Descrivere tre possibili tipologie di attacchi

Gli attacchi si differenziano in tre livelli, il livello di progettazione, implementazione o operazione.

Livello di **Progettazione**

- **Man-in-the-middle attack**: accade quando un attacker intercetta una trasmissione di rete tra due host, e si spaccia per una delle due parti coinvolte nella transazione, possibilmente aggiungendo ulteriori direttive nel dialogo. Difesa: usare intensivamente tecniche di encryption (in particolare autenticazione crittografica forte), usare session checksum e shared secrets come cookies (usare ssh invece ch **Goodenough** e telnet, criptare i file usando utilities come PGP o Entrust).
- **Race condition attack**: certe operazioni comuni ad applicazioni software sono, dal punto di vista del computer, costituite da passi discreti (anche se noi le riteniamo atomiche). Il tempo per completare questi passi apre una finestra durante cui un utente malizioso può compromettere la sicurezza. Un esempio è l'operazione: 1.controllare se un file contiene comandi sicuri da eseguire; 2.eseguirlo. Un utente malizioso potrebbe sostituire il file tra i passi 1 e 2. Difesa: Comprendere la differenza tra operazioni atomiche (indivisibili) e non-atomiche, ed evitare le ultime a meno che non si sia sicuri che non abbiano delle implicazioni non sicure. Se non si è sicuri che un'operazione sia atomica, assumere che non lo sia, ossia che il sistema operativo possa eseguirla in due o più passi interrompibili.
- **Replay attack**: se un attacker riesce a catturare o ottenere il record di una intera transazione tra un programma client ed uno server, ha la possibilità di "riprodurre" parte della conversazione a scopo sovversivo. Difesa: come per il man-in-the-middle attack. In più, introdurre in ogni dialogo alcuni elementi (come un sequence identifier) che differisca da sessione in sessione in modo da far fallire il byte-for-byte replay.
- **Sniffer attack**: uno sniffer è un programma che silenziosamente memorizza tutto il traffico spedito su una rete locale. Gli sniffer sono talvolta tool legittimi con fini diagognistici, ma sono anche utili agli attaccanti per memorizzare usernames e passwords trasmesse in chiaro. Difesa: come sistemista, attente configurazione della rete, e uso di "switched" network routers; come programmatore, massimizzare l'uso della crittografia.
- **Session hijacking attack**: sfruttando le debolezze del protocollo TCP/IP, un attaccante può riuscire a prendere il controllo (o hijacking) di una connessione già stabilita. Esistono molti tool che sono stati scritti e distribuiti su Internet per implementare questo tipo di attacco. Difesa: E' un attacco di rete da cui un'applicazione software riesce difficilmente a difendersi. La crittografia è di aiuto, e se un attento logging fornisce abbastanza informazioni sulla sessione, procedure operazionali possono essere in grado di rilevare un attacco di hijacking dopo che si è verificato.
- **Session killing attack**: sezioni legittime di TCP/IP possono terminare se una delle due parti che comunicano invia il packet TCP reset. Un attaccante potrebbe essere in grado di forgiare gli indirizzi di una delle due parti e resettare prematuramente la connessione. Un attacco di questo tipo può essere usato per distruggere una comunicazione, o per prendere il controllo di una parte della trasmissione. Difesa: E' difficile a livello di applicazione potersi difendere da questi tipi di attacchi alla rete. Tuttavia, l'applicazione potrebbe essere in grado di controbattere gli effetti dell'attacco ristabilendo la connessione interrotta.

Livello di **Implementazione**

- **Buffer overflow attack:** molti linguaggi di programmazione consentono di allocare un buffer di lunghezza fissa per una stringa di caratteri ricevuta in input, ad es. come argomento a linea di comando. Una condizione di buffer overflow si verifica quando l'applicazione non effettua un adeguato bounds checking alla stringa ed accetta più caratteri di quanto sia

possibile memorizzare nel buffer. In molti casi, un attaccante può causare un overflow del buffer e forzare il programma ad eseguire comandi o azioni non autorizzate. Difesa: Usare un linguaggio di codifica (come Java) che esclude overflows per design; altrimenti evitare di leggere stringhe di testo di lunghezza indeterminata in buffer di lunghezza fissa a meno che non si possa leggere in modo sicuro sottostringhe di lunghezza specificata che possono essere contenute nel buffer.

- **Back door attack:** molti sistemi di applicazioni vengono compromessi da attacchi introdotti durante la scrittura del codice. Può accadere che un programmatore scriva all'interno dell'applicazione del codice speciale che consenta in seguito di bypassare il controllo d'accesso (back door). Difesa: Adottare procedure di garanzia delle qualità che controllano la presenza di back doors sull'intero codice.
- **Parsing error attack:** Le applicazioni spesso accettano input da utenti remoti senza controllare opportunamente i dati in input. Il controllo (o parsing) dei dati in input ai fini della sicurezza è importante per bloccare gli attacchi. Un famoso esempio di errore di parsing riguardava server web che non controllavano le richieste con all'interno sequenze "../", e ciò consentiva all'attaccante di raggiungere parte delle directory del filesystem il cui accesso dovrebbe essere proibito. Difesa: Si raccomanda il riuso di codice esistente, scritto da una specialista, che è stato opportunamente controllato, testato e mantenuto. Quando si scrive del codice, bisogna tener conto che è molto più sicuro controllare che ogni carattere di input compaia in una lista di caratteri "safe", piuttosto che comparare ciascun carattere con quelli di una lista di caratteri "pericolosi".

Livello di **Operazione**

- **Denial-of-service**: si verifica quando un utente legittimo si vede "negato un servizio". Un sistema di applicazioni, un host, o una rete possono essere resi inutilizzabili per via di una cascata di richieste di servizio, o persino di un flusso di input ad alta-frequenza. In un attacco negazione-servizio su ampia-scala, l'attaccante può usare hosts su Internet precedentemente compromessi come piattaforme di passaggio per l'assalto. Difesa: Progettare il software e in particolare l'allocazione delle risorse in modo tale che l'applicazione faccia moderata richiesta di risorse del sistema (come spazio su disco o numero di file aperti). Quando si progettano grandi sistemi, prevedere il monitoraggio dell'utilizzo delle risorse, e un modo che consenta al sistema il trabocco di caricamento eccessivo (cioè il software non dovrebbe lamentarsi o morire nel caso in cui si esauriscano le risorse).
- **Default accounts attack**: molti sistemi operativi e programmi applicativi sono configurati, per default, con username e password "standard" (es.guest/guest). Username e password di defautl permettono facile accesso a potenziali attaccanti che conoscono o riescono ad indovinare queste informazioni. Difesa: Rimuovere gli accounts di default; controllare nuovamente dopo aver installato nuovo software o nuove versioni di software esistente (infatti scripts di installazione possono talvolta reinstallare questi tipi di accounts di default).
- **Password cracking attack**: gli attaccanti ripetutatmente indovinano password scelte male, per mezzo di speciali programmi di cracking: questi programmi usano speciali algoritmi e dizionari di parole e frasi d'uso comune per indovinare centinaia di migliaia di parole; password deboli, come nomi comuni, date di nascita, o contenenti parole "secret" o "system", possono essere indovinate da questi programmi nella frazione di un secondo. Difesa: Come utente, scegliere password intelligenti. Come programmatore, usare tool per richiedere passwords robuste, cioè facili da ricordare e difficili da indovinare. Le norme per una buona scelta di password sono: offuscare le parole in un linguaggio sconosciuto; usare combinazioni di caratteri formate da lettere iniziali (o finali) di ciascuna parola di una frase che si ricordi; includere simboli di punteggiatura o numeri.

Bisognerebbe inoltre evitare il riuso di password già in fase di design, usando metodi alternativi di autenticazione, come dispositivi biometrici e smart cards. E comunque la lunghezza di una password è sempre più sicura della combinazione di caratteri complessi.

Descrivere i principi su cui si basa il *desing by contract* e portare un esempio di contratto su un metodo Java e JML.

L'idea del desing by contract è quella di realizzare un'interfaccia di un modulo che definisce un contratto.
Un contratto è l'accordo tra il cliente e il fornitore che: lega le due parti è esplicito cioè scritto, e specifica gli obiettivi e i benefici delle parti normalmente mappa gli obblighi di una parte come benefici dell'altra non contiene clausole nascoste.
In un contratto si può definire cosa ogni metodo richiede: gli obblighi del cliente e precondizioni gli obblighi del fornitore e postcondizioni.
Possiamo inoltre specificare, in un contratto, una proprietà detta invariante cioè che vale sempre per tutte le istanze, che non varia a seguito di un comando.
I contratti possono essere inseriti anche prima di una vera implementazione del metodo.
Dalle definizioni dei contratti si può estrarre in modo automatico le precondizioni, le postcondizioni e le invariatni che documentano cosa fa la classe generando così una documentazione automatica sempre aggiornata.
Infine, la prova di correttezza del sofware, dimostra che il contratto è stato rispettato già dall'implementazione, le eccezioni si sollevano solo quando il contratto è violato.

In JML le precondizioni si esprimono con la clausola //@ requires mentre le postcondizioni con la clausola //@ ensures.
E' possibile inoltre dichiarare una variabile che vale per tutte le istanze con la clausola //@ invariant.
E' possibile accedere al valore dei parametri prima dell'invocazione del metodo con la clausola \old(PARAM) oppure accedere al valore restituito dal metodo con la clausola \result.

```
public class TriangoloNonDegenere {
	/*@ spec_public @*/ int a;
	/*@ spec_public @*/ int b;
	/*@ spec_public @*/ int c;

	public TriangoloNonDegenere() {
		// TODO Auto-generated constructor stub
	}

	public TriangoloNonDegenere(int l1, int l2, int l3) {
		a=l1;
		b=l2;
		c=l3;
	}
	// Precondizioni
	//@ requires ((a!=0)&&(b!=0)&&(c!=0)&&(a!=(b+c))&&(b!=(a+c))&&(c!=(a+b)));

	// Postcondizioni Area >=0
	//@ ensures (\result >= 0);
	public double calcolaArea(){
		double sp;
		if ((a==0)||(b==0)||(c==0)||(a==(b+c))||(b==(a+c))||(c==(a+b))){
			return 0;
		}else{
			sp=(a+b+c)/2;
			return Math.sqrt(sp*(sp-a)*(sp-b)*(sp-c));
		}
	}

	public static void main(String[] args) {
```

```
            TriangoloNonDegenere t = new TriangoloNonDegenere(3,4,5);
            System.out.println(t.calcolaArea());
        }

}
```

Dare la definizione di test set, affidabile, valido e ideale.

