High Order Mutation Testing in PIT-HOM

Università degli Studi di Milano

Verifica e Convalida del Software

Anno accademico 2020/21

Mattia Marchionna

Riassunto

- Breve ripasso su testing mutazionale di primo ordine
- Testing mutazionale di ordine superiore
- Presentazione PIT-HOM
- Presentazione LittleDarwin

High Order Mutation Testing

Ricapitolando: testing mutazionale (primo ordine)

Nel testing mutazionale, dato un programma *p*, vengono generati una serie di programmi difettosi *p'* (mutanti) iniettando errori nel programma originale *p*, al fine di valutare la qualità dei test, in termini di capacità di trovare il difetto iniettato.

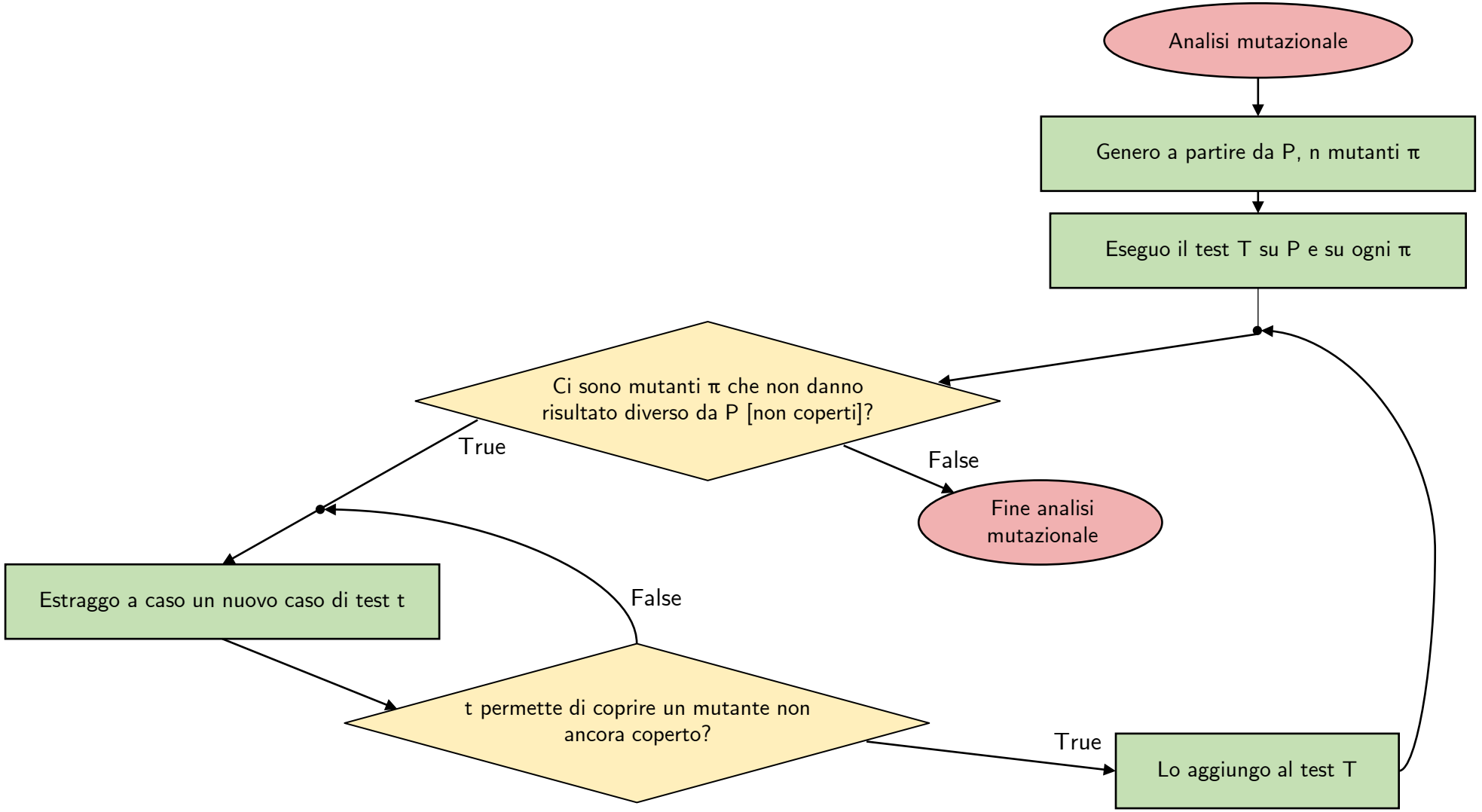
Un mutante viene generato da una singola piccola modifica

- mutante del primo ordine

Programma *p*	Programma *p'*

...
while(a > 0) {
  a = b - c;
  c++;
}
...
	...
	while(a >= 0) {
	  a = b - c;
	  c++;
	}
	...

