Applicability of Mutation Testing to Space Software

Fabrizio Pastore,
Jaekwon Lee, Enrico Viganò, Oscar Cornejo, Lionel Briand
University of Luxembourg
(fabrizio.pastore@uni.lu)

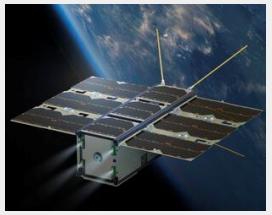
ADCSS 2023 - November 15th, 2023

Fault-based, Automated Quality Assurance Assessment and Augmentation for Space Software 2 (FAQAS-1 & 2)

SnT/University of Luxembourg

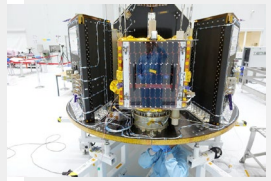
- Technology provider

Gomspace Luxembourg (GSL)



- Develop Nanosatellites
- Case study provider
- Technology validator

LuxSpace



- Develop Microsatelites
- Case study provider
- Technology validator

Huld - Finland

- Develop SW solutions
- ISVV supplier
- Case study provider
- Technology validator

Dr. Fabrizio Pastore
PI, Chief Scientist II, **SVV**, SnT

Jaekwon Lee
PhD Candidate, **SVV**, SnT

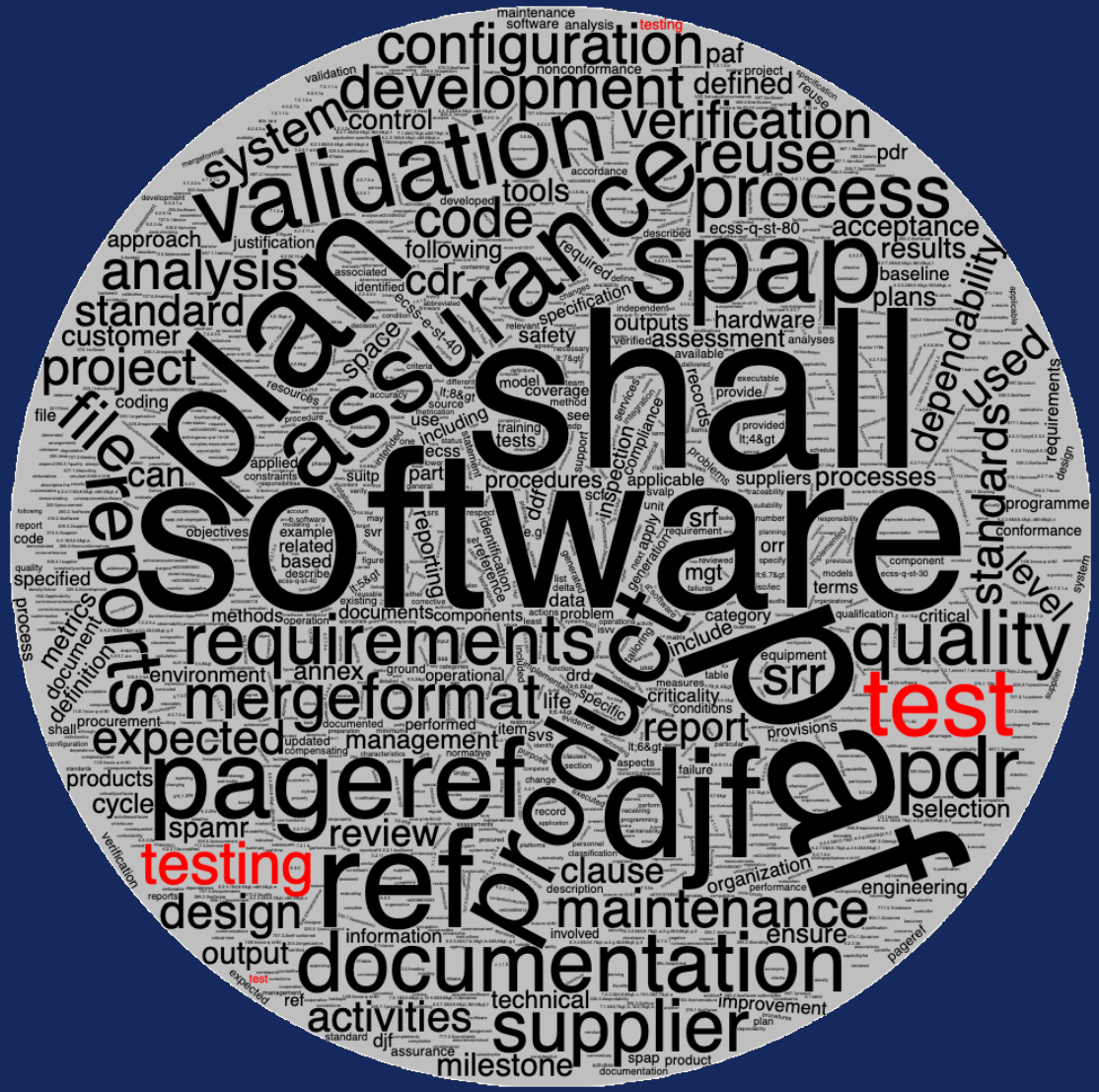
Prof. Lionel Briand
Full Professor, Head of **SVV**, SnT

Enrico Viganò
R&D Specialist, **SVV**, SnT

LUXEMBOURG SPACE AGENCY

https://faqas.uni.lu/

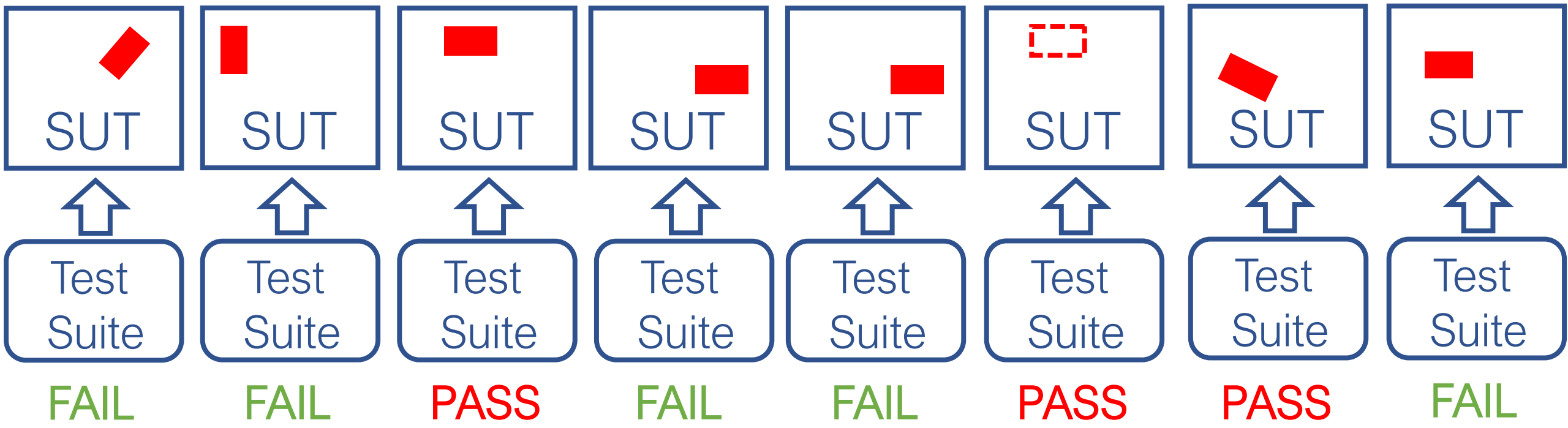
Software testing is prevalent in V&V

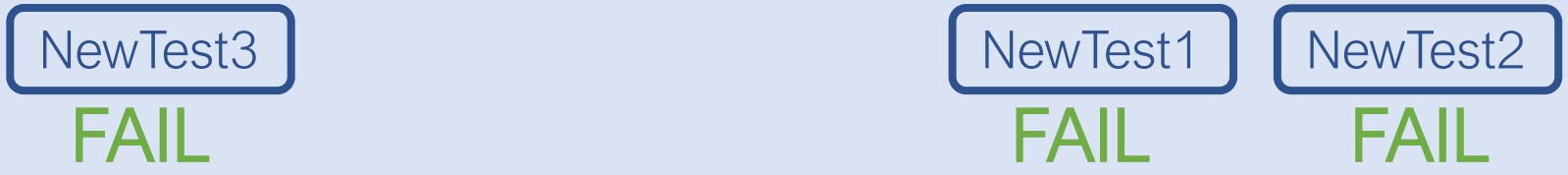


Word cloud for
ECSS-Q-ST-80C standard

How to ensure thorough testing?

Mutation Analysis and Testing



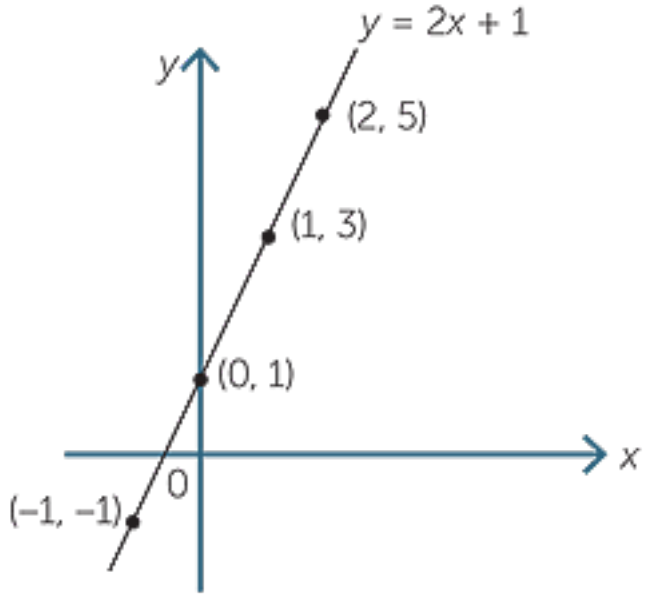


Improve with Automatically Generated Test Cases

Simplified example

Software Under Test

```
//
// Computes the y in a straight line.
// Return the result of m*x+c
//
int computeY( int m, int x, int c){ …
```



Software Under Test

```
//
// Computes the y in a straight line.
// Return the result of m*x+c
//
int computeY( int m, int x, int c){ …
```

Is this test suite effective?

Test Suite

```
void test_non_zero() {
    int res = computeY( 1, 2, 2);
    //1*2+2 = 4
    assert( 4 == res);
}

void test_zero_coefficients() {
    int res = computeY( 0, 3, 0);
    //0*2+0 = 0
    assert( 0 == res);
}

void test_zero_x() {
    int res = computeY( 2, 0, 0);
    //2*0+0 = 0
    assert( 0 == res);
}
```

Software Under Test

```
//
// Computes the y in a straight line.
// Return the result of m*x+c
//
int computeY( int m, int x, int c){
    return m*x*c;
}
```

Test Suite

```
void test_non_zero() {
    int res = computeY( 1, 2, 2);
    //1*2+2 = 4
    assert( 4 == res);
}

void test_zero_coefficients() {
    int res = computeY( 0, 3, 0);
    //0*2+0 = 0
    assert( 0 == res);
}

void test_zero_x() {
    int res = computeY( 2, 0, 0);
    //2*0+0 = 0
    assert( 0 == res);
}
```

Software Under Test

```
//
// Computes the y in a straight line.
// Return the result of m*x+c
//
int computeY( int m, int x, int c){
    return m*x*c;
}
```

100% statement coverage

100% branches coverage

100% MC/DC

Test Suite

```
void test_non_zero() {
    int res = computeY( 1, 2, 2);
    //1*2+2 = 4
    assert( 4 == res);
}
```

PASS

1*2*2 = 4

```
void test_zero_coefficients() {
    int res = computeY( 0, 3, 0);
    //0*2+0 = 0
    assert( 0 == res);
}
```

PASS

0*2*0 = 0

```
void test_zero_x() {
    int res = computeY( 2, 0, 0);
    //2*0+0 = 0
    assert( 0 == res);
}
```

PASS

2*0*0 = 0

Software Under Test

```
//
// Computes the y in a straight line.
// Return the result of m*x+c
//
int computeY( int m, int x, int c){
    return m*x*c;
}
```

We create mutants by automatically applying a set of “mutation operators”.

For simplicity, we apply only the “arithmetic operator deletion” operator

Test Suite

```
void test_non_zero() {
    int res = computeY( 1, 2, 2);
    //1*2+2 = 4
    assert( 4 == res);
}

void test_zero_coefficients() {
    int res = computeY( 0, 3, 0);
    //0*2+0 = 0
    assert( 0 == res);
}

void test_zero_x() {
    int res = computeY( 2, 0, 0);
    //2*0+0 = 0
    assert( 0 == res);
}
```

Software Under Test

```
//
// Computes the y in a straight line.
// Return the result of m*x+c
//
int computeY( int m, int x, int c){
    return m*x*c;
}
```

MUTANT1

```
int computeY( int m, int x, int c){
    return m*x;
}
```

MUTANT2

```
int computeY( int m, int x, int c){
    return x*c;
}
```

Test Suite

```
void test_non_zero() {
    int res = computeY( 1, 2, 2);
    //1*2+2 = 4
    assert( 4 == res);
}

void test_zero_coefficients() {
    int res = computeY( 0, 3, 0);
    //0*2+0 = 0
    assert( 0 == res);
}

void test_zero_x() {
    int res = computeY( 2, 0, 0);
    //2*0+0 = 0
    assert( 0 == res);
}
```

Software Under Test

```
//
// Computes the y in a straight line.
// Return the result of m*x+c
//
int computeY( int m, int x, int c){
    return m*x*c;
}

MUTANT1
int computeY( int m, int x, int c){
    return m*x;
}

MUTANT2
int computeY( int m, int x, int c){
    return x*c;
}
```

Test Suite

```
void test_non_zero() {
    int res = computeY( 1, 2, 2);
    //1*2+2 = 4
    assert( 4 == res);
}
```

FAIL
1*2 = 2

```
void test_zero_coefficients() {
    int res = computeY( 0, 3, 0);
    //0*2+0 = 0
    assert( 0 == res);
}
```

PASS
0*2 = 0

```
void test_zero_x() {
    int res = computeY( 2, 0, 0);
    //2*0+0 = 0
    assert( 0 == res);
}
```

PASS
2*0 = 0

Software Under Test

```
//
// Computes the y in a straight line.
// Return the result of m*x+c
//
int computeY( int m, int x, int c){
    return m*x*c;
}
```

MUTANT1

```
int computeY( int m, int x, int c){
    return m*x;
}
```

MUTANT2

```
int computeY( int m, int x, int c){
    return x*c;
}
```

Test Suite

```
void test_non_zero() {
    int res = computeY( 1, 2, 2);
    //1*2+2 = 4
    assert( 4 == res);
}
```

PASS

2*2 = 4

```
void test_zero_coefficients() {
    int res = computeY( 0, 3, 0);
    //0*2+0 = 0
    assert( 0 == res);
}
```

PASS

2*0 = 0

```
void test_zero_x() {
    int res = computeY( 2, 0, 0);
    //2*0+0 = 0
    assert( 0 == res);
}
```

PASS

0*0 = 0

Software Under Test

```
//
// Computes the y in a straight line.
// Return the result of m*x+c
//
int computeY( int m, int x, int c){
    return m*x*c;
}
```

MUTANT1

```
int computeY( int m, int x, int c){
    return m*x;
}
```

MUTANT2

```
int computeY( int m, int x, int c){
    return x*c;
}
```

Mutation score = 50%
(test suite incomplete)

The live mutant lacks ‘m’.
It means that
we are not testing
the effect of
‘m’ on the result

Software Under Test

```
//
// Computes the y in a straight line.
// Return the result of m*x+c
//
int computeY( int m, int x, int c){
    return m*x*c;
}
```

MUTANT1

```
int computeY( int m, int x, int c){
    return m*x;
}
```

MUTANT2

```
int computeY( int m, int x, int c){
    return x*c;
}
```

Test Suite

Create an additional test case that verifies the effect of 'm' on the result

```
void test_MUTANT_2() {
    int res = computeY( 3, 2, 2);
    //3*2+2 = 8
    assert( 8 == res);
}
```

Mutant killed: 3*2*2 != 2*2

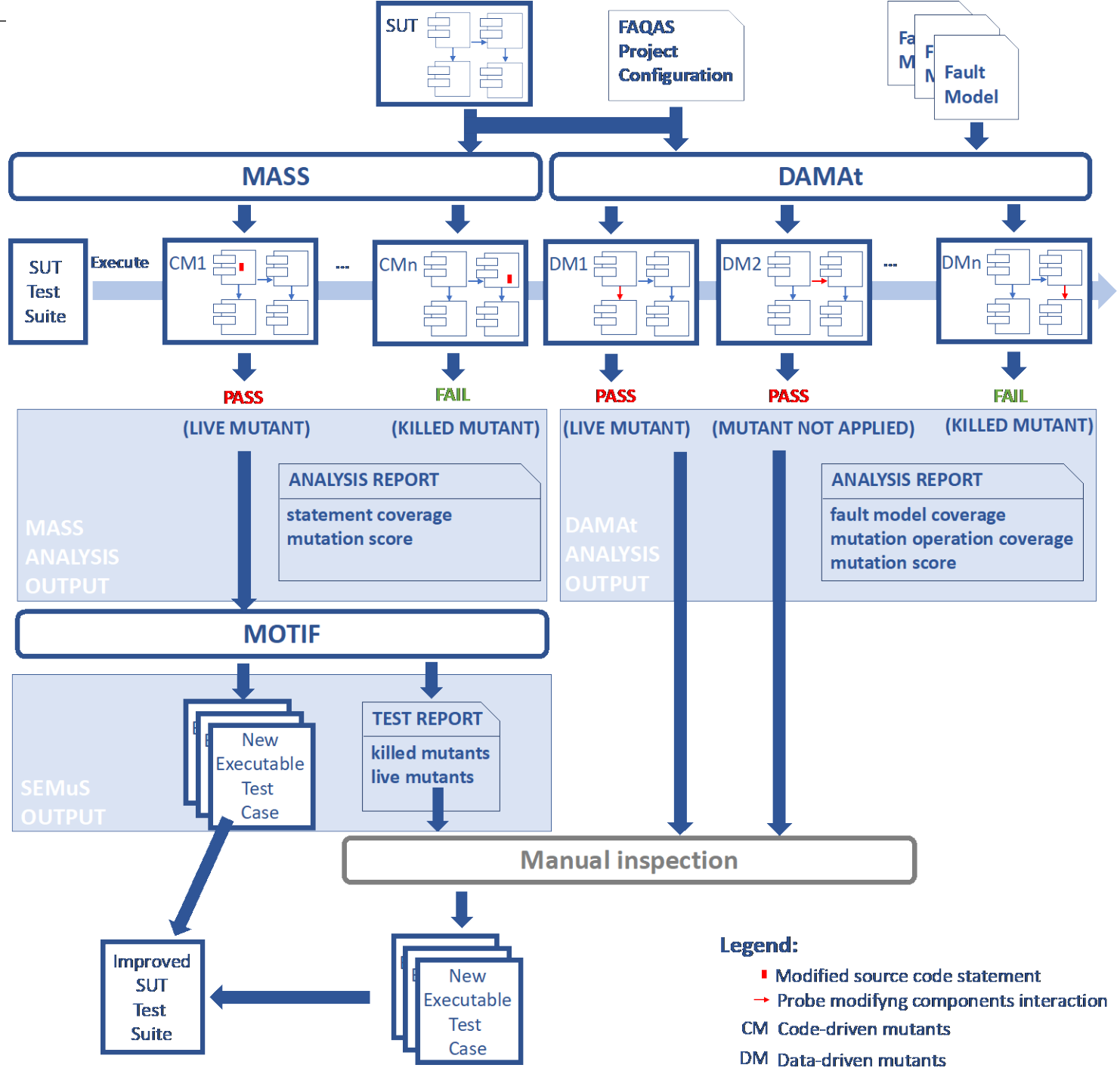
Bug discovered: test case fails with orginal program 3*2*2=12

FAQAS - FAQAS2

Fault-based, Automated Quality Assurance Assessment and Augmentation for Space Software

APPLICABILITY OF MUTATION TESTING METHOD FOR FLIGHT SOFTWARE.
Activity No. 100028866 in the esa-star system

IMPROVE MUTATION TESTING IN SPACE SOFTWARE SYSTEMS.
Activity No. 1000033730 in the esa-star system


SUT
FAQAS Project Configuration
Fault Model
MASS
DAMAt
SUT Test Suite
Execute
CM1
...
CMn
DM1
DM2
...
DMn
PASS
FAIL
PASS
PASS
FAIL
(LIVE MUTANT)
(KILLED MUTANT)
(LIVE MUTANT)
(MUTANT NOT APPLIED)
(KILLED MUTANT)
ANALYSIS REPORT
statement coverage
mutation score
MASS ANALYSIS OUTPUT
ANALYSIS REPORT
fault model coverage
mutation operation coverage
mutation score
DAMAt ANALYSIS OUTPUT
MOTIF
New Executable Test Case
TEST REPORT
killed mutants
live mutants
SEMuS OUTPUT
Manual inspection
Improved SUT Test Suite
New Executable Test Case
Legend:
Modified source code statement
Probe modifyng components interaction
CM Code-driven mutants
DM Data-driven mutants

Problems Addressed

- How to make mutation analysis scale?
- How to assess if test suites verify components integration properly?
- How to generate test cases that kill mutants in C software?