- Affidabile: se per ogni coppia di test set T1 e T2 adeguati secondo il criterio C, se T1 individua un malfunzionamento, allora anche T2 e viceversa
- Valido: un criterio C è valido se, qualora il programma P non sia corretto, esiste almeno un test set T che soddisfa C che è in grado di individuare il difetto
- Ideale: test set T che è selezionato con un criterio affidabile e valido

Quali sono i principi guida di un'architettura sicura che riguardano malfunzionamenti ed errori del sistema? Si descrivano tre di questi principi.

Possibili errori che riguardano malfunzionamenti ed errori sono:

1) **Principio dell'error handling**: cioè molti sistemi vengono scartati a causa di un improprio trattamento di errori inaspettati quindi è necessario, per realizzare un buon software sicuro, pianificare correttamente l'error handling.
2) **Principio di graceful degradation**: cioè nel caso in cui nel sistema si verificasse un errore o un malfunzionamento, esso non si deve fermare ma deve garantire l'esecuzione anche in maniera minima o ridotta.
3) **Principio di fault safely**: cioè in caso di fallimento il sistema deve terminare in una configurazione sicura.
4) **Principio di auditability:** cioè deve essere possibile ricostruire la sequenza di eventi che hanno portato ad azioni chiave o critiche che per esempio hanno causato un errore, questo principio richiede la creazione di audit log.
5) **Principio di resource-consumption limitation**: dice che si devono usare le funzionalità che il SO fornisce per limitare il consumo di risorse del sistema. La limitazione del consumo di risorse deve essere combinata con un significativo ripristino da errore(error recovery).

Dare la definizione di macchina a stati finiti estesa, e portare un semplice esempio.

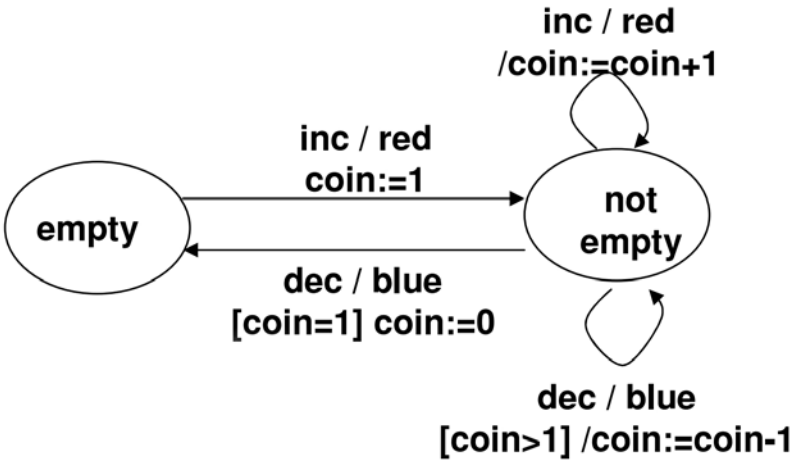
Le EFSM (Extended Finite State Machines) estendono le FSM con il concetto di variabile.
Una EFSM è una tupla (S, I, O, V, T) dove:

- S: insieme finito di stati
- I: insieme finito di eventi di input
- O: insieme finito di eventi di output
- V: insieme finito di variabili
- T: insieme finito di transizioni. Una transizione è una tupla (s, i, o, g, a, s'):
  - s: stato sorgente
  - i: evento di input
  - o: evento di output
  - g: predicato sulle variabili in V, detto Guardia
  - a: assegnamento ad una variabile V, detto Azione
  - s': stato target

Esempio: salvadanaio elettronico:

- inserendo monete (inc) il valore di coin viene incrementato
- emettendo monete (dec) il valore di coin viene decrementato

- la luce diventa red quando si inseriscono monete, diventa blue quando si richiedono monete
- la macchina non restituisce monete quando è vuota
    - S={empty, not-empty}
    - I= {inc, dec}
    - O={red, blue}
    - V={coin}
    - T={(empty, inc, red, coin:=1, not-empty),(not-empty, dec, blue, coin=1, coin:=0, empty)…}



Definire cosa si intende per *program-based testing* e *specification-based testing*. Indicare quali sono i criteri di copertura per il program-based testing.

Il program-based testing o white box testing, assume che il programma sorgente sia disponibile. Il white box testing esamina inizialmente la struttura del programma, quali i punti critici, le decisioni importanti, ecc. Successivamente trova i casi di test, ovvero gli Input, che soddisfano un certo criterio di copertura, applica poi gli input uno ad uno e osserva il comportamento del programma (osserva gli output). Infine esamina che non si verifichino degli errori e che gli output siano quelli attesi.

Il specification-based testing o black box testing, assume che non si guardi il programma sorgente ma solo quello che dovrebbe fare.

Descrivere il ciclo di vulnerabilità del software

Il ciclo di vulnerabilità è una sequenza di eventi che si susseguono molto comunemente nel mondo della sicurezza, va dalla determinazione all'eliminazione di una vulnerabilità del Software.

È un ciclo ortagonale al ciclo di vita del Software, ed è così suddiviso:

- Viene scoperta una nuova vulnerabilità in un pezzo di Software;
- I cattivi analizzando l'informazione e cercano di lanciare attacchi contro il sistema o la rete;
- I buoni cercano una soluzione al problema:
    - analizzando la vulnerabilità, sviluppano una soluzione e la testano in un ambiente controllato, distribuiscono la soluzione
- Se la vulnerabilità è seria, o gli attacchi sono gravi, anche i media ne danno pubblica informazione con catastrofiche conseguenze;

- La patch distribuita, viene installata dagli addetti;
- Tecnici della sicurezza analizzando frammenti di codice per scoprire vulnerabilità simili.

Descrivere i meccanismi di sicurezza in Java2

Java2 introduce, rispetto a alla versione Java1 e 1.1, il controllo degli accessi a livello più fine basato sulle security policy e permessi, rende quindi l'accesso più flessibile superando il limite del modello “tutto o niente” di JDK 1.
L’applicazione delle security policy è fatta dal SecurityManager tramite un metodo generale chiamato checkPermission (e da AccessController).

L'espressione dei diritti di una applicazione avviene usando istanze della classe Permission e delle sue sottoclassi.
La classe CodeSource definisce da dove proviene il codice invece, l'URL da cui proviene la classe, è un array di certificati ed è immutabile.
La classe Permission rappresenta con la sua gerarchia i permessi:
- FilePermission permessi di scrittura/lettura di file
- SocketPermission per connettersi ai socket

Un ProtectionDomain:
- viene creato da un CodeSource e una PermissionCollection (insieme di permessi) definisce i permessi attribuiti alla classe.
- ogni classe appartiene ad una istanza di ProtectionDomain, impostata al momento della creazione e che non può essere cambiata.
- L'accesso a questi oggetti è ristretto (come il caffè).

Policy Class: è l'interfaccia per definire politiche di sicurezza particolari:
- dato un CodeSource, restituisce una PermissionCollection
- viene usata durante il caricamento di una classe per impostare il ProtectionDomain, in particolare l'insieme di Permission
- viene caricata dal ClassLoader che chiede alla Policy Class quali sono i ProtectionDomain della classe

Caricamento di una classe:
Quando una classe C1 vuole caricare una classe C2 che non è stata ancora caricata:
1) Il classloader di C1 cerca il C2.class, lo carica e chiama il byte code verifier.
2) Il codesource di C2 viene determinato
3) L’oggetto della policy del ClassLoader restituisce le Permissions dato il codesource di C2.
4) Se un ProtectionDomain già esistente ha lo stesso CodeSource e Permission, viene riusato, altrimenti viene creato un nuovo ProtectionDomain; C2 viene assegnato al suo ProtectionDomain.
5) La classe può essere istanziata ed i suoi metodi eseguiti ma i controlli runtime continuano.

Controlli Runtime
Se una classe C invoca un metodo M il quale richiede un permesso P, la JVM chiama il checkPermission(P) del SecurityManager corrente, questo attiva un AccessController che controlla il chiamante di M cioè C, se il ProtectionDomain di C contiente il permesso P non ci sono problemi altrimenti, in caso contrario, solleva un’eccezione.

Confrontare i linguaggi C e Java in termini di sicurezza.

I principali problemi di sicurezza del linguaggio C sono:

- Deferenziazione del null: Ovvero l'accesso ad una cella nulla tramite un puntatore;
- Type cast non controllato: Conversione non controllata di un tipo;
- Pointer Arithmetic: Gestione della virgola mobile;
- Accesso alla memoria non valido: Violazione spaziale (out of bound) o temporale (dangling pointers).

Nel linguaggio Java vengono introdotti diverse novità a favore della sicurezza del linguaggio:
- Sicurezza dei tipi;
- Verifica del bytecode, ovvero vine accettato solo codice Java legittimo
- Controllo runtime delle classi create
- Accesso mediato alle risorse

Infine la sicurezza viene gestita da due classi, la SecurityManager che definisce i metodi di controllo dei permessi di sicurezza chiamati dal sistema e dalla ClassLoader che carica effettivamente le classi.

