Techniques for Improving Software Productivity

Instructor: Mooly Sagiv
TA: Kalev Alpernas

http://cs.tau.ac.il/~msagiv/courses/software-productivity.html

Slides from Eran Yahav, Zach Tatlock and the Noun Project, Wikipedia

Course Prerequisites

- Logic in Computer Science
- Software Project

Course Requirements

- The students must solve all homework assignments but one (40%)
  - Apply a tool
  - ~10 hours per project
  - First assignment available on next Thursday
- 60% final exam

Software is Everywhere

Exploitable Software is Everywhere

Exploitable Soft re is Everywhere

Sony PlayStation

ch: An

RSA hacked, information leaks

SA's co

Stuxnet Worm Still Out of Control at Iran's Nuclear Sit

Experts S

Security Advisory for Adobe Flash Player, Adobe Read nd Acrobat

This vulnerability c

and potentially **allo**

RSA tokens may be behind major network security problems at

Buffer Overrun

```
void foo (char *x) {
  char buf[2];
  strcpy(buf, x);
}
int main (int argc, char *argv[]) {
  foo(argv[1]);
}
```

source code

```
> ./a.out abracadabra
```

```
Segmentation fault
```

terminal

memory

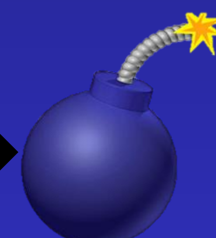
Buffer Overrun Exploits

```
int check_authentication(char *password) {
 int auth_flag = 0;
 char password_buffer[16];

 strcpy(password_buffer, password);
 if(strcmp(password_buffer, "brillig") == 0) auth_flag = 1;
 if(strcmp(password_buffer, "outgrabe") == 0) auth_flag = 1;
  return auth_flag;
}
int main(int argc, char *argv[]) {
    if(check_authentication(argv[1])) {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");           }
   else
        printf("\nAccess Denied.\n");
}
```

(source: “hacking – the art of exploitation, 2nd Ed”)

Attack

evil input → Application →



AAAAAAAAAAAA

-=-=-=-=-=-=-=-=-=-=-=-=-=-

Access Granted. 65

-=-=-=-=-=-=-=-=-=-=-=-=-=-

- A sailor on the U.S.S. Yorktown entered a 0 into a data field in a kitchen-inventory program
- The 0-input caused an overflow, which crashed all LAN consoles and miniature remote terminal units
- The Yorktown was dead in the water for about two hours and 45 minutes

One Day Last Summer…

The New York Times

The Stock Market Bell Rings, Computers Fail, Wall Street Cringes

By NATHANIEL POPPER JULY 8, 2015

Problems with technology have at times roiled global financial markets, but the 223-year-old New York Stock Exchange has held itself up as an oasis of humans ready to step in when the computers go haywire.

On Wednesday, however, those working on the trading floor were left helpless when the computer systems at the exchange went down for nearly four hours in the middle of the day, bringing an icon of capitalism's ceaseless energy to a costly halt.

The exchange ultimately returned to action shortly before the closing bell,

One Day Last Summer…

The New York Times

The Stock Market Bell Rings, Computers Fail, Wall Street Cringes

By NATHANIEL POPPER JULY 8, 2015

Problems with technology have at times roiled global financial markets, but the 223-year-old New York Stock Exchange has held itself up as an oasis of humans ready to step in when the computers go haywire.

On Wednesday, however, those working on the trading floor were left helpless when the computer systems at the exchange went down for nearly four hours in the middle of the day, bringing an icon of capitalism's ceaseless energy to a costly halt.

The exchange ultimately returned to action shortly before the closing bell,

THE WALL STREET JOURNAL. Digital Network WSJ.com MarketWatch BARRON'S

THE WALL STREET JOURNAL.

Home

WSJ.com is having technical difficulties. The full site will return shortly.

Cyber Sleuths Trac… to China's Military

The story of a Chinese military staffer's … ent in hacking provides a detailed look into B… state-controlled cyberespionage machinery.

Aw, Snap!

Debt Relief for Students Snarls Market for Their Loans

Federal programs designed to ease the burden of college loans are causing snarls in the bond market and raising concerns that banks may soon ratchet back lending.

The New Bond Market: Algorithms Trump Humans

Computerized trading strategies, or algorithms, are remaking the $12.7 trillion Treasury market, emulating earlier sea changes in stock and currency trading.

One Day Last Summer…

The New York Times

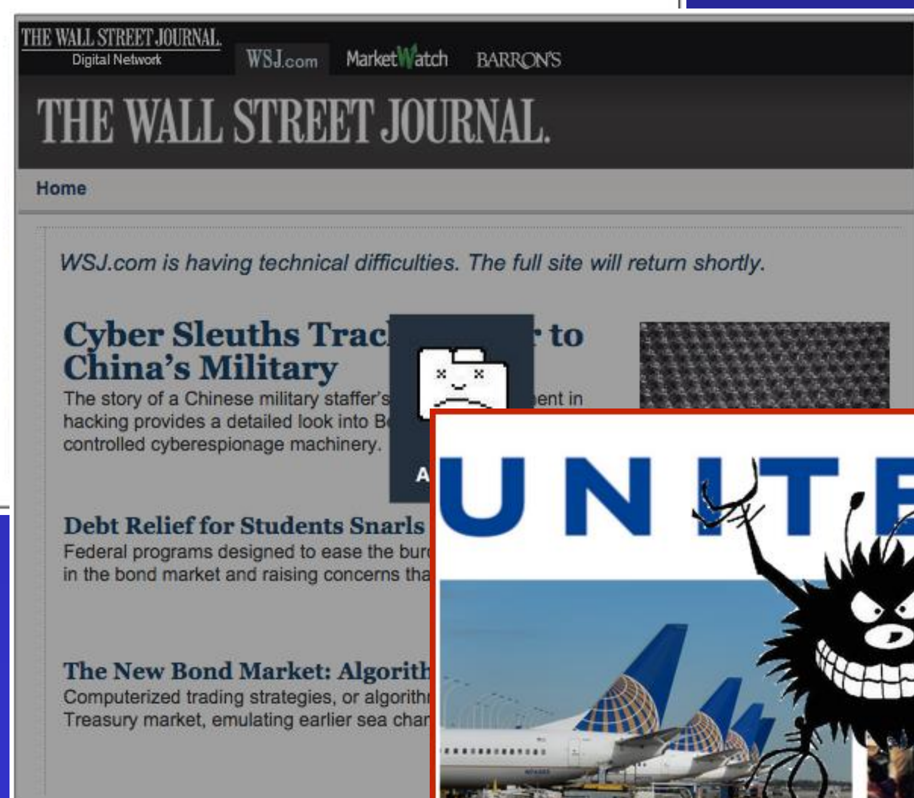
The Stock Market Bell Rings, Computers Fail, Wall Street Cringes

By NATHANIEL POPPER JULY 8, 2015

Problems with technology have at times roiled global financial markets, but the 223-year-old New York Stock Exchange has held itself up as an oasis of humans ready to step in when the computers go haywire.

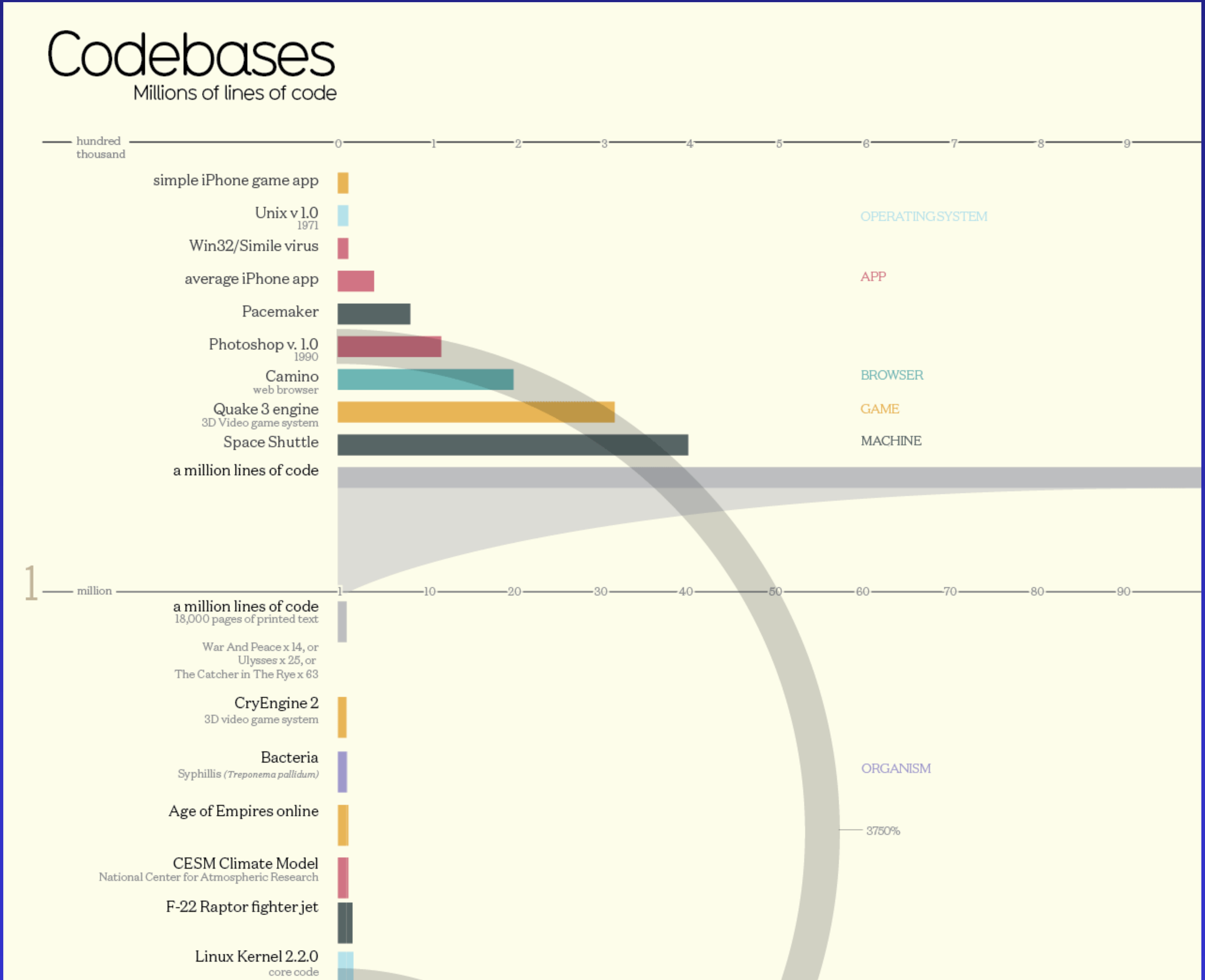
On Wednesday, however, those working on the trading floor were left helpless when the computer systems at the exchange went down for nearly four hours in the middle of the day, bringing an icon of capitalism's ceaseless energy to a costly halt.

