Testing Combinatoriale

Andrea Fornaia, Ph.D.

Department of Mathematics and Computer Science

University of Catania

Viale A.Doria, 6 - 95125 Catania Italy

fornaia@dmi.unict.it

http://www.cs.unict.it/~fornaia/

Ingegneria e controllo qualità

- L'ingegneria alterna a fasi di design e costruzione delle fasi di verifica della qualità del prodotto
- Le attività di verifica variano a seconda di:
  - **Unicità:**
    - Prodotti di massa (creati in serie) vengono testati statisticamente (viene validato il processo)
    - Prodotti unici richiedono una verifica per singolo prodotto
  - **Complessità**:
    - Maggiore è la complessità di un prodotto, più elementi dovranno essere soggetti al controllo di qualità
  - **Variabilità:**
    - Beni con bassa variabilità, anche se unici, possono essere verificati usando procedure standard (parametriche)
  - **Criticità:**
    - Anche se prodotti in serie (es. automobili, aeroplani) alcuni beni richiedono procedure di verifica più accurate a causa della loro criticità di impiego

Verifica di un prodotto software

- Il software è **unico**, **complesso, variabile** e (in certi casi) **critico**

- Inoltre il software è **non lineare** e con una **distribuzione disomogenea dei difetti**:
  - Se un ascensore è testato per supportare 1000 kg potrà reggere anche carichi minori
  - Se una procedura riesce ad ordinare 256 elementi potrebbe fallire ordinandone 255, 257, 53, 12 …

Attività di controllo

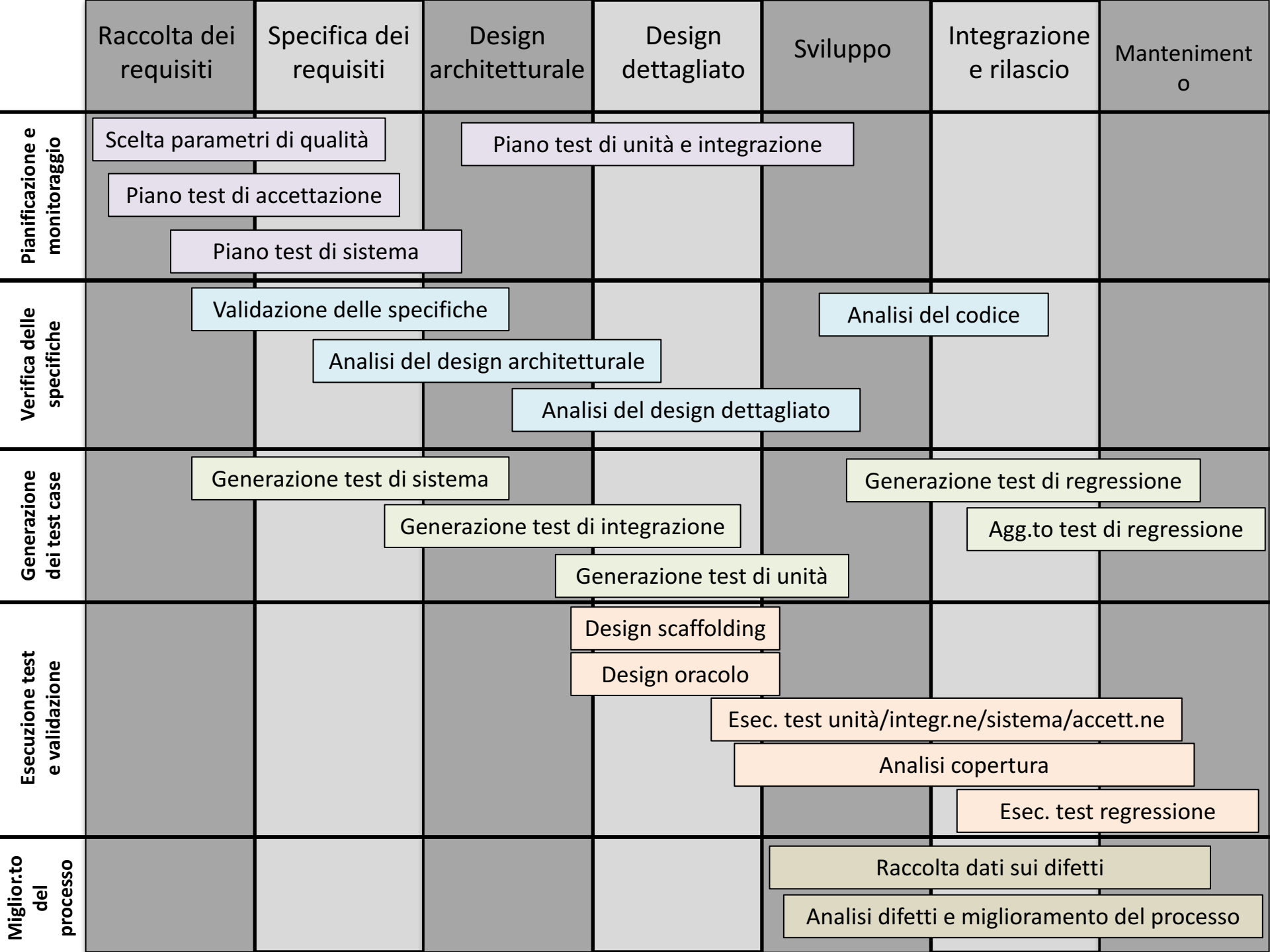
- La maggior parte delle vulnerabilità dei sistemi informatici sono riconducibili ad errori di implementazione
- Il *software testing* è una delle pratiche più importanti e costose del processo di sviluppo