Ricapitolando: procedura



High Order Mutation Testing (HOMT)

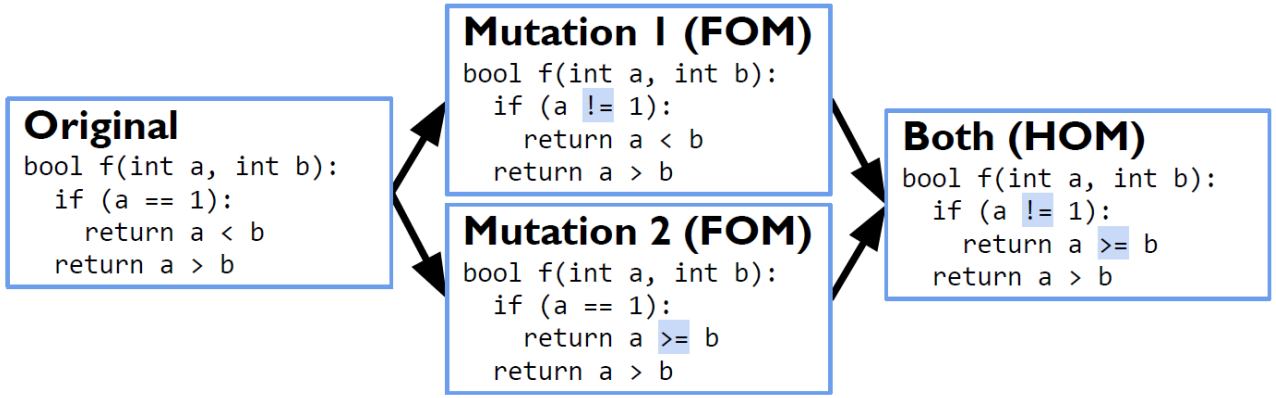
First Order Mutation Testing (FOMT)

I mutanti di primo ordine (FOM) denotano spesso difetti banali che possono essere facilmente uccisi.

High Order Mutation Testing (HOMT)

I mutanti (HOM) vengono generati applicando operatori di mutazione più di una volta

Un HOM si avvicina maggiormente ad un errore reale



Problema dell'HOM testing

Un HOM viene costruito combinando diversi FOM

- Numero degli HOM combinatorio al numero di mutanti del primo ordine
- Esplosione esponenziale del numero degli HOM
- considerato in passato così computazionalmente costoso da essere poco pratico

Numero FOM	Numero HOM
$\sum_{i=0}^{n} m_i$	$\sum_{i=2}^{n} \binom{i}{n} m^i$

Per ciascun HOM:

- $p_{1..n}$ i posti che possono essere mutati
- $m_{1...n}$ il numero di modifiche che possono essere applicate a ciascun posto $p_{1..n}$

Classificazione degli HOM

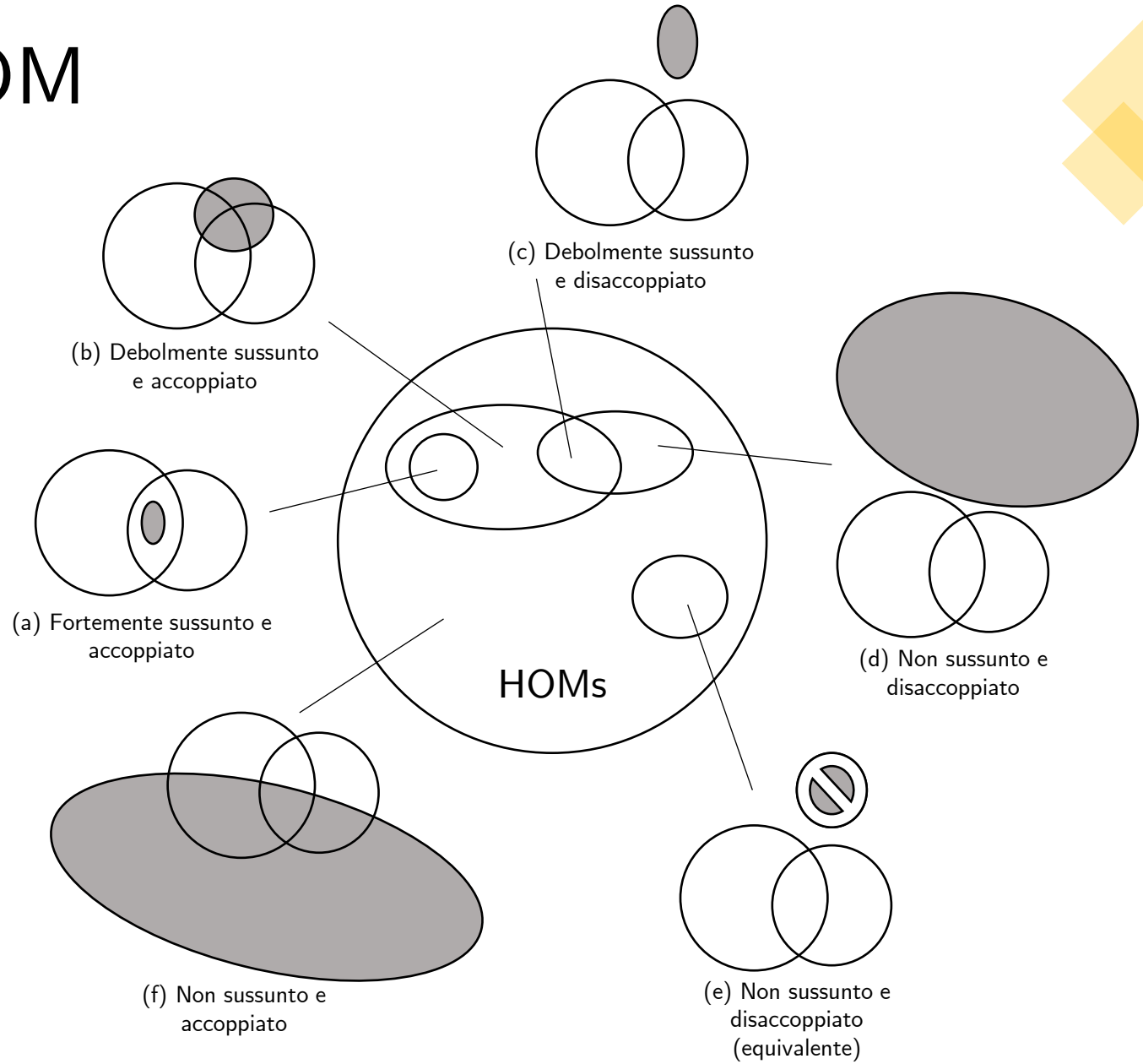
In base al modo in cui sono accoppiati e sussunti

