

Zero false positive program analysis in Java

with dynamic symbolic execution

Dmitry Ivanov

*Huawei Saint Petersburg
Research Center*

*Director of R&D Software
Toolchain Laboratory*

korifey@gmail.com,
[@korifey_ad](https://twitter.com/korifey_ad)



Dmitry Mordvinov

*Huawei Saint Petersburg
Research Center*

*Team leader of symbolic
execution research group*

Associate professor at SpbU

mordvinov.dmitry@gmail.com



What is plan for next 50 minutes?

- [Episode I, **Survey**] – What kinds of analysis exist?
- [Episode II, **Theory**] – How symbolic execution works?
- [Episode III, **Practice**] – How to use SMT-solver?
- [Epilogue, **Product**] – How to get value for products?

EPISODE I : ANALYSIS

Program analysis: Why?

- Conformance to specification
- **Error detection**
- **Security vulnerability detection**
- Coding standards checks
- Metrics calculation
- Architecture extraction & analysis

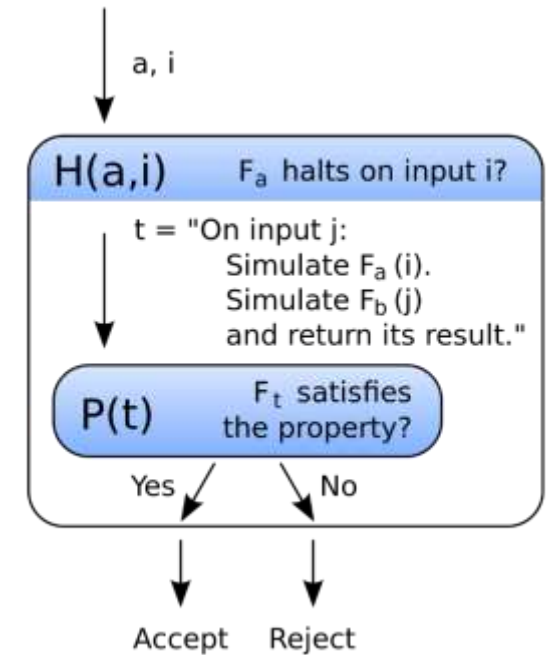
Program analysis: History

- 9 September 1947 – physical moth removing (debugging)



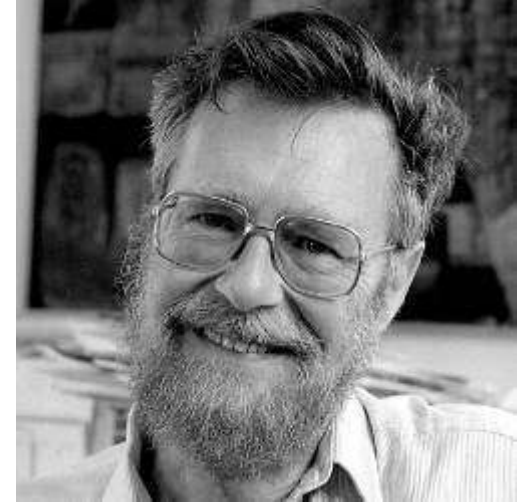
Program analysis: History

- 9 September 1947 – physical moth removing (debugging)
- 1950th – testing to specification conformance
- 1951st Henry Gordon Rice's Theorem: All non-trivial semantic properties of programs behavior are undecidable ☹️



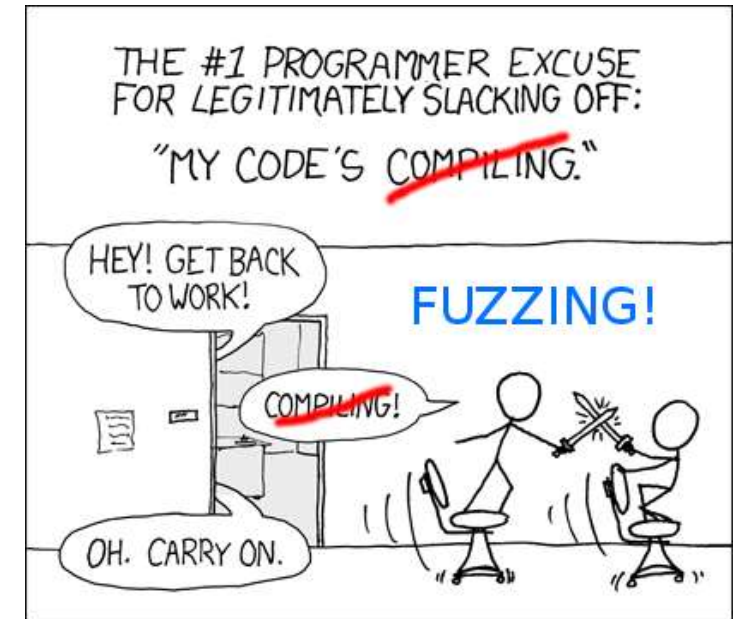
Program analysis: History

- 9 September 1947 – physical moth removing (debugging)
- 1950th – testing to specification conformance
- 1951st Henry Gordon Rice's Theorem: All non-trivial semantic properties of programs behavior are undecidable.
- 1970th – coding standards and first generation of source code static analysis
- 1971st – Edsger Wybe Dijkstra: [Program testing can be used to show the presence of bugs, but never to show their absence!](#) 😊



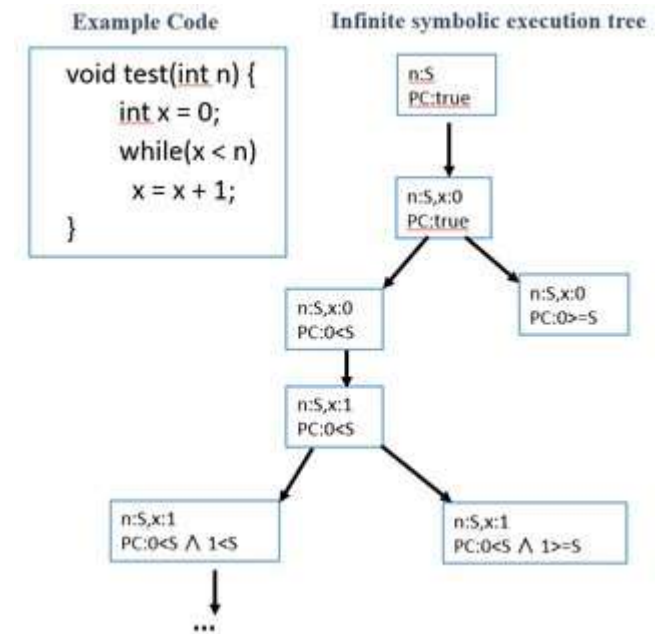
Program analysis: History

- 9 September 1947 – physical moth removing (debugging)
- 1950th – testing to specification conformance
- 1951st Henry Gordon Rice's Theorem: All non-trivial semantic properties of programs behavior are undecidable.
- 1970th – coding standards and first generation of source code static analysis
- 1971st – Edsger Wybe Dijkstra: Program testing can be used to show the presence of bugs, but never to show their absence!
- 1975th – **Symbolic Execution** idea and naïve implementation
- 1980th – randomized testing (**Fuzzing**)



Program analysis: History

- 9 September 1947 – physical moth removing (debugging)
- 1950th – testing to specification conformance
- 1951st Henry Gordon Rice's Theorem: All non-trivial semantic properties of programs behavior are undecidable.
- 1970th – coding standards and first generation of source code static analysis
- 1971st – Edsger Wybe Dijkstra: Program testing can be used to show the presence of bugs, but never to show their absence!
- 1975th – Symbolic Execution idea and naïve implementation
- 1980th – randomized testing (Fuzzing)
- 1990th – second generation of source code static analysis (data-flow analysis)
- 2000th – third generation of source code static analysis (IFDS + **SMT**)
- 2000th – **Symbolic Execution** engines (KLEE, S2E, Java Symbolic Pathfinder, ...)
- 2010th – Hybrid analysis methods



Program analysis **!=** Static code analysis

Method	+	-
Manual code inspection	Can detect complex error	Expert experience needed
Formal verification	Highest confidence	Lowest performance Lowest scalability
Static code analysis	High scalability High performance	False positives
Fuzzing	High performance High scalability No False Positives	Low control over analysis Low probability of error detection
Dynamic symbolic execution	High precision No False Positives	Low performance Low scalability

Program errors for source code static analysis

Plain text

Lexer based

AST based

Coding style guides for structuring source code

- Indentation error
- Naming rules violations
- Empty then-branch
- Lack of default-branch
- Assignment in conditional expression


Path-sensitive

Condition-sensitive

Context-sensitive

Run-time errors, logical errors, security vulnerabilities

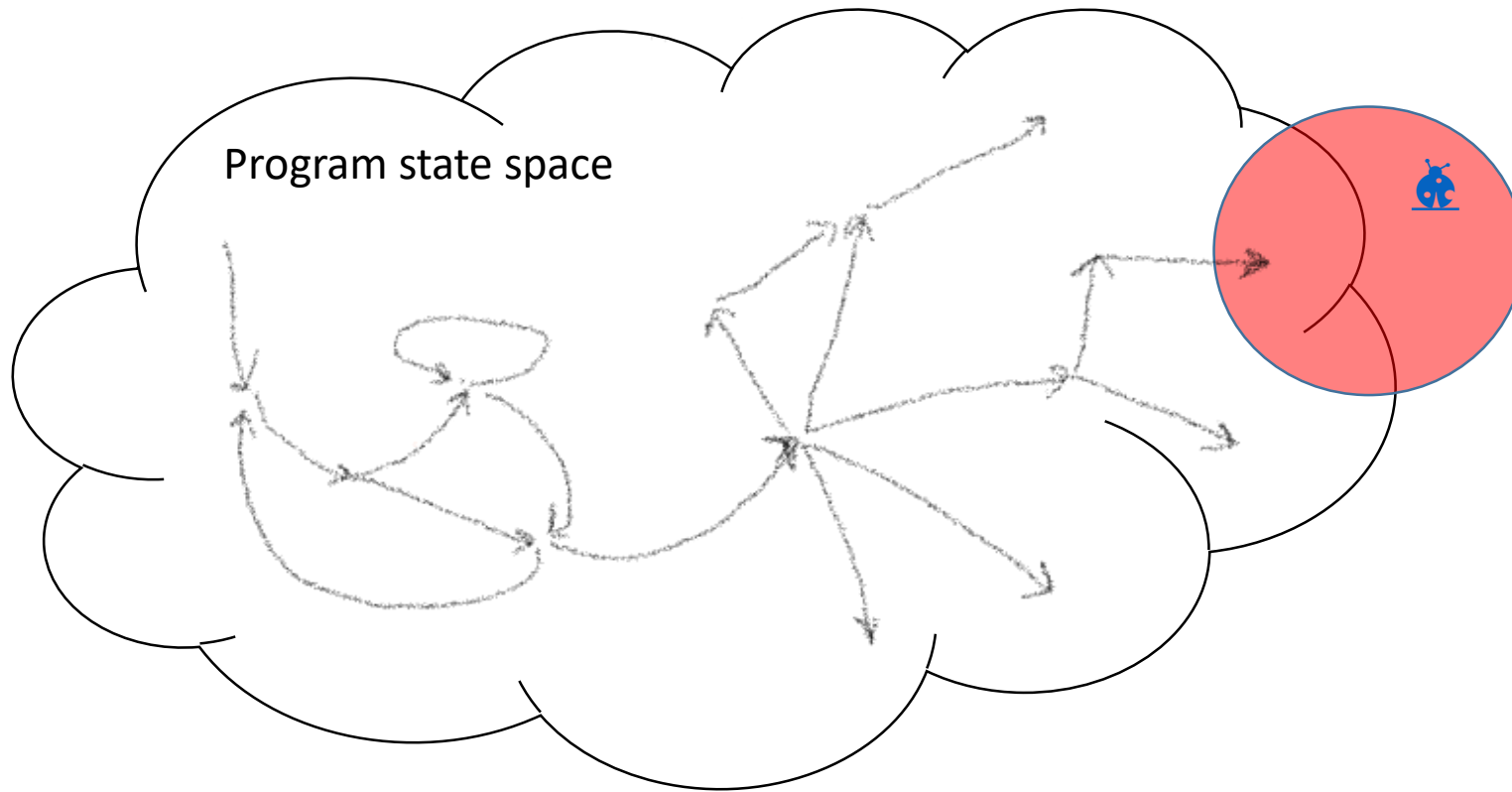
- Division by zero
- Resource leak/uncontrolled consumption
- NULL/uninitialized/dangling pointer dereference
- Buffer under/overflow
- Use after free
- SQL/Command injection
- ...

