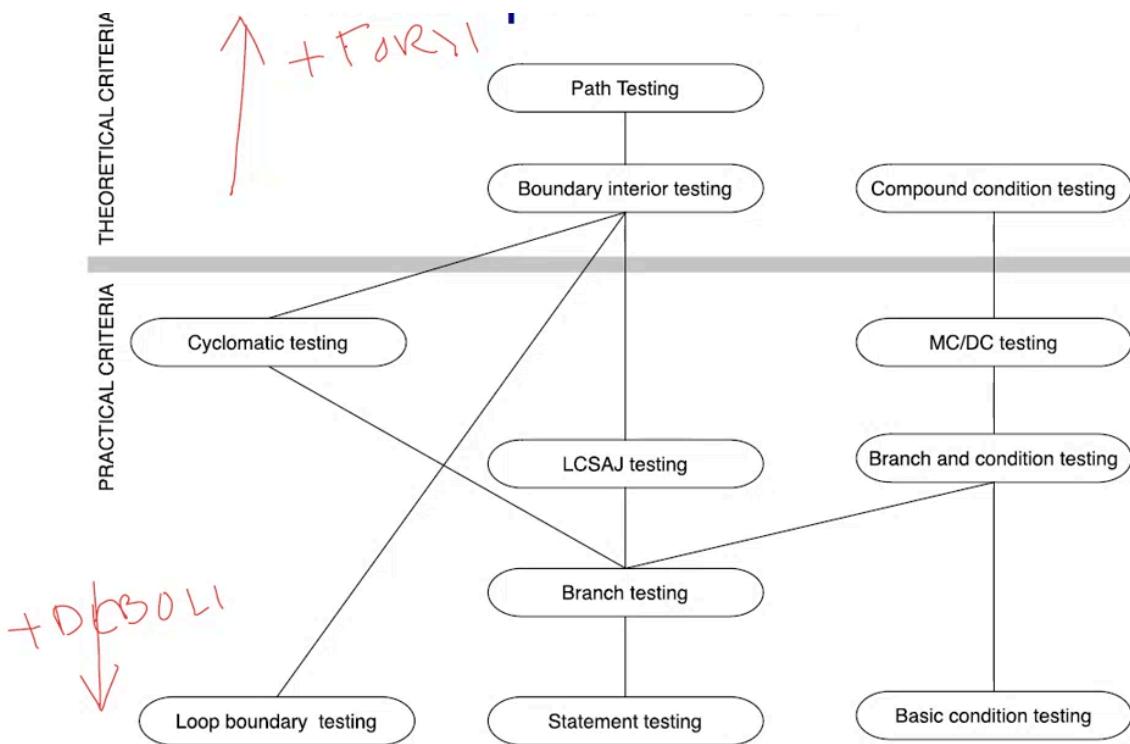
Path = sequenza di branch

- Path testing = criterio teorico che testa tutti i possibili path
- path adequacy = #(path eseguiti) / #(path totali)
- Il numero di path potrebbe essere illimitato se ci sono cicli, quindi si limitano i loop o le lunghezze dei path

- Si raggruppano path simili, che si dividono internamente, con la tecnica del Boundary Interior Path Testing
- tanto mi interessa percorrerli tutti e unicamente, quelli multipli possono essere ottimizzati
- Potrebbero generarsi comunque problemi di cicli infiniti, perciò è stato introdotto il criterio di loop boundary adequacy \leftrightarrow per ogni loop:
 - in almeno un t, il loop è iterato 0 volte
 - in almeno un t, il loop è iterato esattamente 1 volta
 - in almeno un t, il loop è iterato almeno 1+ volte

procedure call testing = method coverage

- procedure entry and exit testing
 - per ogni procedura testare ogni suo punto di entrata e uscita
- call coverage
 - testare ogni punto da cui si può invocare un metodo (invoco almeno una volta ogni chiamata di procedura)
 - SC \rightarrow call coverage



Unsatisfiability of criteria

- nel sw è comune trovare cose non raggiungibili, tipo statement illogici, condizioni mai matchate, percorsi non percorribili ..
- è sensato abbassare il goal di coverage < 100% però giustificando gli elementi non coperti (fossil code)

2.3 Test Execution

Scaffolding

- parte di programma scritta per eseguire i test
- Ponteggi = codice di supporto allo sviluppo, non visibile o addirittura non mantenuto alla fine
- Include test drivers (chiamano i test unit), stubs (simula i metodi del test unit), harnesses (imbragature, sostituti per altri componenti del sistema)
- testability = Scrivere il codice in funzione dei test
non implementando tutto nella gui ma scrivendo API e WRAPPER (Controller e Observer)
- controllability e osservability
 - evito di mettere avere funzionalità dentro l'interfaccia grafica, o in generale in luoghi dove non posso testarli
 - portare il più possibile le funzionalità dalla gui alle api per poterle testare

automating test execution

- ideare i casi di test è ancora fatto da noi
- l'esecuzione dei casi di test però dovrebbe essere automatica

Oracoli

- ciò che ci dice se un test è fallito/passato
- Tipi di oracoli:
 - Comparison-based:
 - **confronta l'output** predetto con quello effettivo
 - Self-Checking code:
 - senza dover prevedere l'output, ma **avviene tutto internamente** con dei metodi appositi
 - Human:
 - a volte tocca metterci mano, però almeno possiamo ottimizzare la ripetizione dei test con **Capture and Replay**

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

Mock object

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

CI = Continuous Integration

- il codice viene spesso sottoposto a controlli e a build di test, così si rilevano velocemente i problemi (tipo più volte al giorno)

Unit Testing (JUnit e Codecover)

Codecover

- copertura decisioni e condizioni
- permette di valutare MCDC

JUnit

- framework per automatizzare i test di unità di programmi java
- ogni test contiene le asserzioni che controllano che il programma non contenga difetti
- si ispira all'eXtreme Programming
- definire ed eseguire i test
- formalizza in codice i requisiti sulle unità
- integrato in molti IDE

Test-Driven Development Cycle

- 1) scrivere i casi di test prima del codice partendo dalla specifica
- 2) esegui i test che falliranno sicuro
- 3) scrivi il codice fino a che non riesci a far passare tutti i test
- 4) riparti da 1)
 - all'inizio si va più lenti ma poi si hanno più vantaggi che svantaggi
 - doppio codice (app + test)
 - bisogna automatizzare l'esecuzione dei casi di test scritti a mano

testare una classe

- per testare una classe (Es. Videos.java) creo una classe con la X davanti al nome (Es. X...)
 - click dx sulla classe da testare → New → junit test case → seleziona new junit 4 test
 - genera file in src, trascino nella source folder test allo stesso livello di src
 - differente cartella ma stesso package della classe originale, test relativi a tale classe
 - la classe di test contiene metodi annotati con @Test che rappresentano i casi di test
 - @Test() può anche avere dei parametri di input
 - i metodi devono essere public per poter facilitare i test
 - per eseguire i test: Run as → JUnit test
 - nel tab di junit c'è anche il tasto rerun test
- assert di java
 - come documentazione per fare un controllo durante il debug

Testare un metodo

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

- [vedi counter_esempio](#)
- [vedi EsempioAssert](#)
- assert di java
 - come documentazione per fare un controllo durante il debug
- se un metodo deve essere eseguito una sola volta prima/dopo dei test
 - `@BeforeClass`
 - `@AfterClass`
- metodi che eseguiti prima di ogni test
 - `@Before`
- testare eccezioni
 - `@Test(expected = ArithmeticException.class)`
- AssertJ
 - libreria con asserzioni di vario genere
 - [DB, data, orario, ..](#)

Code coverage

- fornisce la copertura del codice con i casi di test di JUnit
- project → properties → codecover → enable codecover → seleziona BC, CC, LC, SC
 - i casi di test tengono traccia anche della copertura ora
 - devo prima indicare in relazione a quale classe calcolare la copertura:
click dx sulla classe → use for coverage measurement
 - run as → codecover measurement for junit
 - si apre il tab test sessions con i risultati
 - se da lì seleziono un test colora il codice (verde, giallo, rosso) per far capire cosa copro e cosa no
 - window → perspective → other → codecover
 - apre dei tab extra con dei dati sulla copertura
 - nel tab coverage, il campo term indica il livello di copertura di MCDC
 - il tab boolean analyzer ha la tabella del MCDC

Automatic test generation

- tool che producono in automatico anche l'oracolo
 - "riescono" a capire quale è il comportamento atteso
 - dopo serve però guardare a mano
- randoop
 - crea oggetti e input random
 - tasto dx classe da testare → run as → randoop test input → conferma selezione → imposta limite per generare test (tempo, num casi) → finish
 - introduce casi di test con relativo oracolo
 - tab randoop con numero di test totali
 - crea package randoop con dentro i vari test

- posso poi lanciarli con run as → codecov (anche l'intero package in una volta sola)
- crea test set molto ampi che potrebbero mandare in crisi codecov
- evosuite
 - più intelligente di randoop
 - si basa su delle euristiche per cercare di coprire tutti i casi
 - tasto dx su classe da testare → generate tests with evosuite
 - più pesante ma genera test più compatti

2.4 Some extra

Mutation Testing

- aka Fault-Based testing
- si basa sulla modifica (mutazione) del codice sorgente del programma in modi specifici e controllati, al fine di valutare l'efficacia dei test esistenti
 - I test esistenti vengono poi eseguiti sui mutanti
 - scoprire se i test sono in grado di "uccidere" i mutanti, cioè di rilevare le modifiche apportate e fallire a causa di esse
 - Se un test fallisce → il test ha rilevato la modifica e il mutante è considerato "ucciso". Questo è generalmente un buon segno, indicando che il test è efficace nel rilevare errori o anomalie
 - Se invece tutti i mutanti superano con successo tutti i test allora è considerato "sopravvissuto". Questo può indicare che i test non sono sufficientemente rigorosi o che mancano alcuni casi di test

mutanti equivalenti

- sw mutati sintatticamente diversi dal sw originale ma semanticamente identici
- i test faranno sempre sopravvivere questi mutanti
- [mutation_testing/Example](#)
- valutare la capacità di un test set → pitest
 - aggiungendo meaven tramite pitest si possono generare dei mutanti e vedere come si comportano i test
 - genera un report per capire quali branch e statement copro con i test

3. Program-based verification

3.0 Logic for computer programming

Propositional Logic

- analisi di relazioni tra proposizioni con connettori logici.

Declarative sentences

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

Atomic sentences

- sono sentenze dichiarative minimali e non scomponibili

¬ negazione

∧ and

∨ or

→ conseguenza logica (\neq implicazione)

↔ iff

⊢ Natural deduction : insieme di regole che danno una conclusione

Le regole dicono che "se la parte sopra è verificata allora lo è quella sotto", ad dx c'è il nome della regola

And-Introduction [$\wedge i$]

$$\frac{a \ b}{a \wedge b} \wedge i$$

Se si ha una proposizione P e una proposizione Q, è possibile concludere P AND Q. Ecco un esempio:

P: Il sole è caldo.

Q: Il cielo è azzurro.

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

And-Elimination [$\wedge e$]

$$\frac{a \wedge b}{a} \wedge e1 \quad \frac{a \wedge b}{b} \wedge e2$$

Se si ha una proposizione P AND Q, è possibile dedurre sia P che Q. Ecco un esempio:

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

P: Il sole è caldo. (And-elimination)

Q: Il cielo è azzurro. (And-elimination)

Double Negation [$\neg\neg$]

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

P: Marco non è scontento.

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

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

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

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

P: Oggi è venerdì.

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

Modus Tollens [MT]

$$\frac{a \rightarrow b \quad \neg b}{\neg a} MT$$

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

$\neg Q$: Marco non è andato al cinema.

$\neg P$: Quindi, oggi non è venerdì. (Modus tollens)

3.1 Design by contract

Contratto

- accordo tra cliente e fornitore che lega le parti
- scritto esplicitamente, dovrebbe essere parte stessa del sw
- specifica obblighi e benefici delle parti
- non ha clausole nascoste, gli obblighi sono quelli dichiarati
- utile nel testing come oracolo
- utile per la verifica runtime del codice
- a cosa serve
 - documentazione, anche javadoc con contratti inclusi
 - runtime checking \rightarrow self-check oracles per testing
 - formal verification

Precondizioni [P]

- obbligo per il cliente, ciò che viene richiesto
- definiscono input di un metodo e cosa fa
- possono essere deboli o forti

Postcondizioni [Q]

- obbligo per il fornitore, ciò che viene restituito
- definiscono output di un metodo

cliente

- controlla le precondizioni prima di invocare un metodo ma assume (non controlla) che le postcondizioni valgano dopo

fornitore

- assume (non controlla) che le precondizioni valgano e garantisce le postcondizioni

Per una funzione che restituisce il massimo in un array:

- int max(int[] a)

Possibili precondizioni

- a non è nullo: a!=null
- a contiene almeno un elemento: a.length >0

Possibili postconditioni

- Il risultato è maggiore uguale di ogni elemento:
 - per i da 0 a a.length-1 result >= a[i]
- Il risultato è uno degli elementi nell'array
 - esiste i da 0 a a.length -1 tale che result = a[i]

"è meglio scrivere un metodo semplice che soddisfa un contratto ben definito (con pre/post condizioni stringenti) piuttosto che un metodo che cerca di gestire tutte le situazioni possibili"