Descrivere la differenza tra validazione e verifica del SW

La **Validazione** (building the right system?) controlla che il sw sia corretto accertando che un software soddisfi i requisiti dell'utente intesi come i suoi bisogni e si confrontano i requisiti informali espressi dall'utente.
La **Verifica (**building the system right?): controlla che l'implementazione corrisponda alla sua specifica.

In altre parole, la **Validazione** riguarda la convalida che il sistema soddisfa le reali esigenze del cliente, mentre la verifica riguarda se il sistema è ben progettato, senza errori, e così via. La **Verifica** aiuterà a determinare se il software è di alta qualità, ma non assicura che il sistema è utile.

Dare la definizione di Stato globale, transizione globale e grafo di raggiungibilità di una FSM- Portare un semplice esempio.

- stato globale: Per una EFSM (S, I, O, V, T) una coppia (s, σ) è detta stato globale se s è uno stato e σ è una valutazione su V; esempio: (empty, coin=0), (not-empty, coin=1)
- Transizione globale: per una EFSM (S, I, O, V, T) una tupla ((s, σ), i, o, (s', σ')) è detta transizione globale se esiste una transizione (s, i, o, g, a, s') tale che σ soddisfa la guardia g e σ'(v)= σ(exp) dove l'azione a è v:=exp; esempio: ((empty, coin=0), inc, red, (not-empty, coin=1))
- Grafo di raggiungibilità: Il grafo di raggiungibilità di una EFSM (S, I, O, V, T) è un grafo direzionato in cui: i nodi sono gli stati globali e gli archi sono le transizioni globali. Se le variabili hanno un range finito, il grafo di raggiungibilità di una EFSM è una FSM

Esempio:
SG=(empty, coin=0), S'G(notempty, coin=1), S"G(notemtpy, coin=2)
TG=((empty,coin=0), inc,red,(notempty, coin=1)), ((empty,coin=1), inc,red,(notempty, coin=2))

Definire le fasi in cui è organizzato un processo per lo sviluppo del SW

- **Produttività:** misura l'efficienza del processo di produzione del SW in termini di velocità di consegna del SW
- **Tempestività:** misura la capacità del processo di produzione del SW di valutare e rispettare i tempi di consegna del prdotto.

- **Trasparenza:** misura la capacità del processo di produzione del SW di capire il suo stato attuale e tutti i suoi passi.

Nello sviluppo di software sicuro, motivare il perché la sicurezza è definita come proprietà context-sensitive.

La valutazione della sicurezza è un'attività molto più complessa dell'applicare un target di sicurezza ad un target di valutazione. I problemi di sicurezza sono context-sensitive, e quindi è difficile valutarli in base a criteri comuni. Inoltre, come molti standard, definiscono il "cosa" e non il "come", mentre nel campo della sicurezza è fondamentale sapere come affrontare problemi di sicurezza e gestire i rischi.

Classificare l'attività di testing rispetto ai requisiti, all'architettura ed al codice

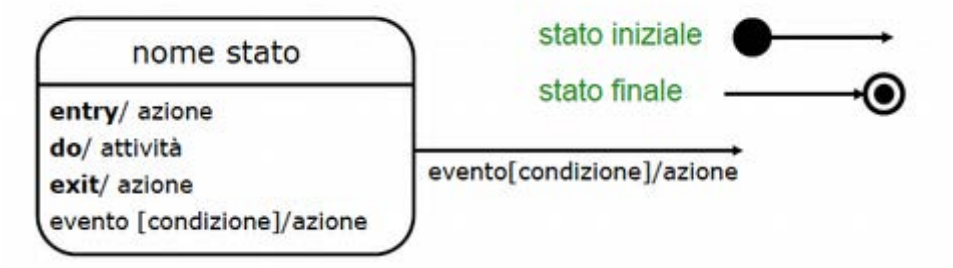
I costo del testing sono normalmente molto maggiori rispetto ai requisiti, al progetto e all'implementazione.

Descrivere il meccanismo degli state pattern per la codifica di una FSM.

Il State Pattern permette di cambiare il comportamento di un oggetto quando cambia il suo stato interno. Si usa quando il suo comportamento dipende dal suo stato. Si definisce una classe per ogni stato con un'interfaccia comune che reagisce ad ogni possibile input. Gli oggetti di stato possono richiamare azioni sull'oggetto e decidono il prossimo stato (transizioni).

Dare la definizione di stato e della sua struttura e di transizione nelle macchine di stato UML.

Uno stato rappresenta una condizione di esistenza dell'oggetto cioè l'attesa di un evento, l'esecuzione di azioni cioè operazioni atomiche e non interrompibili (entry o exit) ed esecuzione di attività cioè operazioni che richiedono un certo tempo. E' così strutturato:



Una transazione indica un passaggio di stato, è etichetta con le seguenti informazioni:

- gli eventi che comportano il cambiamento di stato e le condizioni sotto cui ha effetto
- le azioni che l'oggetto esegue prima di cambiare stato

Descrivere il ciclo di vulnerabilità per sistemi sicuri.

È una sequenza di eventi che si susseguono molto comunemente nel mondo della sicurezza: va dalla determinazione alla eliminazione di una vulnerabilità del software. È un ciclo ortogonale al ciclo di vita del

software, e solitamente accade dopo la sua distribuzione. Sequenza di eventi:

- Viene scoperta una nuova vulnerabilità in un pezzo di software.
- I cattivi analizzano l'informazione e sfruttano la vulnerabilità per lanciare attacchi contro il sistema o la rete.
- I buoni cercano una soluzione al problema: analizzano la vulnerabilità, sviluppano una soluzione e la testano in un ambiente controllato, distribuiscono la soluzione.
- Se la vulnerabilità è seria, o gli attacchi sono gravi, anche i media ne danno pubblica informazione con catastrofiche conseguenze.
- La patch distribuita (spesso rilasciata come parte di update automatico), viene installata agli addetti.
- Tecnici della sicurezza analizzano frammenti di codice per scoprire vulnerabilità simili.

Complicazioni: Sistemi e reti vulnerabili raramente vengono riparate durante il ciclo di vita della vulnerabilità: la patch sono ricevute come parte di una versione aggiornata del software. Il software maligno rilasciato via Internet può sfruttare la vulnerabilità dei sistemi causando e propagando danni senza misura.

Elencare le qualità esterne di un prodotto software sicuro.

Le qualità esterne (black box view) sono quelle percepite da un osservatore esterno che esamina il prodotto come se fosse una scatola nera visibile all'utente e riguardano soprattutto le funzionalità del programma. Esse sono:

- **Correttezza**: se rispetta le specifiche funzionali di progetto.
- **Affidabilità**: se si comporta in modo corretto, un sw affidabile è anche corretto ma non viceversa.
- **Efficienza**: se utilizza correttamente le risorse di calcolo a disposizione come la memoria
- **Portabilità**: se può funzionare su più piattaforme come Java.
- **Riusabilità**: se il codice può essere riusato, tutto o in parte, per altri sistemi.
- **Interoperabilità**: cioè la capacità di un sistema di coesistere e cooperare con altri sistemi.
- **Robustezza**: se si comporta in modo adeguato in caso di errori.

Dare la definizione di macchina di comunicazione e portare un semplice esempio.

Alcuni sistemi possono essere modellati in maniera più semplice con un insieme di FSM che operano in modo concorrente ed interagiscono scambiandosi messaggi, piuttosto che con una singola FSM
Le CFSM (Communicating Finite State Machines) modellano un insieme di FSM concorrenti che interagiscono tra loro
Le CFSMs sono particolarmente utilizzate per la specifica di sistemi embedded e protocolli di comunicazione

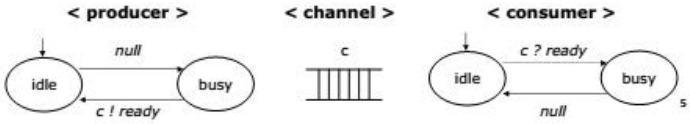
Macchine di comunicazione (Communicating finite state machines [CFSM]): coppia (C, P) dove:

- C: numero finito di canali
- P: numero finito di processi (tupla (S, I, O, T)):
    - S: stati
    - I: Input
    - O: Output
    - T: Transizioni (s, null, s') (s, c?i, s') (s, c!o, s')

Esempio: Un produttore e un consumatore comunicano attraverso un canale

Esempio

- C = {c}
- P = {producer, consumer}
  - producer = (S, I, O, T)
    - S = {idle, busy}, I = {}, O={ready}
    - T = {(idle,null,busy), (busy,c!ready,idle)}
  - consumer = (S', I', O', T')
    - S' = {idle, busy}, I' = {ready}, O' ={}
    - T' = {(idle, c?ready,busy), (busy,null,idle)}



Esempio

- (<idle, idle>, c=empty)
- (<idle, idle>, c=ready)
- (<busy, idle>, c=ready)

sono stati globali

Dare la definizione di processo e descrivere le proprietà relative al processo software

Per processo si intende come viene realizzato il software ossia le fasi di sviluppo relative al software. Il processo ha le seguenti proprietà:
- Produttività: ossia l'efficienza del processo di produzione in termini di velocità di consegna, viene migliorato dal riuso del codice e dall'automazione della creazione del codice.
- Tempestività: ossia la capacità del processo di produzione di valutare e rispettare i tempi di consegna del prodotto.
- Trasparenza: ossia la capacità del processo di produzione di capire il suo stato e tutti i suoi passi. E' trasparente se tutti i passi successivi allo stato iniziale sono documentati in modo chiaro e accessibili da un utente esterno per far fronte, ad esempio, a richieste di rapporti di stato avanzamento lavori.

Descrivere le caratteristiche del linguaggio C che possono causare problemi di sicurezza e motivare la risposta.

Non tutti i linguaggi, offrono lo stesso grado di efficienza e di sicurezza.