The logo for Checkstyle, featuring the word "checkstyle" in a black, sans-serif font. A red squiggly line underlines the word, and a yellow pencil icon is positioned at the end of the line.The logo for PMD, featuring the letters "Pmd" in a stylized font. The "P" is black, and "md" is red. Below the letters is the tagline "DON'T SHOOT THE MESSENGER" in small, black, uppercase letters.The logo for Detekt, featuring the word "Detekt" in a white, sans-serif font inside a purple rectangular box.The logo for SpotBugs, featuring the word "SpotBugs" in a white, sans-serif font inside a blue rectangular box. Below the box is the tagline "Find bugs in Java Programs" in small, white, uppercase letters.The logo for Fortify, featuring the word "Fortify" in a blue, sans-serif font. Above the word is the "MICRO FOCUS" logo in small, blue, uppercase letters.The logo for SonarQube, featuring the word "sonarqube" in a black, sans-serif font. To the right of the word is a blue icon consisting of three curved lines.The logo for Infer, featuring a purple circle with a white "T" shape inside, followed by the word "Infer" in a black, sans-serif font.The logo for Coverity, featuring a black circle with a white "V" shape inside, followed by the word "coverity" in a black, sans-serif font.The logo for Klocwork, featuring a blue and green "K" shape followed by the word "klocwork" in a black, sans-serif font.