HOM accoppiato

- se un insieme di test che uccide i FOM contiene anche casi di test che uccidono l'HOM

HOM sussunto

- più difficili da uccidere rispetto alle FOM costituenti



HOM interessanti

1. HOM disaccoppiato
   - HOM che viene ucciso da casi di test differenti da quelli che uccidono i FOM che lo compongono

2. HOM fortemente sussunto
   - ucciso da un sottoinsieme della suite di test che uccide tutti i FOM che lo compongono

3. HOM sottile/delicato
   - HOM che non viene ucciso da nessun test presente nella suite di test

Vantaggi di HOMT

- Maggiore sottigliezza
  - HOM sussunti denotano difetti che test più elaborati potrebbero non rivelare
- Minore sforzo
  - HOMT riduce il numero di mutanti considerati aumentandone contemporaneamente la qualità
  - Meno mutanti (ma migliori) significano meno casi di test
- Numero ridotto di mutanti equivalenti

  Densità relativamente bassa di mutanti equivalenti
  - il 10% dei mutanti del primo ordine risulta equivalente
  - circa l'1% dei mutanti del secondo ordine risulta equivalente

Algoritmi di ricerca (1)

- A causa dell'elevato numero di HOM, il costo per trovare HOM di valore potrebbe rivelarsi estremamente costoso
- L'utilizzo di una normale ricerca non indirizzata non è abbastanza efficiente per trovare HOM interessanti

Algoritmi di ricerca adottati da HOMT

- Greedy
- Genetici
- Hill-climbing

Algoritmi di ricerca (2)

Per misurare l'idoneità dell'HOM è necessario calcolare un valore che rappresenti la facilità con cui esso possa essere ucciso

$$fragility(\{M_1, \ldots, M_n\}) = \frac{|\bigcup_{i=1}^{n} kill(M_i)|}{|T|}$$

$kill(\{M_1, \ldots, M_n\})$ ritorna l'insieme dei casi di test che uccidono i mutanti M1, ..., M2.

Valore di fragilità compreso tra 0 ed 1

- Uguale a 0: non esiste alcun caso di test in grado di uccidere l'HOM
- Tra 0 ed 1: HOM valutato come più debole
- Uguale ad 1: HOM così debole che può essere ucciso da uno qualsiasi dei casi di test

$$fitness(M_{1\ldots n}) = \frac{fragility(\{M_{1\ldots n}\})}{fragility(\{F_1, \ldots, F_n\})}$$

Sia $M_{1\ldots n}$ un HOM composta dai FOM $F_1, \ldots, F_n$.

Valore di fitness compreso tra 0 ed 1

- Uguale a 0: viene considerato come un potenziale HOM
- Valore che tende a 0: l'HOM diventa gradualmente più forte dei suoi FOM costituenti
- Maggiore di 1: HOM più debole dei FOM costituenti

Miti del testing mutazionale

- I programmatori sono generalmente persone competenti e commettono pochi errori
- I guasti complessi sono accoppiati a quelli semplici in modo tale che i test che trovano tutti quelli semplici rileveranno anche un'alta percentuale di guasti complessi