JML = Java Modeling Language

- insieme di tool per Dbc in Java
- si usa con //@ oppure /*@ */ per il multilinea
 - attenziona all'auto formatting di eclipse perché rende non riconoscibili queste righe
- quando importo un progetto jml in eclipse potrebbe servire click dx → enable jml
 - se poi ho varie cartelle o package potrebbe servire click dx → jml type-check
- ad ogni riga jml che scrivo e salvo vedo in console l'output di jml, se errore evidenzia di rosso nel codice
- conviene sempre premere il tasto RAC barra in alto per compilare jml nonostante sembra fare lo stesso ad ogni salvataggio del file
 - così runnare semplicemente come java application la classe e mi controlla le asserzioni jml nella console
 - se tutto ok non stampa nulla
- se eclipse da errore dicendo che manca il main, ma questo è presente, assicurarsi che la classe sia in una cartella e all'interno di un package, altrimenti click dx → build path → Use as Source Folder

Sintassi JML

- [vedi 3_jml/22](#)
- informal specification
 - `//@ requires (* comment *);`
 - come fosse un commento in jml
 - non controllato runtime, serve solo per la doc

- formal specification
 - `//@ requires x > 0;`
 - attenzione all'auto format di eclipse
 - `controllato` runtime
 - non può avere side effects, la valutazione deve essere T/F
`x++ modificherebbe il valore, x > 0 va bene invece`
- Precondizioni
 - `//@ requires .. ;`
- Post Condizioni
 - `//@ ensures .. ;`
- per riferirmi ad un valore di una variabile prima dell'invocazione del metodo da testare
 - `\old(variabile)`
- per riferirmi al valore restituito
 - `\result(variabile)`
- Invariante
 - regola sempre vera dopo la creazione di un oggetto e per prima di ogni metodo
 - (sulla classe!) controlla il valore dell'oggetto da un preciso momento (solitamente dalla creazione) in avanti
 - `//@ invariant`
 - deve essere public → `//@ public invariant ..`
- se e solo se
 - `< = = >`
 - `//@ ensures eta < 18 < = = > \result;`
 - se `result = true` allora $\eta < 18$, altrimenti $\eta > 18$
- implica
 - `= = >`
 - `// ensures eta < 18 = = > \result = true;`
- ne segue che (implica al contrario): `< = =`
- non se e solo se: `< = ! = >`
- quantificatore per ogni elemento: `\forall`
- esistenza di almeno un elemento: `\exists`
- operatori generali: `\sum , \product , \min , \max`
- contatore di occorrenze: `\num_of`
- quantificatori
 - (quantificatore tipo_variabile; range; espressione)
 - `\forall , \exists , \sum , \product , \min , \max , \num_of`
 - `//@ ensures (\forall int i; 0 ≤ i && i < a.length; \result ≤ a[i]);`
 - `//@ ensures (\exists int i; 0 ≤ i && i < a.length; \result = a[i]);`
 - `(\forall tipo_generico s; p; q)`
 - `(\min tipo_generico x; p; q)`
 - `(\forall int i; 0 < i; \result < 30)`
 - sulle collezioni
 - `/*@ ensures (\num_of int i; i ≥ 0 && i < \result.length; \result[i] > 0) = 1;`

- \num_of restituisce un numero
 - istruzioni su più righe:
 - /*@ .. @*/ oppure //@ ad ogni riga
 - Eccezioni
 - si sollevano con la violazione del contratto
 - manifestazioni di un bug
 - Null
 - si può specificare la non nullità
 - /*@ not_null @*/
 - public /*@ not_null @*/ File[] files;
 - void metodo(/*@ not_null @*/ String name) {}
 - Assert
 - per richiedere che una condizione sia verificata ad un certo punto del codice
 - non deve avere side-effects
 - può richiamare metodi puri
 - sono sia dentro JML che dentro Java normalmente, direi che conviene usare JML perché più espressivo
 - tutti gli elementi di a positivi in JML:
//@ assert(\forall int i; 0 ≤ i && i < a.length; a[i] > 0);
 - in java:
//assert Arrays.asList(a).stream().allMatch(x → (x > 0));
 - Metodi puri
 - query : metodi che non usano "=" o "+ - ="
 - che non hanno side-effects, cioè non cambiano lo stato dell'oggetto
 - indicati con /*@ pure @*/
 - public /*@ pure @*/ int balance()
 - posso usare metodi puri nei contratti (nasconde la complessità di un contratto in un metodo)
 - //@ pure;
boolean pos(int x) {}
 - //@ requires pos(s);
void foo(int s) {}
 - Assignable
 - metodi quasi puri che specificano esplicitamente cosa può essere modificato
 - poco usato
 - //@ assignable \nothing
 - Exceptional Behaviour
 - si applica in caso di post-state inaspettato, cioè quando il metodo lancia un'eccezione
 - gestisce nel contratto le eccezioni lanciabili
 - signals_only specifica che tipo di eccezioni un metodo può lanciare
 - signals specifica le post-condizioni da avere se un metodo ha lanciato delle eccezioni

```

/*@ 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;

- Also
  - contratti opzionali, come fosse un if-else nei contratti
  - /*@ requires x > 0;
    @ ensures \result > 0;
    also
    requires x < 0;
    ensures \result ≤ 0;
  */
- Loop Invariant
  - condizione che deve essere vera in ogni iterazione del loop (iniziale, durante e all'uscita)
  - da mettere dentro il metodo appena prima dei loop
  - per ricavarlo parto dalle postcondizioni del metodo
  - public void foo() {
      int i = 0;
      //@ loop_invariant 0 ≤ i && i ≤ 10;
      while (i < 10) {
          i++;
      }
  }

```

Information Hiding

- in JML il private/public segue lo standard di java
- si usano solo oggetti public nelle specifiche
- se serve usarne qualcuno privato si mette *spec_public* nella dichiarazione
private /*@ spec_public @*/ int peso;

Sottotipazione e JML

Sottotipazione

- ogni sottoclasse deve rispettare i contratti della sua superclasse, al massimo può espanderli (=aggiungere altri contratti)
- /*@ also
 requires sum > 1000;
 ensures balance > 1000;
 */

- facendo override di un metodo jml fa specification inheritance, ovvero eredita il contratto del metodo che modifica, quindi al massimo può estendere i suoi vincoli
 - posso solo restringere le preconditions
 - posso solo allargare le postconditions

```
public class Account {
    //@ requires x >= 1000;
    //@ ensures A;
    void deposit(int x){ ... }

    public class CheckingAccount extends Account{
        /*@ also
         * @ requires x > 0;
         * @ ensures B;*/
        void deposit(int x){ ...}}
```

Assert di Java

- parole proprie del linguaggio
 - di default gli assert sono disattivati a runtime
 - int x = 0;
assert x > 0 : "numero negativo vale " + x;
 - dopo : posso anche mettere un oggetto di cui fare `toString`
- controllati durante l'esecuzione, ma non in automatico, devo forzarlo io
 - click dx sulla classe → run as → run configurations.. → tab arguments → scrivi -ea in VM arguments → apply → run
 - dopo aver fatto così posso runnare come java application
 - in JUnit dovrebbe essere già selezionata la voce "add -ea to vm arguments .." in window → preferences → java → JUnit

Unreachable code

- In Java si possono usare in zone che non dovrebbero essere raggiunte per lanciare eccezioni
`assert false : "errore"`

condizioni implicite

- **implicit**
- per forzare un controllo in un if
`if (x < 0) {...} else { assert x ≥ 0; .. }`

Correttezza del sw

Correttezza

- consistenza dell'implementazione rispetto alla specifica
- in DbC specifica = contratto
- notazione base:
 - A = istruzioni del programma

- P = precondizioni di A
- Q = postcondizioni di A
- correttezza = {P} A {Q}

Classe Corretta se sono corretti:

- creazione : {P_costruttore} costruttore {inv ^ Q_costruttore}
- metodi : {Pmetodo ^ inv} metodo {Q_costruttore ^ inv}

Tool vari

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

3.2 Program Verification (OpenJML)

Verifica \neq Testing

Testing

- prova la **presenza** di difetti
- non può dimostrare l'assenza perché il test esaustivo è spesso difficile

Verifica

- prova **l'assenza** di difetti
- dimostra proprietà che sono garantite per ogni esecuzione (facendo assunzioni a priori)
 - **indip.** da un particolare input
- Metodi
 - ProofBased : dimostrazione logica
 - SemiAuto : forse serve intervento uomo
 - Property-oriented : si provano singole proprietà
 - Sequential programs : si ignora la concorrenza
 - pre/post development : durante lo sviluppo
- Vantaggi
 - Documentazione : più facile da scrivere
 - Time to market : minor tempo di debug
 - Refactoring : facilità di riuso
 - Certificazione : più facile ad essere certificato per applicazioni speciali (militare, medico...)
- Tools:
 - Java Path Finder → model checking
 - KeY → usato con JML tramite JMLKey

Struttura di test

- OpenJML e ESC con Z3 come solver
- non è presente nello zip, incluso però nel cheatsheet da me
 - copio nel progetto tutta la cartella "z3-4.3.0-x86"
 - window → preferences → openjml → openjml solvers → default = z3_4_3 → selezionare "external" → path "z3-4.3.0-x86\bin\z3.exe"

Arithmetic Modes: posso aver diversi scenari numerici in base a cosa sto considerando:

- Java Mode: overflow e underflow sono silenti
- Safe Mode: lancia errori di verifica se potrebbe verificarsi overflow
- Math (BigInt) Mode: niente under/overflow, unbounded

Si imposta con @CodeJavaMath, @CodeSafeMath, @codeBigintMath

Correttezza dei Loop

- trovare un loop invariant
 - deve implicare all'uscita la post condizione
 - e quindi risponde alla domanda "perché all'uscita vale?"
- deve valere:
 - prima di iniziare il loop
 - durante il loop (sfrutto condizione del loop)
 - dopo il loop
 - allora anche le postconditions di un codice corretto saranno valide
 - `//@ requires timer >= 0;`
`//@ ensures \result = 0;`
`/@ loop_invariant`
 `@ timer >= 0;`
 `@/`
 - ```
public static int countdown(int timer){
 while (timer > 0){
 timer --;
 }
 return timer;
}

//@ requires n >= 0;
//@ ensures \result == n * n;
//@ diverges true;
public int quadrato(int n) {
 int sum = 0;
 int i = 0;
 //@ loop_invariant sum == i * n && i <= n;
 while (i < n) {
 sum += n;
 i++;
 }
 return sum;
}
```
  - `//@ diverges true` assicura che il loop nel metodo finisce

- sempre posto come ultimo JML statement prima che inizi il metodo

Consistenza = Soundness

- ogni cosa che può essere provata è corretta

Completezza

- sistema di dimostrazioni completo = se ogni asserzione vera può essere dimostrata
  - logica proposizionale

Hoare

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

Correttezza totale

- sw corretto + termina
- da provare con la dimostrazione del fatto che termina sempre
  - ciclo che decrementa fino a zero: arriverà sempre a zero, anche se parte da un numero molto grande

### 3.3 Analisi Statica

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

Uno strumento di as S analizza il sorgente di un programma P per determinare se soddisfa una proprietà  $F_i$ , tipo "P non dereferenzia un puntatore nullo" oppure "P chiude tutti i file che ha aperto".

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

è un concetto più ampio di "cercare i bugs", si controlla la struttura dei file di risorse xml, json ,yaml etc etc, la compilazione, il controllo ortografico etc etc...

P è positivo se viola la proprietà  $F_i$ , negativo se la soddisfa.

Errori di AS:

- 1) Se S è sound (pessimistico): se accetta P è perché sicuramente soddisfa  $F_i$ 
  - a) Non ha falsi negativi, accetta solo chi soddisfa  $F_i$
  - b) Ha falsi positivi, rifiuta quelli ok
- 2) Se S è completo (ottimistico): se P soddisfa  $F_i$  allora accetta
  - a) Ha falsi negativi, può accettare anche chi non soddisfa  $F_i$

b) Non ha falsi positivi, se soddisfa accetta sempre

Meglio falsi negativi o falsi positivi?

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

Tools:

a) FindBugs:

loop infiniti, override di equals e hashCode incompatibili, dereferenziazione di null pointer, confronti con null, operazioni binarie strane, equals non avverabili, bad naming, exceptions ignore, uso di this, sincronizzazioni...  
classificati in corretti, bad practice, performance, correttezza multithread, malicious.

b) MyPy : tool python

c) Stan4J

## 3.4 Inspection

Esaminare un sw cercando costrutti conosciuti che portano problemi.

Usato soprattutto quando non ci sono altre tecniche o le altre tecniche non hanno abbastanza coverage.

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

Perché farle?

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

Code Inspection

- formale, di solito guidati da un moderatore che non ha scritto il codice, richiedono di conoscere le specifiche prima, viene prodotto un report dettagliato.
- usate checklist, esaminazioni paritarie (io controllo il tuo codice, tu il mio)
  - checklist tipo: Errori di design, errori di computazione, errori di confronto, errori di control flow, errori di subroutine, input/output, memoria...

Inspection  $\neq$  Testing

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

Solitamente non ci sono addetti standard alle ispezioni, ma sono ruoli temporanei, così da non far sentire tutti giudicati

## 4. Model verification

### 4.0 Models and Modeling

Modello

- utilità ingegneristica che permette di fare test e analisi su sistemi che ancora non esistono, tra cui analisi preliminari e soprattutto test automatizzati

Un modello deve essere:

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

Notazioni possibili:

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

Modello operazionale VS dichiarativo

- Operazionale = Imperativo = "Come faccio a far accadere X?"
- Dichiarativo = "Come mi accorgo che X è accaduto?"