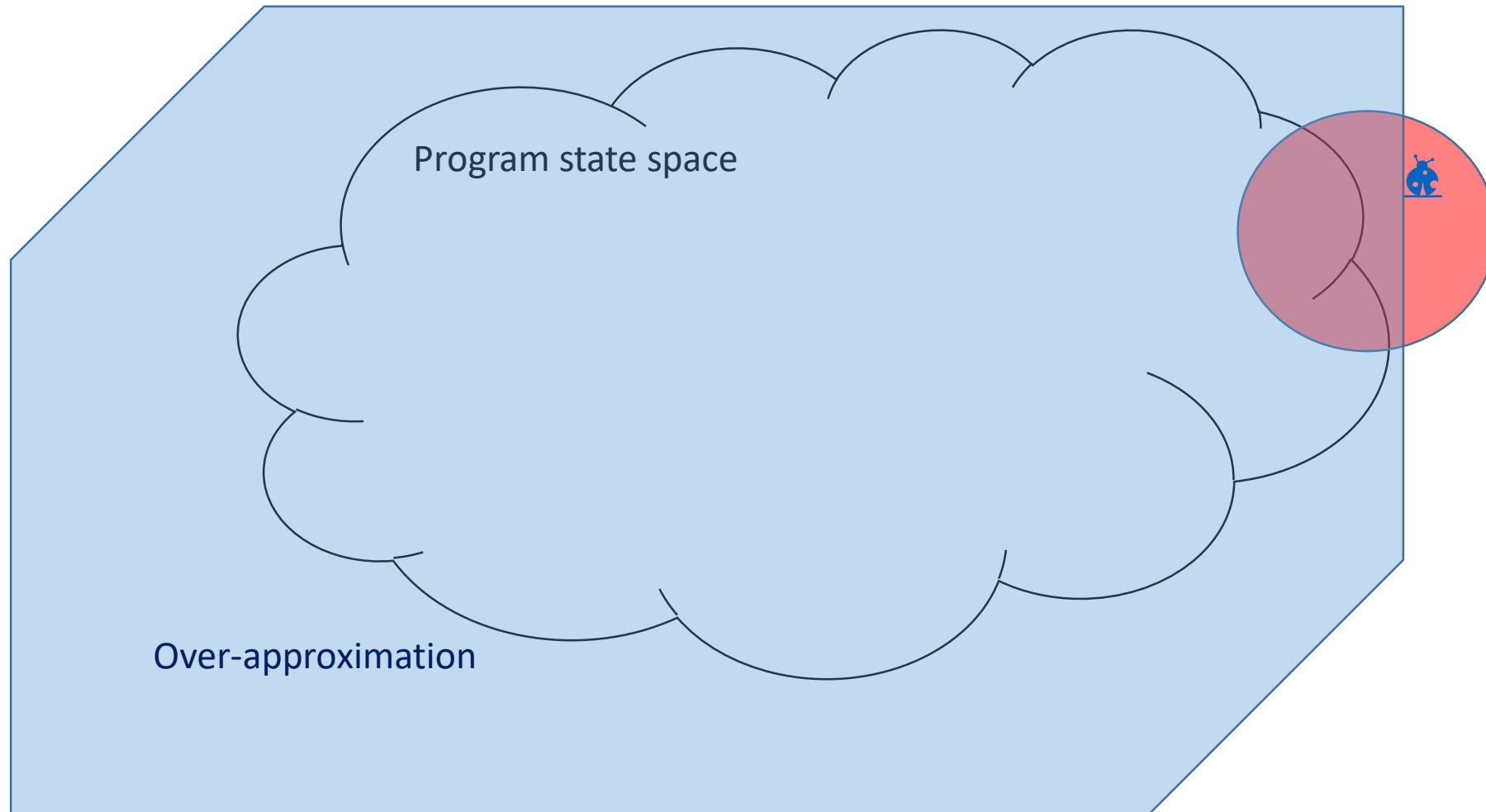
LIVE DEMO

IntelliJIDEA

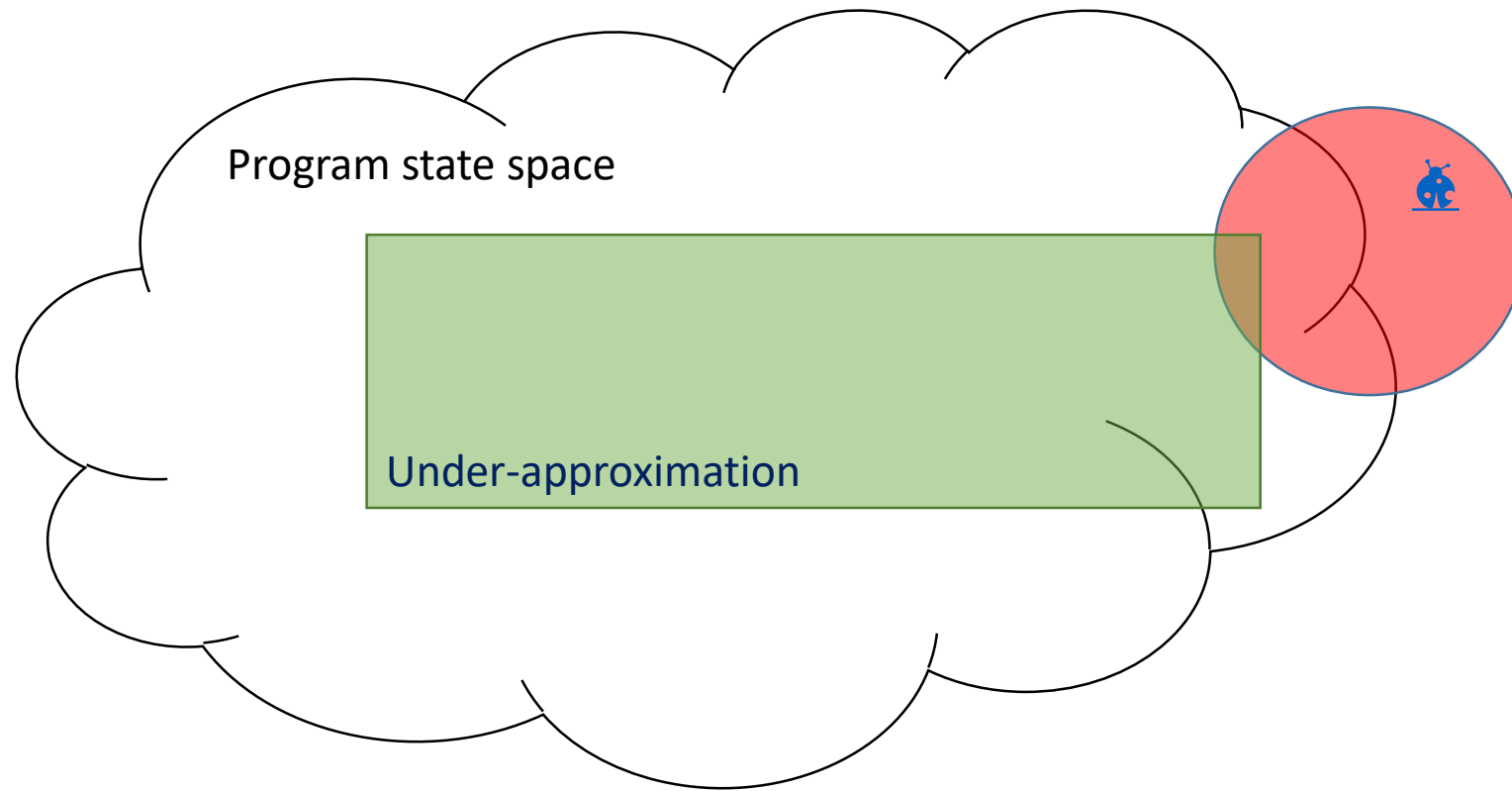
Program approximations



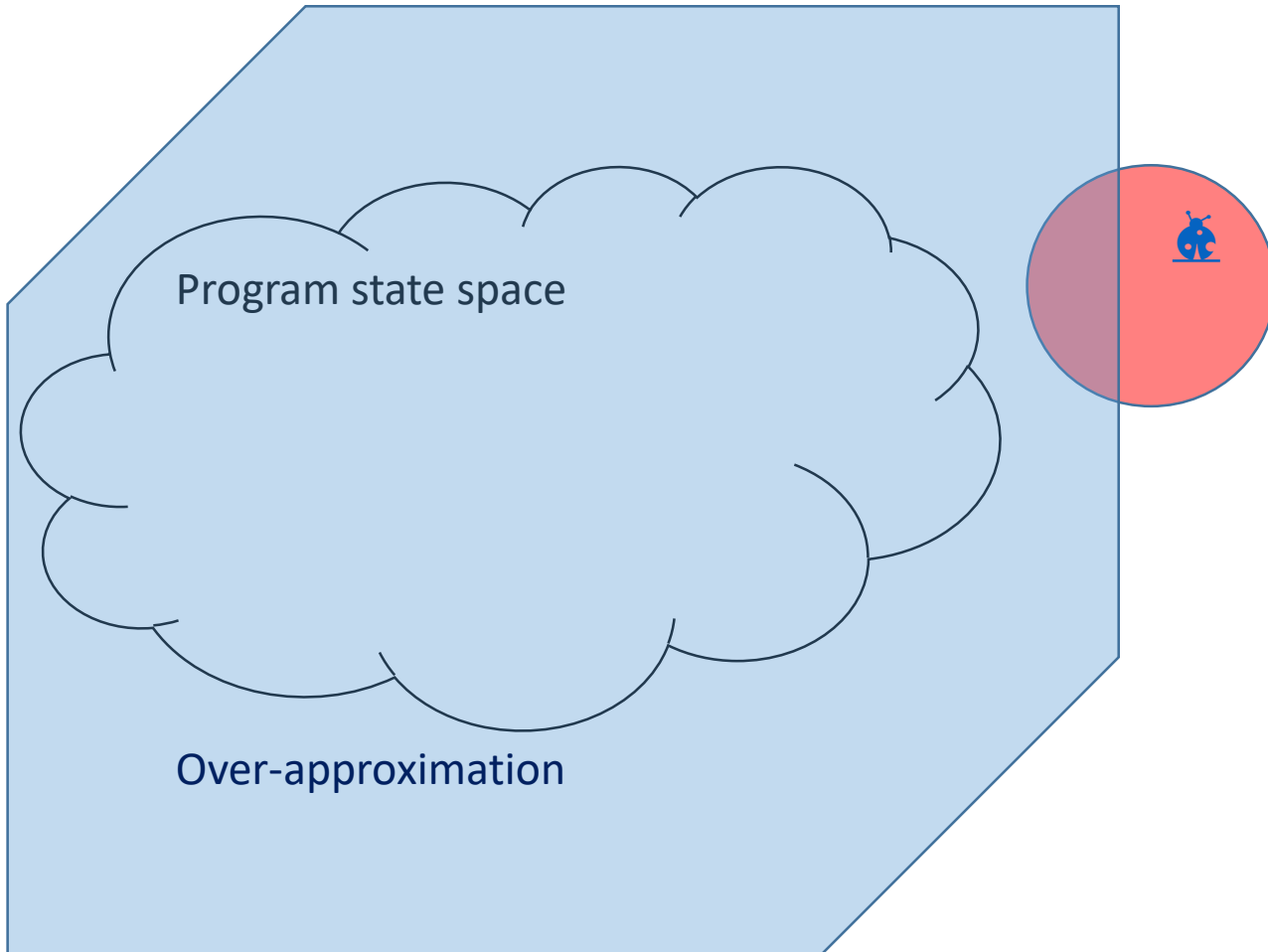
Program approximations: over-approximation



Program approximations: under-approximation

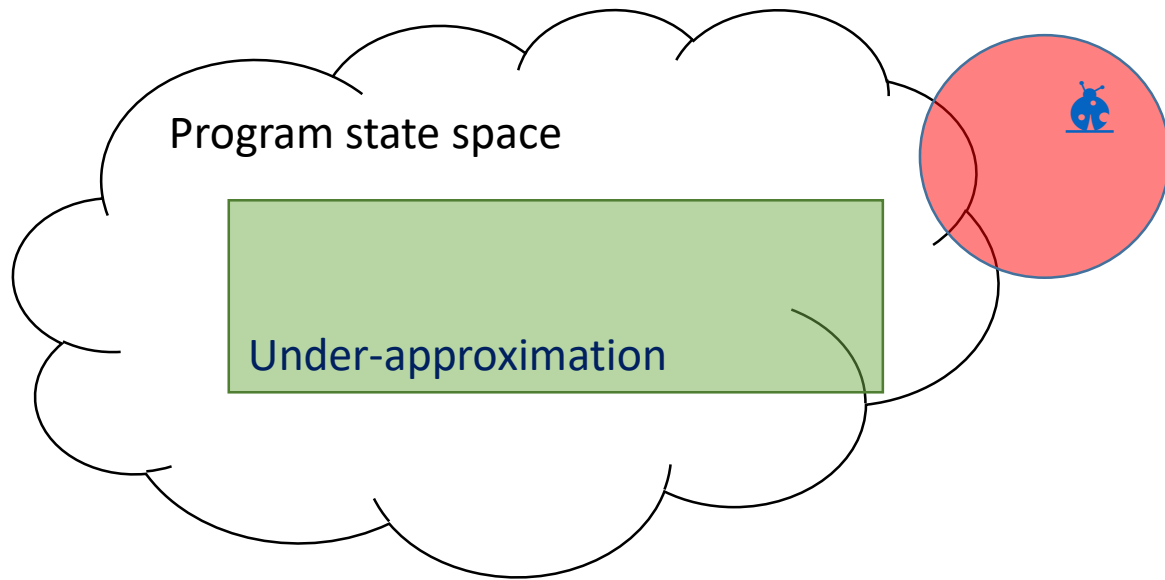


Over-approximation: false-positives



- Approximations can be **too coarse**
- It might lead to **false positives**
- In practice every static analyzer yield false positives
- False positives are **frustrating** and cause **waste of time**

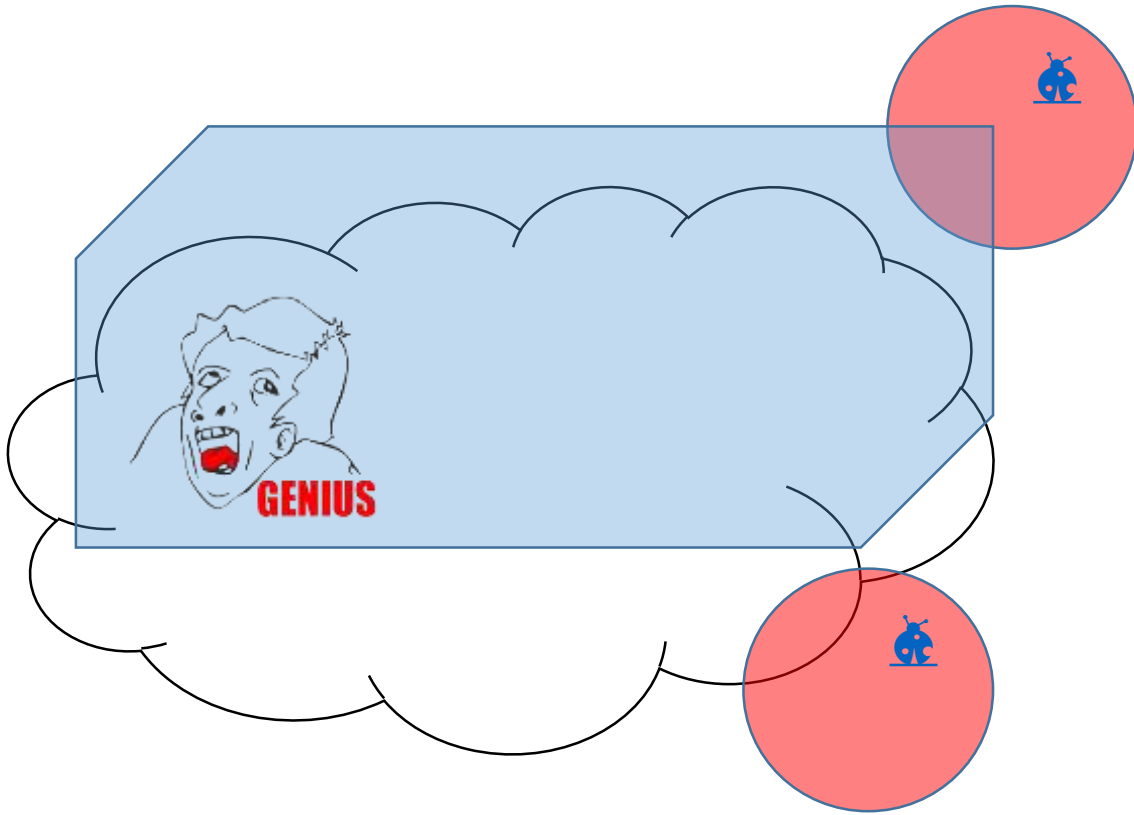
Under-approximation: false-negatives



How bad are false negatives?

- False negatives = missed errors
- Missing errors is fatal only for **safety-critical** systems
- In practice, most of over-approximating analyzers **miss errors as well**

Under-approximation: false-negatives



How bad are false negatives?

- False negatives = missed errors
- Missing errors is fatal only for **safety-critical** systems
- In practice, most of over-approximating analyzers **miss errors as well**
 - Avoiding too many false alarms
 - Intraprocedural analysis
 - Environment interactions
 - ...

EPISODE II: THEORY

How to verify program is correct?

```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            assert false;  
    // else  
    ...  
}
```



ETAPS
EUROPEAN CONFERENCE ON THEORETICAL COMPUTER SCIENCE
THEORY & PRACTICE OF SOFTWARE

FASE 2021

3rd Competition on Software Testing (Test-Comp 2021)

FASE '21
Tue, March ??, 2021
Luxembourg

3rd Intl. Competition on Software Testing held at FASE 2021 in Luxembourg.

- Competition Description
- 2020 Competition Report

Motivation

Tool competitions are a special form of comparative evaluation, where each tool has a team of

<https://test-comp.sosy-lab.org/2021/>



ETAPS
EUROPEAN CONFERENCE ON THEORETICAL COMPUTER SCIENCE
THEORY & PRACTICE OF SOFTWARE

TACAS 2021

10th Competition on Software Verification (SV-COMP 2021)

TACAS '21
Thur, March ??, 2021
Luxembourg

10th Intl. Competition on Software Verification held at TACAS 2021 in Luxembourg.

