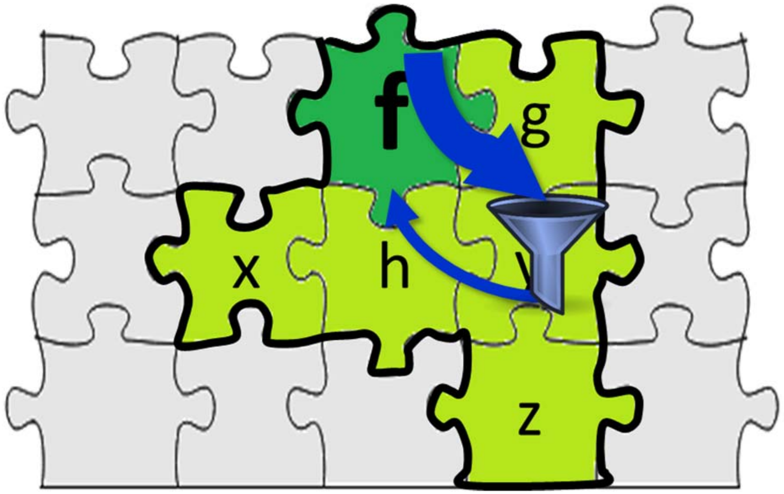
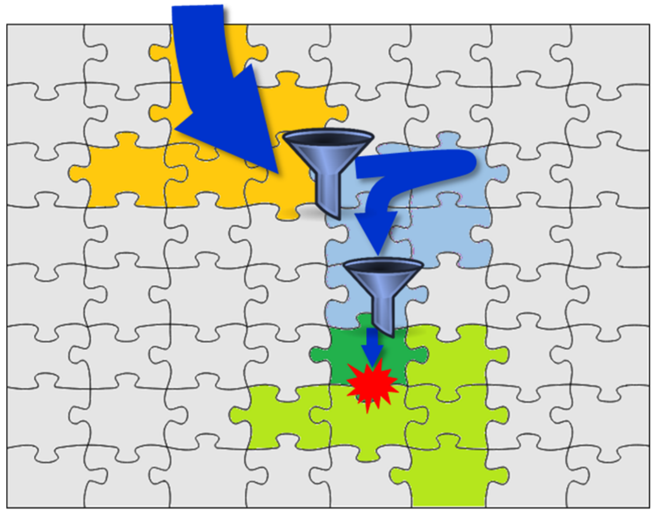
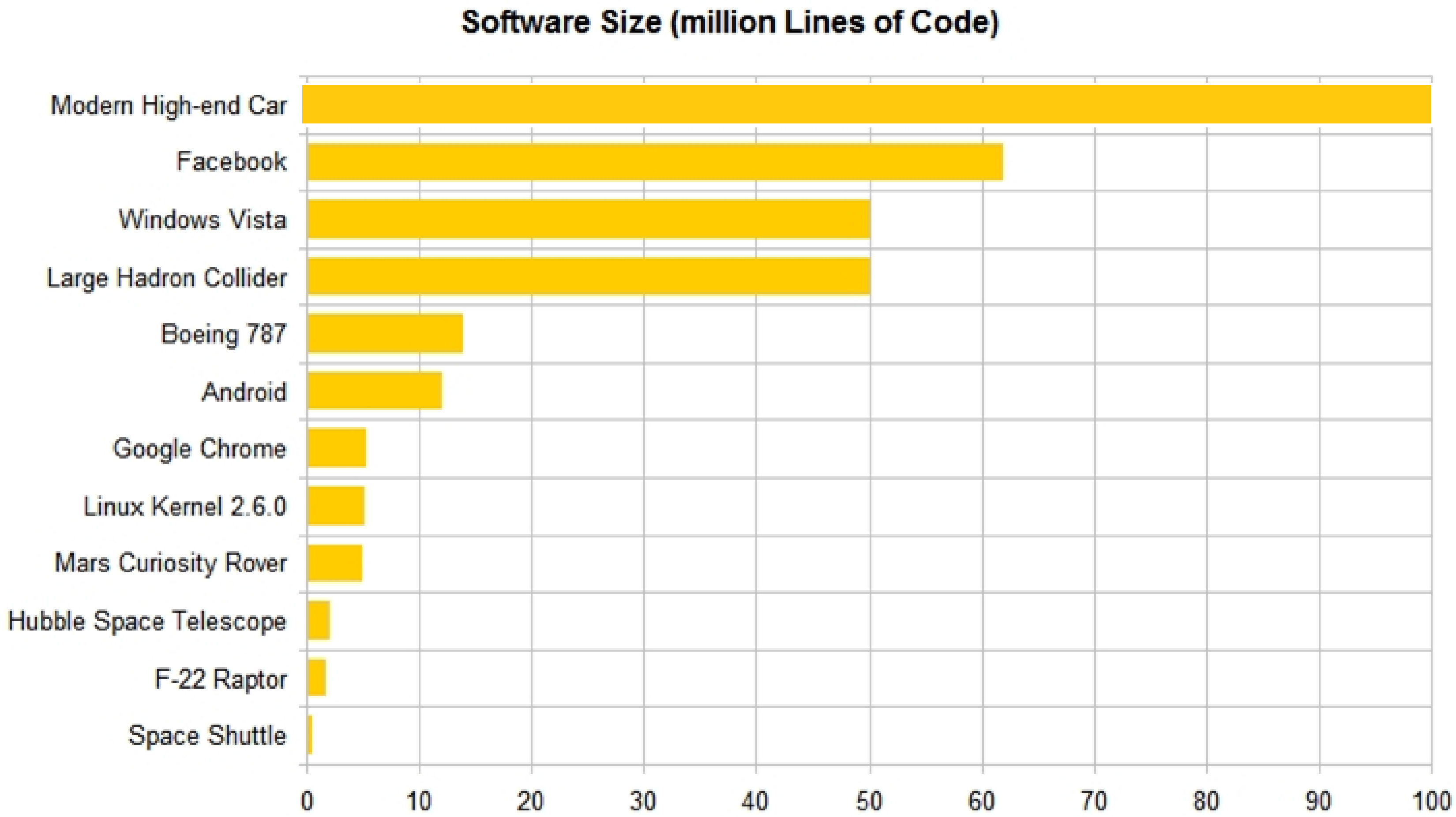
Concolic Unit Testing





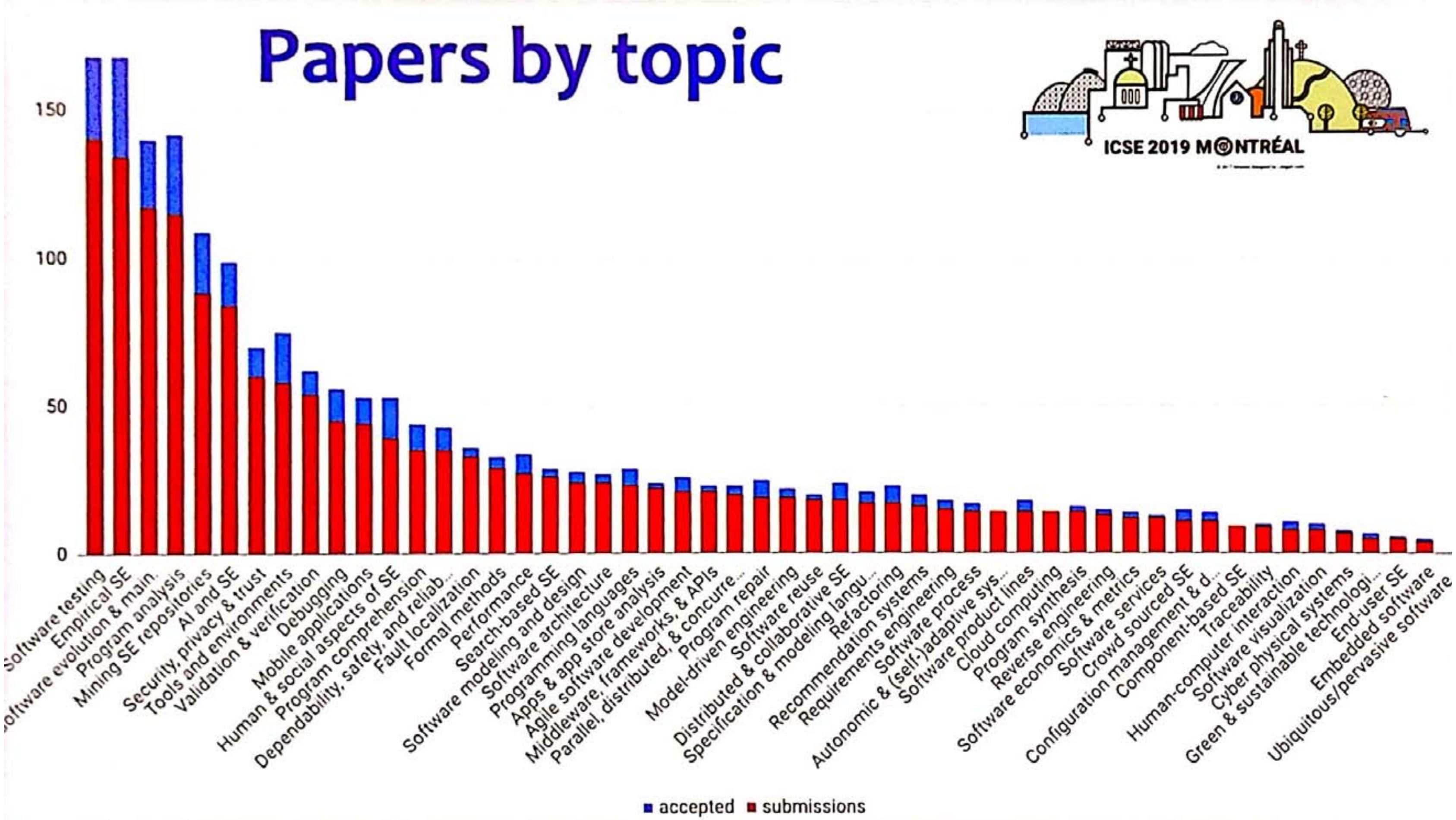
Yunho Kim
SWTV group, KAIST

KAIST School of Computing



A.Busnelli, Counting, *https://www.linkedin.com/pulse/20140626152045-3625632-car-software-100m-lines-of-code-and-counting*
http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

ICSE 2019 Topics (Top SE conf. w/ accept. rate: 20%)

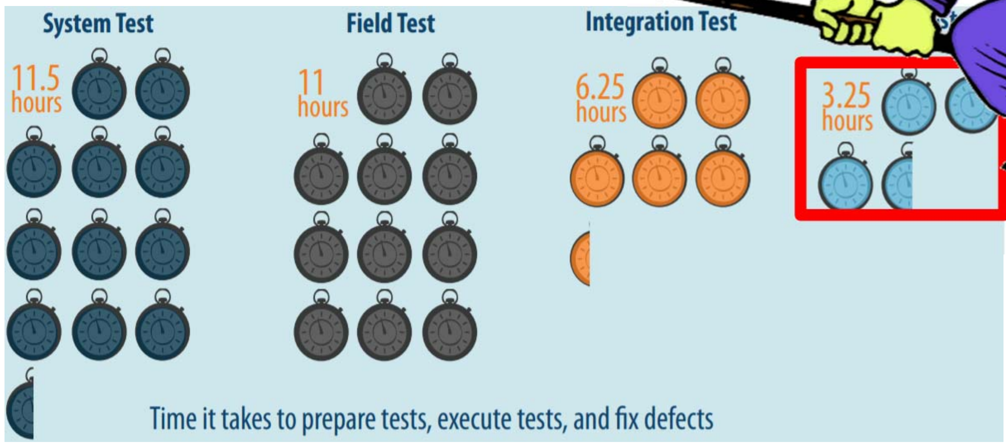


Benefits of Unit Testing

- **Bug correction cost: 7x cheaper** than system tests
  - $937 (unit test) vs $7,136 (system test)

Source: B. Boehm and V. Basil, Software Defect Reduction Top 10 List, IEEE Computer, January 2001

