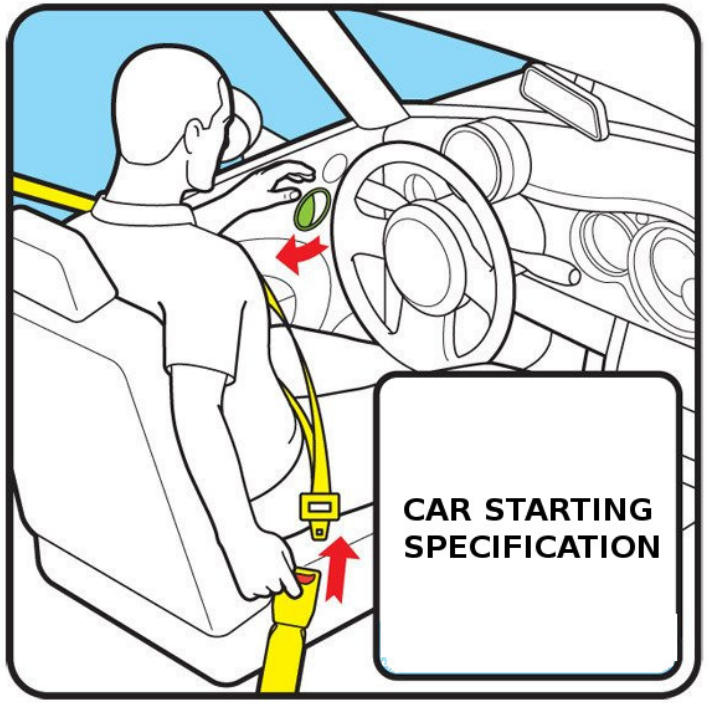
Mutation based testing : Benefits, issues and demo

P. Arun Babu, Resiltech

Table of contents

Introduction



/* Program */	/* Mutant */
if (door_is_closed && seat_belt_is_on)	if (door_is_closed || seat_belt_is_on)
{	{
car_can_start = true ;	car_can_start = true ;
}	}
else	else
{	{
car_can_start = false ;	car_can_start = false ;
}	}

Demo

Demo with mutate.py

1. A simple python script which mutates C programs, which I wrote during my Ph.D
2. Works on simple substitution of strings, and can be easily extended by adding new mutant operators.
3. Can be extended for other programming languages too.

Demo

Demo with mutate.py

1. A simple python script which mutates C programs, which I wrote during my Ph.D
2. Works on simple substitution of strings, and can be easily extended by adding new mutant operators.
3. Can be extended for other programming languages too.

Software under test

Comments are removed, code is indented or beautified

↓

Software run against input.txt

↓

expected-output.txt

Demo

A program which takes a number n and prints a triangle like this:

```
    1
   232
  34543
 4567654
567898765
```

This is a simple application and produces different results for different input.

Demo

For each mutant

Mutant run against input.txt

↓

mutant's output.txt

↓

Mutant's output.txt = expected-output.txt ?

↙ ↘

Yes: Mutant could not be killed **No:** Mutant killed

Demo

For each mutant

Mutant run against input.txt

↓

mutant's output.txt

↓

Mutant's output.txt = expected-output.txt ?

↙ ↘

Yes: Mutant could not be killed **No:** Mutant killed

Software is tested after inducing a fault intentionally

- Inducing a bug in source code obviously would show failure !
- As safety-critical software is expected to be well tested.
- **Then why mutation testing ?**

Table of contents

Table of contents

1. To know if adequate testing has been performed

Quality of testcases is:

$$\begin{cases} \textit{Good} & \text{— } \textit{if they catch most of the faults} \\ & \quad (\textit{ie} : \textbf{kill } \textit{most of the mutants}) \\ \\ \textit{Poor} & \text{— } \textit{if they do not} \end{cases}$$

1. To know if adequate testing has been performed

Quality of testcases is:

$$\begin{cases} \textit{Good} & -\ \textit{if they catch most of the faults} \\ & \quad (\textit{ie} : \textbf{kill}\ \textit{most of the mutants}) \\ \textit{Poor} & -\ \textit{if they do not} \end{cases}$$

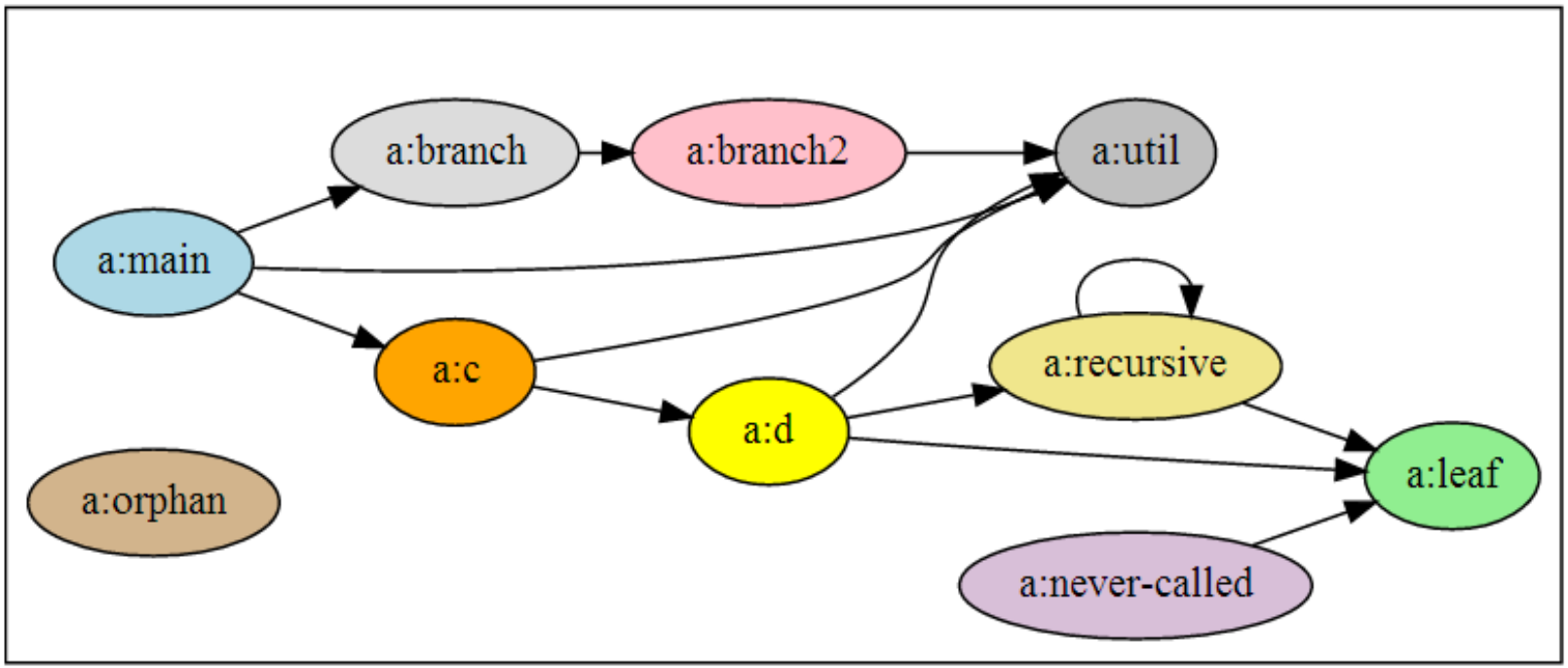
An indication of test adequacy is the mutation score :

$$\frac{\textit{No. of mutants killed}(K)}{\textit{No. of mutants generated}(G) - \textit{No. of equivalent mutants}(E)}$$

Value is between : 0 (poor) – 1 (good)

Table of contents

2. Identifying Safety/Mission/Security critical code



Testers and programmers are 2 independent group

Tester needs to identify which part of the code is critical.