- 2020 Competition Report (results of the competition and a lot of detailed information on SV-COMP 2020)

Motivation

Competition is a driving force for the invention of new methods, technologies, and tools. This web page describes the competition of software-verification tools, which will take place at TACAS.

<https://sv-comp.sosy-lab.org/2021/>

Fuzzers

Random testing

Random inputs generation until crash

BLACK-BOX
TECHNIQUE

Adaptive random testing (ART)

```
while (coverage is not enough) {  
    generate new test()  
    add to suite if coverage increases  
}
```

Instrument
program

GRAY-BOX
TECHNIQUE

Evolutionary algorithm

- Generations (test suites)
- Cross-over existing tests
- Mutate existing tests

Tool	Score	Ranking
t3	145.27	2.30
evosuite	255.43	2.38
randoop	154.34	2.51
tardis	66.80	3.73
sushi	39.84	4.09

Symbolic execution

WHITE-BOX
TECHNIQUE

Symbolic Virtual Machine State

Instruction: **<Enter>**

Symbolic Memory (SM): $x = x0, y = y0$

Path Condition (PC): true



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: `if (x > 0)`

SM: `x = x0, y = y0`

PC: `true && x0 > 0`



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: **abs = x**

SM: $x = x0$, $y = y0$, **abs = $x0$**

PC: $x0 > 0$



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```


Symbolic execution

Symbolic Virtual Machine State

Instruction: `if (y == 42)`

SM: $x = x0, y = y0, \text{abs} = x0$

PC: $x0 > 0 \ \&\& \ y0 == 42$



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: `if (abs + y < 0)`

SM: `x = x0, y = y0, abs = x0`

PC: `x0 > 0 && y0 == 42 && x0 + y0 < 0`



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: **ERROR**

SM: $x = x0$, $y = y0$, $abs = x0$

PC: $x0 > 0 \ \&\& \ y0 == 42 \ \&\& \ x0 + y0 < 0$

(Ask SMT Solver “Is PC satisfiable?”)



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

Symbolic execution

Symbolic Virtual Machine State

Instruction: `if (abs + y < 0) //else`

SM: `x = x0, y = y0, abs = x0`

PC: `x0 > 0 && y0 == 42 && ¬(x0+y0<0)`

(Go to **else** branch: negate last condition)



```
void absAndSum(int x, int y) {  
  
    int abs;  
  
    if (x > 0)  
        abs = x;  
    else  
        abs = -x;  
  
    if (y == 42)  
        if (abs + y < 0)  
            ERROR;  
    // else  
    ...  
}
```

SMT solver

SMT = Satisfiability modulo theories

PC:

$x0 > 0$

&&

$y0 == 42$

&&

$x0 + y0 < 0$

```
(declare-const x0 Int)
```

```
(declare-const y0 Int)
```

```
(assert (> x0 0))
```

```
(assert (= y0 42))
```

```
(assert (< (+ x0 y0) 0))
```

```
(check-sat)
```

SMT-LIB2 syntax

SMT solver: Z3

PC:

```
x0 > 0
&&
y0 == 42
&&
x0 + y0 < 0
```

```
(declare-const x0 Int)
(declare-const y0 Int)

(assert (> x0 0))
(assert (= y0 42))
(assert (< (+ x0 y0) 0))

(check-sat) ;unsatisfiable

sample.smt
```

Try it here online:

<https://compsys-tools.ens-lyon.fr/z3/index.php>

Or use command line tool:

```
#> apt-get install z3
```

```
#> z3 -smt2 sample.smt
```

```
#unsat
```

SMT solver: theories

Integer arithmetic theory

PC:

$x_0 > 0$

&&

$y_0 == 42$

&&

$x_0 + y_0 < 0$

```
(declare-const x0 Int)
```

```
(declare-const y0 Int)
```

```
(assert (> x0 0))
```

```
(assert (= y0 42))
```

```
(assert (< (+ x0 y0) 0))
```

```
(check-sat) ;unsatisfiable
```

Bitvector theory

```
(set-option :pp.bv-literals false)
```

```
(declare-const x0 (_ BitVec 32))
```

```
(declare-const y0 (_ BitVec 32))
```

```
(assert (bvsgt x0 (_ bv0 32)))
```

```
(assert (= y0 (_ bv42 32)))
```

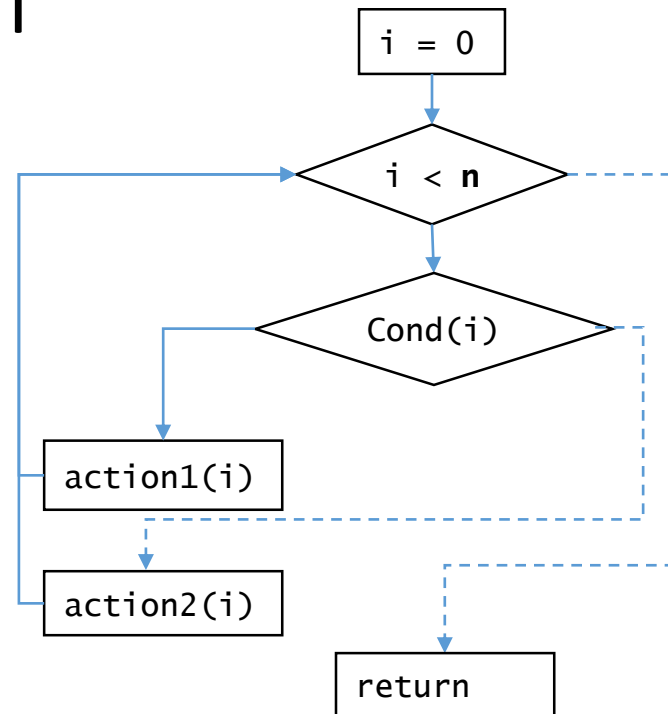
```
(assert (bvslt (bvadd x0 y0) (_ bv0 32)))
```

```
(check-sat) ;satisfiable
```

```
(get-model) ;x0 = MAXINT-41, y0 = 42
```

Path Explosion

```
for (i=0; i<n; i++)  
    if (cond(i))  
        action1(i);  
    else  
        action2(i);  
  
return;
```

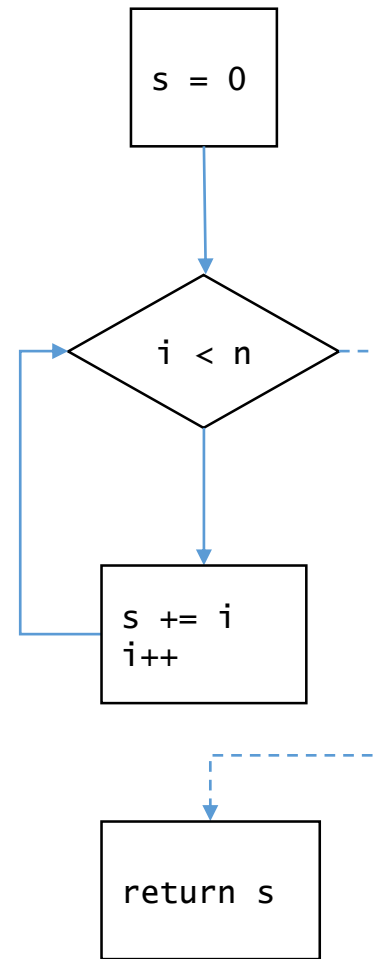


N	#paths
0	1
1	2
2	4
3	8
4	16
8	32

- Symbolic conditional statements fork the execution state
- N iterations of loop with conditional statement can fork 2^N times

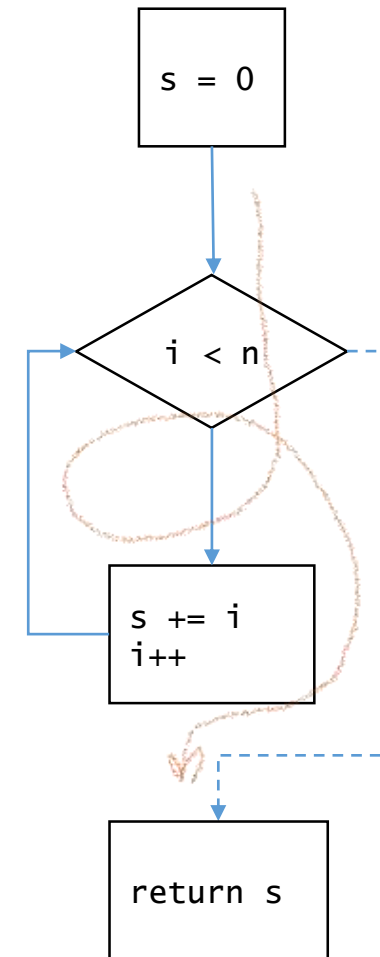
Path Selection

```
int s = 0;  
for (i=0; i<n; i++)  
    s += i;  
  
return s;
```



Path Selection

```
int s = 0;  
for (i=0; i<n; i++)  
    s += i;  
  
return s;
```

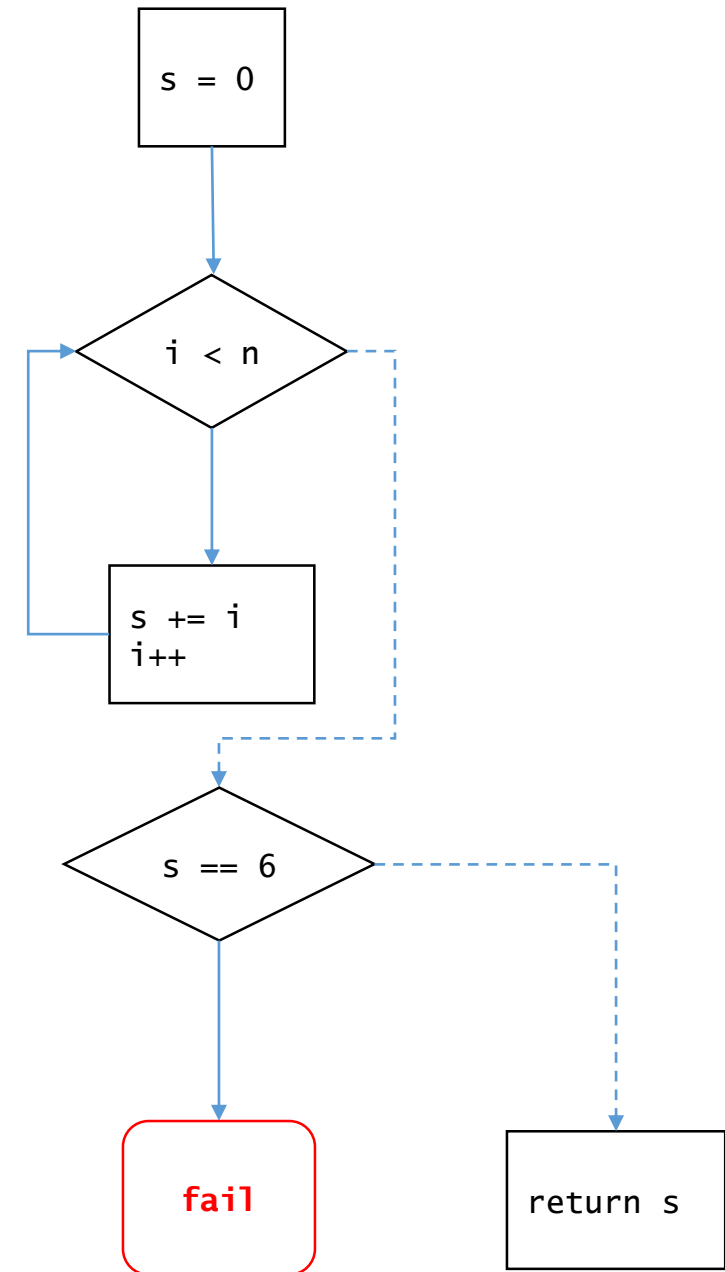


Path Selection: Heuristics

```
int s = 0;  
for (i=0; i<n; i++)  
    s += i;
```

```
if (s == 6)  
    assert false;
```

```
return s;
```