The exchange ultimately returned to action shortly before the closing bell,



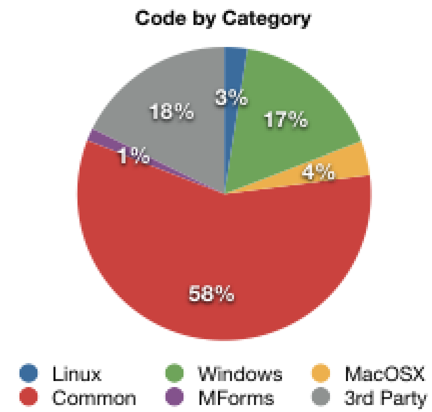
Software is Complex

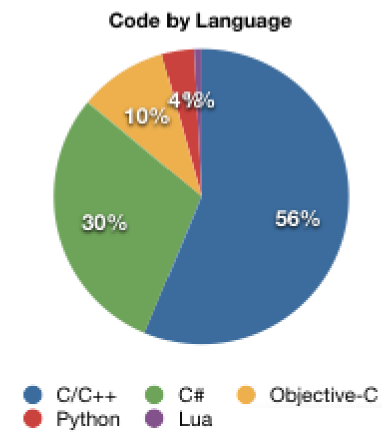
http://www.informationisbeautiful.net/visualizations/million-lines-of-code/



MySQL Workbench 5.2 Code Statistics

	Files	Lines of Code
Linux	65	43782
Windows	430	93065
MacOSX	97	19198
Common	1497	325001
MForms	36	9499
3rd Party	457	201401
Total	2582	691946
C/C++	752	340492
C#	397	100297
Objective-C	129	29129
Python	47	15260
Lua	11	3061
XML Files	63	
Icons	390	
Glade UI	24	
NIB	89	
.NET Designer	76	





Cost of software bugs

- 59.5 billion dollars in the US due to software bugs
- Software security
  - Cars, Planes, Radiotherapy, Internet, ….
- Software agility

Improving Software Productivity

- High level programming languages
  - Abstractions
- Software Engineering
  - Software designs
- Software tools
  - Software testing
  - Software debugging
  - Formal Verification

Software Testing

- Goal: “to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances” [E. F. Miller, Introduction to Software Testing Technology]

The Testing Spectrum

- Unit Testing: basic unit of software
- Integration Testing: combination
- System’s testing: end-to-end
- Acceptance testing: client check

Testing Techniques

- Random testing: Runs the program on random inputs
- Symbolic techniques
- Concolic techniques
- Adequacy of test suit
  - Coverage
  - Mutation testing: Modify the program in a small way
    - Check the adequacy of the test suit

Symbolic vs. Concrete Testing

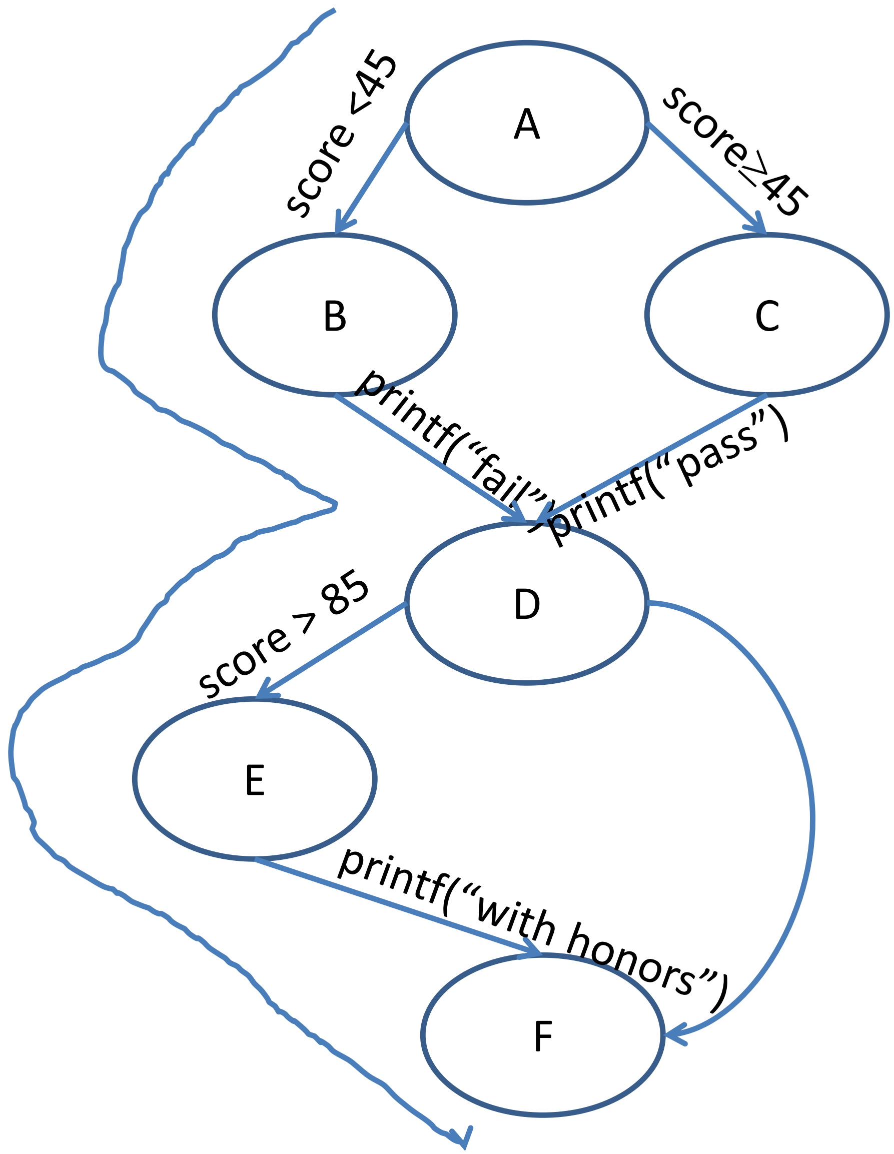
Mooly Sagiv

Program Path

- Program Path
  - A path in the control flow of the program
    - Can start and end at any point
    - Appropriate for imperative programs
- Feasible program path
  - There exists an input that leads to the execution of this path
- Infeasible program path
  - No input that leads to the execution

Infeasible Paths

```
void grade(int score) {
A:  if (score <45) {
B:     printf("fail");
     }
    else
 C:    printf("pass");
     }
 D:  if (score > 85) {
 E:     printf("with honors");
          }
F:
}
```



Concrete vs. Symbolic Executions

- Real programs have many infeasible paths
  - Ineffective concrete testing
- Symbolic execution aims to find rare errors

Symbolic Testing Tools

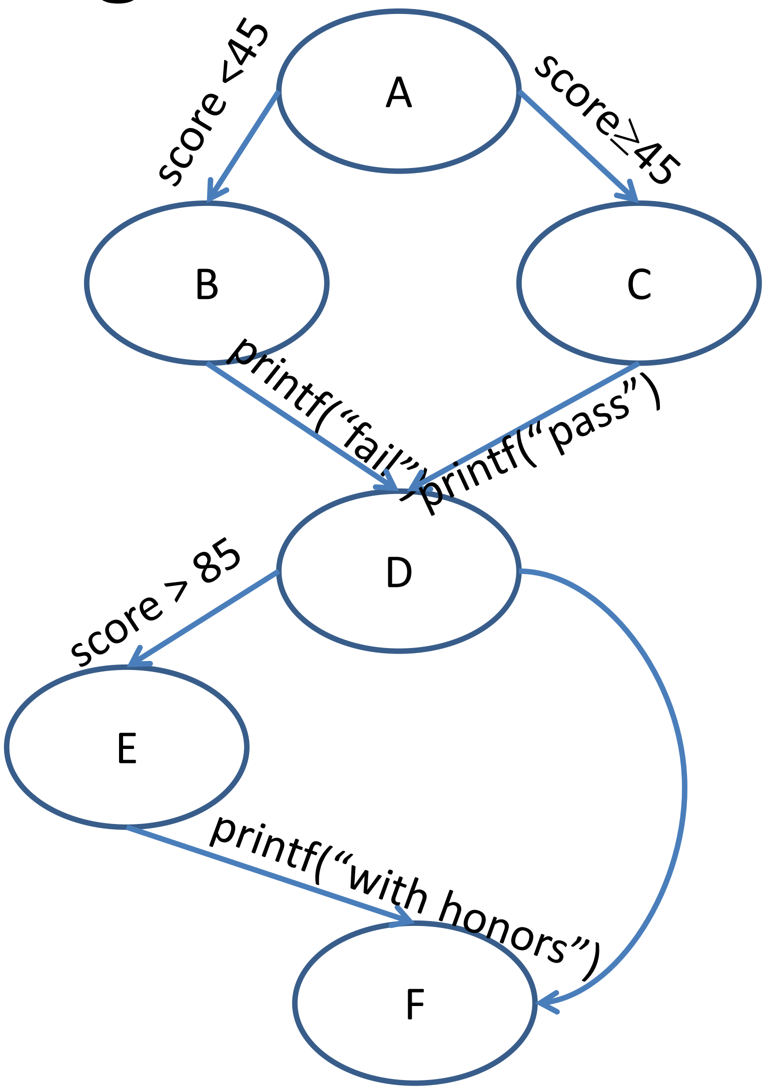
- EFFIGY [King, IBM 76]
- PEX [MSR]
- SAGE [MSR]
- SATURN[Stanford]
- KLEE[Stanford]
- Java pathfinder[NASA]
- Bitscope [Berkeley]
- Cute [UIUC, Berkeley]
- Calysto [UBC]

Finding Infeasible Paths Via Constraint Solving

```
void grade(int score) {
A:  if (score <45) {
B:    printf("fail");
     }
    else
 C:    printf("pass");
     }
 D:  if (score > 85) {
 E:     printf("with honors");
         }
F:
}
```

score < 45 ∧ score > 85

UNSAT



Plan

- Random Testing
- Symbolic Testing
- Concolic Testing

Fuzzing [Miller 1990]

- Test programs on random unexpected data
- Can be realized using black/white testing
- Can be quite effective
  - Operating Systems
  - Networks
- …
- Usually implemented via instrumentation
- Tricky to scale for programs with many paths

```
If (x == 10001) {

   ….
   if (f(*y) == *z) {
  ….
```

```
int f(int *p) {

  if (p !=NULL) {
    return q ;

}
```

Success Stories Fuzzing

- Crashes to Unix [90s]
- Crashes to all systems
  - American Fuzzy Lop http://lcamtuf.coredump.cx/afl/

Symbolic Exploration

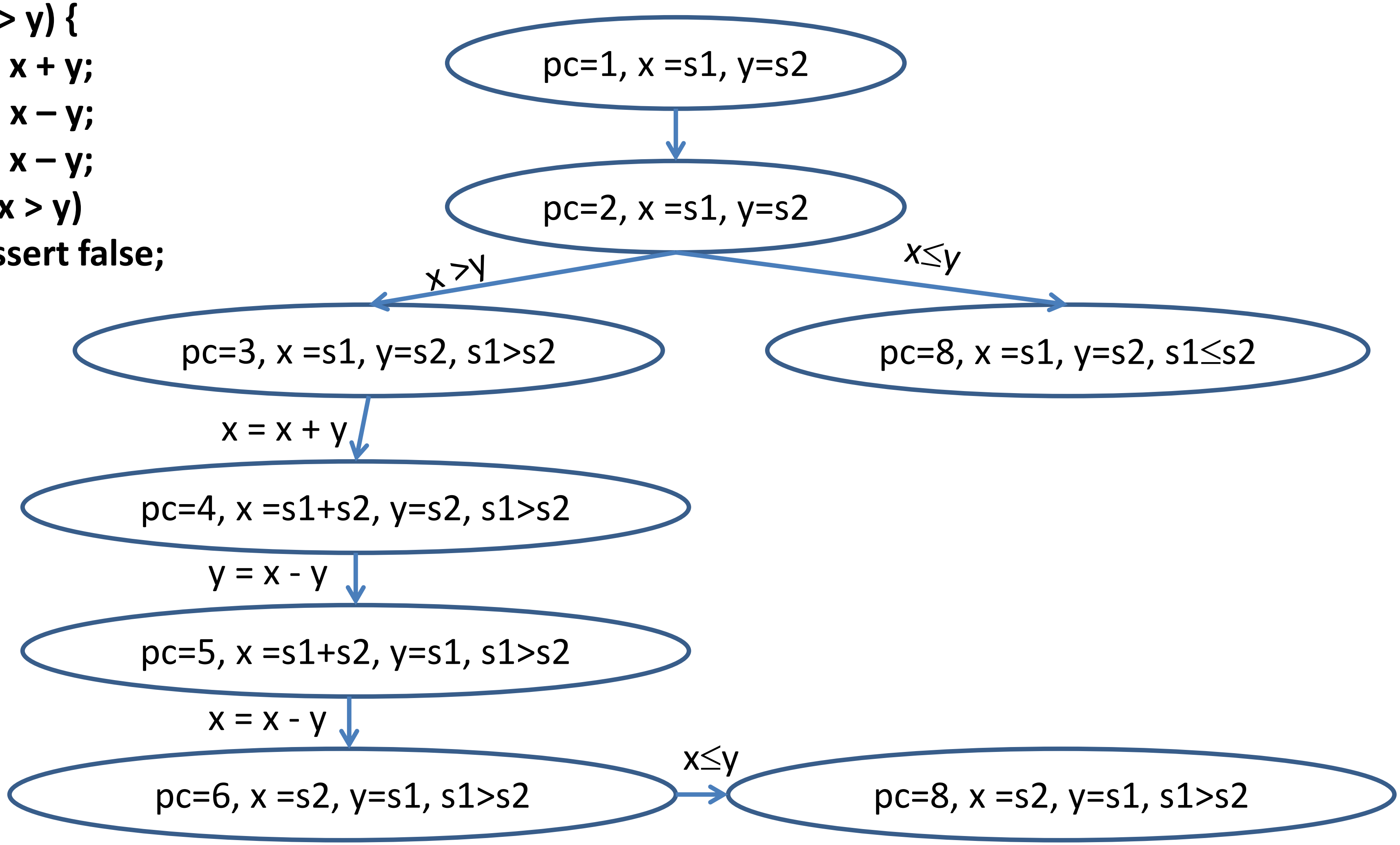
- Execute a program on symbolic inputs
- Track set of values symbolically
- Update symbolic states when instructions are executed
- Whenever a branch is encountered check if the path is feasible using a theorem prover call