### 4.1 Temporal Logic

Model Checking

- modo completamente automatico di fare una verifica formale composto da tre componenti all'interno di un framework:
  - Modello = ciò che descrive il sistema
  - Proprietà = ciò che vogliamo verificare (in una formula temporale)
  - Verifica = chiedere al model checker se il modello implica la proprietà
- do al tool il modello e mi dice se le proprietà sono verificate
  - M = comportamento della macchina
  - phi = proprietà della macchina
  - se M soddisfa phi → phi è vera per M

- model checking → verifica che una proprietà valga per un modello

## Logiche temporali

- Linear Time Logics (LTL)
  - il tempo è un **insieme di cammini**
  - cammino = sequenza di istanti di tempo
- Branching Time Logics (CTL)
  - il tempo è un **albero con istante corrente come radice**
- useremo entrambe ma senza metriche → non daremo dei limiti temporali **entro 3s**
  - il tempo sarà discreto  
**0,1,2,..**
  - tra un istante e l'altro il sistema resta fermo

## LTL

formule/proposizioni atomiche = AP

- {p, q, r, ..}
- lettere con cui si costruisce l'alfabeto

- Definiamo in maniera ricorsiva le formule LTL:
  - come la logica proposizionale (1) (| significa “oppure”) - in stile come grammatica BNF
- $$\phi ::= \top | \perp | p \in AP | \neg \phi | \phi \wedge \phi | \phi \vee \phi | \phi \rightarrow \phi$$
- $\top, \perp$  sono vero e falso
  - $\neg, \wedge, \vee, \rightarrow$  sono connettivi logici classici

## operatori temporali

$$\phi ::= \begin{array}{l} \top | \perp | p \in AP | \neg \phi | \phi \wedge \phi | \phi \vee \phi | \phi \rightarrow \phi \\ X\phi | F\phi | G\phi | \phi U \phi | \phi W \phi | \phi R \phi \end{array}$$

## Connettivi temporali

- X = next state → X vero nel prossimo stato
- F = some future state → X vero in qualche stato futuro
- G = globally in all future state → X sempre vero nei futuri stati

## operatori binari

- hanno due argomenti
- U = Until → phi\_1 vero finché vale phi\_2
- W = Release
- R = Weak-Until

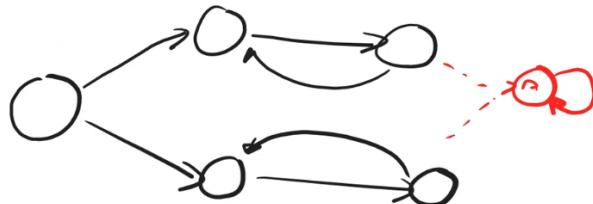
$\phi_1 \cup \phi_2$  $\phi_1 \wedge \phi_2$  $\phi_1 R \phi_2$ 

Ordine degli operatori

- 1) Prima il not, X F G
- 2) Poi  $\cup$   $\wedge$   $R$   $W$
- 3) Poi  $\wedge$  e  $\vee$
- 4) Poi  $\rightarrow$

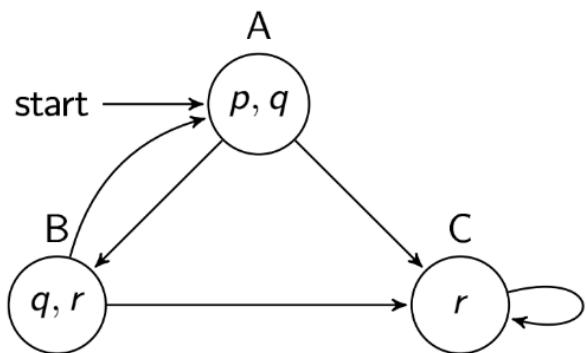
Transition System

- tipologia di modellazione che usiamo per i sistemi che interessano a noi
- formato da:
  - State Set = S
  - Stato iniziale =  $s_0$
  - Relazione di transizione
    - lega un qualsiasi s ad un  $s'$  (1:1 o 1:N)
  - Labelling function
    - $L : S \rightarrow P(AP)$
    - $P(AP)$  = power set (insieme delle parti) di proposizioni atomiche AP
    - $L$  indica le proposizioni di AP che sono vere in S
    - da ogni s esce una freccia verso un altro s, o sé stesso
      - impedisce i deadlock per design
      - per i sistemi reali però si inserisce sempre uno stato extra di deadlock



- Formalmente indicati come  $M = (S, s_0, \rightarrow, L)$
- possono essere facilmente rappresentati da grafi

**Example:** M has only three states A, B, and C. The atomic propositions AP = {p,q,r}. The only possible transitions are A → B, A → C, B → A, B → C and C → C; and if L(A) = {p, q}, L(B) = {q, r} and L(C) = {r}:



...

path

- sequenza di stati percorsa dentro una macchina
- scritto come  $s_0 \rightarrow s_1 \rightarrow s_2 \dots$
- sequenza potenzialmente infinita, a meno di deadlock
- si può troncare il percorso per fare dei ragionamenti

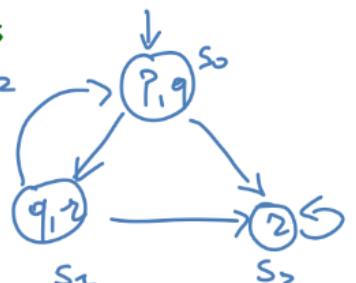
$$\pi : \overset{1}{s_0} \rightarrow \overset{2}{s_1} \rightarrow \overset{3}{s_0} \rightarrow \overset{4}{s_2} \rightarrow \overset{5}{s_2}$$

~~$s_0 \rightarrow s_2 \rightarrow s_0$~~

$$\pi^1 : s_0 \rightarrow s_1 \dots$$

$$\pi^2 : s_1 \rightarrow s_0 \dots$$

$$\pi^3 : s_0 \rightarrow s_2 \rightarrow s_2$$



...

- validità di una formula LTL su un path
  - 3) valido se tale AP vale nel primo stato ( $s_1$  in realtà  $s_0$ )  $\rightarrow$  se una LTL non ha operatori temporali, allora guardo solo il primo stato

$$1. \pi \models \top$$

$$2. \pi \not\models \perp$$

$$3. \pi \models p \text{ iff } p \in L(s_1)$$

$$4. \pi \models \neg \varphi \text{ iff } \pi \not\models \varphi$$

$$5. \pi \models \varphi_1 \wedge \varphi_2 \text{ iff } \pi \models \varphi_1 \text{ and } \pi \models \varphi_2$$

$$6. \pi \models \varphi_1 \vee \varphi_2 \text{ iff } \pi \models \varphi_1 \text{ or } \pi \models \varphi_2$$

$$7. \pi \models \varphi_1 \rightarrow \varphi_2 \text{ iff } \pi \models \varphi_2 \text{ whenever } \pi \models \varphi_1$$

- 8) nel prossimo stato devo valere phi  $\rightarrow$  salto il primo stato e dico che dal secondo deve valere phi
- 9) phi vale in ogni sotto path
- 10) phi vale almeno in uno stato futuro

$$G \text{ not(phi)} = \text{not}(F\text{phi})$$

`Fnot(phi) = not(Gphi)`

## Definition

Let  $M = (S, \rightarrow, L)$  be a model and  $\pi = s_1 \rightarrow \dots$  be a path in  $M$ . Whether  $\pi$  satisfies an LTL formula is defined by the satisfaction relation  $\models$  as follows:

- 8.  $\pi \models X \varphi$  iff  $\pi^2 \models \varphi$
  - 9.  $\pi \models G \varphi$  iff, for all  $i \geq 1$ ,  $\pi^i \models \varphi$
  - 10.  $\pi \models F \varphi$  iff there is some  $i \geq 1$  such that  $\pi^i \models \varphi$

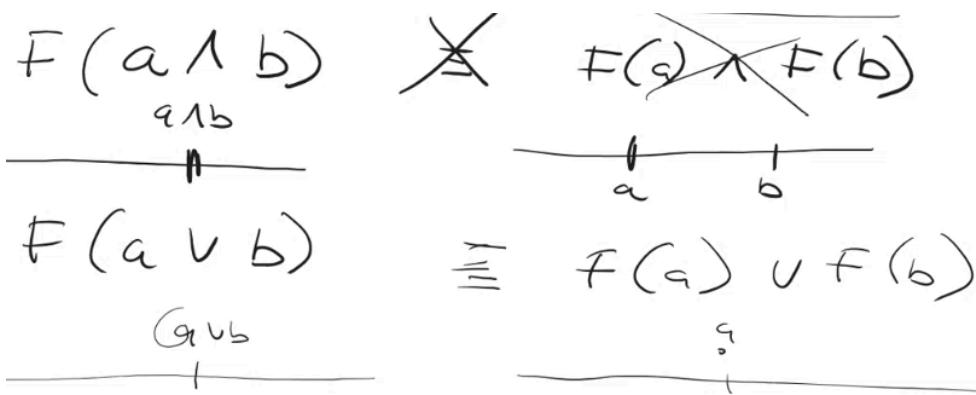
- until: vale a finché non inizia a valere b (a può essere valido o meno nello stato in cui inizia a valere b) → deve per forza esserci uno stato in cui varrà b  
 $a \sqcup b = (a \sqcap b) \text{ and } Fb$   
weak until: come until ma può anche non esserci uno stato futuro in cui vale b, ma deve continuare a valere sempre allora  
 $a \sqcup b = (a \sqcup b) \text{ or } Ga$   
release: quando a diventa vero allora b può anche diventare falso

  11. (**Until**)  $\pi \models a \sqcup b$  iff there is some  $i \geq 1$  such that  $\pi^i \models b$  and for all  $j = 1, \dots, i-1$  we have  $\pi^j \models a$
  12. (**Weak Until**)  $\pi \models a \sqcup b$  iff either there is some  $i \geq 1$  such that  $\pi^i \models b$  and for all  $j = 1, \dots, i-1$  we have  $\pi^j \models a$ ; or for all  $k \geq 1$  we have  $\pi^k \models a$
  13. (**Release**)  $\pi \models a \sqcap b$  iff either there is some  $i \geq 1$  such that  $\pi^i \models a$  and for all  $j = 1, \dots, i$  we have  $\pi^j \models b$ , or for all  $k \geq 1$  we have  $\pi^k \models b$ .

## distributività di $G$ e $F$

$$G(a \wedge b) = G(a) \wedge G(b)$$

$$G(a \vee b) \neq G(a) \vee G(b)$$



validità di una formula per una macchina

- la formula deve valere in tutti gli stati di tutti i percorsi fattibili

pattern LTL

- safety:  $G(a)$
- liveness (vitalità del sistema, evolve e non sta fermo):  $F(a)$
- $GF(a)$ : in ogni stato in futuro varrà a
- $G(a \rightarrow Fb)$ : se vale a allora in futuro varrà b

**Absence – P is false:**

|                 |                                                               |
|-----------------|---------------------------------------------------------------|
| Globally        | $G(\neg P)$                                                   |
| Before R        | $F R \rightarrow (\neg P \vee R)$                             |
| After Q         | $G(Q \rightarrow G(\neg P))$                                  |
| Between Q and R | $G((Q \wedge \neg R \wedge F R) \rightarrow (\neg P \vee R))$ |

**Existence P becomes true :**

|                     |                                                                         |
|---------------------|-------------------------------------------------------------------------|
| Globally            | $F_{\exists}(P)$                                                        |
| (*) Before R        | $\neg R \rightarrow (P \wedge \neg R)$                                  |
| After Q             | $G(\neg Q) \mid F(Q \wedge F P)$                                        |
| (*) Between Q and R | $G(Q \wedge \neg R \rightarrow (\neg R \rightarrow (P \wedge \neg R)))$ |
| (*) After Q until R | $G(Q \wedge \neg R \rightarrow (\neg R \rightarrow (P \wedge \neg R)))$ |

**Universality P is true :**

|                     |                                                          |
|---------------------|----------------------------------------------------------|
| Globally            | $G(P)$                                                   |
| Before R            | $F R \rightarrow (P \vee R)$                             |
| After Q             | $G(Q \rightarrow G(P))$                                  |
| Between Q and R     | $G((Q \wedge \neg R \wedge F R) \rightarrow (P \vee R))$ |
| (*) After Q until R | $G(Q \wedge \neg R \rightarrow (P \wedge R))$            |

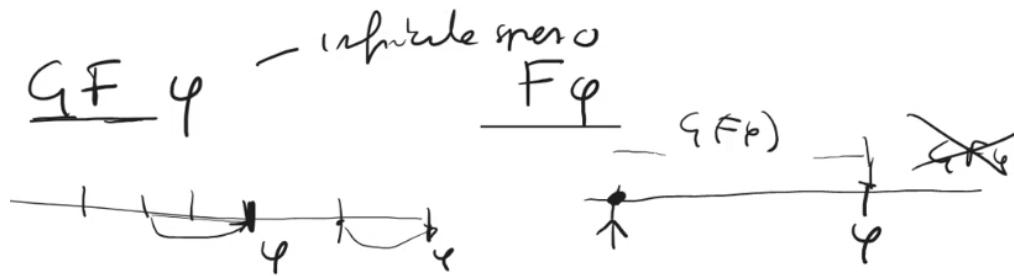
- "a è sempre vero prima che accada b" :

$$F(b) \rightarrow (a \vee b)$$

- "a sempre vero dopo che accade b" :

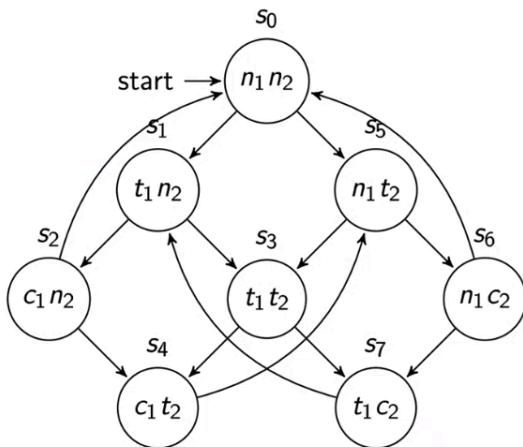
$$G(b \rightarrow G(a))$$

- $GF(\phi)$  = infinitamente spesso = in qualsiasi stato sono in futuro vale  $\phi$



### mutual exclusion

- due processi condividono la stessa risorsa  $\rightarrow$  no ( $c_1, c_2$ )



Every process can be in state: {non critical (n), trying to enter (t), critical state (c)}.