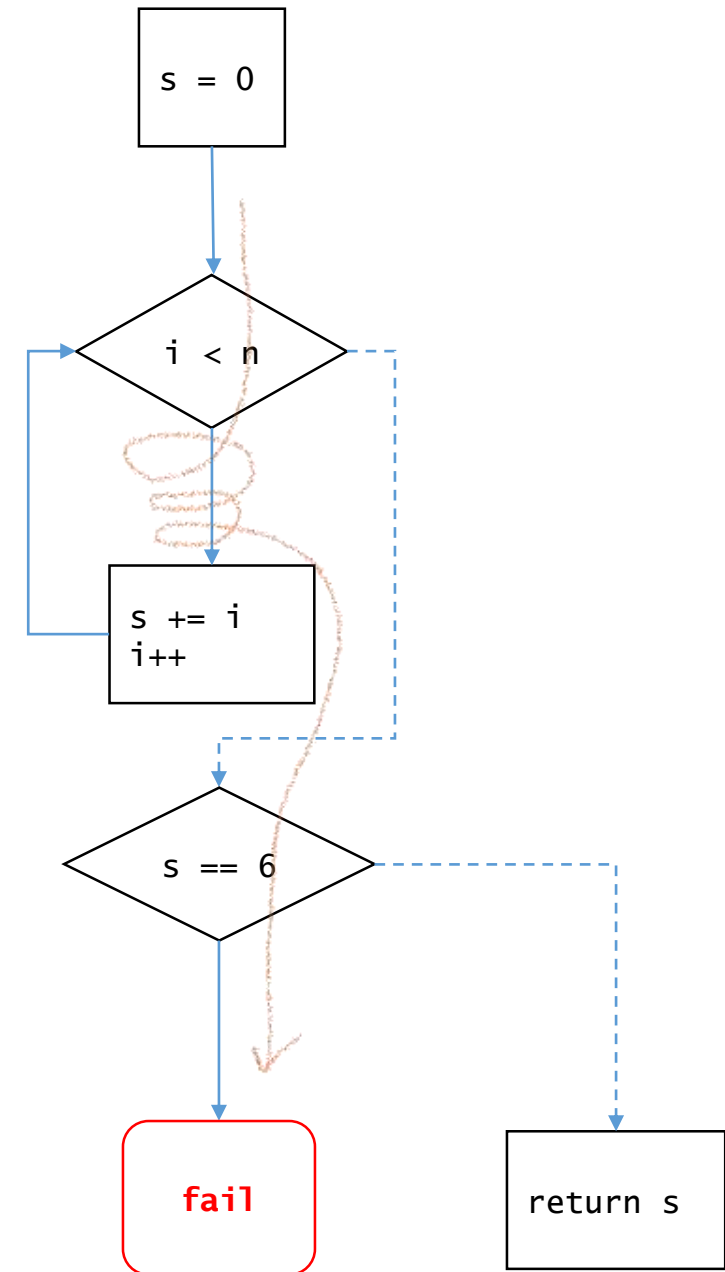


Path Selection: Heuristics

```
int s = 0;  
for (i=0; i<n; i++)  
    s += i;
```

```
if (s == 6)  
    assert false;
```

```
return s;
```

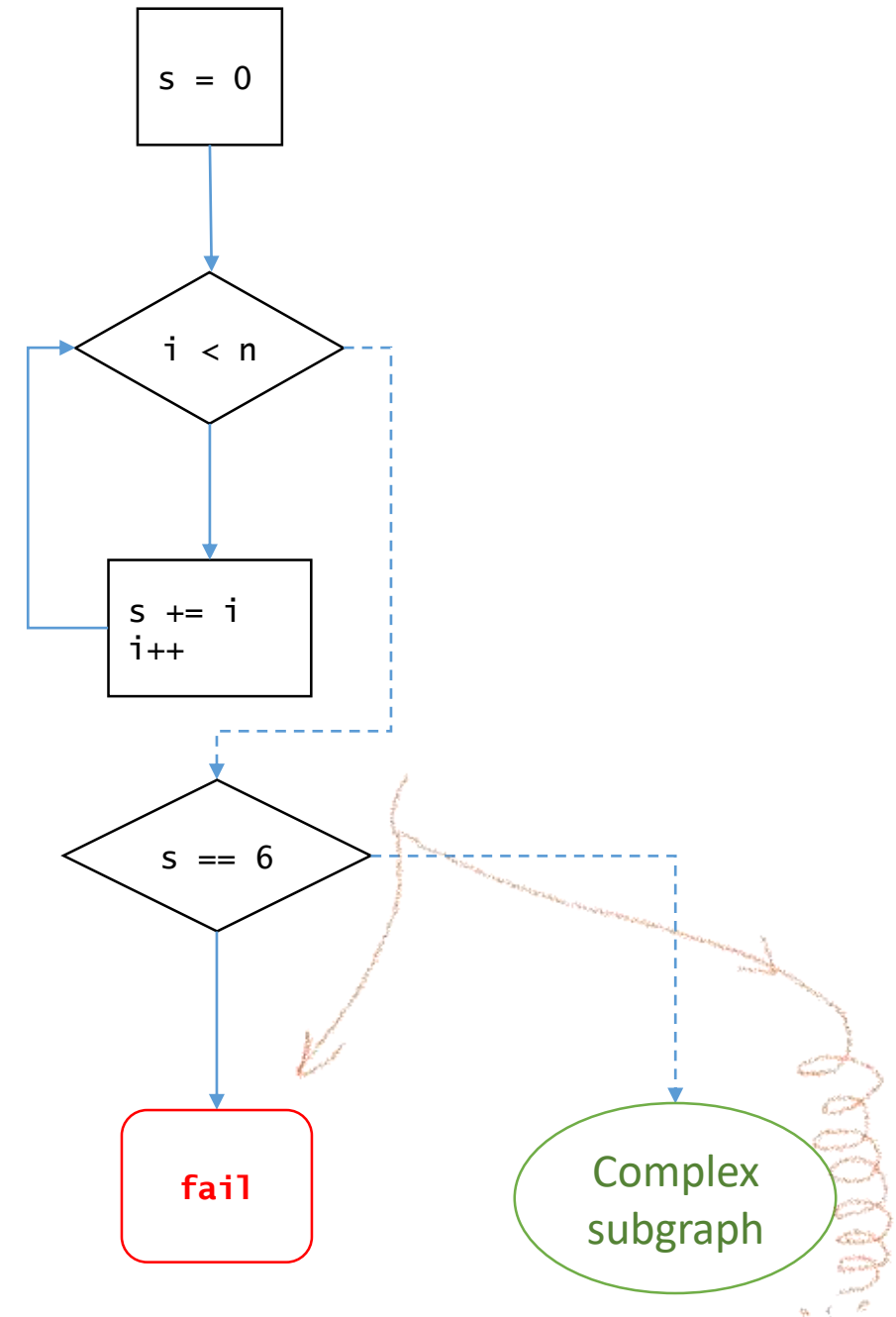


Path Selection

```
int s = 0;  
for (i=0; i<n; i++)  
    s += i;
```

```
if (s == 6)  
    assert false;
```

```
complexSubgraph();  
...
```



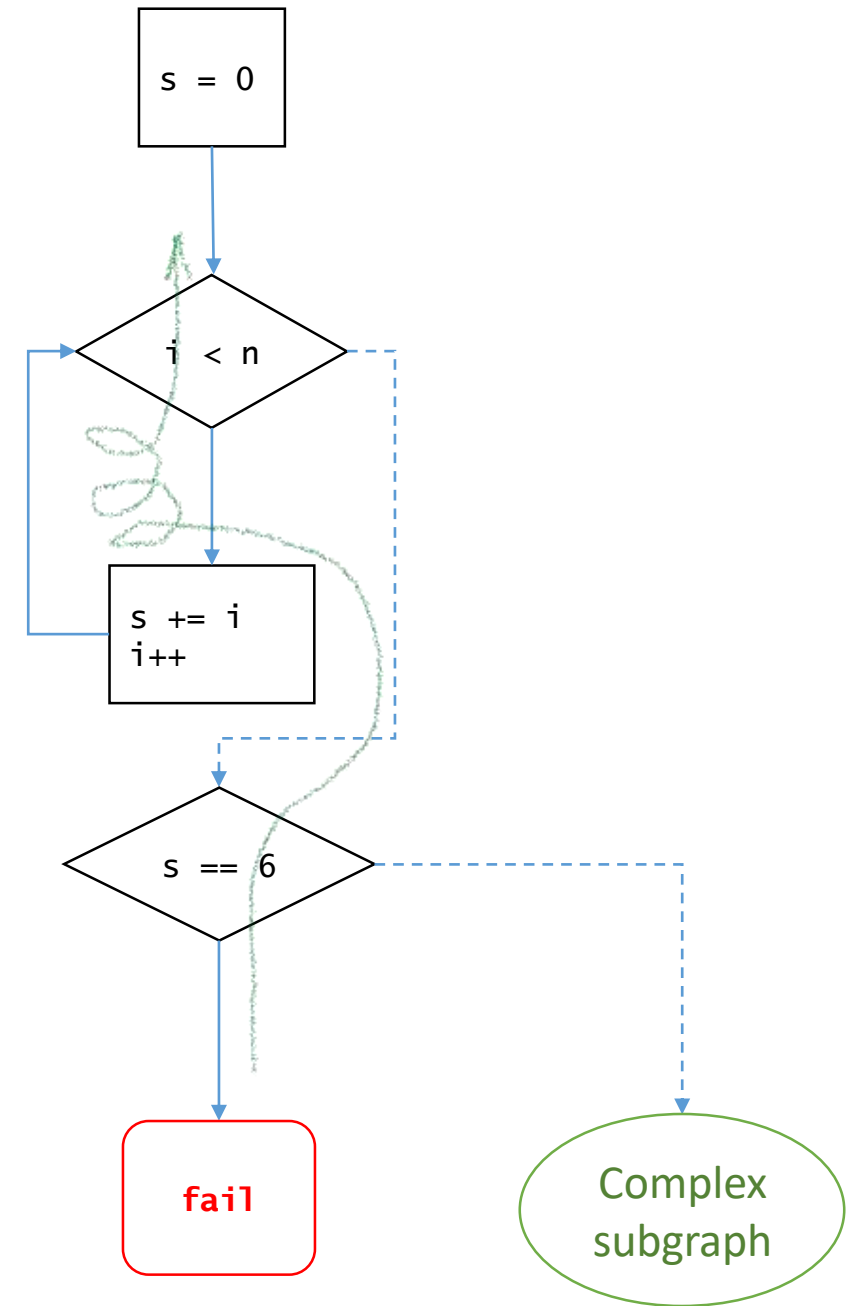
Path Selection: Backward

```
int s = 0;  
for (i=0; i<n; i++)  
    s += i;
```

```
if (s == 6)  
    assert false;
```

```
complexSubgraph();
```

...