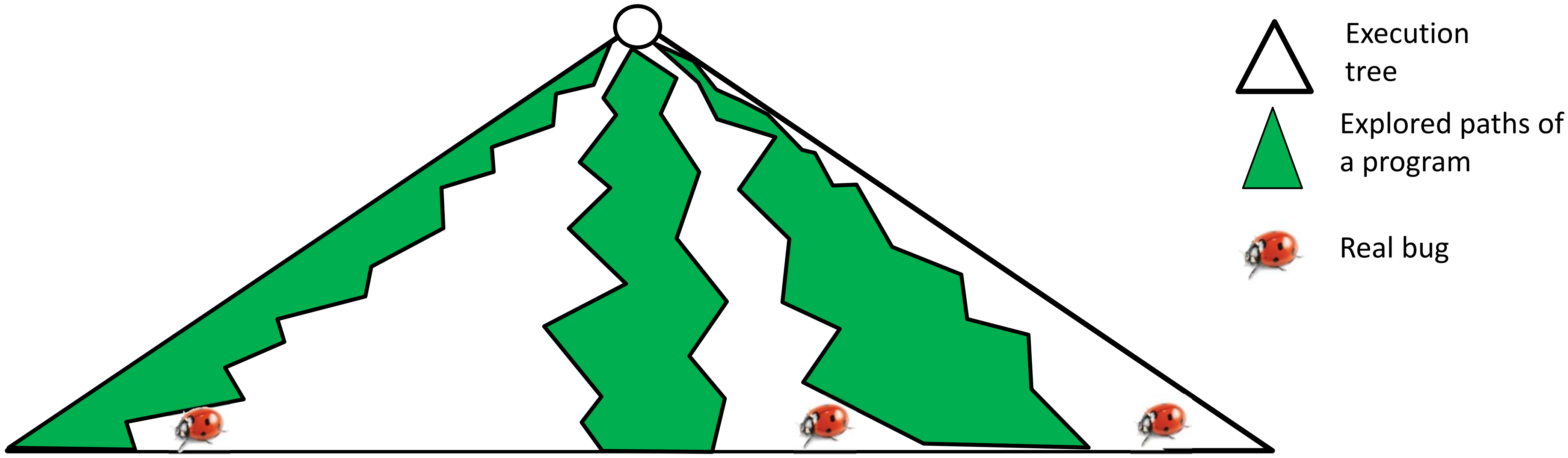
- **Bug correction time: 3x fa** than system testing
  - 3.25 hours vs 11.5 hours



Source: Capers Jones, Applied Software Measurement: Global Analysis of Productivity and Quality

Pros and Cons of Auto. Test Gen. at System-level (1/2)

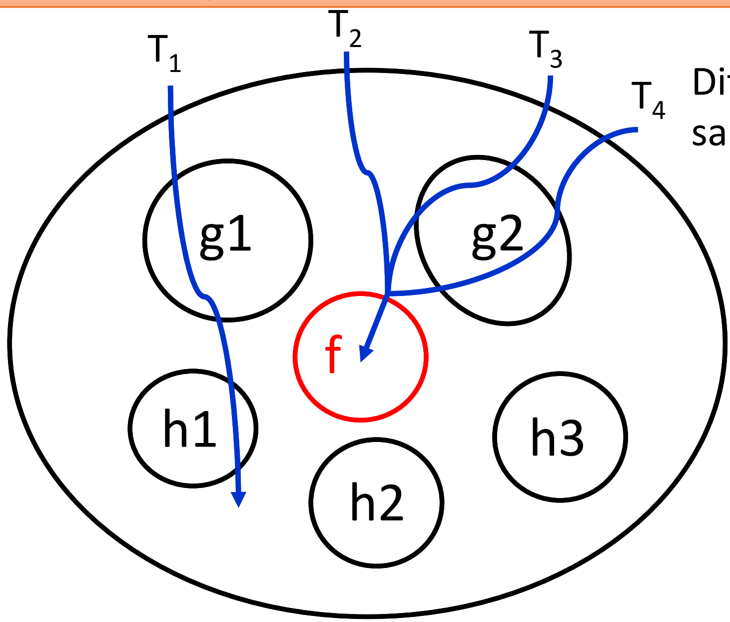
- Pros: **No false alarms**
- Cons: Low bug detection power due to **large search space**



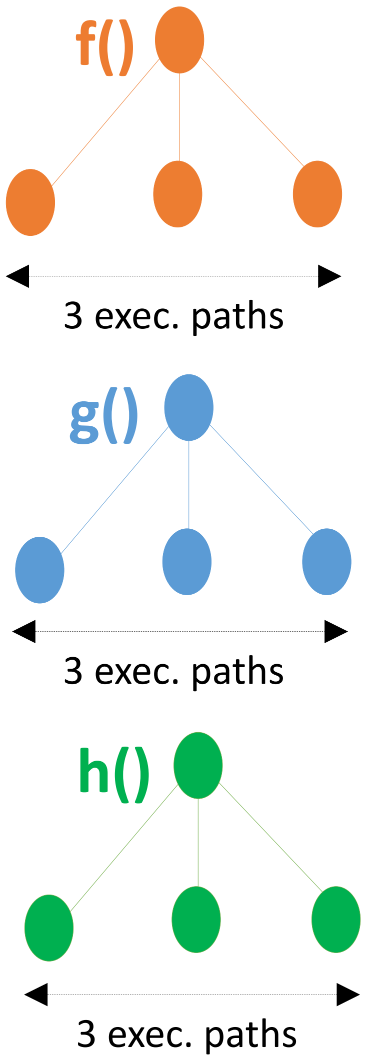
Pros and Cons of Auto. Test Gen. at **System-level** (2/2)

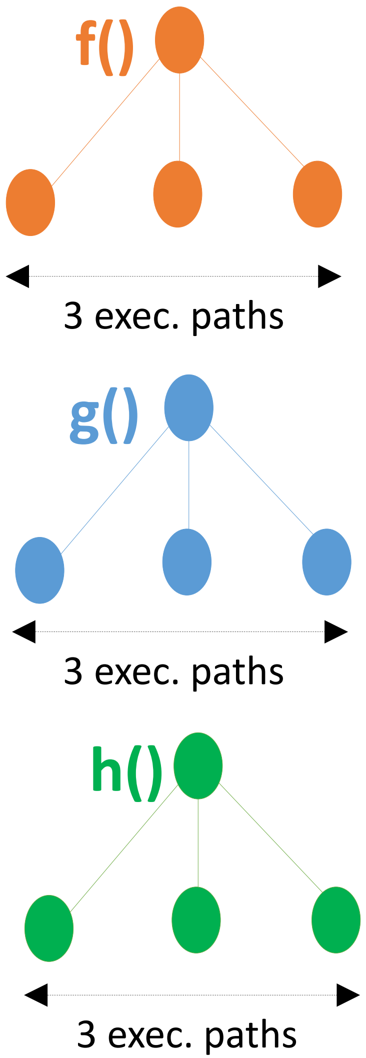
Pros
+ Can be easy to generate system TCs due to clear interface specification
+ No false alarm (i.e., no assert violation caused by infeasible execution scenario)

Cons
- Low controllability of each unit
- Large and complex search space to explore in a limited time
- Hard to detect bugs in corner cases

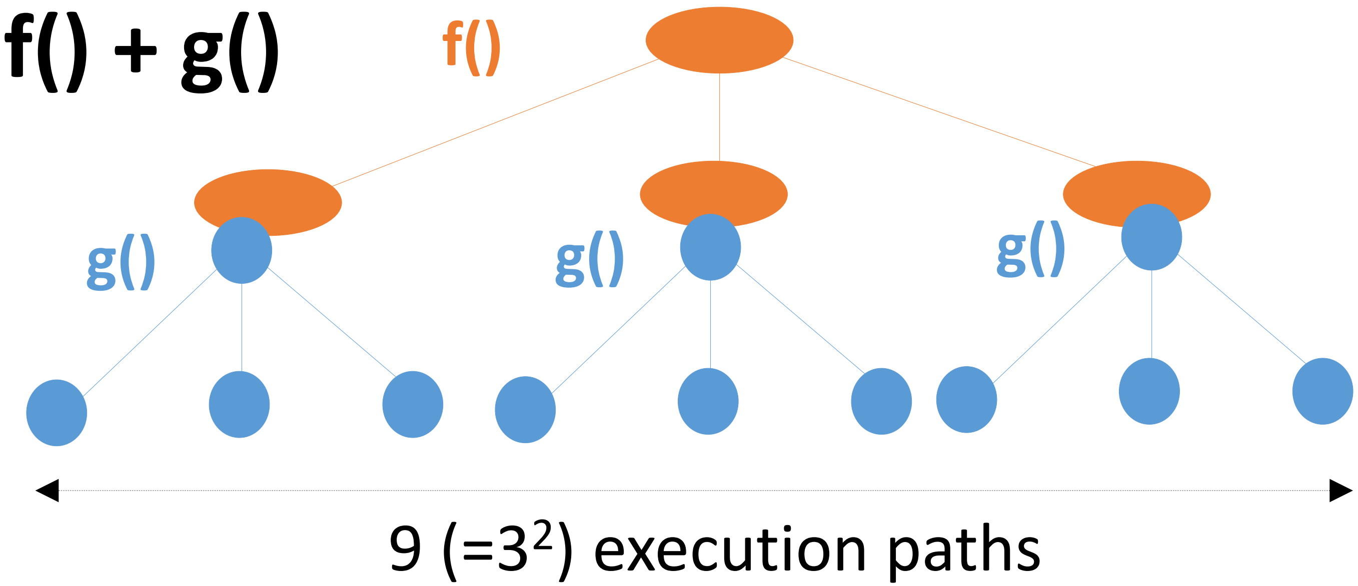


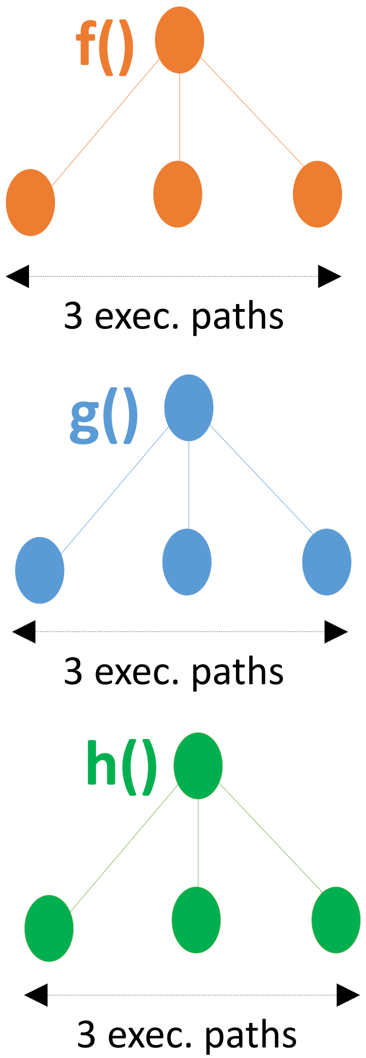
Different system tests T_2 to T_4 exercise the same behavior of the target unit


f()
3 exec. paths
g()
3 exec. paths
h()
3 exec. paths

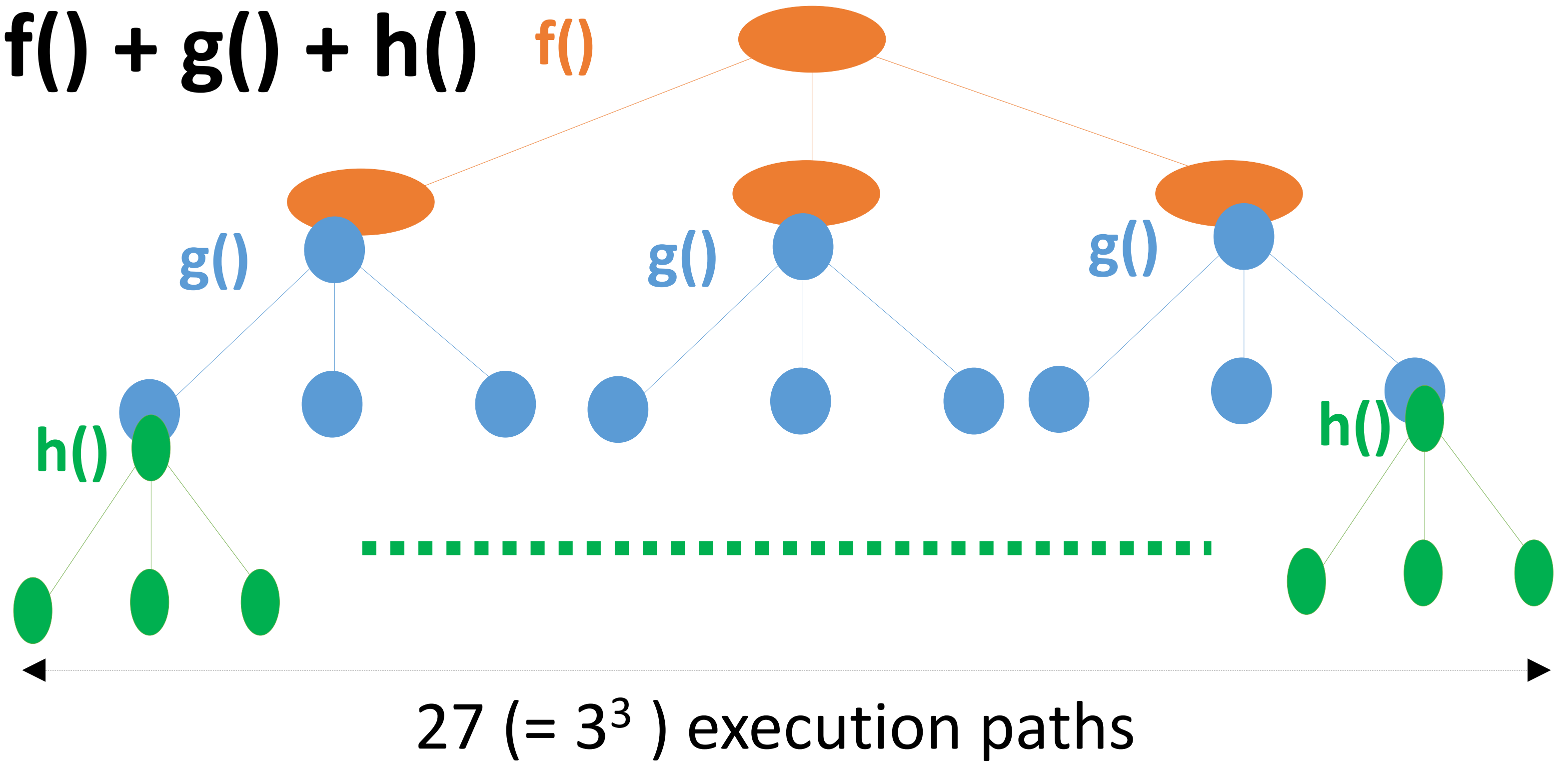

f()
3 exec. paths
g()
3 exec. paths
h()
3 exec. paths

f() + g()