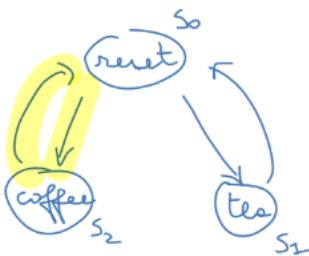
### - proprietà

- safety = solo un processo alla volta in zona critica  
 $\rightarrow G \text{ not}(c_1 \text{ and } c_2)$
- liveness = se un processo chiede di entrare in zona critica prima o poi potrà farlo  
 $\rightarrow G (t_1 \rightarrow F(c_1))$   
FALSA perché se path = {s0, (s1, s3, s7, ...) ...} vale t1 ma mai c1  $\rightarrow$  non riesco in LTL
- non-blocking = un processo può sempre chiedere di entrare in zona critica  
 $\rightarrow$  non riesco in LTL
- no strict sequencing = non è detto che il primo processo a fare richiesta sia il primo ad entrare in zona critica  
 $\rightarrow$  trovo un path in cui non c'è strict sequencing

### limiti LTL

- vede il tempo come un path  $\rightarrow$  per dimostrare che qualcosa vale sempre occorre dimostrarlo per ogni path

- non posso esprimere il fatto che in futuro, non sempre, potrà succedere qualcosa

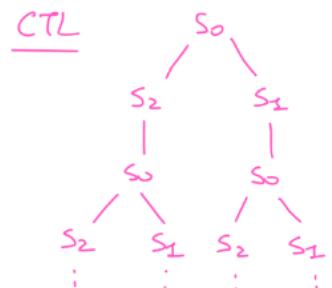


$$AP = \{ \text{reset}, \text{coffee}, \text{tea} \}$$

"è possibile prendere un tea"

$$\Rightarrow LTL = F(\text{tea})$$

non vole per t path



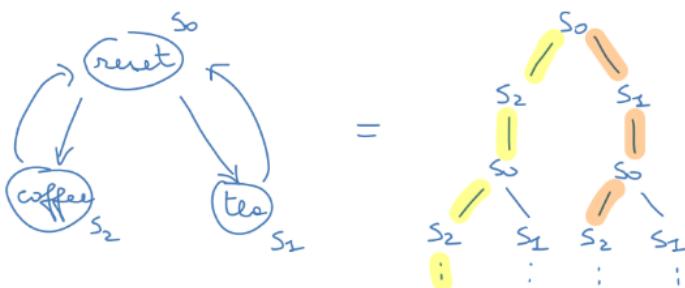
## CTL

branching-time temporal logic

- il tempo non è più visto come una sequenza, ma come un albero infinito
  - permette di specificare proprietà temporali
  - futuro non deterministico (modelli non più lineari)
- ad ogni ramo posso prendere tutti i possibili stati

## sintassi

- LTL allargata
- operatori temporali  $\rightarrow$  connettivi temporali a due lettere
  - prima lettera = quantificatore sul path
    - A = su tutti i path
    - E = almeno su un path
  - seconda lettera = come LTL sul tempo = X, F e G
  - AX(p) = in ogni stato successivo vale p
  - EX(p) = esiste uno stato successivo in cui vale p
  - AF(p) = per tutti i path prima o poi vale p
  - EF(p) = almeno un path dove prima o poi vale p
  - AG(p) = p vale in ogni stato di ogni path
  - EG(p) = esiste almeno un path in cui in ogni stato vale p



$AF(\text{tea}) = \text{"primo o poi è sempre possibile prendere il tea"} \Rightarrow \text{FALSO}$

$EF(\text{tea}) = \text{"primo o poi è possibile prendere il tea"} \Rightarrow \text{VERO}$

## Ordine degli operatori

- operatori unari = AX, EX, AF, EF, AG, EG

2. Poi URW
3. operatori binari  $\wedge$  e  $\vee$
4.  $\rightarrow$
5. AU ed EU

formule CTL ben formate

$AG (q \rightarrow EG r)$   
 $EF E(r \cup q)$   
 $A[p \cup EF r]$   
 $EF EG p \rightarrow AF r$

formule CTL non ben formate

$EF G r$

- ▶  $A!G!p$
- ▶  $F[r \cup q]$
- ▶  $EF(r \cup q)$
- ▶  $AEF r$
- ▶  $A[(r \cup q) \wedge (p \cup r)]$

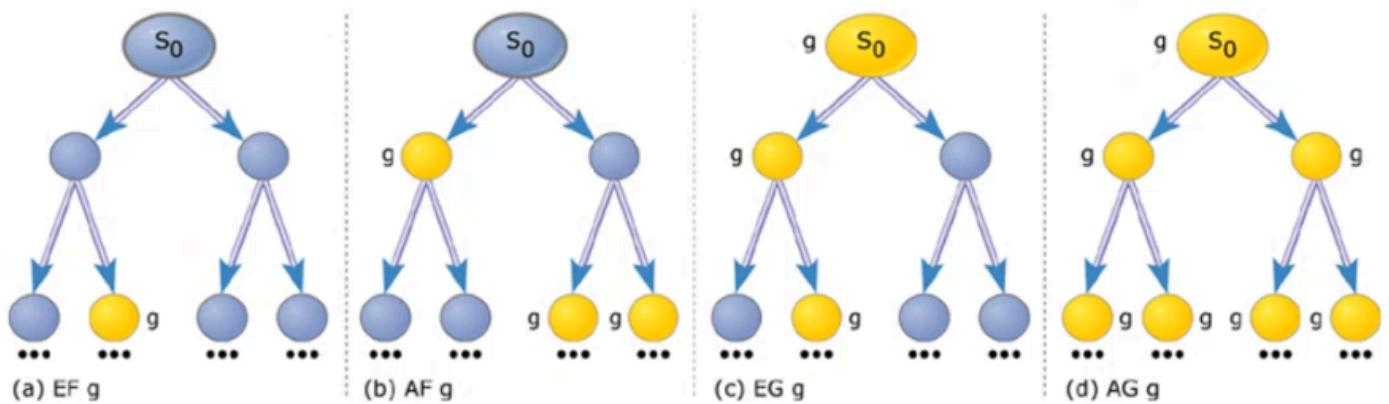
semantica

### Definition

Let  $M = (S, \rightarrow, L)$  be a model for CTL,  $s$  in  $S$ ,  $\varphi$  a CTL formula.  
The relation  $M, s \models \varphi$  is defined by structural induction on  $\varphi$ .

- ▶ If  $\varphi$  is atomic, satisfaction is determined by  $L$ .
  - ▶ If the top-level connective of  $\varphi$  is a boolean connective ( $\wedge$ ,  $\vee$ ,  $\neg$ , etc.) then the satisfaction question is answered by the usual truth-table definition and further recursion down  $\varphi$ .
  - ▶ If the top level connective is an operator beginning A, then satisfaction holds if all paths from  $s$  satisfy the ‘LTL formula’ resulting from removing the A symbol.
  - ▶ Similarly, if the top level connective begins with E, then satisfaction holds if some path from  $s$  satisfy the ‘LTL formula’ resulting from removing the E.

validità di una formula CTL



### CTL properties pattern

| Absence – P is false: |                                   |
|-----------------------|-----------------------------------|
| Globally              | $AG(\neg P)$                      |
| Before R              | $A[(\neg P \mid AG(\neg R)) W R]$ |
| After Q               | $AG(Q \rightarrow AG(\neg P))$    |

| Existence P becomes true : |                                                         |
|----------------------------|---------------------------------------------------------|
| Globally                   | $AF(P)$                                                 |
| (*) Before R               | $A[\neg R W (P \& \neg R)]$                             |
| After Q                    | $A[\neg Q W (Q \& AF(P))]$                              |
| (*) Between Q and R        | $AG(Q \& \neg R \rightarrow A[\neg R W (P \& \neg R)])$ |
| (*) After Q until R        | $AG(Q \& \neg R \rightarrow A[\neg R U (P \& \neg R)])$ |

| Universality P is true : |                                                          |
|--------------------------|----------------------------------------------------------|
| Globally                 | $AG(P)$                                                  |
| (*) Before R             | $A[(P \mid AG(\neg R)) W R]$                             |
| After Q                  | $AG(Q \rightarrow AG(P))$                                |
| (*) Between Q and R      | $AG(Q \& \neg R \rightarrow A[(P \mid AG(\neg R)) W R])$ |
| (*) After Q until R      | $AG(Q \& \neg R \rightarrow A[P W R])$                   |

### equivalenza tra formule CTL

$$\neg AF \varphi \equiv EG \neg \varphi \text{ and } EG \varphi \equiv \neg AF \neg \varphi$$

$$\neg EF \varphi \equiv AG \neg \varphi \text{ and } AG \varphi \equiv \neg EF \neg \varphi$$

$$\neg AX \varphi \equiv EX \neg \varphi.$$

We also have the equivalences  $AF \varphi \equiv A[\top U \varphi]$  and

- $EF \varphi \equiv E[\top U \varphi]$  which are similar to the corresponding equivalences in LTL.

→ useremo solo AF, EU ed EX per snellire CTL sfruttando queste equivalenze

### LTL vs CTL

- ci sono ancora delle cose che ci possono esprimere in LTL ma non in CTL  
"per ogni path dove vale p allora vale anche q"  
LTL:  $F(p) \rightarrow F(q)$   
CTL:  $AF(p) \rightarrow AF(q)$  ma non intendo per tutti i path, solo per quelli dove prima vle p, con EF invece vorrebbe dire che i path con p e q potrebbe essere anche diversi fra loro

## 4.2 Model Checking

algoritmo

- scopre in modo automatico se una proprietà phi vale per una certa macchina
- NuSMV
  - model checker
  - verifica formale di sistemi dinamici
  - sfrutta logica temporale

model checking in CTL

- macchina in asmeta
- proprietà in CTL
- risolve con algoritmo di labeling
  - input: model M ( $S \rightarrow L$ ) e formula phi in CTL
  - output: set di stati di M che soddisfano phi
  - primo passo: trasforma phi tramite le equivalenze di CTL
  - secondo passo: segna gli stati di M dove vale phi

### Algoritmo di Labeling

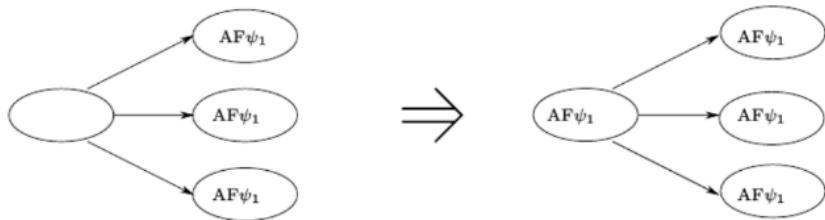
Case analysis over  $\psi$ . If  $\psi$  is

- ▶  $\perp$ : then no states are labelled with  $\perp$
- ▶  $p$ : then label every  $s$  such that  $p \in L(s)$
- ▶  $\psi_1 \wedge \psi_2$ :
  - ▶ do labelling with  $\psi_1$  and with  $\psi_2$
  - ▶ label  $s$  with  $\psi_1 \wedge \psi_2$  if  $s$  is already labelled both with  $\psi_1$  and with  $\psi_2$
- ▶  $\neg\psi$ :
  - ▶ do labelling with  $\psi$
  - ▶ label  $s$  with  $\neg\psi$  if  $s$  is not labelled with  $\psi$ .

► AF  $\psi$ :

► do labeling with  $\psi$

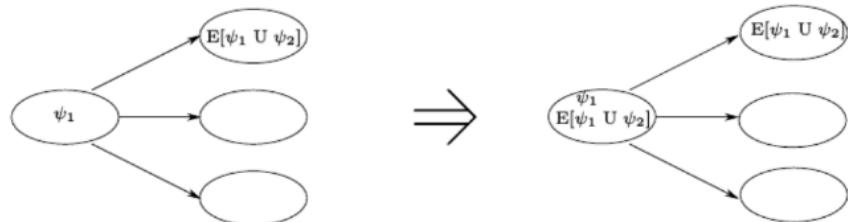
- If any state  $s$  is labelled with  $\psi$ , label it with AF  $\psi$ .
- Repeat: label any state with AF  $\psi$  if all successor states are labelled with AF  $\psi$ , until there is no change. See picture



► E[ $\psi_1 \cup \psi_2$ ]

► do labeling for  $\psi_1$  and  $\psi_2$  *beste un solo stato successivo*

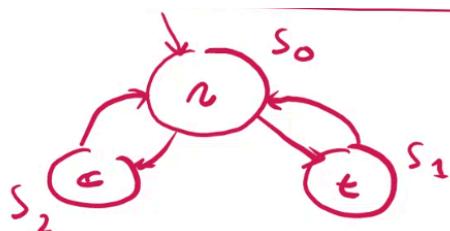
- If any state  $s$  is labelled with  $\psi_2$ , label it with E[ $\psi_1 \cup \psi_2$ ]
- Repeat: label any state with E[ $\psi_1 \cup \psi_2$ ] if it is labelled with  $\psi_1$  and at least one of its successors is labelled with E[ $\psi_1 \cup \psi_2$ ], until there is no change.



