Program

Week	Lecture, Lecture hall Chip		
1	14 Feb	Today, Fuzzing	
2	21 Feb	No lecture	Holiday Twente
3	28 Feb	Test Case Generation	
4	7 Mar	Concolic Execution	
5	14 Mar	Mutation Analysis	
6	21 Mar	State Machine Learning	Deadline Report
7	28 Mar	Model-Based Testing	
8	4 Apr	Binary Analysis	
9	11 Apr	What's next?	Deadline Video

Lectures on Tuesday, 13:45 till 15:45
Office hours Sicco Verwer and Andy Zaidman (online)
Skype: live:9e3207a8ea5fdf16, azaidman
Thursdays 10:00 till 12:00

TODAY:
(CONCRETE & SYMBOLIC) CONCOLIC EXECUTION

From Wikipedia

```
1. void f(int x, int y) {
2.     int z = 2*y;
3.     if (x == 100000) {
4.         if (x < z) {
5.             assert(0); /* error */
6.         }
7.     }
8. }
```

"Simple random testing, trying random values of *x* and *y*, would require an impractically large number of tests to reproduce the failure."

Q. Why?

From Wikipedia

```
1. void f(int x, int y) {
2.     int z = 2*y;
3.     if (
4.
5.
6.
7.     }
8. }
```

AFL finds it in milliseconds

So does EvoSuite….

"Simple random testing, trying random values of *x* and *y*, would require an impractically large number of tests to reproduce the failure."

Q. Why?

From Wikipedia

```
1. void f(int x, int y) {
2.      int z = 2*y;
3.      if (
4.
5.
6.
7.      }
8. }
```

AFL finds it in milliseconds

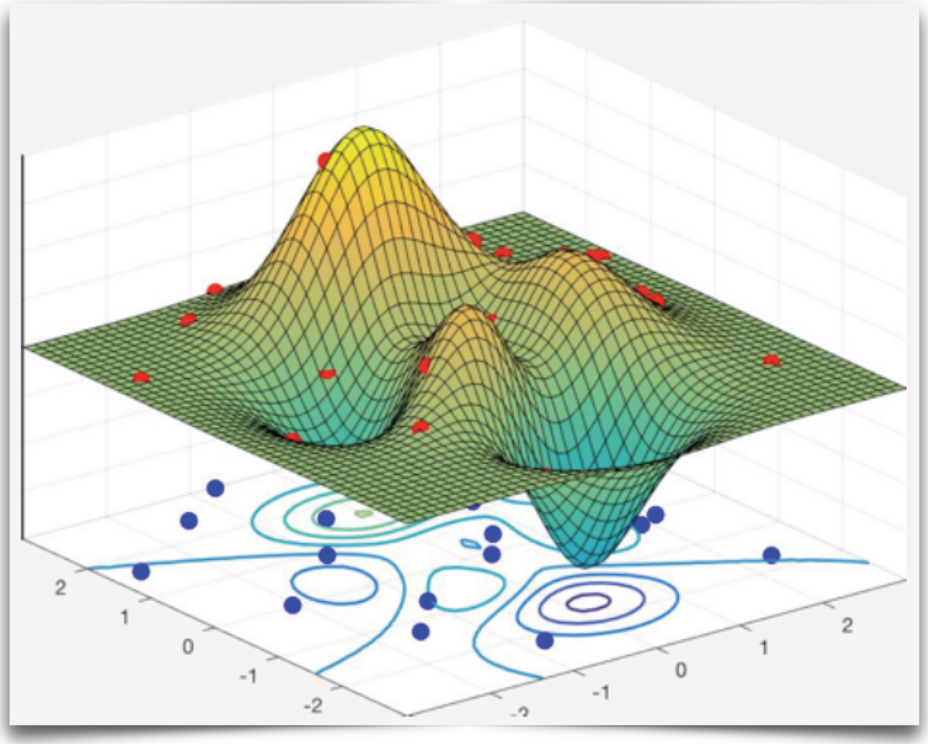
So does

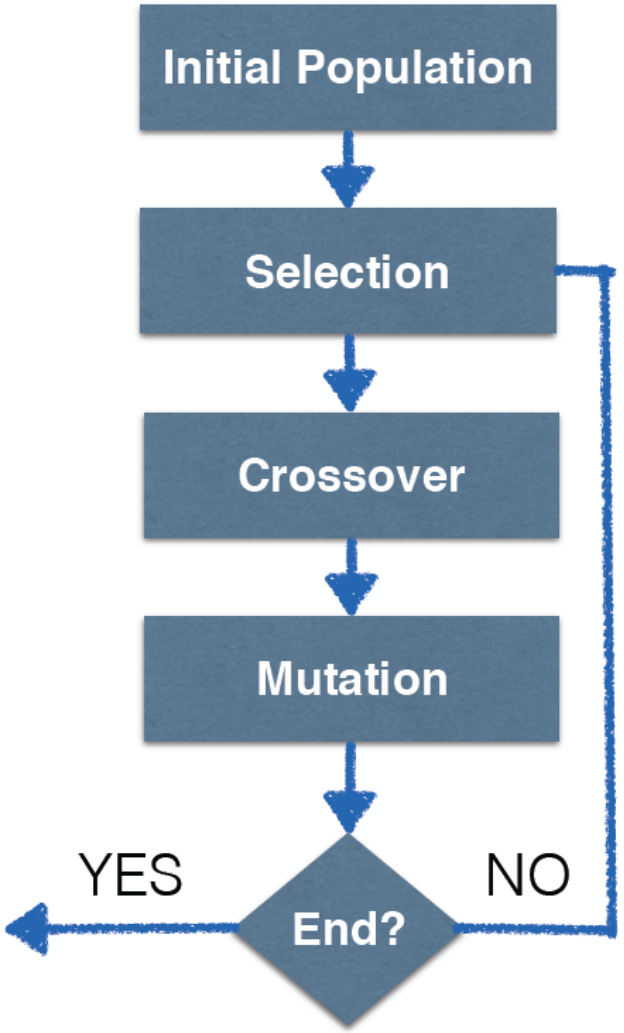
Why?

"Simple random testing, trying random values of *x* and *y*, would require an impractically large number of tests to reproduce the failure."

Q. Why?

Genetic Algorithms

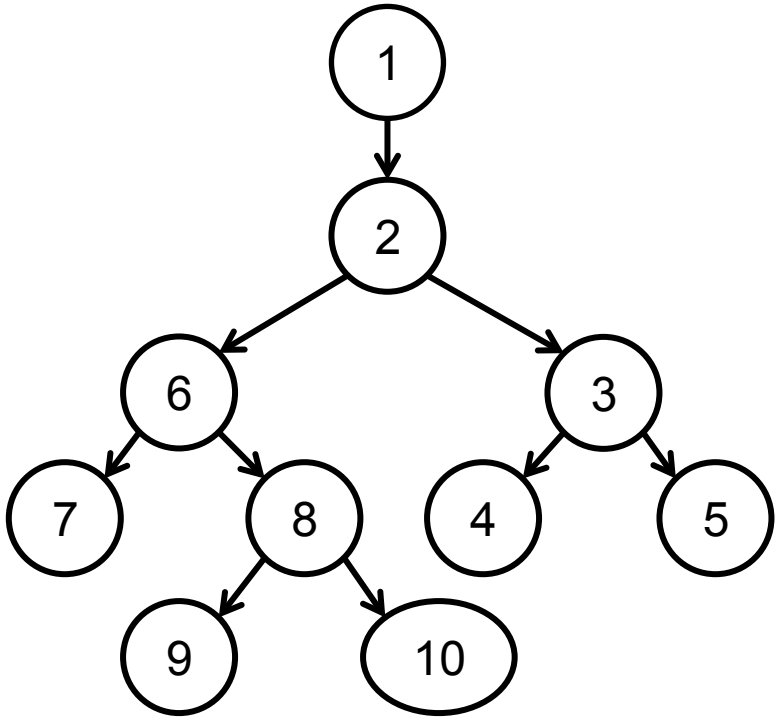




Slides from Annibale Panichella

Genetic Algorithms and Code

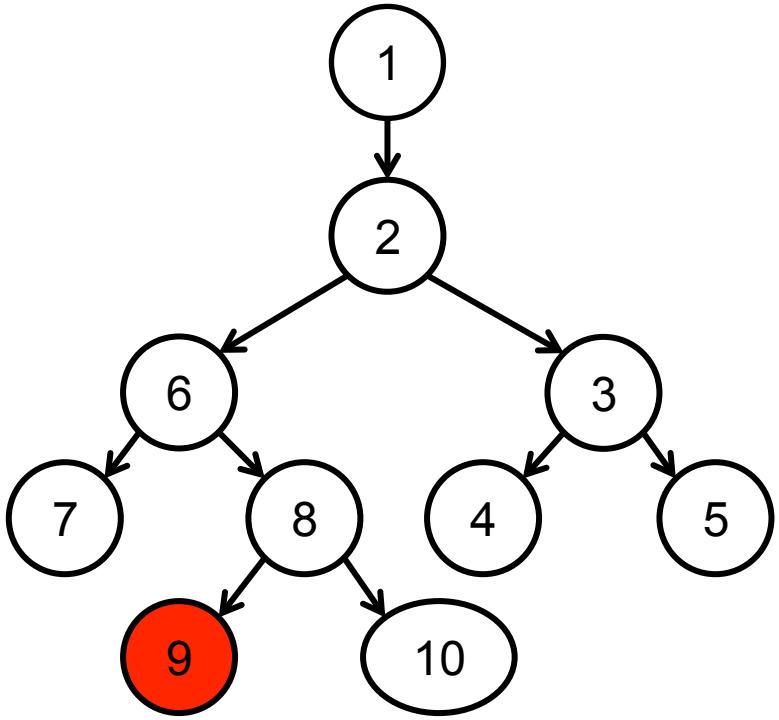
```
class Triangle {
    void computeTriangleType() {
1.    if (isTriangle()){
2.      if (side1 == side2) {
3.        if (side2 == side3)
4.          type = "EQUILATERAL";
          else
5.          type = "ISOSCELES";
          } else {
6.        if (side1 == side3) {
7.          type = "ISOSCELES";
          } else {
8.          if (side2 == side3)
9.            type = “ISOSCELES”;
            else
10.           checkRightAngle();
          }
        }
      }
    }
}
```



Dependency Graph

Fitness: does it reach the target?