Symbolic Execution Tree

- The constructed symbolic execution paths
- Nodes
  - Symbolic Program States
- Edges
  - Potential Transitions
- Constructed during symbolic evaluation
- Each edge requires a theorem prover call

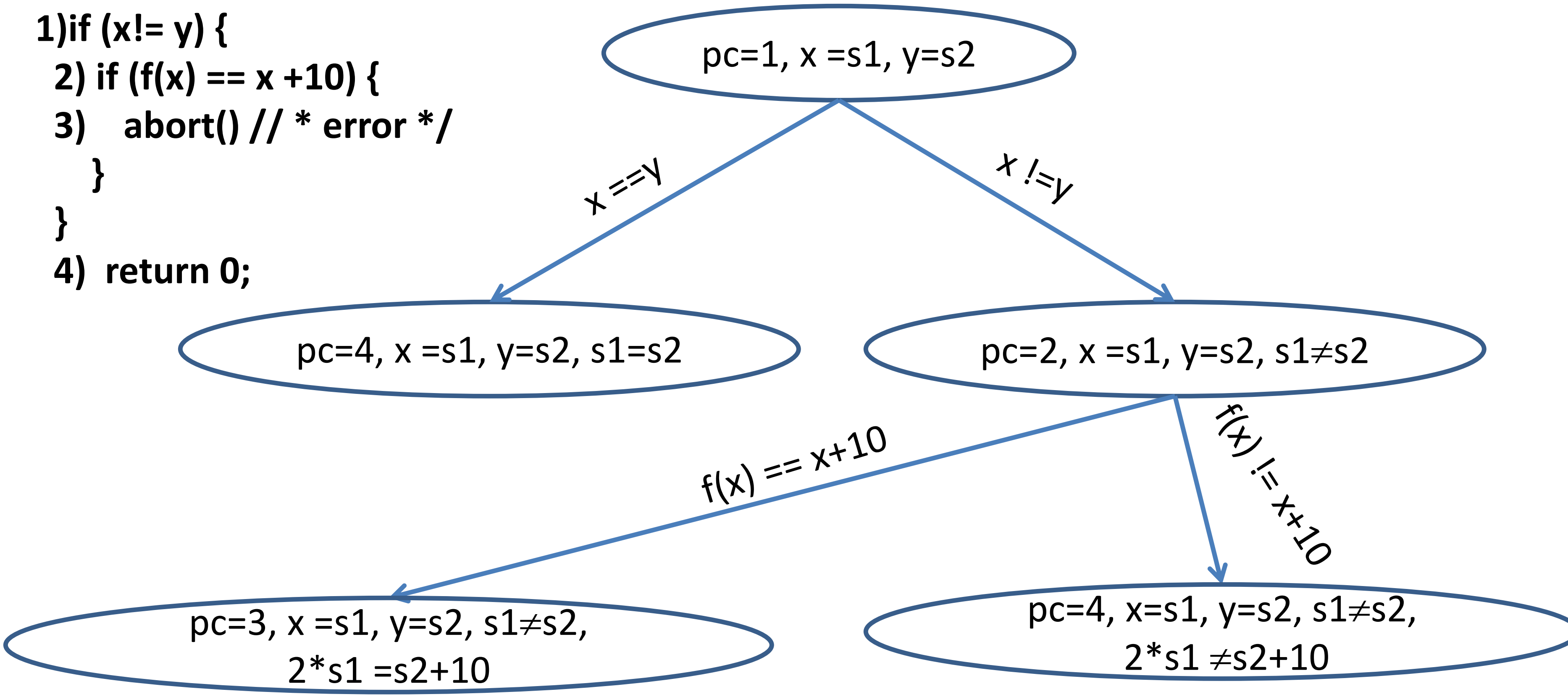
Simple Example

1)int x, y;
2)if (x > y) {
3) x = x + y;
4) y = x – y;
5) x = x – y;
6) if (x > y)
7) assert false;
8)}



Another Example

```
int f(int x) { return 2 * x ;}
int h(int x, int y) {
1)if (x!= y) {
  2) if (f(x) == x +10) {
  3)    abort() // * error */
      }
  }
  4)  return 0;
```



Non-Deterministic Behavior

```
int x; y;
1) if (nondet()) {
     2) x = 7;
     }
 else  {
    3) x = 19 ;
   }
4)
```

Loops

```
1) int i;
2) while  i < n {
      i = i + 1;
     }
3) if (n ==10^6) {
4)     abort();
5) }
```

Scaling Issues for Symbolic Exploration

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22, y = 7	$x = x_0$, $y = y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
$z = double(y);$ →	$x = 22, y = 7, z = 14$	$x = x_0, y = y_0, z = 2*y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$2*y_0 \text{ != } x_0$
x = 22, y = 7, z = 14	$x = x_0$, $y = y_0$, $z = 2*y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }
}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
$x = 22$, $y = 7$, $z = 14$	$x = x_0$, $y = y_0$, $z = 2*y_0$	$2*y_0 \mathrel{!=} x_0$

Solve: $2*y_0 == x_0$
Solution: $x_0 = 2$, $y_0 = 1$

Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	x = 2, y = 1	$x = x_0$, $y = y_0$	

Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 2, y = 1, z = 2	$x = x_0$, $y = y_0$, $z = 2*y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
`if (z == x)`			$2*y_0 == x_0$
`if (x > y+10)` ←	$x = 2, y = 1, z = 2$	$x = x_0, y = y_0, z = 2*y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

concrete state

symbolic state

path condition

$2*y_0 == x_0$

$x_0 \cdot y_0+10$

$x = 2, y = 1, z = 2$

$x = x_0, y = y_0, z = 2*y_0$

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition

Solve: $(2^*y_0 == x_0) \wedge (x_0 > y_0 + 10)$
Solution: $x_0 = 30$, $y_0 = 15$

$2^*y_0 == x_0$

$x_0 \cdot y_0+10$

$x = 2, y = 1,$
$z = 2$

$x = x_0, y = y_0,$
$z = 2^*y_0$

Concolic Testing Approach

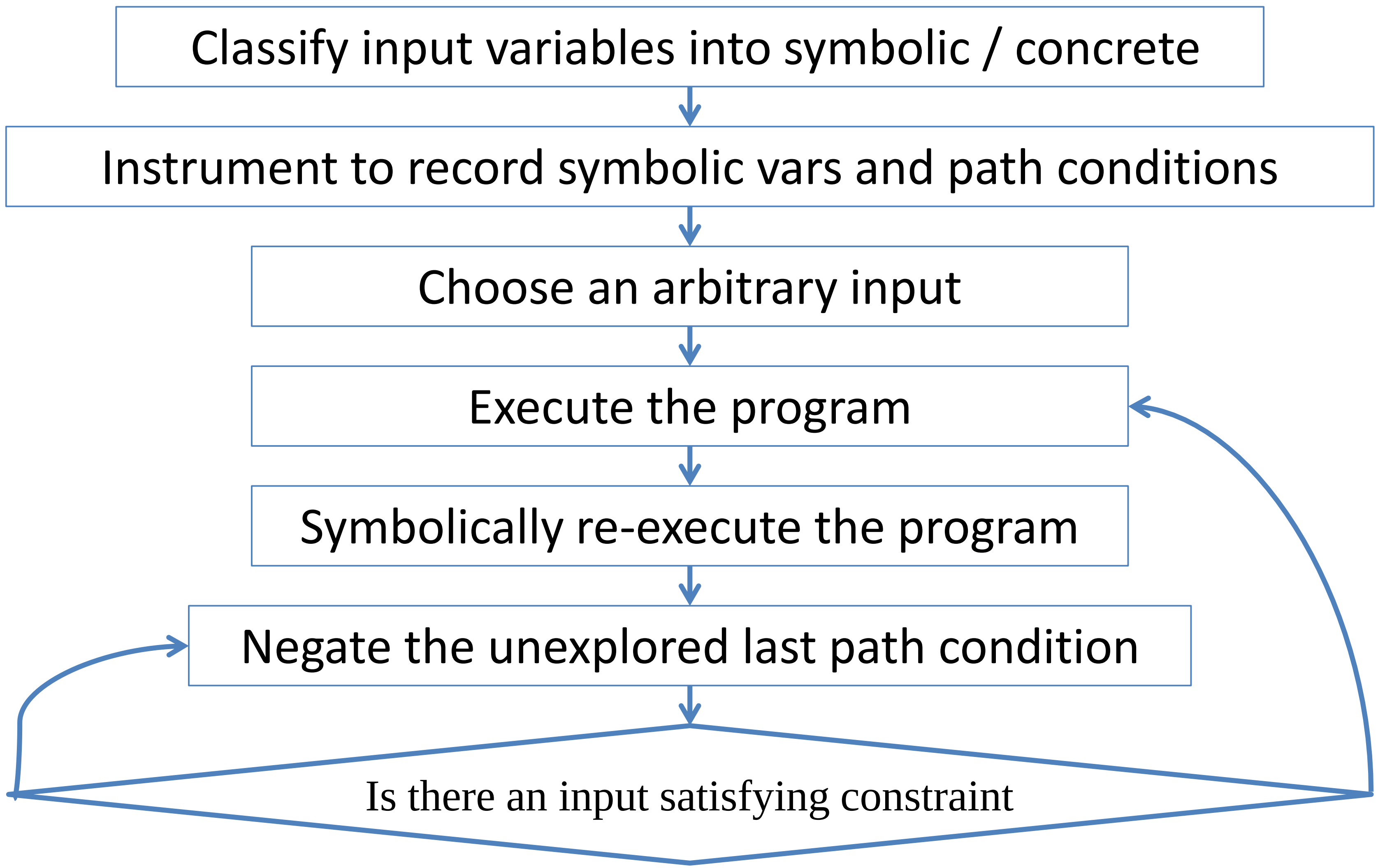
```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution | Symbolic Execution

concrete state | symbolic state | path condition

$x = 30, y = 15$ | $x = x_0, y = y_0$

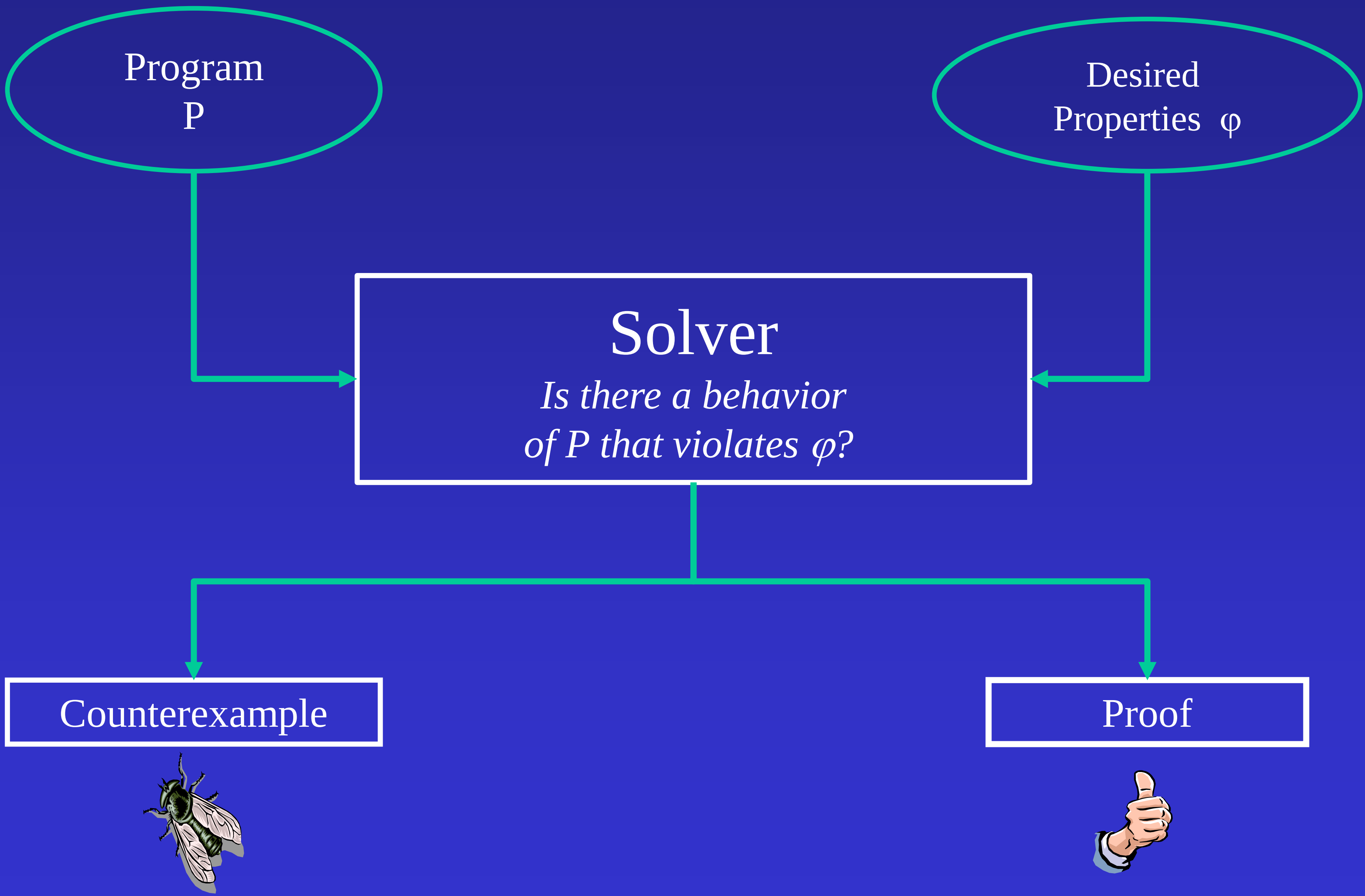
The Concolic Testing Algorithm



SAGE: Whitebox Fuzzing for Security Testing

- Check correctness of Win’7, Win’8
- 200+ machine years
- 1 Billion+ SMT constraints
- 100s of apps, 100s of bugs
- 1/3 of all Win7 WEX security bugs found
- Millions of dollars saved

Automatic Program Verification



Example