► EX  $\psi$ : *faccio un solo passo indietro e mi fermo*

► do labeling for  $\psi$

- label any state with EX $\psi$  if one of its successors is labelled with  $\psi$



$E(t \vee c \cup n)$

|       | $n$ | $t$ | $c$ | $t \vee c$ | $E(t \vee c \cup n)$ |
|-------|-----|-----|-----|------------|----------------------|
| $s_0$ | X   |     |     |            | X                    |
| $s_1$ |     | X   | X   |            | X                    |
| $s_2$ |     |     | X   | X          | X                    |



$EX(\neg P)$

|       | P | $\neg P$ | $EX(\neg P)$ |
|-------|---|----------|--------------|
| $S_0$ | x |          | x            |
| $S_1$ | - | x        |              |
| $S_2$ | - | x        | x            |
| $S_3$ | - | x        | x            |



$$AG(q \vee r) \equiv \neg E F (\neg (q \vee r)) \\ \neg E(T \cup \neg (q \vee r))$$

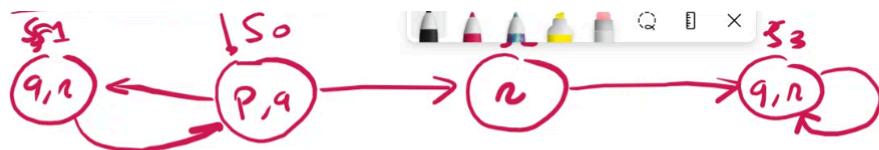
|       | q | r | $q \vee r$ | $\neg(q \vee r)$ | T | $E(T \cup \neg(q \vee r))$ | $\neg$ |
|-------|---|---|------------|------------------|---|----------------------------|--------|
| $S_0$ | x |   | x          | -                | x | -                          | x      |
| $S_1$ | x | x | x          | -                | x | -                          | x      |
| $S_2$ | x | x | -          | x                | - | -                          | x      |
| $S_3$ | x | x | x          | -                | x | -                          | x      |



$$AG(P) \equiv \neg E F \neg P \equiv \neg E(T \cup \neg P)$$

|       | P | $\neg P$ | T | $E(T \cup \neg P)$ | $\neg$ | $AG(P)$ |
|-------|---|----------|---|--------------------|--------|---------|
| $S_0$ | x | -        | x | x                  | -      |         |
| $S_1$ | . | x        | x | x                  | -      |         |
| $S_2$ | . | x        | x | x                  | -      |         |
| $S_3$ | . | x        | x | x                  | -      |         |

$E(true \cup \neg P) \rightarrow$  prendo subito gli stati dove vale  $\neg P$  + stati precedenti in cui esiste uno stato futuro già marcato



$$AG(\textcolor{red}{r}) \equiv \neg EF \neg r \equiv \neg E(T \cup \neg r)$$

|       | $r$ | $\neg r$ | T | $E(\neg r \wedge r)$ | $\neg L$ | $\dots$ | $AG(r)$ |
|-------|-----|----------|---|----------------------|----------|---------|---------|
| $S_0$ | -   | x        | x | x                    | -        |         |         |
| $S_1$ | x   | -        | x | x                    | -        |         |         |
| $S_2$ | x   | -        | x | -                    | x        |         | x       |
| $S_3$ | x   | -        | x | -                    | x        |         | x       |

## 4.3 Abstract State Machines ASMETA

- specificare il comportamento delle macchine su cui poi fare model checking per dimostrare proprietà scritte in LTL/CTL
- modellazione di sw complessi
- metodo di specifica

### ASM vs FSM

- ASM molto adatte a sistemi reattivi che leggono di continuo input dall'ambiente
- ASM = FSM con stati generalizzati e più ricchi
  - uno stato non è più solo un insieme di formule
  - alfabeto input/output infinito (espressioni, azioni, ...)
  - input/output come label delle transizioni  
push/spegni
  - transizione dipende solo dallo stato corrente
- concetto di sottomacchina
- composizione sequenziale/parallela
- stato astratto
- macchina a stati finiti = notazione formale per rappresentare in modo astratto il comportamento di un sistema

### Asmeta

- linguaggio/editor = AsmetaL
- ambiente eclipse = Asmee
- simulatore e animatore = AsmetaS
- linguaggio per scenari = Avalla
- tool analisi statica = asmetaMA
- tool model checking = AsmetaSMV

### AsmetaS

- da linea di comando: `java -jar AsmetaS.jar <filename>`

- [Asmeta.jar](#)
- opzioni per la terminazione
  - numero fisso di passi: `java -jar AsmetaS.jar -n 3 <filename.asm>`
  - finché l'insieme degli aggiornamenti è vuoto:  
`java -jar AsmetaS.jar -n? <filename.asm>`

## Asmee

- simulatore grafico in eclipse
- nuovo file AsmetaL: file → new file → other → asmetaL new file
- syntax highlighting : window → preferences → asmee
- check di un file asmetaS con asmetaL:
  - click sx su file asm → click simbolo a v verde barra strumenti in alto
    - da in console eventuali errori
- simulazione
  - click simbolo in alto viola con trg bianco = simulazione interattiva con l'utente
  - click simbolo con trg viola e foglio = simulazione con valori random
  - oppure usando l'animatore grafico (A a sx della v verde)
    - click "do one interactive step" per vedere l'evoluzione stato per stato
    - click "do random step/s" non domanda valori all'utente
      - posso anche specificare quanti passi random fare in automatico

## Linguaggio AsmetaL

### • Linguaggio strutturale

- . costrutti per definire la struttura (scheletro) di una ASM mono-agente o sinc./asinc. multi-agente

### • Linguaggio delle definizioni

- . costrutti per introdurre (dichiarare e definire) domini (*tipi del linguaggio*), funzioni (con domini e codomini), regole di transizione, e assiomi

### • Linguaggio dei termini

- . **termini di base** come nella logica del primo ordine (costanti, variabili, termini funzionali  $f(t_1, t_2, \dots, t_n)$ )
- . **termini speciali** come tuple, collezioni (insiemi, sequenze, bag, mappe), ecc.

### • Linguaggio delle regole

- . **regole di base** come skip, update, parallel block, ecc.
- . **turbo regole** come seq, iterate, turbo submachine call, ecc.

[vedi 5\\_asmeta](#)

## struttura di una ASM in AsmetaL

- `asm <nome_file>` → nome asm deve essere uguale nome file

- `import StandardLibrary` → la cosa più semplice è copiarla dentro il progetto, altrimenti si specifica il path nel sistema
- `signature: ..` = definizione di **stato**
- `definitions: ..` = definizione delle **regole**
- `default init initial_state;`

## ASM stati

- stato definito da un insieme di valori di qualsiasi tipo
  - memorizzati in celle di memoria = locazioni
  - in programmazione sono le variabili
  - in OOP sono i campi dell'oggetto
  - in ASM sono dette funzioni
    - in senso matematico (dominio : codominio), non in senso informativo
- cardinalità delle funzioni
  - 0-arie : variabili e costanti, **senza argomenti**
  - n-arie : mappe/array/**funzioni**
- funzioni statiche VS dinamiche
  - dinamiche = cambia valore nel passaggio da uno stato all'altro
    - se arità 0 sono le comuni **variabili della programmazione**
  - statiche = interpretazione costante
    - se arità 0 sono dette **costanti**

## Domini

### domini predefiniti (basic type-domains)

- Complex, Real, Integer, Natural, String, Char, Boolean, Rule, Undef
- **basic domain Real**

### domini custom

- definiti dall'utente, devono iniziare con la maiuscola
- astratti
  - non definiti se non con funzioni definite su tale dominio
- enumerativi
  - tutte lettere maiuscole
- a partire da altri domini già esistenti
- `signature:`  
`// stato della mia macchina`

```
// domini
```

```
abstract domain Persona
```

```
enum domain Light = {ON, OFF}
```

- molto utile usare enum perché con le stringhe NuSMV non lavora bene
- domini concreti = sottoinsiemi dai domini
- `signature:`
- `signature:`
- `domain Age subsetof Integer`

```
definitions:
domain Age = {0 : 300}
```

vedi Clock.asm

domini strutturati

- sequenze
  - Seq(D) = D è il dominio base delle possibili sequenze
  - domain MyDomain subsetof Seq(Integer) // lista di interi subset di Integer
- insiemi
  - Powerset(D) = D è il dominio base dell'insieme delle parti
  - Bag(D)
    - D è il dominio base dei possibili bag
    - insiemi con ripetizione
  - Prod(D1, .., Dn) = D1, .., Dn sono i domini del prodotto cartesiano  
domain CoppiaIntString subsetof Prod(Integer, String)

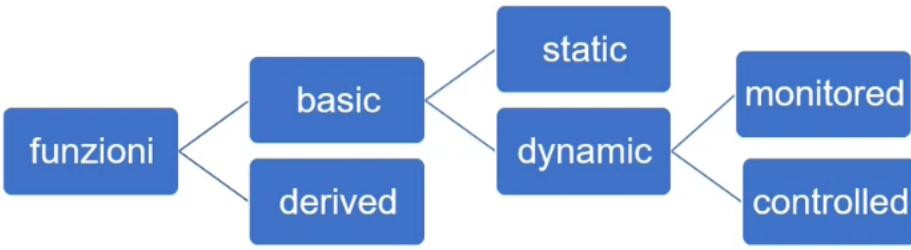
## Funzioni

costanti

- funzioni 0-arie statiche
  - no argomenti
  - non cambiano nel tempo
- simboli definiti una volta per tutte
- ogni vocabolario ASM contiene le costanti undef, true, false
- i numeri sono costanti numeriche
- le stringhe sono costanti
- aggiungibili dall'utente  
`Costante minVoto = 18;`

classificazione

- dynamic
  - i valori dipendono dagli stati della asm M
  - monitored
    - solo lette da M
    - scritte dall'ambiente in cui M è inserita
    - in simulazione vanno domandate all'utente o lette da un file esterno
  - controlled
    - lette e scritte da M
- derived
  - non fanno propriamente parte dello stato
  - il loro valore è computabile a partire dalle altre funzioni dello stato



definire funzioni in Asmetal

- tipo nome : codominio
- costante
  - signature:  
static oreDelGiorno : Integer // dichiarazione  
definitions:  
function oreDelGiorno = 24 // definizione
- controlled
  - signature:  
dynamic controlled hour: Hour
- funzioni statiche
  - definite tramite una legge fissa  
somma tra due numeri
  - definibili custom  
signature:  
**static massimo: Prod(Integer, Integer) → Integer**  
static lunghezza: String → Natural
    - "→" per dire che prende un dominio e ne restituisce un altro
    - il dominio delle funzioni n-arie sono n domini, serve quindi "Prod" per creare un dominio strutturato
  - oltre a dichiarle vanno anche definite  
definitions:  
**function massimo(\$a in Integer, \$b in Integer) = \$a + 1**  
\$ per indicare le variabili logiche, cioè quelle che non fanno parte dello stato, sono parametri delle funzioni
  - le variabili statiche di domini astratti sono istanze predeterminate  
→ non vanno definite  
signature:  
abstract domain Computer  
**static miocomputer : Computer**
- funzioni dinamiche n-arie
  - accettano parametri in input ma **quello che restituiscono cambia nel tempo**

funzioni ASM vs campi di Java

- simili tra loro
- **java: class Student{String name}**

- ASM:  
abstract domain Student  
controlled name: Student → String

## Definizioni

domini concreti statici  
domain Age = {0 : 300}

funzioni statiche  
static maxG: Prod(Integer, Integer) → Integer  
function maxG(\$x in Integer, \$y in Integer) = 10

funzioni matematiche

- function opp(\$x in Integer) = -\$x
- max(2,3)
- abs(-4)
- abs(max(-2,-3))

sequenze e insiemi

- sequence: [t1, .., tn]
- set: {t1, .., tn}
- bag: <t1, .., tn>
- map: {t1→s1, .., tn→sn}

if e let terms

- if per definire un valore condizionale

function maxG(\$x in Integer, \$y in Integer) =  
    if \$x > \$y then  
        \$x  
    else  
        \$y  
    endif

- let per introdurre variabili temporanee (locali ad una procedura)

function doppioQuadrato(\$x in Integer) = // \$x \*\$x + \$x \*\$x  
    let (\$q = \$x \* \$x) in \$q + \$q endlet

exists/forall terms

- check condizioni sugli insiemi
- return true/false

|                  |                                                             |
|------------------|-------------------------------------------------------------|
| Exist Term       | <b>(exist v1 in D1,...,vn in Dn with Gv1,...,vn)</b>        |
| ExistUnique Term | <b>(exist unique v1 in D1,...,vn in Dn with Gv1,...,vn)</b> |
| Forall Term      | <b>(forall v1 in D1,...,vn in Dn with Gv1,...,vn)</b>       |

**(exist \$x in {2,5,7} with \$x=2)**

**(exist unique \$x in X with \$x=0)**

Esercizio: funzione che dato un insieme di interi e un intero dice se quel valore è più grande di ogni elemento

## Regole e transizioni di stato

definiti domini e funzioni devo specificare come evolve lo stato della ASM

stato corrente = snapshot del sistema al momento attuale

- check delle condizioni
- applicazione di eventuali modifiche
- eventuale cambio di stato secondo le regole definite

### regole

- sintassi di un programma ASM
- espressioni sintattiche generate tramite l'uso di costruttori appositi
- iniziano sempre con **r\_**
- skip rule = non fare niente, resta fermo nello stato corrente
  - **rule r\_inc\_min = skip**
- update rule = aggiorna una precisa locazione della ASM
  - a sx ho una locazione di memoria, quindi non un'espressione
  - a dx può esserci un'espressione
  - update fatto solo quando passo nell'altro stato
    - raccoglie tutti gli update (= regole che possono scattare) e poi li applica per ottenere lo stato successivo
  - **rule r\_inc\_min = minute := minute + 1**
    - minute = locazione da aggiornare, valutato nello stato corrente
    - minute + 1 = valore che ci metto dentro, aggiornato nello stato successivo
    - assegnamento → **=**
    - confronto → **:=**
- main rule
  - regole che il simulatore deve chiamare
  - come fosse il main di java → quindi le regole definite ma non incluse in questa sezione non sono usate?