Problems Addressed

- How to make mutation analysis scale?
- How to assess if test suites v
  properly?
- How to generate test cases

https://github.com/SNTSVV/MASS

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 48, NO. 10, OCTOBER 2022 3913

Mutation Analysis for Cyber-Physical Systems: Scalable Solutions and Results in the Space Domain

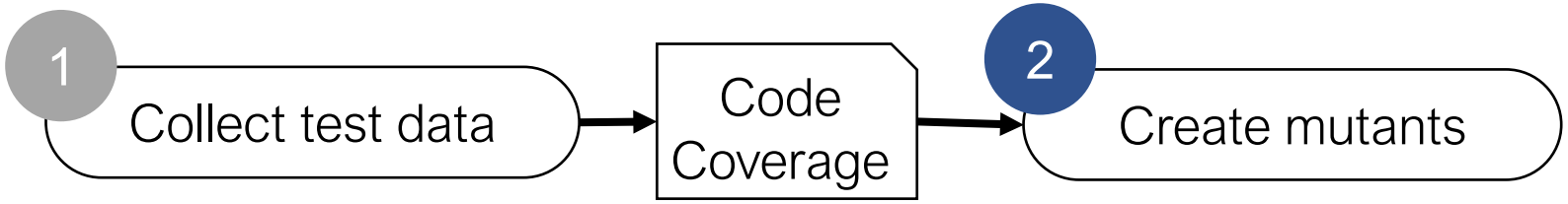
Oscar Cornejo, Fabrizio Pastore, *Member, IEEE*, and Lionel C. Briand, *Fellow, IEEE*

Abstract—On-board embedded software developed for spaceflight systems (*space software*) must adhere to stringent software quality assurance procedures. For example, verification and validation activities are typically performed and assessed by third party organizations. To further minimize the risk of human mistakes, space agencies, such as the European Space Agency (ESA), are looking for automated solutions for the assessment of software testing activities, which play a crucial role in this context. Though space software is our focus here, it should be noted that such software shares the above considerations, to a large extent, with embedded software in many other types of cyber-physical systems. Over the years, mutation analysis has shown to be a promising solution for the automated assessment of test suites; it consists of measuring the quality of a test suite in terms of the percentage of injected faults leading to a test failure. A number of optimization techniques, addressing scalability and accuracy problems, have been proposed to facilitate the industrial adoption of mutation analysis. However, to date, two major problems prevent space agencies from enforcing mutation analysis in space software development. First, there is uncertainty regarding the feasibility of applying mutation analysis optimization techniques in their context. Second, most of the existing techniques either can break the real-time requirements common in embedded software or cannot be applied when the software is tested in Software Validation Facilities, including CPU emulators and sensor simulators. In this paper, we enhance mutation analysis optimization techniques to enable their applicability to embedded software and propose a pipeline that successfully integrates them to address scalability and accuracy issues in this context, as described above. Further, we report on the largest study involving embedded software systems in the mutation analysis literature. Our research is part of a research project funded by ESA ESTEC involving private companies (GomSpace Luxembourg and LuxSpace) in the space sector. These industry partners provided the case studies reported in this paper; they include an on-board software system managing a microsatellite currently on-orbit, a set of libraries used in deployed cubesats, and a mathematical library certified by ESA.

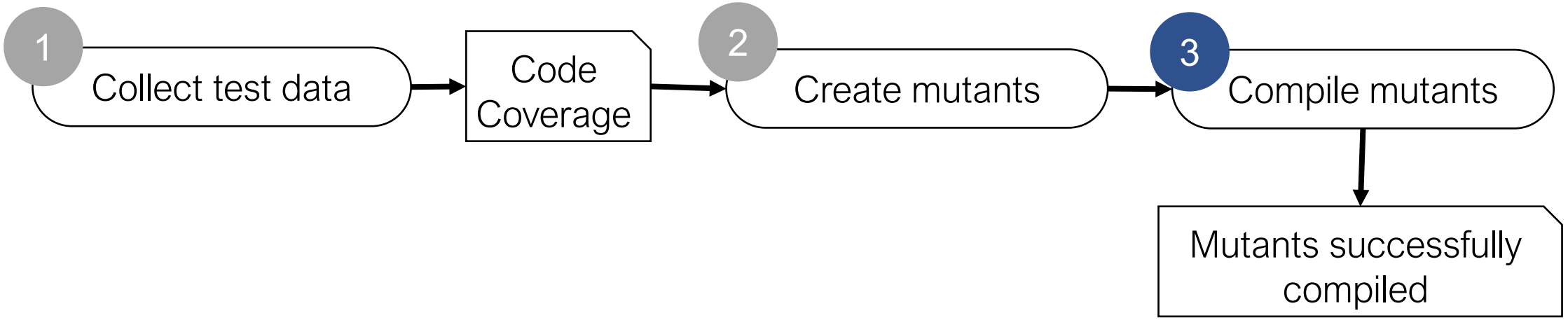
Index Terms—Mutation analysis, mutation testing, space software, embedded software, cyber-physical systems

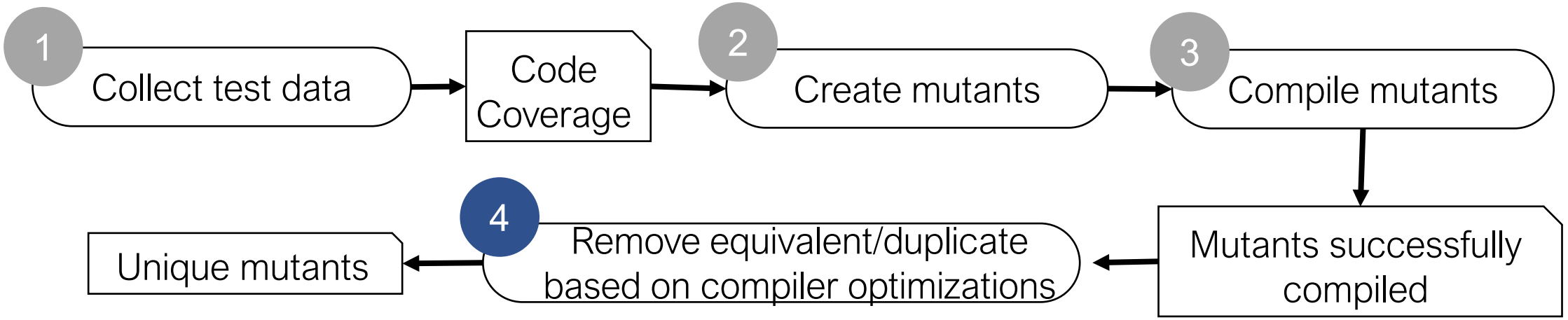
Mutation Analysis for Space Sofware (MASS)

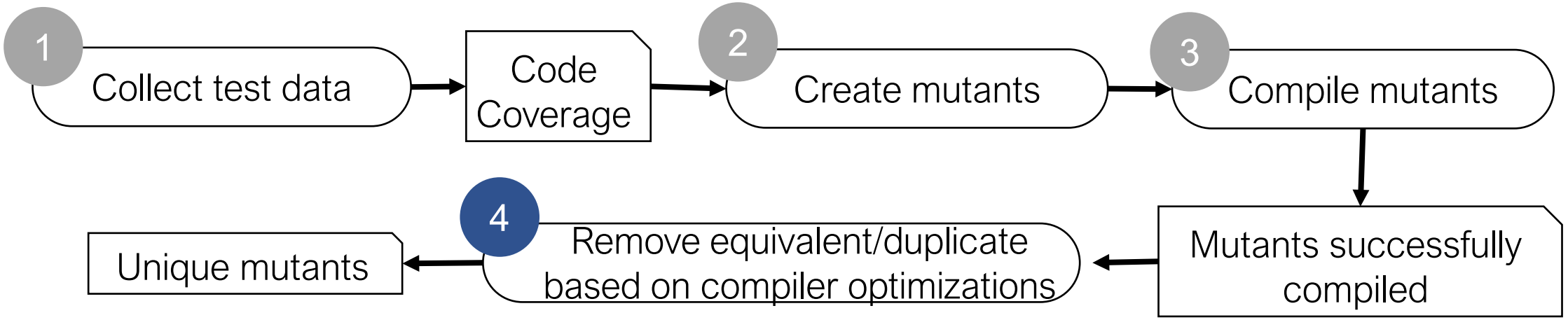

1
Collect test data
Code
Coverage



	Operator	Description*
Sufficient Set	ABS	$\{(v, -v)\}$
	AOR	$\{(op_1, op_2) \mid op_1, op_2 \in \{+, -, *, /, \%\} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{+=, -=, *=, /=, \%=\} \wedge op_1 \neq op_2\}$
	ICR	$\{i, x) \mid x \in \{1, -1, 0, i+1, i-1, -i\}\}$
	LCR	$\{(op_1, op_2) \mid op_1, op_2 \in \{\&\&, ||\} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{\&=, |=, \&=\} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{\&, |, \&\&\} \wedge op_1 \neq op_2\}$
	ROR	$\{(op_1, op_2) \mid op_1, op_2 \in \{>, >=, <, <=, ==, !=\}\}\{(e, !(e)) \mid e \in \{\texttt{if(e)}, \texttt{while(e)}\}\}$
	SDL	$\{(s, \texttt{remove}(s))\}$
	UOI	$\{(v, \texttt{-}v), (v, v\texttt{-}), (v, \texttt{++}v), (v, v\texttt{++})\}$
OODL Set	AOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{+, -, *, /, \%\}\}$
	LOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{\&\&, ||\}\}$
	ROD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{>, >=, <, <=, ==, !=\}\}$
	BOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{\&, |, \wedge\}\}$
	SOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{», «\}\}$
Other	LVR	$\{(l_1, l_2) \mid (l_1, l_2) \in \{(0, -1), (l_1, -l_1), (l_1, 0), (\mathit{true}, \mathit{false}), (\mathit{false}, \mathit{true})\}\}$


1
Collect test data
Code Coverage
2
Create mutants
3
Compile mutants
Mutants successfully compiled


1
Collect test data
Code Coverage
2
Create mutants
3
Compile mutants
Mutants successfully compiled
4
Remove equivalent/duplicate based on compiler optimizations
Unique mutants



Original program

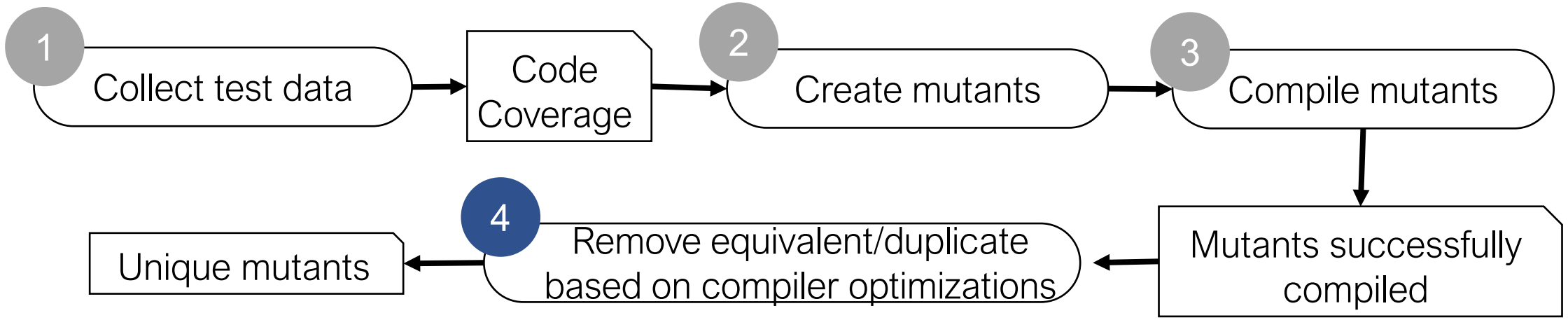
```
int abs( int x){
    if ( x < 0 )
        return -x;
    return x;
}
```

Mutant

```
int abs( int x){
    if ( x <= 0 )
        return -x;
    return x;
}
```

Equivalent:

returns the same output for any input



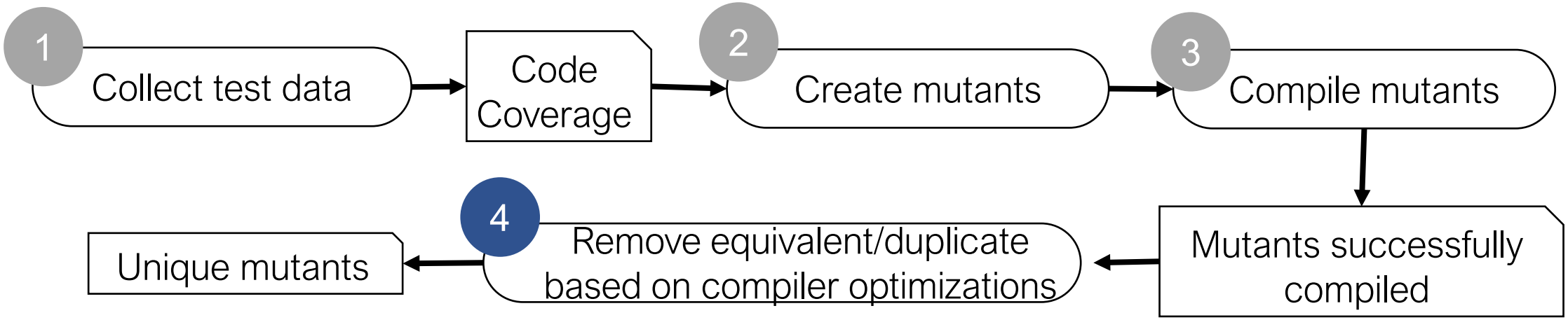

```
int same( int a[],
          int b[],
          int len){
   int i=0;
   if ( len < 0 ) return -1;
   while ( i < len ){
      if ( a[i] != b[i] )
         return 0;
      i++;
   }
   return 1;
}
```

Original program

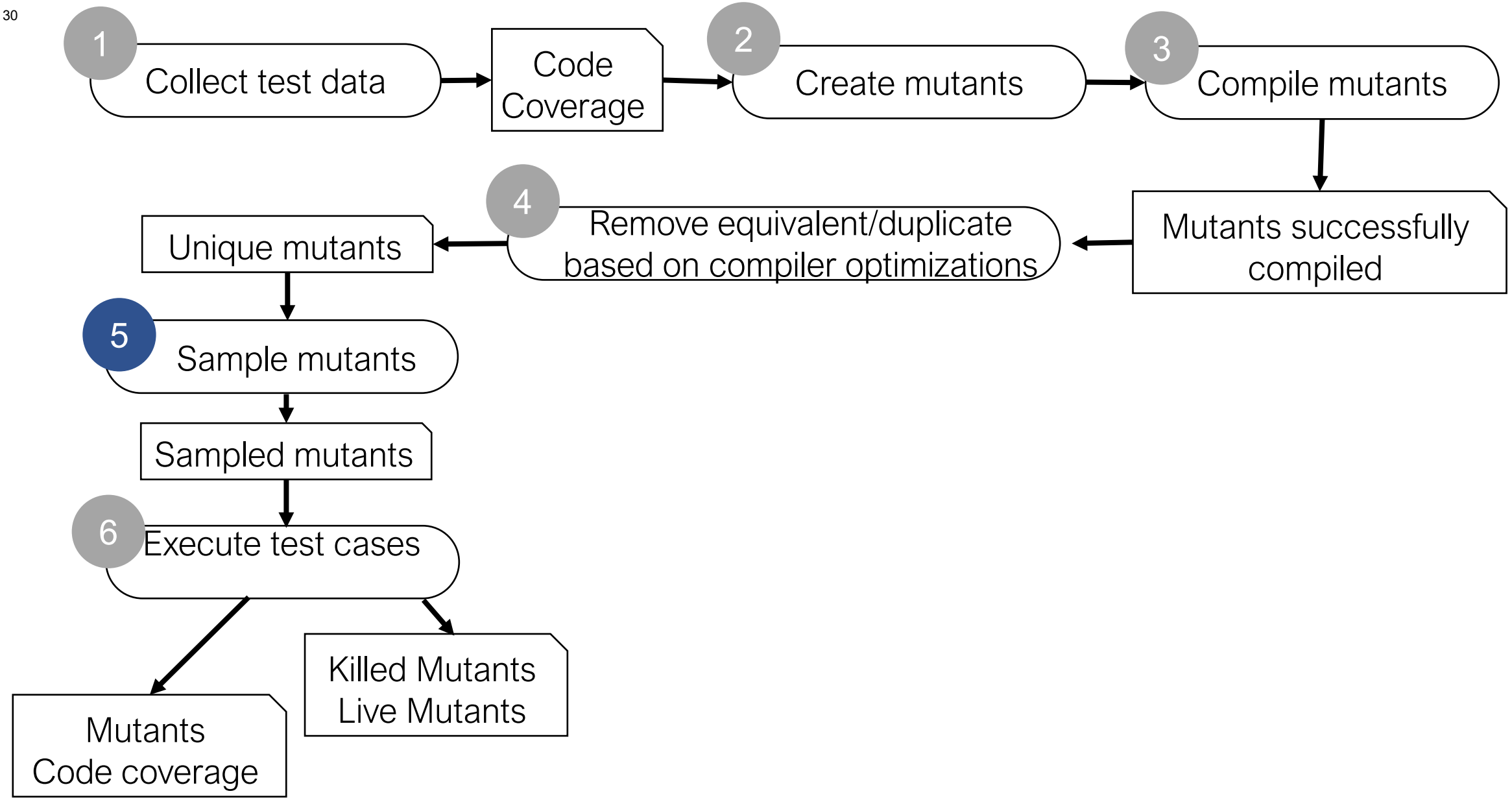
```
int same( int a[],
          int b[],
          int len){
   int i=0;
   if ( len < 0 ) return -1;
   while ( i < 0 ){
      if ( a[i] != b[i] )
         return 0;
      i++;
   }
   return 1;
}
```