Ipotesi del programmatore competente

Effetto di accoppiamento

PIT-HOM Demo

Processo di mutazione PIT

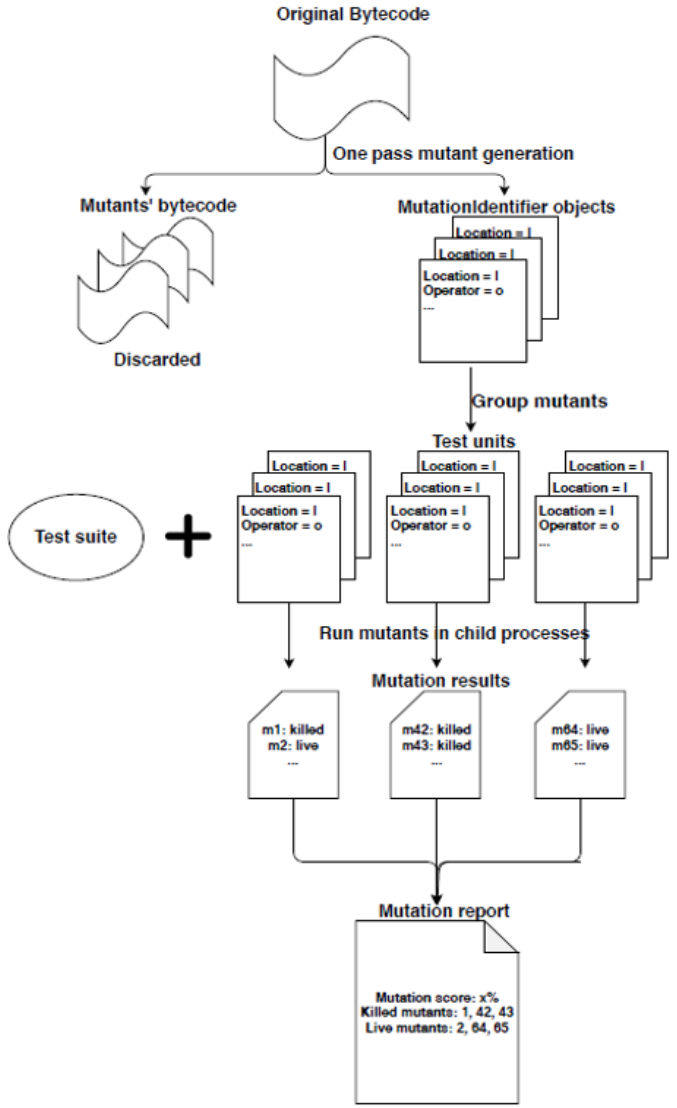
`MutationIdentifier`

- Oggetto che identifica in modo univoco il mutante (posizione del mutante + operazione di mutazione)
- Posizione: classe + metodo + numero di istruzione

`MutationDetails`

- Collegato all'oggetto `MutationIdentifier`
- Contiene informazioni aggiunte, come i test da eseguire contro il mutante

Poca memoria utilizzata per memorizzare i mutanti



Processo di mutazione PIT-HOM (1)

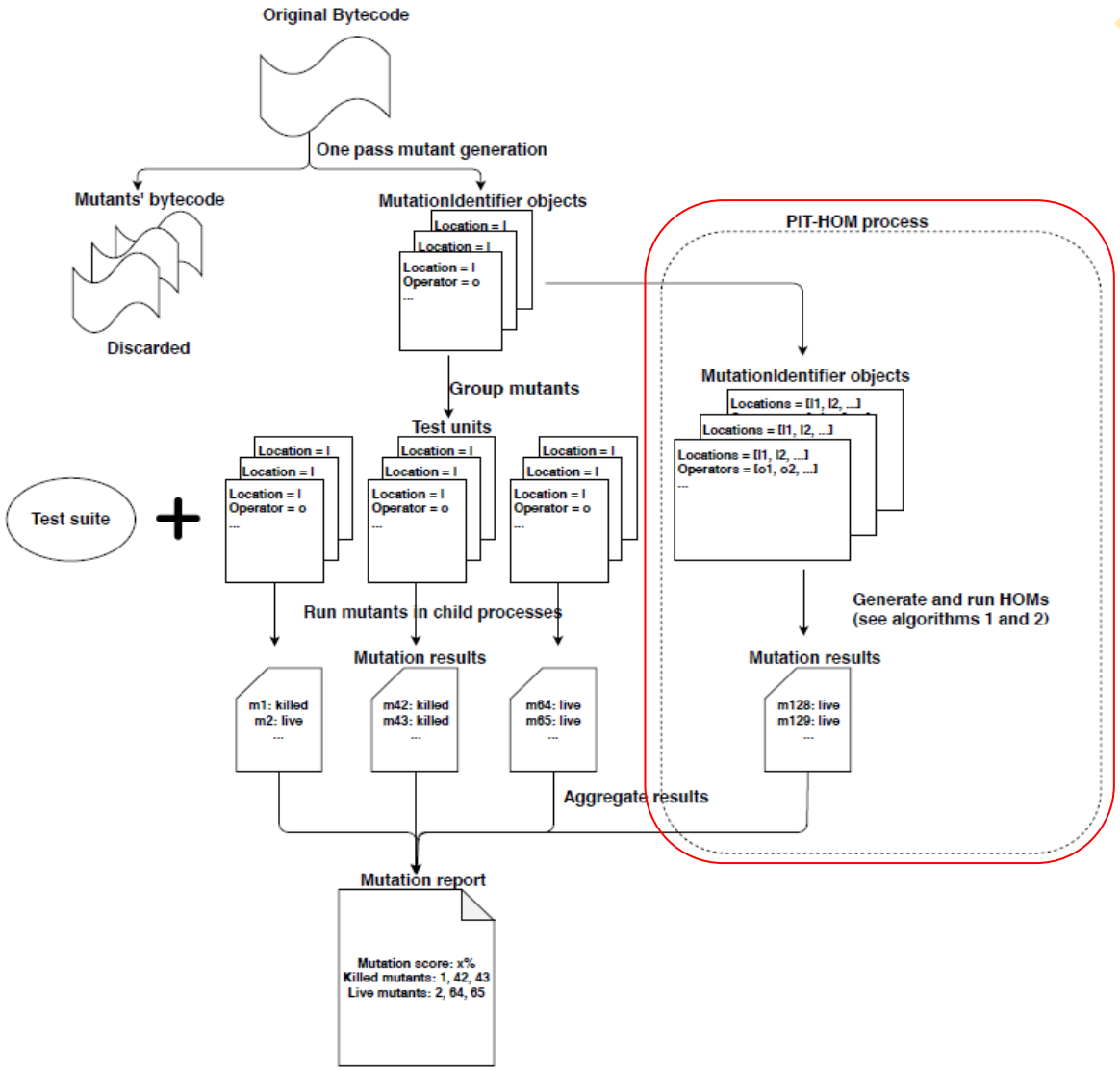
Mantiene gli stessi vantaggi in termini di prestazione