- rule r\_inc\_sec = ...
 

```
main rule r_main = r_inc_sec[]
 - rule[param1, ..] quando devo richiamare un'altra rule
```
- conditional rule
 

```
- if <espressione booleana> then <rule1> else <rule2> endif
```

```
rule r_inc_sec =
 if signal then
 if second < 59 then
 second := second + 1
 else
 second := second + 2
 endif
 endif

rule r_inc_sec =
 if signal and second < 59 then
 second := second + 1
 else
 second := second + 2
 endif
```

- block rule
  - costruttore di esecuzione parallela
  - qualsiasi blocco con più di un'operazione va racchiuso dentro il par, anche par annidati multipli sono ammessi
  - raccoglie l'update set in parallelo e poi lo applica

```
rule r_inc_sec =
 if signal then
 if second < 59 then
 second := second + 1
 else
 par
 second := 0
 r_inc_min[] // richiama un'altra rule
 endpar
 endif
 endif
```

- update inconsistenti
  - una rule di transizione tenta di aggiornare una variabile con due update diversi
 

```
par
 x := x + 1
 x := 1
 endpar
```
  - errore segnalato nel simulatore
- choose rule
  - implementa il non-determinismo

- come fosse un random
- posso mettere anche delle condizioni
- preferences → asmeta → avalla → simulator → shuffle choose rule, permette di scegliere random ma in modo eguale

par

```
x := y + 1
choose $r in {0:5} with $r < 4 do y := y + $r
```

endpar

stato iniziale

- inizializzo il valore delle funzioni
- posso avere anche più stati iniziali

default init s0:

```
function x = 0
function y = 0
```

impostare un null value

erogato := **undef**

## 4.5 ASM model checking

proprietà CTL/LTL

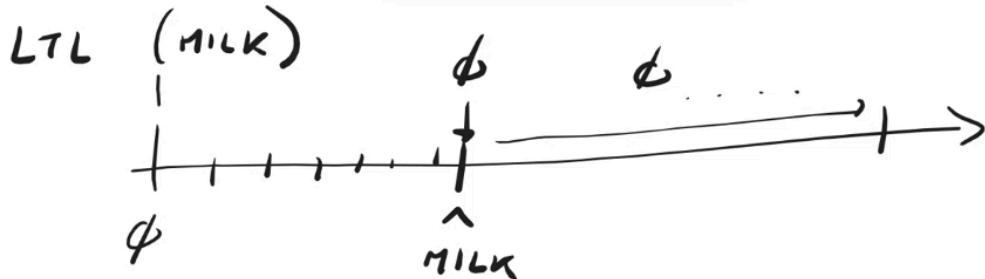
- dichiarate dopo la sezione degli invarianti
  - appena prima della main rule
- prima le CTL, poi le LTL
  - **CTLSPEC p**  
**CTLSPEC q** con p e q espressioni booleane
- serve importare le librerie CTLlibrary.asm e LTLLibrary.asm
  - occorre che siano nella stessa cartella del file.asm
  - guarda dentro le librerie per la sintassi

NuSMV

- model checker
- verifica formale di sistemi dinamici
- sfrutta logica temporale
- verifica se la proprietà è vera
  - tasto T a sx della v verde → traduce la ASM per NuSMV
  - tasto TE → traduce ed esegue il model checker
    - in console esce se la proprietà è true o false
    - se false da la traccia di esecuzione fino al momento in cui trova che la proprietà è falsa

[5\\_asmeta/model\\_checker/coffeeVendingMachine.asm](#)

- se un prodotto è finito lo sarà anche in futuro



$$G(MILK = 0 \rightarrow G(MILK = 0))$$

LTLSPEC  $g(\text{available}(\text{MILK}) = 0 \text{ implies } g(\text{available}(\text{MILK}) = 0))$

- check se una proprietà è falsa

- per avere una traccia dimostrativa che una proprietà è vera la si trasforma in falsa, in modo da avere lo stesso la traccia esecutiva (witness della proprietà)

CTLSPEC  $\text{ef}(\text{available}(\text{MILK}) = 0 \text{ (vera)})$

$\rightarrow$  CTLSPEC  $\text{not } \text{ef}(\text{available}(\text{MILK}) = 0 \text{ (falsa, cioè vera al contrario ma così ho la traccia) }$

- forall

LTLSPEC  $(\text{forall } \$p \text{ in Product with } g(\text{available}(\$p) = 0 \text{ implies } g(\text{available}(\$p) = 0)))$

//  $\rightarrow$  in automatico la spezza e valuta per ogni Product

state explosion problem

- il problema che NuSMV risolve è esponenziale
- aumentando i casi di test inizia a fare fatica

5\_asmeta/model\_checker/ferryman.asm

- funzioni derivate
  - non fanno parte dello stato ma sono derivate da esso
  - utili per abbreviare le specifiche

derived allRight : Boolean

function allRight = (forall \$a in Actors with position(\$a) = RIGHT)

- until

// esiste un path in cui non c'è pericolo fino a che sono tutti a right

CTLSPEC  $e(\text{not pericolo}, \text{allRight})$

se qualcosa sembra non tornare controllare di non avere in console "flatten? false"

- altrimenti preferences  $\rightarrow$  asmetasmv  $\rightarrow$  togli flag da "do not flatten asm model before translation"

## 4.6 Automatic review of ASM by Meta-Property verification

### Concepts and principles

revisione delle ASM simile ad analisi statica

#### verifica

- check interno delle proprietà definite

#### validazione

- vedere i bisogni dell'utente
- model review
  - simile ad analisi statica
  - revisione del modello senza esecuzione

#### model review

- model inspection = model walk-through
- controllo qualità ancora prima di fare attività più dispendiose
- check proprietà di un buon modello = meta-properties
  - diverse dalle proprietà del **singolo modello** scritte con LTL/CTLSPEC
  - **sono generiche e devono valere per ogni modello**
  - corrispondono a degli attributi

#### Rule Firing Condition (RFC)

- per scrivere le meta-proprietà
- per ogni regola da le condizioni che la fanno scattare
- costruita visitando il modello
- RFC: Rules → Conditions

```
main rule R =
 if x > 0 then
 if y < 0 then
 x := 5
 endif
 endif
```

**Rule Firing Condition:**

$x > 0 \text{ and } y < 0$

### Meta-Properties of ASMs

meta review che verrà fatta in modo automatico

#### consistenza

- non devono esserci errori nella ASM che portano ad update inconsistenti

#### completezza

- la ASM esplicita tutti i possibili comportamenti

## minimalità

- la ASM non contiene elementi inutili

## Meta-properties definition

- Always(phi) : phi must be true in **any** reachable state
- Sometime(phi) : phi must be true in **a** reachable state

## MP1: No inconsistent update is ever performed

- update inconsistente = assegno alla stessa locazione di memoria due valori diversi

Example

```
main rule R =
par
 1:=1
 1:=2
endpar
```

Inconsistent update

For every rule R1 and R2

**MP1**

$R_1$ :

$$f(a_1) := t_1$$

$R_2$ :

$$f(a_2) := t_2$$

$$\text{Always} \left( \begin{array}{l} RFC(R_1) \wedge RFC(R_2) \\ \wedge a_1 = a_2 \\ \rightarrow t_1 = t_2 \end{array} \right)$$

- posso fare gli assegnamenti, ma il valore deve essere lo stesso
- [5\\_asmeta/updateinc2.asm](#)
  - prima di tutto occorre specificare quali meta-proprietà voglio controllare:  
preferences → asmeta → asmetaMA
  - check anche si "show NuSMV output"
  - ora posso chiamare il model advisor con il pulsante MA a sx della v verde

## MP2: Every conditional rule must be complete

- ogni condizione deve avere sempre un else, oppure deve essere sempre vera quando viene valutata (inutile)
- vale anche per gli switch case
- errore stilistico, di convenzione, non sintattico

## MP3: Every rule can eventually fire

- death code

```

main rule R =
 if x > 0 then
 if x < 0 then 1:=1
 endif
 endif

```

Never fires

MP3

For every rule R in the model:

*Sometime(RFC(R))*

```

if mon1 then
 if not mon1 then
 foo := 2
 endif
endif

```

MP4: No assignment is always trivial

- assegno ad una variabile un valore che ha già
- può capitare, ma vorrei che un assegnamento sia sempre triviale

```

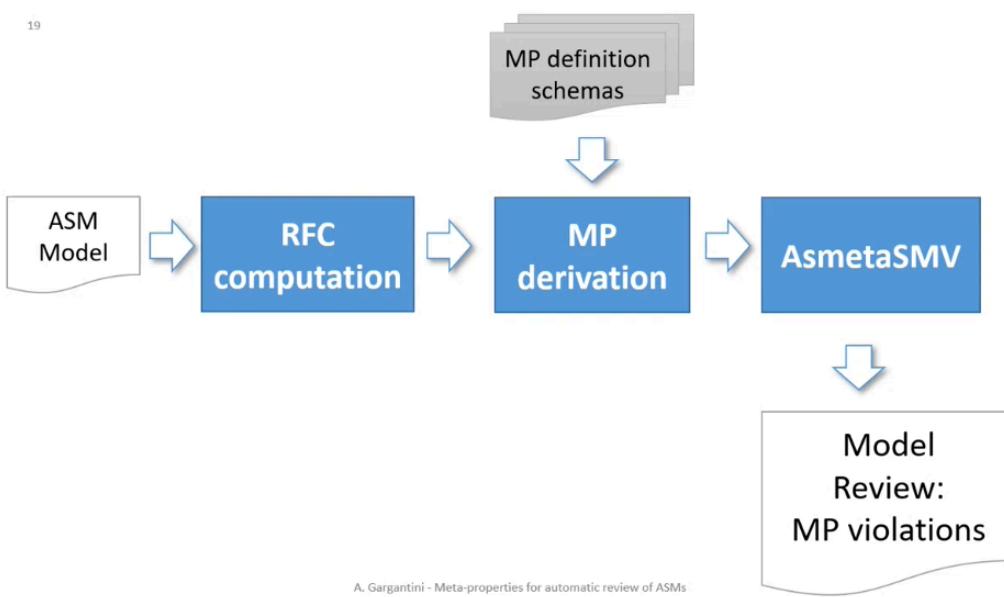
par
 if foo = 2 then
 foo := 2
 endif
endpar

```

altre MP..

MP verification

- traduce le meta/proprietà in proprietà da testare sul modello



A. Gargantini - Meta-properties for automatic review of ASMs

uso degli invarianti

- oltre a LTL e CTL asmeta supporta anche delle proprietà più semplici che sono simili agli invarianti di JML
- non supportano operatori temporali

- controllano solo lo stato corrente
- però vengono **controllati anche in simulazione** a differenza delle proprietà temporali
- posso dare un nome all'invariante
- il focus è sul controllo durante l'esecuzione piuttosto che sulla dimostrazione
- il check degli invarianti si può attivare/disattivare
  - preferences → asmeta → availa → simulator → check invariants
  - controllarli e fermarmi se violati
  - controllarli e continuare anche se violati
  - non controllarli

```
// la variabile controllata non è mai 7
invariant inv_1 over c_var : c_var ≠ 7
```

## 5. Model-based testing

### 5.0 Model Based Testing

generazione di casi di test a partire da modelli (asmeta o anche altri tipi)

#### tipologie

- generazione dati di input a partire da un modello del dominio
- derivare casi di test a partire da modelli di comportamento dell'ambiente
- generare degli oracoli a partire da un modello di comportamento
- script di test a partire da test astratti

#### Step del processo

##### 1. Modello astratto del SUT

- astratto = piccolo e semplice, ma con lo stesso funzionamento di quello reale
- prima di proseguire devo validare e verificare il modello (simulatore, animatore, model checking, verifica proprietà)

##### 2. Generazione test astratti a partire dal modello

- criteri per la copertura del modello
- fatto in automatico
- fornisce anche matrice di tracciabilità dei requisiti

##### 3. Concretizzazione dei testi astratti in test fattibili

- spesso fatto a mano, o con dei wrapper automatici
- [Avalla → JUnit](#)

##### 4. Esecuzione dei test e collezione dei verdetti

- online model-based testing
  - generazione ed esecuzione del test nello **stesso momento**
  - tramite i risultati possono live guidare la generazione di futuri casi di test
- offline model-based testing

- generazione ed esecuzione in **momenti diversi**
- posso generare una sola volta e poi fare varie volte tutti i casi di test in blocco

## 5. Analisi dei risultati

### Modellazione

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

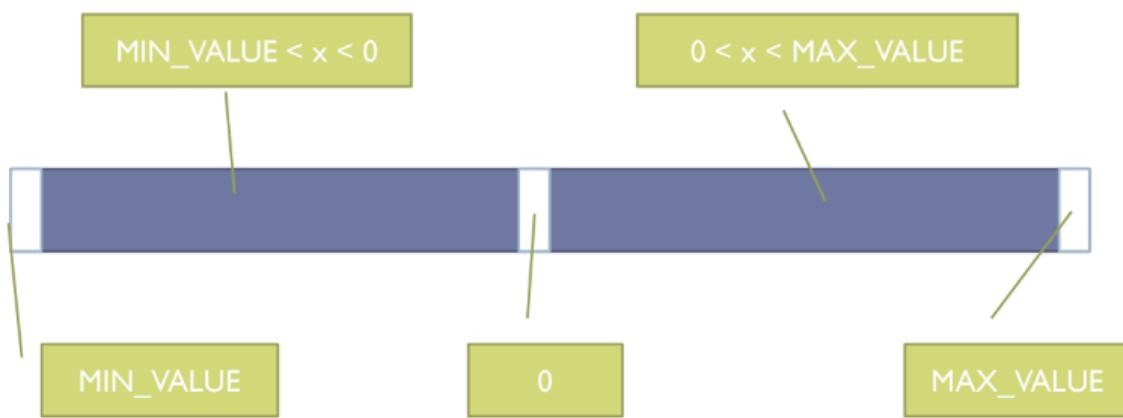
## 5.1 Input Testing

### Input Testing

- testing basato sulle interfacce, tipo black box
- guardo solo gli input che prende, senza nessuna info riguardante il sistema sottostante
- semplice da applicare
- si può usare a diversi livelli di testing (unit, integration, system, ..)