f()
g()
g()
g()
9 (=3²) execution paths


f()
3 exec. paths
g()
3 exec. paths
h()
3 exec. paths

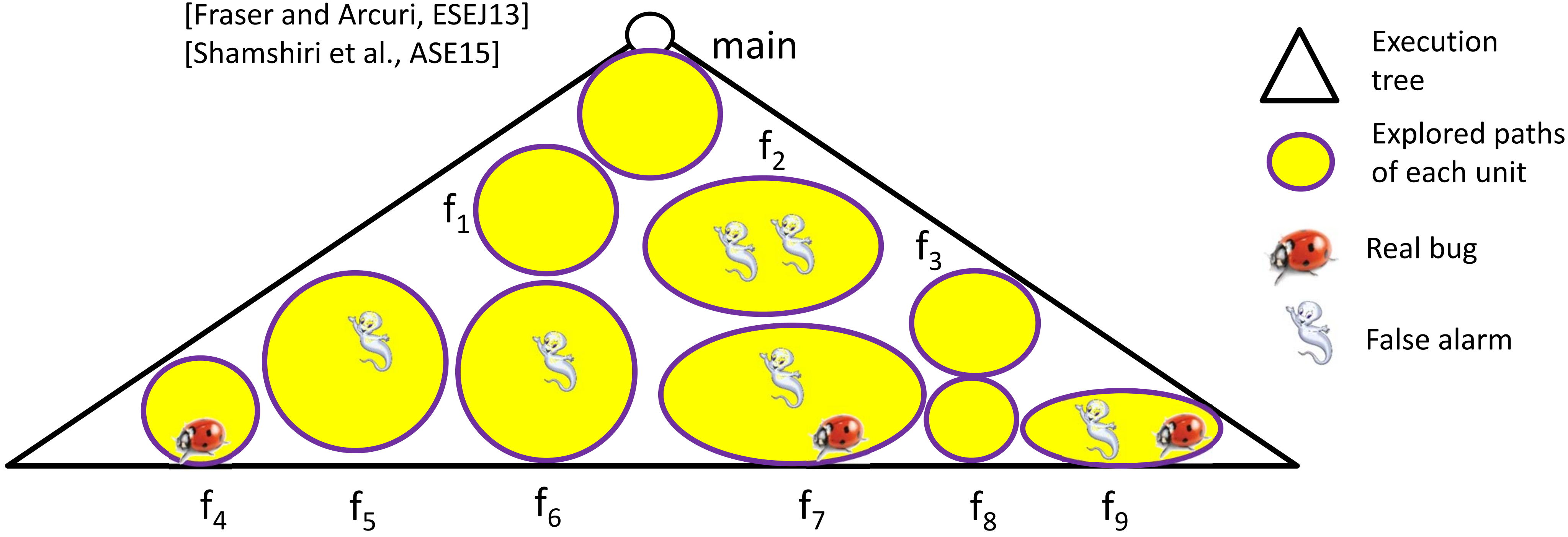
f() + g() + h()


f()
g()
g()
g()
h()
h()
27 (= 3³) execution paths

Pros and Cons of Auto. Test Gen. at **Unit-level** (1/2)

- Pros: High bug detection power for **small search space**
- Cons: **Many false alarms** due to over-approximated context of a unit

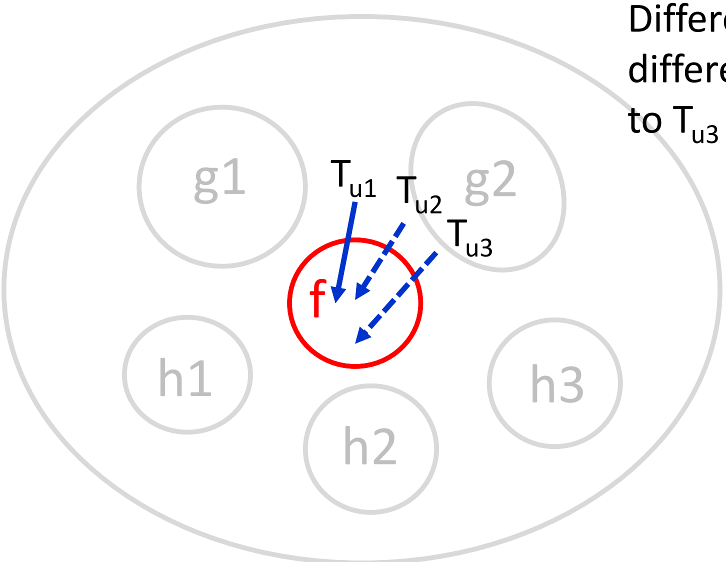
[Gross et al., ISSTA12]
[Fraser and Arcuri, ESEJ13]
[Shamshiri et al., ASE15]



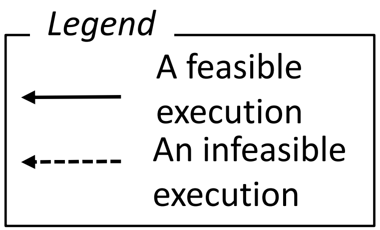
Pros and Cons of Auto. Test Gen. at **Unit-level** (2/2)

Pros
+ High controllability of a target unit
+ Smaller search space to explore than system testing
+ High effectiveness for detecting corner cases bugs

Cons
- Hard to write down accurate unit test drivers/stubs due to unclear unit specification
- High false/true alarm ratio



Different unit tests T_{u1} to T_{u3} directly exercise different behaviors of the target unit, but T_{u2} to T_{u3} exercise infeasible paths



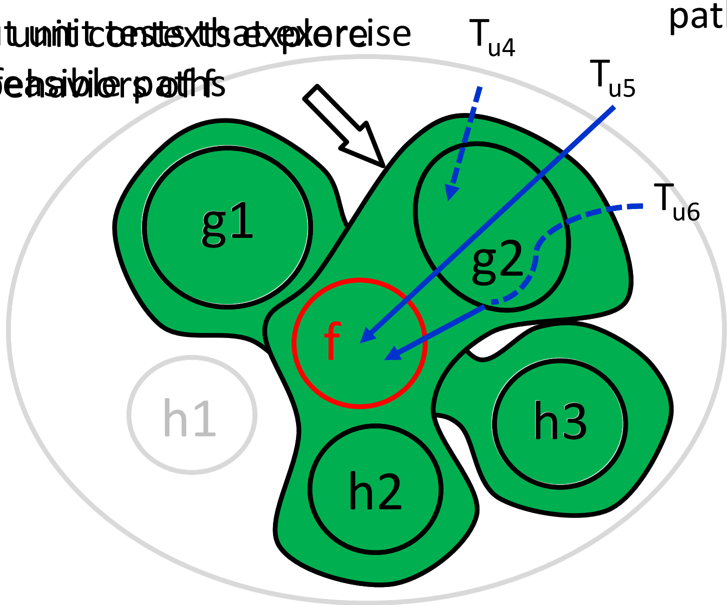
Automated Unit Test Generation with Realistic Unit Context

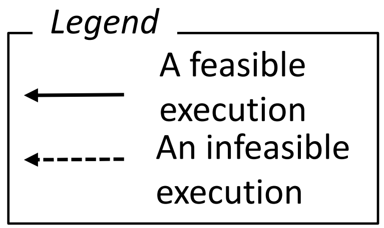
Pros
+ High controllability of a target unit
+ High effectiveness for detecting corner cases bugs
+ Low false alarm ratio

Realistic unit context filters out unit tests that exercise infeasible paths

Different unit contexts explore various behaviors of f

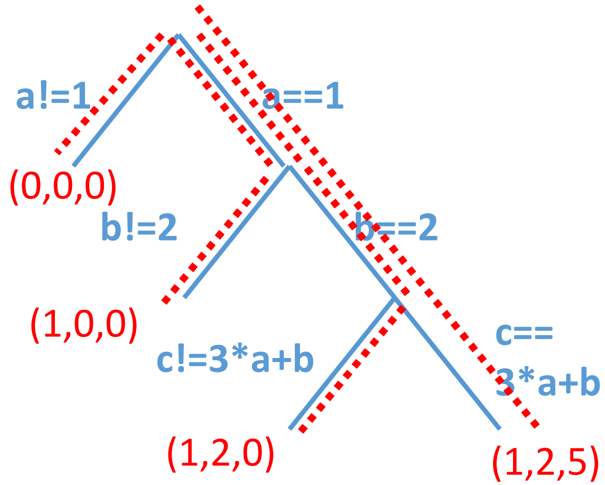
Unit tests T_{u4} that exercises an infeasible path is filtered out by the unit context





Concolic Testing Example

```
// Test input a, b, c
void f(int a, int b, int c) {
  if (a == 1) {
   if (b == 2) {
     if (c == 3*a + b) {
       error();
} } }}
```



- Random testing
  - Probability of reaching error( ) is extremely low
- Concolic testing generates the following 4 test cases
  - (0,0,0): initial random input
    - Obtained symbolic path formula (SPF) ϕ: a!=1
    - Next SPF ψ generated from ϕ: !(a!=1)
  - (1,0,0): a solution of ψ (i.e. !(a!=1))
    - SPF ϕ: a==1 && b!=2
    - Next SPF ψ: a==1 && !(b!=2)
  - (1,2,0)
    - SPF ϕ: a==1 && (b==2) && (c!=3*a +b)
    - Next SPF ψ: a==1 && (b==2) && !(c!=3*a +b)
  - (1,2,5)
    - Covered all paths and

error() reached

Constructing Test Driver/Stubs (1/2)

- CONBRIO **automatically generates a unit test driver/stub** functions for unit testing of a target function
  - A unit test driver symbolically sets all visible global variables and parameters of the target function
  - The test-generator module replace sub-functions invoked by the target function with symbolic stub functions

Type	Description	Code Example
Primitive	Set a corresponding symbolic value	`int a;` **`SYM_int(a);`**
Array	Set a fixed number of elements	`int a[3];` **`SYM_int(a[0]);`** … **`SYM_int(a[2]);`**
Structure	Set symbolic value to all fields recursively	`struct _st{int n,struct _st*p}a;` **`SYM_int(a.n);`** **`a.p=&a;`**
Pointer	Allocate memory whose size is equal to the size of a pointee and set symbolic value according to pointee type.	`int *a;` **`a = malloc(sizeof(int));`** **`SYM_int(*a);`**

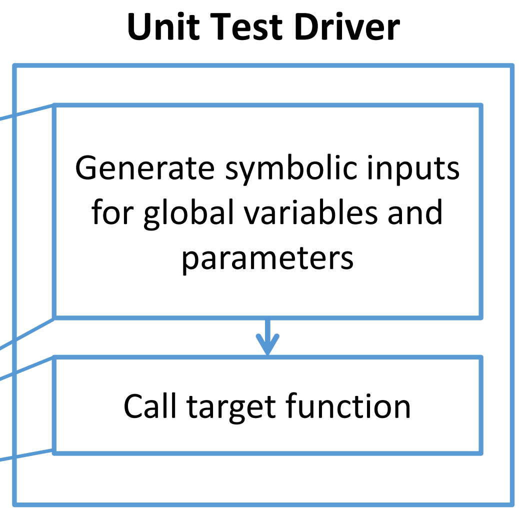
Constructing Test Driver/Stubs (2/2)