- L'inadeguatezza del software testing nel 2003 è costata agli Stati Uniti ***59,5 miliardi di dollari*** *(Stima NIST)*
- Nonostante le aziende spendano in verifica e validazione del software il 50-80% del budget progettuale

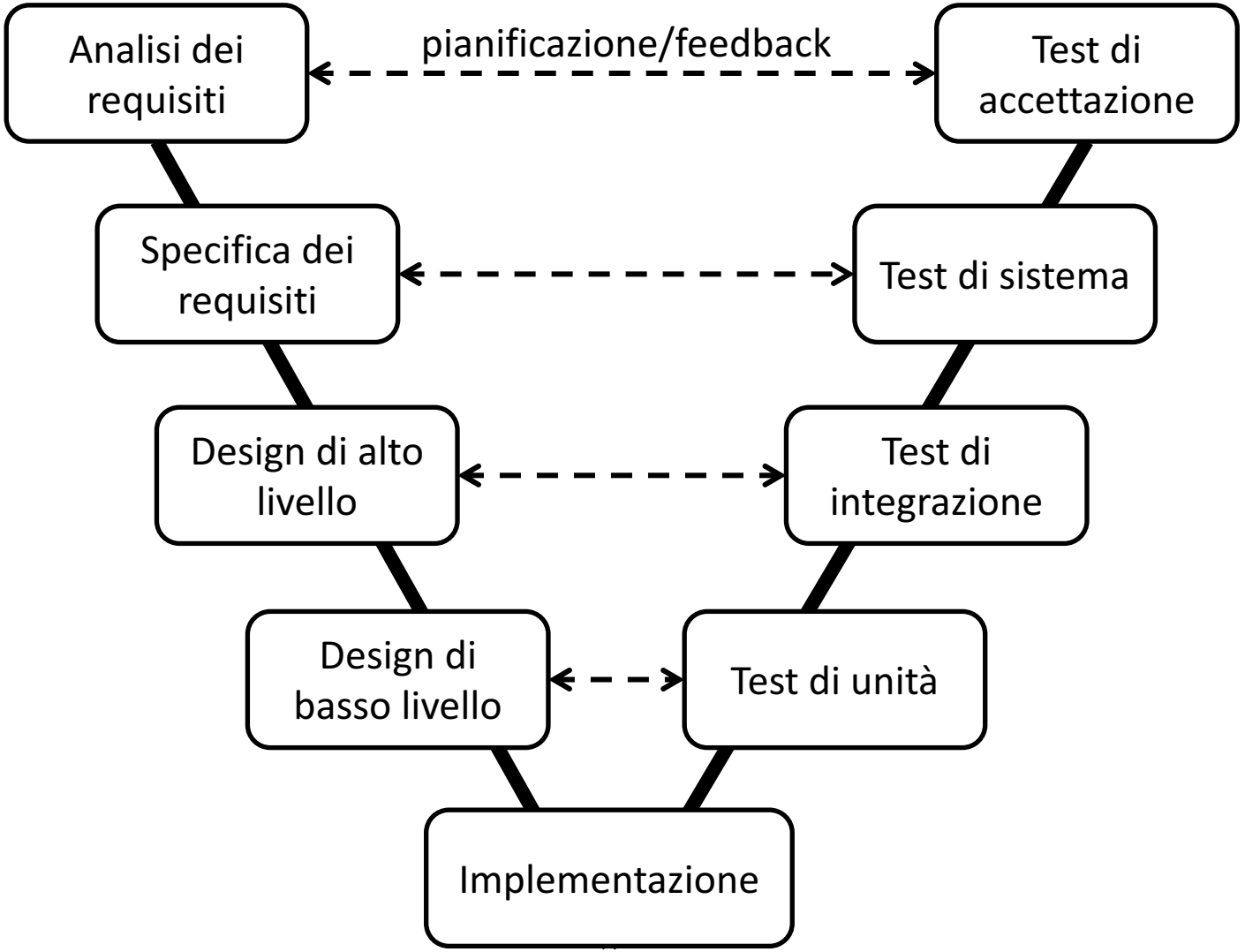
Verifica & Validazione

Verifica
Stiamo realizzando il prodotto correttamente?

Validazione
Stiamo realizzando il prodotto utile?


Raccolta dei requisiti
Specifica dei requisiti
Design architetturale
Design dettagliato
Sviluppo
Integrazione e rilascio
Mantenimento
Pianificazione e monitoraggio
Scelta parametri di qualità
Piano test di unità e integrazione
Piano test di accettazione
Piano test di sistema
Verifica delle specifiche
Validazione delle specifiche
Analisi del codice
Analisi del design architetturale
Analisi del design dettagliato
Generazione dei test case
Generazione test di sistema
Generazione test di regressione
Generazione test di integrazione
Agg.to test di regressione
Generazione test di unità
Esecuzione test e validazione
Design scaffolding
Design oracolo
Esec. test unità/integr.ne/sistema/accett.ne
Analisi copertura
Esec. test regressione
Miglior.to del processo
Raccolta dati sui difetti
Analisi difetti e miglioramento del processo

V-model


Analisi dei requisiti
pianificazione/feedback
Test di accettazione
Specifica dei requisiti
Test di sistema
Design di alto livello
Test di integrazione
Design di basso livello
Test di unità
Implementazione