### Partition Testing

- si sceglie una partizione del dominio di input (ISP process) del programma, dividendo il dominio in regioni
- si sostituisce un dominio (anche infinito) in un numero finito di regioni
  - ogni regione corrisponde ad una caratteristica che l'input può avere **array ordinato, ordine inverso, a caso, ..**
- l'unione delle regioni deve dare tutto il dominio
- intersezioni nulle fra le partizioni



se dominio = prodotto cartesiano di altri domini

- partiziono come fosse un unico grande dominio

- partiziono ogni singolo dominio + prodotto cartesiano
- partiziono ogni singolo dominio + combinational testing

## Input Domain Modeling (IDM)

passi da fare

- individuare le funzioni testabili
- trovare i parametri
- modello il dominio di input
  - identificare dei blocchi = set di valori
  - fare in cui l'umano fa la differenza scegliendo tra i vari metodi esistenti
- ottengo vari parametri con dei possibili valori associati
  - ogni valore rappresenta una partizione del dominio
  - un blocco per ogni caratteristica  
*numeri negativi, 0, positivi*
  - poi sceglierò un valore per ogni blocco
- costruzione dei casi di test
  - serve usare un criterio di test per scegliere le combinazioni di valori
  - scelgo dei subset grazie a un criterio di coverage
  - scegliere dei valori appropriati di input per i test

approcci IDM

- Interface-based approach
  - l'interfaccia suggerisce quali sono i parametri di input e i loro valori
  - più semplice e banale, *ignora i possibili vincoli* dei parametri
  - parzialmente automatizzabile
- Functionality-based approach
  - identificare le funzionalità = input da testare
  - *considera anche la semantica del dominio, i possibili vincoli*
  - si possono anche introdurre *relazioni fra i parametri basate sui requisiti*

## Input Space Partition (ISP) coverage criteria

- All combination (ACoC)
  - *tutte le combinazioni dei blocchi, non di tutti i valori dei blocchi*
  - più forte ma onerosa
  - N-Wise con N = #(parametri)

$S_1 : \{ \text{POS}, \emptyset, \text{NEG} \}$

$S_2 : \{ \text{POS}, \emptyset, \text{NEG} \}$

$S_3 : \{ \text{POS}, \emptyset, \text{NEG} \}$

Ael.

$$3 \times 3 \times 3 = 27$$

- Each Choice (EC)

- per ogni caratteristica, usare almeno una volta un valore di ogni blocco
- più debole ma meno onerosa
- 1-Wise

$S_1 : \text{POS} \quad S_2 : \emptyset \quad S_3 : \text{NEG}$

Pos      Neg       $\emptyset \rightarrow 3\text{-1}\emptyset$

$\emptyset \quad \text{Pos} \quad \text{Neg} \quad \text{Pos}$

Neg       $\emptyset \quad \text{Pos} \quad -7\emptyset 15$

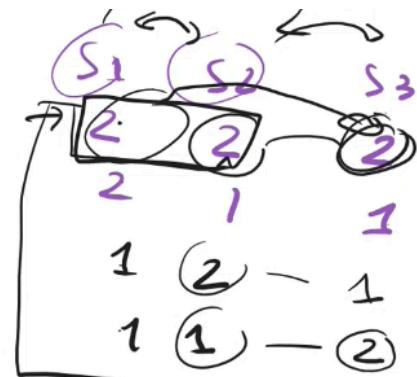
- Pair-Wise (PW)

- copre ogni possibile coppia
- un valore, per ogni blocco, di ogni caratteristica, viene accoppiato con un valore, di ogni blocco, di ogni altra caratteristica
- deve valere per qualsiasi coppia di parametri
- con un caso di test verifico più coppie allo stesso tempo, al posto di verificare solo la singola coppia

$S_1 : \{ 2, 1 \} \quad y$

$S_2 : \{ 2, 1 \} \quad y$

$S_3 : \{ 2, 1 \} \quad -y$



- T-Wise

- si sposta verso ACoC, considera set di T elementi  
 $T = 3 \rightarrow \text{triple}$

Test combinatoriali

semplificazioni:

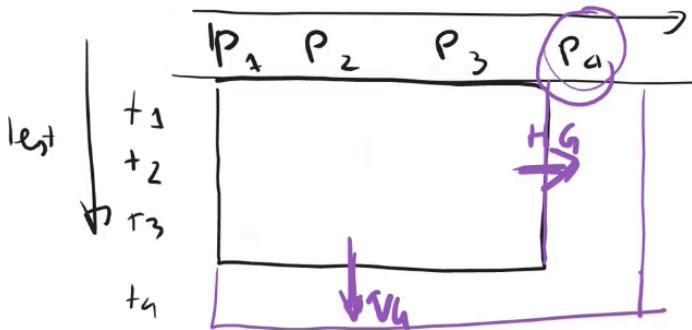
- ogni caratteristica è un parametro di input
- ogni blocco è un valore enumerativo → ho solo un valore per blocco

Metodi greedy per generare casi combinatoriali

- Basati sui parametri
  - considera un **parametro** alla volta
  - In Parameter Order (IPO)
- Basati sui testi
  - considera un **test** alla volta
  - AETG

In Parameter Order (IPO)

- costruisce un test set in modo incrementale
  - aggiunge un parametro alla volta
  - se serve poi aggiunge per ogni parametro anche più test



- parto dal pairwise scelgo due parametri (in genere prima i più grandi)
- poi aggiungo a dx gli altri parametri (a caso o cerco di minimizzare i pair inutili, **horizontal growth**)
- controllo se ho coperto tutti i pair tra tutti i parametri, e se mancano li aggiungo sotto (**vertical growth**)

## 5.2 Testing basato su specifiche

finora abbiamo ignorato il comportamento del sistema concentrandoci solo sull'input. Si possono però introdurre anche dei constraint per modellare il comportamento

ruoli della specifica dei requisiti

- definire i criteri con cui selezionare i test
- capire quando i test svolti hanno una copertura sufficiente
- specifica usata come oracolo (dice output atteso)
  - "applico gli input dei test alla specifica" e confronto con i risultati dei test sul sistema
  - conformance testing = la specifica, essendo oracolo, è corretta → se ho risultati diversi allora il sw è sbagliato

# Final State Machines (FSM)

notazione della specifica

- FSM ( $S, I, \delta$ )
- usiamo la macchina di Mealy =  $(S, I, O, \delta, \lambda)$
- produce un output per ciascuna transizione
  - $S$  = insieme finito di stati
  - $I$  = insieme finito di eventi di input
  - $O$  = insieme finito di eventi di output
  - $\delta$  = funzione di transizione ( $S \times I \rightarrow S$ ) cioè dato uno stato e un input so in che stato andare
  - $\lambda$  = funzione di output ( $S \times I \rightarrow O$ ) cioè per ogni stato e input mi da l'output di quella transizione
  - spesso è fissato anche lo stato iniziale  $s_0$

grafico FSM di Mealy

- archi etichettati con  $I/O$  = coppie evento\_Input e evento\_Output
  - evento output può anche essere un'azione della macchina  
*se collegata ad un codice potrebbe essere una chiamata di metodo*
- non posso avere due transizioni con lo **stesso input** che escono da uno stato → la FSM non sarebbe deterministica
- **posso** invece avere più transizioni con **stesso output** uscenti da uno stato

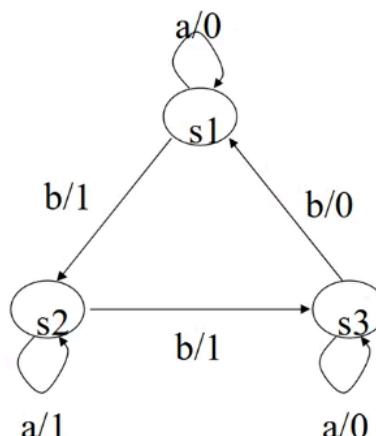
$$S = \{s_1, s_2, s_3\}$$

$$I = \{a, b\}$$

$$O = \{0, 1\}$$

$$\delta = \{ (s_1, a, s_1), (s_1, b, s_2), (s_2, a, s_2), (s_2, b, s_3), (s_3, a, s_3), (s_3, b, s_1) \}$$

$$\lambda = \{ (s_1, a, 0), (s_1, b, 1), (s_2, a, 1), (s_2, b, 1), (s_3, a, 0), (s_3, b, 0) \}$$



limiti FSM

- solo un numero finito di stati, nemmeno troppo grande
  - si usano allora di Statecharts di UML (Yakindu) per avere composizione sequenziale e parallela, oltre che modularità

Conformance testing con FSM

- specifica = FSM  $S$  di cui so tutto (stati, transizioni,  $\lambda$ )
- sistema implementato  $I$  = FSM black box di cui posso so solo l'output a fronte di certi input
- obiettivo = check se  $I$  è conforme con ad  $S$  applicando una sequenza di test e osservando l'output (confronto fra due FSM)

## Utilità testing con FSM

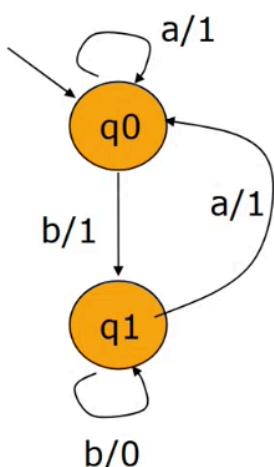
- semplice modellazione di sistemi con protocolli o sistemi di controllo embedded
- molte altre notazioni sono simili alle FSM
- spesso si può astrarre con FSM solo una parte del sistema per testarla a parte
- sotto alcune ipotesi garantiscono di avere un test ideale, cioè di trovare tutti i difetti → se passa non ci sono difetti

## assunzioni (forti)

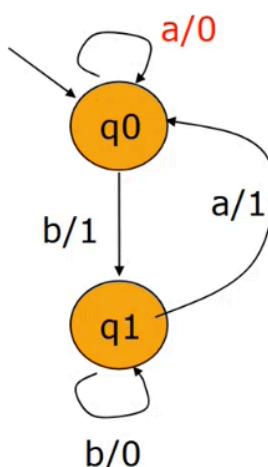
- $S$  ed  $I$  sono deterministiche e inizializzate
  - non ci sono transizioni doppie a partire da uno stesso stato una volta ricevuto lo stesso input
  - si trovano entrambe in uno stato noto
- $S$  ed  $I$  sono completamente specificate
  - per ogni input, per ogni stato, output e stato di arrivo sono definiti
- specifica  $S$  fortemente connessa
  - da qualsiasi stato si può arrivare in ogni altro stato
- $S$  è minimizzata (ridotta)
  - non esiste una specifica più piccola con lo stesso comportamento
- $I$  ha lo stesso alfabeto di  $S$
- $I$  non ha più stati di  $S$ 
  - assumo che eventuali difetti non aumentino il numero di stati

## modello degli errori

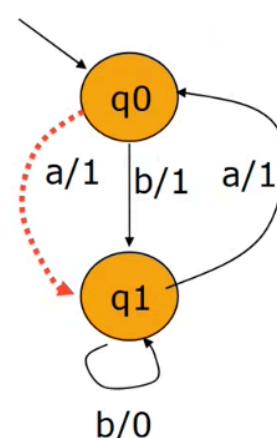
- modello ipotetico su quali tipi di difetti possono accadere in una  $I$ 
  - senza ci sono infinite  $I$  errate e il testing non può garantire nulla
- Errori considerati
  - Output error = vado nello stato giusto con l'input giusto, ma **sbaglio l'output**
  - Transfer error = vado nello **stato sbagliato** con l'input e output giusto



macchina corretta

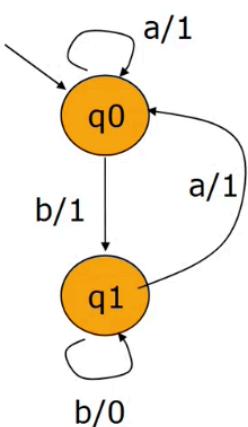


difetto di output

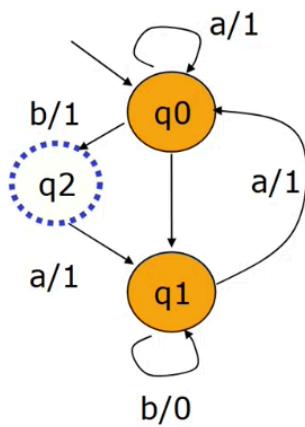


difetto di trasferimento

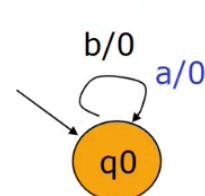
- Errori non considerati
  - stati extra/in meno



macchina corretta



extra state



missing state  
stato mancante

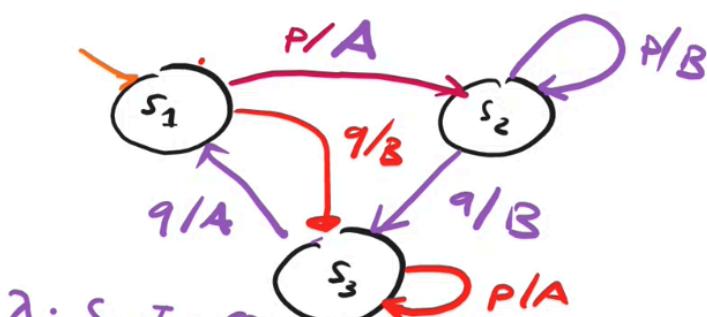
### Messaggi speciali

- Reset = mando a stato iniziale
  - come fosse un input extra che permette di ignorare l'output
- Status = stato attuale
  - non cambia lo stato, da solo info

test sequence = ts

- i metodi che vediamo generano un test set spesso formato da una sola sequenza di test, finita ma anche molto lunga
- test sequence = sequenza di input applicati alla FSM + output attesi

ts: P P q q q q P  
A B B . A B A A



- applicandola a S e poi ad I, confrontando gli output di testa la conformità di I ad S