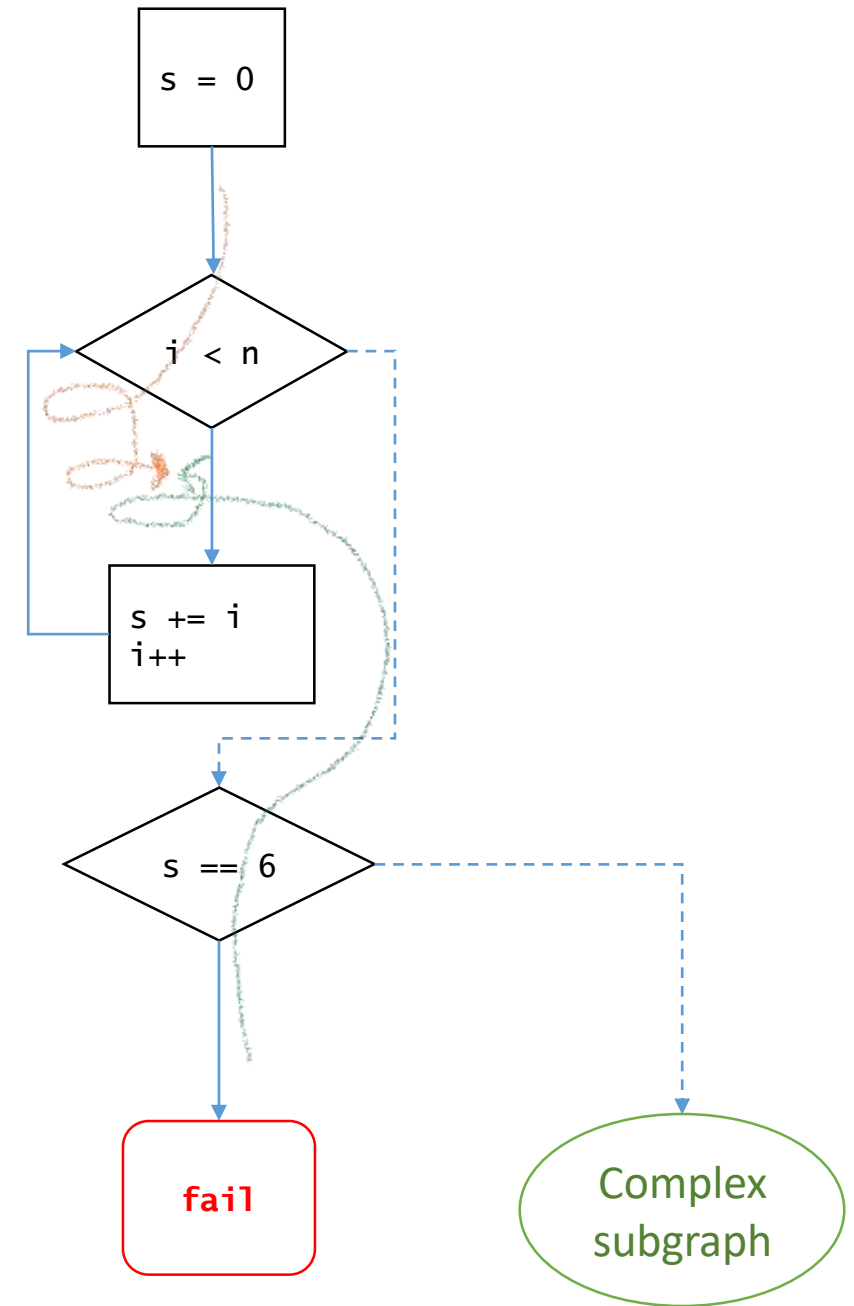
Path Selection: Bidirectional

```
int s = 0;  
for (i=0; i<n; i++)  
    s += i;
```

```
if (s == 6)  
    assert false;
```

```
complexSubgraph();
```

...



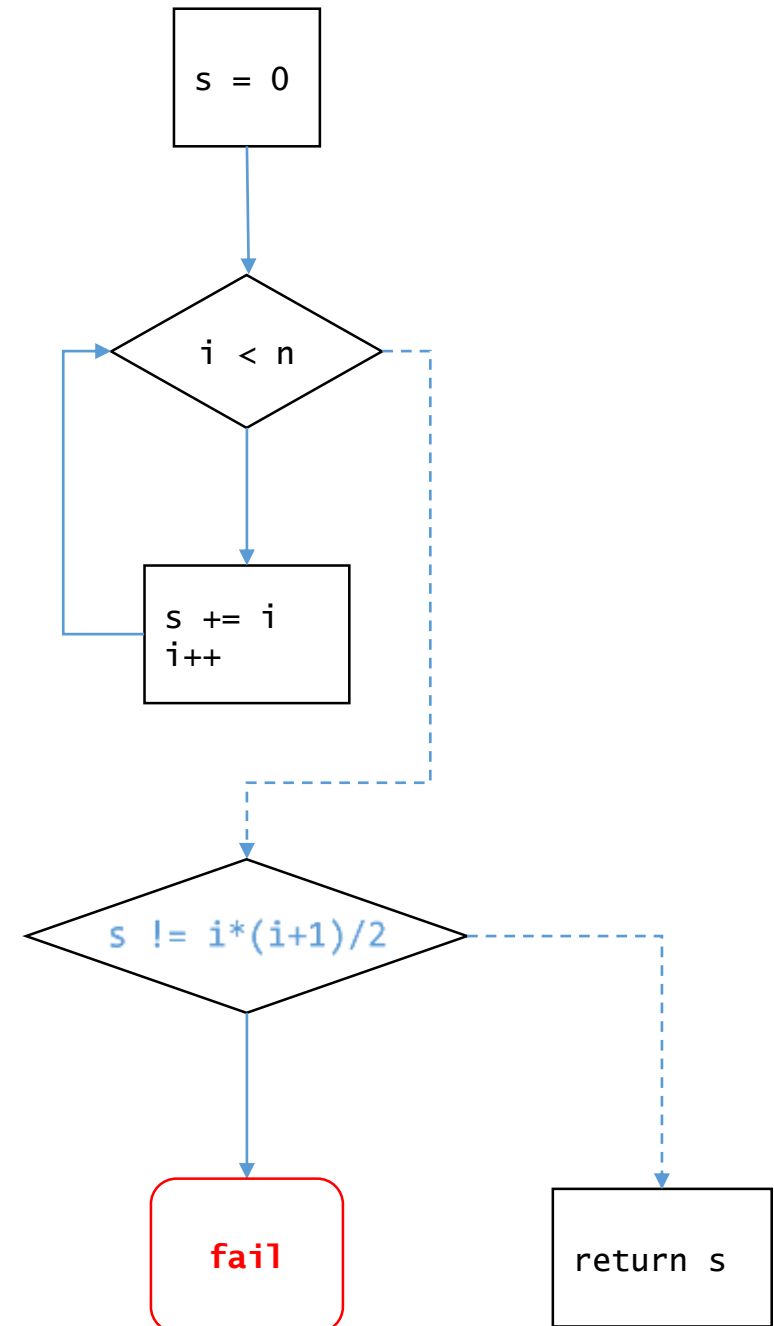
Path Selection: Invariants

```
int s = 0;  
for (i=0; i<n; i++)  
    s += i;
```

```
if (s != i*(i+1)/2)  
    assert false;
```

```
return s;
```

Inductive invariant:
 $s == i*(i+1)/2$



Code block invariants

- Loops and recursion can lead to the unbounded amount of different program behaviours
- Solution: **over-approximate** them!

```
int max = 0;
```

```
for (i=0; i<a.length; i++)  
    max = Math.max(Math.abs(a[i], max));
```

```
if (max < 0)  
    assert false;
```

```
return s;
```

Over-approximate the whole loop with:
`max >= 0`

Heap and Symbolic Execution

```
public void check(A x, A y)
{
  x.i = 1;
  y.i = 2;
  if (x.i == 2) {
    assert false;
  }
}
```

```
f = new HashMap<Object, Int>()
```

```
(declare-const x0 Addr)
(declare-const y0 Addr)
(declare-const f0 (Array Addr Int))
f := (store f0 x0 1)
f := (store (store f0 x0 1) y0 2)
(select (store (store f0 x0 1) y0 2) x0 == 2)
```

```
(check-sat)
x0 == y0
```

Theory of arrays

Symbolic execution: problems

1. How to deal with *path explosion* ?
2. How to handle loops / recursion?
3. How to present Heap in symbolic memory?
4. How to invoke native functions/syscalls?
5. What to do with concurrency?
6. if (sha256(x) == “try to solve this!”)