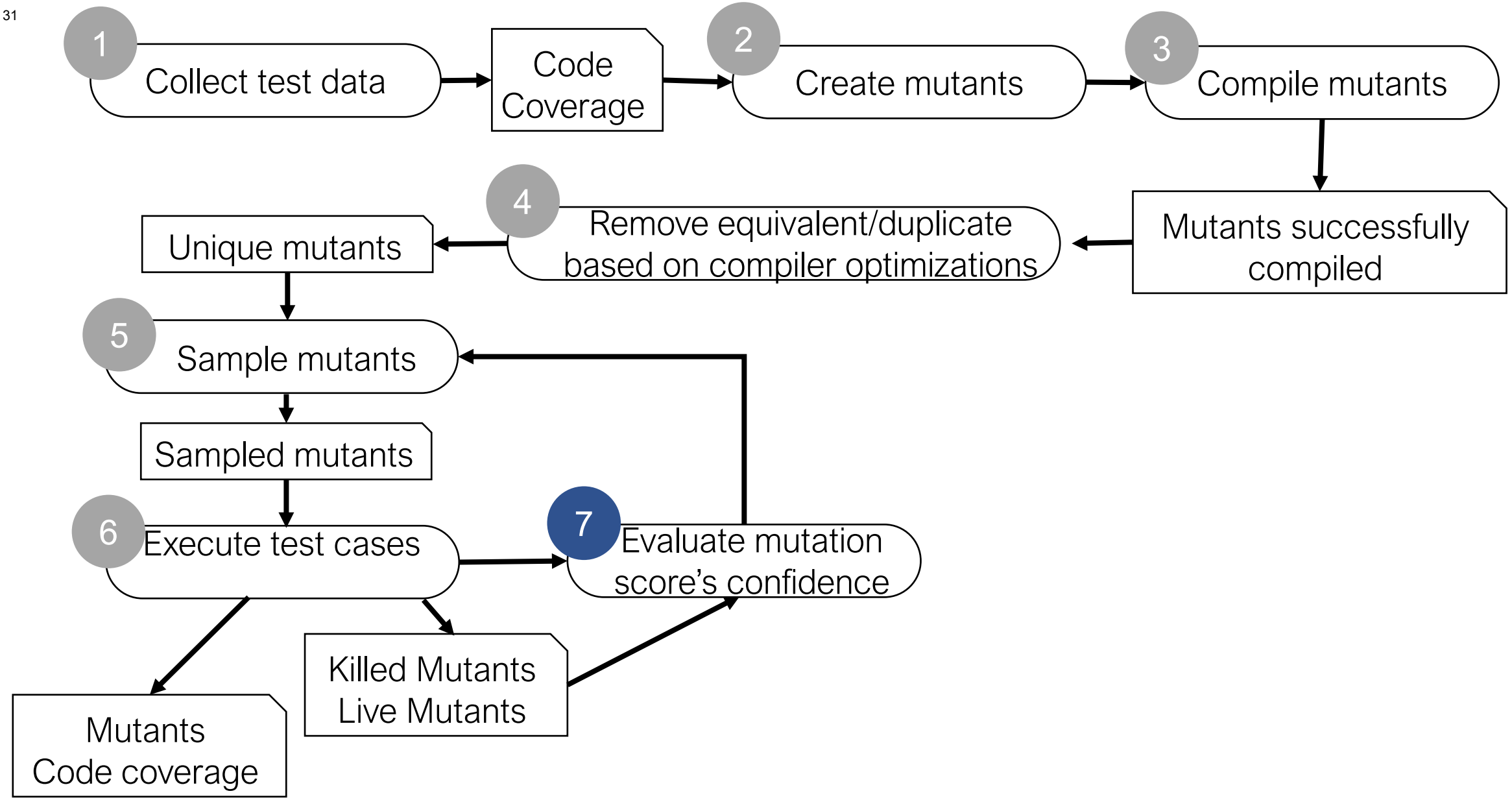
```
int same( int a[],
          int b[],
          int len){
   int i=0;
   if ( len < 0 ) return -1;
   while ( i > len ){
      if ( a[i] != b[i] )
         return 0;
      i++;
   }
   return 1;
}
```

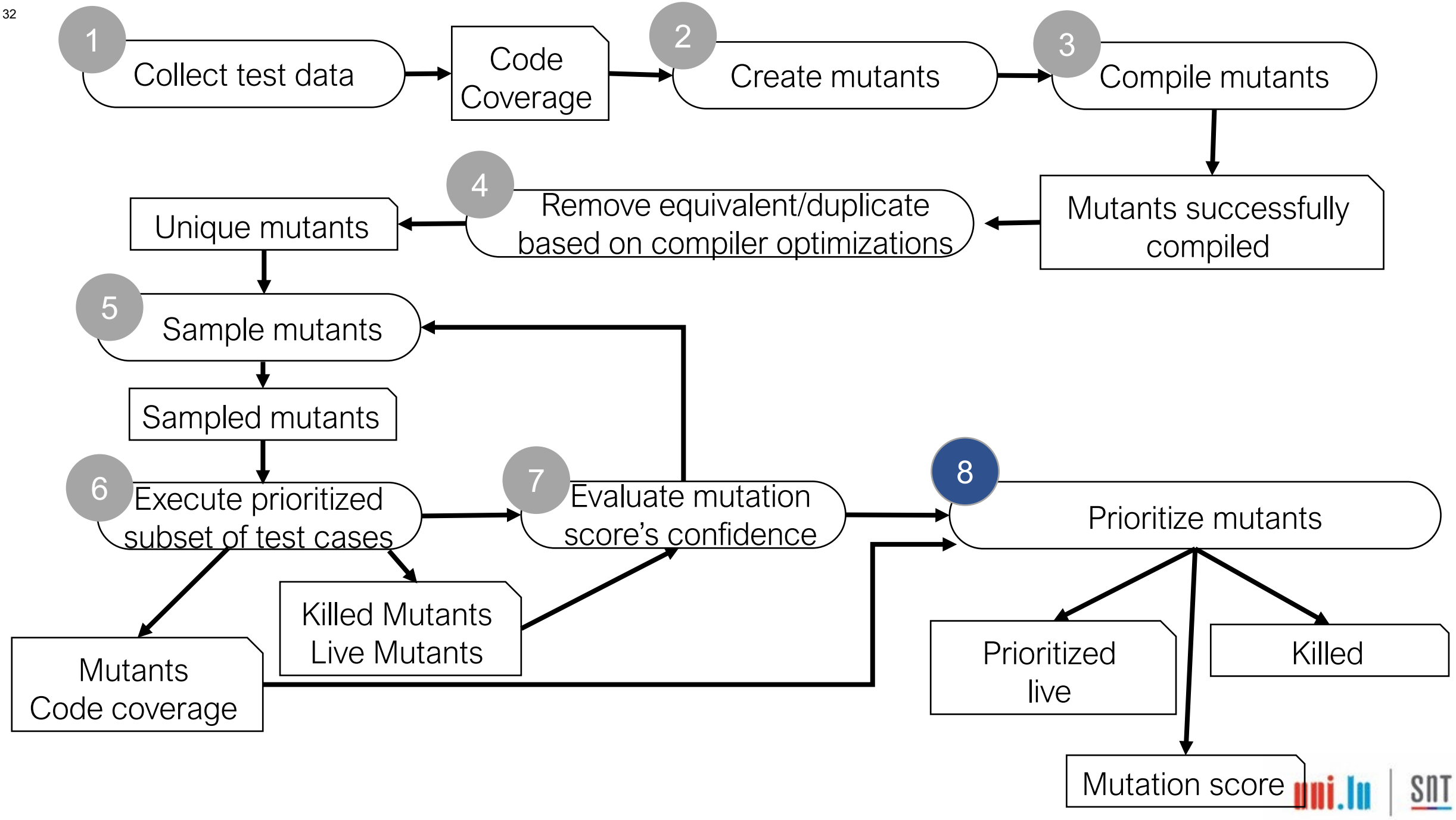
Duplicate mutants: both skip the loop and return 1

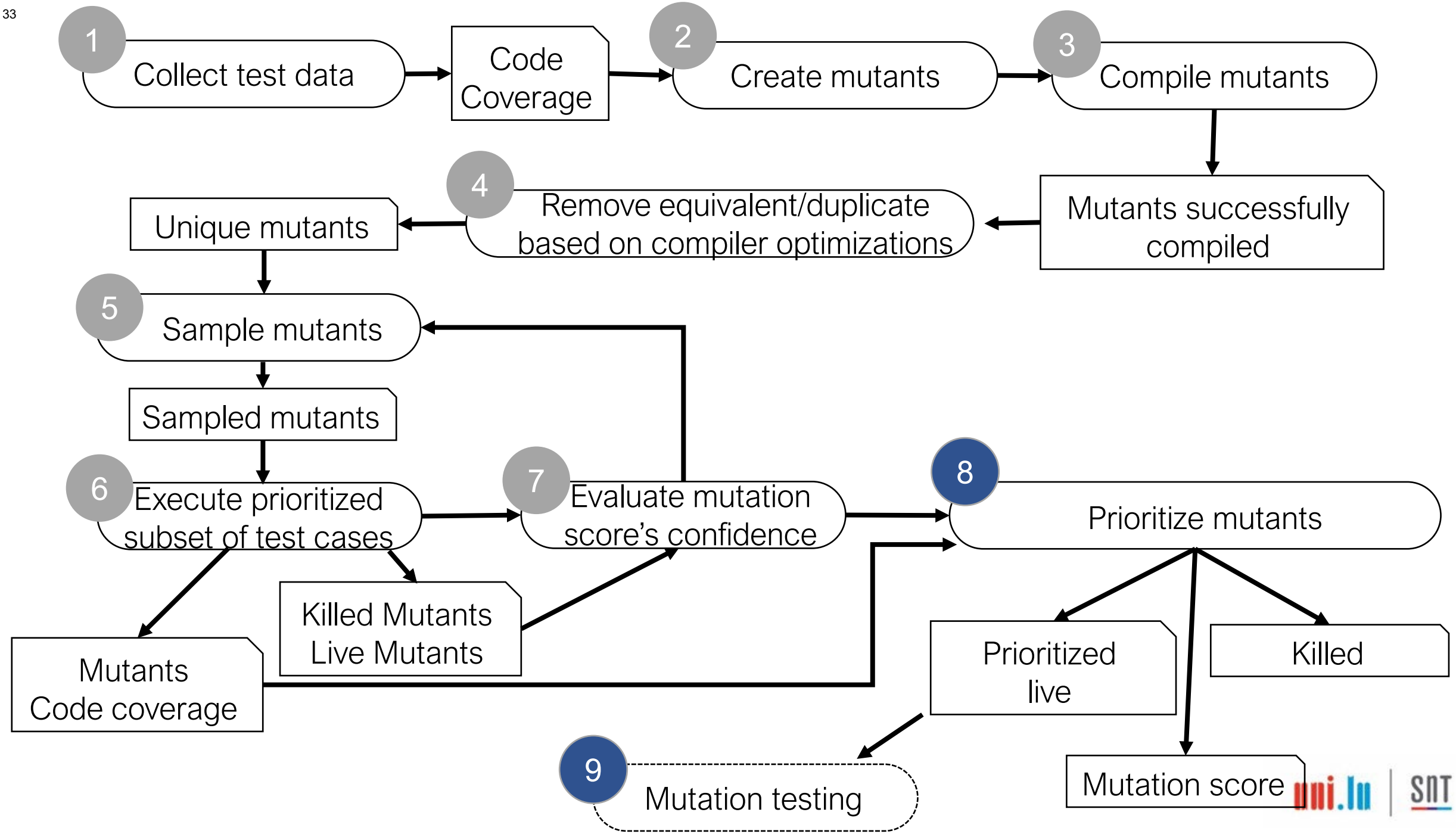


- Equivalent programs may lead to same executables after compiler optimizations
- We compile the original software and every mutant multiple times
  - once for each optimization option (i.e., -O0, -O1, -O2, -O3, -Os, -Ofast in GCC)
  - we compute the SHA-512 hash summary of the generated executable
  - we compare hash summaries


1
Collect test data
Code Coverage
2
Create mutants
3
Compile mutants
Mutants successfully compiled
4
Remove equivalent/duplicate based on compiler optimizations
Unique mutants
5
Sample mutants
Sampled mutants
6
Execute test cases
Killed Mutants
Live Mutants
Mutants
Code coverage


1
Collect test data
Code Coverage
2
Create mutants
3
Compile mutants
Mutants successfully compiled
4
Remove equivalent/duplicate based on compiler optimizations
Unique mutants
5
Sample mutants
Sampled mutants
6
Execute test cases
7
Evaluate mutation score's confidence
Killed Mutants
Live Mutants
Mutants
Code coverage

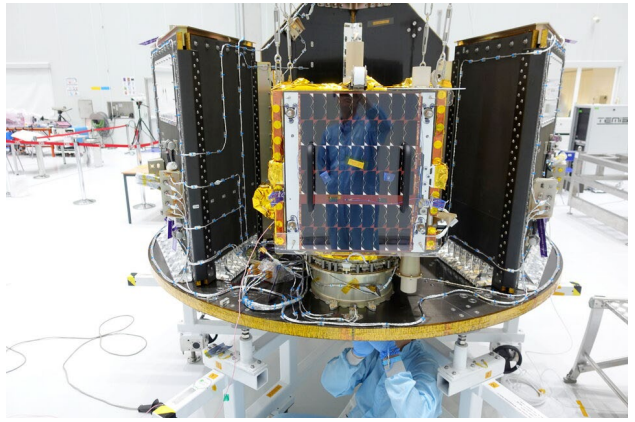

1
Collect test data
Code Coverage
2
Create mutants
3
Compile mutants
Mutants successfully compiled
4
Remove equivalent/duplicate based on compiler optimizations
Unique mutants
5
Sample mutants
Sampled mutants
6
Execute prioritized subset of test cases
7
Evaluate mutation score's confidence
8
Prioritize mutants
Killed Mutants
Live Mutants
Mutants
Code coverage
Prioritized
live
Killed
Mutation score

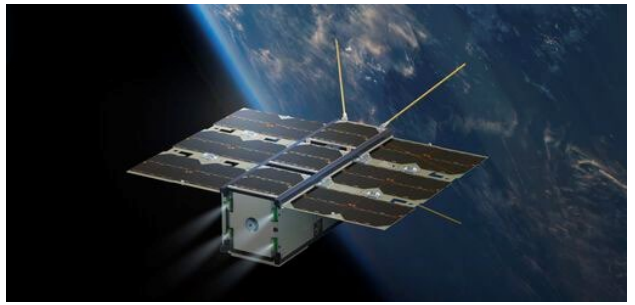

1
Collect test data
Code Coverage
2
Create mutants
3
Compile mutants
Mutants successfully compiled
4
Remove equivalent/duplicate based on compiler optimizations
Unique mutants
5
Sample mutants
Sampled mutants
6
Execute prioritized subset of test cases
7
Evaluate mutation score's confidence
8
Prioritize mutants
Killed Mutants
Live Mutants
Mutants
Code coverage
Prioritized
live
Killed
9
Mutation testing
Mutation score

Empirical Evaluation

Case study subjects

- Control software of LuxSpace's ESAIL satelite
- Control software of Huld's Bepicolombo satelite
- Network, Configuration, and Utility libraries from GomSpace's nanosatellites
- MLFS - Mathematical Library for Flight Software by ESA





Subject	LOC	Test suite	# Test Cases
ESAIL (subset)	2,235	System	384
LibNet	9,836	Integration	89
LibConf	3,179	Integration	170
LibUtil	10,576	Unit	201
MLFS	5,402	Unit	4042
Bepicolombo	30,000	System	133

Results

- For our largest subject, MASS makes mutation analysis feasible in **half a day** with 100 nodes in a grid infrastructure (more than 100 days without MASS)

- Overall key findings:
  - Discovered **lack of oracles**
    (e.g., do not verify all the entries of a structure)
  - reported **missing test cases** for exception handling code
  - detected **incomplete verification** of math formulae

```
# Copyright (c) University of Luxembourg 2021.
# Created by Oscar Eduardo CORNEJO OLIVARES, oscar.cornejo@uni.lu, SnT, 2021.
#

# set SRCIROR path
# example: $HOME/FAQAS/MASS
export SRCIROR=/home/vagrant/FAQAS/MASS

export LLVM_BUILD=${SRCIROR}/llvm-build
export SRCIROR_SRC_MUTATOR=${SRCIROR}/SRCMutation/build/mutator
export SRCIROR_LLVM_BIN=${LLVM_BUILD}/Release+Asserts/bin/
export SRCIROR_LLVM_INCLUDES=${LLVM_BUILD}/Release+Asserts/lib/clang/3.8.0/include/
export SRCIROR_COMPILER=$SRCIROR/PythonWrappers/mutationClang
export MASS=$SRCIROR/MASS

####

# set directory path where MASS files can be stored
# example: APP_RUN_DIR=/opt/example
export APP_RUN_DIR="/home/vagrant/mass_workspace"

# specifies the building system, available options are "Makefile" and "waf"
# example: BUILD_SYSTEM="Makefile"
export BUILD_SYSTEM="waf"

# directory root path of the software under test
# example: PROJ=$HOME/mlfs
export PROJ=/home/vagrant/libutil

# directory src path of the SUT
# example: PROJ_SRC=$PROJ/libm
export PROJ_SRC=/home/vagrant/libutil/src

# directory test path of the SUT
# example: PROJ_TST=$HOME/unit-test-suite
export PROJ_TST=/home/vagrant/libutil/tst

# directory coverage path of the SUT
# example: PROJ_COV=$HOME/blts_workspace
export PROJ_COV=$PROJ_TST

# directory path of the compiled binary
# example: PROJ_BUILD=$PROJ/build-host/bin
export PROJ_BUILD=/home/vagrant/libutil/build/

# list of folders not to be included during the analysis
# example: COVERAGE_NOT_INCLUDE="tst\|libutil\|libgscsp\|libparam_client"
export COVERAGE_NOT_INCLUDE=

# filename of the compiled file/library
# example: COMPILED=libmlfs.a
export COMPILED=libgsutil.so

# path to original Makefile
```

```
Len of  time.mut.45.2_5_35.ROR.gs_time_sleep_ns|src/linux/time.c 1
size list A: 3
size list B: 6
size list equivalents: 0
MutationScore
step number 7
waiting for IdentifyEquivalents to be completed
Executing MutationScore...
##### MASS Output #####
## Total mutants generated: 639
## Total mutants filtered by TCE: 171
## Sampling type: uniform
## Total mutants analyzed: 47
## Total killed mutants: 38
## Total live mutants: 9
## Total likely equivalent mutants: 0
## MASS mutation score (%): 80.85
## List A of useful undetected mutants: /home/vagrant/mass_workspace/DETECTION/test_ru
## List B of useful undetected mutants: /home/vagrant/mass_workspace/DETECTION/test_ru
## Number of statements covered: 118
## Statement coverage (%): 4.15
## Minimum lines covered per source file: 0
## Maximum lines covered per source file: 32
```