During mutation testing if inducing fault in a module causes :

1. Safety violation : Safety-critical code
2. Security violation : Security-critical code

Table of contents

3. Useful in testing diversified N-version software

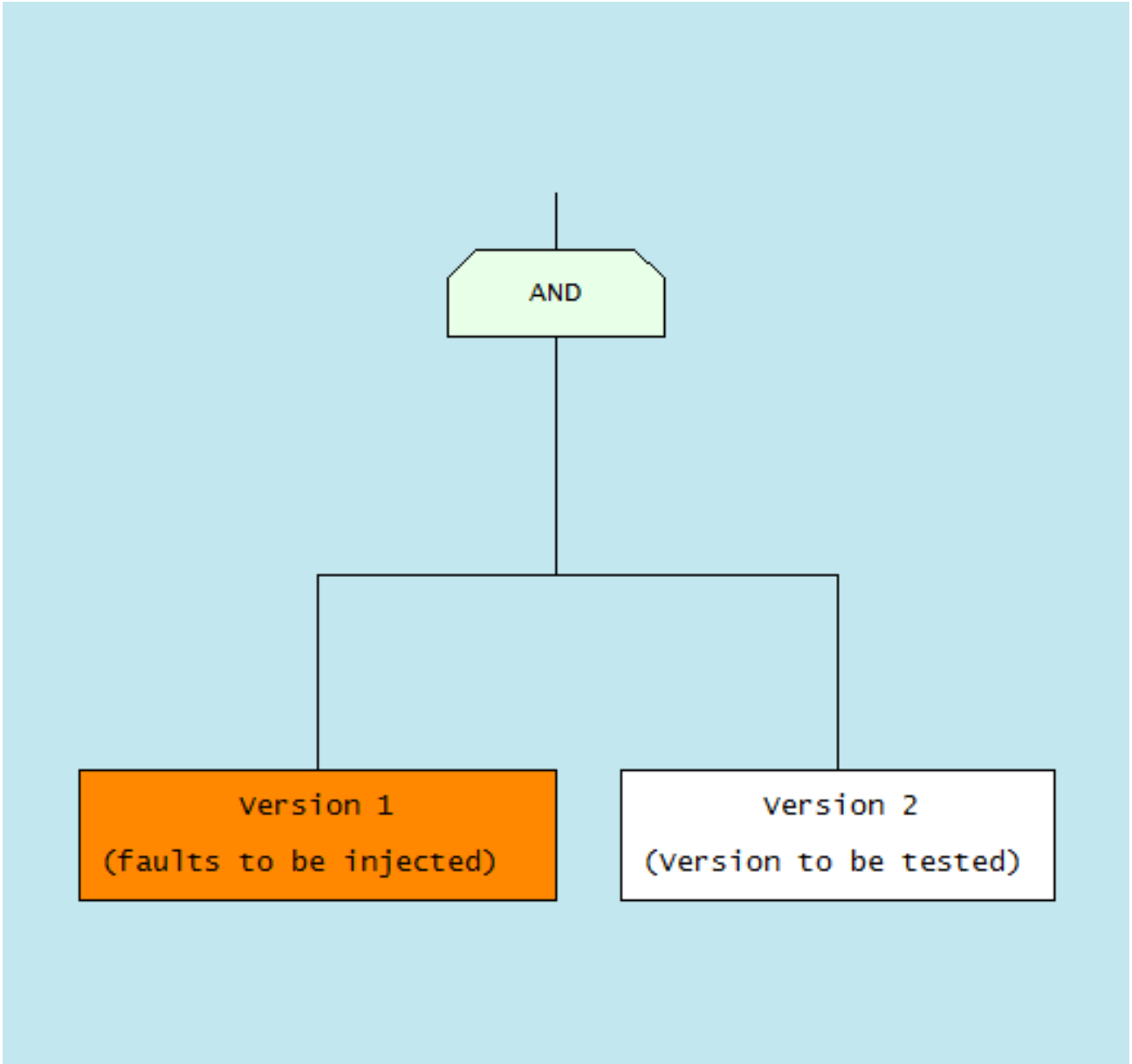


Table of contents

4. Testing system diagnosis/monitoring software

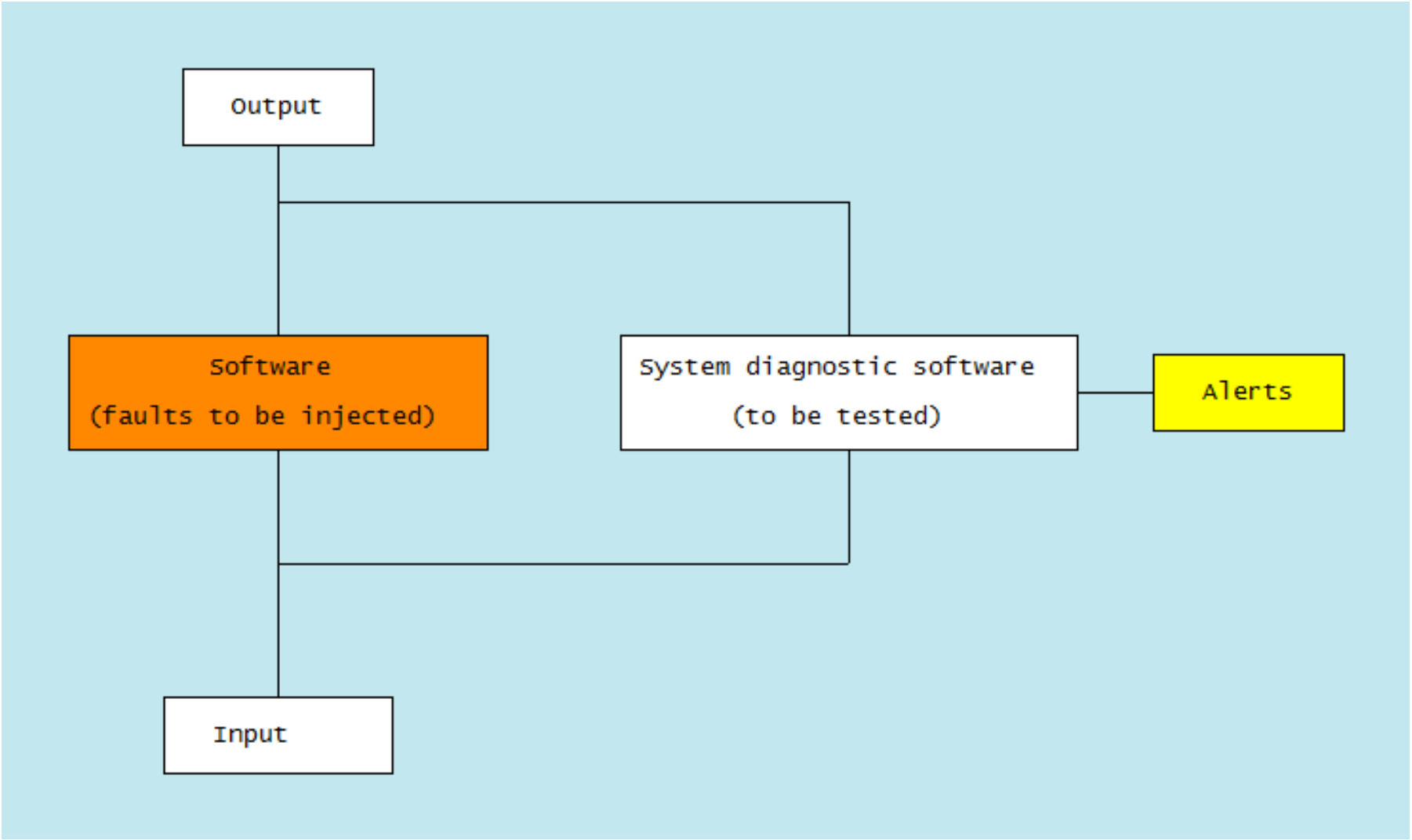
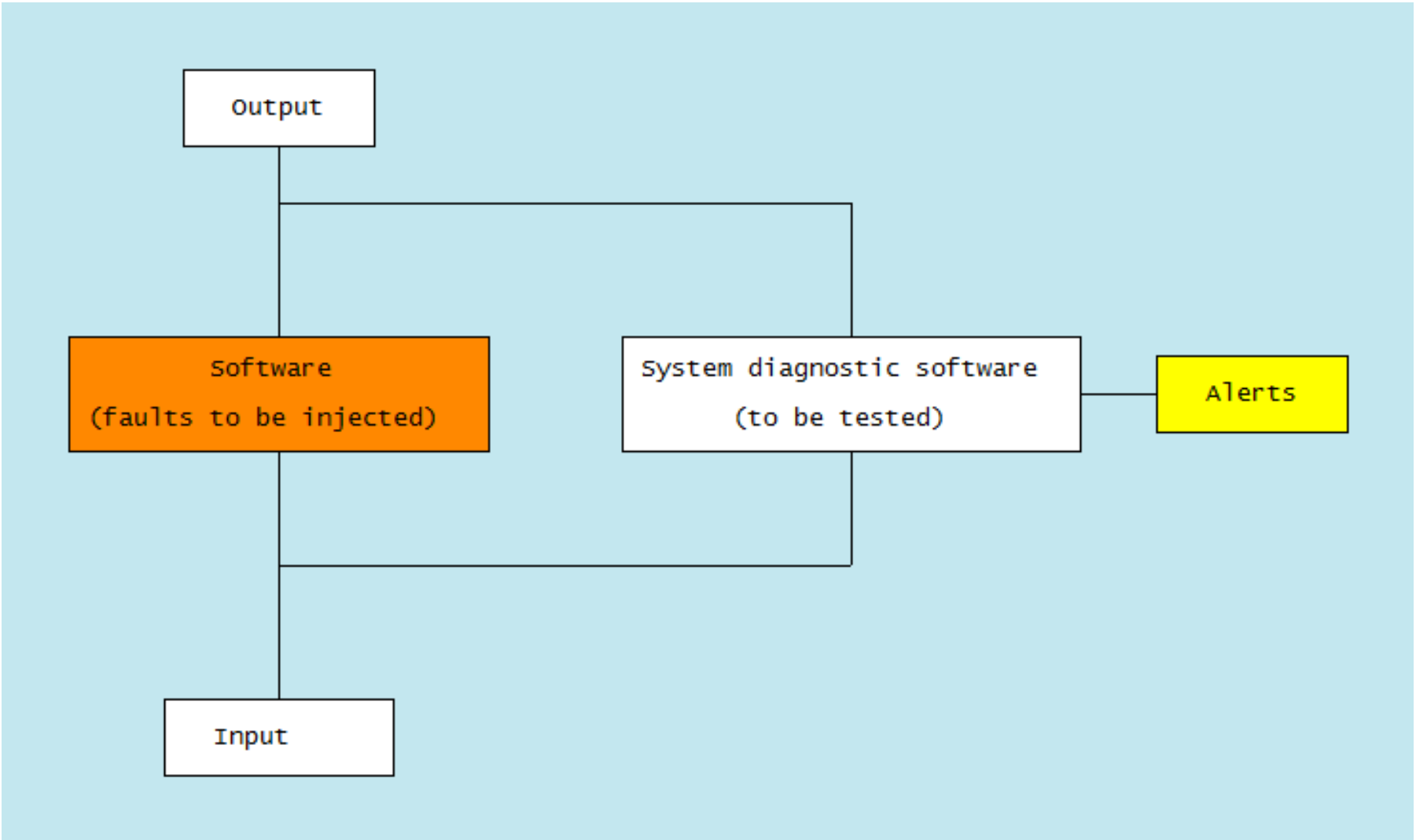