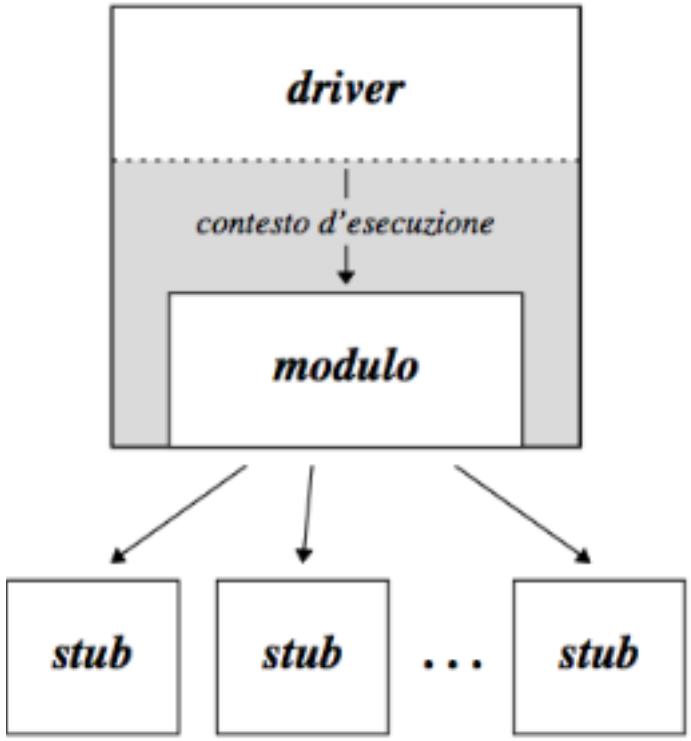
Software Testing

- Auspicabile un approccio **esaustivo**
  - Testare ogni possibile combinazione di parametri in input e path di esecuzione
- Numero di test case crescerebbe in modo **esponenziale** con il numero di parametri
  - Insostenibilità dei **costi**
- Tradeoff tra:
  - Numero di **test case** considerati (costi)
  - Livello di **copertura** raggiunto (affidabilità)

Test & Test Suite

- Una **test suite** è un insieme di test case
- Un **test case** è una tripla
  - <input, output, ambiente>

- **Input**: combinazione di parametri
- **Output**: risultato atteso
- **Ambiente**: contesto di esecuzione
  - Simulazione componenti tramite **scaffolding** (stub, mock)
    - Sistema incompleto
    - Isolamento modulo da testare



Problema dell'input

- Quali **combinazioni** di parametri in input considerare (test case)?
  - Spesso solo gli scenari descritti nei requisiti
  - Combinazioni di parametri casuali

- Cerchiamo **combinazioni di parametri** che portano ad un **malfunzionamento** del sistema (**efficacia**)
  - Spesso relativi a scenari "non previsti nei requisiti"

Combinazione di parametri

double peso(String gas, int pressione, int volume)

Si suppone una temperatura costante

“H”	5	100
“O”	10	200
	15	300
		400

Test case scelti manualmente:

1. peso(“H”, 5, 100)
2. peso(“O”, 10, 200)
3. peso(“H”, 5, 300)
4. peso(“O”, 15, 400)

Approccio Esaustivo:

2 x 3 x 4 = 24 test case

Problema dell'output

- Scelta una combinazione di input, qual è l'output atteso?
  - Problema della creazione dell'Oracolo

- Varie soluzioni:
  - Uso dei requisiti
  - Modelli analitici
  - Versioni precedenti del sistema
  - Prodotti concorrenti
  - Verificare se il sistema va in errore (es. eccezione)

Problema della scala

- Più il modulo da testare è grande (e complesso) più è difficile creare test efficaci manualmente

- Manuale, tramite asserzioni, per gli **Unit Test**:
  - Numero ridotto di parametri (combinazioni)
  - Numero ridotto di codice da coprire
  - Più semplice determinare l'output atteso

- **Test di Integrazione** e **Test di Sistema** devono verificare il comportamento di più parti del sistema nel maggior numero di situazioni possibili
  - Quante?
  - Quali?
  - Come?!

Generazione dei test

- Necessario trovare approcci sistematici:
  - Creare test suite di dimensioni ridotte ma efficaci
  - Garanzie di copertura prevedibile e misurabile
  - Inferire il comportamento atteso del sistema a partire dalle specifiche (oracolo)
  - Possibilmente, generare codice di test automaticamente

Outline

- **Testing Combinatoriale**
  - Generazione automatica di test suite di dimensioni ridotte (basso costo) ma ad alta efficacia
- **Model Checking**
  - Modellare il sistema con una macchina a stati finiti
  - Verificare la correttezza delle specifiche
  - Generare i test in base alle possibili transizioni di stato (idealmente tutte)
  - Creazione Oracolo (risultato atteso)
- **Use case: Testing Combinatoriale + Model Checking**
  - Creazione automatica test suite in ambito di sicurezza
  - Validazione del Bytecode Verifier per JavaCard

Testing Combinatoriale

Applicazione Web

- Vogliamo accertare la qualità dell'applicazione al variare di determinate scelte di configurazione

Client	Web Server	Payment	Database
FIREFOX	WEB SPHERE	MASTER CARD	DB/2
IE	APACHE	VISA	ORACLE
OPERA	.NET	AMEX	ACCESS