Problems Addressed

- How to make mutation analysis scale?
- How to assess if test suites verify components integration properly?
- How to generate test cases th

https://github.com/SNTSVV/DAMAT

2182 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 49, NO. 4, APRIL 2023

Data-Driven Mutation Analysis for Cyber-Physical Systems

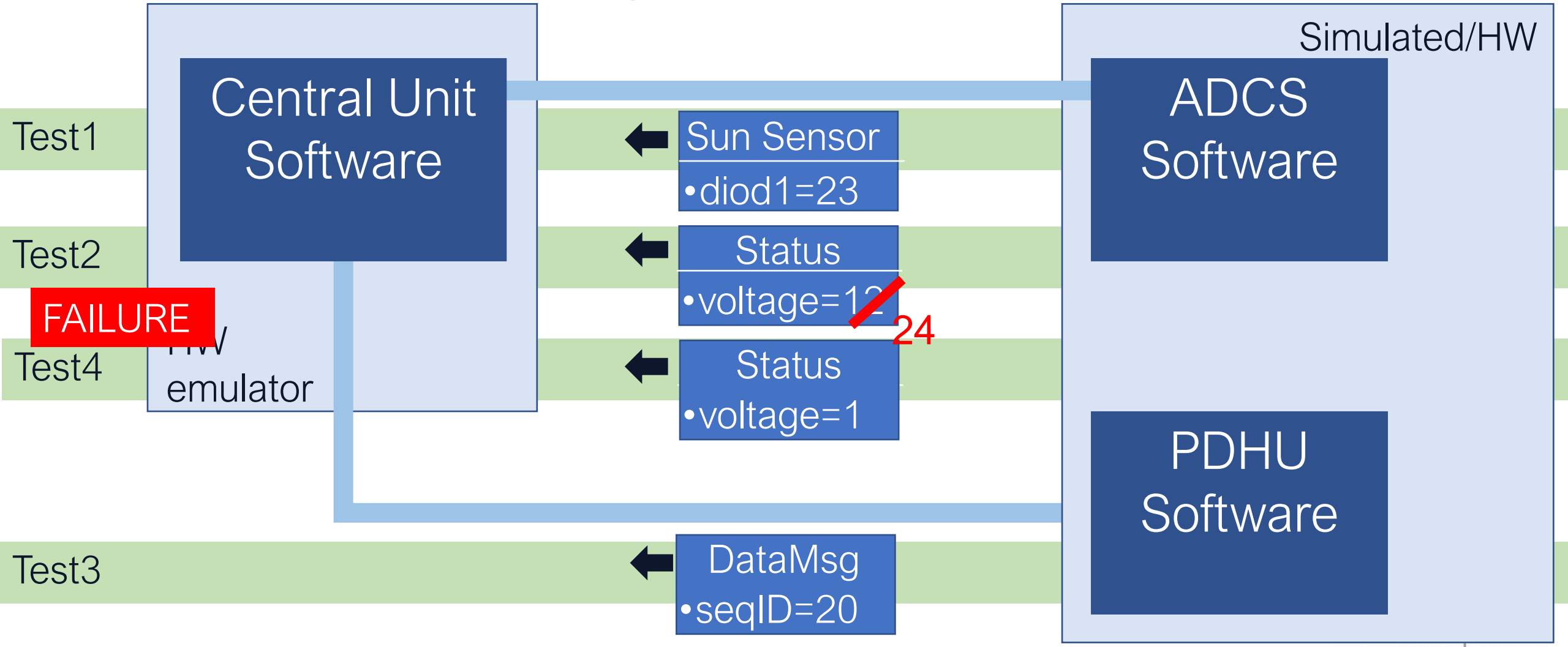
Enrico Viganò, Oscar Cornejo, Fabrizio Pastore, *Member, IEEE*, and Lionel C. Briand, *Fellow, IEEE*

Abstract—Cyber-physical systems (CPSs) typically consist of a wide set of integrated, heterogeneous components; consequently, most of their critical failures relate to the interoperability of such components. Unfortunately, most CPS test automation techniques are preliminary and industry still heavily relies on manual testing. With potentially incomplete, manually-generated test suites, it is of paramount importance to assess their quality. Though mutation analysis has demonstrated to be an effective means to assess test suite quality in some specific contexts, we lack approaches for CPSs. Indeed, existing approaches do not target interoperability problems and cannot be executed in the presence of black-box or simulated components, a typical situation with CPSs. In this article, we introduce *data-driven mutation analysis*, an approach that consists in assessing test suite quality by verifying if it detects interoperability faults simulated by mutating the data exchanged by software components. To this end, we describe a data-driven mutation analysis technique (*DaMAT*) that automatically alters the data exchanged through data buffers. Our technique is driven by fault models in tabular form where engineers specify how to mutate data items by selecting and configuring a set of mutation operators. We have evaluated *DaMAT* with CPSs in the space domain; specifically, the test suites for the software systems of a microsatellite and nanosatellites launched on orbit last year. Our results show that the approach effectively detects test suite shortcomings, is not affected by equivalent and redundant mutants, and entails acceptable costs.

Index Terms—Cyber-physical systems, CPS interoperability, integration testing, mutation analysis

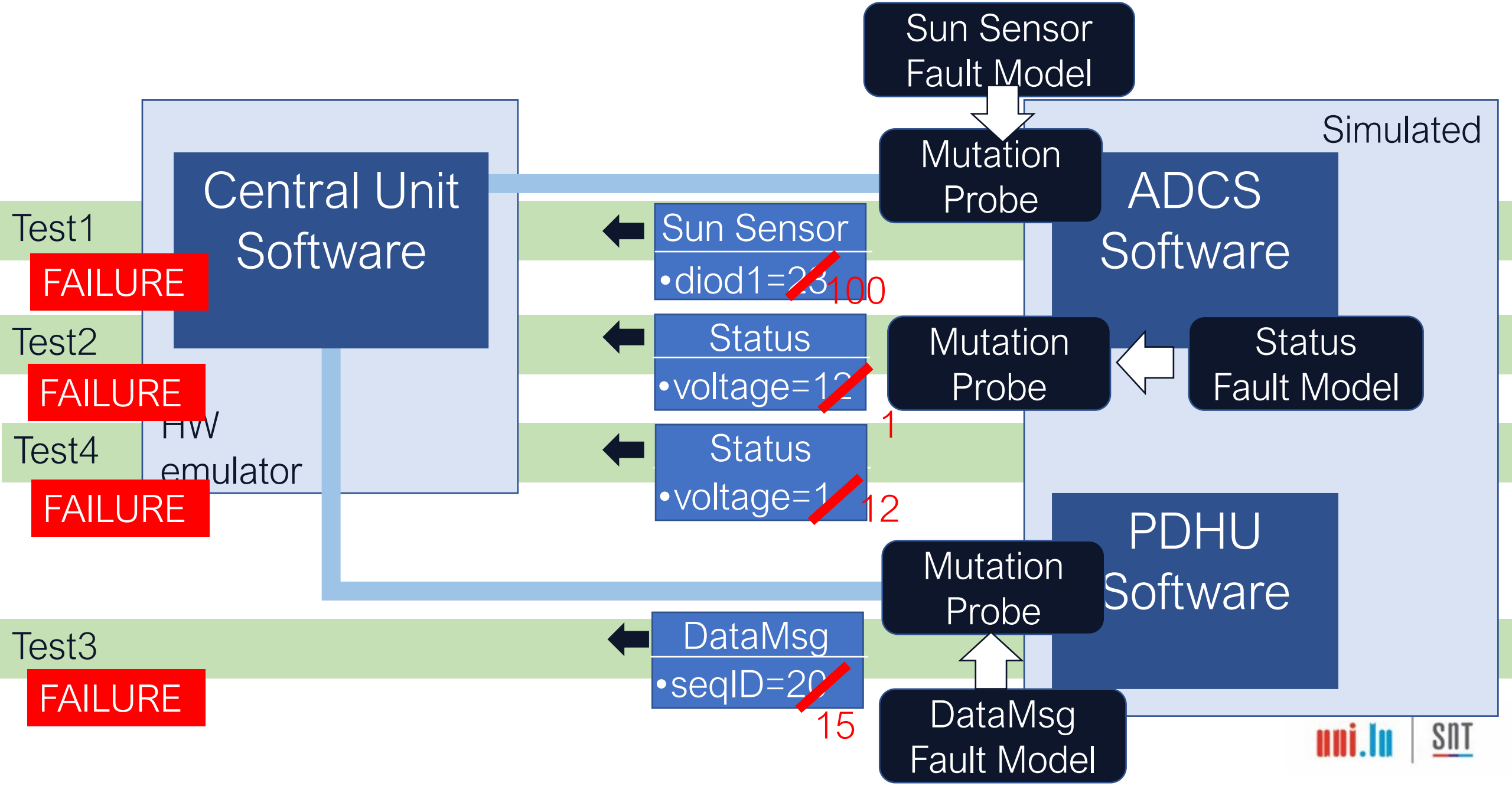
CPS Components

Testing CPS Software

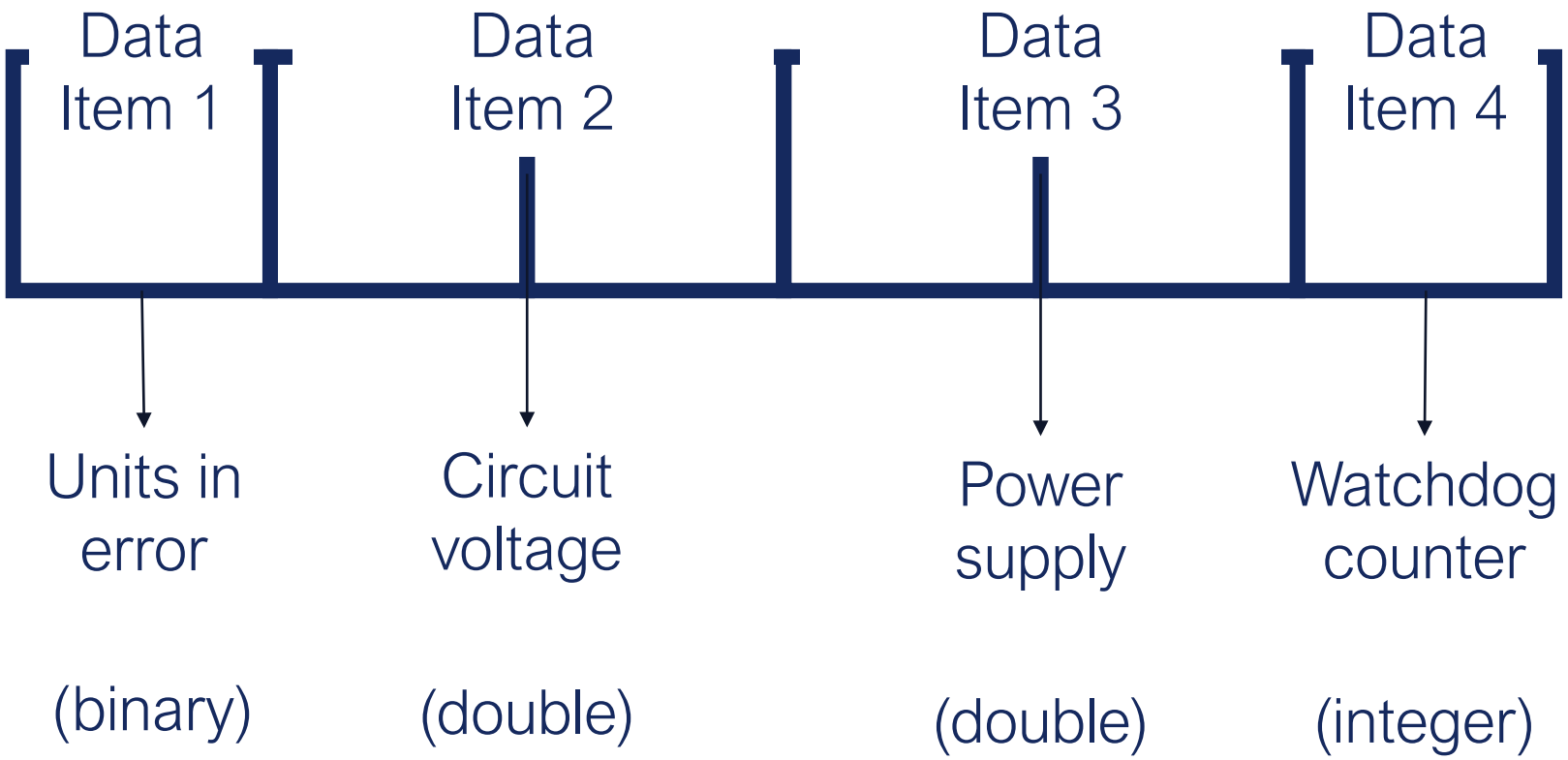


How to assess if a test suite exercises components' integration properly?

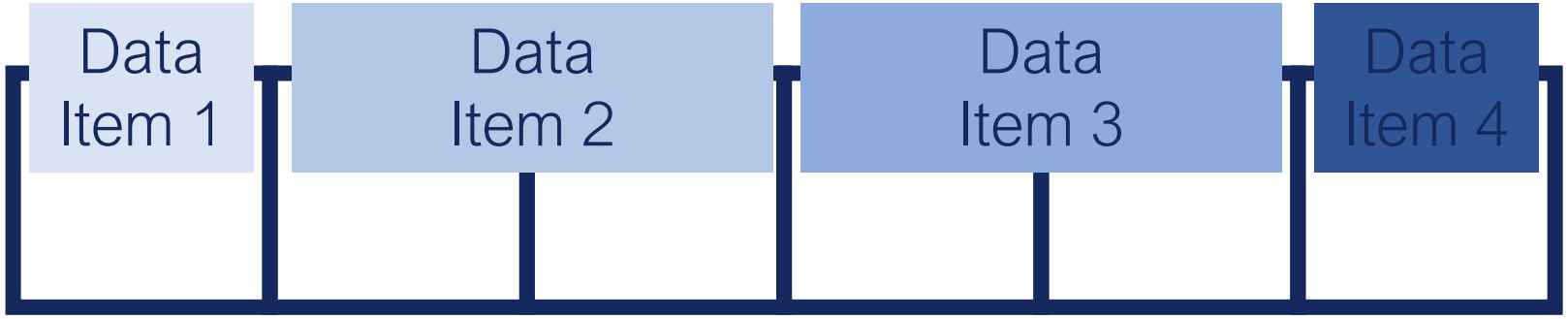
Data-driven Mutation Analysis


Sun Sensor Fault Model
Simulated
Mutation Probe
ADCS Software
Central Unit Software
Test1
FAILURE
Sun Sensor
•diod1=23
100
Test2
FAILURE
Status
•voltage=12
Mutation Probe
Status Fault Model
1
HW emulator
Test4
FAILURE
Status
•voltage=1
12
PDHU Software
Mutation Probe
Test3
FAILURE
DataMsg
•seqID=20
15
DataMsg Fault Model

Message Data Buffers



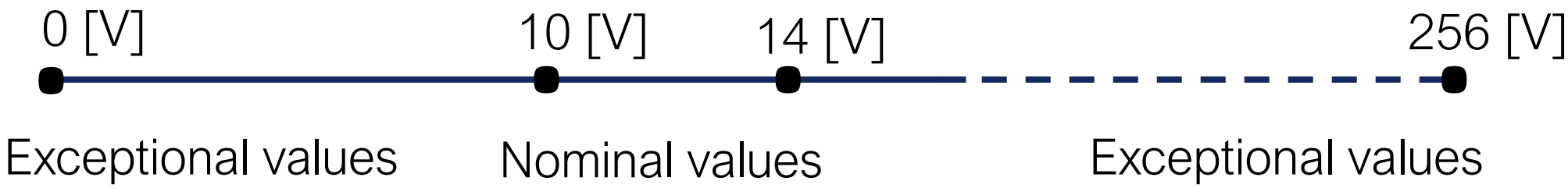
From Fault Model to Mutants



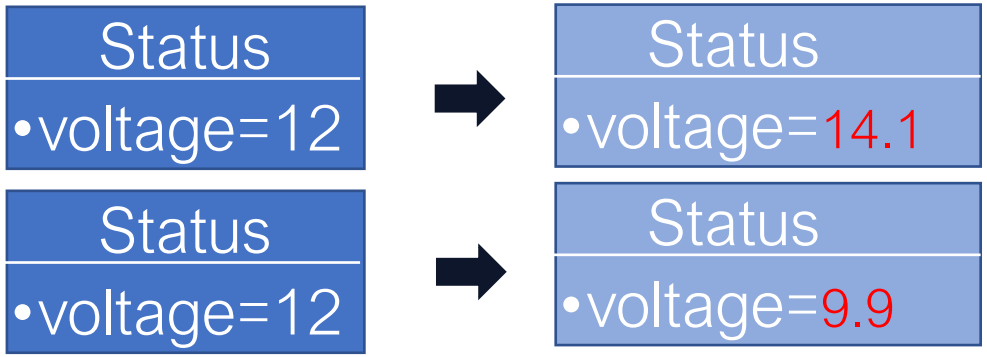
Position	Span	Operator	Delta	Min	Max
0	1	Bit flip			
1	2	Value Out of Range	0.1	10	14
1	2	Fix Value Out of Range	0.1	10	14
3	2	Value Above Threshold	0.1		6
5	1	Value Above Threshold	1		10

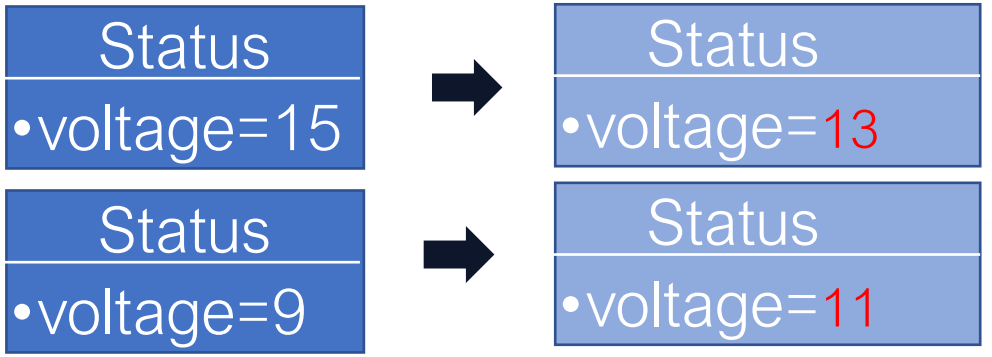
- Mut 1: flip bits
- Mut 2: set to 14.1
- Mut 3: set to 9.9
- Mut 4: set > to [10,14]
- Mut 5: set< to [10,14]
- Mut 6: set to 6.1
- Mut 7: set to 10.1

Example: Stateless Numerical



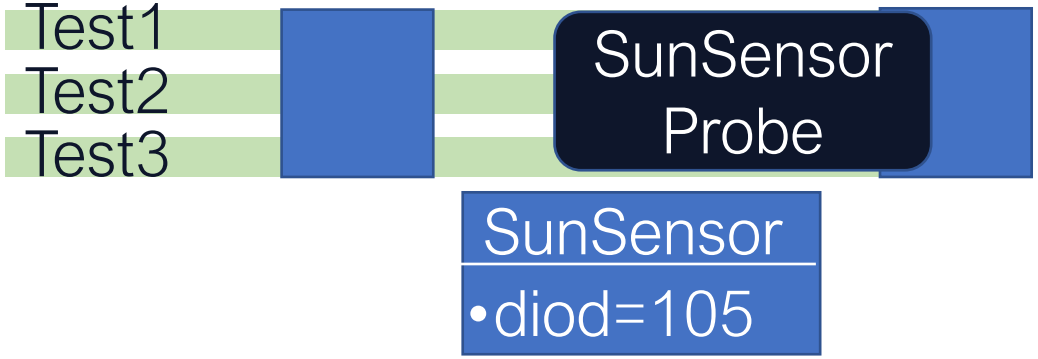
Operator	Delta	Min	Max
Value Out of Range	0.1	10	14