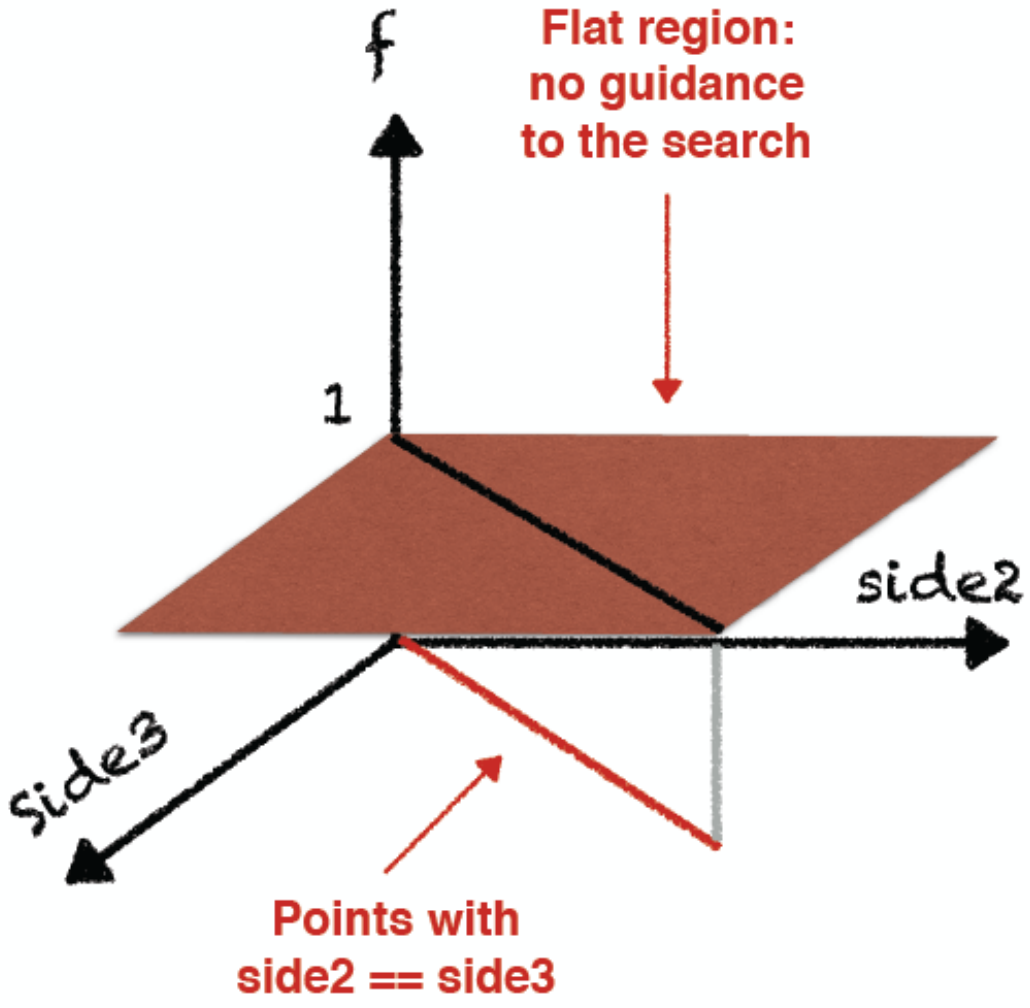
```
class Triangle {
    void computeTriangleType() {
1.    if (isTriangle()){
2.      if (side1 == side2) {
3.        if (side2 == side3)
4.          type = "EQUILATERAL";
          else
5.          type = "ISOSCELES";
          } else {
6.        if (side1 == side3) {
7.          type = "ISOSCELES";
          } else {
8.          if (side2 == side3)
9. target     type = “ISOSCELES”;
            else
10.           checkRightAngle();
          }
        }
      }
    }
}
```



A simple Fitness Function could be:

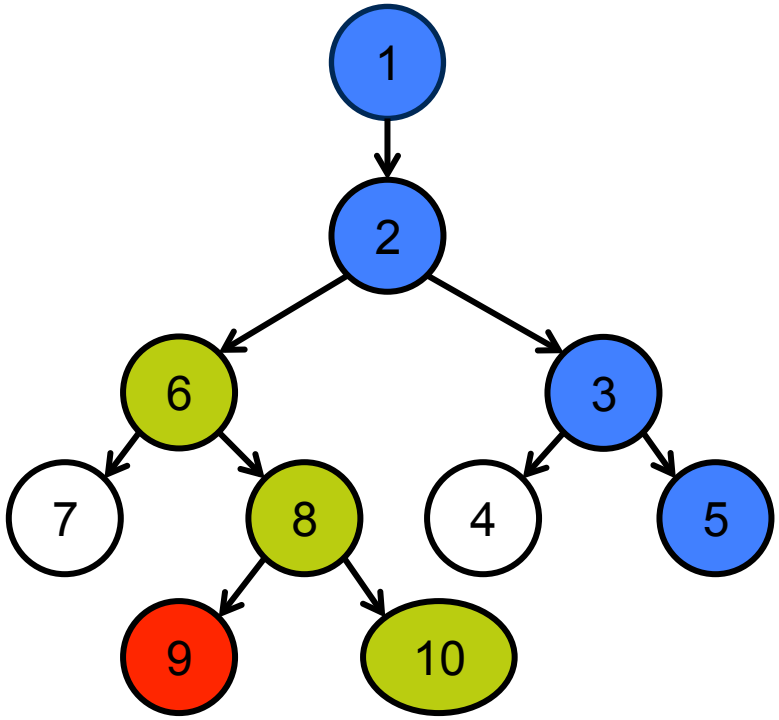
$$f(side2; side3) = \begin{cases} 0 \text{ if side2 == side3} \\ 1 \text{ if side2 != side3} \end{cases}$$

Result: a very flat optimization surface



Measuring distance to target

```
class Triangle {
    void computeTriangleType() {
1.    if (isTriangle()){
2.      if (side1 == side2) {
3.        if (side2 == side3)
4.          type = "EQUILATERAL";
          else
5.          type = "ISOSCELES";
          } else {
6.        if (side1 == side3) {
7.          type = "ISOSCELES";
          } else {
8.          if (side2 == side3)
9. target     type = “ISOSCELES”;
            else
10.           checkRightAngle();
          }
        }
      }
    }
}
```



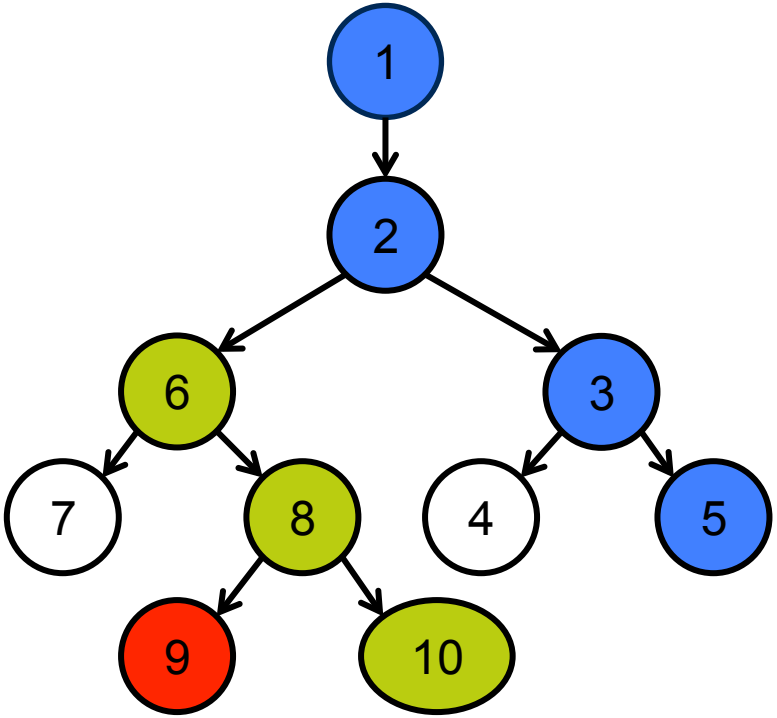
Some traces:

x1 = (2, 2, 3) Path(x1) = <1, 2, 3, 5>
x2 = (2, 3, 5) Path(x2) = <1, 2, 6, 8, 10>

Fitness: Approach Level

The number of nodes between the target node and the execution trace:

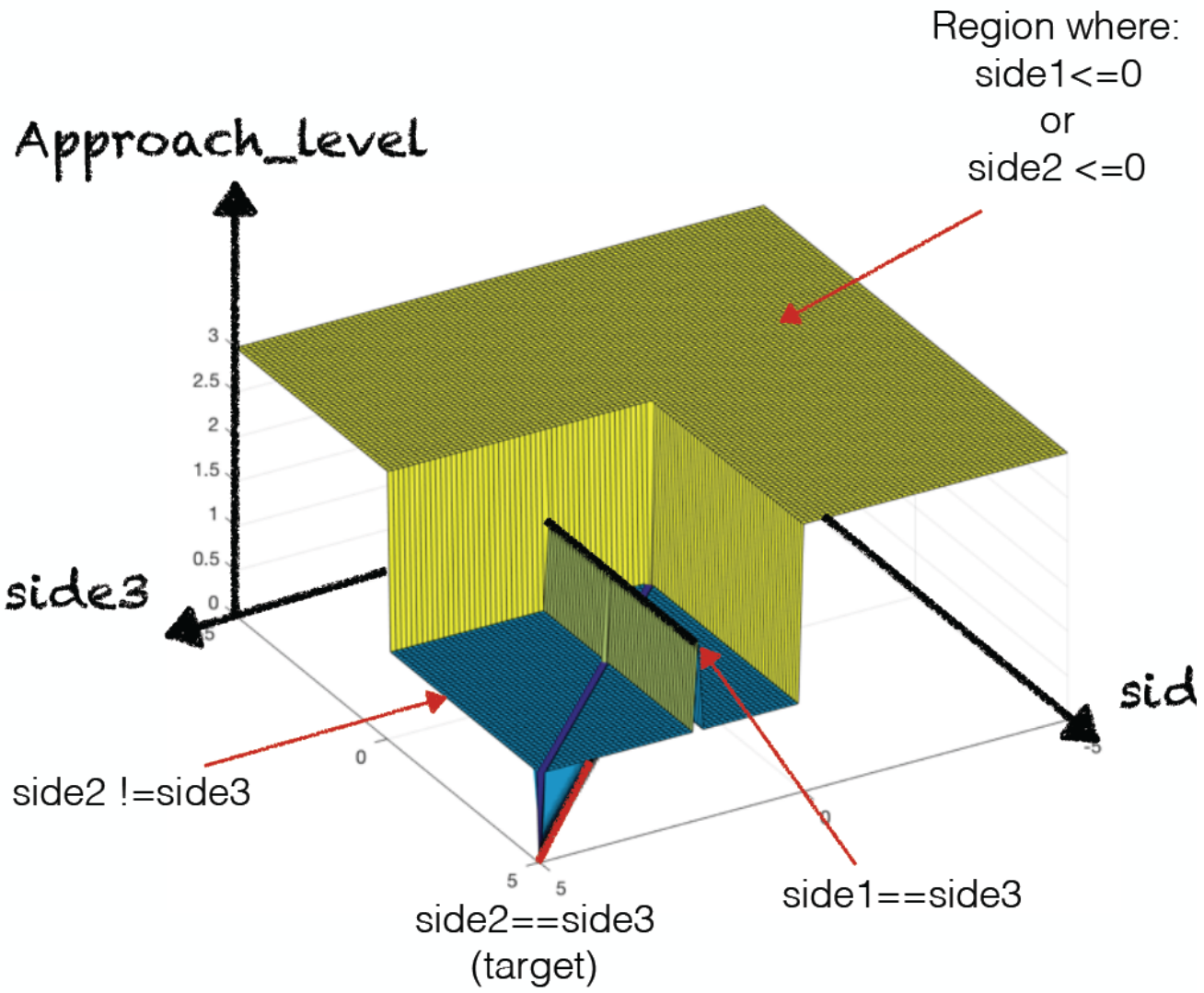
f(x1) = 2
f(x2) = 0



Some traces:

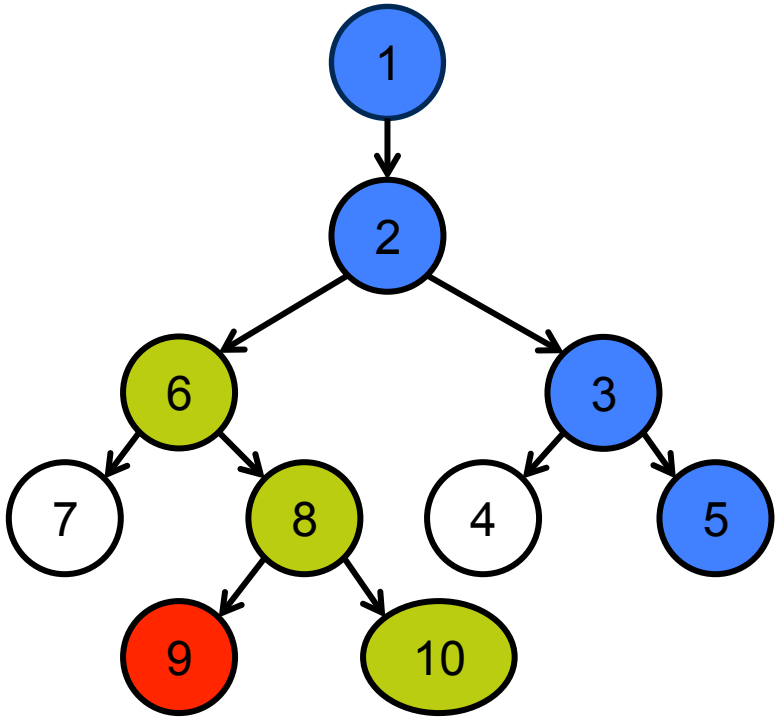
x1 = (2, 2, 3) Path(x1) = <1, 2, 3, 5>
x2 = (2, 3, 5) Path(x2) = <1, 2, 6, 8, 10>

Result: still too flat….