- Problema di **dimensione 3^4**
  - 4 parametri di confgurazione
  - 3 valori possibili ciascuno
- Utilizzando un approcio esaustivo (quadruple): **3^4 = 81 test case**
- Utilizzando un approccio pairwise: $3^2\binom{4}{2}$ = **54 test goal pairwise**
- Ogni test case può coprire fino a $\binom{4}{2}$ = **6 test goal per test case**
- Test suite pairwise ottimizzale: 54/6 = **9 test case pairwise ottimale**

Covering Array

Client	Web Server	Payment	Database
FIREFOX	WEB SPHERE	MASTER CARD	DB/2
FIREFOX	.NET	AMEX	ORACLE
FIREFOX	APACHE	VISA	ACCESS
IE	WEB SPHERE	AMEX	ACCESS
IE	APACHE	MASTER CARD	ORACLE
IE	.NET	VISA	DB/2
OPERA	WEB SPHERE	VISA	ORACLE
OPERA	.NET	MASTER CARD	ACCESS
OPERA	APACHE	AMEX	DB/2


https://silodrome.com/

Ricapitolando

- Cerchiamo **combinazioni di parametri** che portano ad un **malfunzionamento** del sistema (efficacia)

- Spesso relativi a scenari “non previsti nei requisiti” → **Interaction Failures**

Interaction Failures

- Malfunzionamento che si manifesta solo con una **specifica combinazione** di parametri in input
  - ***t-way interaction:*** una combinazione di t parametri di interazione (input sistema/modulo)
    - 2-way: coppia di parametri (*pairwise*)
    - 3-way: tripla di parametri
    - ecc.
- Malfunzionamenti **latenti**, che possono sfuggire alle attività di controllo
  - Bug rilevati dopo il rilascio
  - “Hai 37 nuovi aggiornamenti da installare!” ☺

Esempio 2-way interaction

double peso(String gas, int pressione, int volume)

```
if (pressione < 10) {
    // fai qualcosa
    if (volume > 300) {
        codice difettoso!
        }
    else {
        codice corretto, nessun problema
    }
}
else {
    // fai qualcos'altro
}
```

Malfunzionamento rilevato se:
(pressione < 10)
&&
(volume > 300)

Test case scelti manualmente:

1. peso("H", **5**, 100)
2. peso("O", 10, 200)
3. peso("H", **5**, 300)
4. peso("O", 15, **400**)

Malfunzionamento non rilevato!

Se avessimo considerato tutte le coppie di valori (pressione, volume) lo avremmo rilevato

Test suite: 2-way interaction

Gas	Pressione
H	5
H	10
H	15
O	5
O	10
O	15

+

Gas	Volume
H	100
H	200
H	300
H	400
O	100
O	200
O	300
O	400

+

Pressione	Volume
5	100
5	200
5	300
5	400
10	100
10	200
10	300
10	400
15	100
15	200
15	300
15	400

2-way:

(2 x 3) + (3 x 4) + (2 x 4) = 26 test goal

Creare un test case separato per ogni test goal (settando gli altri parametri ad un valore casuale)