```
 int check_authentication(char *password) {
  int auth_flag = 0;
  char password_buffer[16];

  strcpy(password_buffer, password);
  if(strcmp(password_buffer, "brillig") == 0) auth_flag = 1;
  if(strcmp(password_buffer, "outgrabe") == 0) auth_flag = 1;
   return auth_flag;
}
int main(int argc, char *argv[]) {
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("      Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");          }
   else
      printf("\nAccess Denied.\n");
}
```

Undecidability

- The Halting Problem
  - Does the program P terminate on input I
- Rice’s Theorem
  - Any non-trivial property of partial functions, there is no general and effective method to decide if program computes a partial function with that property

Coping with Undecidability

- Permits occasional divergence
- Limited programs (not Turing Complete)
- Unsound Verification
  - Explore limited program executions
- Incomplete Verification
  - Explore superset of program executions
- Programmer Assistance
  - Inductive loop invariants

Limited Programs

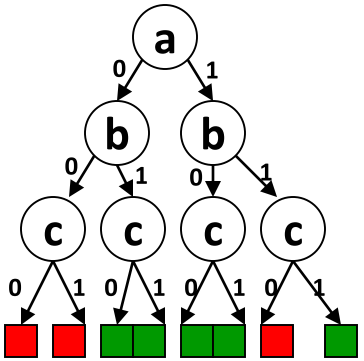
- Finite state programs
  - Finite state model checking
    - Explicit state SPIN, CHESS
    - Symbolic model checking SMV
- Loop free programs
  - Configuration files

Unsound Verification

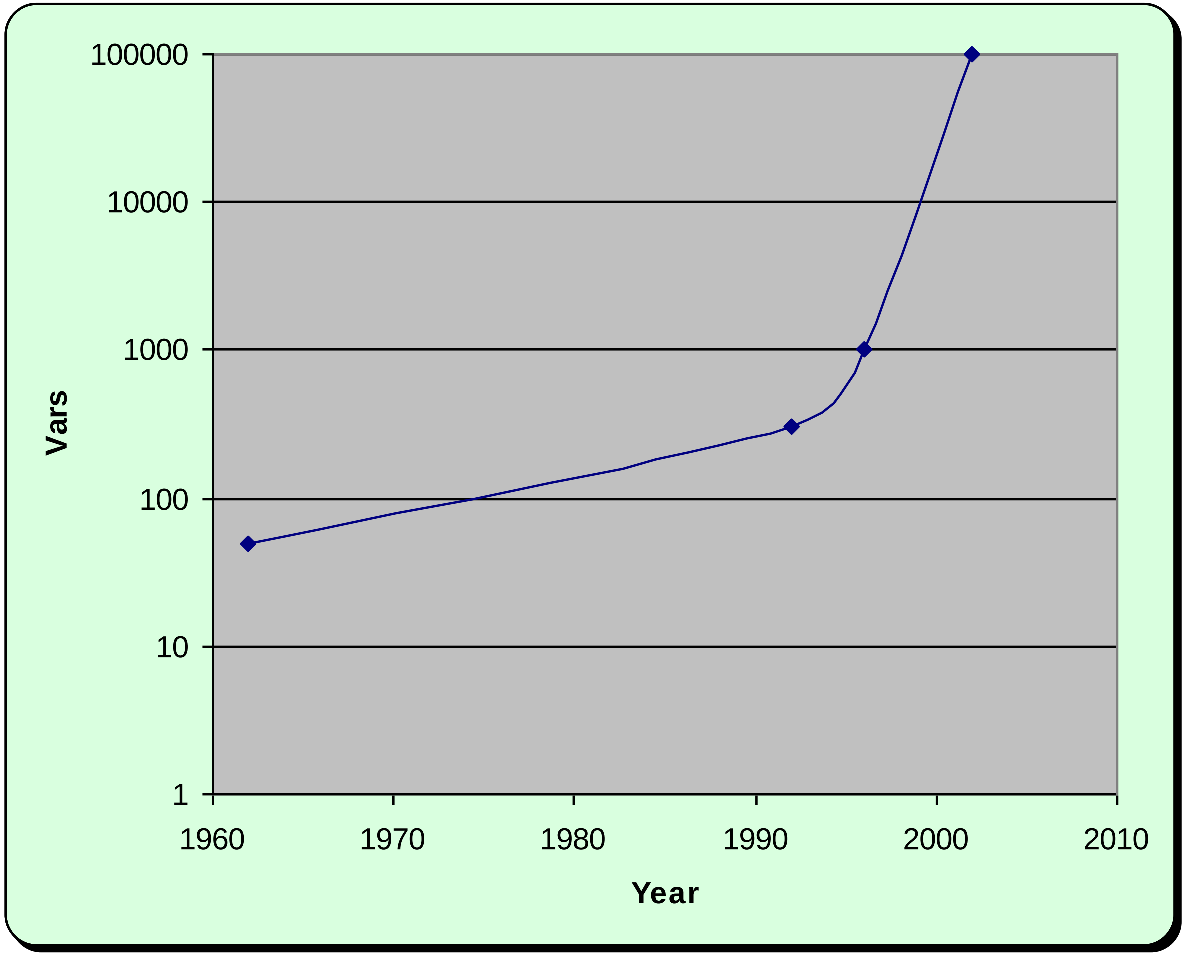
- Dynamic checking
  - Valgrind, Parasoft Insure, Purify, Eraser
- Bounded Model Checking
- Concolic Executions

The SAT Problem

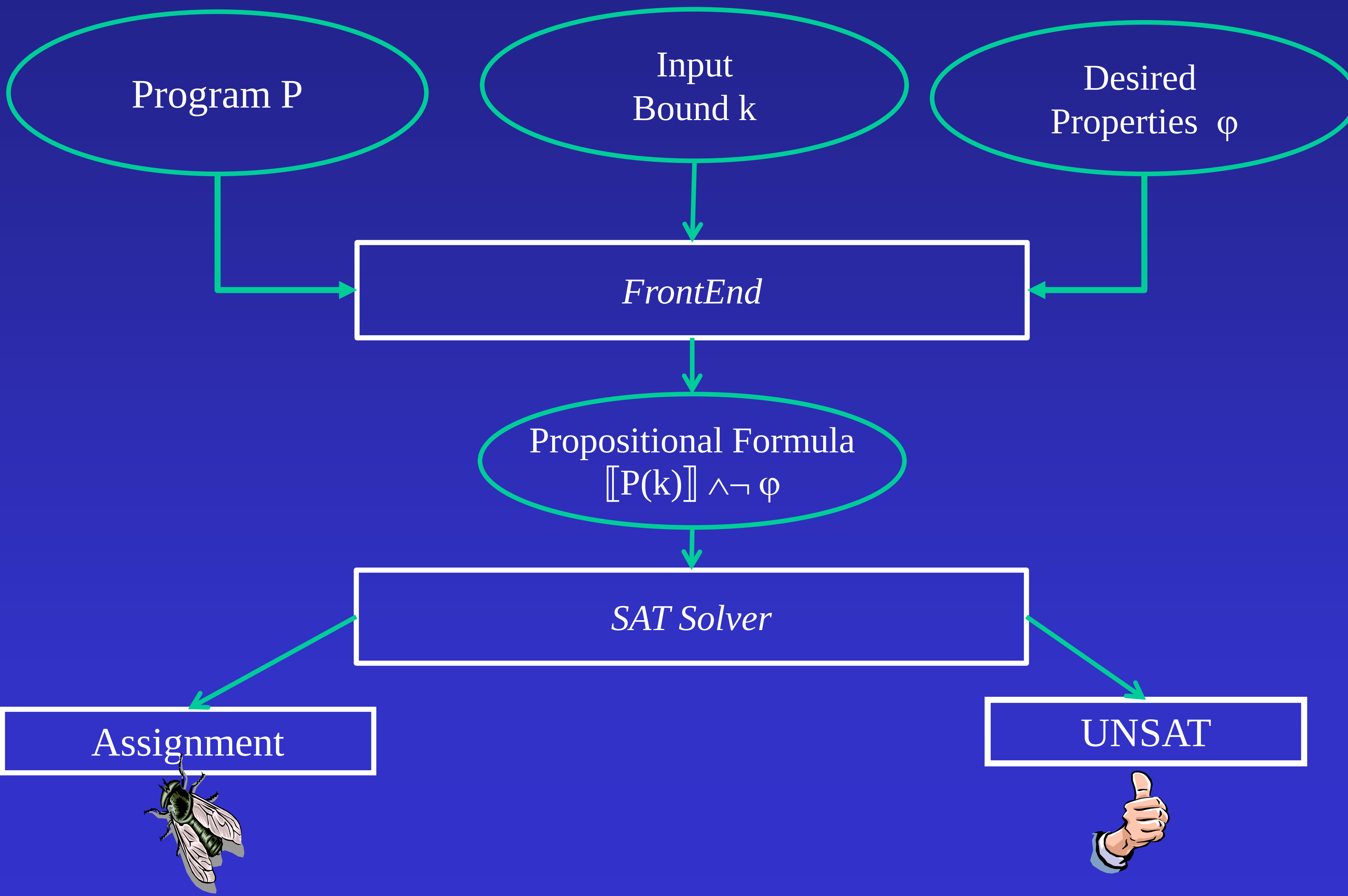
- Given a propositional formula (Boolean function)
  - $\varphi = (a \vee b) \wedge (\neg a \vee \neg b \vee c)$
- Determine if φ is satisfiable
  - Find a satisfying assignment or report that such does not exit
- For *n* variables, there are 2^n possible truth assignments to be checked



SAT made some progress…



Bounded Model Checking



A Simple Example

Program

```
int x;
int y=8,z=0,w=0;
if (x)
  z = y - 1;
else
  w = y + 1;
assert (z == 5 ||
        w == 9)
```

Constraints

y = 8,
z = x ? y – 1 : 0,
w = x ? 0 :y + 1,
z != 5,
w != 9

SAT
counterexample found!

y = 8, x = 1, w = 0, z = 7

A Simple Example

Program