Fitness: which one is closer x2 or x3?

```
class Triangle {
    void computeTriangleType() {
1.    if (isTriangle()){
2.      if (side1 == side2) {
3.        if (side2 == side3)
4.          type = "EQUILATERAL";
          else
5.          type = "ISOSCELES";
          } else {
6.        if (side1 == side3) {
7.          type = "ISOSCELES";
          } else {
8.          if (side2 == side3)
9. target     type = “ISOSCELES”;
            else
10.           checkRightAngle();
          }
        }
      }
    }
}
```



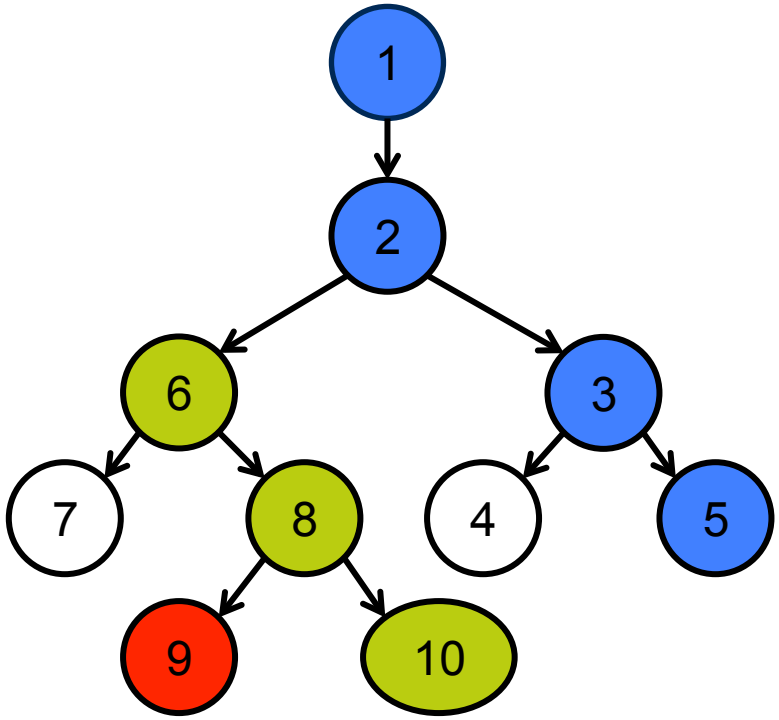
Some traces:

x1 = (2, 2, 3) Path(x1) = <1, 2, 3, 5>
x2 = (2, 3, 5) Path(x2) = <1, 2, 6, 8, 10>
x3 = (2, 3, 10) Path(x3) = <1, 2, 6, 8, 10>

Fitness: which one is closer x2 or x3?

```
class Triangle {
    void computeTriangleType() {
1.    if (isTriangle()){
2.      if (side1 == side2) {
3.        if (side2 == side3)
4.          type = "EQUILATERAL";
5.
6.
7.
8.          if (side2 == side3)
9. target     type = "ISOSCELES";
            else
10.           checkRightAngle();
          }
        }
      }
    }
}
```

line 8:
x2: if 3 == 5
x3: if 3 == 10



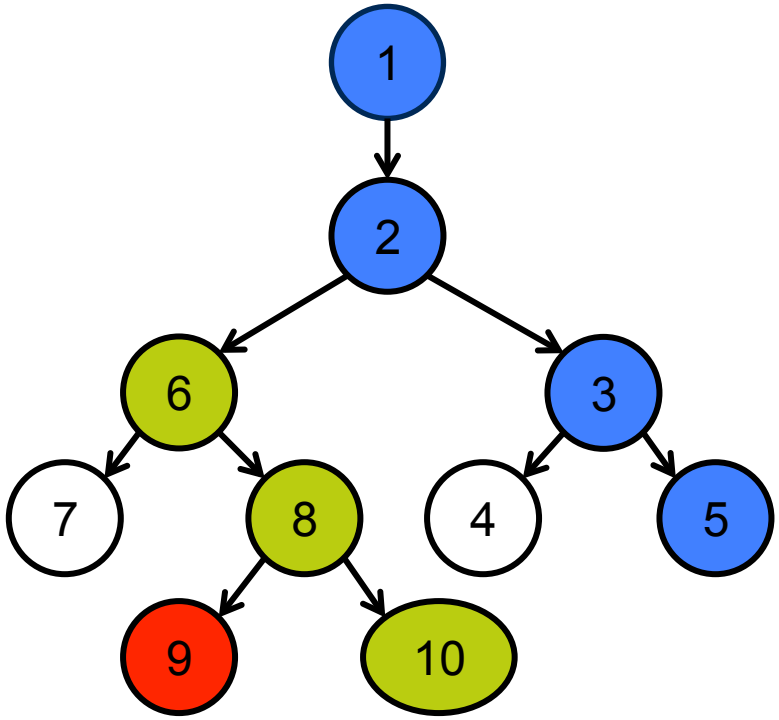
Some traces:

x1 = (2, 2, 3) Path(x1) = <1, 2, 3, 5>
x2 = (2, 3, 5) Path(x2) = <1, 2, 6, 8, 10>
x3 = (2, 3, 10) Path(x3) = <1, 2, 6, 8, 10>

Fitness: which one is closer x2 or x3?

```
class Triangle {
    void computeTriangleType() {
1.    if (isTriangle()){
2.      if (side1 == side2) {
3.        if (side2 == side3)
4.          type = "EQUILATERAL";
5.        else
6.          type = "SCALENE";
7.      }
8.        if (side2 == side3)
9.  target    type = "ISOSCELES";
          else
10.         checkRightAngle();
        }
      }
    }
}
```

branch distance:
f(x2) = abs(3-5) = 2
f(x3) = abs(3-10) = 7



Some traces:

x1 = (2, 2, 3) Path(x1) = <1, 2, 3, 5>
x2 = (2, 3, 5) Path(x2) = <1, 2, 6, 8, 10>
x3 = (2, 3, 10) Path(x3) = <1, 2, 6, 8, 10>

Normalized branch distance

- For numerical and Boolean variables

Condition c = atomic predicate	Distance BD(c) = d / (d + 1)
a	d = {0 if a == true; K otherwise}
!a	d = {K if a == true; 0 otherwise}
a == b	d = {0 if a == b; abs(a - b) + K otherwise}
a != b	d = {0 if a != b; K otherwise}
a < b	d = {0 if a < b; a - b + K otherwise}
a <= b	d = {0 if a <= b; a - b + K otherwise}
a > b	d = {0 if a > b; b - a + K otherwise}
a >= b	d = {0 if a >= b; b - a + K otherwise}

Normalized branch distance

- For Strings

Condition c = atomic predicate	Distance BD(c) = d / (d + 1)
a == b	d = {0 if a == b; edit_dist(a, b) + K otherwise}
a != b	d = {0 if a != b; K otherwise}
a < b	d = {0 if a < b; a[j] - b[j] + K otherwise}
a <= b	d = {0 if a <= b; a[j] - b[j] + K otherwise}
a > b	d = {0 if a > b; b[j] - a[j] + K otherwise}
a >= b	d = {0 if a >= b; b[j] - a[j] + K otherwise}

- j is the position of the first different character such that a[j]! =b[j], while a[i] == b[i] for i<j (a[j]-b[j]) is set to zero if a==b)

Normalized branch distance

- For Composite Predicates

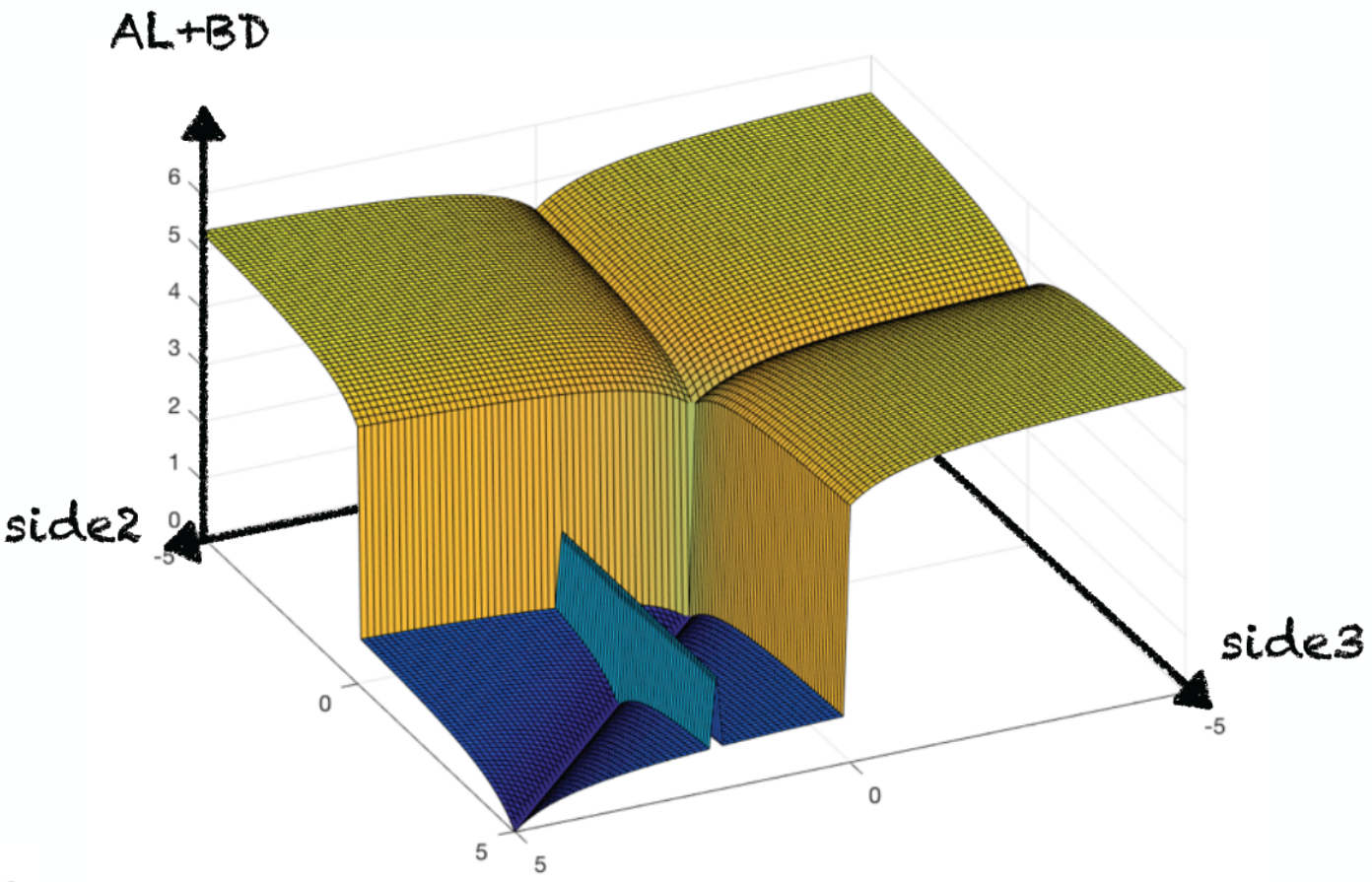
Condition c = composite predicate	Distance BD(c) = d / (d + 1)
!p	Negation is propagated inside p
p & q	d = d(p) + d(q)
p | q	d = min(d(p), d(q))
p XOR q = p & !q | !p & q	d = min(d(p)+d(!q), d(!p)+d(q))

- The final fitness is a combination of Approach Level and Branch Distance:

$$f(x) = approach\_level(x,t) + branch\_distance(x,t)$$

where t is the target

Result: a nice smooth optimization surface!