Table of contents

5. (semi-) Automated generation of diagnostic software

Can we generate a system diagnostic software automatically ?

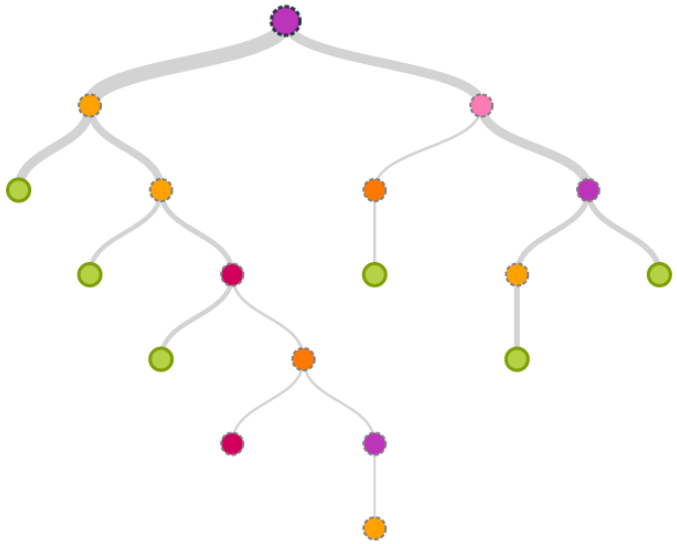


5. (semi-) Automated generation of diagnostic software

Collect lot of mutant outputs

↓

Classify using decision tree algorithms



↓

Diagnostic software

(Can also be made adaptive – re-learn from false positives)