- Example of an automatically generated unit-test driver

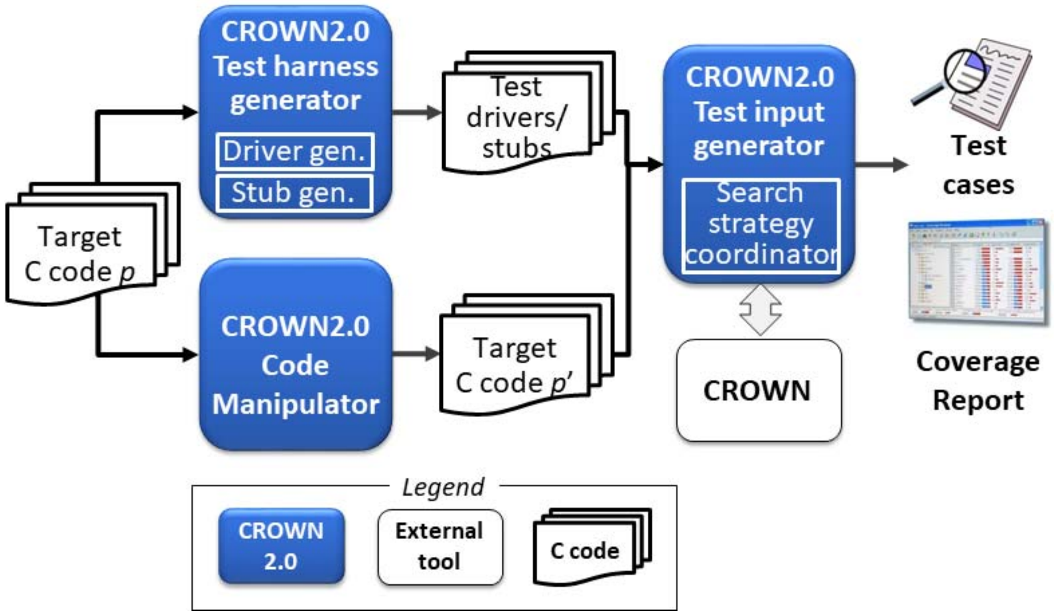
```
typedef struct Node_{
  char c;
  struct Node_ *next;
} Node;
Node *head;
// Target unit-under-test
void add_last(char v){
  // add a new node containing v
  // to the end of the linked list
  ...}
// Test driver for the target unit
void test_add_last(){
  char v1;
  head = malloc(sizeof(Node));
  SYM_char(head→c);
  head→next = NULL;
  SYM_char(v1);
  add_last(v1); }
```

Set global variables

Set parameter



CROWN 2.0: 저비용 고효율 SW 자동 테스팅 도구



높은 테스트 커버리지를 자동으로 달성하여, SW 오류를 빠르게 발견하는 SW 자동 테스트 도구

SW 개발/테스팅 비용절감 및 SW 품질 향상

- Concolic 테스팅으로 **SW의 모든 가능한 동작 경로**를 실행하는 테스트 자동 생성
- 유닛 테스팅 100% 자동화를 위해, 유닛 테스트 driver/stub 까지 자동 생성
- 안전필수 시스템용 SW 작성 언어 중, 널리 사용되는 C 프로그램 테스팅

Limitations of Automated Unit Testing

- High false/true alarm ratio

"This function f() assumes that values of the first parameter is between 0 and 100 and callers of f() should never pass a value less than 0 or greater than 100 to f() as the first parameter."

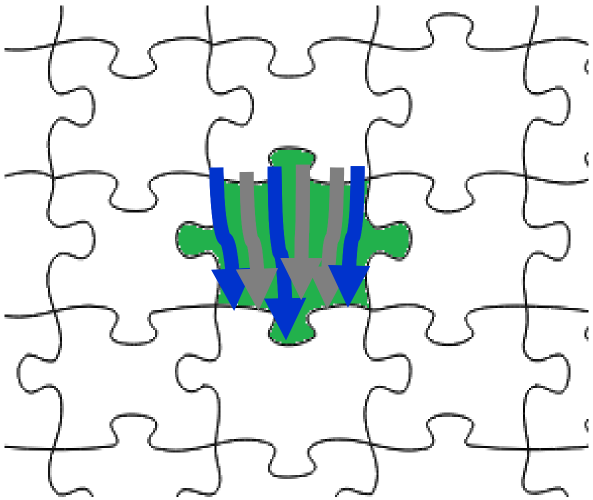
- We need to refine unit test generation technique to reduce false alarms by utilizing unit context

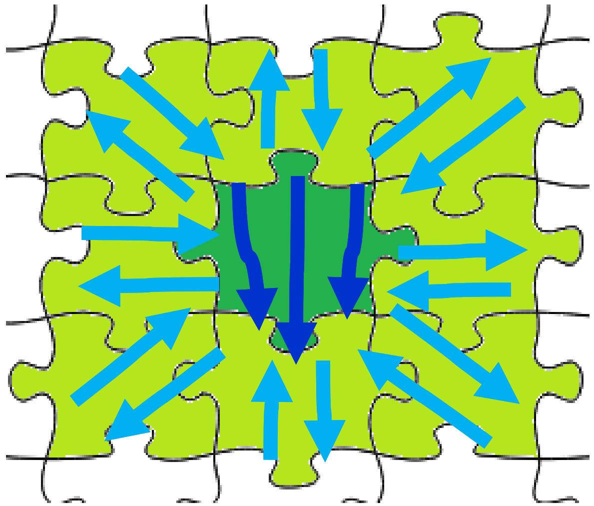
False Alarms Caused by Missing Context

```
// Target function under test
int f(int x, int y){
  int array[5] = {1,3,5,7,9};
  int n, res;
  // Alarm: Array overflow
  n = array[x];
  g(&n);
  if ((y % 2) != 0){
    // Alarm: Array overflow
    res = array[n];
  }
  return res;}
```

```
// x, y, and z are inputs
int main(int x, int y, int z){
  if (x>0) return b(y,z);
}

int b(int x, int y){
  if (0<=x && x<5)
    return f(x,y);
  else return 0;}




void g(int *p){
  *p = *p / 2;}
```

Unit Testing *f* without or with Contexts of *f*





Without Contexts of *f*

Pros: fast exploration of target unit execution paths

Cons: infeasible target unit executions

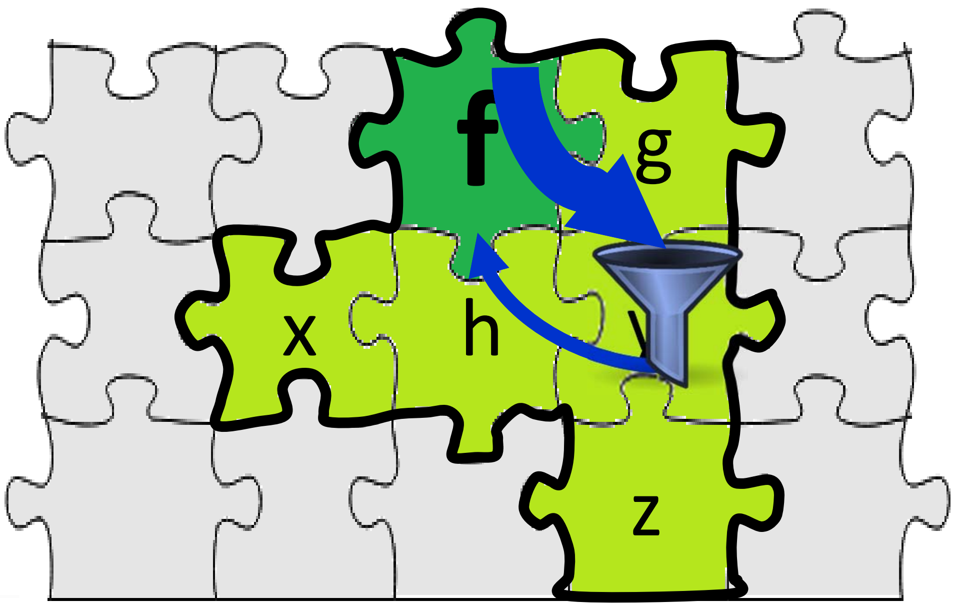
With Contexts of *f*

Pros: reduced infeasible target unit executions

Cons: slow exploration of target unit execution due to large cost of exploring context functions

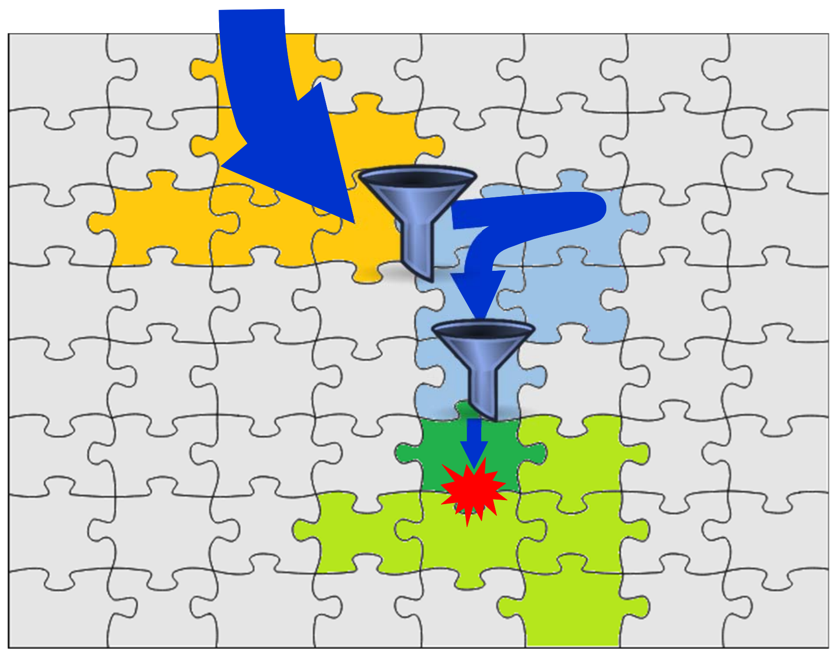
CONBRIO: 2 Main Ideas

- Extended Units



Green: Target function ***f***
Light green: Functions highly relevant to the target
Grey: Functions not relevant to the target
Green + **light green**: **Extended Unit of *f***

- Symbolic Alarm Filtering



Yellow & **sky blue**: a calling context of a target func. ***f***

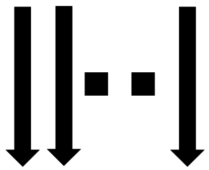
Overview of CONBRIO

CONcolic unit testing with sym**B**olic ala**R**m f**I**ltering using symbolic calling c**O**ntexts