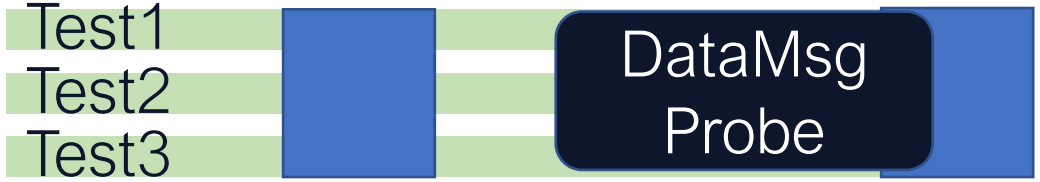


Mutants

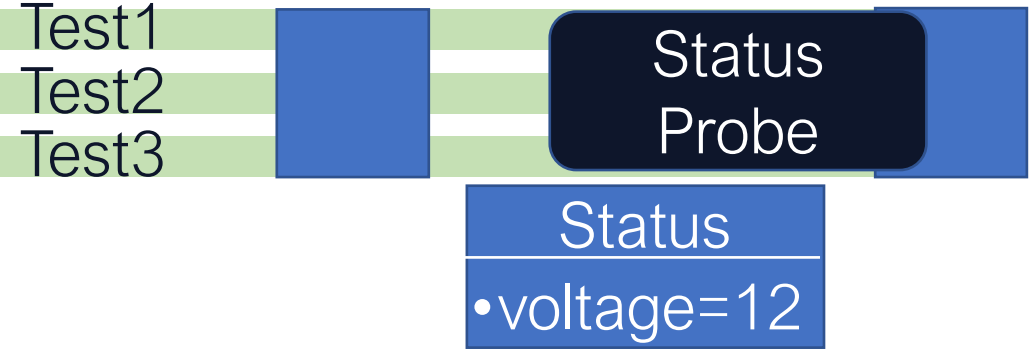
Mut1: set *diod* above 100 threshold



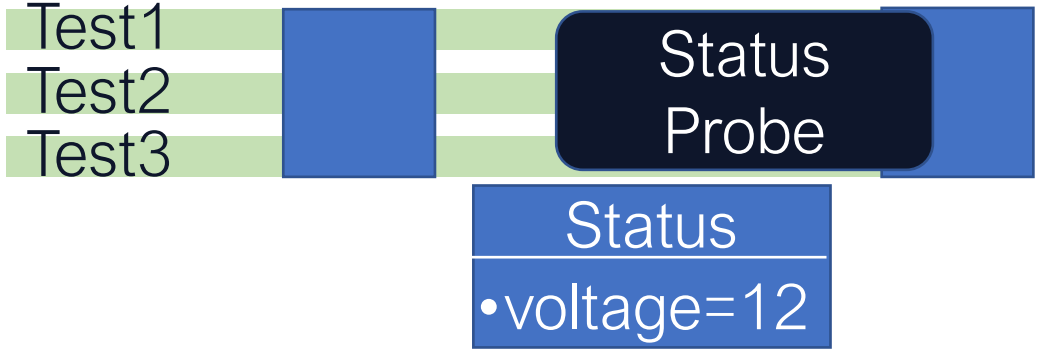
Mut2: replace *seqID* with random



Mut3: set *voltage* above range

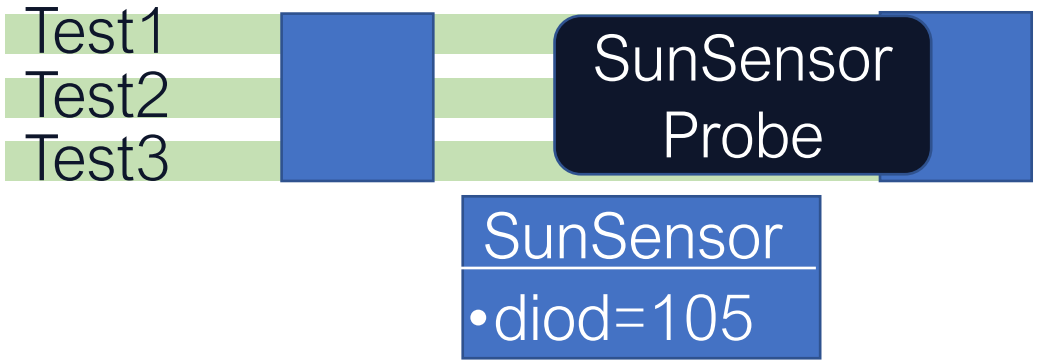


Mut4: set *voltage* below range

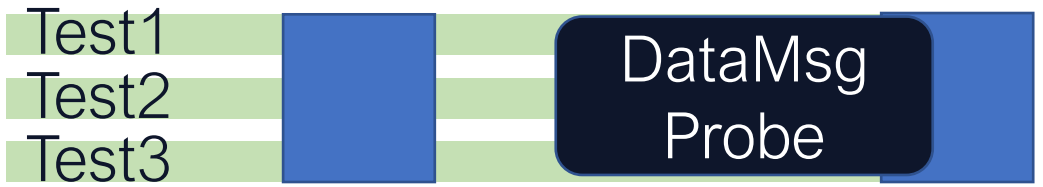


All Mutants are Tested

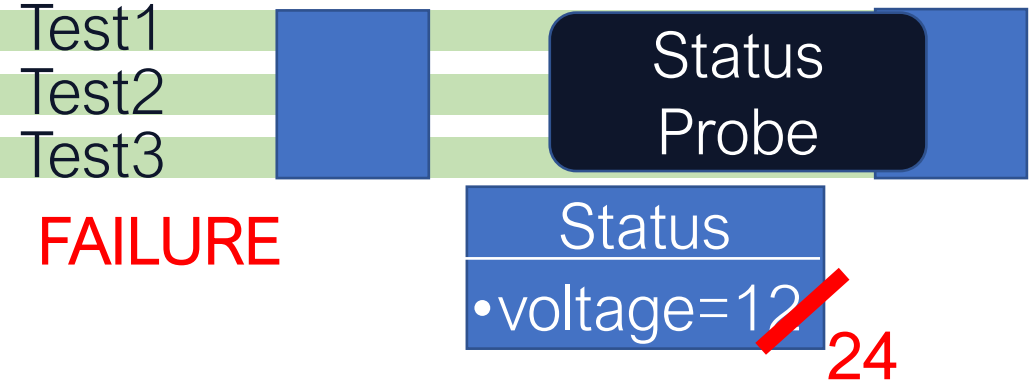
Mut1: set above 100 threshold



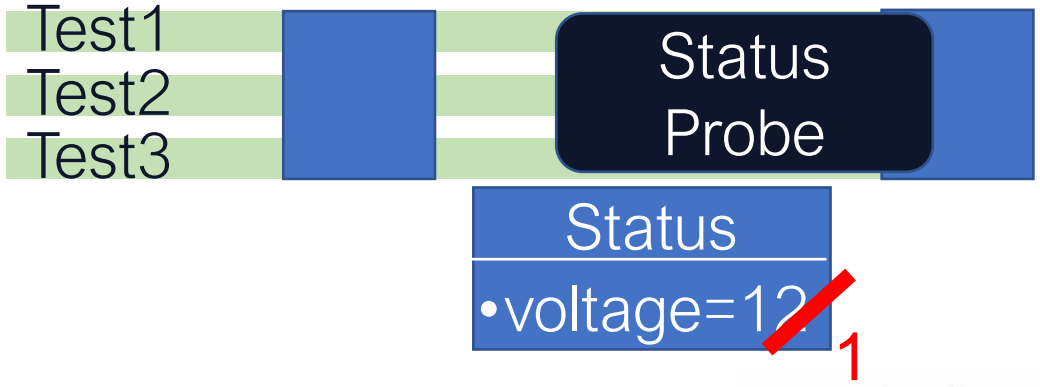
Mut2: replace with random



Mut3: set above range



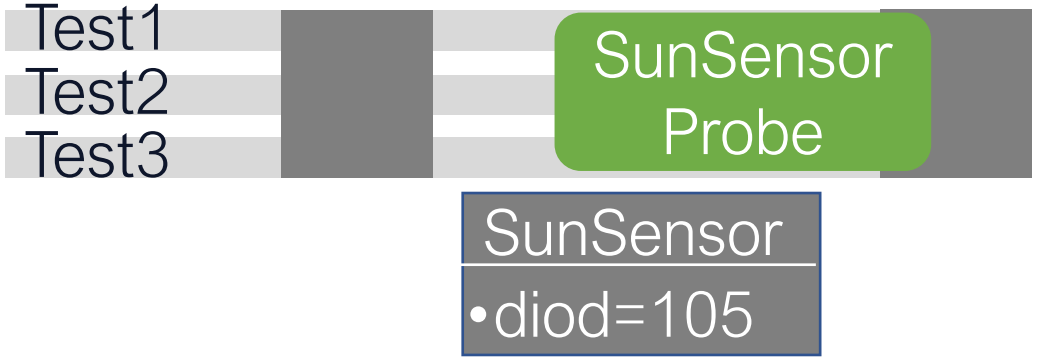
Mut4: set below range



Three Metrics

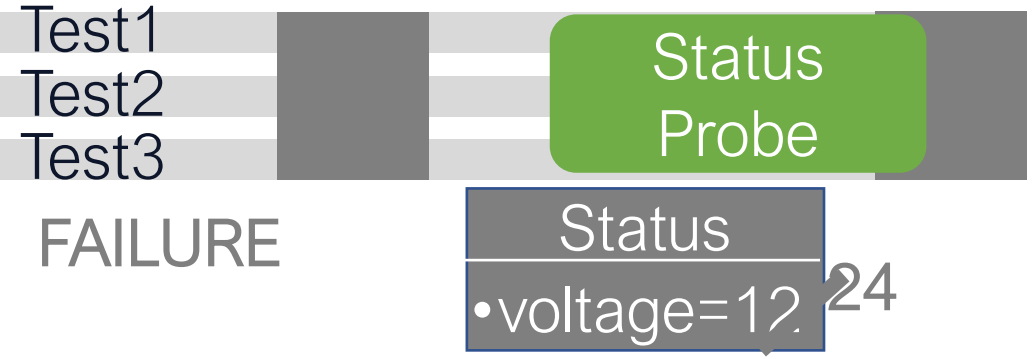
Fault Model Coverage:
% fault models covered (%message types exercised)

Mut1: set above 100 threshold

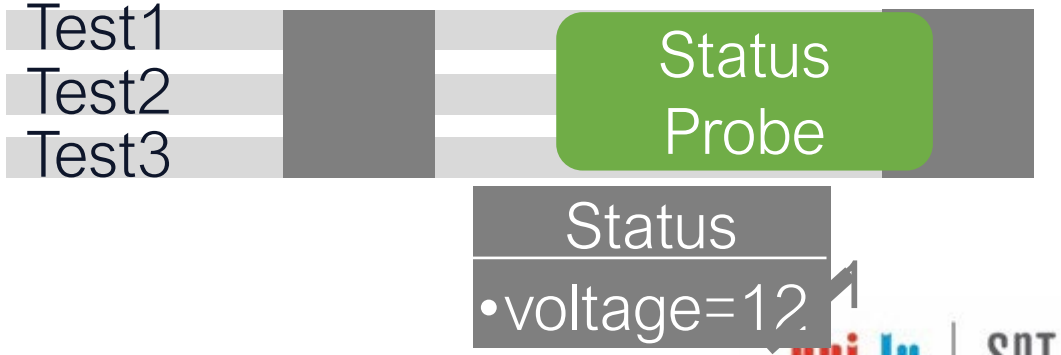


Mut2: replace with random

Mut3: set above range



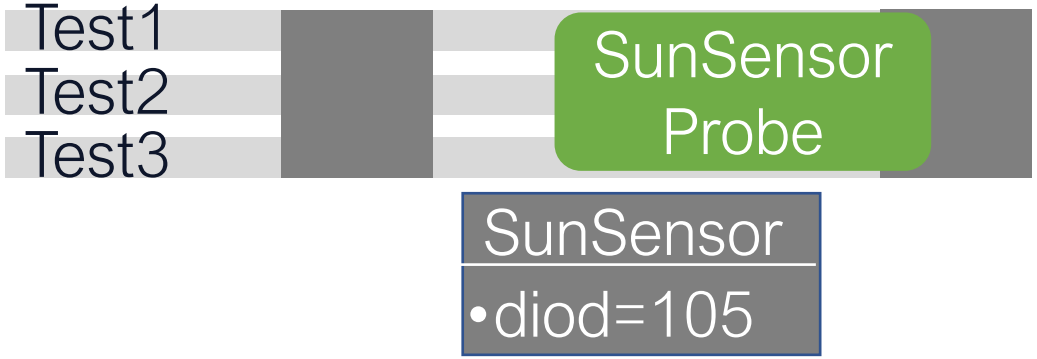
Mut4: set below range



⅔=66.6%

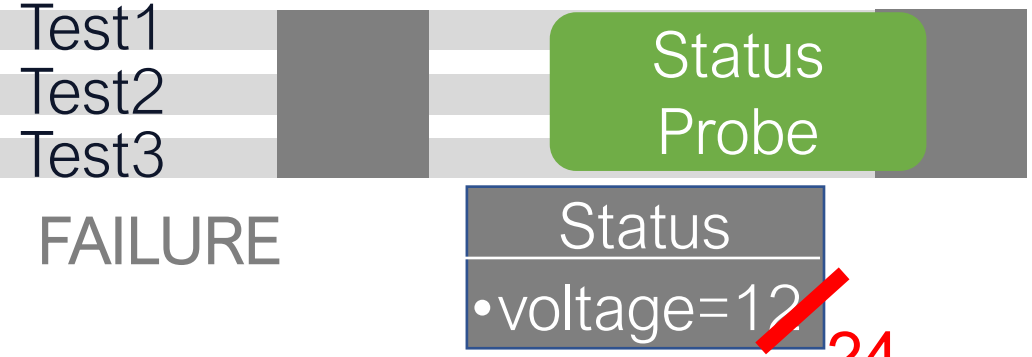
Mutation Operation Coverage:
% executed probes that mutated some data item (% input partitions being exercised)

Mut1: set above 100 threshold



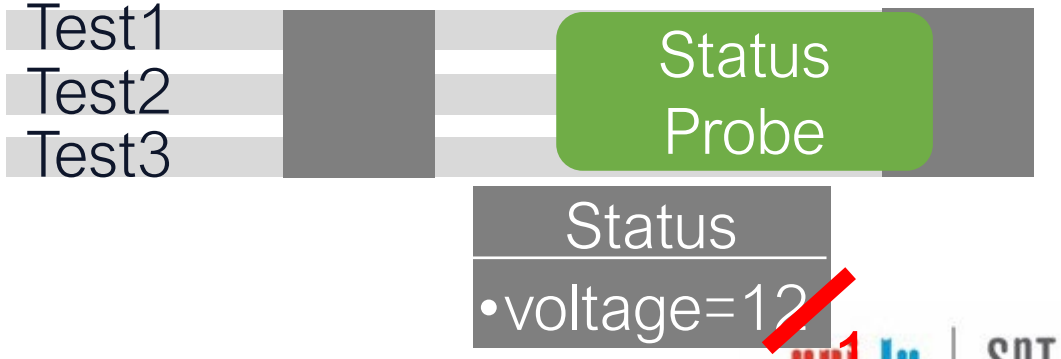
Mut2: replace with random

Mut3: set above range



FAILURE

Mut4: set below range

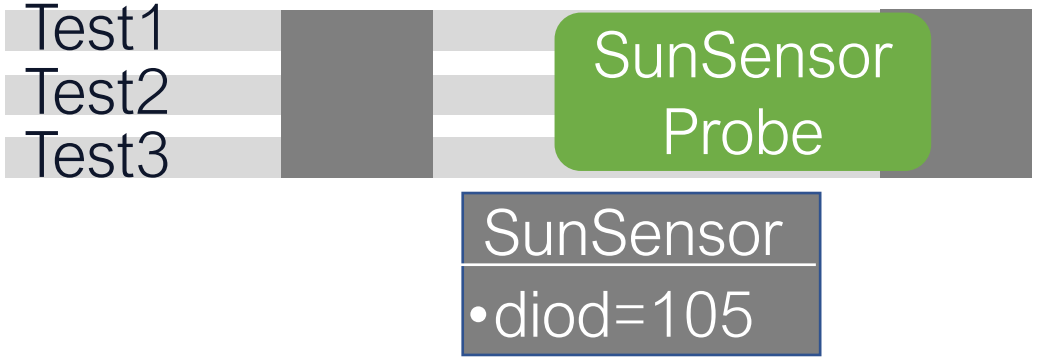


⅔=66.6%

Covered Mutation Score:

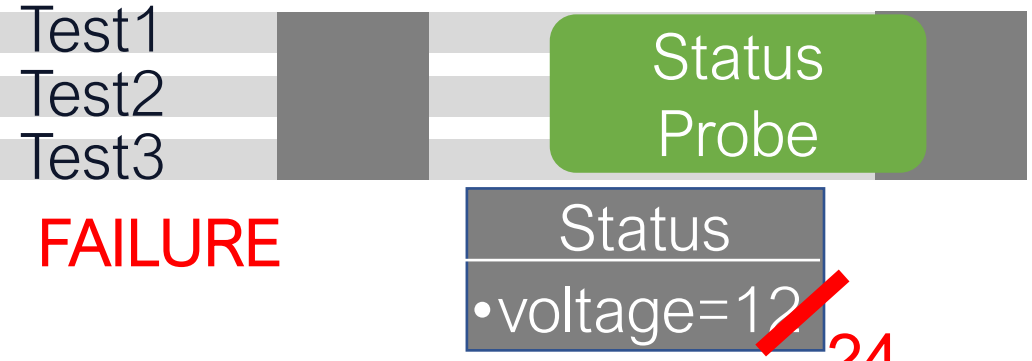
% mutants that mutated some data and lead to failures