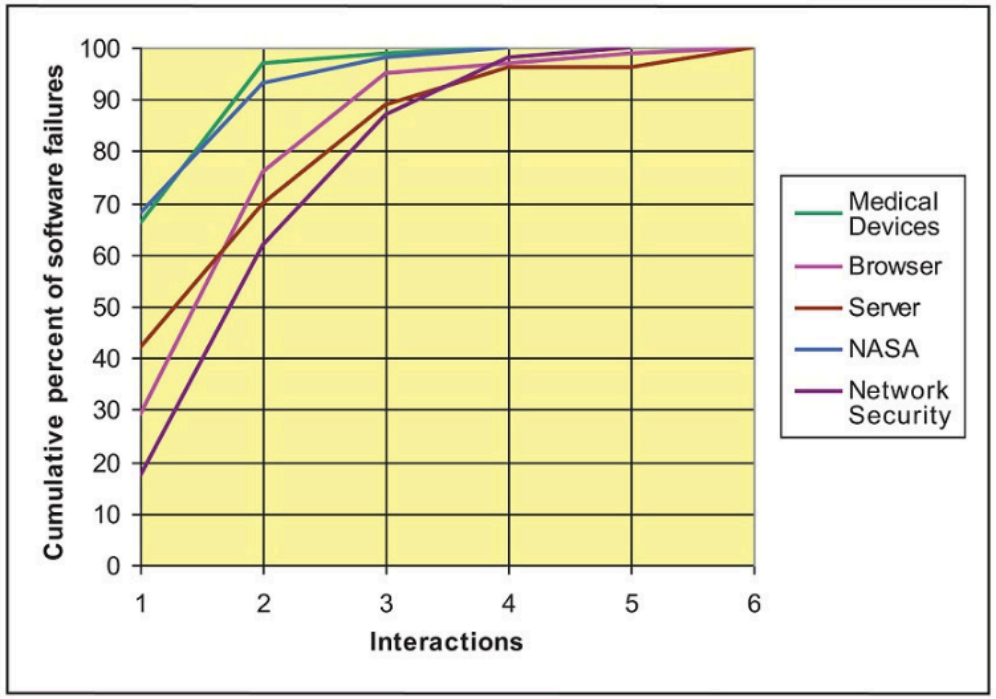
t-way VS esaustivo

- **Pairwise:** tutti i **malfunzionamenti** generati dall'**interazione di due (o meno)** parametri verranno **rilevati!**

- Aumentando il **grado di interazione** (ampiezza tuple) per aumentare la copertura

- Utile in presenza di **molti parametri**
  - Testing **esaustivo costoso** (10 interruttori = 1024 test case)
  - **Non realmente necessario**
    - malfunzionamenti generati dall'interazione di **pochi parametri**
    - **2-3 da già una buona copertura**
    - Ci si può fermare a **6-way**! (in molti casi)

Error Detection Rate



- Molti malfunzionamenti vengono attivati dal valore di 1 parametro
- La maggior parte (~90%) viene tipicamente scaturita dall'interazione di un numero ridotto di parametri in input (3 parametri)
- La quasi totalità viene tipicamente scaturita dall'interazione di 4-6 parametri

[https://ws680.nist.gov/publication/get_pdf.cfm?pub_id=910001]

Ottimizzazione combinatoriale

26 test goal VS 24 test case esaustivi…

Un singolo test case può coprire più test goals!

Es di Test case e test goal coperti

1.	peso(“H”, 5, 100)	→	{ (G:“H”, P:5);	(G:“H”, V:100);	(P:5, V:100) }
2.	peso(“O”, 10, 200)	→	{ (G:“O”, P:10);	(G:“O”, V:200);	(P:10, V:200) }
3.	peso(“H”, 5, 300)	→	{ (G:“H”, P:5);	(G:“H”, V:300);	(P:5, V:300) }
4.	peso(“O”, 15, 400)	→	{ (G:“O”, P:15);	(G:“O”, V:400);	(P:15, V:400) }

Ma dobbiamo coprirli tutti nel minor numero di test possibile. Quanto possiamo ottimizzare?

Covering Array

Test suite (lista di combinazioni) che **copre tutti i test goal** validi nel **minor numero possibile** di test case

Si cerca di ridurre le ripetizioni il più possibile

È un problema **combinatoriale**

Gas	Pressione	Volume
H	10	400
H	15	200
O	5	200
H	5	300
O	5	400
O	15	100
H	5	100
O	10	300
H	10	200
O	15	300
H	15	400
H	10	100

Problema NP-completo

- Consigliabile **discretizzare** le variabili definite in range continui molto ampi
  - partizionamento domini di input
- Problema non facilmente trattabile fino a pochi anni fa (solo pairwise)
- Oggi il problema **6-wise** è diventato trattabile

- Algoritmi **greedy**
  1. Aggiungere un test case alla volta
  2. Coprire più test goal possibili con il test case
  3. Evitando il più possibile ridondanze
     - test goal coperti più volte