Il linguaggio C, oltre ad essere efficiente e facile da usare, presenta alcuni problemi di sicurezza:

- **dereferenziazione del null:** la dereferenziazione di un puntatore in C non viene controllata quindi se accedo ad una cella puntata da un puntatore nullo ottengo un segmentation fault.
- **type cast non controllato:** il C permette la conversione non controllata da un tipo ad un altro, da un tipo ad un sopratipo con possibile perdita di informazioni e da intero ad una funzione per cercare di eseguire una certa locazione di memoria che potrebbe però essere un istruzione non corretta o fare qualcosa di non permesso.
- **pointer arithmetic:** mediante l'aritmetica dei puntatori è possibile puntare a zone di memoria con tipo di diversa o a cui non si avrebbe il diritto di accesso.
- **out of bounds** (violazione spaziale): il classico buffer overflow
- **dangling pointers** (violazione temporale): un puntatore che punta ad un'area di memoria non più valida in quanto già liberata.

Definire cosa sono i test di accettazione, conformità, integrazione e regressione.

- **accettazione**: il comportamento del software è confrontato con i requisiti dell'utente finale
- **conformità**: il comportamento del software (tutto) è confrontato con le specifiche dei requisiti
- **sistema**: controlla il comportamento dell'intero sistema (hw + sw) come monolitico.
- **integrazione**: controllo sul modo di cooperazione delle unità (come previsto dal progetto)

- **unità**: test del comportamento delle singole unità
- **regressione**: test del comportamento di release successive

Dare la definizione di affidabilità e di robustezza per il software.

- **Affidabilità:** se il sw si comporta in modo corretto, un software affidabile è anche corretto ma non viceversa.
- **Robustezza:** se il sw si comporta in modo adeguato in caso di errori.

Dare la definizione di artifatti e stakeholders in un processo di sviluppo software.

L'analisi di sicurezza comporta l'impiego di tutti gli artifatti e degli stakeholder coinvolti come risorse per l'identificazione dei rischi.

- Gli **artifatti** sono quindi i documenti dei requisiti, documenti dell'architettura e del design, modelli, codice esistente ad anche i casi di test.
- Gli **stakeholders** sono quelle persone che hanno interesse nel sistema ossia utenti, clienti, ingegneri del software/della sicurezza, esperti di dominio.

Definire il principio di auditability per architetture sicure.

Principio di auditability: cioè deve essere possibile ricostruire la sequenza di eventi che hanno portato ad azioni chiave o critiche che per esempio hanno causato un errore, questo principio richiede la creazione di audit log.

Descrivere come è possibile dimostrare la correttezza dell'operazione di una classe Java sfruttando i contratti del design-by-contract.

Nel Design-by-Contract una classe è corretta se sono corretti:
- **operazione di creazione**: per ogni nuova istanza INV vale e le precondizioni valgono prima del costruttore e successivamente valgono sia le postcondizioni e INV.
- **Ogni altro metodo OP**: se chiamo un metodo OP con le precondizioni vere e INV vera allora dopo l'esecuzione del metodo valgono sia le postcondizioni e INV.

In Java è possibile utilizzare del tool per la verifica della correttezza come JML che se non sollevano eccezioni dimostrano la correttezza di una classe.

Descrivere succintamente i modelli di ciclo di vita del SW per i processi incrementali.

I processi incrementali hanno la capacità di adattarsi ai cambiamenti di requisiti, di specifica e di design. Permettono il riciclo del codice e consentono validazione o verifica.
Sono processi ottimali per lo sviluppo di software corretto, affidabile e robusto.
I modelli di processi incrementali sono: prototipazione, modello a fasi di release (o incremental delivery) e modello a spirale.

Prototipazione: è basato sul principio "Do It twice", è un modello approssimato che ha lo scopo di fornire un feedback necessario ad individuare con esattezza tutte le caratteristiche del sistema e tutti gli errori di progettazione effettuati.
Si deve massimizzare la modularizzazione ed ingegnerizzare le sole componenti critiche del prototipo a favore della riusabilità per contenere i costi.
A fasi di release: è basato sul principio "early subset, early delivery, early feedback", si parte da

sottoinsiemi critici sui quali richiedere il feedback del cliente, man mano attraverso un processo di raffinamento successivo si otterrà un prodotto che soddisfi i requisiti.
Il modello a spirale è un processo ciclico tra le seguenti attività (4 fasi): analisi dei rischi, sviluppo (specifica, design, codifca), testing e revisione della release. Ad ogni ciclo il costo aumenta come se fosse una spirale, ingloba la prototipazione ed approcio iterativo,
può essere visto come un meta modello.
Il mantenimento è semplicemente una forma di sviluppo continuo.

Esprimere le proprietà di affidabilità e validità per un criterio di test; enunciare il teorema di Goodenough e Gerhart.

Un criterio è **affidabile** se per ogni coppia di test set T1 e T2 adeguati secondo il criterio C, se T1 individua un malfunzionamento, allora anche T2 e viceversa.

- OK(P,T) non dipende dal particolare T selezionato
- Tutti i test set o riescono a trovare difetti o non ci riescono
- Posso prendere il più piccolo T in quanto se li trovo con uno più grande li trovo anche con uno più piccolo.
- Un criterio affidabile produce sempre risultati consistenti e ugualmente utili.
- Non è detto però che un criterio affidabile riesca a scoprire malfunzionamenti.

Affidabilità è importante ma non sufficiente.

Un criterio C è **valido** se, qualora il programma P non sia corretto, esiste almeno un test set T che soddisfi C in grado di individuare il difetto.
Un criterio valido riesce (ma non garantisce) a trovare il difetto.

Valido e affidabile= scoprire sempre il difetto.

Teorema di **Goodenough e Gerhart**

Dato un Criterio C, un programma P.
Se C è affidabile per P e C è valido per P allora c è ideale e
T test suite selezionato con C e
T non trova malfunzionamenti in P: ok(P,T) → OK(P), T è ideale

Dare la definizione di una FSM estesa e la definizione di stato globale e transizione globale per questa classe di macchine.

Una macchina a stati finiti estesa è una tupla (S,I,O,V,T)

- S: insieme finito di stati
- I: insieme finito di eventi di input
- O: insieme finito di eventi di output
- V: insieme finito di variabili
- T: insieme finito di transazioni

Una transazione è una tupla (s, i, o, g, a, s') con

- s: stato sorgente
- i: evento di input
- o: evento di output
- g: predicato sulle variabili in V (guardia)

- a: assegnamento ad una variabile in V (azione)
- s': stato target

Per una EFSM (S,I,O,V,T), una coppia (s, o) è detta STATO GLOBALE se:
- s è uno stato
- o è una valutazione su V

Per una EFSM (S,I,O,V,T), una tupla ((s,o),i,o,(s',o')) è detta TRANSIZIONE GLOBALE se:
- esiste una transizione (s,i,o,g,a,s') tale che:
  - o soddisfa òa guarda g e
  - o'(v)=o(exp) dove l'azione a è v:=exp

Definire gli elementi che costituiscono un contratto SW secondo i principi del design-bycontract.

Un contratto è un accordo tra cliente e fornitore che:
- Lega le due (o più) parti: fornitore e cliente
- È esplicito (scritto)
- Specifica gli obblighi e i benefici delle due parti
- Normalmente mappa gli obblighi di una parte come benefici dell'altra parte
- Non contiene clausole nascoste: gli obblighi sono quelli dichiarati

Un contratto è composta da preconzioni, postcondizioni e invarianti.
- Le precondizioni definiscono cosa ogni metodo richiede e sono obbligatori per il cliente
- Le postcondizioni definisci cosa fornisce il metodo e sono obbligatorie per il fonitore
- Le invarianti sono proprietà che valgono sempre per tutte le istanze. È un ulteriore precondizione che non deve essere violata.

Le precondizioni e le postcondizioni sono usate molto per evitare che entrambi, cliente e fornitore, effettuino i medesimi controlli.

Descrivere la classificazione dei *requisiti* in fase di specifica.

La descrizione informale di un sistema SW è data in termine di requisiti. I requisiti specificano cosa un sistema SW deve fare, non come deve farlo.

I requisiti in fase di specifica si classificano in:

- **Requisiti funzionali:** specificano una funzione che il sistema deve compiere.
- **Requisiti non funzionali**: prestazioni, affidabilità, efficienza, portabilità, modificabilità.
- **Requisiti del processo e manutenzione**: procedure per il controllo della qualità, priorità di sviluppo e dei possibili cambiamenti.

La specifica dei requisiti è una descrizione completa e non ambigua dei requisiti. Si fa attraverso un'analisi iterativa e cooperativa del problema, richiede l'impiego di linguaggi formali o semiformali e il controllo della comprensione del problema.

Le qualità delle specifiche:
- **Chiarezza:** la specifica deve descrivere quanto più chiaramente possibile i termini e le operazioni coinvolte.
- **Non ambiguità**: la specifica non deve generare interpretazioni ambigue.

- **Consistenza**: la specifica non deve contenere contraddizioni.
- **Completezza**: il processo descritto dalla specifica deve essere definito in modo completo e dettagliato. Si divide in due parti:
  - **Completezza interna:** la specifica deve definire ogni concetto nuovo e ogni terminologia usata(definire un glossario)
  - **Completezza esterna**: la specifica deve essere completa rispetto ai requisiti
- **Incrementalità**: la specifica viene sviluppata in più passi successivi.
- **Comprensibilità**: la specifica, in quanto contratto tra committente e produttore, deve essere intuitiva e comprensibile per il cliente.

Descrivere le tre modalità di modifica di un sistema sicuro nella fase di manutenzione.