`MutationIdentifier`

- Contiene un elenco di posizioni e di operatori di mutazione

`MutationDetails`

- Qualsiasi test che copre uno qualsiasi dei componenti FOM è considerato coprire l'HOM
- Sono considerati tutti i test che possono potenzialmente uccidere un HOM



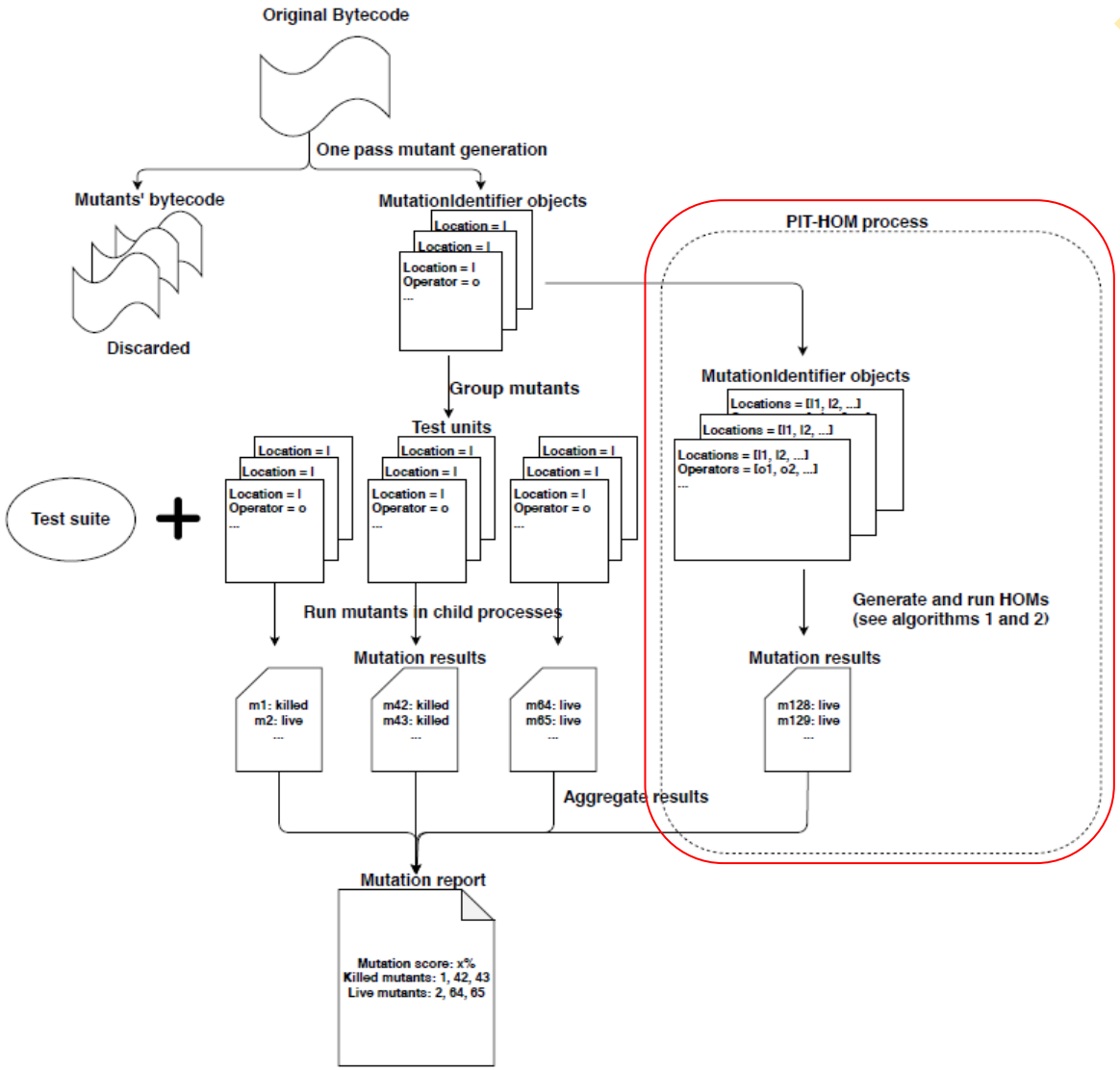
Processo di mutazione PIT-HOM (2)

Problema

- Numero HOM combinatorio con quello dei FOM
- Tenere in memoria principale tutti i possibili HOM diventa problematico

Soluzione adottata

- Utilizzare uno dei due processi di analisi delle mutazioni:
  - Streaming
  - Batch-streaming



Streaming

Per ogni classe in analisi viene eseguito il passaggio classico di PIT

Le combinazioni di FOM vengono trovate utilizzando la funzione `findNextCombination`

1. Input: elenco dei FOM e ordine di destinazione
2. Output: valido HOM

Mutanti non memorizzati contemporaneamente nella memoria principale

Ogni mutante viene elaborato in nuova JVM (overhead significativo)