CIT tool a confronto

TASK	IPO-S	ATGT	AETG	IPO	TCONFIG	CTS	JENNY	DDA	PICT
3^4	9	11	9	9	9	9	11	-	9
3^13	17	19	15	17	15	15	18	18	18
$4^{15}3^{17}2^{29}$	32	38	41	34	40	39	38	35	37
$4^13^{39}2^{35}$	23	27	28	26	30	29	28	27	27
2^{100}	10	12	10	15	14	10	16	15	15
10^{20}	220	267	180	212	231	210	193	201	210

Problema 2^{100} risolto con un covering array pairwise di 16 test case (nel peggiore dei casi) a fronte di 19800 test goal

$$2^2\binom{100}{2} = 19800$$

Coffee4j

- framework java per il combinatorial testing e il fault characterization
- Basato su IPOG
- Supporta la generazione vincolata
- Supporta la fault localization
- Generazione automatica di unit test per JUnit 5
- Sito al progetto:
  - https://coffee4j.github.io

Coffee4j

```
class TestPeso {

    private static InputParameterModel model() {
        return inputParameterModel("peso-model")
                .strength(2)
                .parameters(
                        parameter("Gas").values("H", "O"),
                        parameter("Pressione").values(5, 10, 15),
                        parameter("Volume").values(100, 300, 400)
                ).errorConstraints(
                        constrain("Pressione", "Volume").by((Integer pressione, Integer volume) -> {
                            if (pressione == 10 && volume == 300)
                            return false;
                            return true;
                        })
                ).build();
    }

    @CombinatorialTest
    @Generator({Ipog.class})
    @ModelFromMethod("model")
    void testPeso(String gas, int pressione, int volume) {
        System.out.println("Gas:" + gas + " Pressione:" + pressione + " Volume:" + volume);
    }
}
```

Input Parameter Model

```
class TestPeso {

    private static InputParameterModel model() {
        return inputParameterModel("peso-model")
                .strength(2)
                .parameters(
                        parameter("Gas").values("H", "O"),
                        parameter("Pressione").values(5, 10, 15),
                        parameter("Volume").values(100, 300, 400)
                ).build();
    }

    @CombinatorialTest
    @Generator({Ipog.class})
    @ModelFromMethod("model")
    void testPeso(String gas, int pressione, int volume) {
        System.out.println("Gas:" + gas
                  + " Pressione:" + pressione
                  + " Volume:" + volume);
    }
}
```

Test Suite Pairwise Ottimizzata

1. peso(“H”, 10, 400)
2. peso(“H”, 15, 200)
3. peso(“O”, 5, 200)
4. peso(“H”, 5, 300)
5. peso(“O”, 5, 400)
6. peso(“O”, 15, 100)
7. peso(“H”, 5, 100)
8. peso(“O”, 10, 300)
9. peso(“H”, 10, 200)
10. peso(“O”, 15, 300)
11. peso(“H”, 15, 400)
12. peso(“H”, 10, 100)

Covering array
composto da 12 test case

24 test case esaustivi (3-way)
26 test goal pairwise (2-way)
12 test case pairwise (ottimale)

Vincoli di enumerazione

- Per imporre delle condizioni sulla generazione delle combinazioni di parametri
  - Scartare in partenza quelle che non verranno mai fornite in input al sistema (es. regole sintattiche)

- Es. (fantasioso!)
  - Se la pressione è a 10 il volume non può essere a 300 e viceversa, indipendentemente dal gas

Vincoli di Enumerazione

```
class TestPeso {

    private static InputParameterModel model() {
        return inputParameterModel("peso-model")
                .strength(2)
                .parameters(
                        parameter("Gas").values("H", "O"),
                        parameter("Pressione").values(5, 10, 15),
                        parameter("Volume").values(100, 300, 400)
                ).errorConstraints(
                        constrain("Pressione", "Volume").by((Integer pressione, Integer volume) -> {
                            if (pressione == 10 && volume == 300)
                              return false;
                            return true;
                        })
                ).build();
    }

    @CombinatorialTest
    @Generator({Ipog.class})
    @ModelFromMethod("model")
    void testPeso(String gas, int pressione, int volume) {
        System.out.println("Gas:" + gas + " Pressione:" + pressione + " Volume:" + volume);
    }
}
```