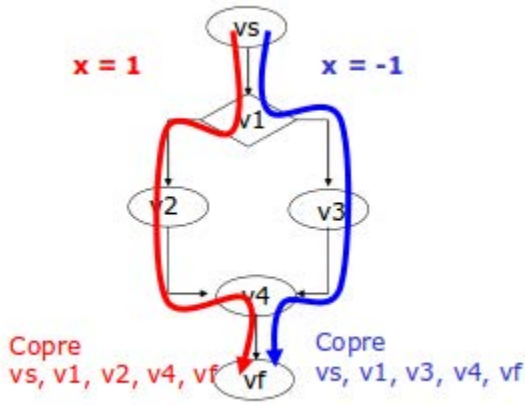
La fase di manutenzione è quella fase in cui si devono applicare modifiche che possono essere:
modifiche correttive: si ricercano gli errori e li si correggono
modifiche perfettive: si aggiungono nuove funzionalità al sistema
modifiche adattive: si adatta il sistema a modifiche del dominio applicativo.

Un software è facile da manutenere se è facile applicare queste modifiche.

Dare la definizione di test set adeguato per il criterio di copertura delle istruzioni. Fare anche un semplice esempio.

Un test set T è adeguato per testare un programma P secondo il criterio di copertura delle istruzioni se per ogni istruzione s di P esiste un caso di test i T che esegue s (ossia ogni istruzione viene eseguita almeno una volta).

```
vs: int main(int x) {
      int result;

v1:   if (x > 0)
v2:       result = x;
      else
v3:       result = 1 / x;
v4:   printf("%d", result);
vf: }
```



Descrivere il processo di valutazione per ottenere la certificazione di SW sicuro in base ai common criteria.

Common Criteria: è uno standard ISO che definisce un insieme di classi e di comportamenti di sicurezza progettate per essere opportunamente combinate per definire profili di protezione per ogni tipo di prodotto IT, incluso hardware, firmware, software.

Common Evaluation Methodology: Definisce le modalità di valutazione di un sistema in base a criteri comuni.

Il processo di valutazione per ottenere la certificazione di sw sicuro ha i seguenti step (i punti 1,2,3

sono a carico del cliente, il 4 del vendor, il 5,6,7 a carico dell'istituto accreditato):

6) Si produce un profilo di protezione.
7) Il profilo viene valutato in base alle Common Evaluation Methodology per garantire che esso sia consistente e completo.
8) Ottenuta la valutazione, il profilo viene pubblicato (target di sicurezza).
9) Un vendor realizza una sua versione (target di valutazione) del profilo di protezione.
10) Il target di valutazione viene inviato ad un istituto accreditato per la valutazione rispetto al target di sicurezza: l'istituto applilca la Common Evaluatoin Methodology per determinare in base ai common criteria se il target di valutazione soddisfa il target di sicurezza.
11) Se la valutazione ha esito positivo, la documentazione di testing dell'istituto viene inviata al NAVLAP (National Voluntary Lavoratory Accreditation Program) per controllarne la correttezza.
12) Se gli atti vengono approvati, il prodotto ottiene la certificazione di prodotto valutato in base ai common criteria.

Limiti dei Common Criteria: la valutazione della sicurezza è un'attività molto più complessa dell'applicare un target di sicurezza ad un target di valutazione. I problemi di sicurezza sono context-sensitive, e quindi è difficile valutarli in base a criteri comuni.
Inoltre, come molti standard, definiscono il "cosa" e non il "come", mentre nel campo della sicurezza è fondamentale sapere come affrontare problemi di sicurezza e gestire i rischi.

Descrivere le possibili classificazioni di testing a seconda del livello a cui si effettua, del tipo di accesso e degli aspetti da testare.

Livelli di testing:

- Accettazione: requisiti utente finale
- Conformità: specifiche dei requisiti
- Sistema: comportante interno Hardware e Software
- Integrazione: Cooperazione tra unità
- Unità: Comportamento singola unità:
- Regressione: successive release

Aspetti da Testare:

- Funzionalità
- Affidabilità-Reliability
- Robustezza
- Performance
- Usabilità

Accesso al Sistema:

- White Box
- Black Box
- Grey Box

Definire la differenza tra testing funzionale e testing di sicurezza in sistemi software sicuri.

Il **testing funzionale** mette alla prova il sistema e valuta se si comporta in modo corretto in circostanze normali e in circostanze critiche.

Il **testing di sicurezza**, valuta il comportamento del prodotto nello stesso modo in cui puo' provarlo

un utente malevolo, ossia sfruttando i punti deboli del sistema.

Per il testing di sicurezza è importante utilizzare il code coverage che valuta il codice trovando parti di codice che non vengono mai utilizzate, codice non eseguito infatti puo' averi seri bachi sfruttabili per attacchi di successo.

Descrivere il Principio di fault safely delle architetture sicure.

Il principio di fault safely (fallimento sicuro) dice che in caso di fallimento un sistema deve terminare in una configurazione sicura. Esempio: se fallisce il programma di una porta a chiusura elettronica la porta deve rimanere aperta.

Descrivere in che cosa consiste la qualità di auditing in sistemi software sicuri.

Per auditing si intende un processo del sistema che prende in considerazione le caratteristiche del prodotto, le politiche di sicurezza e le sue funzionalità al fine di garantire la loro compatibilità. Esso viene utilizzato per migliorare l'introduzione di funzionalità e dissuadere da potenziali attacchi.

Descrivere il Principio di mediazione completa delle architetture sicure.

Il principio di mediazione completa consiste nel fatto di lanciare, passo dopo passo, tutte le possibili azioni
che sono possibili effettuare contro la policy con lo scopo di garantire che la policy possa dissuadere da potenziali attacchi in ogni fase della sua configurazione.

Descrivere tre Principi importanti per architetture sicure.

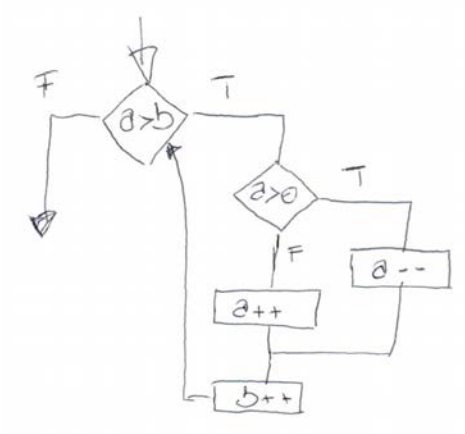
- **Adversary Principle:** progettare il SW come se il nemico più astuto lo attaccherà, per fare ciò si deve avere in mente cosa il nemico potrebbe volere attaccare.
- **Chain of Trust:** non invocare programmi non fidati da programmi fidati. Un programma non deve delegare l'autorità di fare un'azione senza delegare anche la responsabilità di controllare se l'azione è appropriata.
- **Minimo privilegio:** un applicazione deve avere i permessi strettamente necessari al suo funzionamento.
- **Mediazione completa:** consiste nel fatto di lanciare, passo dopo passo, tutte le possibili azioni che sono possibili effettuare contro la policy con lo scopo di garantire che la policy possa dissuadere da potenziali attacchi in ogni fase della sua configurazione.
- **Non offuscamento:** nascondere la modalità di funzionamento di una componente SW o il registro in cui è memorizzato un particolare parametro di policy è pericoloso.
- **Minimal Retained State:** un programma deve tenere in memoria lo stato minimale.
- **Fault Tolerance:** uso delle 3 R:
- **Resistance:** capacità di impedire un attacco.
- **Recognition:** capacità di riconoscere un attacco e la misura dei danni.
- **Recovery:** capacità di fornire servizi seppur ridotti durante un attacco ed il ripristino successivo.
- **Accettabilità psicologica:** le misure di sicurezza non devono essere onerose ed irritanti in modo da spingere l'utente ad evitarle.
- **Accountable Individuals:** gli individui devono essere responsabili delle proprie azioni.

Nello unit testing, definisci cosa si intende con test unit, test driver e test stub.

- **Unit testing:** l'attività di testing (prova, collaudo) di singole unità software.
- **Test-Driven**: è un processo di sviluppo del software in cui lo sviluppo vero e proprio è preceduto (e guidato, driven) dalla stesura di test automatici. Il processo si articola sulla ripetizione di brevi cicli di sviluppo e collaudo (noti come "cicli TDD", TDD cycles) suddivisi in tre fasi successive, sintetizzate dal motto "Red-Green-Refactor". Nella prima ("Red"), il programmatore scrive un test automatico (che necessariamente fallisce) per la funzionalità da sviluppare. Nella seconda, il programmatore scrive la quantità minima di codice necessaria per ottenere il superamento del test. Nella terza, il programmatore ristruttura il codice (ovvero ne fa il refactoring). I colori "rosso" e "verde" si riferiscono alla rappresentazione grafica di fallimento e successo di un test automatico più diffusa negli IDE.
- **Test stub** :Uno stub può simulare il comportamento di codice esistente, quindi un test stub è un test di simulazione.

Dare i casi di test che garantiscono lo statement coverage ed il branch coverage del seguente programma, con a e b interi relativi:

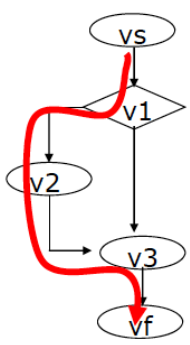
```
read(a,b);
while a > b do
        if a > 0 then a:= a--;
                else a:= a++;
        endif
        b:= b++;
endwhile
        end
```



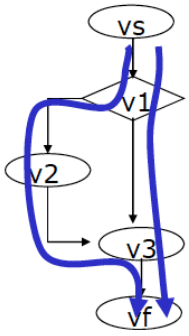
Un criterio tipo “a+b <= -1” genera test set che sono Branch Coverage, non avendo rami delle decisioni privi di istruzioni. È analogo allo Statement Coverage.

Esempio: T1={(5,-6),(2,-4)} T2={(0,-2),(0,-1)}

Il test Statement Coverage prevede che ogni istruzione venga eseguita almeno una volta, il Branch Coverage invece richiede che tutti gli archi del grafo vengano percorse almeno una volta. Quindi se un test set T soddisfa il branch coverage, soddisfa anche il Statement Coverage.