Table of contents

6. (semi-) Automated Software FTA/FMEA

In source based mutation testing

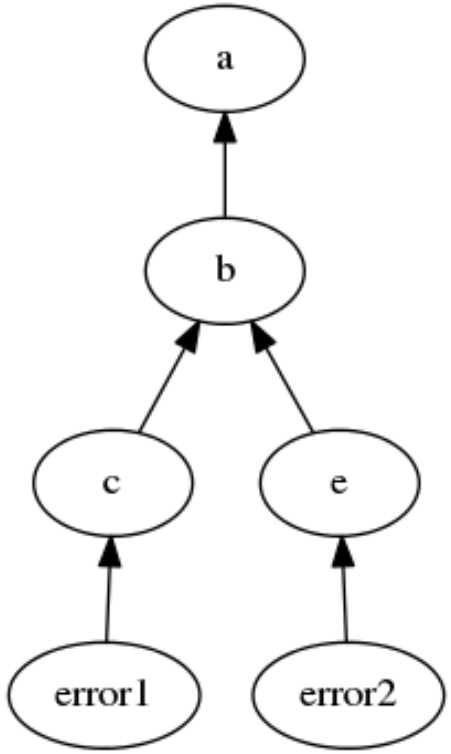
- A function/method can be considered as sub-component
- A module can be considered as a component.
- **Whenever a safety property is violated or a crash occurs:**
  Stack trace is recorded and a FTA is created !

```
Traceback (most recent call last):
  File "tb.py", line 10, in <module>
    a()
  File "tb.py", line 2, in a
    b()
  File "tb.py", line 5, in b
    c()
  File "tb.py", line 8, in c
    error1()
```

```
Traceback (most recent call last):
  File "tb.py", line 10, in <module>
    a()
  File "tb.py", line 12, in a
    b()
  File "tb.py", line 15, in b
    e()
  File "tb.py", line 18, in e
    error2()
```

Stack traces of mutants can be combined to build fault trees and can help in perform FMEA/Safety analysis.

6. (semi-) Automated Software FTA/FMEA




```
Traceback (most recent call last):
  File "tb.py", line 10, in <module>
    a()
  File "tb.py", line 2, in a
    b()
  File "tb.py", line 5, in b
    c()
  File "tb.py", line 8, in c
    error1()
```

```
Traceback (most recent call last):
  File "tb.py", line 10, in <module>
    a()
  File "tb.py", line 12, in a
    b()
  File "tb.py", line 15, in b
    e()
  File "tb.py", line 18, in e
    error2()
```

Table of contents

Table of contents

1. Preprocessing

Decomment

Compiler and other tools are available to remove comments from source file:

Example: gcc -fpreprocessed -E -P source-file.c

Indent or beautify : For easier parsing

/* Before beautifying */	/* After beautifying */

if (condition) {
 	
a= (b * 2+3)/3 ;}
 	
else{
 	
a=(22/7)+(3/2);
}
	if (condition)
	{
	a = (b * 2 + 3) / 3;
	}
	else
	{
	a = (22 / 7) + (3 / 2);
	}

Table of contents

2. Is computationally expensive

Is it practical ?

- Computationally intensive
  (Induce a fault $\rightarrow$ Compile $\rightarrow$ Run test cases $\rightarrow$ Check)
- **Very expensive if you use a low powered embedded system to carryout mutation testing.**
- Watch dog required if your mutant ends in an infinite loop. (Kill the mutants after reaching a threshold time).

Hence, on large projects it is advisable to perform mutation testing on the crtitical code only !

Table of contents

3. Proof that faults are induced at all paths of a program

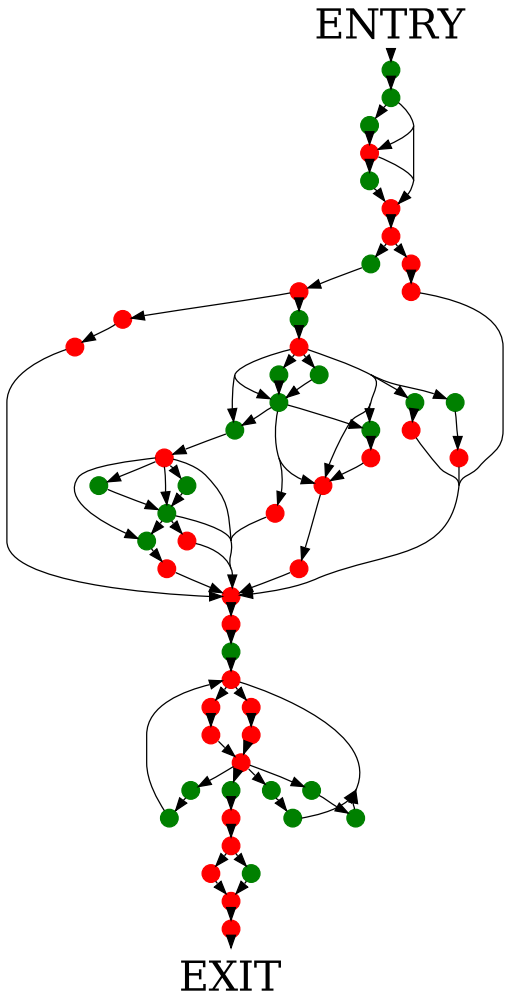


Figure: Concatenating LCSAJs of a program (GES) –(The GREEN colored nodes indicate the LCSAJ points where faults have been induced and caught; the RED colored nodes indicate otherwise)

Table of contents

4. Equivalent mutants problem

An indication of test adequacy is the mutation score :

$$\frac{\textit{No. of mutants killed}(K)}{\textit{No. of mutants generated}(G) - \textit{No. of equivalent mutants}(E)}$$

Mutant score example:

$$\frac{K}{G - E} = \frac{700}{1000 - 0}$$

Unless all the equivalent mutants are correctly detected,
mutant score can never be $= 1$.

4. Equivalent mutants problem : Equivalent mutants – 1

Example – 1

```
for( i = 0 ; i < N ; ++i )
{
        sum = sum + array [i] ;
}
```

```
for( i = 0 ; i != N ; ++i )
{
        sum = sum + array [i] ;
}
```

Example – 2

```
for( i = 0; i < length; ++i )
{
  for( j = i + 1; j < length; ++j)
   {
      ...   sorting function
   }
}
```

```
for( i = 0; i < length; ++i )
{
  for( j = i + 0; j < length; ++j)
   {
      ...   sorting function
   }
}
```

4. Equivalent mutants problem : Equivalent mutants — 2

Example – 3

```
max = array[0] ;

for( i = 1 ; i < N ; ++i )
{
      if ( max < array[i] )
          max = array[i]
}
```

```
max = array[0] ;

for( i = 1 ; i < N ; ++i )
{
      if ( max <= array[i] )
          max = array[i]
}
```