EvoSuite and AFL

- EvoSuite computes distances using such a fitness function
  - *Future generations are guaranteed to get closer to the target*
- What AFL does exactly is not very clear, but see:
- http://lcamtuf.blogspot.nl/2014/08/binary-fuzzing-strategies-what-works.html
  - Walking bit/byte flips, simple arithmetic, ..
  - *But, form http://lcamtuf.coredump.cx/afl/technical_details.txt:*
    - *"AFL generally does not try to reason about the relationship between specific mutations and program states; the fuzzing steps are nominally blind, and are guided only by the evolutionary design of the input queue"*

EvoSuite and AFL

- EvoSuite computes distances using such a fitness function
  - *Future generations are guaranteed to get closer to the target*
- What AFL does exactly is not very clear, but see:
- htt
  v

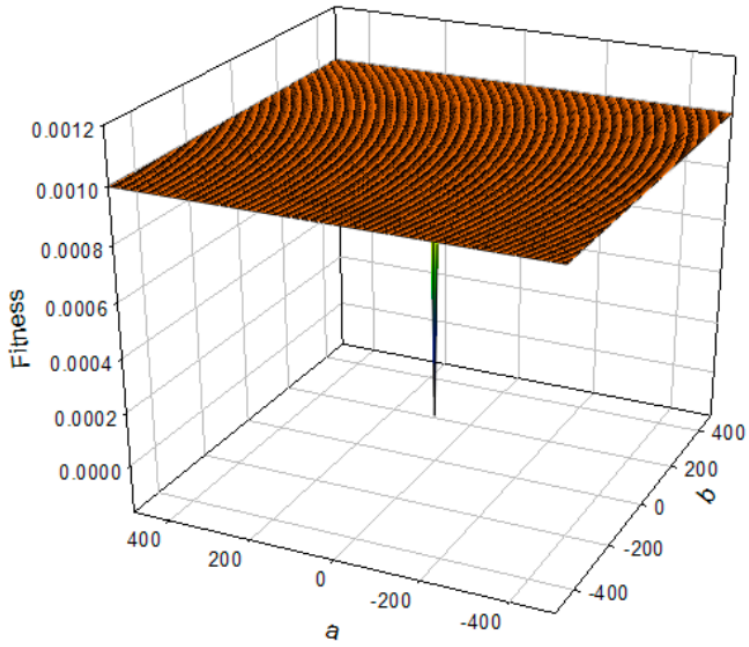
Challenge:

Find small code samples (+- 20 lines) that takes amazingly long to Fuzz/Test!

Code for AFL/EvoSuite is on Git

One famous example: *the flag problem*

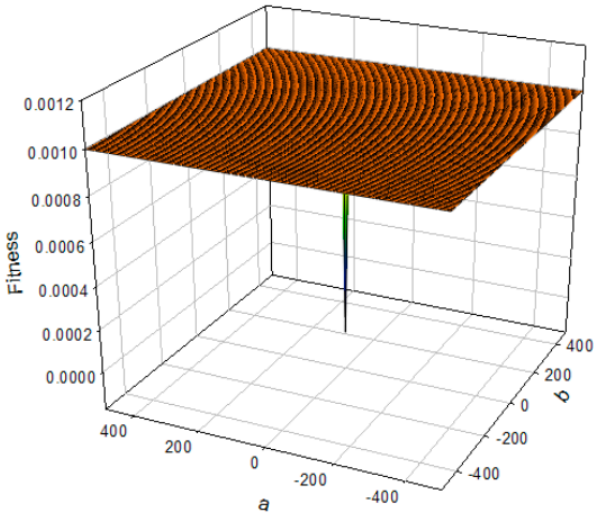
```
void testme(int a, int b, ....)
{
    ...

    flag = (a == 0 && b == 0);

    if (flag) {
        ....
    }
}
```

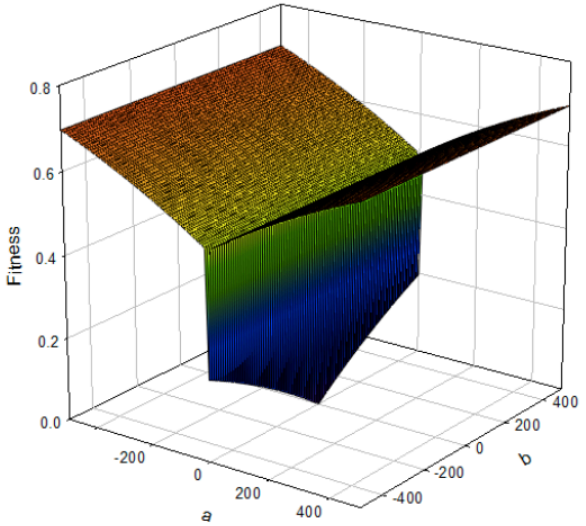


With a fix, but does this always work?

```
void testme(int a, int b, ....)
{
    ...

    flag = (a == 0 && b == 0);

    if (flag) {
        ....
    }
}
```

```
void testme(int a, int b, ....)
{
    ...

    flag = (a == 0 && b == 0);

    if (a == 0 && b == 0) {
        ....
    }
}
```





TU Delft

TODAY:
(CONCRETE & SYMBOLIC) CONCOLIC EXECUTION

Satisfiability & Validity

$p \vee q \Rightarrow q \vee p$

$p \vee q \Rightarrow q$

$p \wedge \neg q \ \wedge (\neg p \vee q)$

ϕ	A	B	$\neg A$	$A \vee B$	$A \wedge \neg A$	$A \Rightarrow B$	$A \Rightarrow (B \vee A)$
$\mathcal{M}_1(\phi)$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\mathcal{M}_2(\phi)$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$
$\mathcal{M}_3(\phi)$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\mathcal{M}_4(\phi)$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\top$

Satisfiability & Validity

$p \vee q \Rightarrow q \vee p$ VALID

$p \vee q \Rightarrow q$ SATISFIABLE

$p \wedge \neg q \wedge (\neg p \vee q)$ UNSATISFIABLE

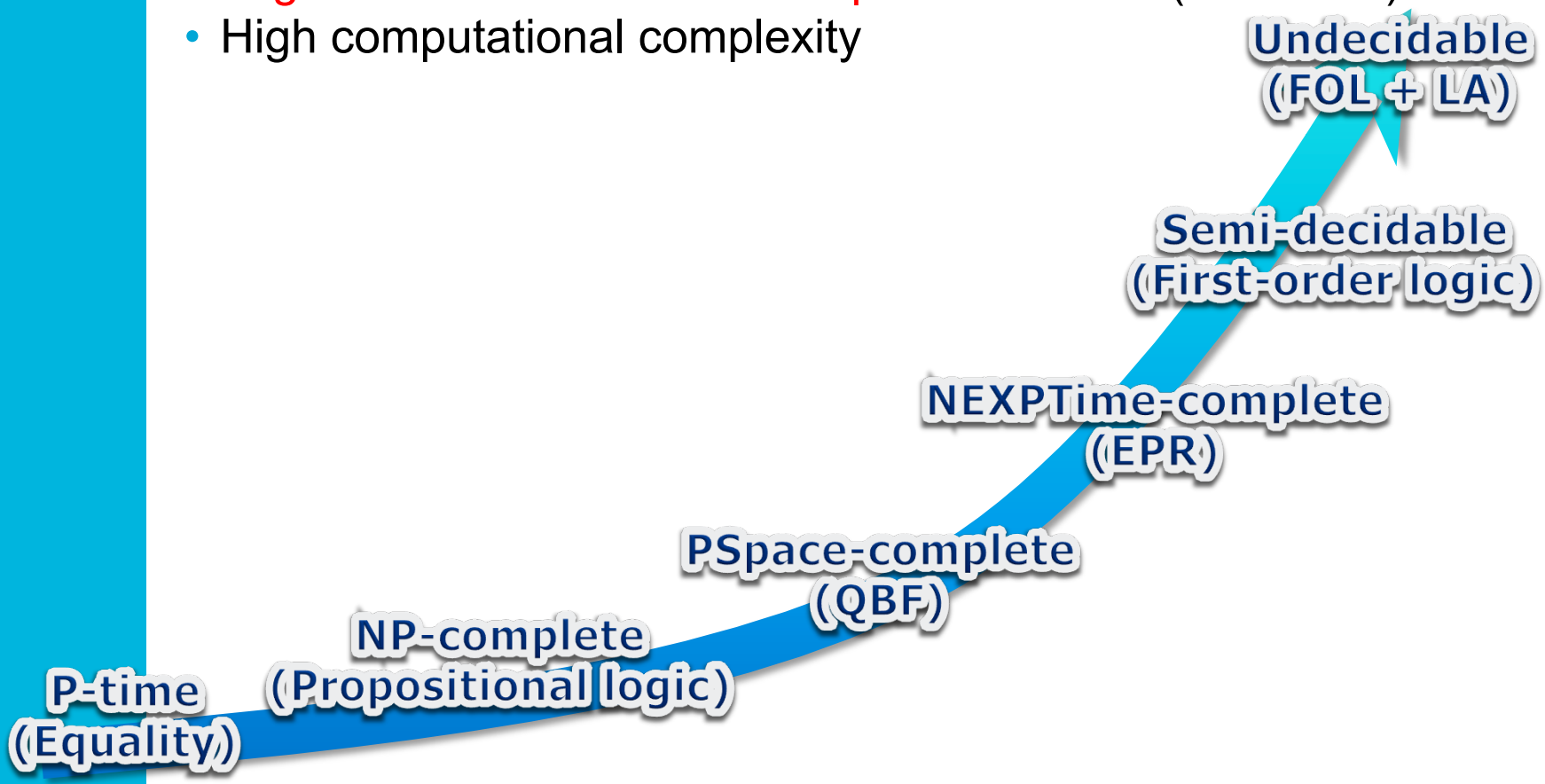
ϕ	A	B	$\neg A$	$A \vee B$	$A \wedge \neg A$	$A \Rightarrow B$	$A \Rightarrow (B \vee A)$
$\mathcal{M}_1(\phi)$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\mathcal{M}_2(\phi)$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$
$\mathcal{M}_3(\phi)$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\mathcal{M}_4(\phi)$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\top$

SAT solvers

- Decades of experience in solving NP-hard problems by translating them to Satisfiability
  - *problem has solution iff formula is satisfiable*
- Yearly competitions pushing the state-of-the-art
  - highly optimized
  - competitive for many problems
  - very general: *every NP-hard problem can be translated to SAT*
  - *(Interested in MSc project? Let me know!)*
- Recently used to solve long-standing open problems:
  - http://www.nature.com/news/two-hundred-terabyte-maths-proof-is-largest-ever-1.19990
- Simply very fast at solving formulas in propositional logic

Symbolic Reasoning

- Logic is “The Calculus of Computer Science” (Z. Manna)
- High computational complexity



Slides from Leonardo de Moura

A CNF formula

$$
\begin{aligned}
\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\
& (\neg b \vee \neg d \vee \neg e) \wedge \\
& (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\
& (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)
\end{aligned}
$$

SAT solver inner workings --- DPLL

- Standard backtrack search
- DPLL(F) :
  - Apply unit propagation
  - If conflict identified, return UNSAT
  - Apply the pure literal rule
  - If F is satisfied (empty), return SAT
  - Select decision variable x
    - If DPLL($F \wedge x$)=SAT return SAT
    - return DPLL($F \wedge \neg x$)

Unit Propagation

(Davis–Putnam–Logemann–Loveland)

DPLL = Unit resolution + Split rule.

$$\frac{\Gamma}{\Gamma, p \mid \Gamma, \neg p}\, split \qquad p \text{ and } \neg p \text{ are not in } \Gamma.$$

$$\frac{C \vee \bar{l}, l}{C, l}\, unit$$

Used in the most efficient SAT solvers.