Statement coverage



Branch coverage

Fare un'analisi completa (e motivata) con i vari criteri di copertura per il seguente frammento di programma. [pt.4]

```
foo (int x, int y, int z, int w ) {
        if ((x >0 && y > 5) || (z < 0))
        then if (x > 0 && z < 0) then statement1;
                                 else statement 2;
        else statement3;
}
```

Dato il seguente programma che dovrebbe fare la somma di due numeri, ma che contiene un difetto:

```
int somma (int x, int y ) { if (x>0 && y>x) return x*y;
                                            else return x+y}
```

determinare un criterio per la costruzione di test set ideali.

Il test set ideale viene generato da un criterio C che è affidabile e valido per P, criterio che potrebbe essere x>0 e y>x. Per questo criterio il test risulta affidabile poiché restituisce sempre SUCCESS presi due Test Set e valido poiché almeno un test riesce a trovare il difetto.

Esempio: T1={(2,3)} T2={(3,4),(5,6)}

Determinare una test suite per il decision coverage del seguente frammento di programma:

```
foo (int x, int y, int z, int w ) {
        if ((x >0 && y ==15) || (z !=0 && w == z+1))
        then statement1;
        else statement2;
}
```

Il Decision Coverage è un test set T adeguato per testate un programma P secondo il criterio di copertura delle decisioni, se per ogni decisione di P esiste:

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

T={(1,15,0,1),(0,15,1,1)}

Dato il seguente programma che dovrebbe fare la somma di due numeri, ma che contiene un difetto:

```
int somma (int x, int y ) {
        if (x > y) return x*y else return x+y;
}
```

fornire un test set affidabile ed uno ideale.

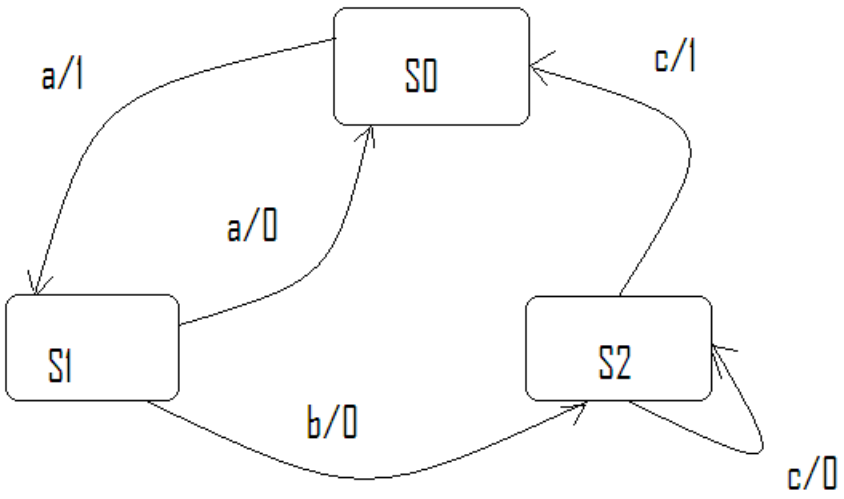
Un criterio C è **affidabile** se per ogni coppia di test set T1 e T2 adeguati secondi C, se T1 individua un malfunzionamento, allora anche T2 lo trova e viceversa.

Per il criterio (x<=y), tutti i test set restituiscono FAIL (non individuano errore) quindi è affidabile.
Esempio: T1={(3,3),(2,3)} T2={(2,2),(0,2)}

Un criterio C è **ideale** se è affidabile è valido. Un criterio C è valido se, qualora un programma P non sia corretto, esiste almeno un test set T che soddisfa C che è in grado di individuare il difetto.

Per il criterio (x>y), tutti i test sono affidabili sempre SUCCESS, è valido in quanto esiste almeno un test che trova il difetto:
Esempio: T1={(3,2),(4,3)} T2={(1,0),(2,1)}

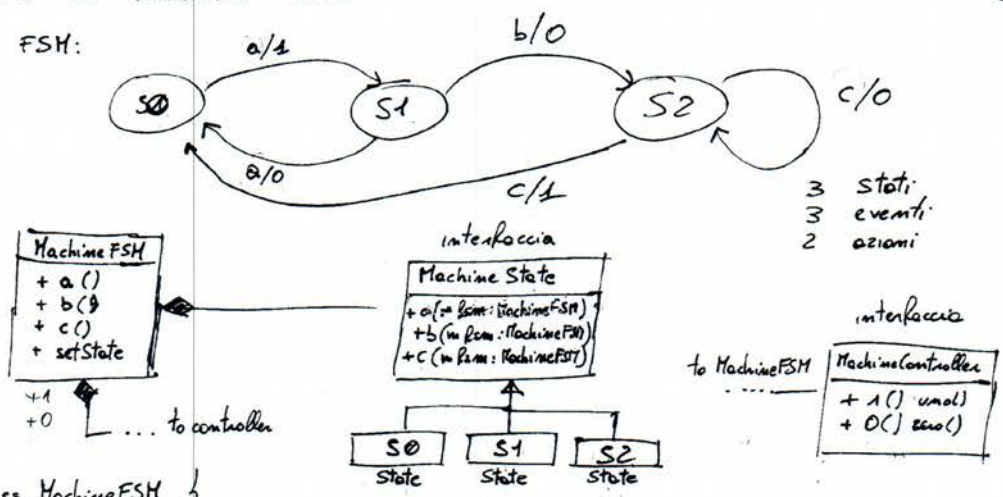
Data la macchina di stato FSM avente
S=(s0,s1,s2); I=(a,b); O=(0,1); T=(<s0,a,1,s1>,<s1,a,0,s0>,<s1,b,0,s2>,<s2,c,0,s2>,<s2,c,1,s0>)}
darne la rappresentazione mediante state pattern.



Data la macchina di stato FSM avente
S=(s0,s1,s2); I=(a,b); O=(0,1);
T=(<s0,a,1,s1>,<s1,a,0,s0>,<s1,b,0,s2>,<s2,c,0,s2>,<s2,c,1,s0>)}
darne la codifica Java mediante state pattern.

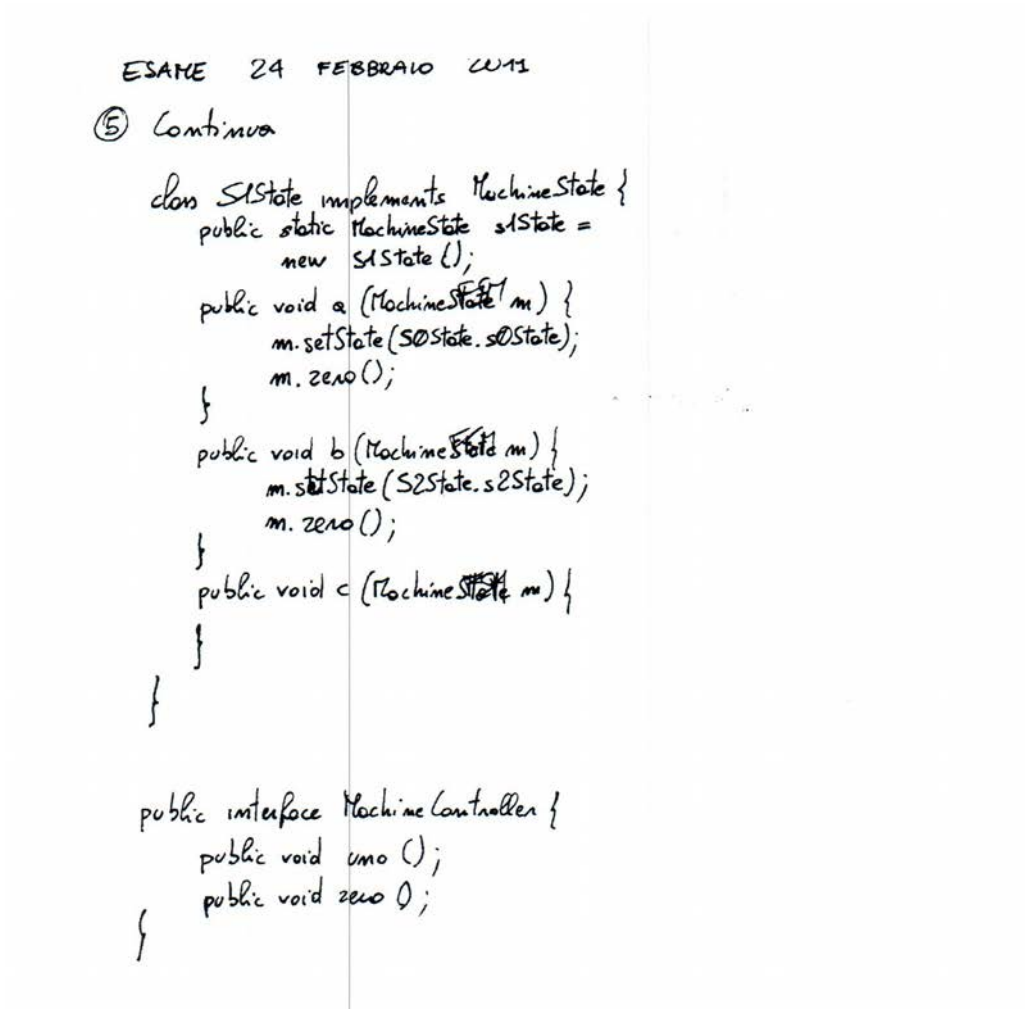
ESAME 24 FEBBRAIO 2011 (2)