Input:
classesToAnalyse: List of classes that should be mutated
ordersToRun: List of mutation orders that should be run

```
for class in classesToAnalyse do
  foms <- MutationScore.findMutants(class)
  if 1 in ordersToRun do
    for mutant in foms do
      run(new TestUnit(mutant))
    end for
  end if
  for order in ordersToRun do
    hom <- findNextCombination(foms, order)
    while hom != null do
      run(new TestUnit(mutant))
      hom <- findNextCombination(foms)
    end while
  end for
end for
```

Batch-Streaming

Fornisce un bilanciamento tra il normale processo di PIT e il metodo di streaming

1. HOM creati e memorizzati fino a quando non ne vengono creati un numero impostato
2. Batch di HOM trasformato in unità di test ed elaborata in una JVM

Numero ridotto di JVM create rispetto al metodo di streaming

Numero di mutanti da mantenere in memoria comunque ragionevole

Input:
classesToAnalyse: List of classes that should be mutated
ordersToRun: List of mutation orders that should be run

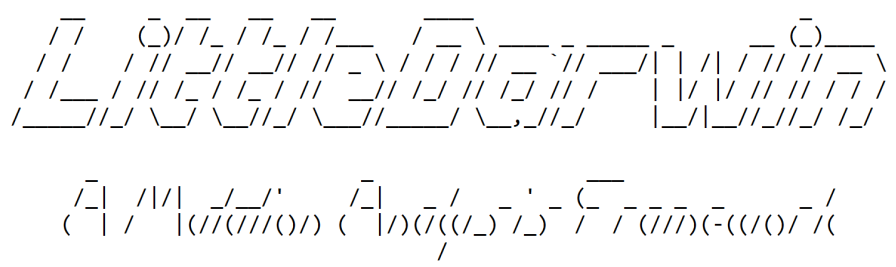
```
for class in classesToAnalyse do
  foms <- MutationScore.findMutants(class)
  if 1 in orderToRun then
    run(makeTestUnits(mutants, maxTestUnitSize))
  end if
  for order in ordersToRun do
    hom <- findNextCombination(foms, order)
    homsToRun <- {}
    while hom != null do
      homsToRun.add(hom)
      if homsToRun.size() == 10000 then
        run(makeTestUnit(homsToRun, maxTestUnitSize))
        homsToRun <- {}
      end if
      hom <- findNextCombination(foms)
    end while
    if homsToRun.size() > 0 then
      run(makeTestUnits(homsToRun, maxTestUnitSize))
    end if
  end for
end for
```

https://github.com/ucd-csl/pitest

Importare PIT-HOM con Maven

```
<plugin>
      <groupId>org.pitest</groupId>
      <artifactId>pitest-maven</artifactId>
      <version>1.5.1-HOM</version>
      <dependencies>
            <dependency>
                  <groupId>org.pitest</groupId>
                  <artifactId>pitest-junit5-plugin</artifactId>
                  <version>0.12</version>
            </dependency>
      <dependencies>
      <configuration>
            <targetClasses>
                  <param>it.unimi.di.vec.*</param>
            </targetClasses>
            <targetTests>
                  <param>it.unimi.di.vec.*</param>
            </targetTests>
            <hom>2</hom>
            <mutators>
                  <mutator>STRONGER</mutator>
            </mutators>
            <mutantProcessingMethod>stream-batch</mutantProcessingMethod>
      </configuration>
</plugin>
```

LittleDarwin



https://github.com/aliparsai/LittleDarwin

LittleDarwin

LittleDarwin is a mutation testing tool designed primarily with the premise of easy deployment in complex systems; in order to provide mutation testing where other tools fail.

- Scritto in Python
- Progettato pensando alla semplicità con l'obiettivo di ottenere una maggiore flessibilità
- A differenza di PIT, muta il codice sorgente piuttosto che il bytecode
- Vantaggi
  - Mutanti facili da definire e inventare
  - Mutanti mimano chiaramente tipi di errori
- Svantaggio
  - Approccio lento

Processo

Articolato in due fasi:

- Fase di mutazione (algoritmo 1)
- Fase di esecuzione del test (algoritmo 2)

Algoritmo 1	Algoritmo 2

Input : Java source files
Output: Mutated Java source files
 	
queue <- all Java source files;
while queue != ∅ **do**
  srcFile <- queue.pop();
  mutants[srcFile] <- *mutate*(srcFile);
end
return mutants;
	Input : Mutated Java source files
	Output: Mutation Testing Report
	
	if executeTestSuite() *is successful* **then**
	  foreach srcFile **do**
	  queue <- mutants[srcFile];
	  backup(srcFile);
	  while queue != ∅ **do**
	  mutantFile <- queue.pop();
	  replace(srcFile,mutantFile);
	  result[mutantFile] <- executeTestSuite();
	  end
	  restore(srcFile);
	  Generate report for srcFile;
	  end
	  Generate overall report;
	end
	return reports;

Componenti

JavaRead

- Fornisce metodi per eseguire operazioni di input/output su file Java
- LittleDarwin utilizza questo componente per leggere i file sorgente e riscrivere i mutanti su disco