Pure Literals

A literal is pure if only occurs positively or negatively.

Example :

$\varphi = (\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_2) \wedge (x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$

$\neg x_1$ and x_3 are pure literals

Pure literal rule :

Clauses containing pure literals can be removed from the formula (i.e. just satisfy those pure literals)

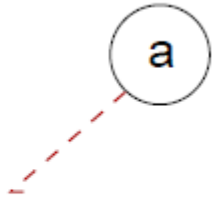
$$\varphi_{\neg x_1, x_3} = (x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$$

Preserve satisfiability, not logical equivalency !

DPLL (example)

Guess

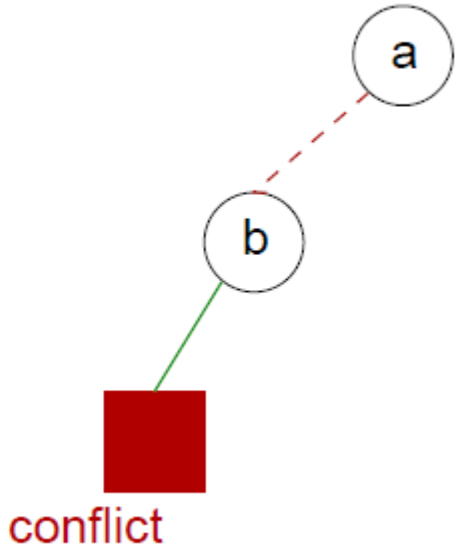
$$\varphi = (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge (\neg b \vee \neg d \vee \neg e) \wedge (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)$$



DPLL (example)

Deduce

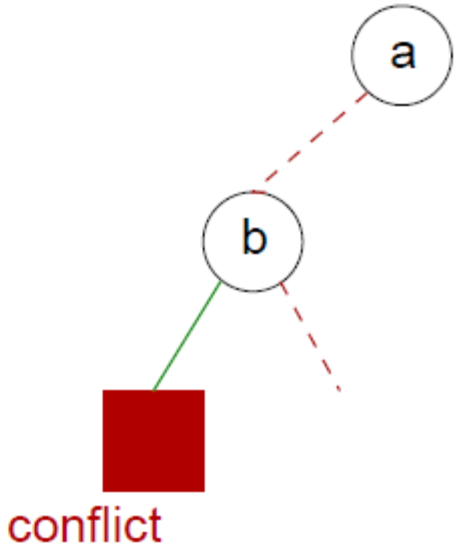
$$\varphi = (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge (\neg b \vee \neg d \vee \neg e) \wedge (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)$$



DPLL (example)

Backtrack

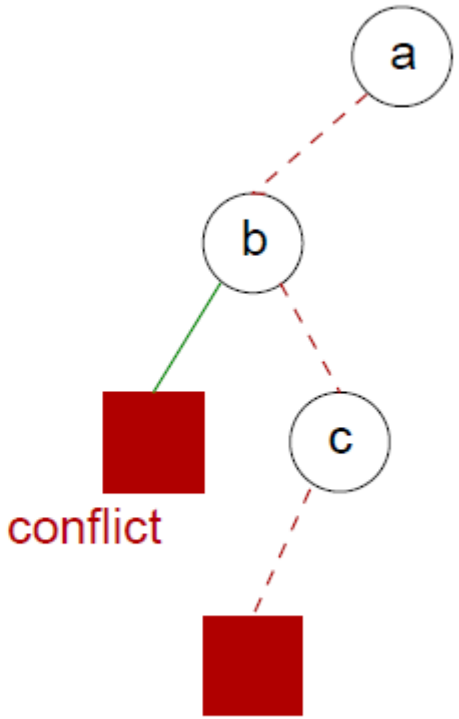
$$\varphi = (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge (\neg b \vee \neg d \vee \neg e) \wedge (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)$$



DPLL (example)

Deduce

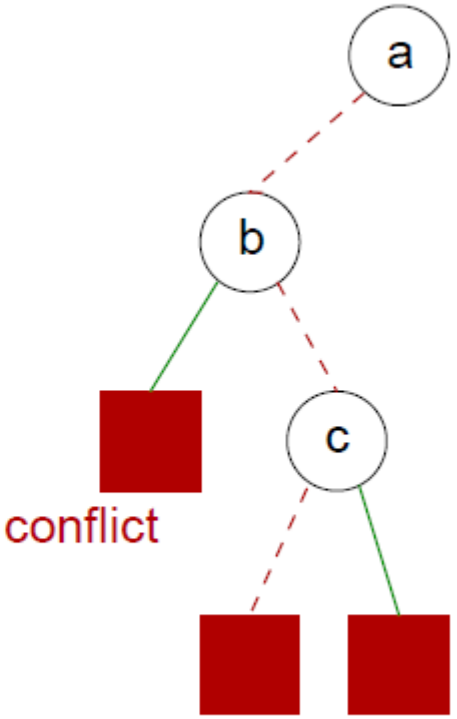
$$\varphi = (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge (\neg b \vee \neg d \vee \neg e) \wedge (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)$$



DPLL (example)

Deduce

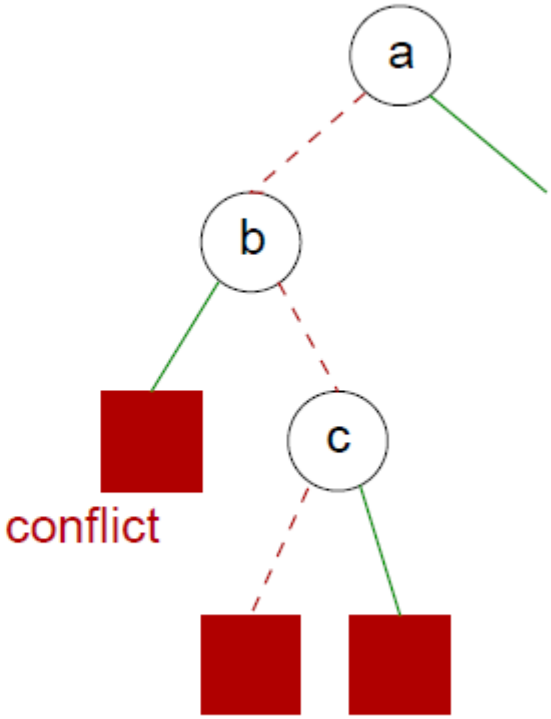
$$\varphi = (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge (\neg b \vee \neg d \vee \neg e) \wedge (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)$$



DPLL (example)

Backtrack

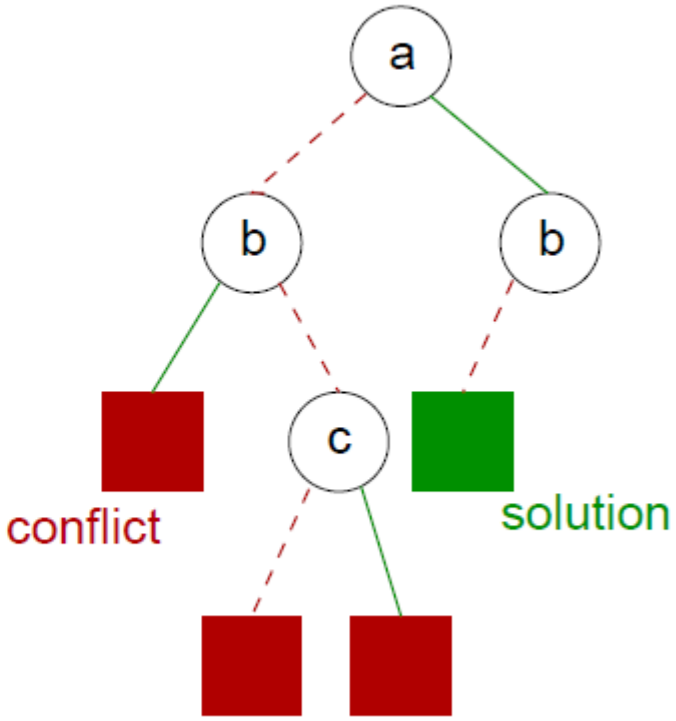
$$\varphi = (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge (\neg b \vee \neg d \vee \neg e) \wedge (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)$$



DPLL (example)

Guess

$$\varphi = (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge (\neg b \vee \neg d \vee \neg e) \wedge (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)$$



Modern DPLL

- Do guess-deduce-backtrack very efficiently
  - all represented using bits and binary operators
- In addition
  - Efficient indexing (two-watch literal)
  - Non-chronological backtracking (backjumping)
  - Lemma learning
- Google if interested…building a SAT solver from scratch is unlikely competitive with the state-of-the-art:
  - They improve every year…
  - See SAT Live!: http://www.satlive.org/solvers/

SAT solvers 2

- ..
- Simply very fast at solving formulas in propositional logic
  - In contrast to GAs and local search, SAT solvers are complete!
- But not so good at modeling numeric variables:
  - integers, floats, (non-)linear arithmetic, …
- Possible using Boolean representation:
  - 0 = 000, 1 = 001 2 = 010, 3 = 011, ..
- But representing arithmetic in logic is slow...
- So use a *seperate solver* for such operations!
  - or represent the problem as an integer program...
  - or use other approaches such as genetic algorithms...

Satisfiability Modulo Theories (SMT)

Is formula *F* satisfiable modulo theory *T* ?

SMT solvers have specialized algorithms for *T*

SAT + Theory solvers

Basic Idea

$x \geq 0, y = x + 1, (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1, p_2, (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0), p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2), p_4 \equiv (y < 1)$

SAT + Theory solvers

Basic Idea

$x \geq 0, y = x + 1, (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

$p_1, p_2, (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0), p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2), p_4 \equiv (y < 1)$

SAT Solver

SAT + Theory solvers

Basic Idea

$x \geq 0, y = x + 1, (y > 2 \vee y < 1)$

Abstract (aka "naming" atoms)