⑤ FSM:




```
class MachineFSM {
    private MachineController mController;
    private MachineState state = S0State.s0State;
    public MachineFSM (MachineController action) {
            mController = action;
    }
    public void a() {
        state.a(this);
    }
    public void b() {
        state.b(this);
    }
    public void c() {
        state.c(this);
    }
    public void setState(MachineState newState){
        state = newState;
    }
    public boolean isS0() {
        return state == S0State.s0State;
    }
    public boolean isS1() {
        return state == S1State.s1State;
    }
    public boolean isS2() {
        return state == S2State.s2State;
    }

        // .. Segue ...

    (*) m.State(s1State.s1State);
```

```
// .... Continuo.. MachineFSM

void uno() {
    mController.uno();
    System.out.println("1");
}

void zero() {
    mController.zero()
    System.out.println("0");
}

interface MachineState {
    void a(MachineFSM m);
    void b(MachineFSM m);
    void c(MachineFSM m);
}

class S0State implements MachineState {
    public static MachineState s0State =
            new S0State();
    public void a(MachineFSM m) {
        m.uno(); (*)
    }
    public void b(MachineFSM m) {
    }
    public void c(MachineFSM m) {
    }
}

// Così anche per gli altri due stati:
// Riscrivono solo evento che li riguarda
```



Dare la definizione di test set ideale e computarlo per il seguente programma che dovrebbe restituire il valore della funzione f(x,y)=x*(y+5) per valori x,y interi, crescenti e maggiori di 1, 0 altrimenti:

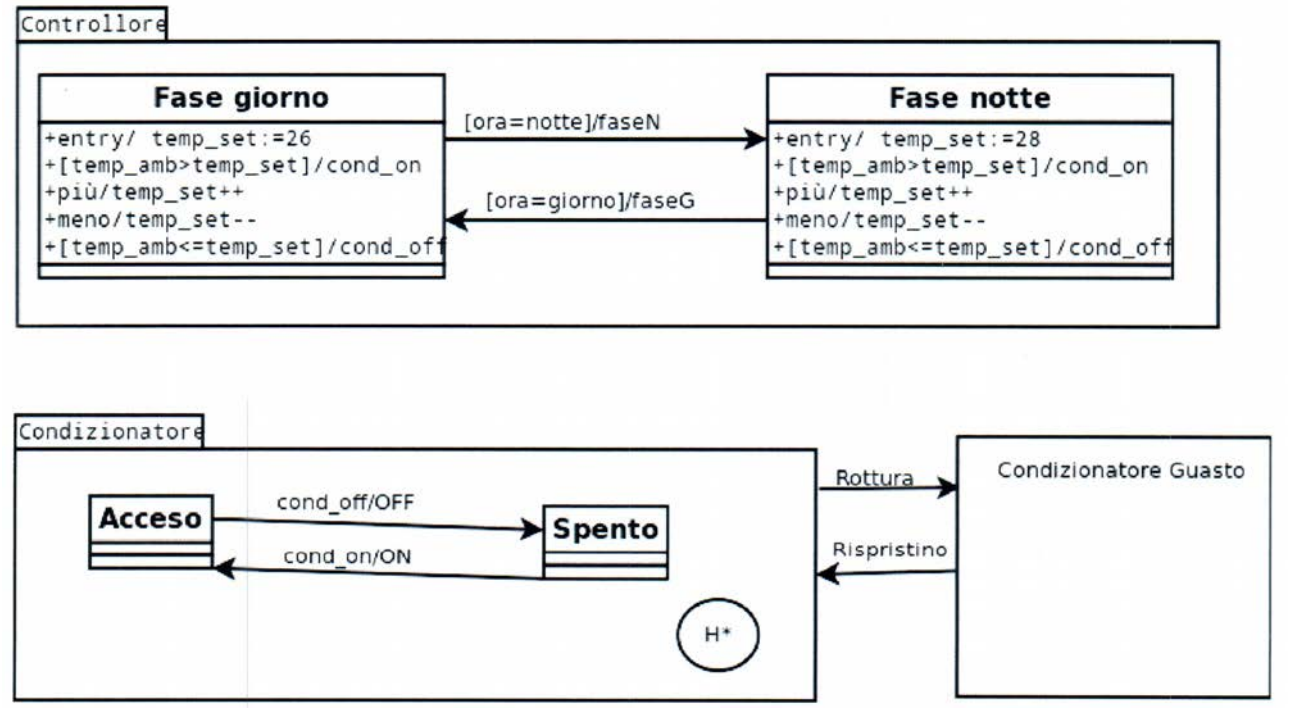
```
int foo (int x, int y ) {
        if (x > 1 & y > x) return x*(y+3)
        else return 0;
}
```

fare un esempio di test set ideale.

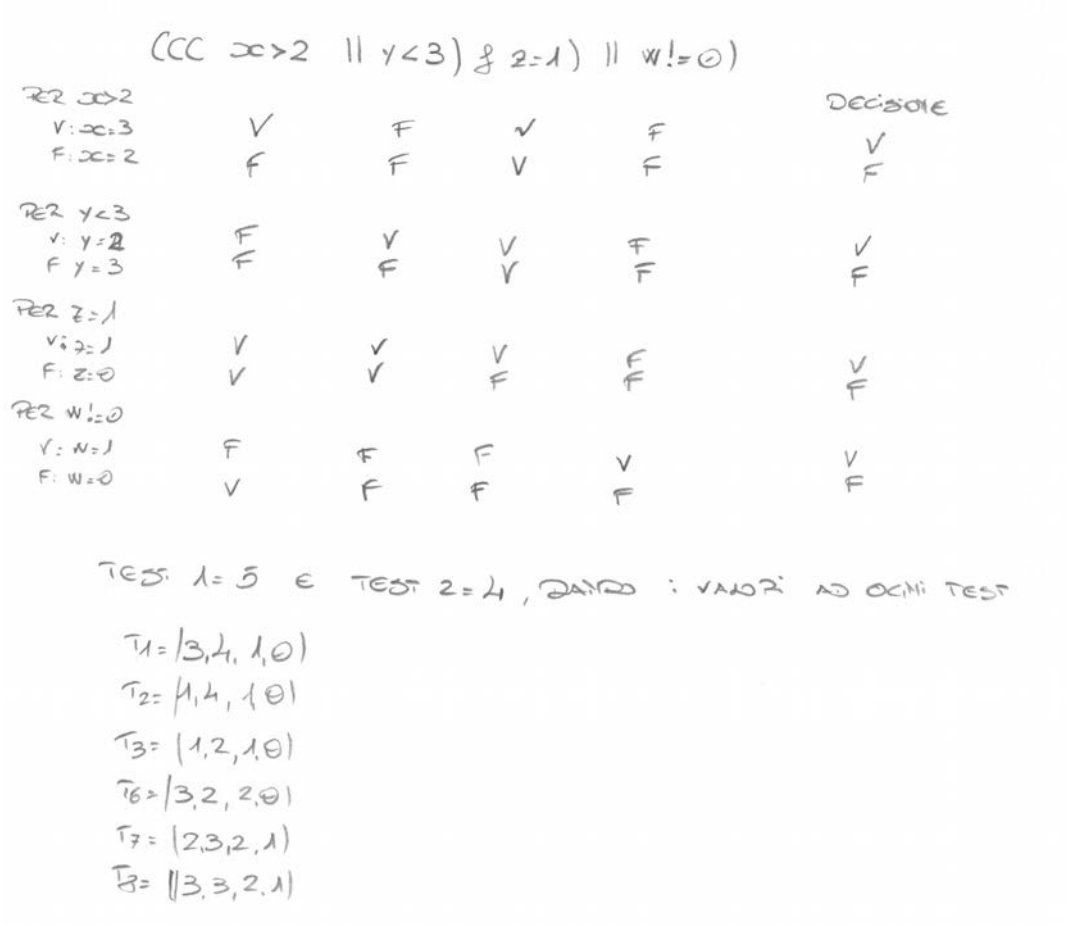
Sfruttando il teorema di Goodenough e Gerhart possiamo date la definizione di test set ideale, dato un criterio C e un programma P. Se C è affidabile per P e C è valido per P e T è un test suite selezionato con C e T non trova malfunzionamenti in P: ok(P,T). => OK(P), T è ideale.

Un criterio affidabile e valido, quindi garantisce che tutte le test suite da lui generate siano in grado di catturare l'errore potrebbe essere: “Il test generato contiene almeno un test t in cui x>1 e y>x” ovvero:
T1={(-1,3),(3,5)}
T2={(4,6),(0,1),…}

Sfruttando il meccanismo di sincronizzazione eventi-azioni di UML, modellare il seguente sistema di condizionamento. Il sistema è composto da un controllore e da una condizionatore. In fase giorno, il controllore invia il comando di accensione al condizionatore se la temperatura ambiente supera il valore settato dall'utente. L'utente può settare a piacere la temperatura dell'ambiente. Per default la temperatura ambiente è fissata a 26°. Il condizionatore si spegne automaticamente quando arriva in temperatura.
In fase notte, il controllore invia al condizionatore un comando di accensione se la temperatura è superiore ad un valore minimo. Tale valore può essere stabilito a piacere dall'utente. Per default, esso è fissato a 28°. Un eventuale guasto del condizionatore, mette l'intero sistema in una situazione di guasto, e da questo stato si torna in modalità di funzionamento normale ripristinando il sistema nella configurazione lasciata al verificarsi del guasto.