1. Path selection strategy, bidirectional symbolic execution
2. Try calculate “Inductive invariant” – Horn clauses
3. SMM: Theory of Arrays + Theory of Bitvectors
4. Write mock / make value concrete and execute / forget
5. Hard: $\#concurrent_states = \#states \wedge \#threads$
6. Rare in real programs, bypass

[A Survey of Symbolic Execution Techniques](#)

EPISODE III: PRACTICE

SAW/SMT by Example

$$\begin{aligned} \text{circle} + \text{circle} &= 10 \\ \text{circle} \times \text{square} + \text{square} &= 12 \\ \text{circle} \times \text{square} - \text{triangle} \times \text{circle} &= \text{circle} \end{aligned}$$

$$\text{triangle} = ?$$

```
#!/usr/bin/python
from z3 import *
```

```
circle, square, triangle = Ints('circle square triangle')
s = Solver()
s.add(circle+circle==10)
s.add(circle*square+square==12)
s.add(circle*square-triangle*circle==circle)
print s.check()
print s.model()
```

```
sat
[triangle = 1, square = 2, circle = 5]
```

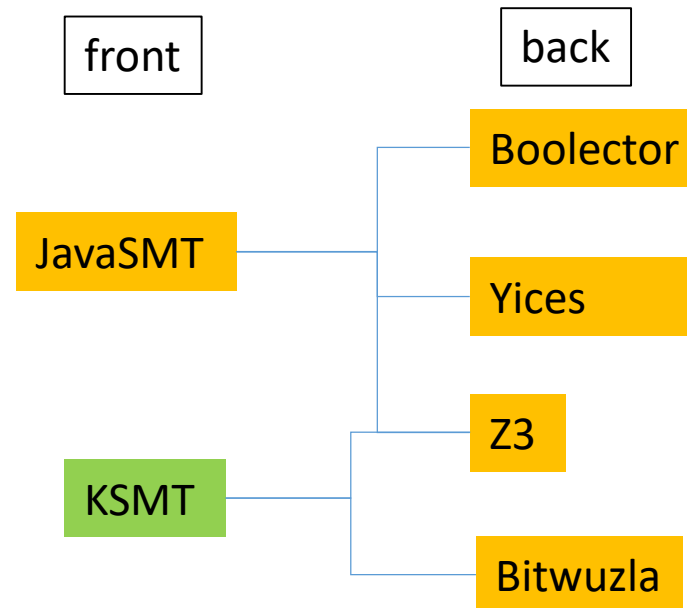
SAWSM by Example

9		6		7		4		3
			4			2		
	7			2	3		1	
5						1		
	4		2		8		6	
		3						5
	3		7				5	
		7			5			
4		5		1		7		8



LIVE DEMO

With KSMT



<https://github.com/UnitTestBot/ksmt>

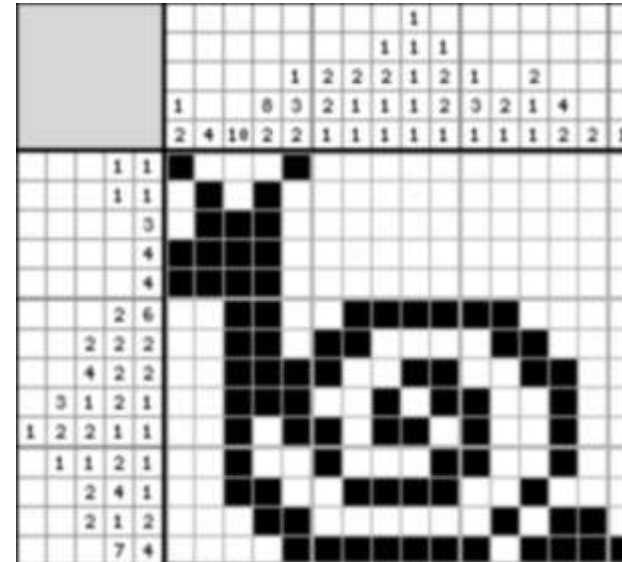
Homework Problem: Nonograms

1. Solve Nonograms using SAT/SMT technique

- <https://webpbn.com/export.cgi>
- *Hint: Book “SAT/SMT by example”*

2. (*) Solve colored nonogram

3. (**) Experiment with solvers and choose the best



SMT-COMP

The International Satisfiability Modulo Theories (SMT) Competition.

SMT-COMP 2020

The 15th International Satisfiability Modulo Theories (SMT) Workshop 2020, affiliated with IJCAR 2020

the competitors and results of the competition

News

- 31 Jul 2020 [Competition Results Available](#)
- 05 Jul 2020 [Competition Results and Benchmark List published](#)
- 29 May 2020 [Benchmark List published](#)



SMT-solvers becomes twice faster every 1.5 years

EPILOGUE : PRODUCT

How to reproduce deep bugs with auto-generated tests?

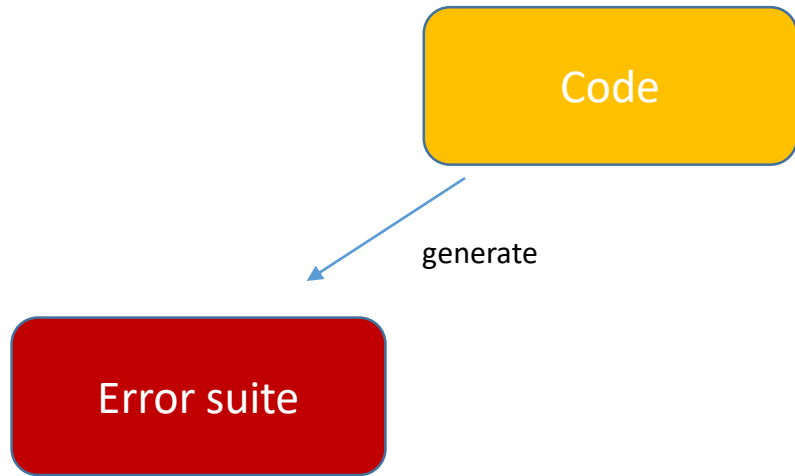


Code



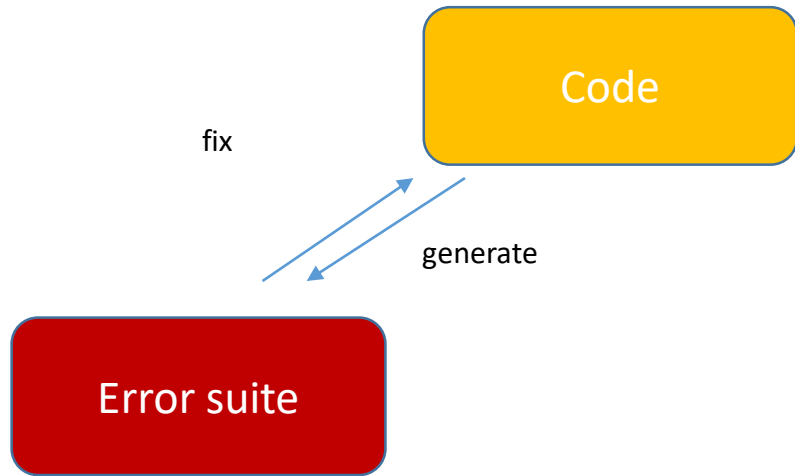
Code contains NPE,
StackOverflows and so on

Error suite



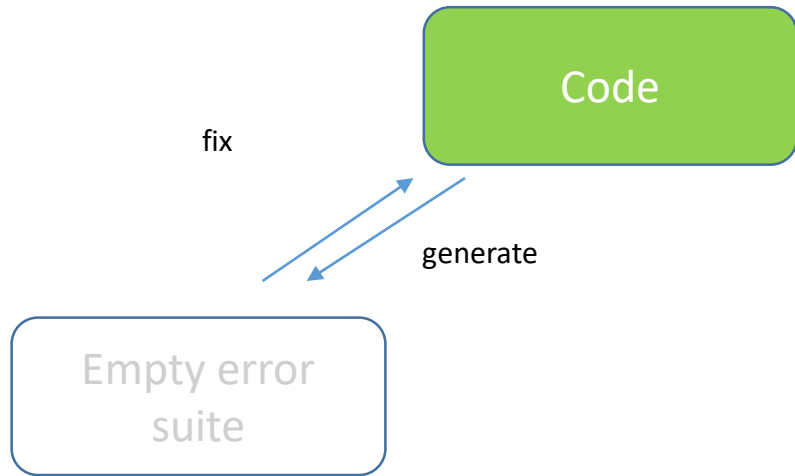
Code contains NPE,
StackOverflows and so on

Error suite



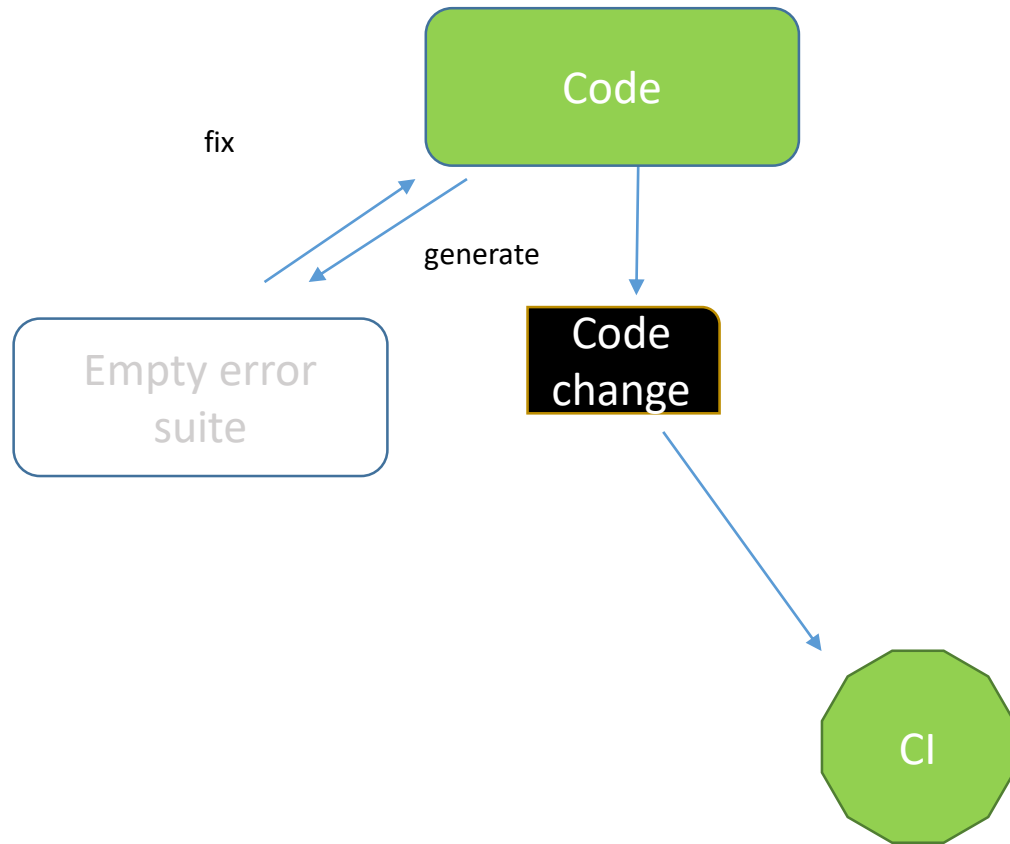
Code contains NPE,
StackOverflows and so on