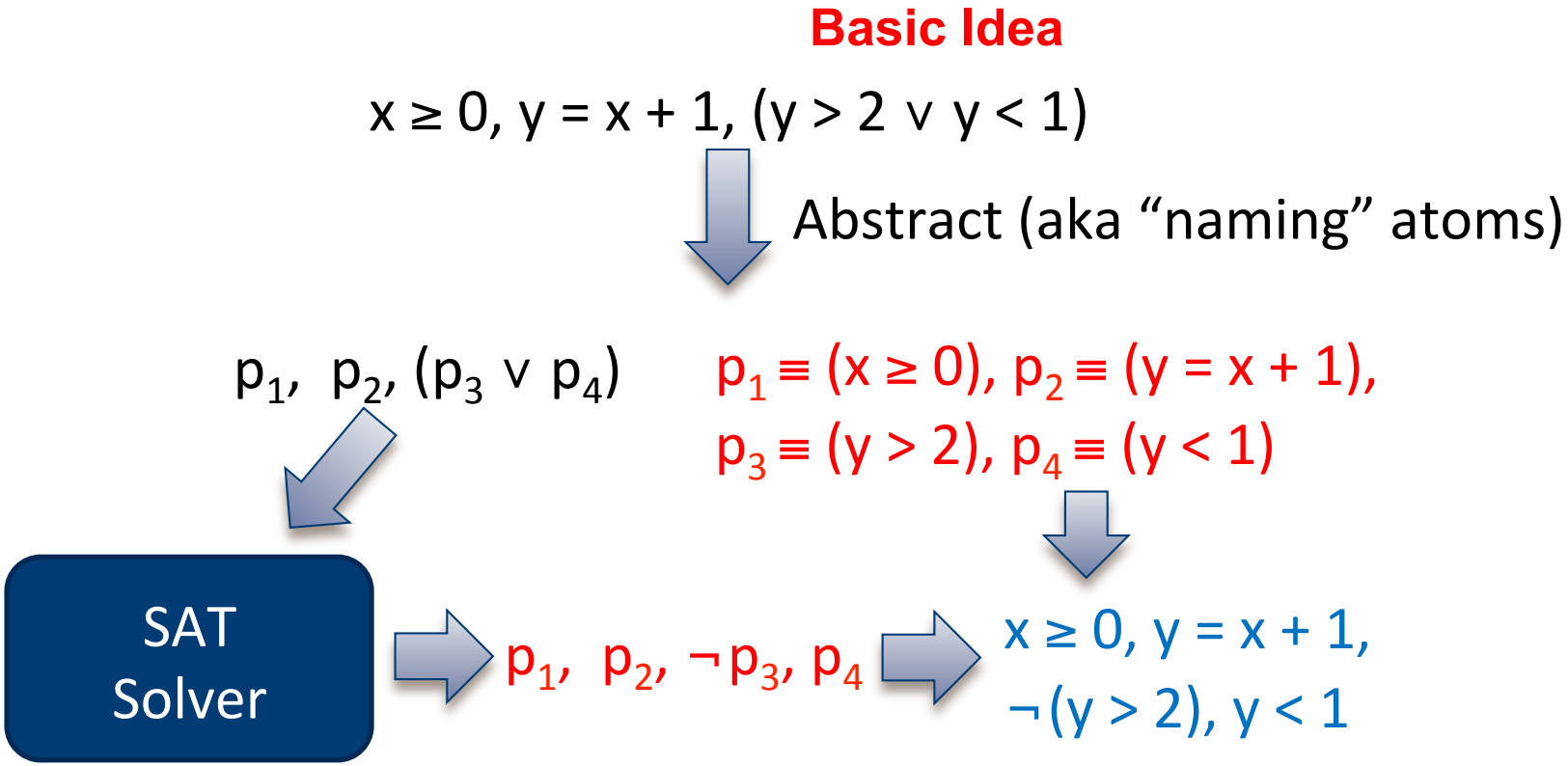
$p_1, p_2, (p_3 \vee p_4)$

$p_1 \equiv (x \geq 0), p_2 \equiv (y = x + 1),$
$p_3 \equiv (y > 2), p_4 \equiv (y < 1)$

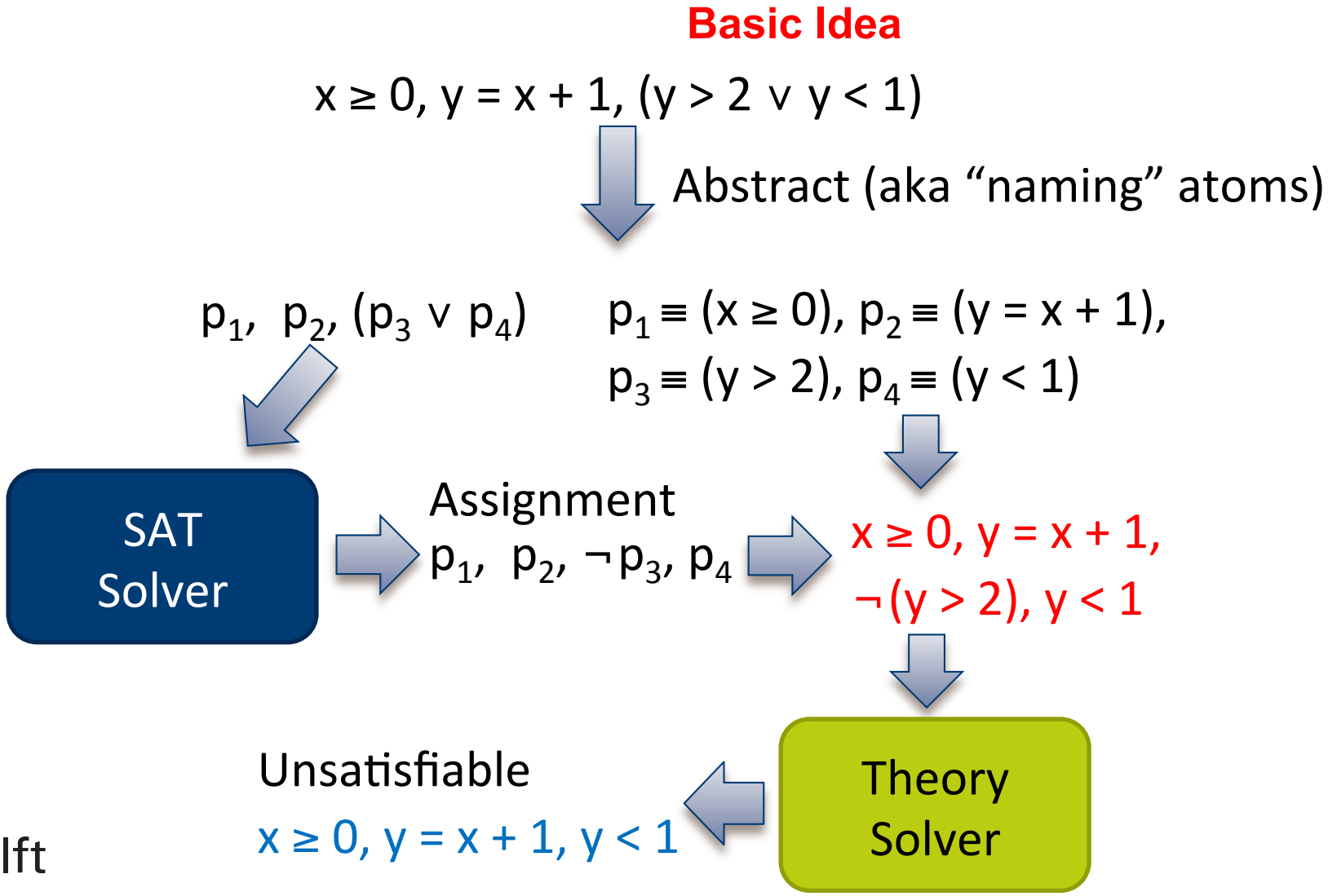
SAT Solver

$p_1, p_2, \neg p_3, p_4$

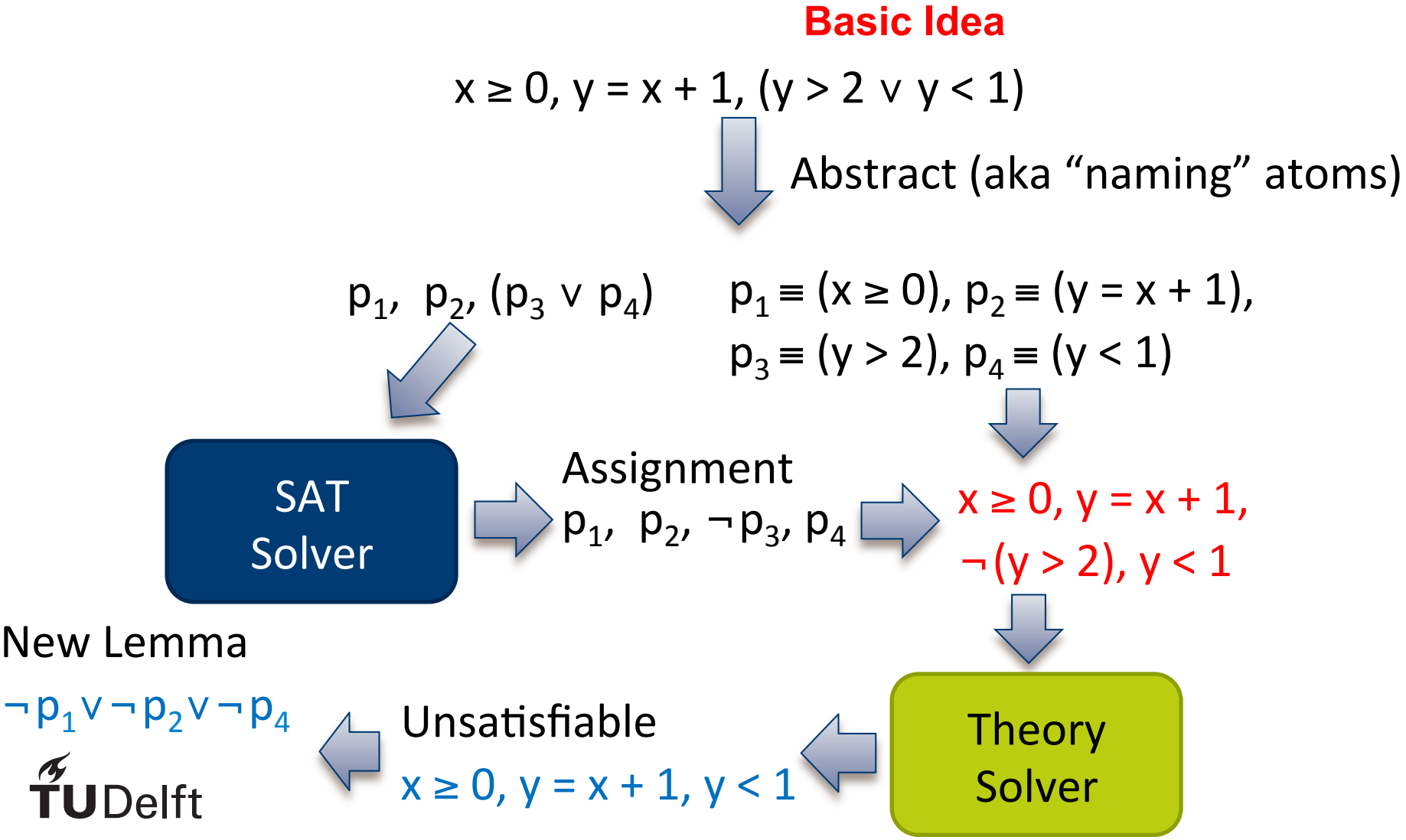
SAT + Theory solvers



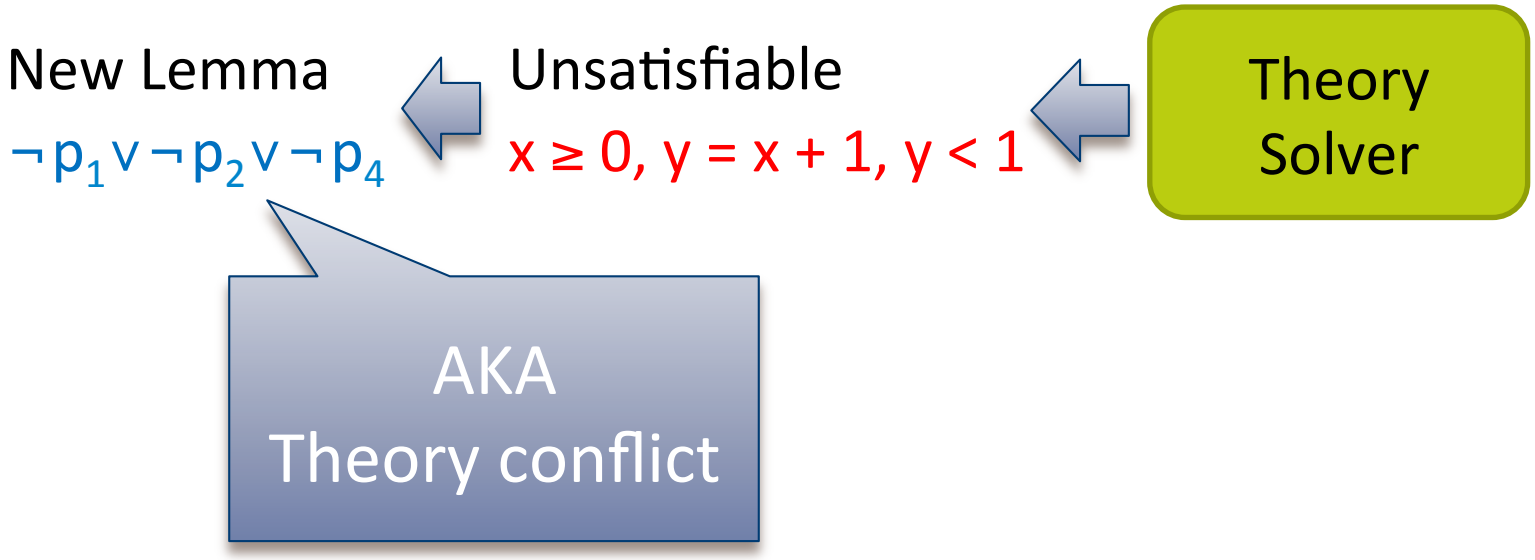
SAT + Theory solvers



SAT + Theory solvers



SAT + Theory solvers



SAT + Theory solvers: Main loop

procedure SmtSolver(F)