KEYFORGE
CALL OF THE ARCHONS
A UNIQUE DECK GAME BY
RICHARD GARFIELD
KEYFORGE
CALL OF THE ARCHONS
CONTAINS A UNIQUE
37 CARD
ARCHON DECK
A UNIQUE DECK GAME BY
RICHARD GARFIELD


Smart Card
5678 9876 5432
1234
MONTH/YEAR
VALID THRU 12/16
JOHN A SMITH

peso.asm con vincoli

```
asm peso
import StandardLibrary
signature:

enum domain Gas = {GG_H|GG_O}
enum domain Pressione = {PP_5|PP_10|PP_15}
enum domain Volume = {VV_100|VV_200|VV_300|VV_400}

dynamic monitored g : Gas
dynamic monitored p : Pressione
dynamic monitored v : Volume

definitions:
invariant inv_PV over p,v : p=PP_10 or v=VV_300
invariant inv_O over g,v,p : g=GG_O implies
                                (p=PP_10 or p=PP_15) and
                                (v=VV_300 or v=VV_400)
default init s1:
```

Risultato (1/2)

```
INFO - 2wise coverage has 26tps
INFO - pair_1_1_2_3 (g = GG_O) and (p = PP_5) is infeasible
INFO - pair_1_1_3_3 (g = GG_O) and (v = VV_200) is infeasible
INFO - pair_1_1_3_4 (g = GG_O) and (v = VV_100) is infeasible
INFO - pair_2_1_3_1 (p = PP_15) and (v = VV_400) is infeasible
INFO - pair_2_1_3_3 (p = PP_15) and (v = VV_200) is infeasible
INFO - pair_2_1_3_4 (p = PP_15) and (v = VV_100) is infeasible
INFO - pair_2_3_3_1 (p = PP_5) and (v = VV_400) is infeasible
INFO - pair_2_3_3_3 (p = PP_5) and (v = VV_200) is infeasible
INFO - pair_2_3_3_4 (p = PP_5) and (v = VV_100) is infeasible
INFO - violated: 17 covered: 0 infeasible: 9
```

Su 26 test goal (paiwise) 9 sono stati scartati perché violavano i due vincoli di enumerazione

Risultato (2/2)

```
INFO - 8 test results are generated
INFO - reducing test suite
INFO - test suite has not been reduced
1 : test@collected1 -> [ v=VV_300 p=PP_10 g=GG_H ] : NORMAL
2 : test@collected2 -> [ v=VV_200 p=PP_10 g=GG_H ] : NORMAL
3 : test@collected3 -> [ v=VV_100 p=PP_10 g=GG_H ] : NORMAL
4 : test@collected4 -> [ v=VV_300 p=PP_15 g=GG_O ] : NORMAL
5 : test@collected5 -> [ v=VV_400 p=PP_10 g=GG_O ] : NORMAL
6 : test@collected6 -> [ v=VV_400 p=PP_10 g=GG_H ] : NORMAL
7 : test@collected7 -> [ v=VV_300 p=PP_5 g=GG_H ] : NORMAL
8 : test@collected8 -> [ v=VV_300 p=PP_15 g=GG_H ] : NORMAL
```

1. peso(“H”, 10, 300)
2. peso(“H”, 10, 200)
3. peso(“H”, 10, 100)
4. peso(“O”, 15, 300)
5. peso(“O”, 10, 400)
6. peso(“H”, 10, 400)
7. peso(“H”, 5, 300)
8. peso(“H”, 15, 300)

Considerazioni

- L'**ottimizzazione combinatoriale** ha permesso la gestione e l'ottimizzazione del numero di test case
- Garanzie sul livello di **copertura degli stati** del verificatore, grazie all'uso di strumenti formali
- Creazione **automatica e completa** della test suite grazie al model checking su un modello formale delle specifiche semantiche del linguaggio
- **Dettagli formali nascosti** ai tester: basta fornire loro la test suite
- Applicabile alla validazione di **generici verificatori Java**
- Strumento **valido** ed **economico** per l'individuazione di bug critici **durante lo sviluppo**
- Efficacia dimostrata grazie all'individuazione di non conformità su verificatori **già rilasciati**