Error suite



Code hasn't NPE,
StackOverflows and so on

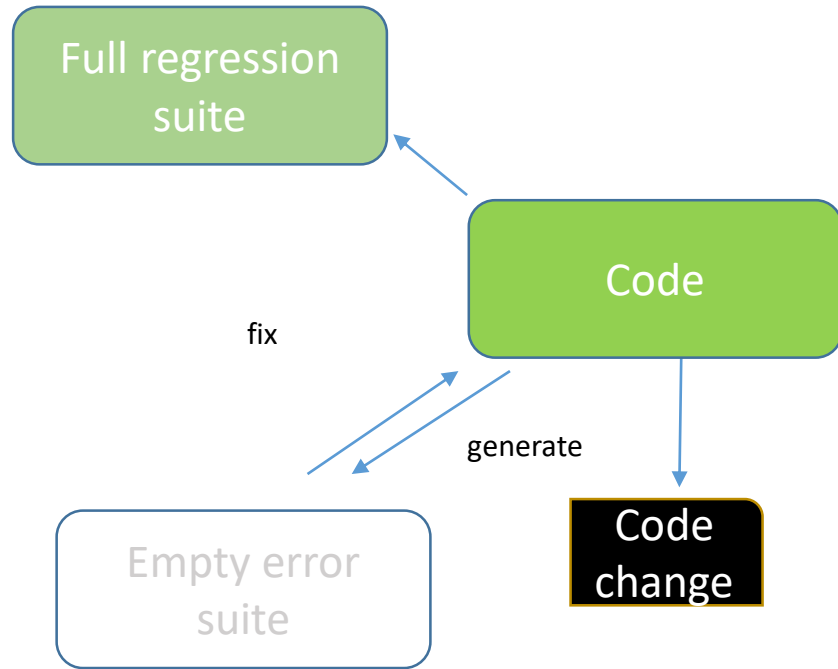
Regression



Code hasn't NPE,
StackOverflows and so on

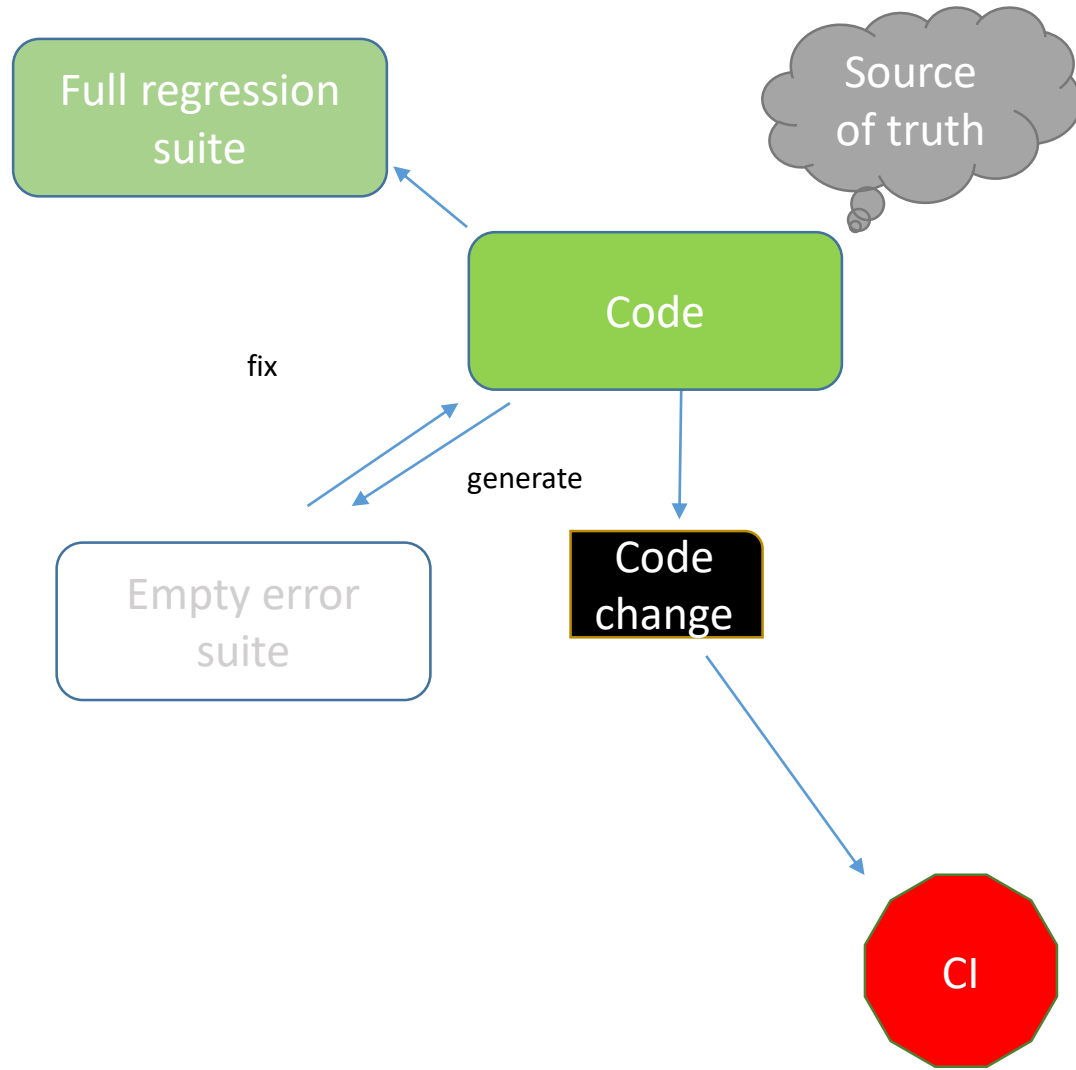
If developer commit change
nobody will notice bug until
it happens on production

Regression suite



Code hasn't NPE,
StackOverflows and so on

Regression suite



Code hasn't NPE,
StackOverflows and so on

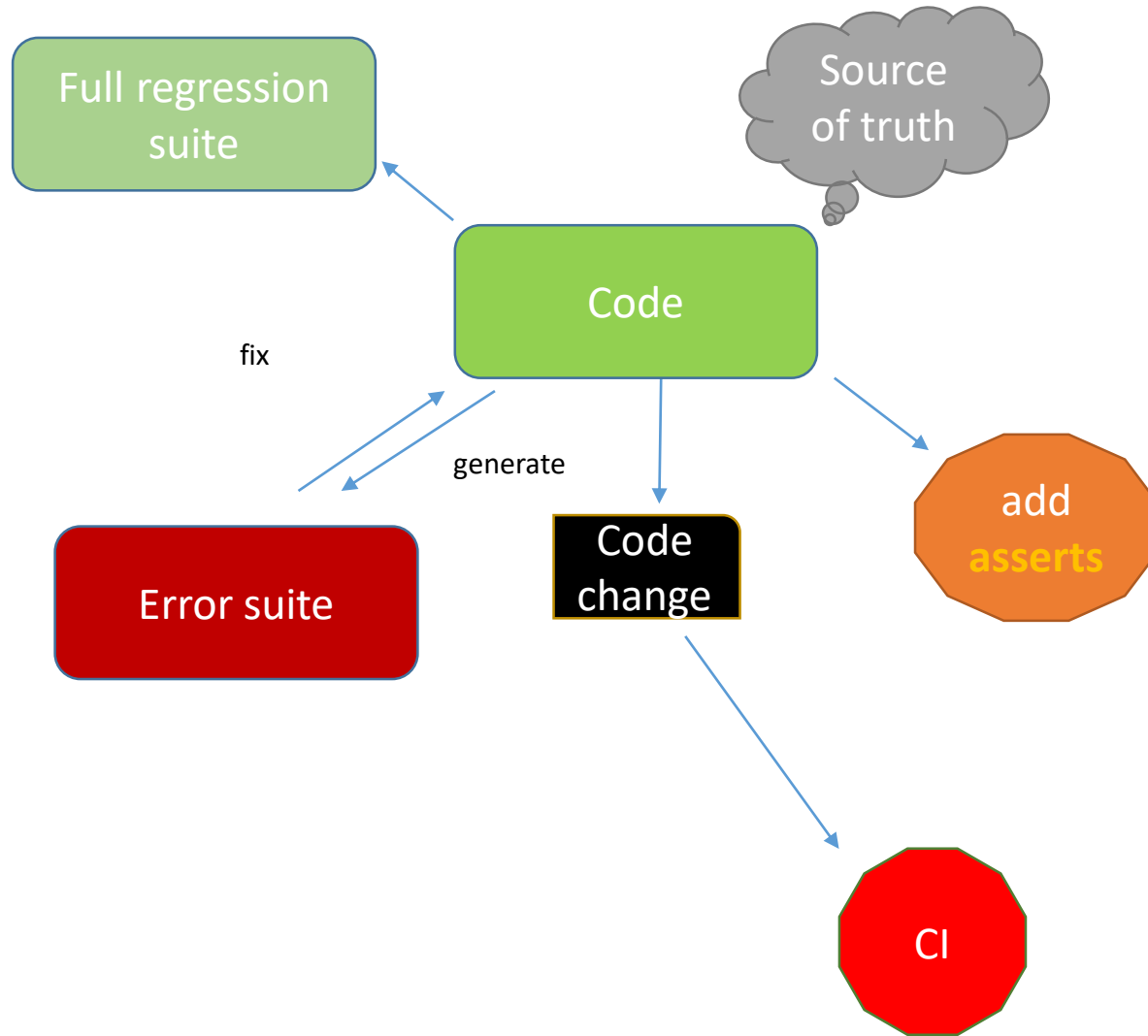
Now behavior is fixated.

Red CI status means one of two things:

- Code change breaks correct behavior
- Initial code behavior wasn't correct

Anyway it's *easy to localize* problem

Specification



Code hasn't NPE,
StackOverflows and so on

Now behavior is fixated.

Red CI status means one of two things:

- Code change breaks correct behavior
- Initial code behavior wasn't correct

Anyway it's easy to localize problem

Code is tested against
specification formalized by
asserts

LIVE DEMO

UnitTestBot

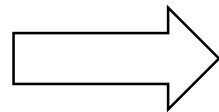


UnitTestBot

Own java symbolic engine

Soot

Z3



UTBot Java/Kt

Python/Go/JS

Own **universal** Symbolic engine

JCDB

KSMT

<http://utbot.org>

What are the key takeaways?

- **[Episode I, Survey]** – Now you know the motivation, history and layout of program analyzers
- **[Episode II, Theory]** – You can write your own simple symbolic execution engine (in theory 😊)
- **[Episode III, Practice]** – Try KSMT or Z3 now to encode and solve any combinatorial problem
- **[Epilogue, Product]** – If you convinced, you can apply program analyzer for your code right now

Thank you for your attention

- [A Survey of Symbolic Execution Techniques](#) – very good introduction into symbolic execution
- “SAT/SMT by Example” – lots of engineering examples how to use SAT and SMT solver
- <https://theory.stanford.edu/~nikolaj/programmingz3.html> - Programming Z3
- <https://github.com/UnitTestBot/ksmt> - Kotlin API for different SMT solvers
- <http://soot-oss.github.io/soot/> - Soot framework for building statics analysis
- <https://github.com/UnitTestBot/jcdb> - Java Compilation Database
- <https://github.com/SymbolicPathFinder/jpf-symbc> - Mature symbolic execution framework for Java
- <https://www.utbot.org/> – Error detection and test generation

Dmitry Ivanov, *Huawei Saint Petersburg Research Center*, korifey@gmail.com, @korifey_ad

Dmitry Mordvinov, *Huawei Saint Petersburg Research Center*, mordvinov.dmitry@gmail.com