  (F_p, M) := Abstract(F)

  loop

   (R, A) := SAT_solver(F_p)

   if R = UNSAT **then return** UNSAT

   S := Concretize(A, M)

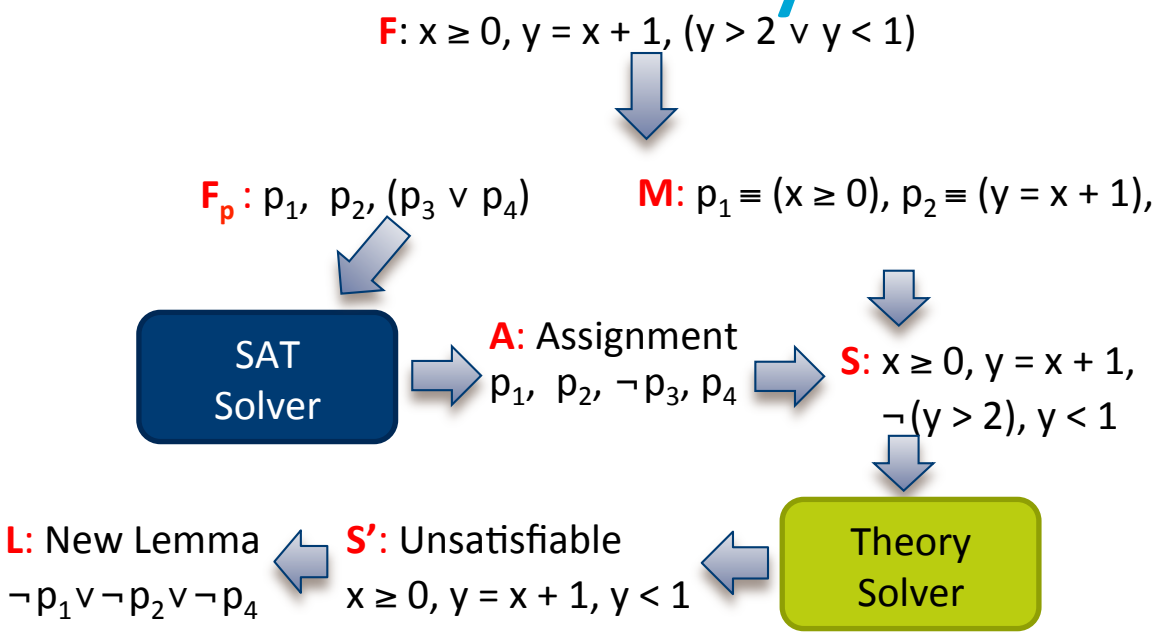
   (R, S') := Theory_solver(S)

   if R = SAT **then return** SAT

   L := New_Lemma(S', M)

   Add L to F_p

SAT + Theory solvers




```
procedure SMT_Solver(F)
    (Fp, M) := Abstract(F)
    loop
     (R, A) := SAT_solver(Fp)
     if R = UNSAT then return UNSAT
     S = Concretize(A, M)
     (R, S') := Theory_solver(S)
     if R = SAT then return SAT
     L := New_Lemma(S, M)
     Add L to Fp
```

How can this be fast?

- SAT solvers are extremely efficient
- The obtained theory S is often easy to solve (not NP-hard)
- and

State-of-the-art SMT solvers implement many improvements.

SAT + Theory solvers

Incrementality

Send the literals to the Theory solver as they are assigned by the SAT solver

$p_1 \equiv (x \geq 0)$, $p_2 \equiv (y = x + 1)$,
$p_3 \equiv (y > 2)$, $p_4 \equiv (y < 1)$, $p_5 \equiv (x < 2)$,

$p_1,\ p_2,\ p_4 \ \mid\ p_1,\ p_2, (p_3 \vee p_4), (p_5 \vee \neg p_4)$

Partial assignment is already Theory inconsistent.

SAT + Theory solvers

Efficient Lemma Generation (computing a small S')

Avoid lemmas containing redundant literals.

$p_1 \equiv (x \geq 0)$, $p_2 \equiv (y = x + 1)$,
$p_3 \equiv (y > 2)$, $p_4 \equiv (y < 1)$, $p_5 \equiv (x < 2)$,

$p_1, p_2, p_3, p_4 \mid p_1, p_2, (p_3 \vee p_4), (p_5 \vee \neg p_4)$

$\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4$

Imprecise Lemma

SAT + Theory solvers

Theory Propagation

$p_1 \equiv (x \geq 0)$, $p_2 \equiv (y = x + 1)$,

$p_3 \equiv (y > 2)$, $p_4 \equiv (y < 1)$, $p_5 \equiv (x < 2)$,

$p_1, p_2 \mid p_1, p_2, (p_3 \vee p_4), (p_5 \vee \neg p_4)$

p_1, p_2 imply $\neg p_4$ by theory propagation

$p_1, p_2, \neg p_4 \mid p_1, p_2, (p_3 \vee p_4), (p_5 \vee \neg p_4)$

From code to test

```
unsigned GCD(x, y) {
  requires(y > 0);
  while (true) {
 unsigned m = x % y;
  if (m == 0) return y;
  x = y;
  y = m;
   }
}
```

Give a trace where the loop is executed twice.

From code to test

```
unsigned GCD(x, y) {
  requires(y > 0);
  while (true) {
 unsigned m = x % y;
  if (m == 0) return y;
  x = y;
  y = m;
   }
}
```

For the Break:

Find such an assignment

You may use KLEE, see Git

Give a trace where the loop is executed twice.

From code to test

```
unsigned GCD(x, y) {
  requires(y > 0);
  while (true) {
 unsigned m = x % y;
  if (m == 0) return y;
  x = y;
  y = m;
   }
}
```

SSA

Static Single Assignment

$(y_0 > 0)$ and
$(m_0 = x_0 \ \% \ y_0)$ and
not $(m_0 = 0)$ and
$(x_1 = y_0)$ and
$(y_1 = m_0)$ and
$(m_1 = x_1 \ \% \ y_1)$ and
$(m_1 = 0)$

Solver

Variable Assignmer

$x_0 = 2$
$y_0 = 4$
$m_0 = 2$
$x_1 = 4$
$y_1 = 2$
$m_1 = 0$

Give a trace where the loop is executed twice.

From code to test

```
unsigned GCD(x, y) {
  requires(y > 0);
  while (true) {
 unsigned m = x % y;
  if (m == 0) return y;
  x = y;
  y = m;
   }
}
```

SSA

Static Single Assignment

$(y_0 > 0)$ and
$(m_0 = x_0 \ \% \ y_0)$ and
not $(m_0 = 0)$ and
$(x_1 = y_0)$ and
$(y_1 = m_0)$ and
$(m_1 = x_1 \ \% \ y_1)$ and
$(m_1 = 0)$

Solver

Variable Assignmer

$x_0 = 2$
$y_0 = 4$
$m_0 = 2$
$x_1 = 4$
$y_1 = 2$
$m_1 = 0$

AKA Path Constraint

Symbolic execution

- Maps code and a target branch/line entirely into sets of constraints that can be solved using, e.g., an SMT solver
- Used to be a static analysis tool:
  - ***code is only interpreted, not executed!***
- Nowadays it is combined with actually running the code via dynamic analysis (concrete execution)
  - ***run, analyze, and iterate***
- Balance concrete and symbolic parts
- Many tools developed, and used to find real bugs:
  - PEX, KLEE, Angr, SAGE, …
  - see https://en.wikipedia.org/wiki/Symbolic_execution

The theory: Difference Logic: $a - b \leq 5$

Very useful in practice!

Most arithmetical constraints in software verification/analysis are in this fragment.

$x := x + 1$

$x_1 = x_0 + 1$

$x_1 - x_0 \leq 1, x_0 - x_1 \leq -1$

Difference Logic

$$
\begin{array}{llll}
z & - & t_{1,1} & \leq & 0 \\
z & - & t_{2,1} & \leq & 0 \\
z & - & t_{3,1} & \leq & 0 \\
t_{3,2} & - & z & \leq & 5 \\
t_{3,1} & - & t_{3,2} & \leq & -2 \\
t_{2,1} & - & t_{3,1} & \leq & -3 \\
t_{1,1} & - & t_{2,1} & \leq & -2
\end{array}
$$

Satisfiable?

Difference Logic

$$
\begin{array}{llll}
z & - & t_{1,1} & \leq 0 \\
z & - & t_{2,1} & \leq 0 \\
z & - & t_{3,1} & \leq 0 \\
t_{3,2} & - & z & \leq 5 \\
t_{3,1} & - & t_{3,2} & \leq -2 \\
t_{2,1} & - & t_{3,1} & \leq -3 \\
t_{1,1} & - & t_{2,1} & \leq -2
\end{array}
$$

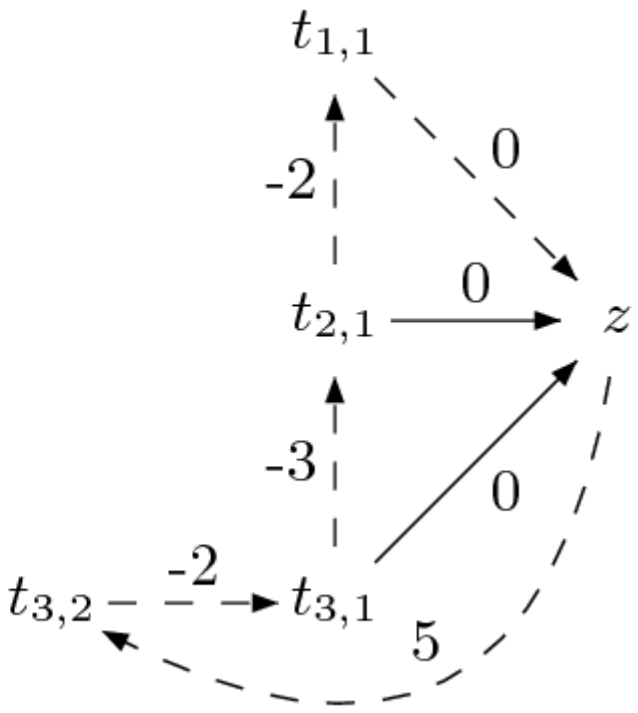
NO!

Difference Logic complexity

Satisfiable if and only if there are no negative cycles!