Phase1:
Defining **extended units** and **calling contexts**

A program P

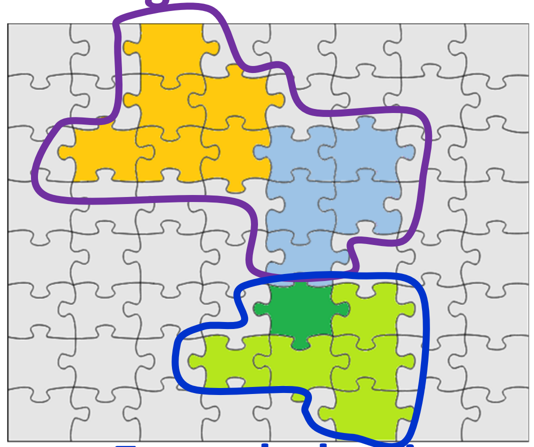
+



Sys. TCs

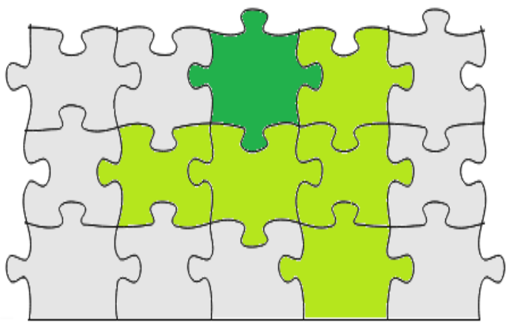
Sys. TC profile analysis

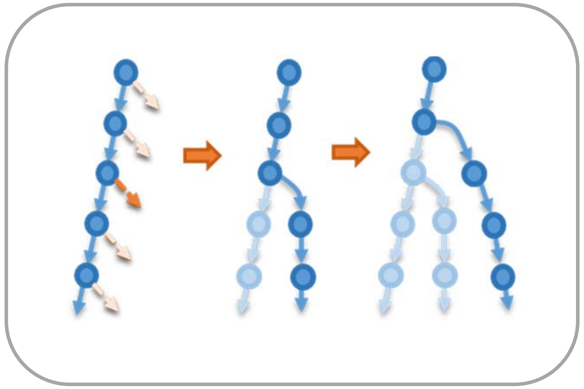
Closely relevant calling context



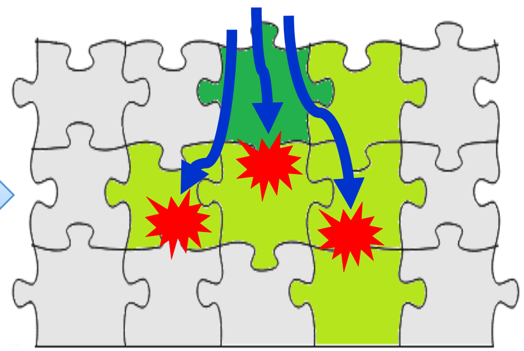
Extended unit

Phase2: Concolic unit testing with an **extended unit**



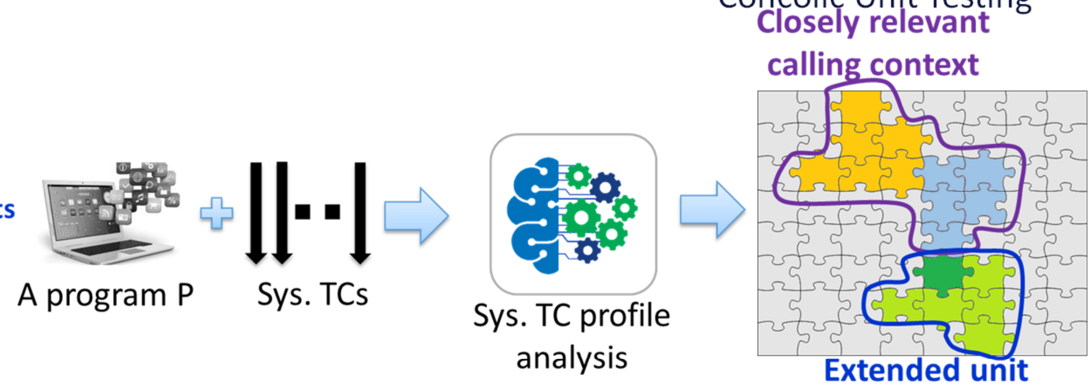


Concolic testing

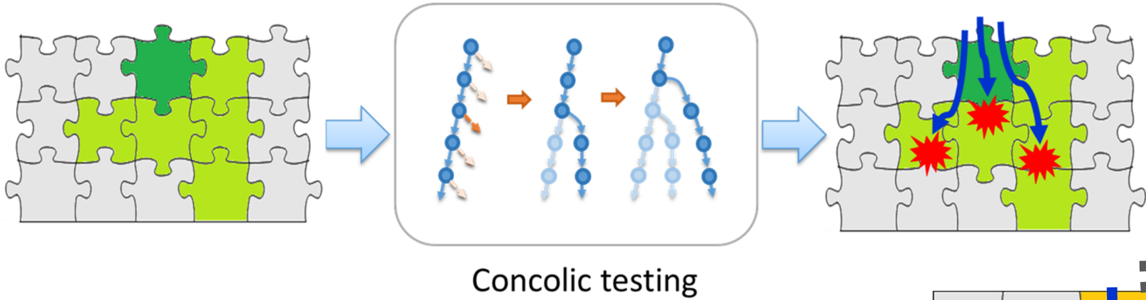


Overview of CONBRIO

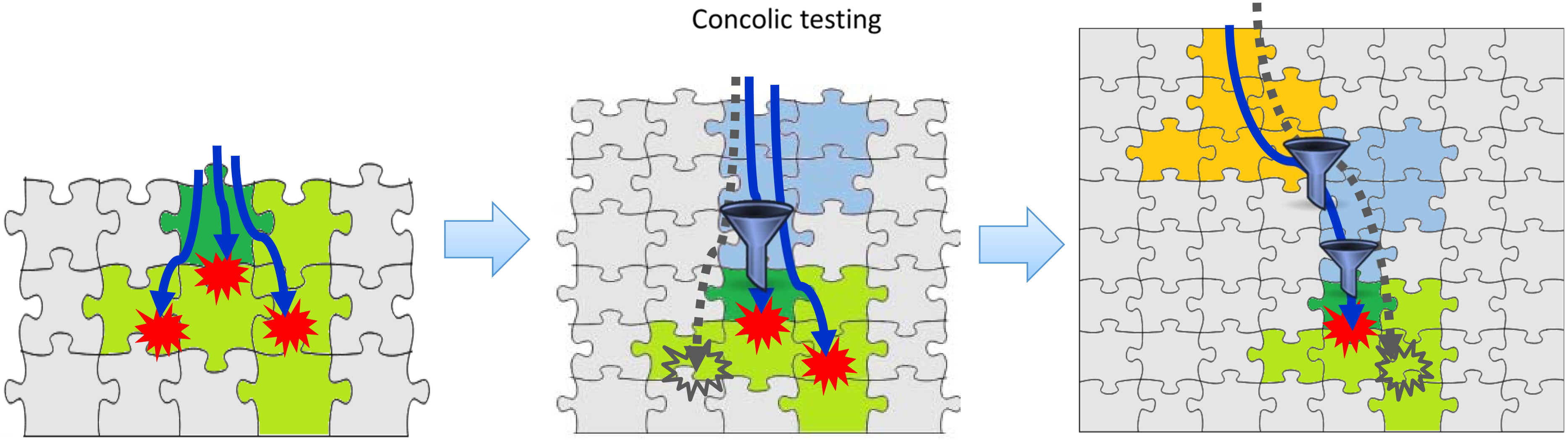
Phase1:
Defining **extended units** and **calling contexts**



Phase2: Concolic unit testing with an **extended unit**

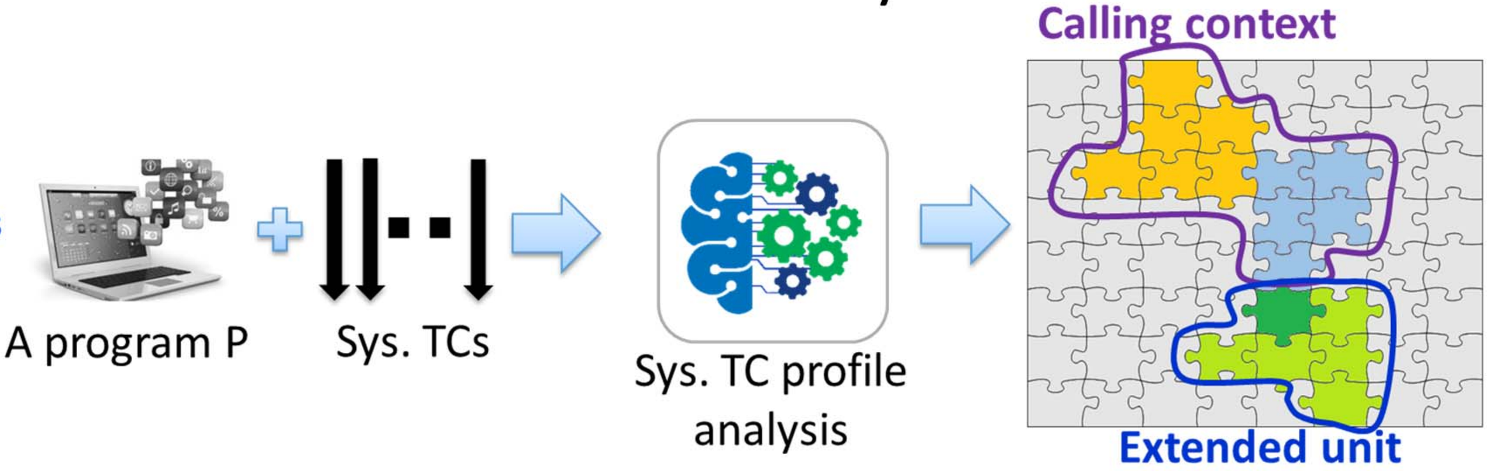


Phase3: Symbolic false alarm filtering

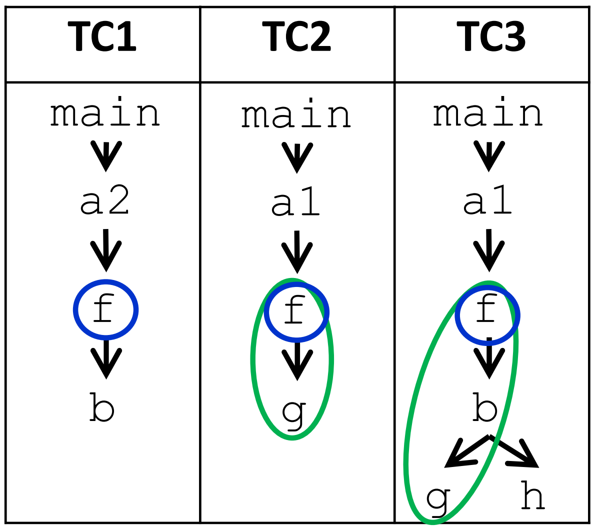


Computing Function Relevance based-on System TCs

Phase1:
Defining **extended units** and **calling contexts**



Function call profile



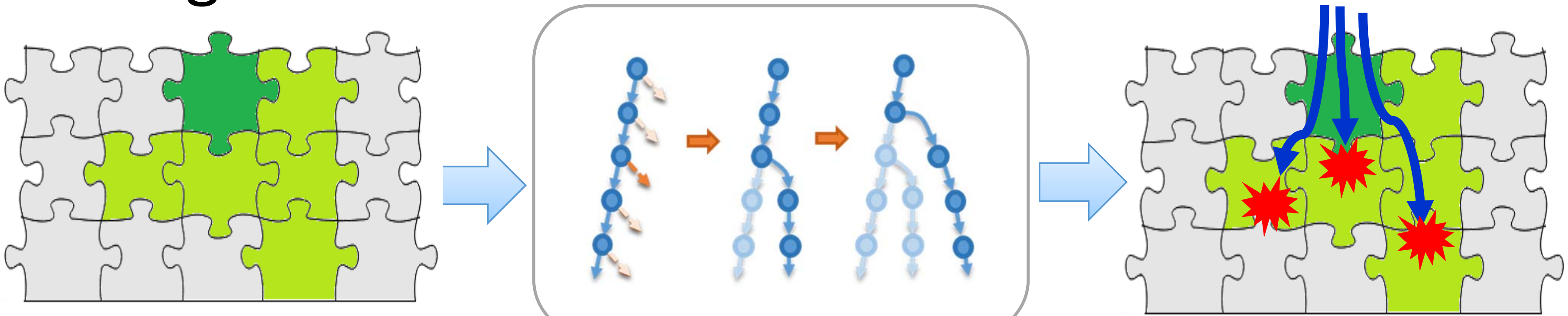
$$P(g|f) = \frac{|f \text{ calls } g|}{|f|} = \frac{|TC2, TC3|}{|TC1, TC2, TC3|} = 0.66$$

Relevance of `f` on other functions (Threshold τ =0.6)

S(e| i) = 0.66
S(j | i) = 0.66
~~S(k | i) = 0.33~~
...

Concolic Testing on Each Extended Unit

Phase2: Concolic unit testing with an **extended unit**



Detecting crash bugs

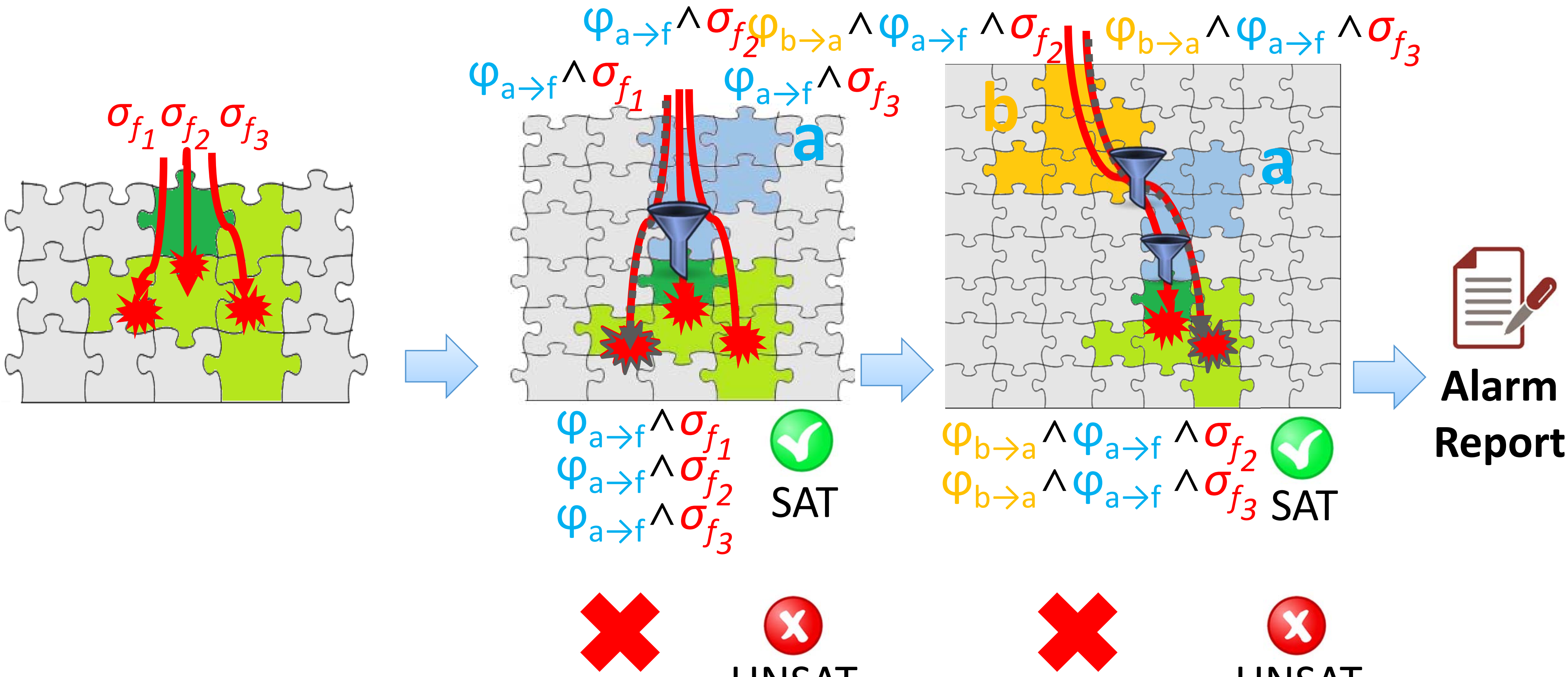
```
concolic_test_driver(){
  set sym. params;
  set sym. globals;
  target_unit(sym. params);
}
```

Buffer overflow

NULL Ptr. Deref.