Mut1: set above 100 threshold

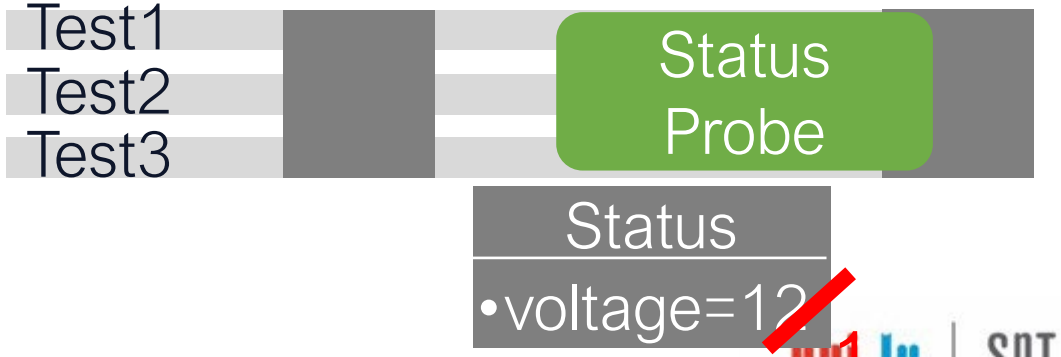


Mut2: replace with random

Mut3: set above range



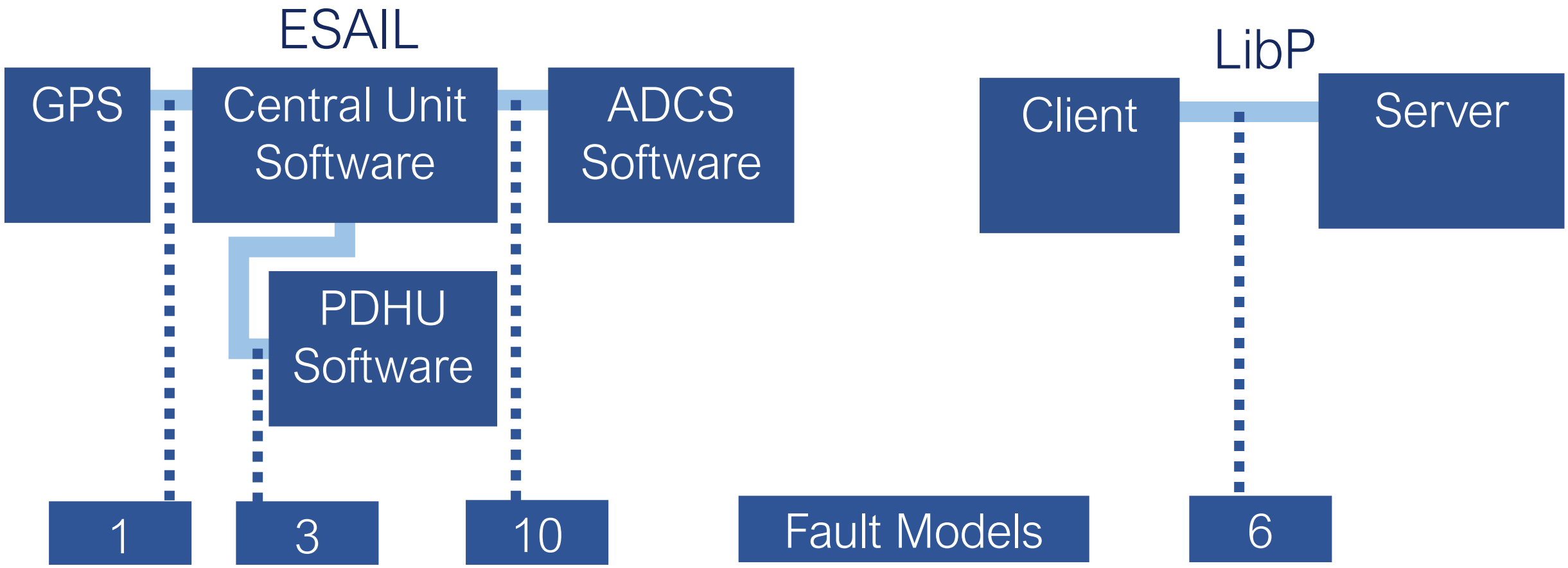
Mut4: set below range



½=50.0%

Empirical Assessment

Subjects


ESAIL
GPS
Central Unit Software
ADCS Software
PDHU Software
1
3
10
Fault Models
LibP
Client
Server
6

Empirical Results

- Successfully identified: uncovered message types (1), uncovered input partitions (48), and poor oracles (60).

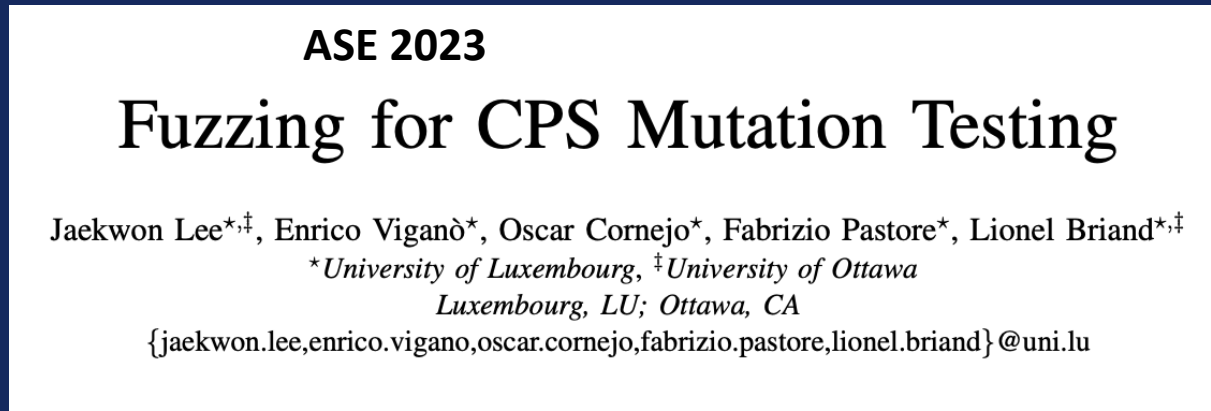
- Live mutants can be killed by introducing oracles that:
  - verify additional entries in the log files
  - verify additional observable state variables
- verify not only the presence of error messages but also their content.

- The configuration of our toolset
(configure operators and insert mutation probes)
for each subject took between 3 and 20 working hours.

Problems Addressed

- How to make mutation analysis scale?
- How to assess if test suites verify components integration properly?
- How to generate test cases that kill mutants in C software?

https://github.com/SNTSVV/MOTIF

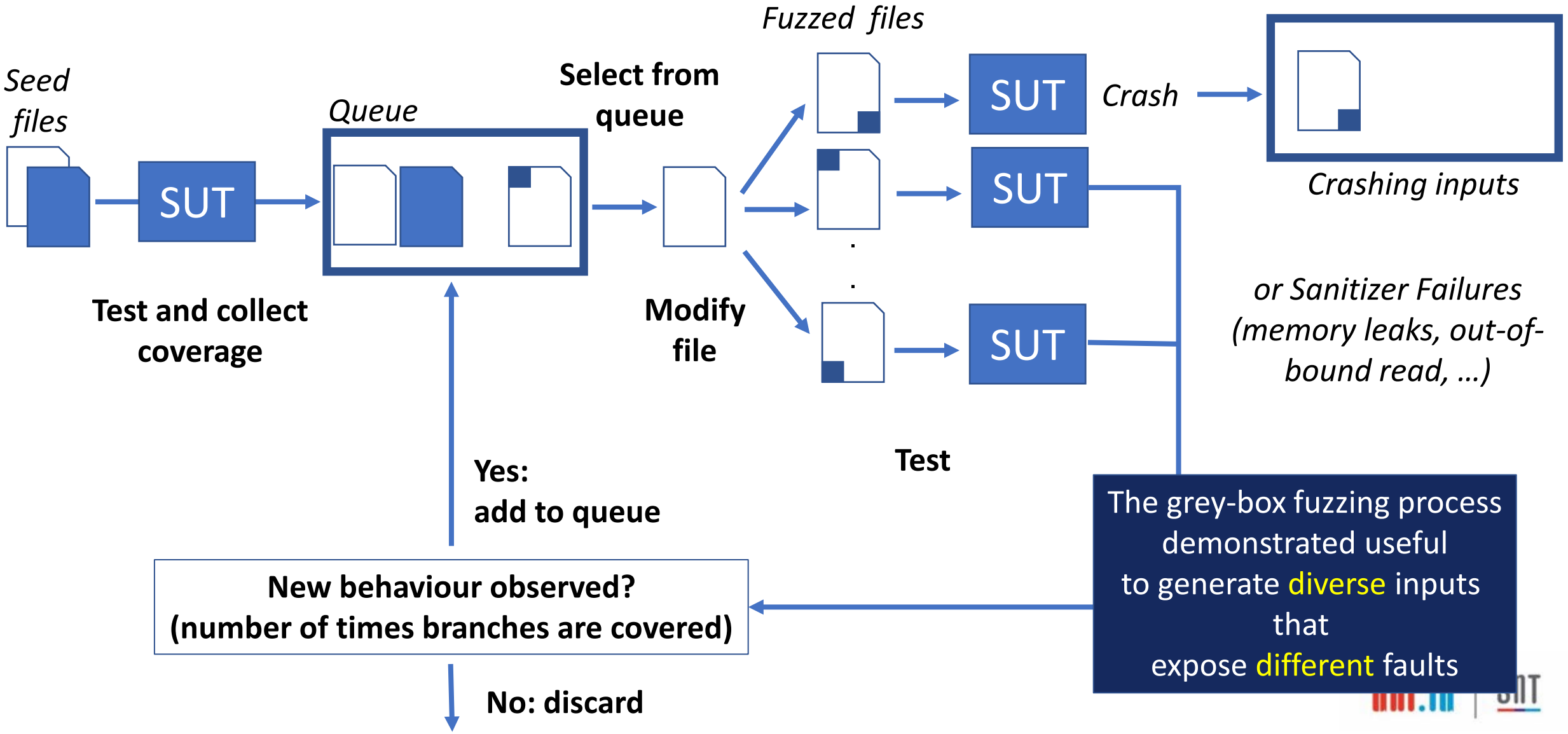


ASE 2023

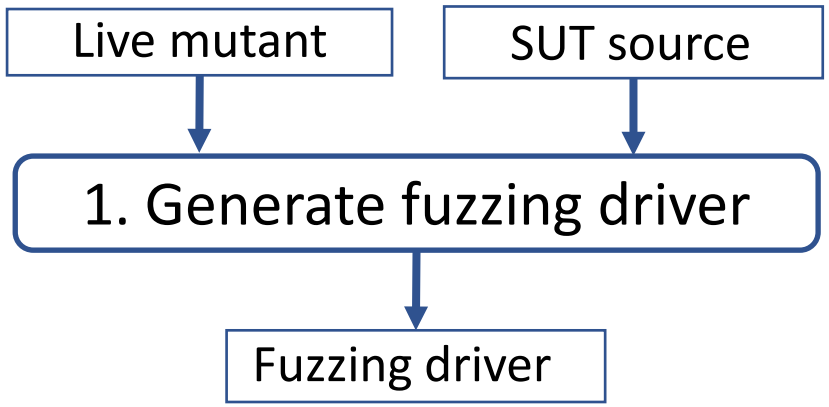
Fuzzing for CPS Mutation Testing

Jaekwon Lee*,‡, Enrico Viganò*, Oscar Cornejo*, Fabrizio Pastore*, Lionel Briand*,‡
*University of Luxembourg, ‡University of Ottawa
Luxembourg, LU; Ottawa, CA
{jaekwon.lee,enrico.vigano,oscar.cornejo,fabrizio.pastore,lionel.briand}@uni.lu

Grey-box Fuzzing: An Evolutionary Testing Approach

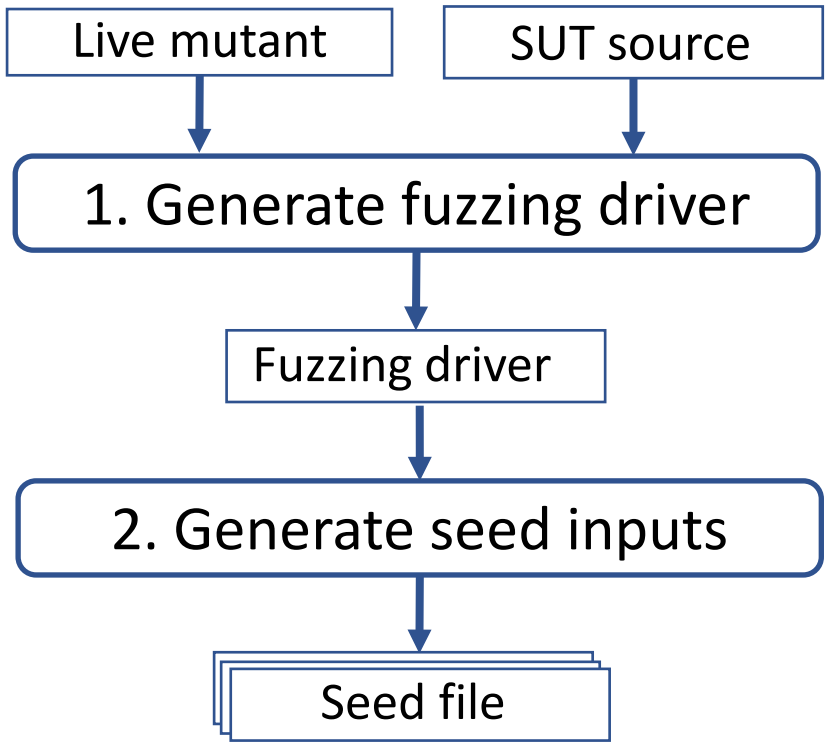


MutatiOn Testing wIth Fuzzing (MOTIF)




```
int main(...){
  double x = load(..);
  double y = load(..);
  int z = load(..);

  double m_x = load(..);
  double m_y = load(..);
  int m_z = load(..);

  ret = max(x,y,z); //invoke original
 //invoke mutated
  mut_ret = mut_max(m_x,m_y,m_z);

  if( ! match ( ret, mut_ret ) ){ abort(); }
  if( ! match (x, m_x ){ abort() };
  if( ! match (y, m_y) { abort() };
  if( ! match (z, m_z) { abort() };
```

MutatiOn Testing wIth Fuzzing (MOTIF)



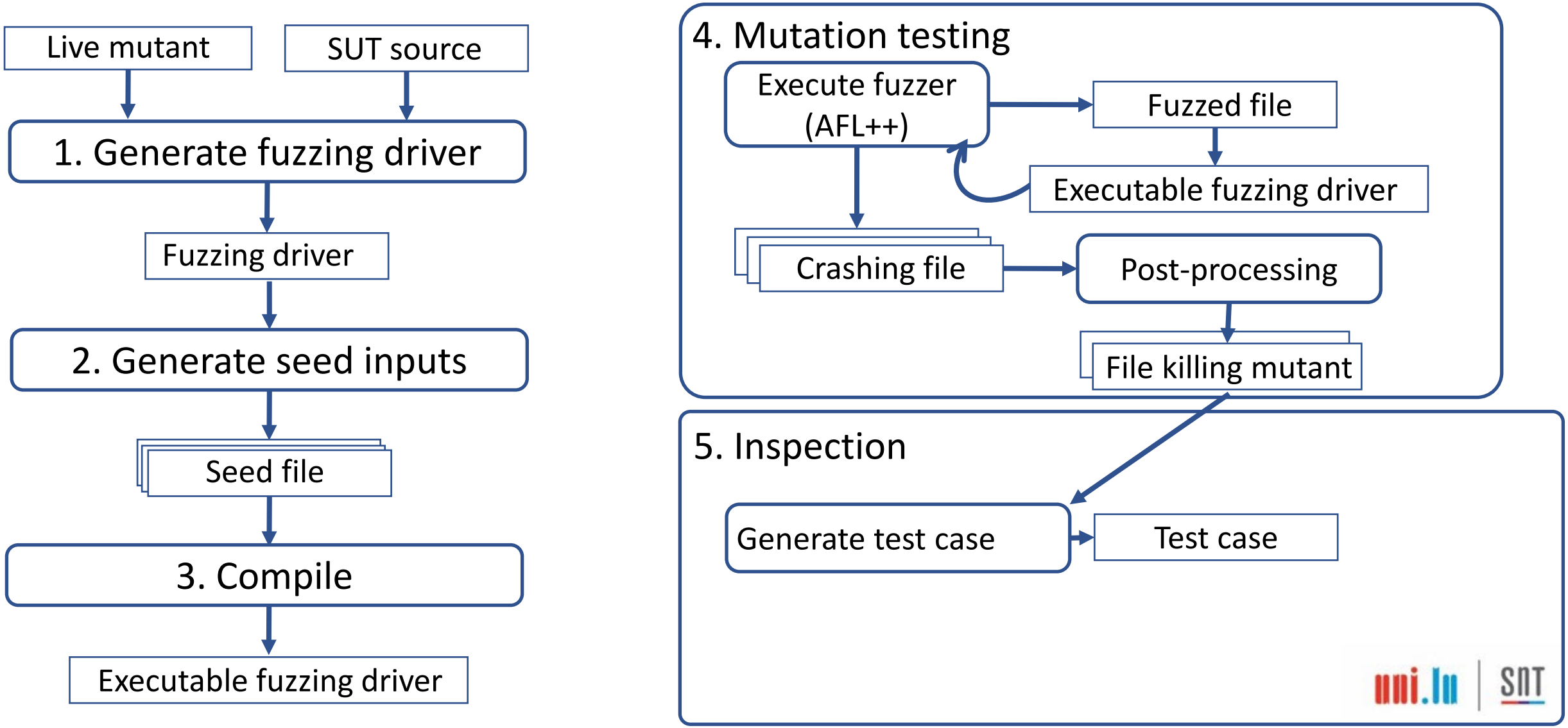

```
1011111111110000000
1101111111110000000
1111111111111111111
```

```
int max(double x, double y, int z){
```

```
1111111111111111111
1111111111111111111
1111111111111111111
0000000000000000001
```

```
int is_valid(struct T_POS p, int p){
```

MutatiOn Testing wIth Fuzzing (MOTIF)