```
int x;
int y=8,z=0,w=0;
if (x)
  z = y - 1;
else
  w = y + 1;
assert (z == 7 ||
        w == 9)
```

Constraints

$y = 8$,
$z = x\ ?\ y - 1 : 0$,
$w = x\ ?\ 0 : y + 1$,
$z\ != 7$,
$w\ != 9$

UNSAT
Assertion always holds!

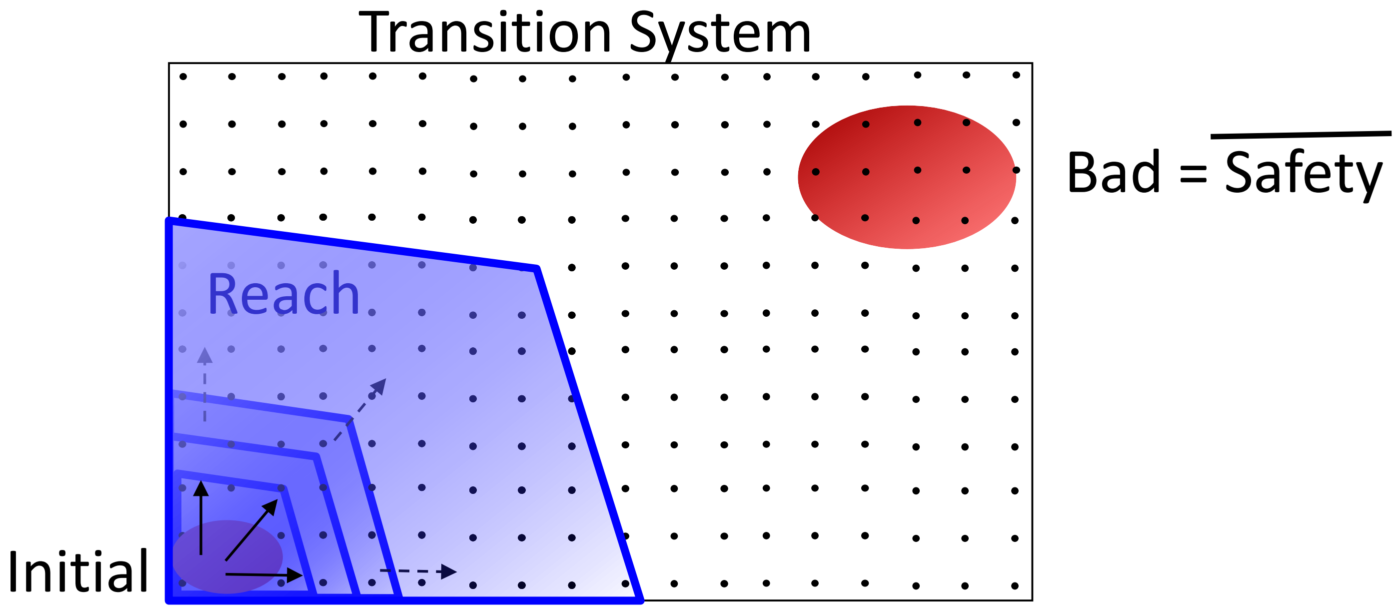
Summary Bounded Model Checking

- Excellent tools exist (CBMC, Alloy)
- Many bugs occur on small inputs
- Useful for designs too
- Scalability is an issue
- Challenging features
  - Bounded arithmetic
  - Pointers and Heap
  - Procedures
  - Concurrency

Success Stories BMC

- Car industry
- Amazon
- Regression

Safety of Transition Systems



System S is **safe** if no bad state is reachable

$R_0 = Init$ – Initial states, reachable in 0 transitions

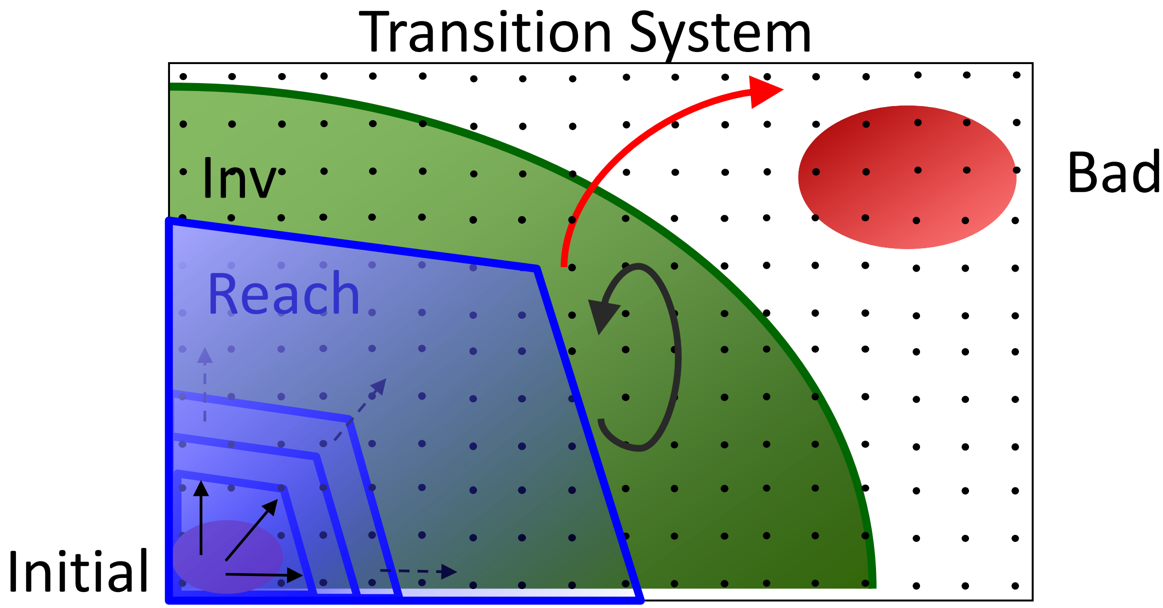
$R_{i+1} = R_i \cup \{\sigma' \mid \sigma \rightarrow \sigma' \text{ and } \sigma \in R_i\}$

$R = R_0 \cup R_1 \cup R_2 \cup \ldots$

Safety: $R \cap Bad = \emptyset$

K-Safety: $R_K \cap Bad = \emptyset$

Inductive Invariants



System S is safe if no bad state is reachable

System S is safe iff there exists an inductive invariant Inv s.t.:

Inv $\cap$ *Bad* = $\varnothing$ *(Safety)*

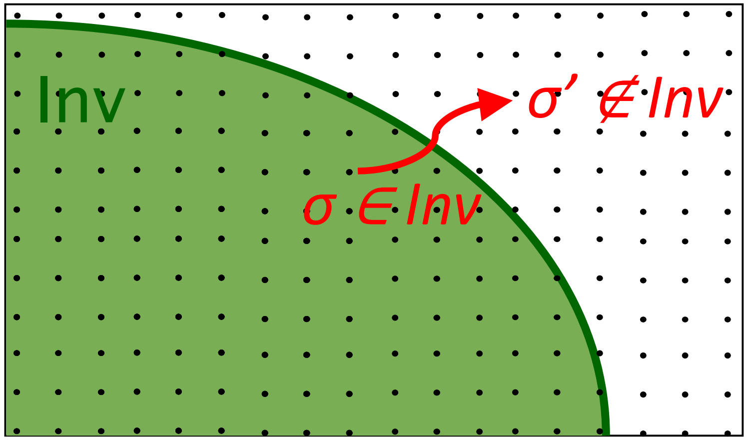
Init $\subseteq$ *Inv (Initiation)*

if $\sigma \in$ *Inv and* $\sigma \rightarrow \sigma'$ *then* $\sigma' \in$ *Inv (Consecution)*

Counterexample To Induction (CTI)

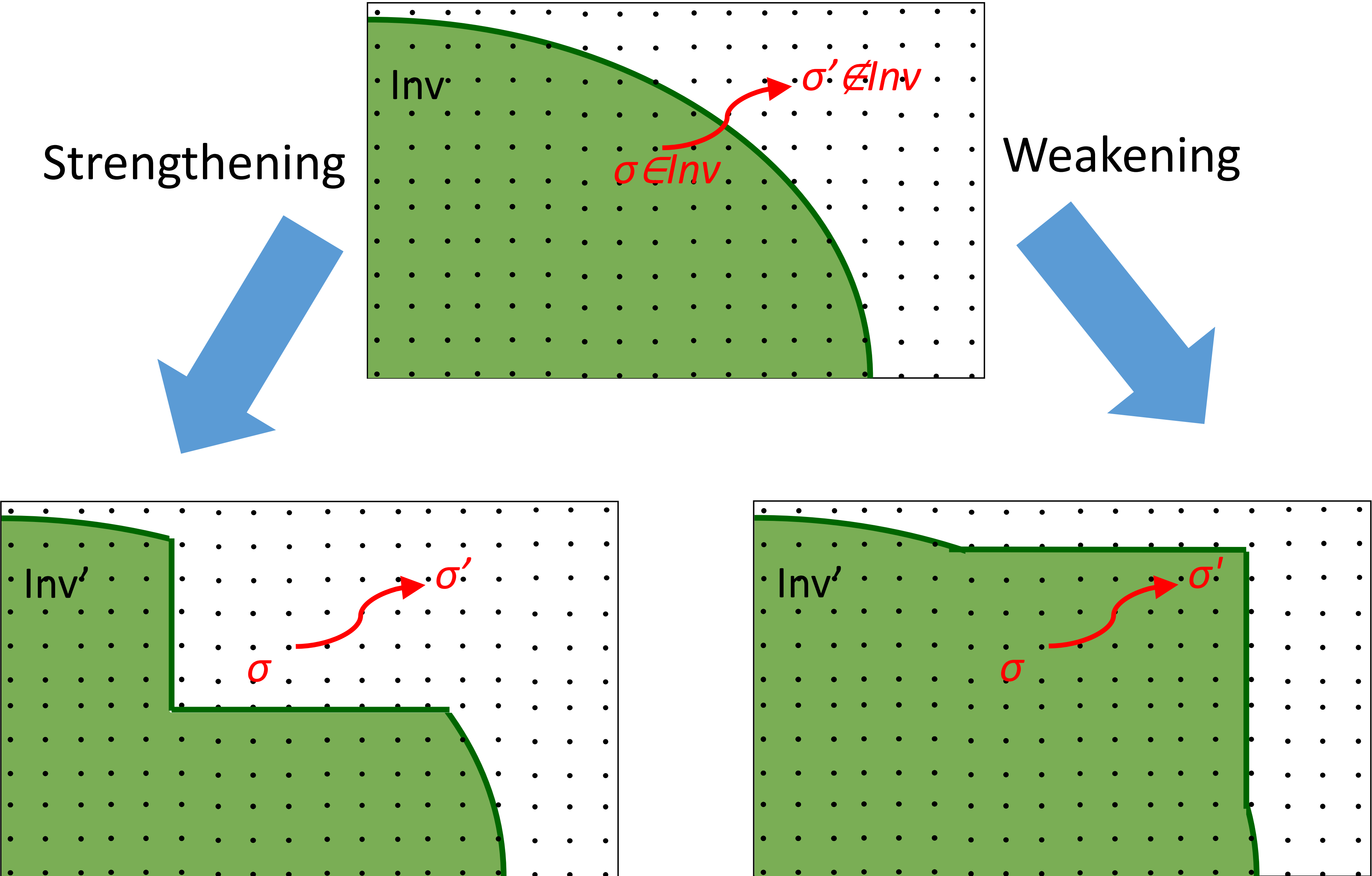
States σ,σ’ are a CTI of Inv if:

- σ ∈ Inv
- σ’ ∉ Inv
- σ → σ’

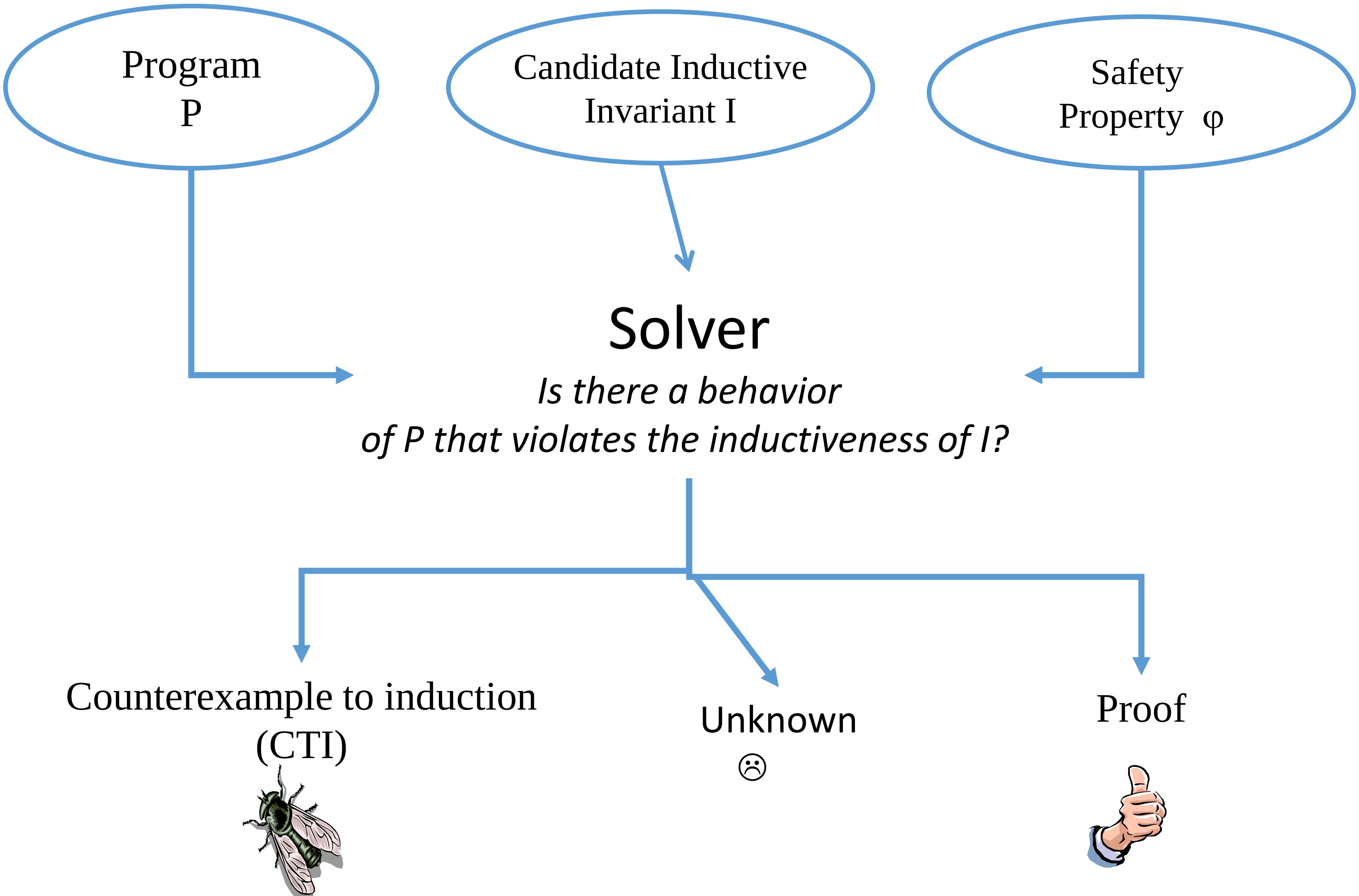


- A CTI may indicate:
  - A bug in the system
  - A bug in the safety property
  - A bug in the invariant
    - Too weak
    - Too strong

Strengthening & Weakening from CTI



Deductive (Semi-Automatic) Verification



Deductive Verification