### Metodi per generare i casi di test

- State cover method = copertura di tutti gli **stati**
- Transition tour (TT) = copertura di tutte le **transizioni**

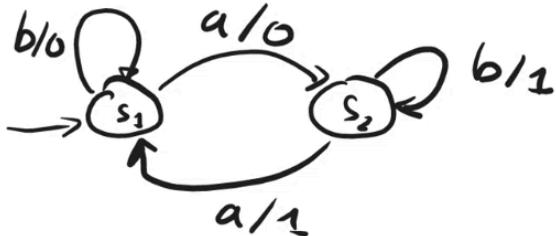
### Copertura stati e msg di status

- Metodo 1: (stati + status message)

- Copertura degli stati e lettura dello status message per controllare che il comportamento sia corretto
- Ho una ts e una sequenza formata da output (delle transizioni) e degli stati (con status message)

$$I = \{a, b\}$$

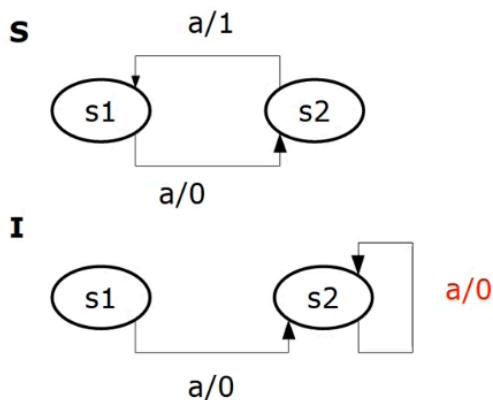
$$O = \{s_1, s_2\}$$



Metodo 1

| ts: | Status | a | Status |
|-----|--------|---|--------|
|     | s1     | ↓ | s2     |

- può trovare errori di output/transfer, ma non lo garantisce
  - se ho una transizione sbagliata e non la copro non me ne accorgo



### Esempio

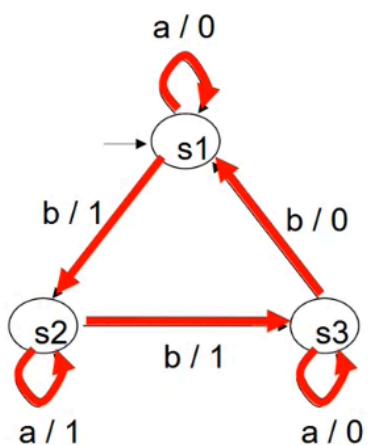
la test sequence:

*status a status*

copre tutti gli stati di S,  
applicata a I produce lo stesso output ( $s_1, 0, s_2$ )

- Metodo 2: (transition tour)

- sequenza di input che, partendo da  $s_0$ , passa in tutte le transizioni almeno una volta, e poi torna in  $s_0$



### Esempio:

la sequenza *ababab*

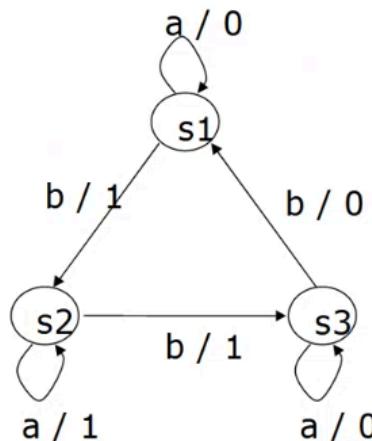
- il tour più veloce è quello Euleriano

- `ts = TT`
- non usa nessuno status message

**S**

Input sequence: *ababab*

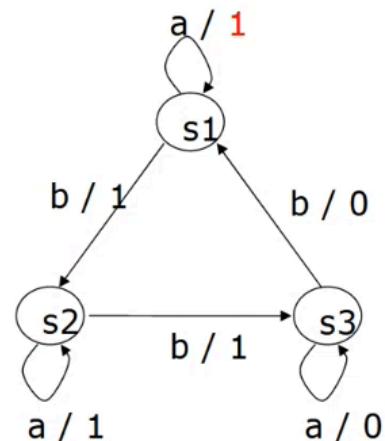
Expected output sequence: *011100*



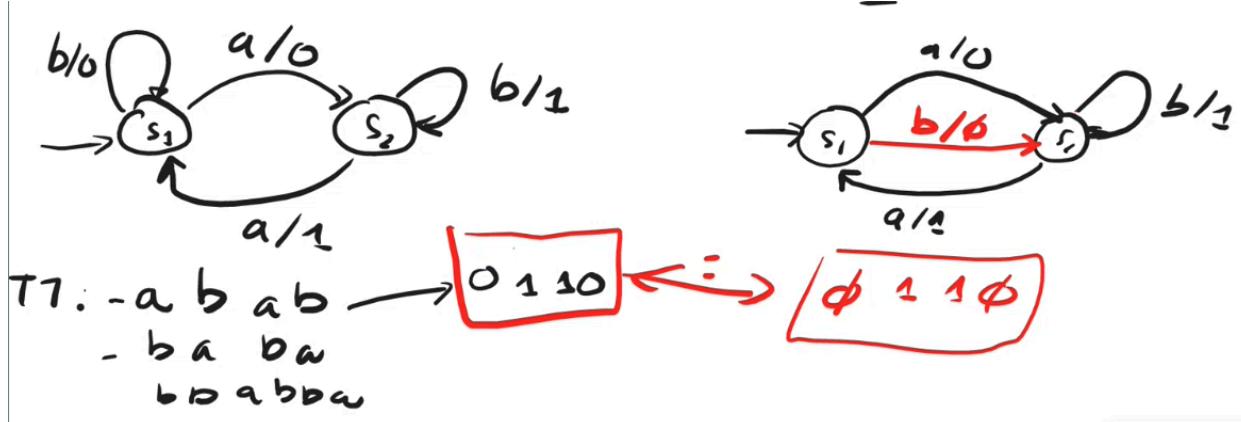
**I (con output fault)**

Input sequence: *ababab*

Observed output sequence: *111100*



- garantisce scoperta di difetti di output, ma non di transfer



- Metodo 3: (TT + status message)
  - status message dopo ogni input per check stato (+ sequenza output)
  - garantisce la scoperta di tutti i difetti

## 5.4 Making tests executable

Concretizzazione

- come rendere i test eseguibili sul SUT i test generati da un modello astratto
- tradurre i test astratti nel linguaggio del sw effettivo

best practices per il modello astratto

- modellare solo ciò che serve
- usare solo input e output necessari ai fini di test
- semplificare i dati complessi
- suppongo che il SUT sia già inizializzato

online testing

- applicare immediatamente i casi di test al sistema che funziona, in tempo reale
- richiede di far funzionare **insieme** sia il generatore di casi di test che il SW

### offline testing

- generazione ed esecuzione dei casi di test sono separati, completamente **indipendenti**

Tecniche per colmare il gap semantico tra test cases e SUT

- adaptation = **non traduco** ma uso un adapter/wrapper (online testing)
- transformation = traduco **ogni test** in script da eseguire sul sistema (offline testing)
- mixed = traduco **un po'** e un po' adatto

## Tools

### Asmeta + ATGT

- generatore automatico di scenari in Avalla per Asmeta
- scenari poi da tradurre a mano in java per fare dei casi di test in JUnit
- file.asm → run as → ATGT test generator
  - si crea nel progetto una directory (se serve fai refresh) in cui ci sono degli scenari di test Avalla
  - genera a partire da criteri di coverage prestabiliti
- processo:
  - Specifico → asmeta
  - Validazione → model checker, avalla
  - Generazione → atgt o a mano

### GraphWalker

- macchina a stati con label su transizioni
- due tipi di elementi:
  - stati = vertici → = verification = assertion
  - transizioni = edge → = azioni
- cinema assurdo per runnare un server in localhost
- ogni volta che trova uno stato o una transizione chiama il rispettivo metodo java

### Yakindu

- modello come macchine di stato (statecharts) tipo UML
- definisco interfacce dei moduli per cosa posso modificare, o quali eventi posso fare
- lo posso validare lanciando la simulazione
  - run as → statechart simulation
  - click sugli eventi per farli verificare e vedere come evolve la macchina
- si possono scrivere a mano degli scenari come Avalla

- run as → SCT Unit (output come JUnit)
- posso avere anche diagramma di copertura nel tab coverage (se serve fai windows → perspective → reset perspective)
- posso convertire da Yakindu a JUnit

#### ModelJUnit

- modello direttamente nel codice java
- metodi delle transizioni annotati con @Action

#### CTWedge

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

#### Input Domain Modeling

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

## Exe Avalla e ASM

### Giugno2021

#### *Esercizio Tema d'esame Giugno 2021*

Un forno può essere in standby (chiuso e spento), con la porta aperta (ma spento) oppure acceso (la porta deve essere chiusa). Modella questi stati e le opportune azioni con ASMETA. Prova le seguenti proprietà con LTL o CTL a tua scelta:

- Quando è acceso la porta è sempre chiusa.
- Prima o poi si può accendere in qualsiasi momento in futuro.
- La porta può essere aperta dopo che viene acceso.
- Quando è acceso, la porta rimane chiusa fino quando rimane acceso (usa Until).

C'è qualche proprietà che invece è giustamente falsa e il cui controesempio ti aiuta a capire come funziona il forno?

Scrivi poi i seguenti scenari, controllando che il valore delle funzioni sia quello atteso:

- Il forno inizialmente è spento. L'utente apre la porta e poi la chiude. Il forno viene acceso (controlla che la porta sia chiusa quando viene acceso)
- Il forno inizialmente è spento. Il forno viene acceso e l'utente prova ad aprire la porta. Controllare che la porta non sia aperta.

- new project → general → project
- new other → asmeta → asmeta wizard → accetta di convertire in Xtest project
- copiare nel progetto le librerie necessarie per Asmeta

funzioni controllate e monitorate

- monitorate = input del sistema
  - la ASM legge il valore ma non controlla il valore che assumono
  - utente o misura ricevuta
- controllate
  - ne leggo il valore
  - però sono gestite dalla ASM

variabili per stato interno del forno → controllate

- stato forno
- stato porta forno

variabili per operare sullo stato di forno e porta → monitorate

- tasto per aprire la porta
- tasto accendi forno
- tasto spegni forno

main rule

- indica come il sistema evolve (main method)
- check stato attuale e poi capisco come può evolvere
- può avere delle rule di appoggio che richiama (generic method)

LTL e CTL

- se deve verificarsi = a
- se potrebbe verificarsi = e
- sempre valido = ag/eg

Avalla

- new file → nome\_scenario.avalla
- inizio file con scenario nome\_scenario  
**scenario Scenario1**
- load macchina.asm per indicare su quale ASM eseguire gli scenari  
**load Forno.asm**
- check condizioni stato iniziale  
**check statoForno = SPENTO;**
- conviene sempre fare uno step tra le sezioni con check e set  
**step**
- set delle variabili, per ordine prima quelle interessate, poi per far andare tutto metto anche quelle che non mi interessano ma a false  
**check statoForno = SPENTO;**  
**set accendi := false;**  
**set spegni := false;**
- click su V nella barra strumenti per eseguire il validatore, guardare poi in console se tutto ha successo

# Set\_21

## Esercizio Tema d'esame Settembre 2021

Scrivi un modello ASM per un semaforo che però diventa verde solo su richiesta (usa ad esempio con una monitorata boolean). Il semaforo fa il ciclo regolare: ROSSO -> VERDE -> GIALLO ->ROSSO.

Prova ad animare il sistema e fai uno screenshot del comportamento.

Prova le seguenti proprietà sia come LTL che come CTL:

- Non può mai passare a VERDE direttamente da ROSSO
- Quando è ROSSO rimarrà sempre ROSSO a meno che ci sia una richiesta
- Se c'è una richiesta allora prima o poi diventa VERDE.
- In qualsiasi istante, prima o poi potrebbe diventare VERDE.

Se qualcuna è falsa spiega perché. Scrivi e spiega anche una tua proprietà vera e una falsa il cui controesempio ti aiuta a capire come funziona il semaforo.

Scrivi poi i seguenti scenari, controllando che il valore delle funzioni sia quello atteso:

- Il semaforo è rosso, per uno stato rimane ancora rosso e non si ricevono richieste di passaggio. Arriva poi una richiesta di passaggio ed il semaforo passa sul verde. Lo stato successivo scatta il timer e si passa sul giallo.
- Il semaforo è rosso, poi si riceve una richiesta di passaggio e passa sul verde. Forzare manualmente lo stato del semaforo di nuovo sul rosso e controllare che sia avvenuto il passaggio corretto.

```
// se rosso resta sempre rosso, a meno che ci sia una richiesta
CTLSPEC ag(statoSemaforo = ROSSO implies aw(statoSemaforo = ROSSO, richiesta =
true))
// aw() perché la richiesta potrebbe non avvenire mai (weak until)
// se invece prima o poi deve accadere per forza allora a()
```

scrittura automatica scenario Avalla

- simulare lo scenario nel simulatore
- click su "export to avalla"
- in console c'è il print dello scenario → copia
- incolla in un nuovo file.avalla
- cambia nome scenario in base a nome del file.avalla

forzare il cambiamento di una variabile controllata

```
check statoSemaforo = VERDE;
exec statoSemaforo = ROSSO; // forzo cambiamento variabile controllata
set richiesta := false;
set timerPassed := false;
```