JavaParse

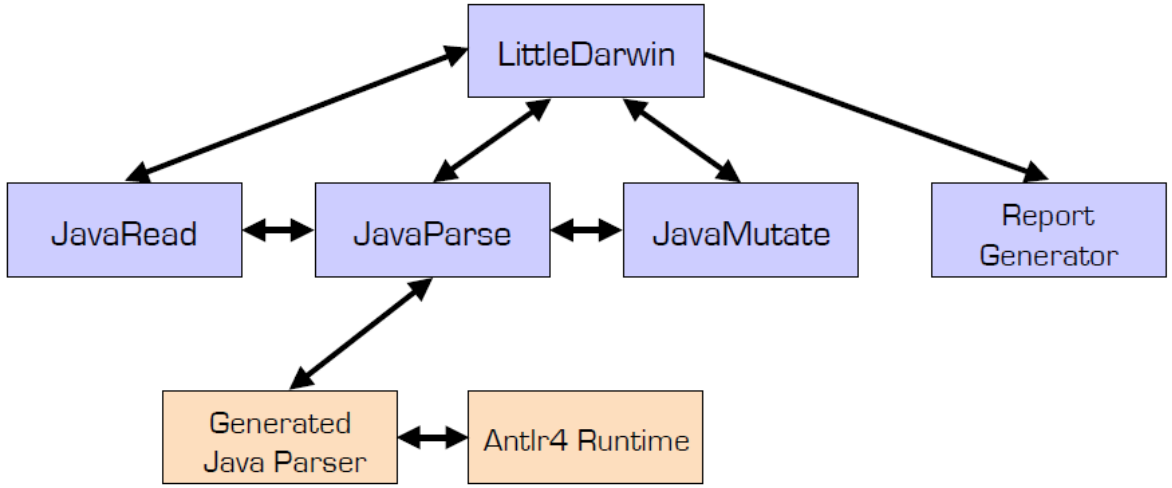
- Questo componente analizza i file Java in un albero sintattico astratto (AST)
- Necessario per produrre mutanti validi e compilabili.
- Fornisce anche la funzionalità per stampare l'albero modificato su un file Java

JavaMutate

- Questo componente manipola l'AST creato dal parser
- Permette di eseguire la mutazione sull'albero stesso

ReportGenerator

- Genera report HTML per ogni file



Esempio

```
python -m littledarwin -m -b -p src/main/ --higher-order=2 -t . -c gradle,clean,test
```

Esegue LittleDarwin sui file sorgenti del progetto per generare mutanti ed esegue la suite di test sui mutanti risultanti utilizzando i comandi specificati.

-m, --mutate
: attiva la fase di mutazione

-b, --build
: attiva la fase di build

-p, --path=SOURCEPATH
: percorso dei file sorgente

-t, --build-path=BUILDPATH
: percorso per effettuare la build della working directory

--higher-order=HIGHERORDER
: ordine dei mutanti

Report

- Un report per ogni file sorgente
- Un report complessivo per il progetto

LittleDarwin Mutation Coverage Report

Project Summary

Number of Files	Mutation Coverage	
1	90.0	9/10

Breakdown by File

Name	Mutation Coverage	
java\it\unimi\di\vec\littledarwin\Triangle.java	90.0%	9/10

Report generated by LittleDarwin 0.10.5

LittleDarwin Mutation Coverage Report

File Summary

Number of Mutants	Mutation Coverage	
10	90.0%	9/10

Aggregate Report

Detailed List

Survived Mutant	Build Output	Killed Mutant	Build Output
1.java	1.txt	2.java	2.txt
		3.java	3.txt
		4.java	4.txt
		5.java	5.txt
		6.java	6.txt
		7.java	7.txt
		8.java	8.txt
		9.java	9.txt
		10.java	10.txt

Report generated by LittleDarwin 0.10.5

Repo

- Un re
- Un re

```
/* LittleDarwin generated order-2 mutant
mutant type: ConditionalOperatorReplacement
----> before:       if (((a + b) < c) || ((a + c) < b) || ((b + c) < a)) {
----> after:        if (((a + b) < c) && ((a + c) < b) || ((b + c) < a)) {
----> line number in original file: 28
----> mutated node: 394

mutant type: ConditionalOperatorReplacement
----> before:     if ((a <= 0) || (b <= 0) || (c <= 0)) {
----> after:      if ((a <= 0) && (b <= 0) || (c <= 0)) {
----> line number in original file: 14
----> mutated node: 185
*/
…
```

L

Project Summary

Number of Files	Mutation Coverage	
1	90.0	9/10

Breakdown by File

Name	Mutation Coverage	
java\it\unimi\di\vec\littledarwin\Triangle.java	90.0%	9/10

Report generated by LittleDarwin 0.10.5

LittleDarwin Mutation Coverage Report

3.java	3.txt
4.java	4.txt
5.java	5.txt
6.java	6.txt
7.java	7.txt
8.java	8.txt
9.java	9.txt
10.java	10.txt

Report generated by LittleDarwin 0.10.5

Report

- Un report per ogni file sorgente
- Un report complessivo per il progetto

LittleDarwin Mutation Coverage Report

Project Summary

Number of Files	Mutation Coverage	
1	90.0	9/10

Breakdown by File

Name	Mutation Coverage	
java\it\unimi\di\vec\littledarwin\Triangle.java	90.0%	9/10