```
1: x := 1;
2: y := 2;
while * do {
   3: assert x ≥1;
   4:  x:= x + y;
   5:  y := y + 1
}
6:
```

$I = at(3) \Rightarrow x \geq 1$

$at(3) \Rightarrow x \geq 1$

Solver

Is there a behavior
of P that violates the inductiveness of I?

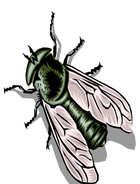
I

3: x=1, y=-2

x := x+y; y :=y+1

$\neg I$

x= -1, y=-1



Deductive Verification

```
1: x := 1;
2: y := 2;
while * do {
   3: assert x ≥1;
   4:  x:= x + y;
   5:  y := y + 1
}
6:
```

at(3) ⇒ x ≥1 ∧y≠-2

at(3) ⇒ x ≥1

Solver

Is there a behavior
of P that violates the inductiveness of I?

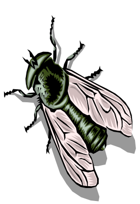
I

¬I

3: x=1, y=-7

x := x+y; y :=y+1

x=-6, y=-6



Deductive Verification

```
1: x := 1;
2: y := 2;
while * do {
   3: assert x ≥1;
   4:  x:= x + y;
   5:  y := y + 1
}
6:
```

$at(3) \Rightarrow x \geq 1 \wedge y \geq 0$

$at(3) \Rightarrow x \geq 1$

Solver

Is there a behavior
of P that violates the inductiveness of I?

Proof

Algorithmic Deductive Verification

- SAT/SMT has made huge progress in the last decade
- Great impact on verification:
  Dafny[ITP'13], IronClad/IronFleet[SOSP'15], and more
- State: finite first-order structure over vocabulary V
- Initial states and safety property (first-order formulas):
  - Init(V) – initial states
  - Bad(V) – bad states
- Transition relation:
  first-order formula TR(V, V')
  V' is a copy of V describing the next state

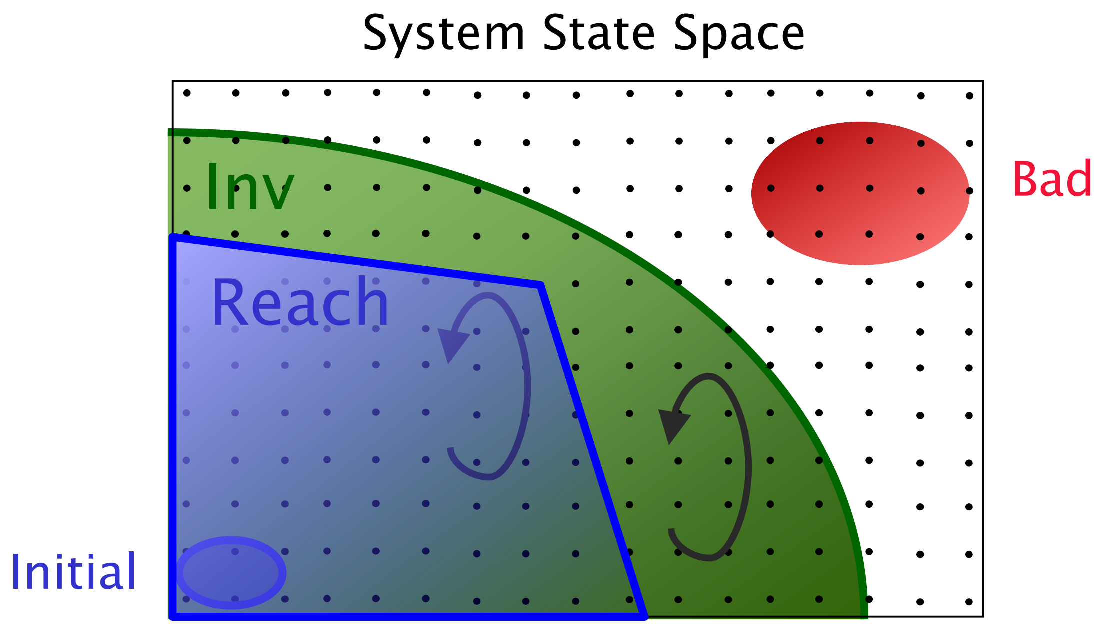
[ITP'13] K.R. Leino: Automating Theorem Proving with SMT. DAFNY

[SOSP'15] C. Hawblitzel, J. Howell, M. Kapritsos, J.R. Lorch, B. Parno, M. Roberts, S. Setty, B. Zill: IronFleet: proving practical distributed systems correct

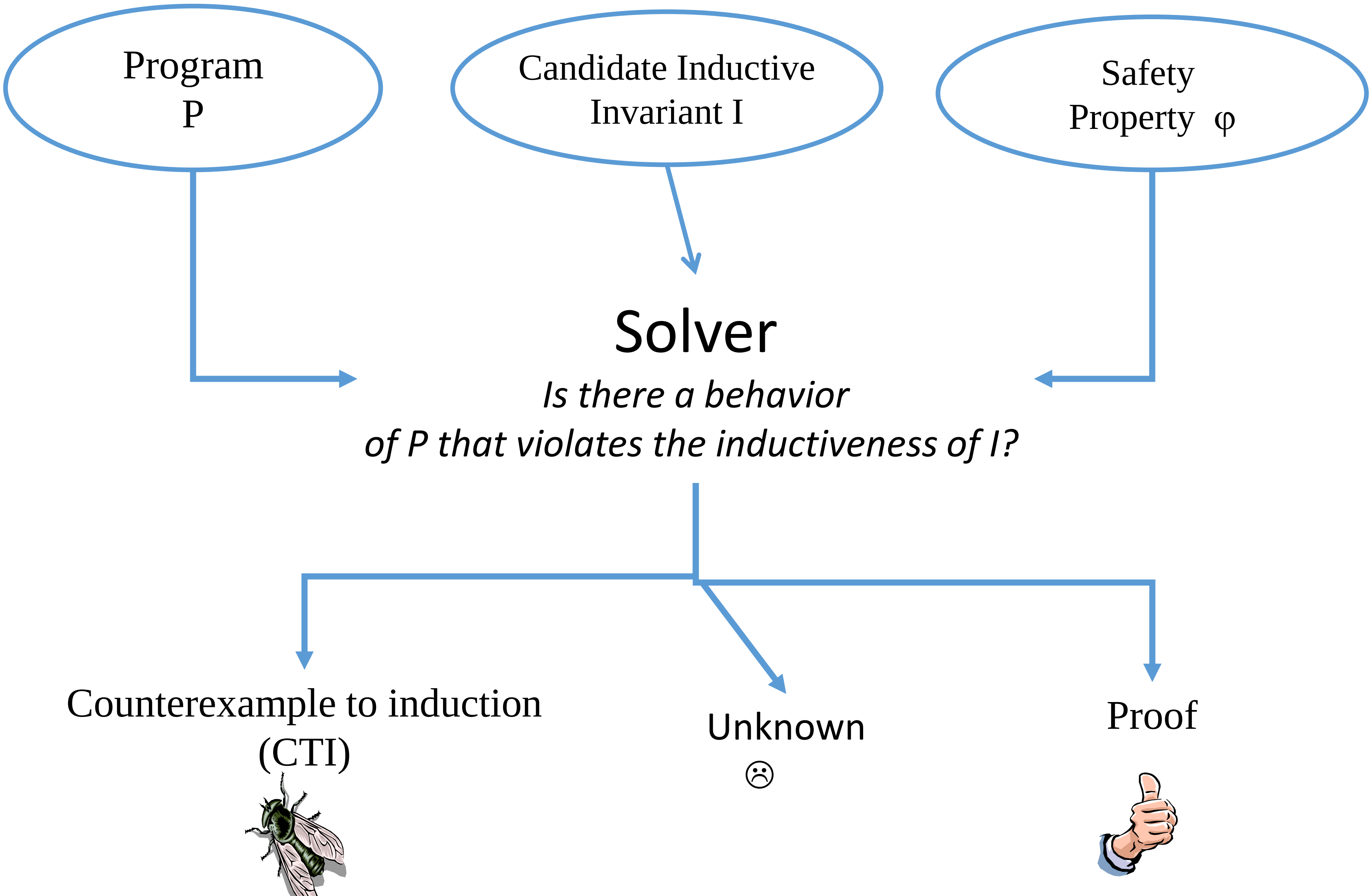
Algorithmically Checking Inductiveness

Inv is an **inductive invariant** if:

• Initiation:	Init ⇒ Inv	Init∧¬Inv	unsat