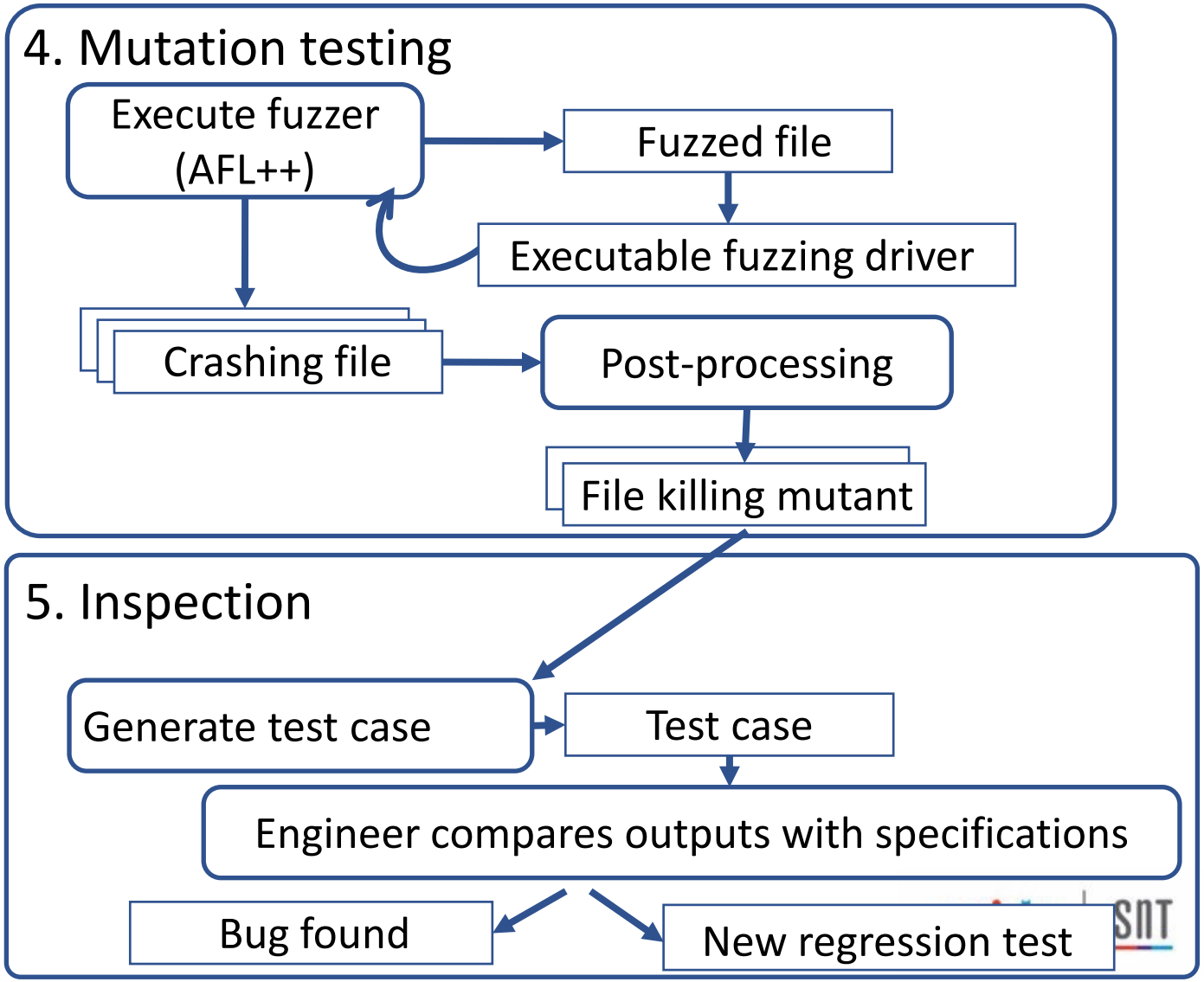


```c
/*************************************************
* Define input and expected values in hex
*************************************************/
const char input_data_base[8] = {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};
const char input_data_add[8] = {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF};

const char expected_base[8] = {0xFF,0xFF,0xFF,0xFF,0xFE,0xD7,0x94,0x1 };
const char expected_add[8] = {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF };

const char expected_return[4] = {0x00,0x00,0x00,0x00};

/*************************************************
*  Entry for test driver
*************************************************/
int main(int argc, char** argv)
{
    (void)argc;
    (void)argv;

    /* Declare variable to hold function returned value */
    int function_return;

    /* declaring the input variables for  function */
    gs_timestamp_t base;
    gs_timestamp_t add;

    /* Copy data  */
    memcpy(&base, input_data_base, sizeof(base));
    memcpy(&add, input_data_add, sizeof(add));

    /* Print parameter values of the  function */
    printf("\n");
    printf_hex("MOTIF-INPUT: base (gs_timestamp_t) = ", "\n", (char *)&base, sizeof(base));
    printf_hex("MOTIF-INPUT: add (gs_timestamp_t) = ", "\n", (char *)&add, sizeof(add));


    /* Calling the  function under test */
    printf("Calling the function... \n");
    function_return = timestamp_add(&base, &add);


    /* Print parameter values of the  function */
    printf("\n");
    printf_hex("MOTIF-FUNCTION-OUTPUT: base (gs_timestamp_t) = ", "\n", (char *)&base, sizeof(base));
    printf_hex("MOTIF-FUNCTION-OUTPUT: add (gs_timestamp_t) = ", "\n", (char *)&add, sizeof(add));
    printf("MOTIF-FUNCTION-OUTPUT: function_return (int) = %d\n\n", function_return);


    assert(0==compare((char *)&base, expected_base, sizeof(base)));
    assert(0==compare((char *)&add, expected_add, sizeof(add)));
    assert(0==compare((char *)&function_return, expected_return, sizeof(function_return)));


    printf("PASS\n");

    return 0;
}
```

Assign fuzzer inputs to input variables

Inspect results

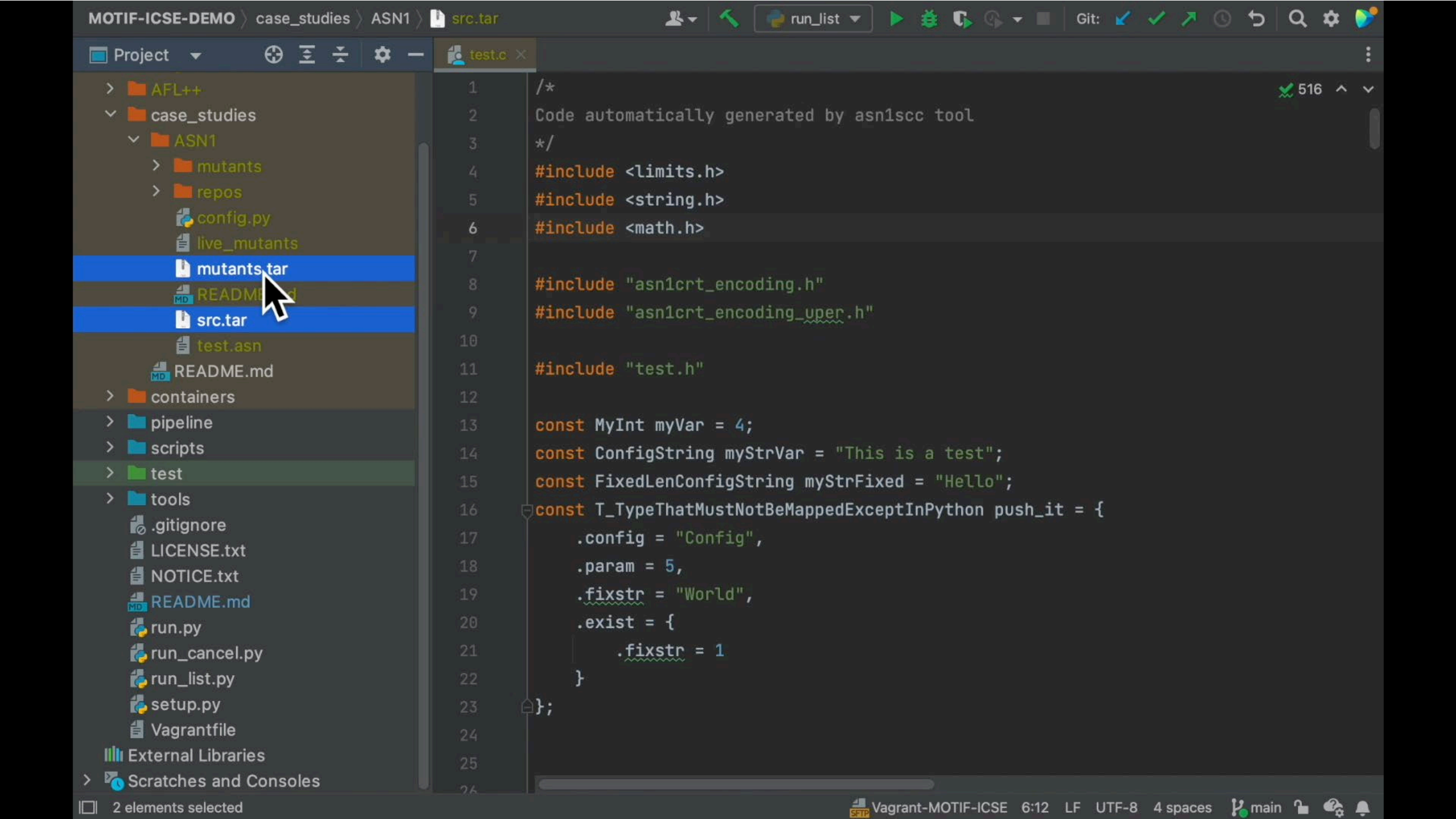
g wlth Fuzzing (MOTIF)



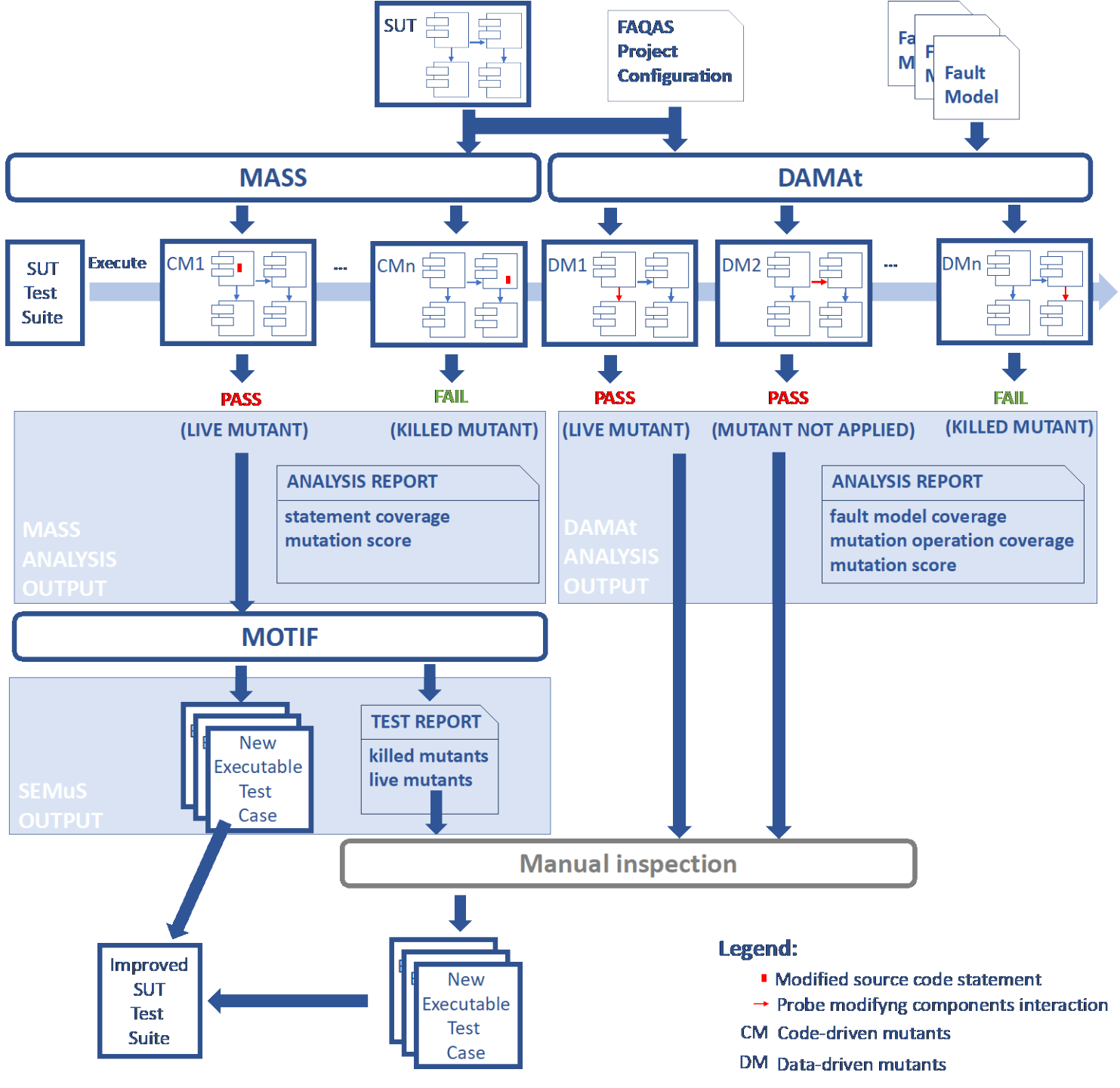
Empirical Results

Subject	Live mutants	Mutants killed by MOTIF	Generated test cases
ASN1Lib	1,347	1,161 – 86.19%	350
MLFS	3,891	1,392 – 35.77%	410
Satellite CSW	581	203 – 34.90%	203
Utility library	443	234 – 52.82%	64

- Effectiveness vary from subject to subject
  - Mathematical functions with narrow constraints are harder to test
- We identified 5 faults in one subject


MOTIF-ICSE-DEMO
case_studies
ASN1
src.tar
run_list
Git:
Project
AFL++
case_studies
ASN1
mutants
repos
config.py
live_mutants
mutants.tar
README.md
src.tar
test.asn
README.md
containers
pipeline
scripts
test
tools
.gitignore
LICENSE.txt
NOTICE.txt
README.md
run.py
run_cancel.py
run_list.py
setup.py
Vagrantfile
External Libraries
Scratches and Consoles
test.c
516
/*
Code automatically generated by asn1scc tool
*/
#include <limits.h>
#include <string.h>
#include <math.h>
#include "asn1crt_encoding.h"
#include "asn1crt_encoding_uper.h"
#include "test.h"
const MyInt myVar = 4;
const ConfigString myStrVar = "This is a test";
const FixedLenConfigString myStrFixed = "Hello";
const T_TypeThatMustNotBeMappedExceptInPython push_it = {
.config = "Config",
.param = 5,
.fixstr = "World",
.exist = {
.fixstr = 1
}
};
2 elements selected
Vagrant-MOTIF-ICSE
6:12
LF
UTF-8
4 spaces
main

FAQAS Methodology

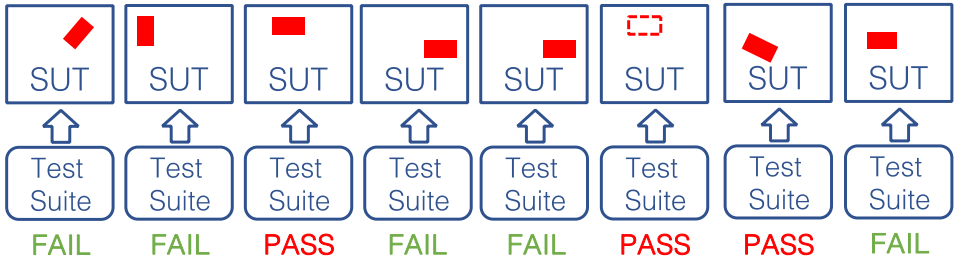


http://faqas.uni.lu

How to ensure thorough testing?

5

Mutation Analysis and Testing



Improve with Automatically Generated Test Cases

18

Problems Addressed

- How to make mutation analysis scale?
- How to assess if test suites verify components integration properly?
- How to generate test cases that kill mutants in C software?

uni.lu | SNT

Ongoing work

- Guidelines for mutation score: what to target?
- Integration with known frameworks (libcheck, googletest)
- Automated configuration for data-driven mutation analysis

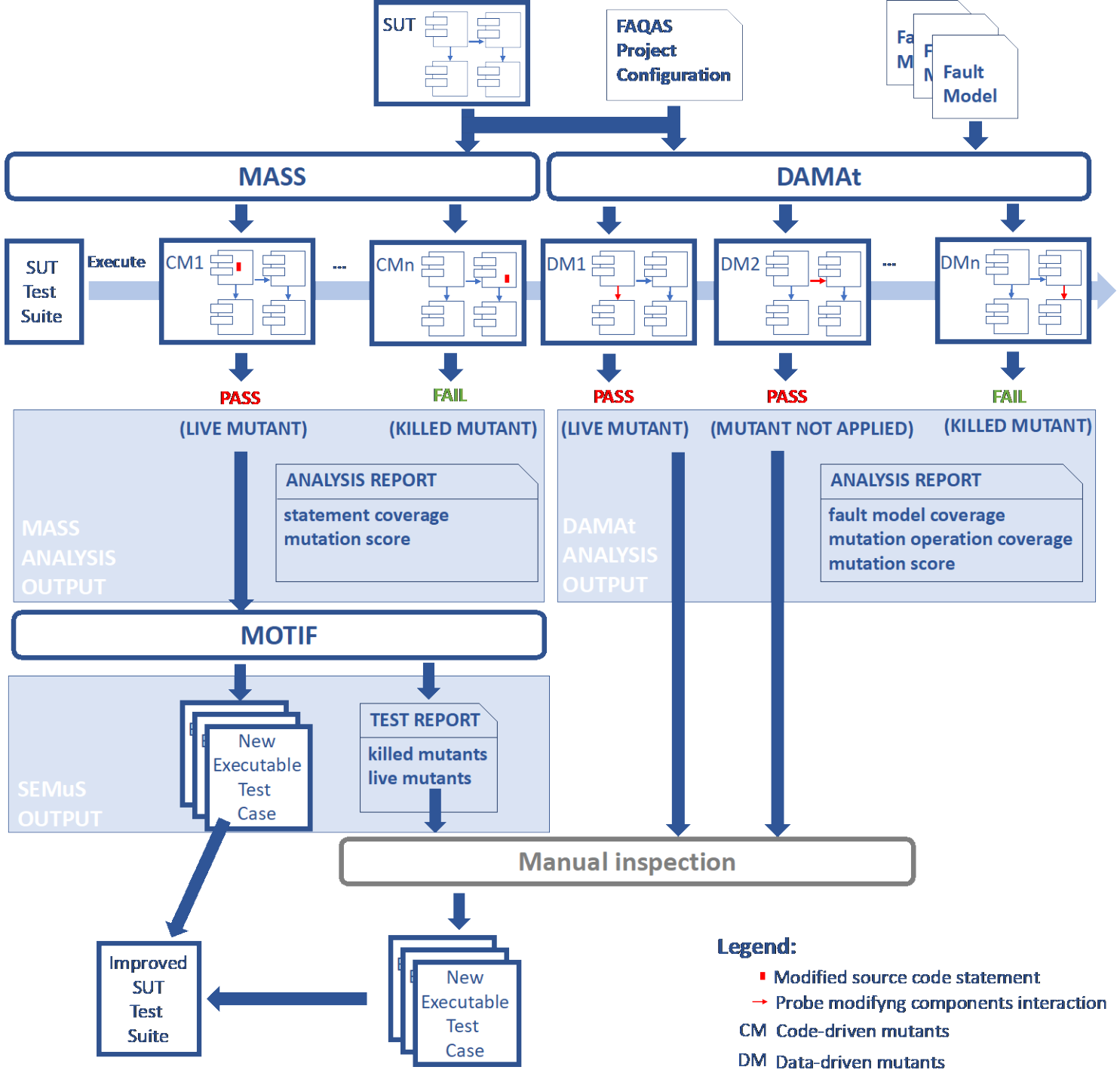
uni.lu | SNT

Applicability of Mutation Testing to Space Software

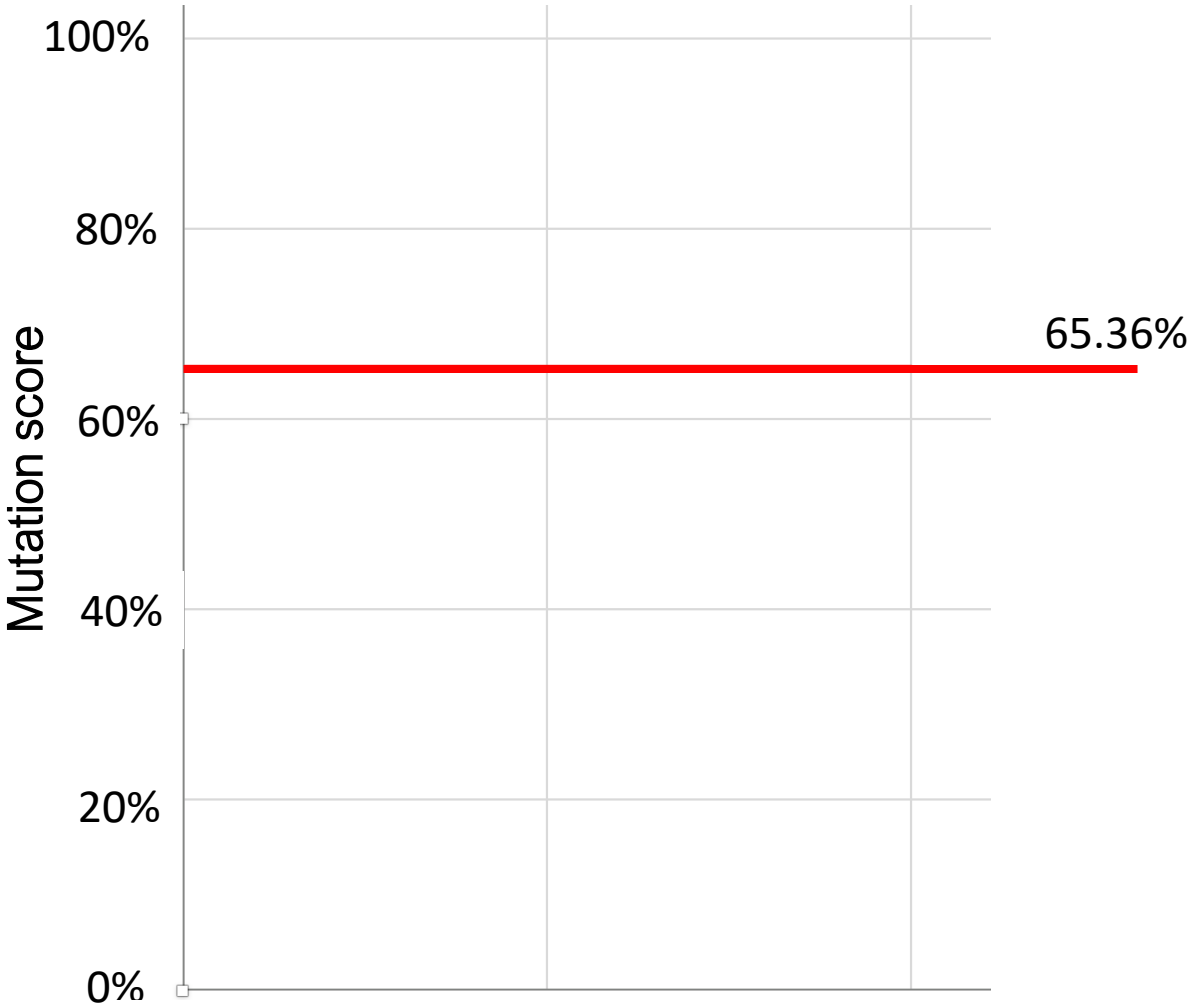
Fabrizio Pastore,
Jaekwon Lee, Enrico Viganò, Oscar Cornejo, Lionel Briand
University of Luxembourg
(fabrizio.pastore@uni.lu)