Report generated by LittleDarwin 0.10.5

LittleDarwin Mutation Coverage Report

File Summary

Number of Mutants	Mutation Coverage	
10	90.0%	9/10

Aggregate Report

Detailed List

Survived Mutant	Build Output	Killed Mutant	Build Output
1.java	1.txt	2.java	2.txt
		3.java	3.txt
		4.java	4.txt
		5.java	5.txt
		6.java	6.txt
		7.java	7.txt
		8.java	8.txt
		9.java	9.txt
		10.java	10.txt

Report generated by LittleDarwin 0.10.5

Report

- Un report pe
- Un report co

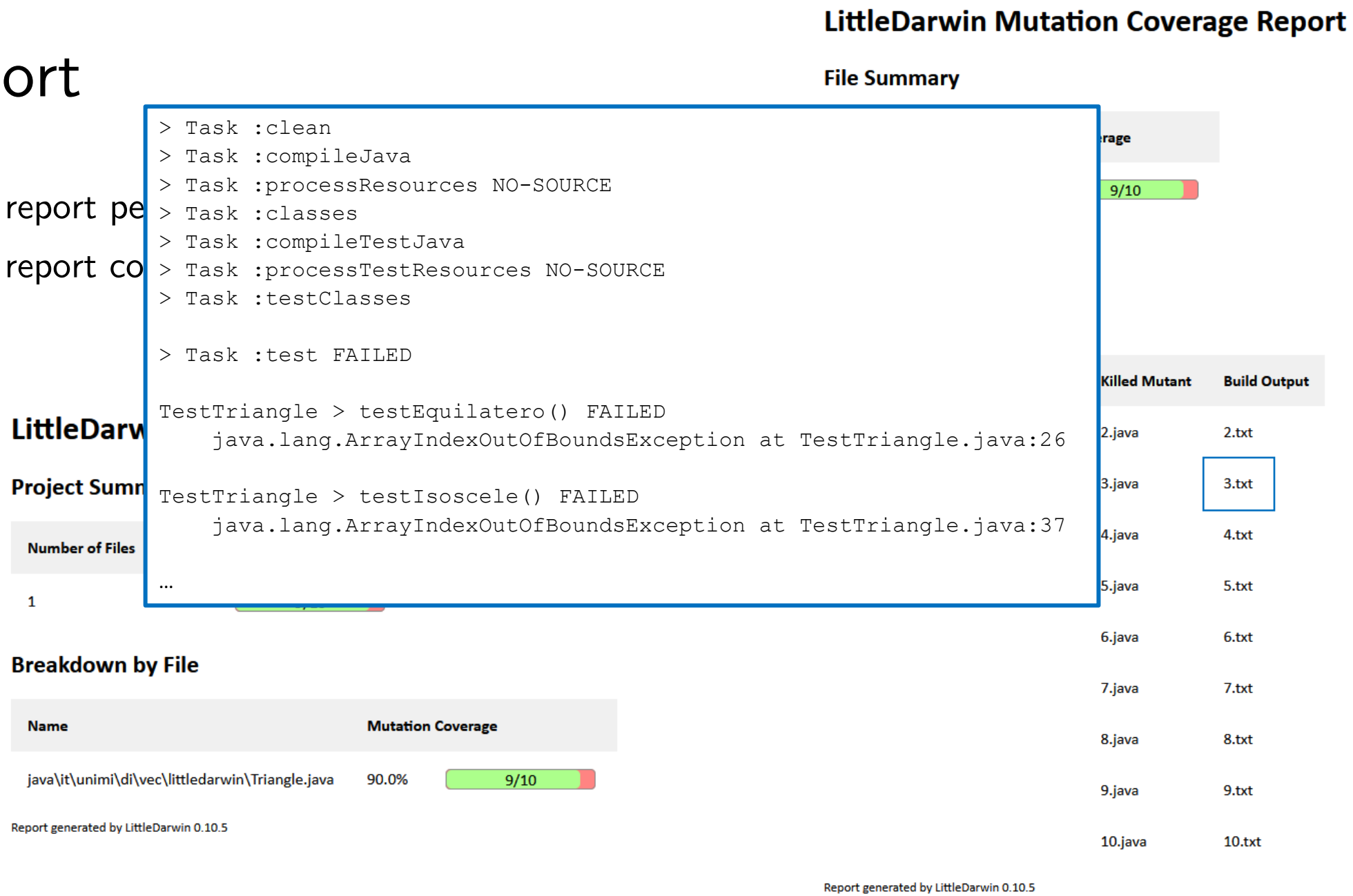
```
> Task :clean
> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses

> Task :test FAILED

TestTriangle > testEquilatero() FAILED
    java.lang.ArrayIndexOutOfBoundsException at TestTriangle.java:26

TestTriangle > testIsoscele() FAILED
    java.lang.ArrayIndexOutOfBoundsException at TestTriangle.java:37

...
```



LittleDarwin Mutation Coverage Report

File Summary

erage

9/10

Killed Mutant | Build Output

2.java | 2.txt

3.java | 3.txt

4.java | 4.txt

5.java | 5.txt

6.java | 6.txt

7.java | 7.txt

8.java | 8.txt

9.java | 9.txt

10.java | 10.txt

Report generated by LittleDarwin 0.10.5

LittleDarw

Project Summ

Number of Files

1

Breakdown by File

Name | Mutation Coverage

java\it\unimi\di\vec\littledarwin\Triangle.java | 90.0% | 9/10

Report generated by LittleDarwin 0.10.5

Grazie per l'ascolto