• Safety:	Inv ⇒ ¬Bad	Inv∧Bad	unsat
• Consecution:	Inv ∧ TR ⇒ Inv’	Inv∧TR∧¬Inv’	unsat



Algorithmic Deductive Verification



Challenges

1. Formal specification:
   - Modeling the system (TR, Init)
   - Formalizing the safety property (Bad)
2. Inductive Invariants (Inv)
   - Hard to specify manually
   - Hard to maintain
   - Hard to infer automatically
3. Deduction – Checking inductiveness
   - Undecidability of implication checking
     - Unbounded state, arithmetic, quantifier alternation

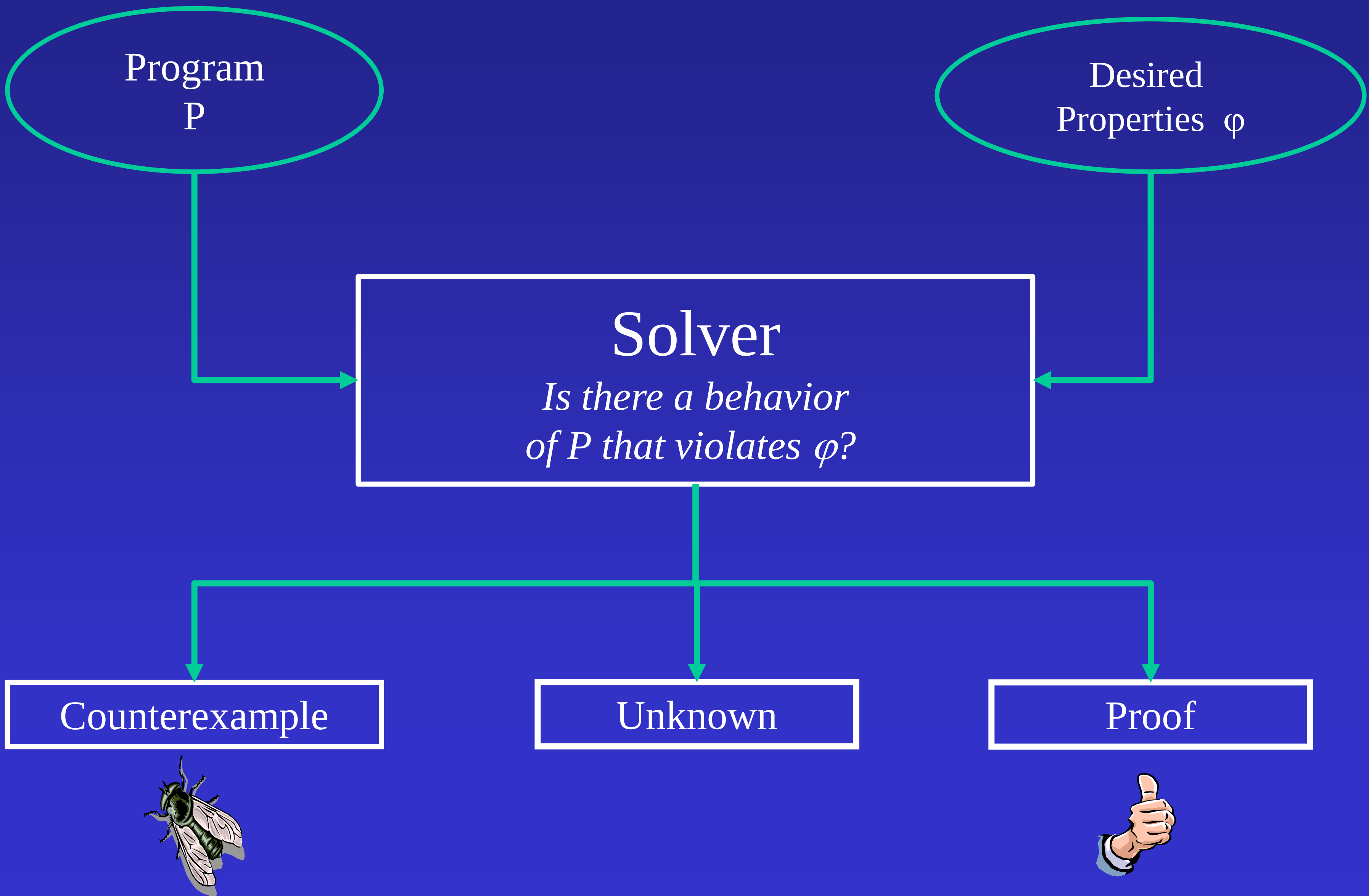
Existing Approaches for Verification

- Automated invariant inference
  - Abstract Interpretation
    - Ultimately limited due to undecidability
- Use SMT for deduction with manual program annotations (e.g. Dafny)
  - Requires programmer effort to provide inductive invariants
  - SMT solver may diverge (matching loops, arithmetic)
- Interactive theorem provers (e.g. Coq, Isabelle/HOL)
  - Programmer gives inductive invariant and proves it
  - Huge effort (10-50 lines of proof per line of code)

Abstract Interpretation

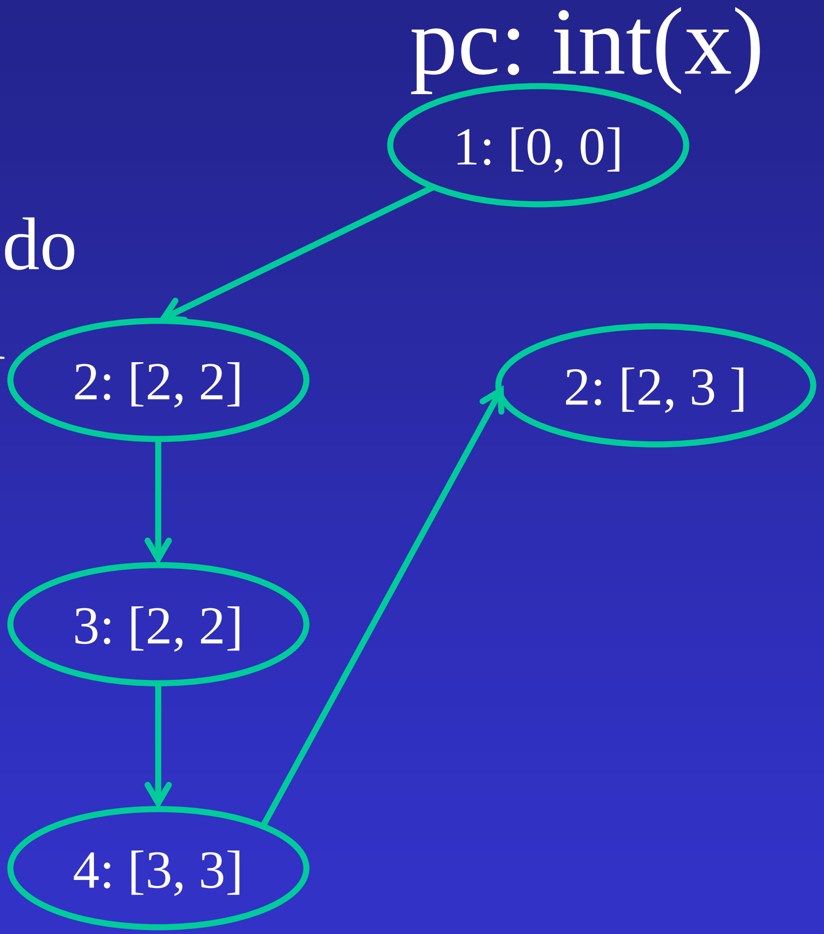
- Automatically prove that the program is correct by also considering infeasible executions
- Abstract interpretation of program statements/conditions
- Conceptually explore a superset of reachable states
- Sound but incomplete reasoning
- Automatically infer sound inductive invariants

Automatic Program Verification



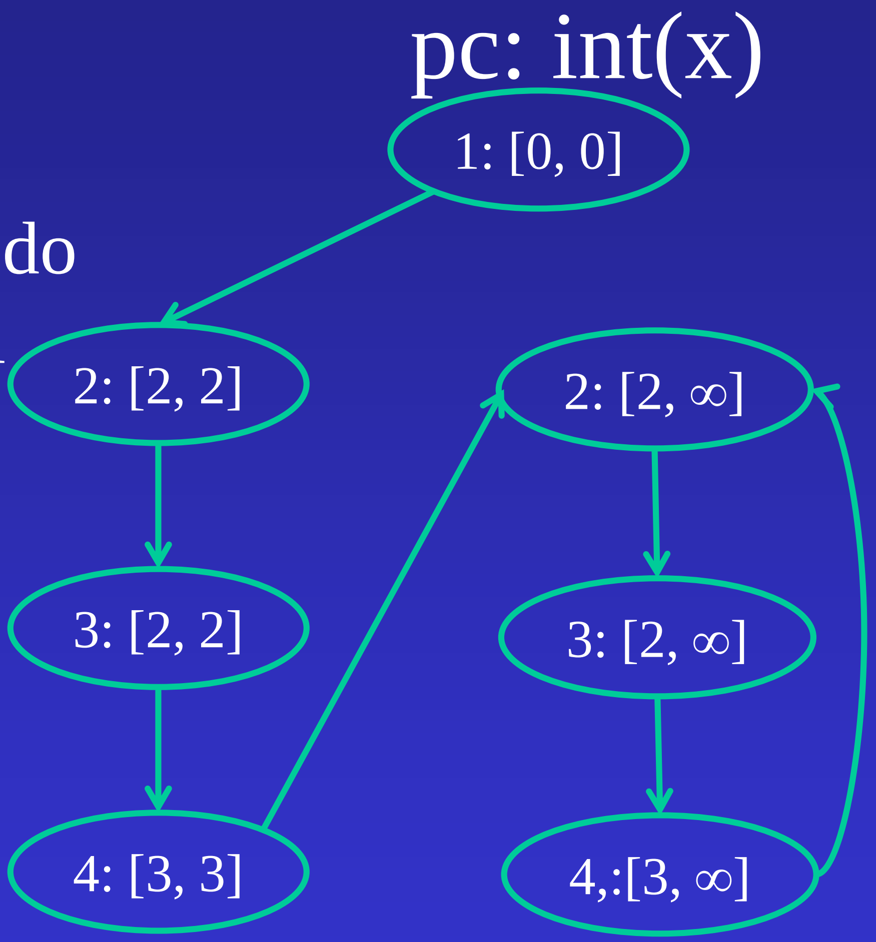
Interval Based Abstract Interpretation

1: x = 2;
2: while true {x > 0} do
    3: x = 2* x – 1
  4:



Interval Based Abstract Interpretation

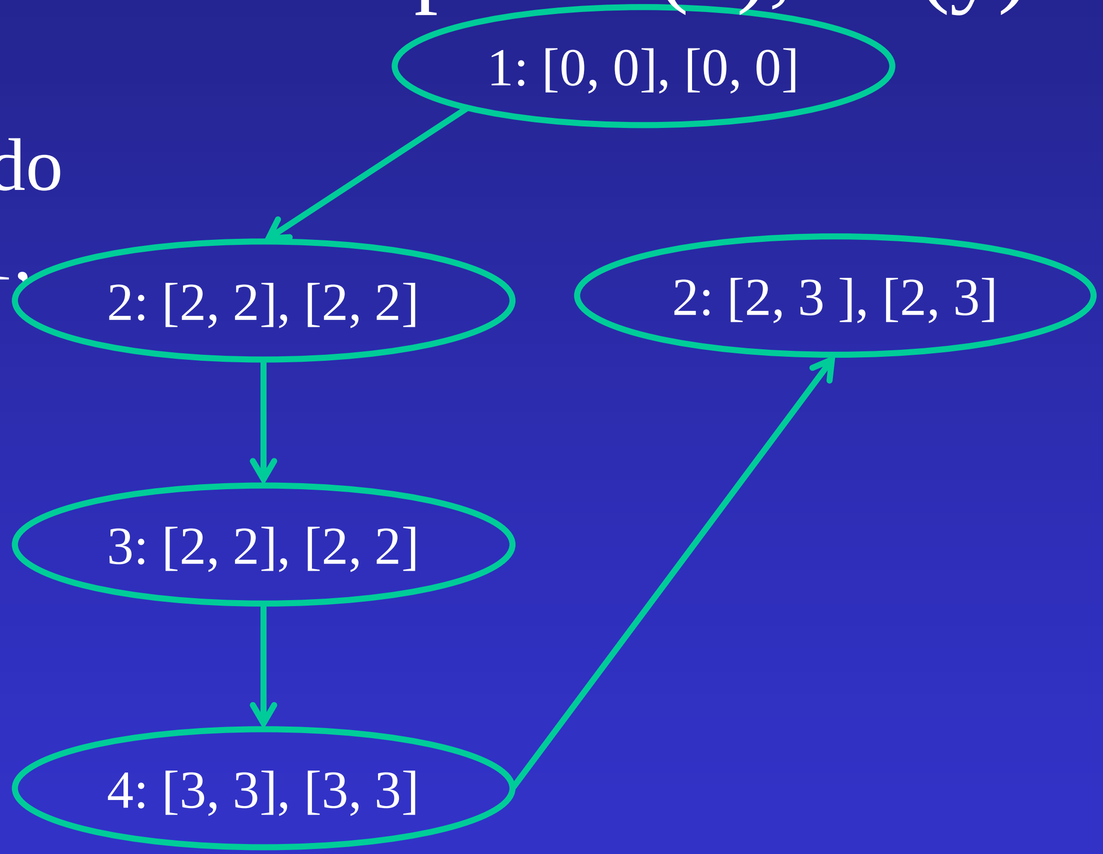
1: x = 2;
2: while true {x > 0} do
3: x = 2* x – 1
4:



Interval Based Abstract Interpretation

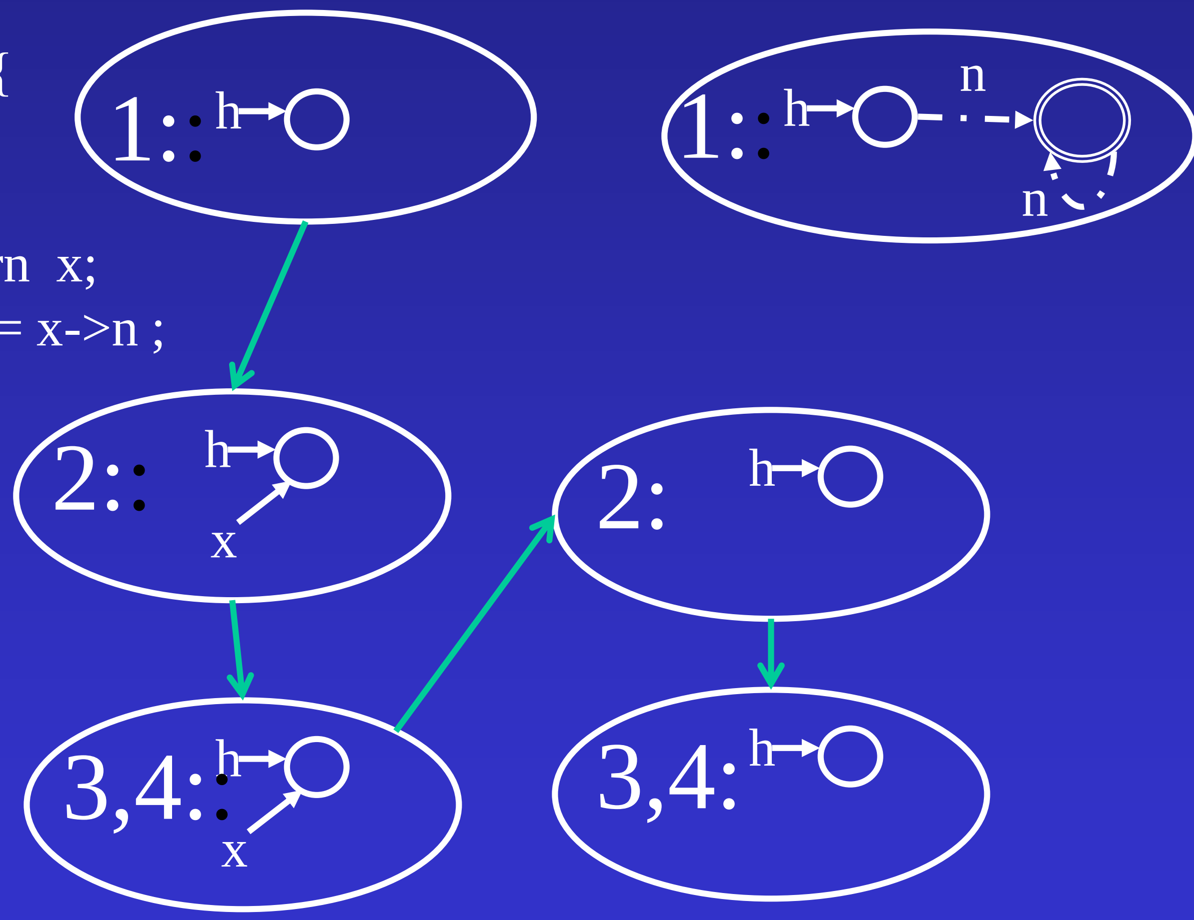
pc: int(x), int(y)