Algorithms based on Bellman-Ford (O(mn)).

$$
\begin{array}{llcl}
z & - & t_{1,1} & \leq & 0 \\
z & - & t_{2,1} & \leq & 0 \\
z & - & t_{3,1} & \leq & 0 \\
t_{3,2} & - & z & \leq & 5 \\
t_{3,1} & - & t_{3,2} & \leq & -2 \\
t_{2,1} & - & t_{3,1} & \leq & -3 \\
t_{1,1} & - & t_{2,1} & \leq & -2
\end{array}
$$



Z3 – a frequently used solver

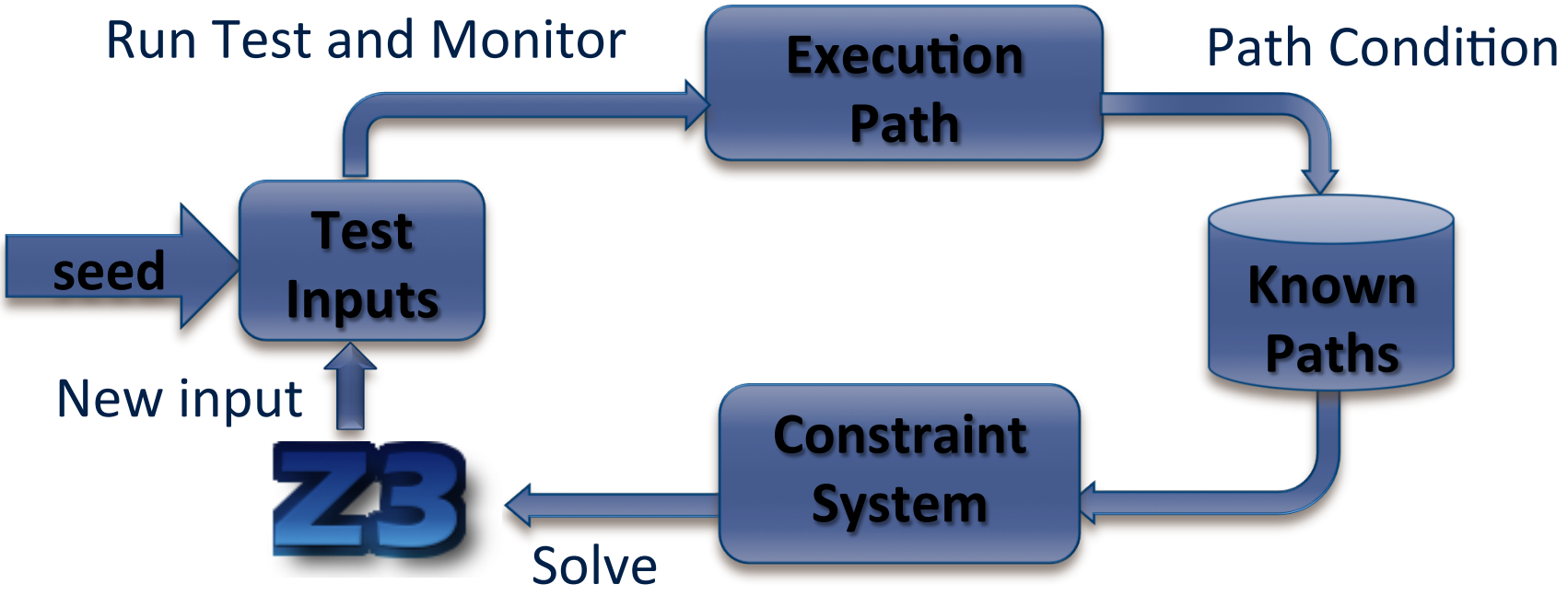
- https://github.com/Z3Prover/z3
- https://github.com/Z3Prover/z3/wiki

- One of the most powerful SMT solvers, developed by Microsoft, check out http://rise4fun.com/

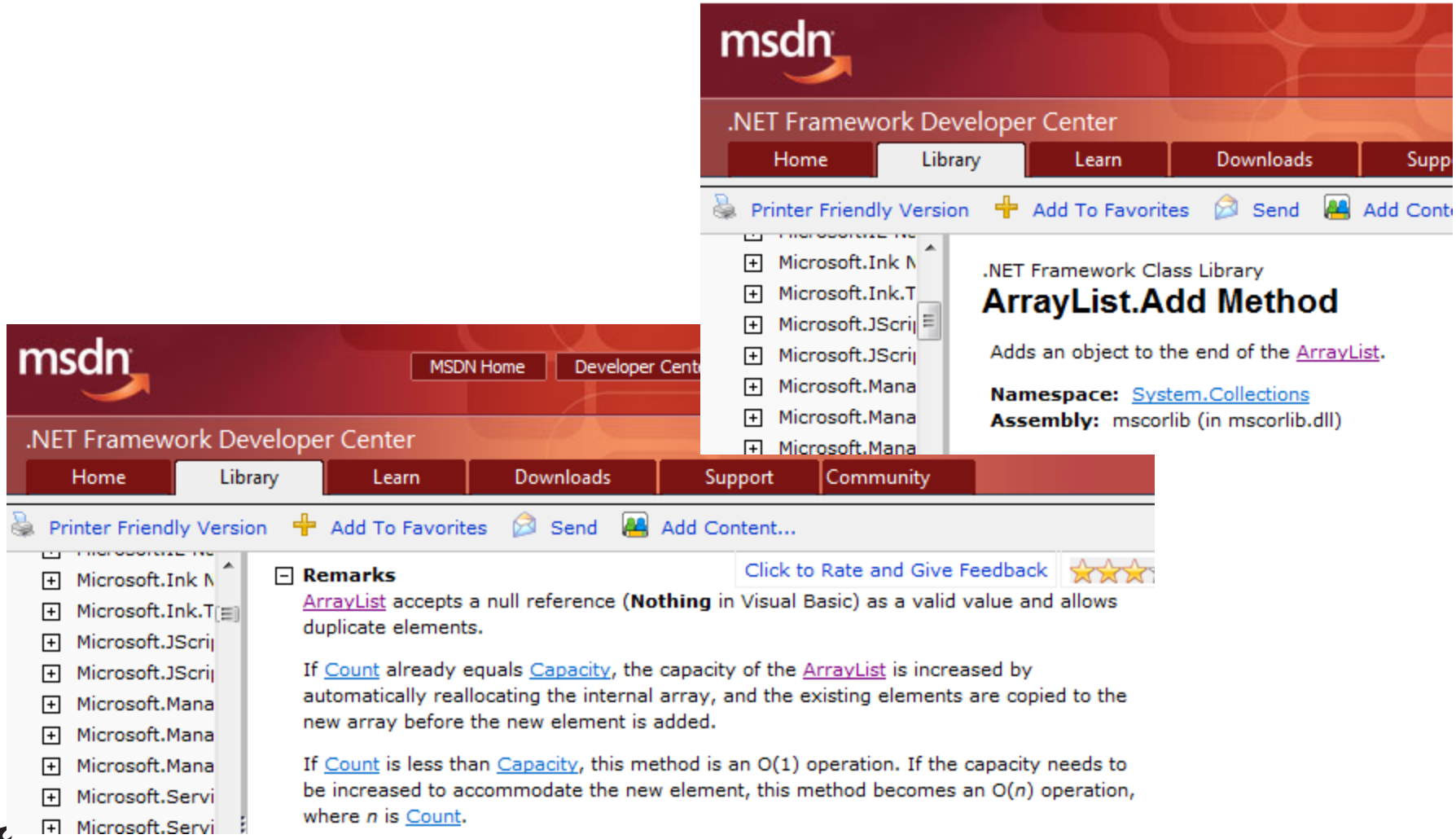
- Can quickly solve problems such as Sudoku, Scheduling, …

- *and efficiently analyze code!*

Example: Directed Automated Random Testing (DART) in Microsoft PEX



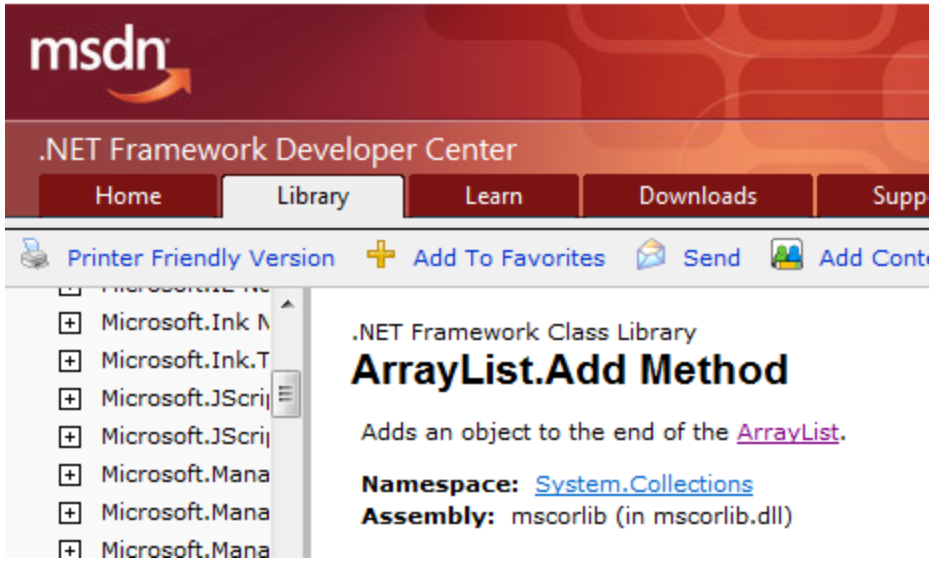
PEX concolic testing of ArrayList



ArrayList: AddItem Test

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```



ArrayList: Starting Pex…

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

Inputs

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

ArrayList: Run 1, (0,null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

Inputs
(0,null)

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

ArrayList: Run 1, (0,null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

	Inputs	Observed Constraints
	(0,null)	!(c<0)

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

c < 0 → false

ArrayList: Run 1, (0,null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

0 == c → true

ArrayList: Run 1, (0,null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c

```
item == item  →  true
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

This is a *tautology*,
i.e. a constraint that is always true,
regardless of the chosen values.

We can ignore such constraints.

ArrayList: Run 1, (0,null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

Q: How to trigger another branch?

ArrayList: Picking next branch

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) && 0!=c		

Negate the observed constraints!

ArrayList: Solve using SMT solver

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) &&	(1,null)	

ArrayList: Run 2, (1, null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) &&	(1,null)	!(c<0) && 0!=c

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

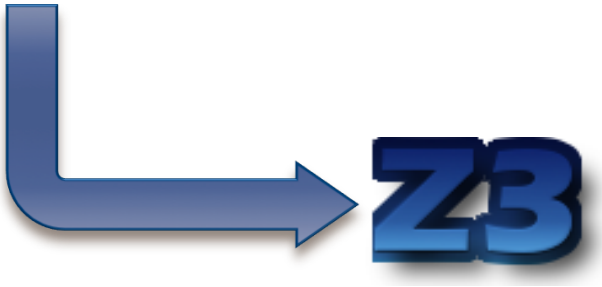
0 == c → false

ArrayList: Pick new branch

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) &&	(1,null)	!(c<0) && 0!=c
c<0		



ArrayList: Run 3, (-1, null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) &&	(1,null)	!(c<0) && 0!=c
c<0	(-1,null)	

ArrayList: Run 3, (-1, null)

```
class ArrayListTest {
  [PexMethod]
  void AddItem(int c, object item) {
      var list = new ArrayList(c);
      list.Add(item);
      Assert(list[0] == item); }
}
```

Constraints to solve	Inputs	Observed Constraints
	(0,null)	!(c<0) && 0==c
!(c<0) &&	(1,null)	!(c<0) && 0!=c
c<0	(-1,null)	c<0

```
class ArrayList {
  object[] items;
  int count;

  ArrayList(int capacity) {
    if (capacity < 0) throw ...;
    items = new object[capacity];
  }

  void Add(object item) {
    if (count == items.Length)
      ResizeArray();

    items[this.count++] = item; }
...
```

c < 0 → true

Overview

- Never simply random testing of code
  - This can indeed take forever…
- Mutation-based Fuzzing
  - manipulate and combine existing valid inputs
- Genetic Algorithms:
  - use some form of branch distance/approach level
- Concolic Execution:
  - represent code in logic and analyze using SMT solvers
- Both:
  - generate tests, execute tests, collect traces, analyze branches, generate new tests, etc.

KLEE

- An LLVM based concolic execution engine
- http://klee.github.io/docs/
- Very easy to use, simply declare parameters as symbolic, and run KLEE!

KLEE

- An LLVM based concolic execution engine
- http://klee.github.io/docs/
- Very easy to use, simply declare parameters as symbolic, and run KLEE!

Now:
Let's get KLEE running on RERS

About the second assignment

- **Be Inspired!**
- We provide you with a set of papers/blogposts/videos, either:
  1. Pick one of these and replicate the performed study
  2. Find your own paper/blogpost/video to replicate, needs teacher's approval

- You are asked to make a video (max 10 mins) demonstrating the code, tool, and study results

- In your study, it is obligatory to also use one of the methods from the second part of the course:
  1. Mutation analysis
  2. Model-based testing
  3. State machine learning
  4. Binary reverse engineering