Example – 4

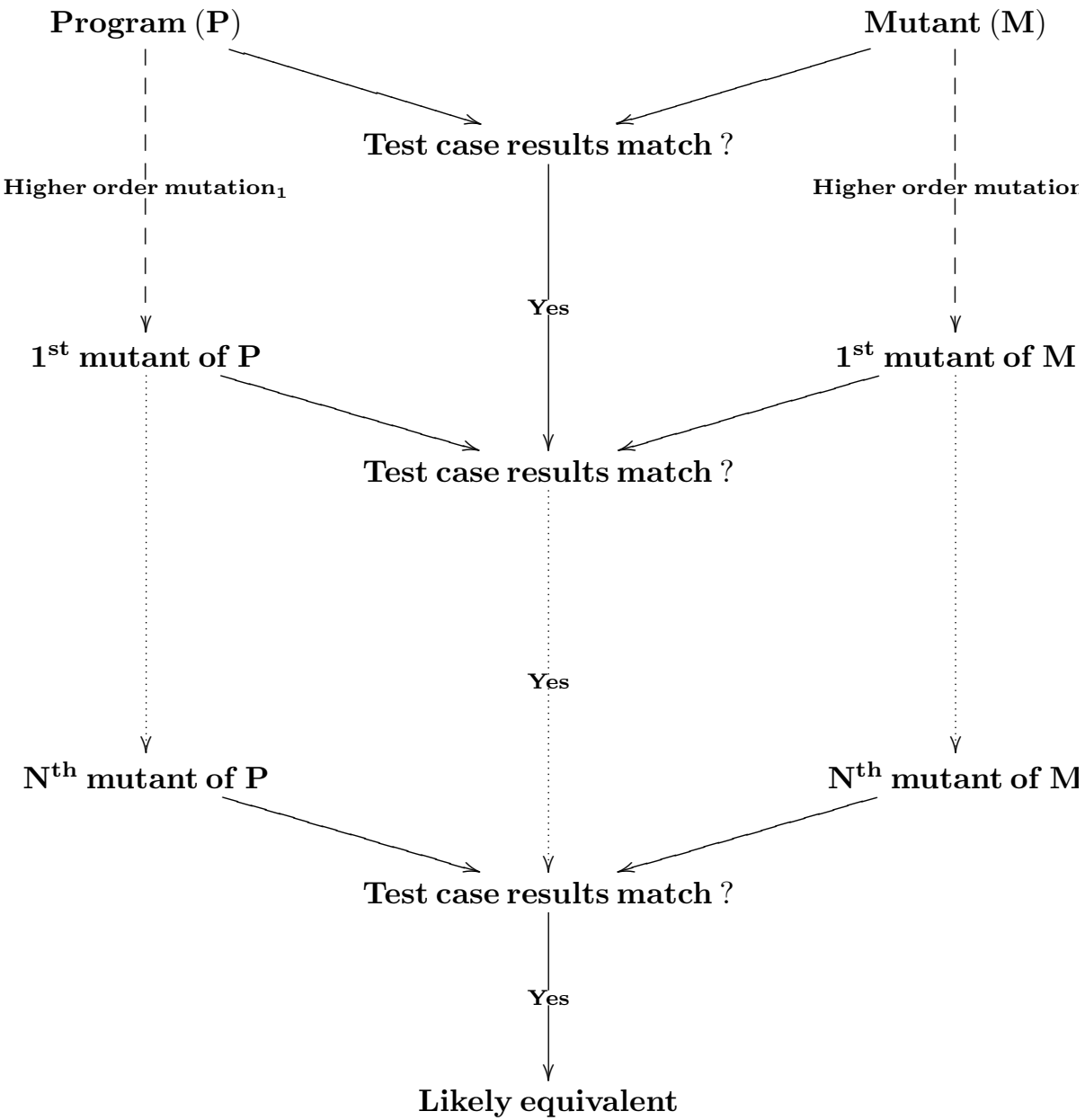
```
int i = 0 ;

for( i = 0 ; i < length ; ++i)
{
   ...
}
```

```
int i = 5 ;

for( i = 0 ; i < length ; ++i)
{
   ...
}
```

4. Equivalent mutants problem : Detection



4. Equivalent mutants problem : Detection

```
int Max( int *array,
         unsigned int length)
{
   unsigned int i ;
   int max = array [0] ;

   for( i = 1 ; i < length ; ++i )
   {
      if ( array [i] > max )
         max = array [i] ;
   }

   return     max ;
}
```

```
int Max (int *array,
         unsigned int length)
{
   unsigned int i ;
   int max = array [0] ;

   for( i = 1 ; i != length ; ++i )
   {
      if ( array [i] > max )
         max = array [i] ;
   }

   return     max ;
}
```

Figure: Example of an equivalent mutant

4. Equivalent mutants problem : Detection

```
int Max( int *array,
         unsigned int length)
{
   unsigned int i ;
   int max = array [0] ;

   for( i = 1 ; i < length ; ++i )
   {
      if ( array [i] <= max )
         max = array [i - 1] ;
   }

   return    -1 * max ;
}
```

```
int Max (int *array,
         unsigned int length)
{
   unsigned int i ;
   int max = array [0] ;

   for( i = 1 ; i != length ; ++i )
   {
      if ( array [i] <= max )
         max = array [i - 1] ;
   }

   return    -1 * max ;
}
```

Figure: Detecting equivalent mutant

Table of contents

5. Which mutants to kill first ?

Aim

To understand properties of mutants.

Motivation

1. Manually checking all mutants is difficult.
2. To prioritize which of the unkilled mutants must be analysed.

Method

1. Static, Dynamic, and Coverage analysis of mutants is performed.
2. And Principal component analysis (PCA) of the above results are plotted.