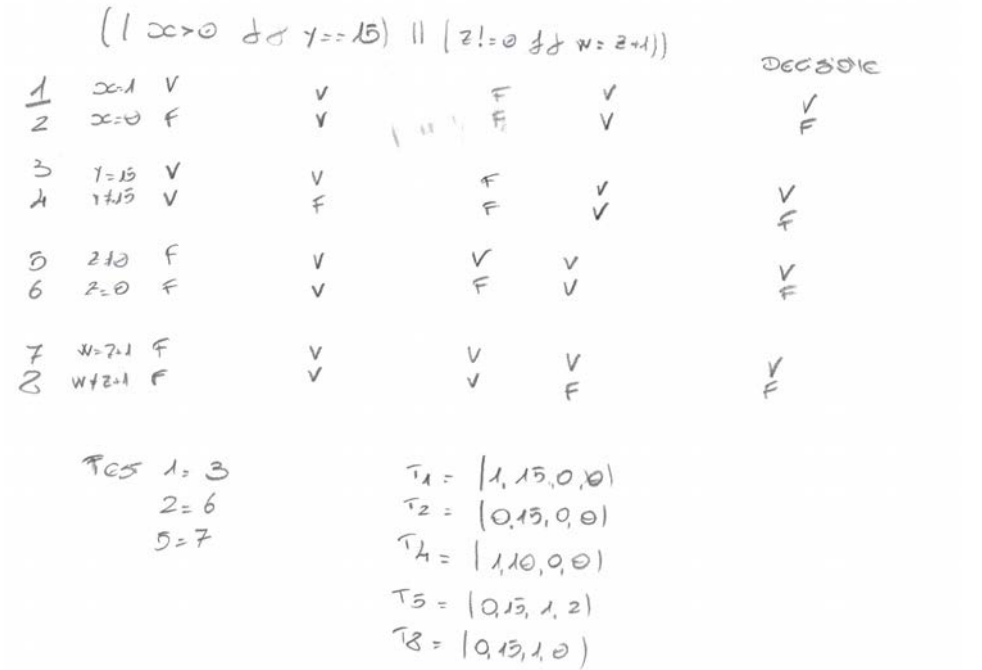
Scrivere i casi di test secondo l' MCDC per l' espressione (((x>2 || y<3) & z=1) || w!=0)



Determinare una test suite per le condizioni mediante MCDC:

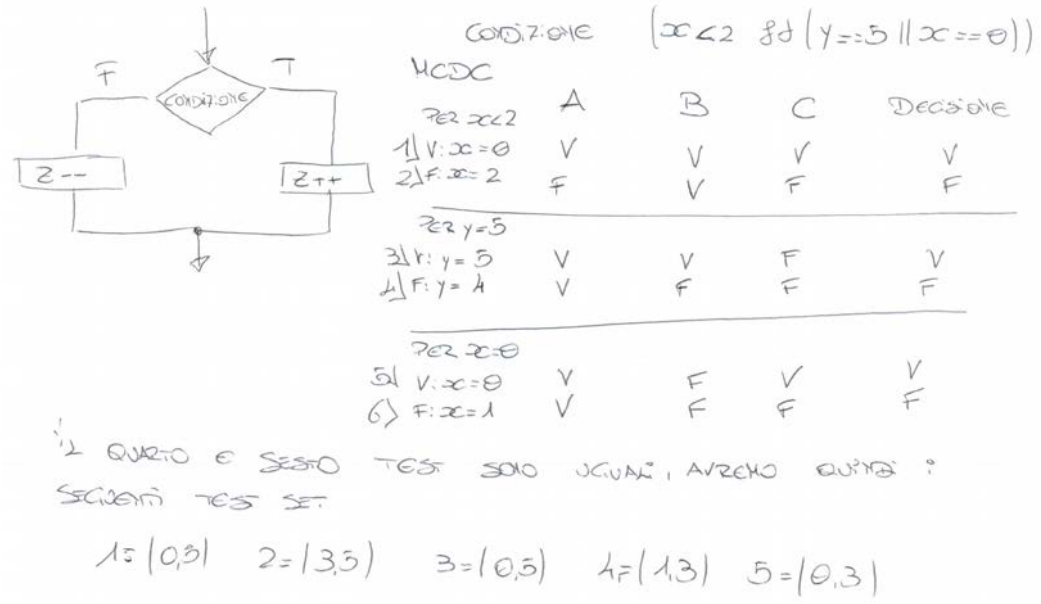
foo (int x, int y, int z, int w) {

if ((x >0 && y ==15) || (z !=0 && w == z+1)) { ….}}



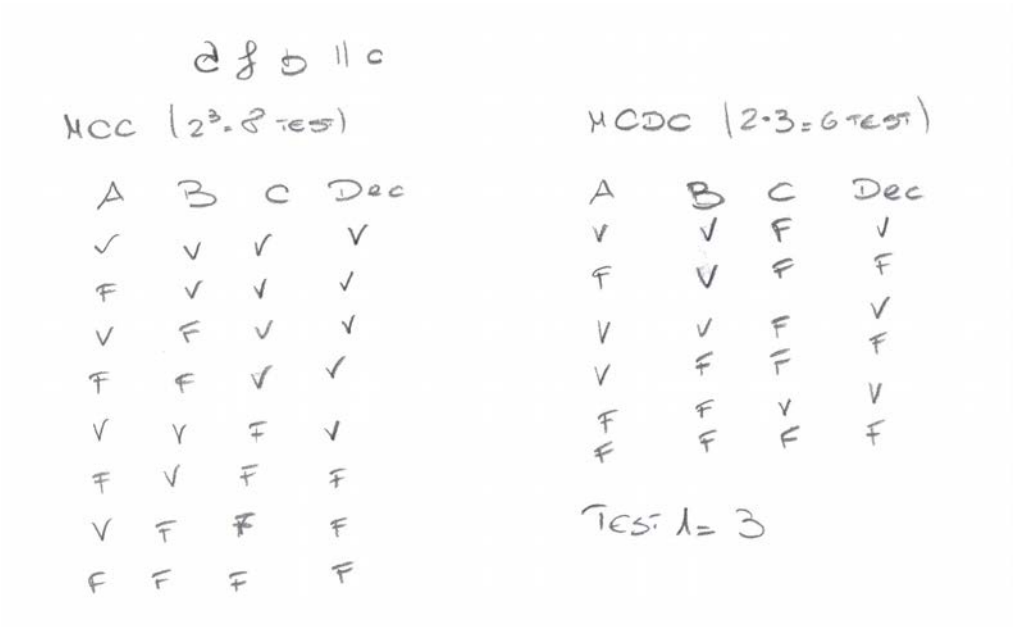
Dato il seguente programma, disegnare il grafo di flusso e determinare una test suite per le condizioni mediante MCDC:

```
foo (int x, int y ) {
        if (x <2 && (y ==5 || x ==0)) {
                z++;
        } else {
                z--;
        }
}
```

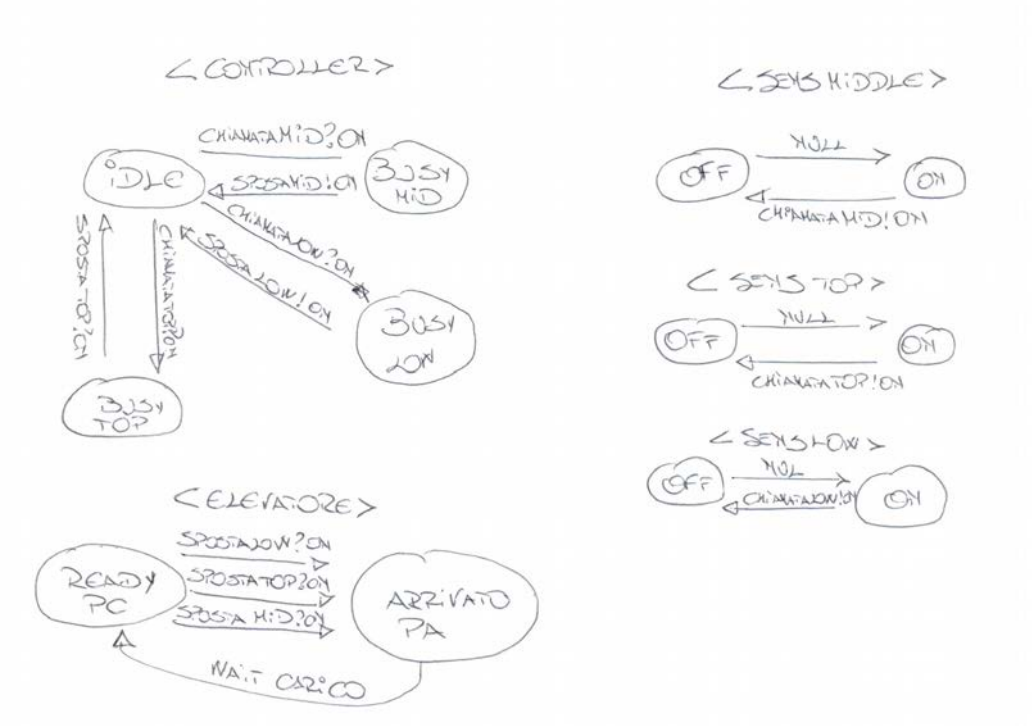


Scrivere i casi di test secondo l' MCC e l' MCDC per la seguente espressione a & b || c

- MCC: I test totali sono 2^al numero di variabili
- MCDC: I test totali sono 2*Il numero di variabili



Utilizzando le macchine di comunicazione, modellare il comportamento di un elevatore comandato da un controller. Il controller comanda l'elevatore in base alla richiesta dell'elevatore ai piani top, middle, low che avviene attraverso segnalazione del sensore request. Su segnalazione del sensore, il controller invia all'elevatore il comando di raggiungere la posizione richiesta. Raggiunto il livello, l'elevatore apre le porte ed attende il carico. Quando viene richiamato ad un livello diverso, prima di muoversi, chiude le porte, e reagisce al comando.



Utilizzando le macchine di comunicazione, modellare il comportamento di un sistema di telesoccorso. Su segnalazione di un utente, la centrale invia all'ambulanza il comando di recarsi ad un dato indirizzo. L'ambulanza impiega da 15 a 30 minuti per raggiungere il posto richiesto. E successivamente impiega da 10 a 30 minuti per portare l'utente in ospedale. Appena l'ambulanza rientra in sede (diversa da ospedale), la centrale può evadere una seconda richiesta, e così via.