Div-by-Zero

Symbolic Alarm Filtering



Main Research Questions

RQ1: **How many known crash bugs does CONBRIO detect?**

RQ2: **How much false/true alarm ratio does CONBRIO decrease?**

Target Programs

Total **15,915 functions** in **15 Programs**

SIR

7 Programs

BASH THE BOURNE-AGAIN SHELL
GNU fLEX
GNU Grep
GZIP
GNU Make
GNU SED
Vim

spec

8 Programs

Perl
bzip2
GCC
HMMER
|libquantum⟩
H.264 MPEG-4/AVC

Trade-off between Bug Detection Ability and Accuracy

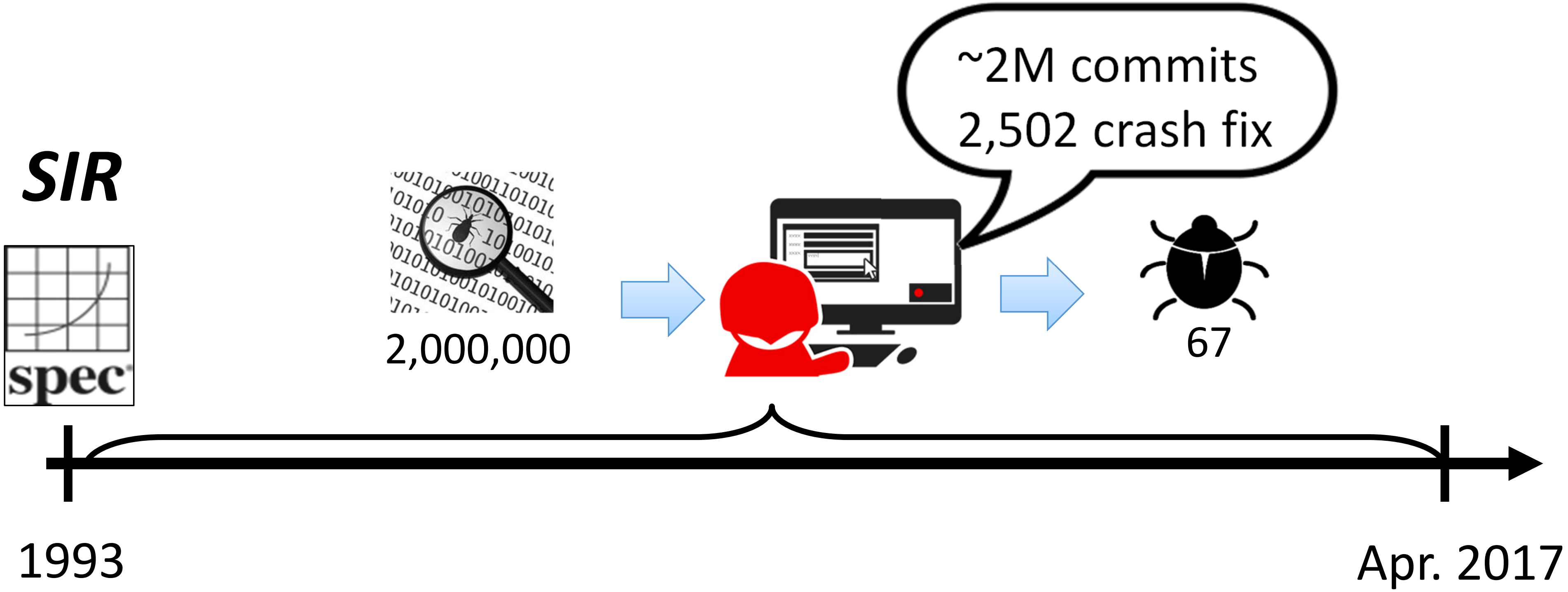
Bug Detection **Ability**
(aiming low false
negative)

Recall

Bug Detection **Accuracy**
(aiming low false
positive)

Precision

CONBRIO Unit Crash Bug Benchmark

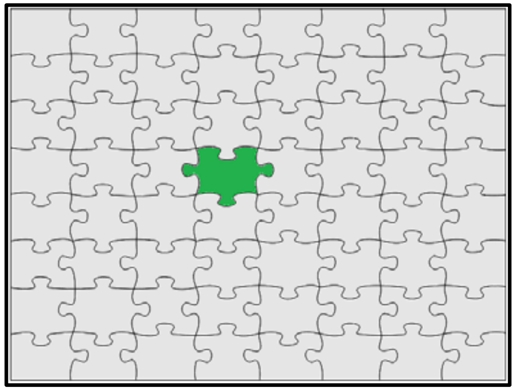


http://github.com/swtv-kaist/conbrio

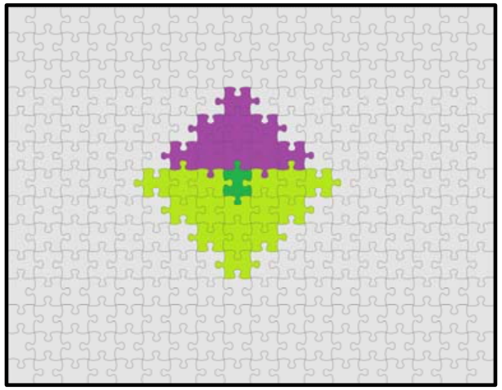
Other Techniques to Compare

Static bound

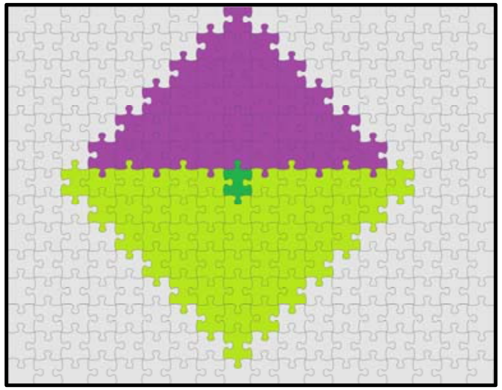
Symbolic Unit Testing (k=0)



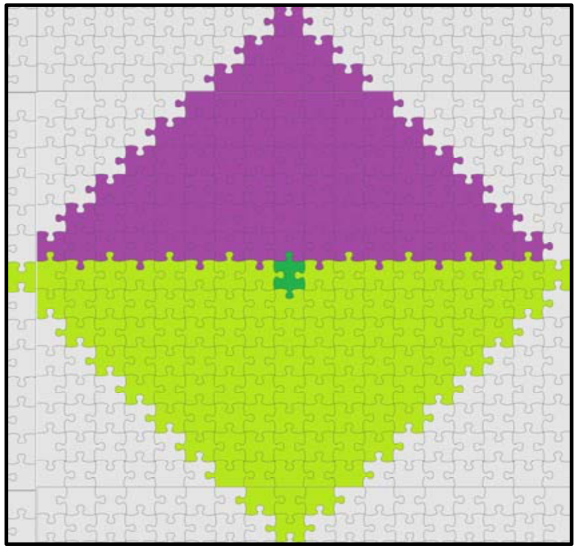
k=3



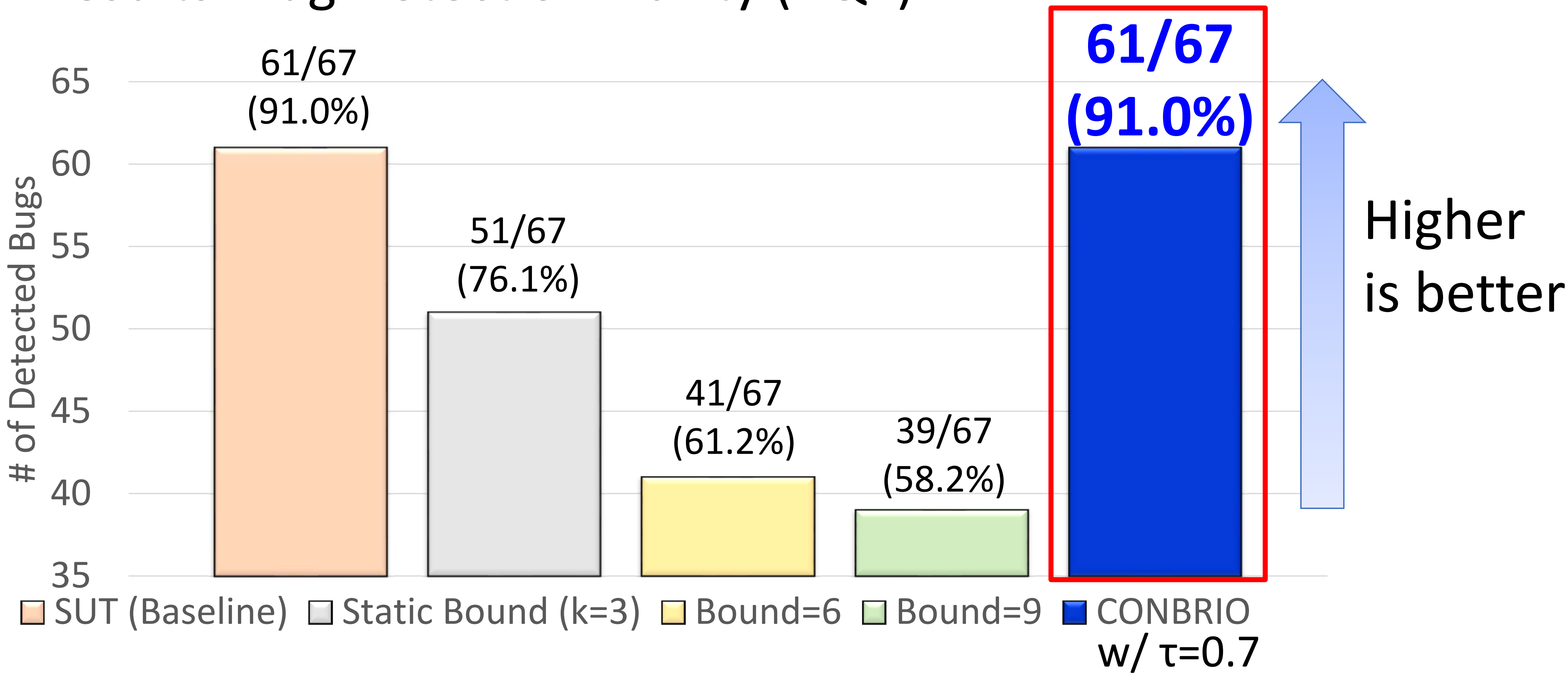
k=6



k=9

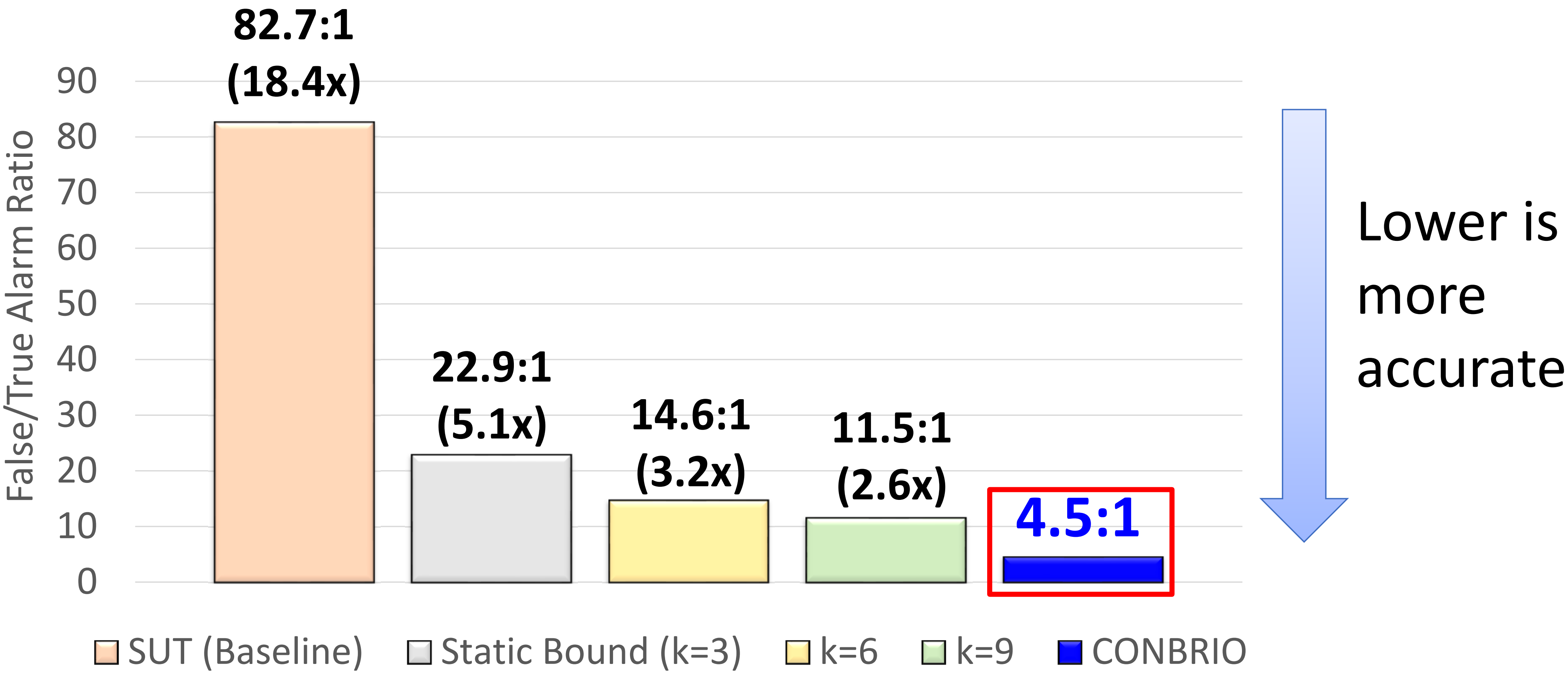


Results: Bug Detection Ability (RQ1)