5. Which mutants to kill first ?

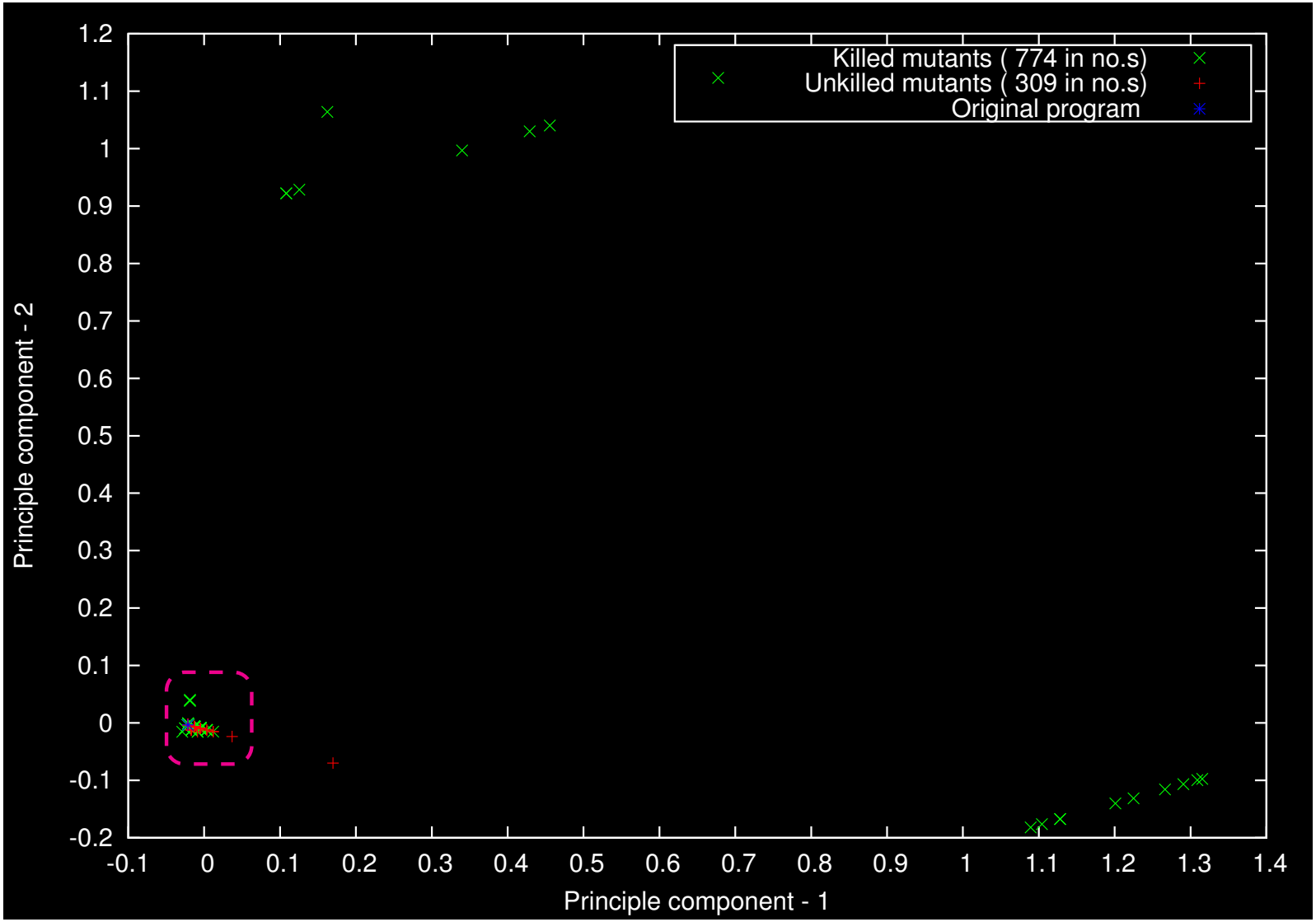


Figure: PCA of mutant characteristics for FSHS

5. Which mutants to kill first ?

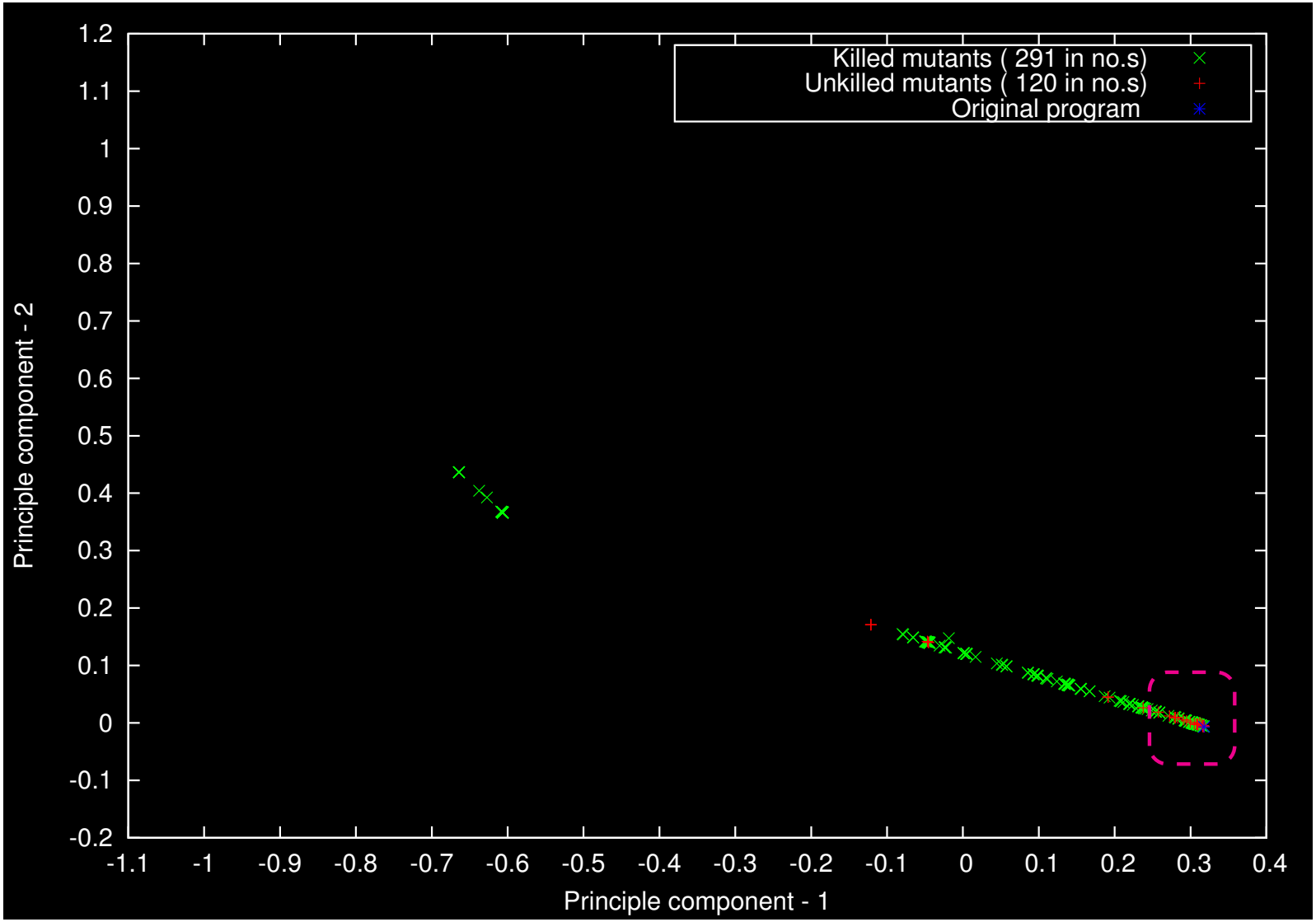


Figure: PCA of mutant characteristics for RSH

5. Which mutants to kill first ?

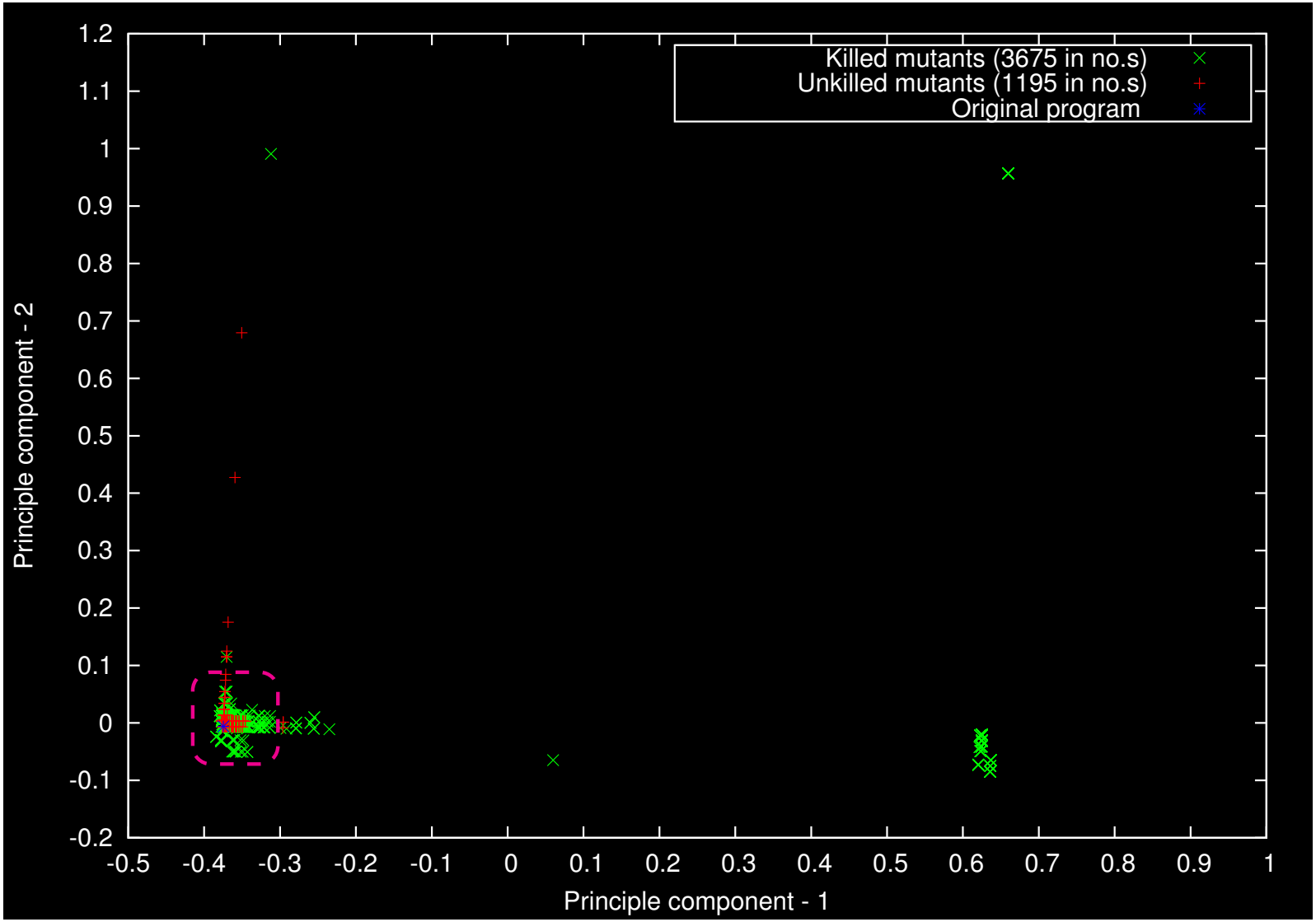


Figure: PCA of mutant characteristics for SGTLD

5. Which mutants to kill first ?