ADCSS 2023 - November 15th, 2023

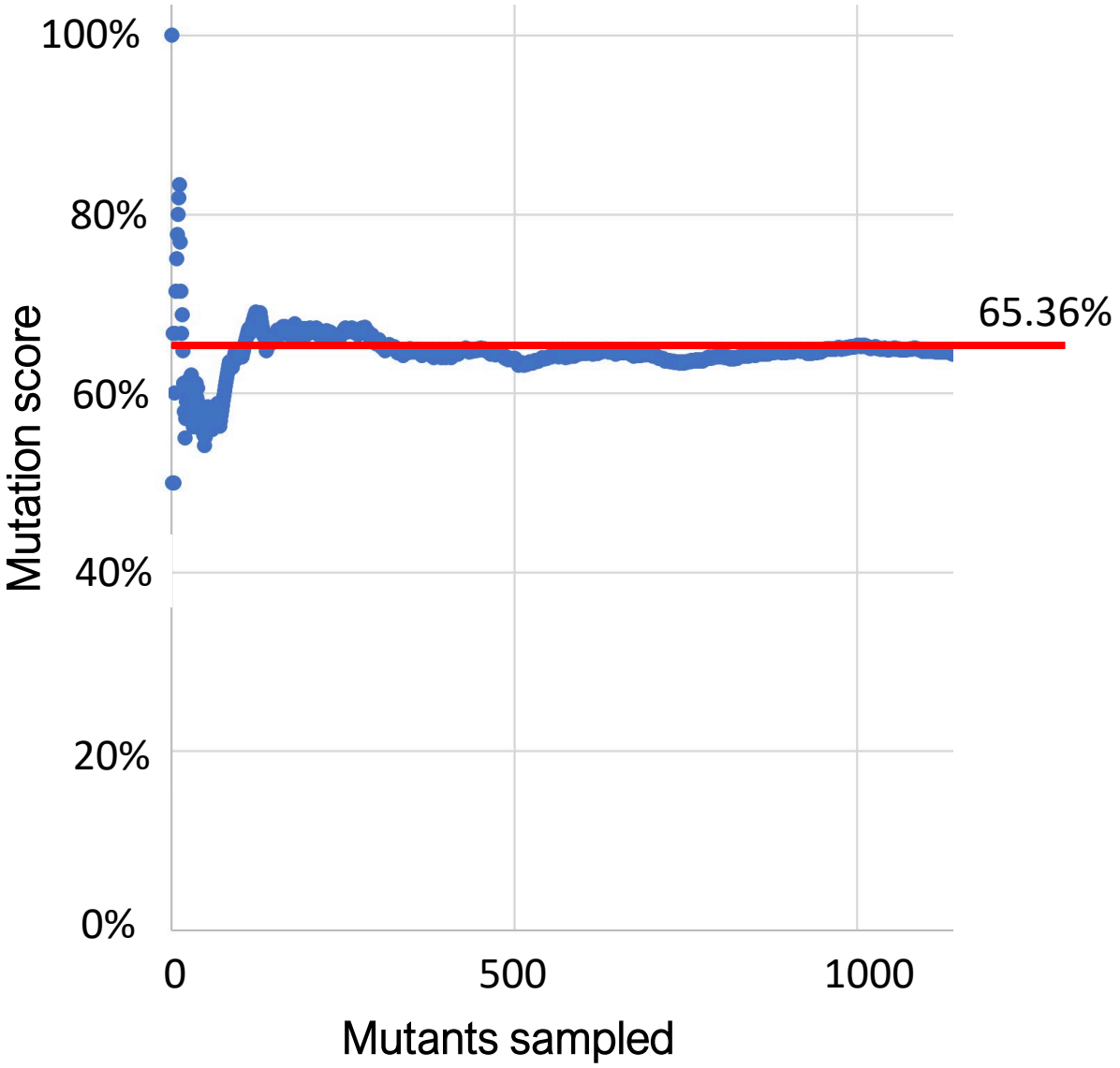
Backup


SUT
FAQAS Project Configuration
Fault Model
MASS
DAMAt
SUT Test Suite
Execute
CM1
CMn
DM1
DM2
DMn
PASS
FAIL
PASS
PASS
FAIL
(LIVE MUTANT)
(KILLED MUTANT)
(LIVE MUTANT)
(MUTANT NOT APPLIED)
(KILLED MUTANT)
ANALYSIS REPORT
statement coverage
mutation score
MASS ANALYSIS OUTPUT
DAMAt ANALYSIS OUTPUT
ANALYSIS REPORT
fault model coverage
mutation operation coverage
mutation score
MOTIF
New Executable Test Case
TEST REPORT
killed mutants
live mutants
SEMuS OUTPUT
Manual inspection
Improved SUT Test Suite
New Executable Test Case
Legend:
Modified source code statement
Probe modifyng components interaction
CM Code-driven mutants
DM Data-driven mutants

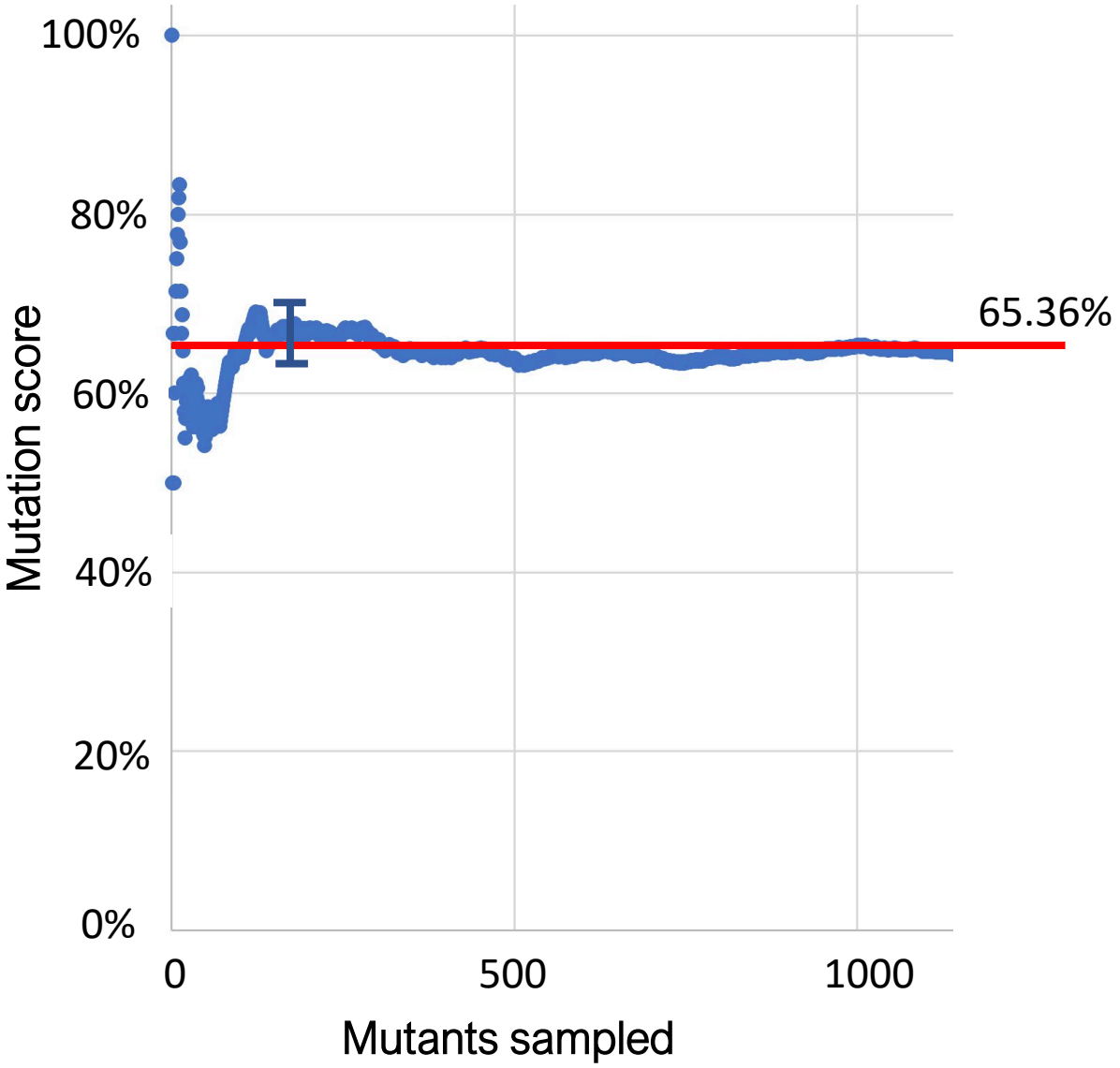
Mutants sampling with FSCI



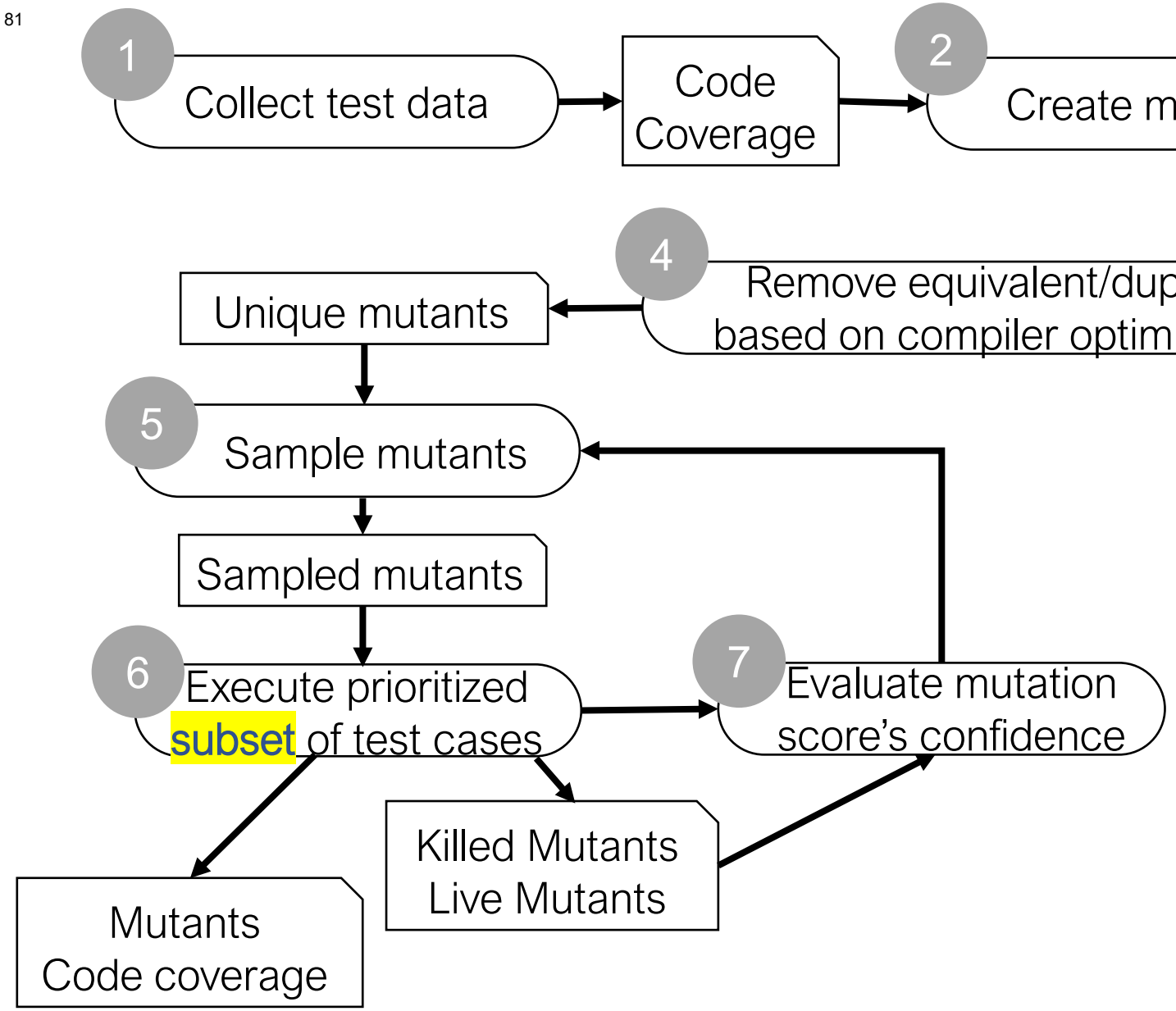
Mutants sampling with FSCI



Mutants sampling with FSCI



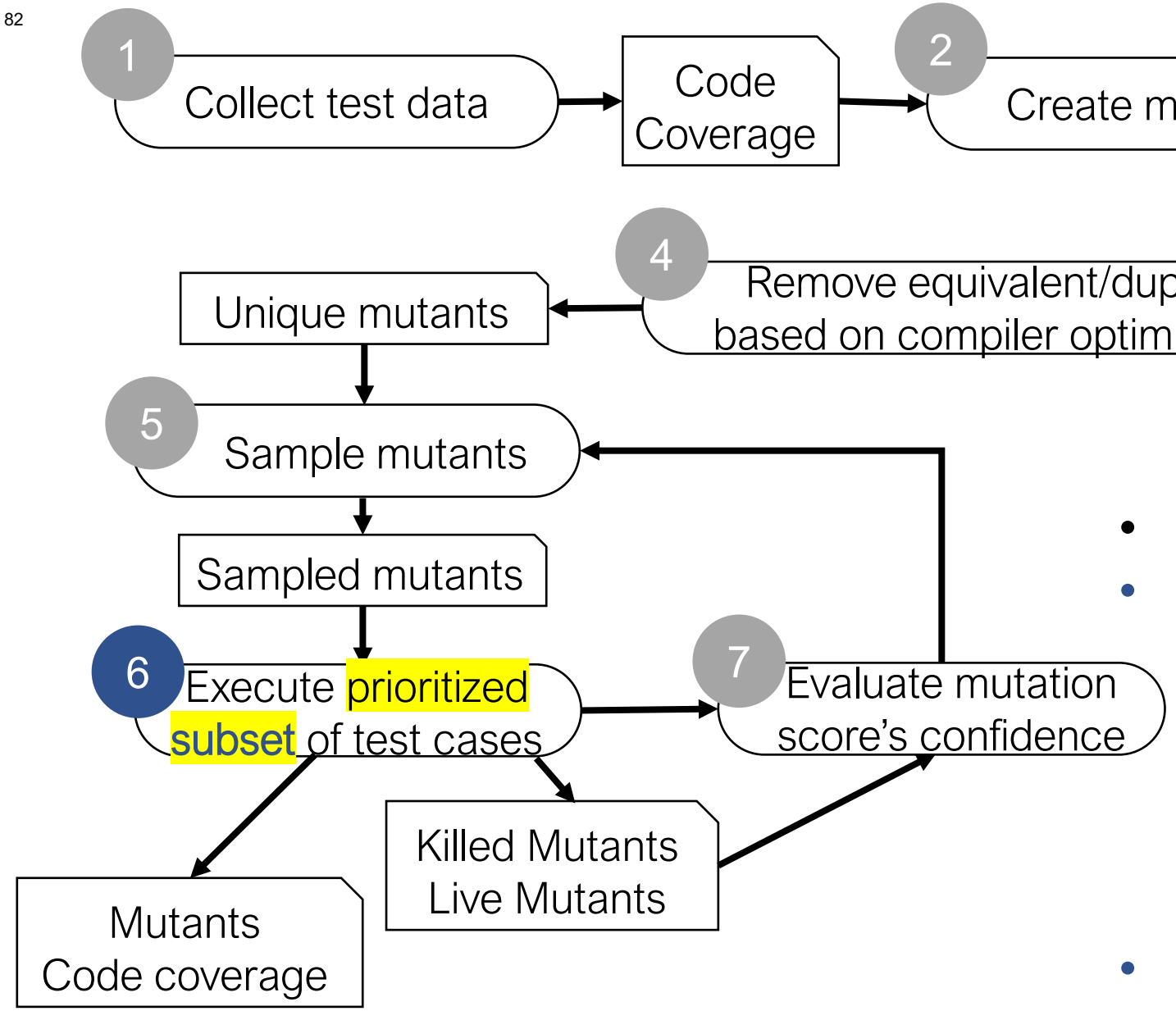
- A **confidence interval** captures a range that has a probability (e.g., 95%) of including the estimated value (the mutation score)
  - $MS \in [L;U]$
- Fixed-width sequential confidence interval (FSCI) method:
  - stop sampling when (U – L) is small
  - difference between estimated MS and actual one is at most 5 percentage p.



Confidence interval correction

- $MS=[L+P_{err};U+P_{err}]$
- Proportion of mutants live by mistake

  $P_{err}=[L_{err};U_{err}]$
- $MS=[L+L_{err};U+U_{err}]$



- Based on statement coverage
- Test suite prioritization:
  - greedy algorithm
    - select first the test case that largely differ from the most similar, already selected, test case
- Test suite reduction: exclude test cases with a distance of zero
- Cosine distance: best accuracy

Fault Model Example

Position	Span	Type	Operator	Delta	Min	Max
0	1	Binary	Bit flip			
1	2	Double	Value Out of Range	0.1	10	14
1	2	Double	Fix Value Out of Range	0.1	10	14
3	2	Double	Value Above Threshold	0.1		6
5	1	Integer	Value Above Threshold	1		10

12 Mutation Operators

Value out of range (VOR)

Fix value out of range (FVOR)

Value above threshold (VAT)

Fix value above threshold (FVAT)

Value below threshold (VBT)

Fix value below threshold (FVBT)

Bit flip (BF)

Random legal value (INV)

Illegal value (IV)

Amplified signal (AS)

Shifted signal (SS)

Flatten signal (HV)

Fault Modelling Methodology