CONBRIO spent **4.1 hours** on 100 machines

Results: Bug Detection Accuracy (RQ2)



Cutting-edge Accuracy of Unit Testing

Randoop: 5.9:1
[Gross et al., 2012]

Evosuite: 6.3:1
[Fraser and Arcuri, 2013]

UC-KLEE: 5.7:1
[Ramos and Engler, 2015]

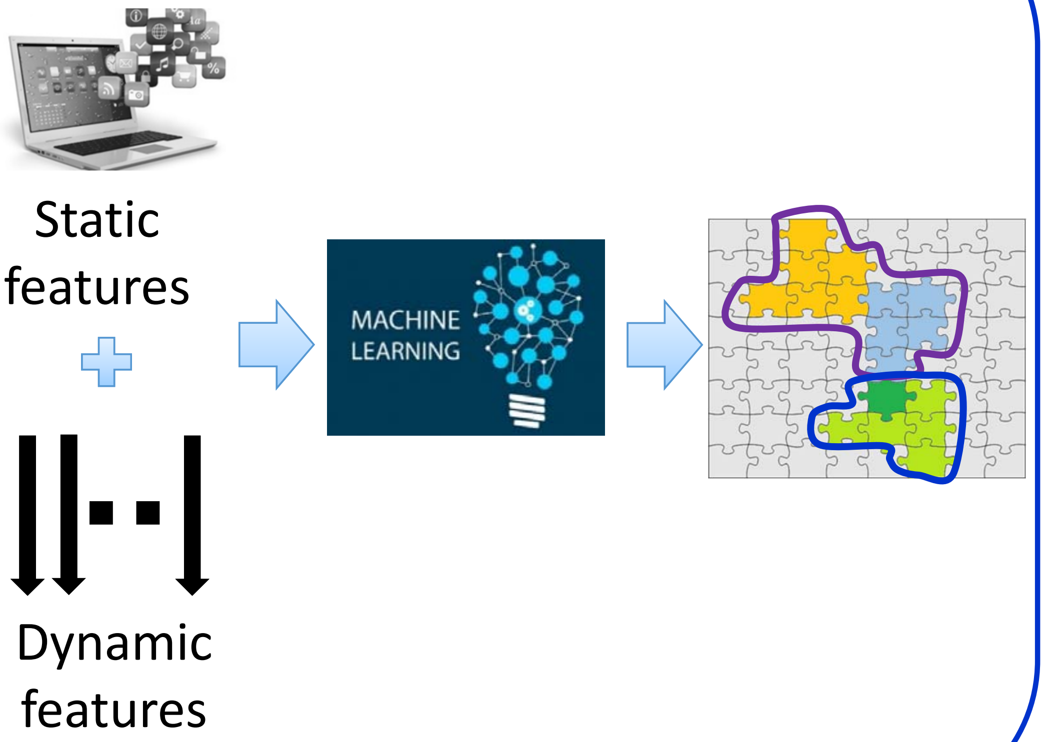
OOP features

Manual Assumption

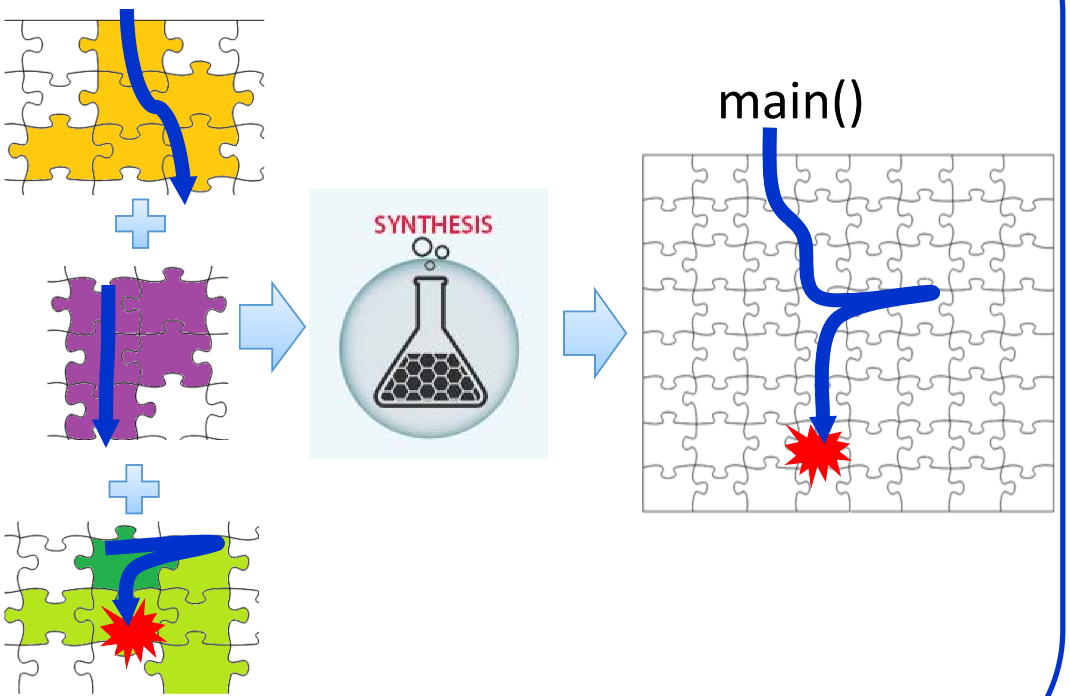
CONBRIO: 4.5:1
w/ 91% of bugs detected

Future Work

Refined Relevance Analysis



Synthesis System TCs from Unit TCs



현대모비스, AI 기반 소프트웨어 검증시스템 도입..."효율 2배로"

2018-07-22 10:00

댓글

가⁻ 가⁺

실제 현대모비스가 통합형 차체제어시스템(IBU)과 써라운드뷰모니터링 시스템(SVM) 검증에 마이스트를 시범 적용한 결과 마이스트가 처리한 검증 업무량 비중은 각각 53%, 70%로 높았다.

현대모비스는 하반기부터 소프트웨어가 탑재되는 제동,

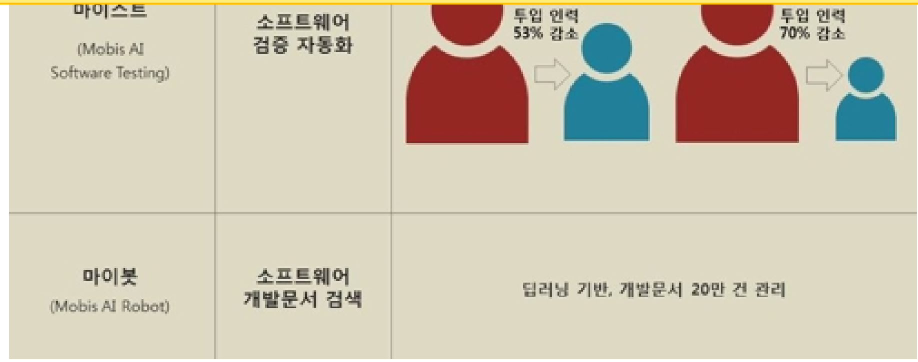
2월 4일 화요일 오후 1시 40분 그랜드홀 2

Concolic Testing for High Test Coverage and Reduced Human Effort in Automotive Industry

입했다고 22일 밝혔다.

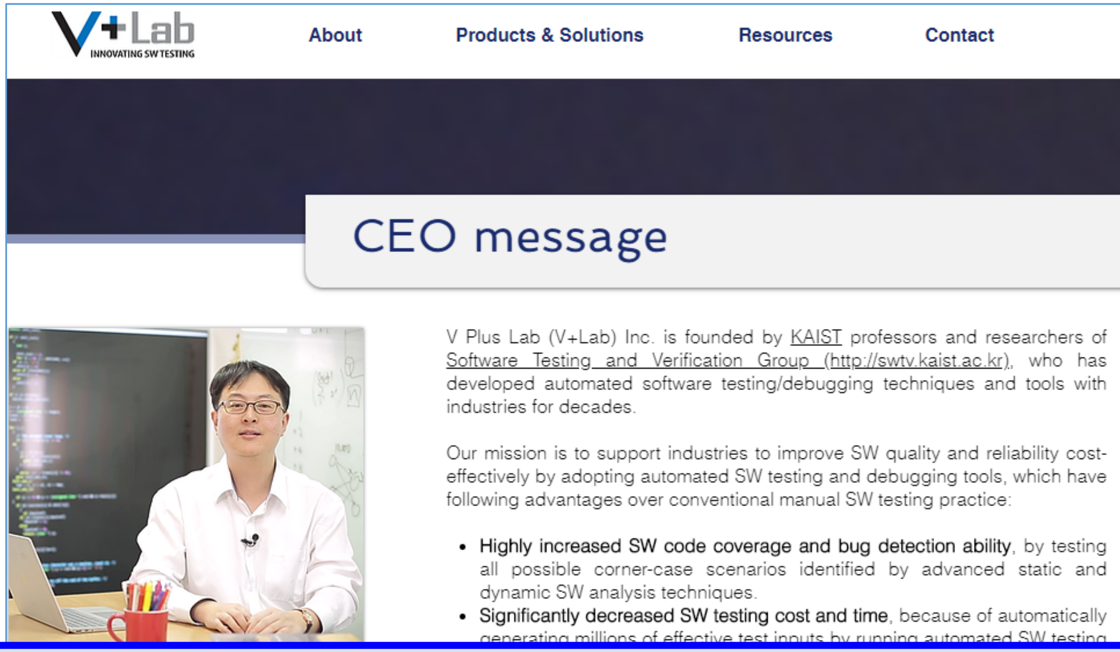
현대모비스가 카이스트 전산학부 김문주 교수와 공동으로 개발한 마이스트는 연구원을 대신해 소프트웨어 검증작업을 수행하는 AI 시스템이다.