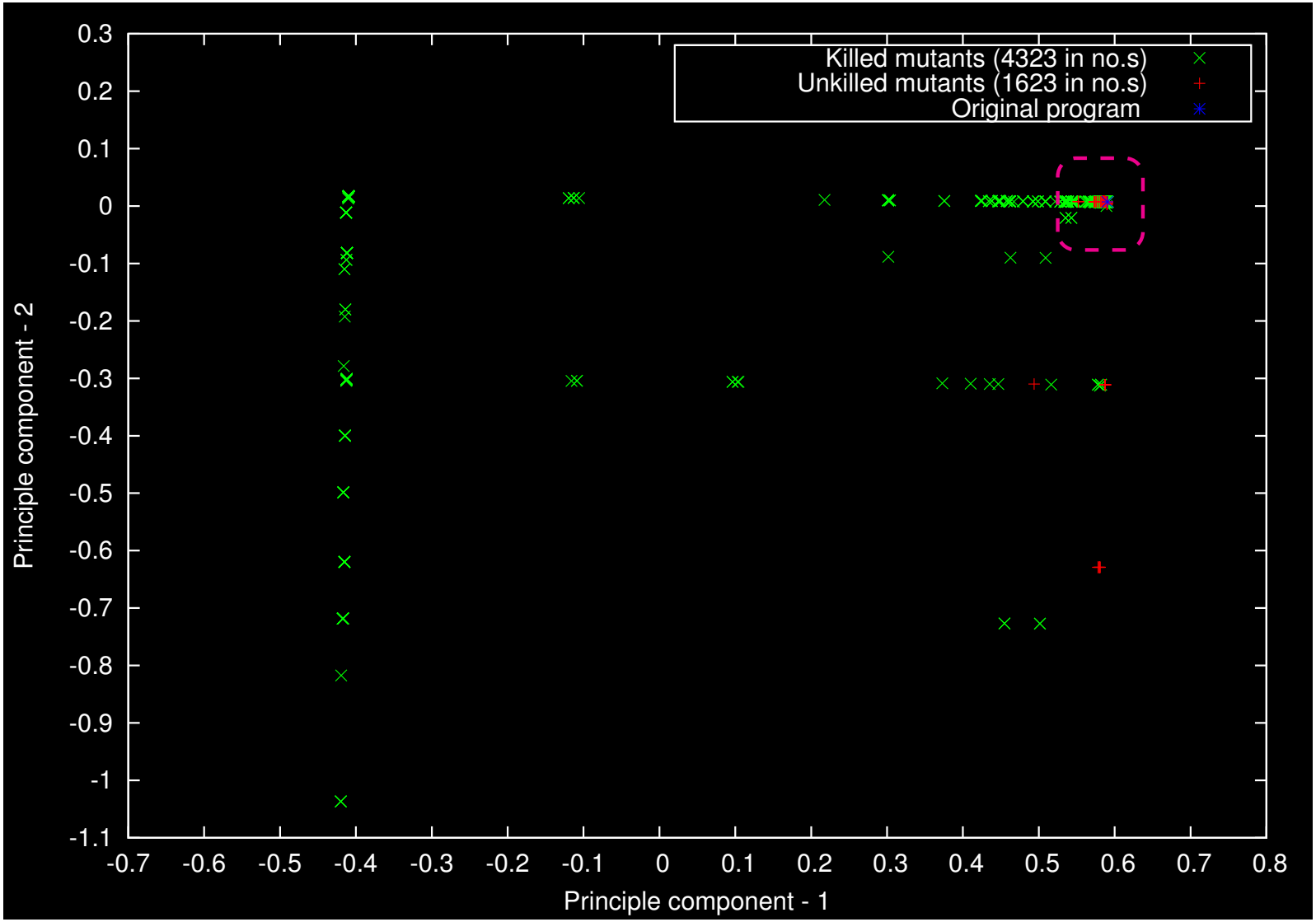


Figure: PCA of mutant characteristics for CTMS

5. Which mutants to kill first ?

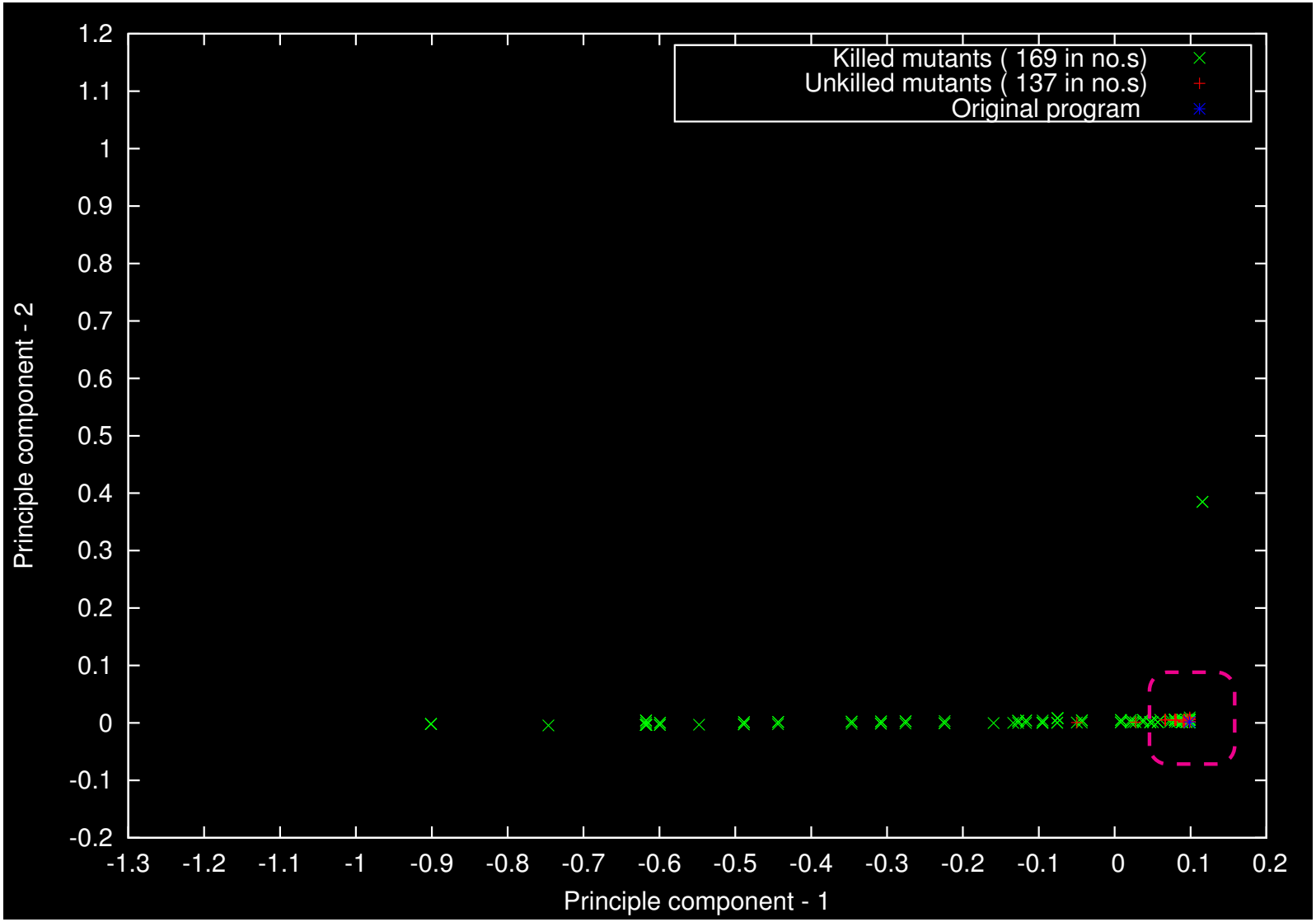


Figure: PCA of mutant characteristics for GES

5. Which mutants to kill first ?

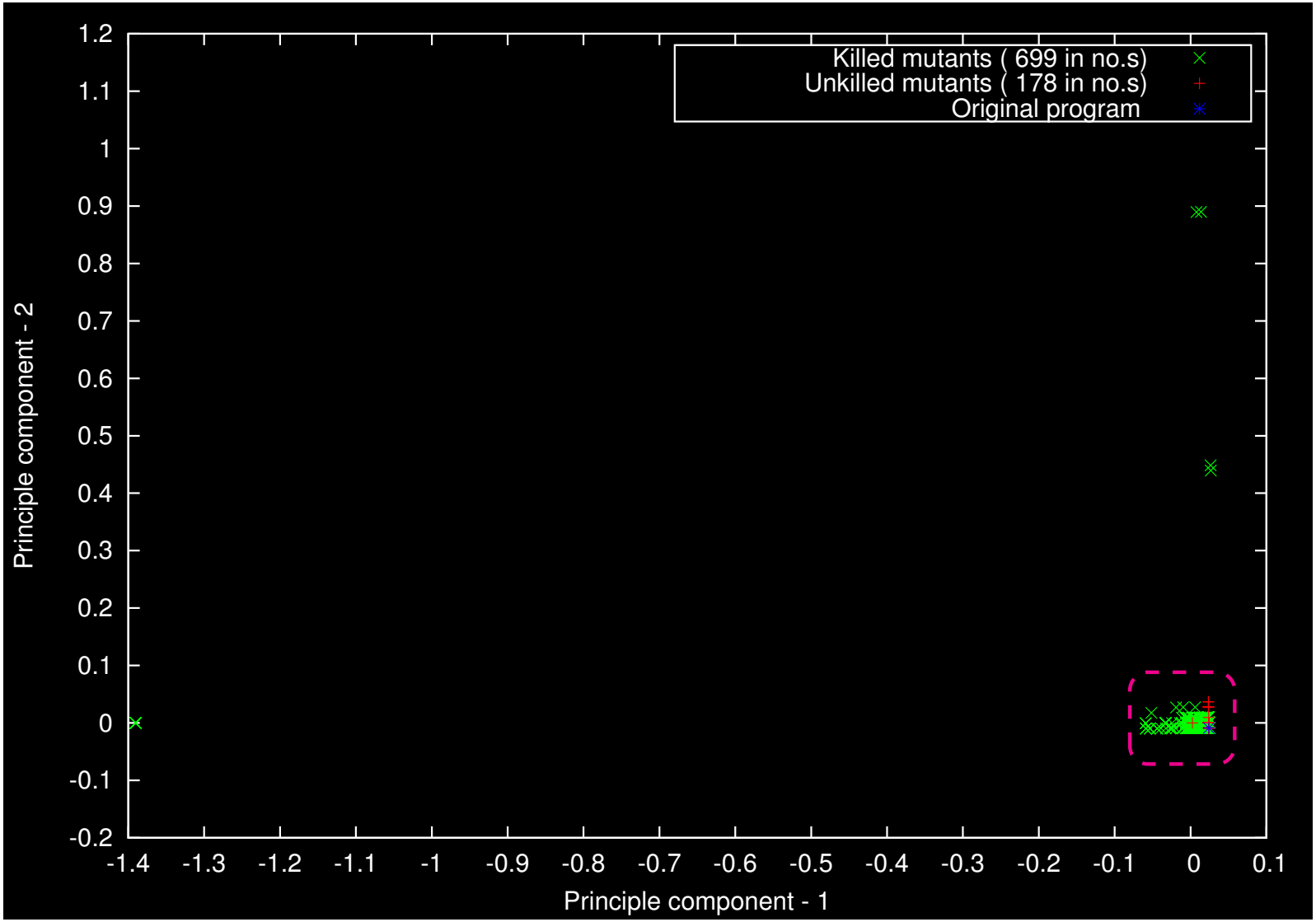


Figure: PCA of mutant characteristics for SGDHR

Table of contents

Summary of my talk

Mutation testing benefits

1. To know if adequate testing has been performed
2. Identifying Safety/Mission/Security critical code or module
3. Useful in testing diversified N-version software
4. Testing system diagnosis/monitoring software

Research areas

1. (semi-) Automated generation of diagnostic software
2. (semi-) Automated Software FTA/FMEA

Some academic areas with limited success

1. Automatic fault localization and software repair
2. Estimating amount of code reused

Thank you

Index