연구원들이 설계한 알고리즘을 바탕으로 소프트웨어의 모든 연산과정을 AI로 검증한다. 기존에 수작업으로 이뤄지던 소프트웨어 검증업무를 자동화한 셈이다.



http://m.yna.co.kr/kr/contents/?cid=AKR20180720158800003&mobile

- **브이플러스랩㈜** 창업
  - 19.10 법인 설립
- 자동차/항공/국방 안전필수 SW 자동 테스팅 도구 **CROWN 2.0** 개발/판매

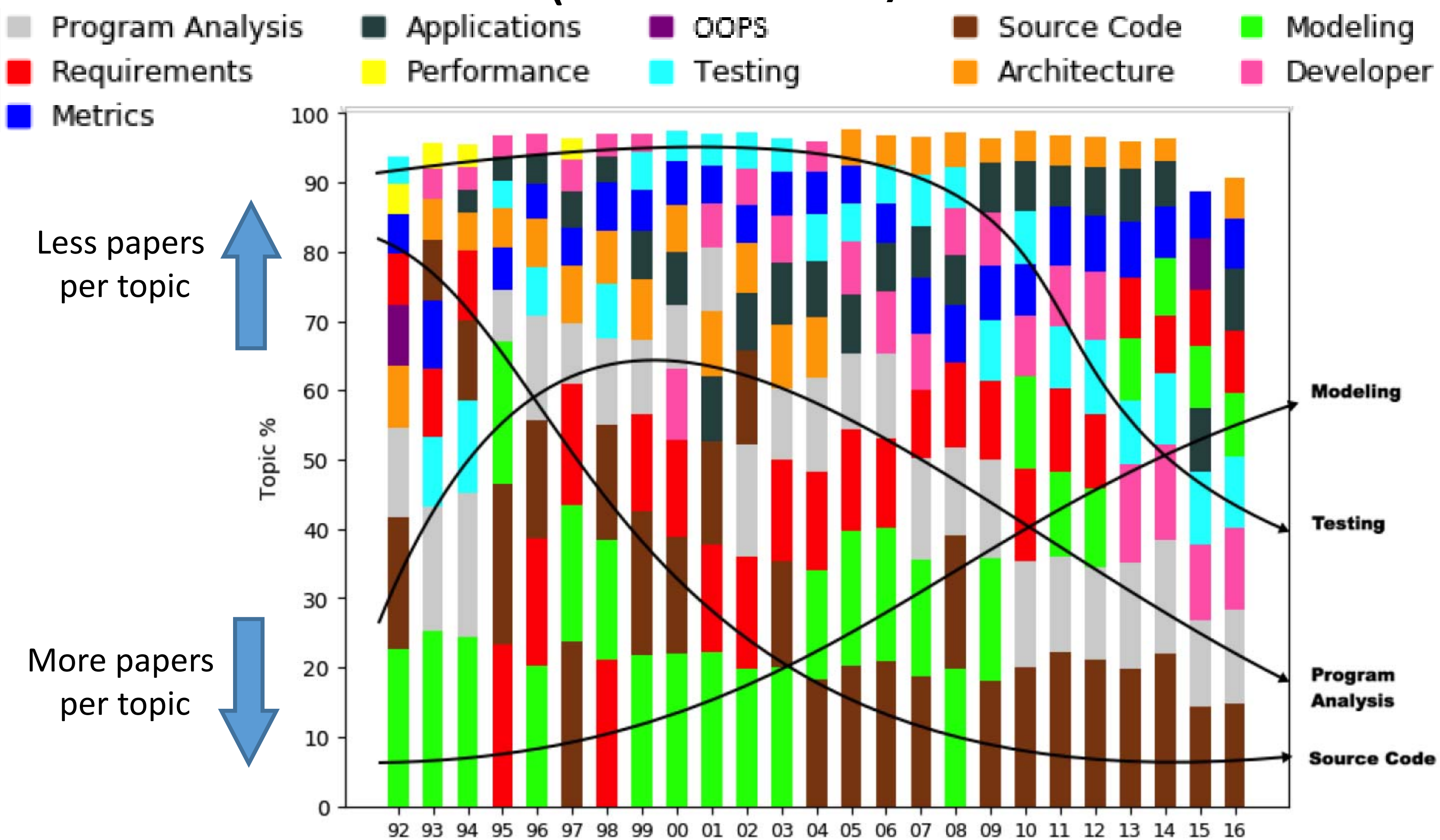


V+Lab 과 함께 하실,
열정있는 SW 분석/테스팅 전문가 모십니다!!!
contact@vpluslab.kr

SE Research Topic Trends among 11 Major Topics (1992-2016)

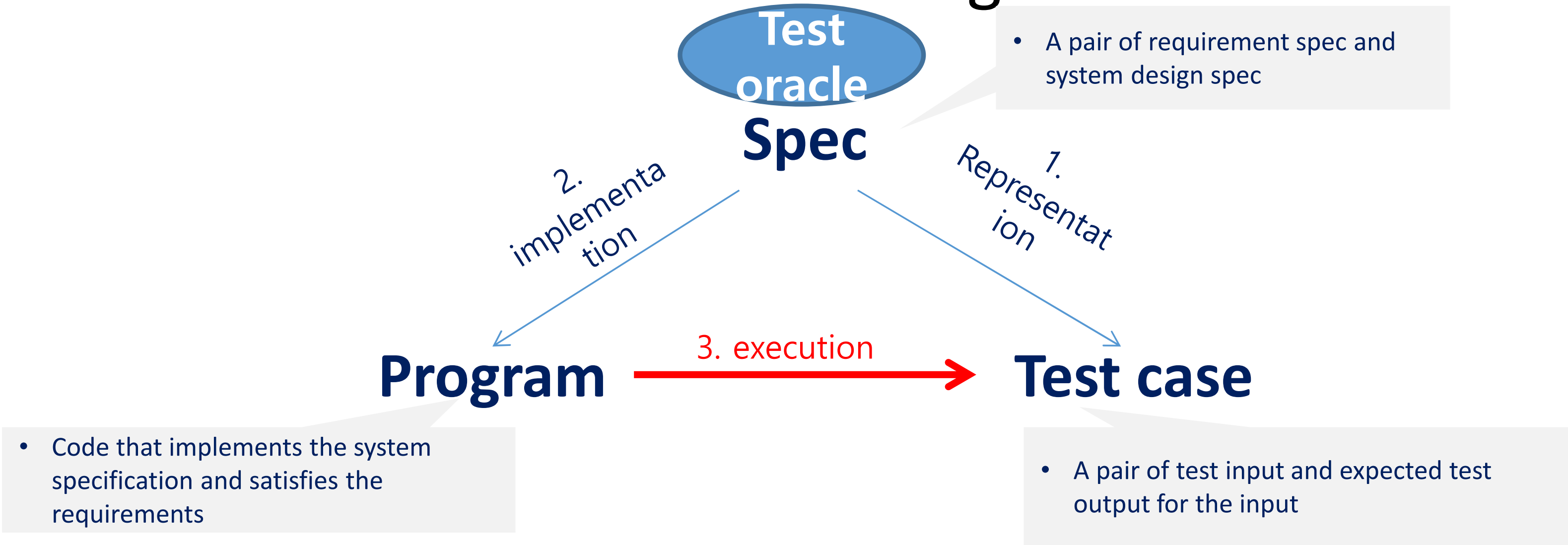
Less papers per topic

More papers per topic



G.Mathew et al., Trends in Topics in Software Engineering, IEEE TSE 2018 submission

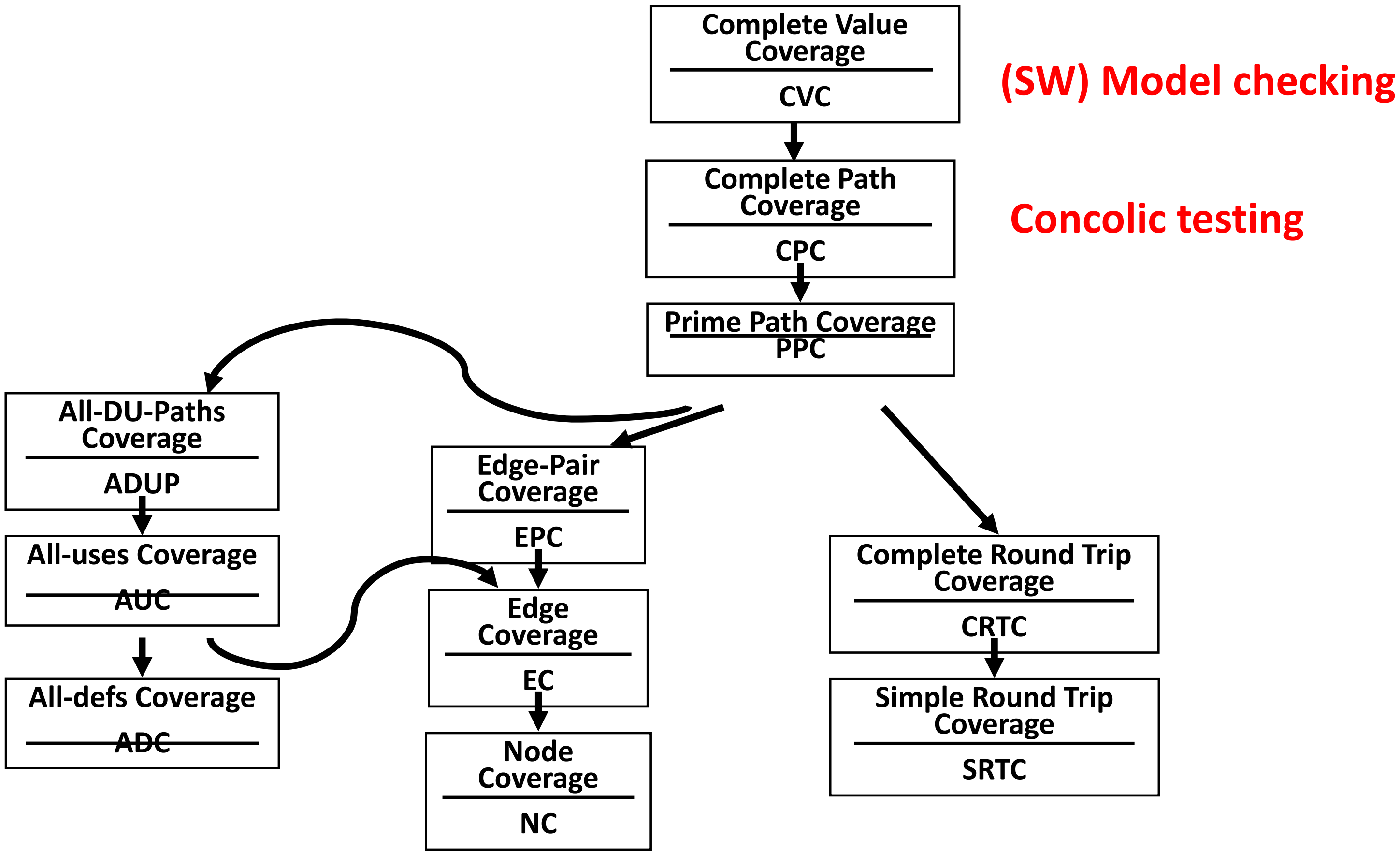
Foundation of Software Testing



Multiple targets for software testing

1. Does the test cases represent the requirement spec correctly?
   → Scenario based testing (black-box testing)
2. Is the design spec implemented as program correctly?
   → Model-based testing (grey-box testing)
3. Does the program satisfy test cases correctly?
   → Code-based testing (white-box testing)

Hierarchy of Structural/graph Coverages



Related Work on Automated Unit Testing

	Bug detection ability	False/True alarm ratio	Target languages
Function input generation [PLDI 05][FSE 05][EMSOFT 06][TAP 08][ISSTA 08][SEC 15]	**High**	**High**	**Procedural or OO languages**
Method-sequence generation [ICSE 07] [ICST 10][FSE 11] [ICSE 13]	**High**	Medium	**Object-oriented languages**
Capture system tests to generate unit tests [TSE 09] [STTT 09][ISSTA 10]	**Low**	**Low**	**Object-oriented languages**
CONBRIO	**High** (91.0% of target bugs in SIR and SPEC2006, 14 new bugs)	**Low** (4.5 false alarms per one true alarm)	**Procedural languages**

False Alarms Caused by Missing Context

```
// Target function under test
int f(int x, int y){
  int array[5] = {1,3,5,7,9};
  int n, res;
  // Alarm: Array overflow
  n = array[x];
  g(&n);
  if ((y % 2) != 0){
    // Alarm: Array overflow
    res = array[n];
  } else res = h(n);
  return res;}
```

```
// x, y, and z are inputs
int main(int x, int y, int z){
  if (z > 0) return b(y,z);
  else return c(y,z);}

int b(int x, int y){
  if (0 <= x && x < 5)
    return f(x,y);
  else return 0;}

int c(int x, int y){
  return x-y;}

void g(int *p){
  *p = *p / 2;}

int h(int x){
  return x + 2;}
```

The 42nd International Conference on Software Engineering

ICSE 2020

23-29 May 2020
Seoul, South Korea

Attending ▾ Sponsorship ▾ Tracks ▾ Organization ▾ Search Series ▾ Sign in Sign up

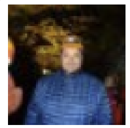
ICSE 2020 Software Engineering in Practice

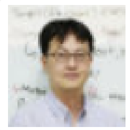
About | Call for Papers

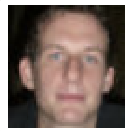
Call for Papers

The Software Engineering in Practice (SEIP) Track is the premier venue for researchers and practitioners to discuss insights, innovations and solutions to concrete software engineering problems. After many decades of research, software engineering (SE) techniques and algorithms are gaining substantial momentum in industry: more companies produce SE-based tools and more software engineers use previously published ideas in their daily projects. SEIP provides a unique forum for networking, exchanging ideas, fostering innovations, and forging long-term collaborations to address SE research that impacts directly on practice. SEIP will gather highly-qualified industrial and research participants that are eager to communicate and share common interests in software engineering. The track will be composed of invited speeches, paper presentations, reviewed talks, interactive sessions with a strong focus on software practice.

Program Committee


Mark Grechanik Program Co-Chair
University of Illinois at Chicago


Moonzoo Kim Program Co-Chair
KAIST / V+Lab
South Korea


Mathieu Acher
Programme Committee


Toshiaki Aoki Programme Committee
JAIST
Japan