```
1: x = 2, y = 2
2: while true  {x =y} do
        3: x = 2* x – 1,
           y = 2*y -1
       4:
```



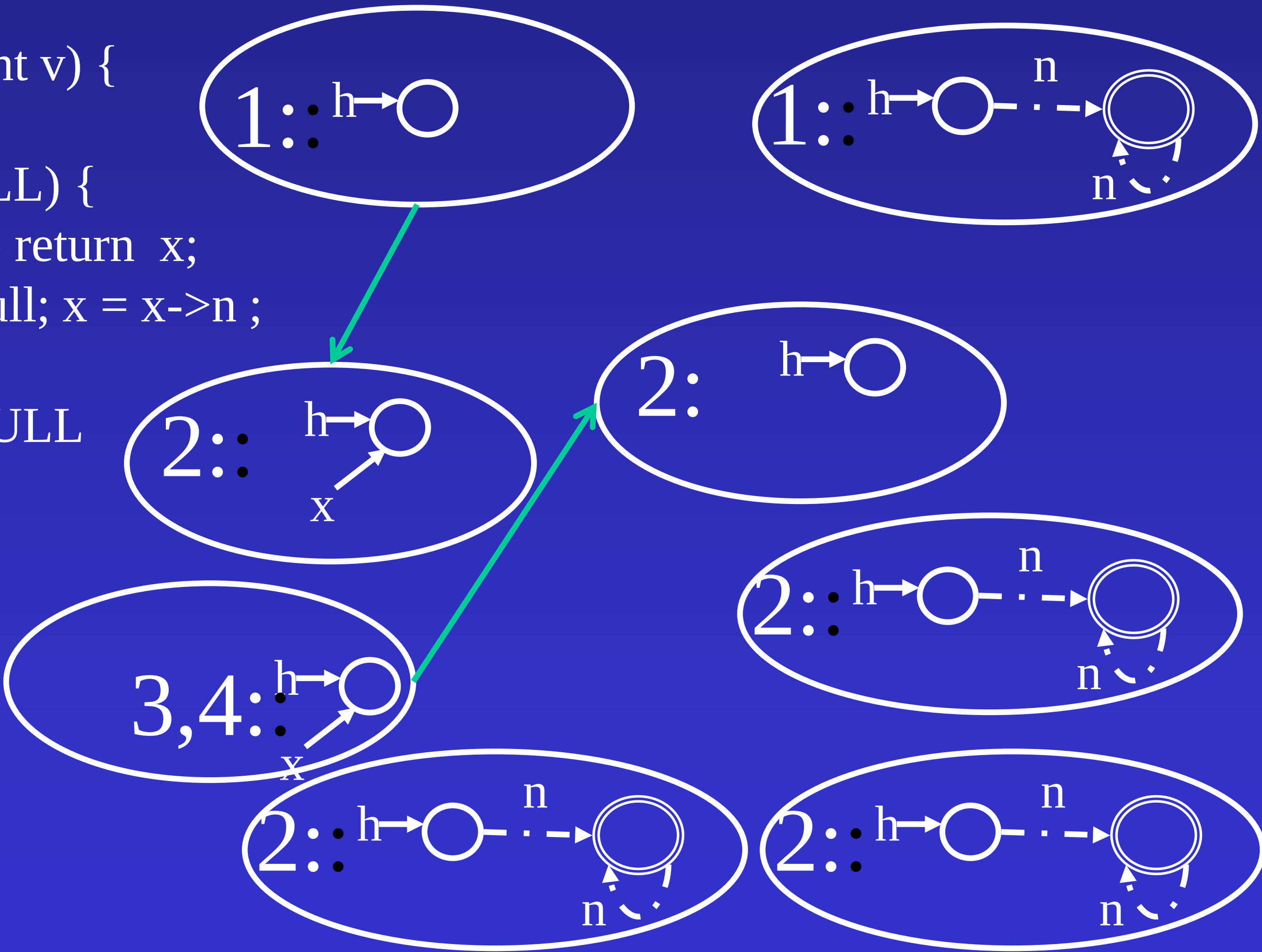
Shape-Based Abstract Interpretation

```
node search(node h, int v) {
   1:  node x = h;
   2:  while (h != NULL) {
      3:  if (x->d == v) return  x;
      4:  assert  x != null; x = x->n ;
  }
  5:   return (node) NULL
```



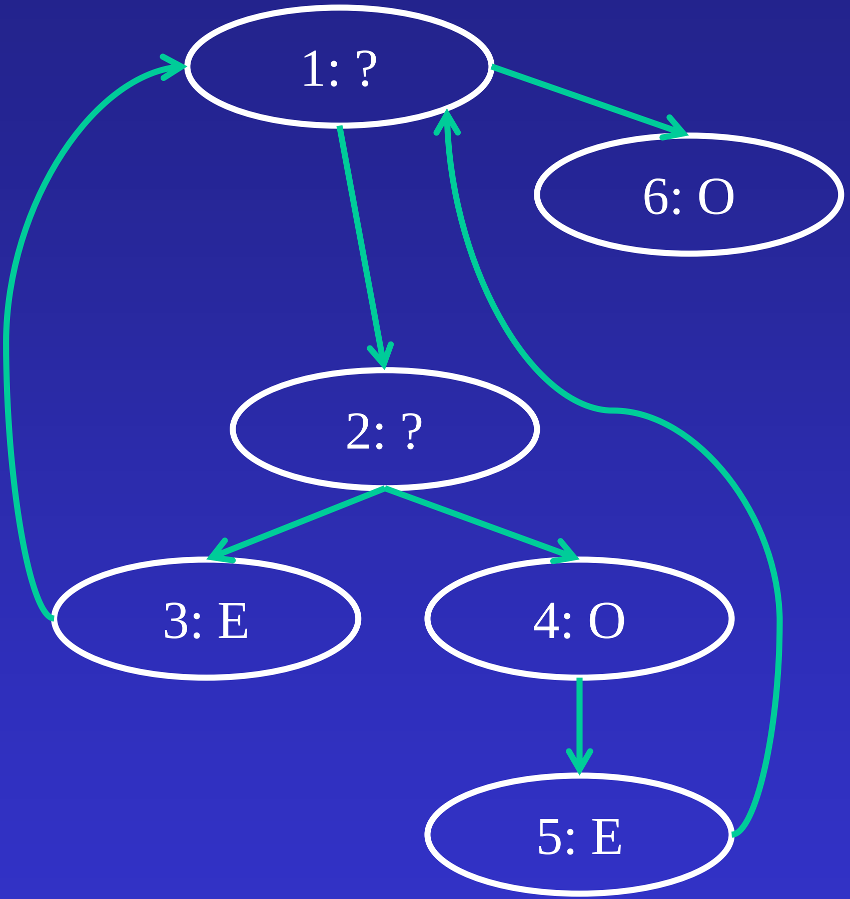
Shape-Based Abstract Interpretation

```
node search(node h, int v) {
   1:  node x = h;
   2:  while (x != NULL) {
      3:  if (x->d == v) return  x;
      4:  assert  x != null; x = x->n ;
  }
  5:   return (node) NULL
```

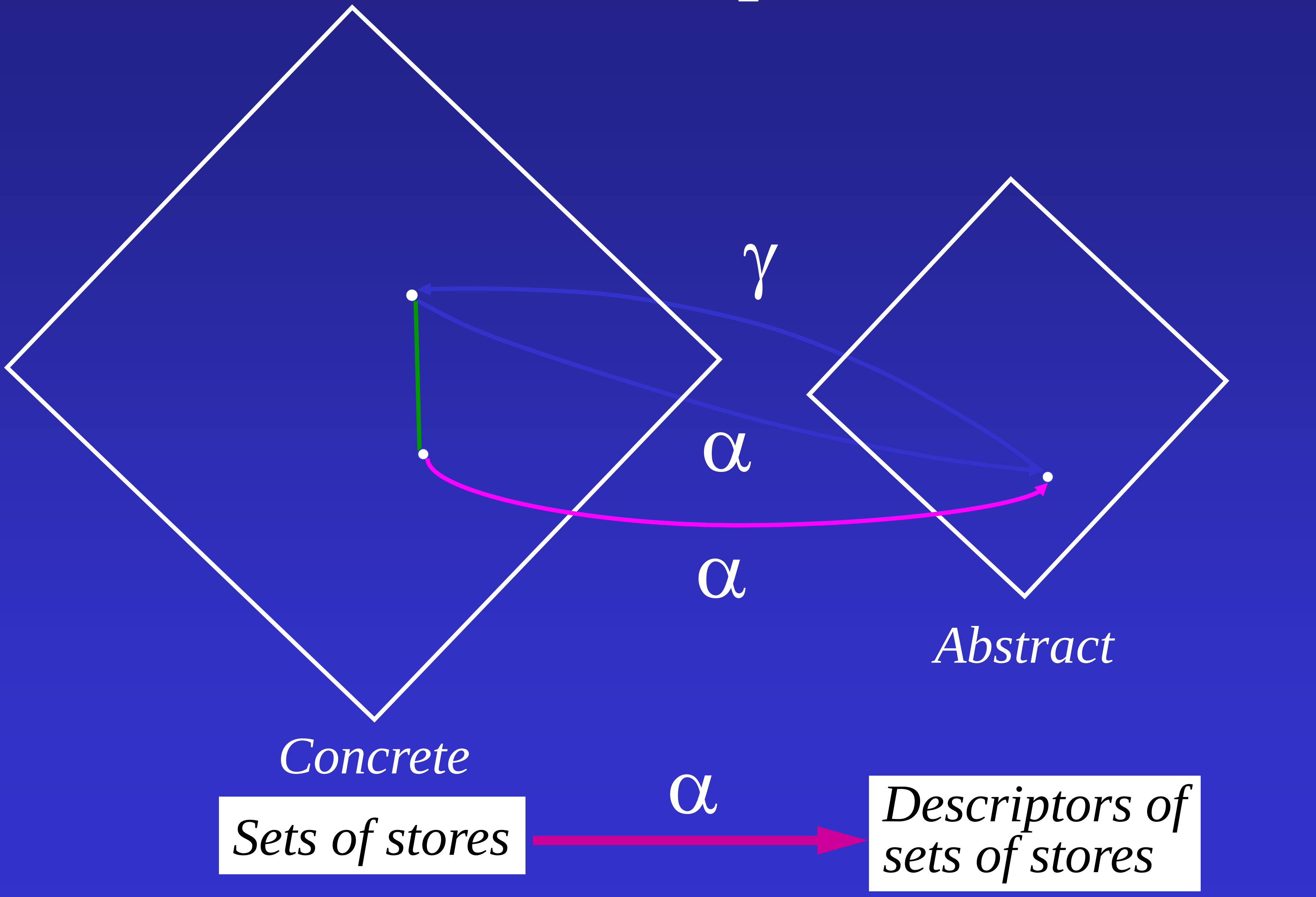


Odd/Even Abstract Interpretation

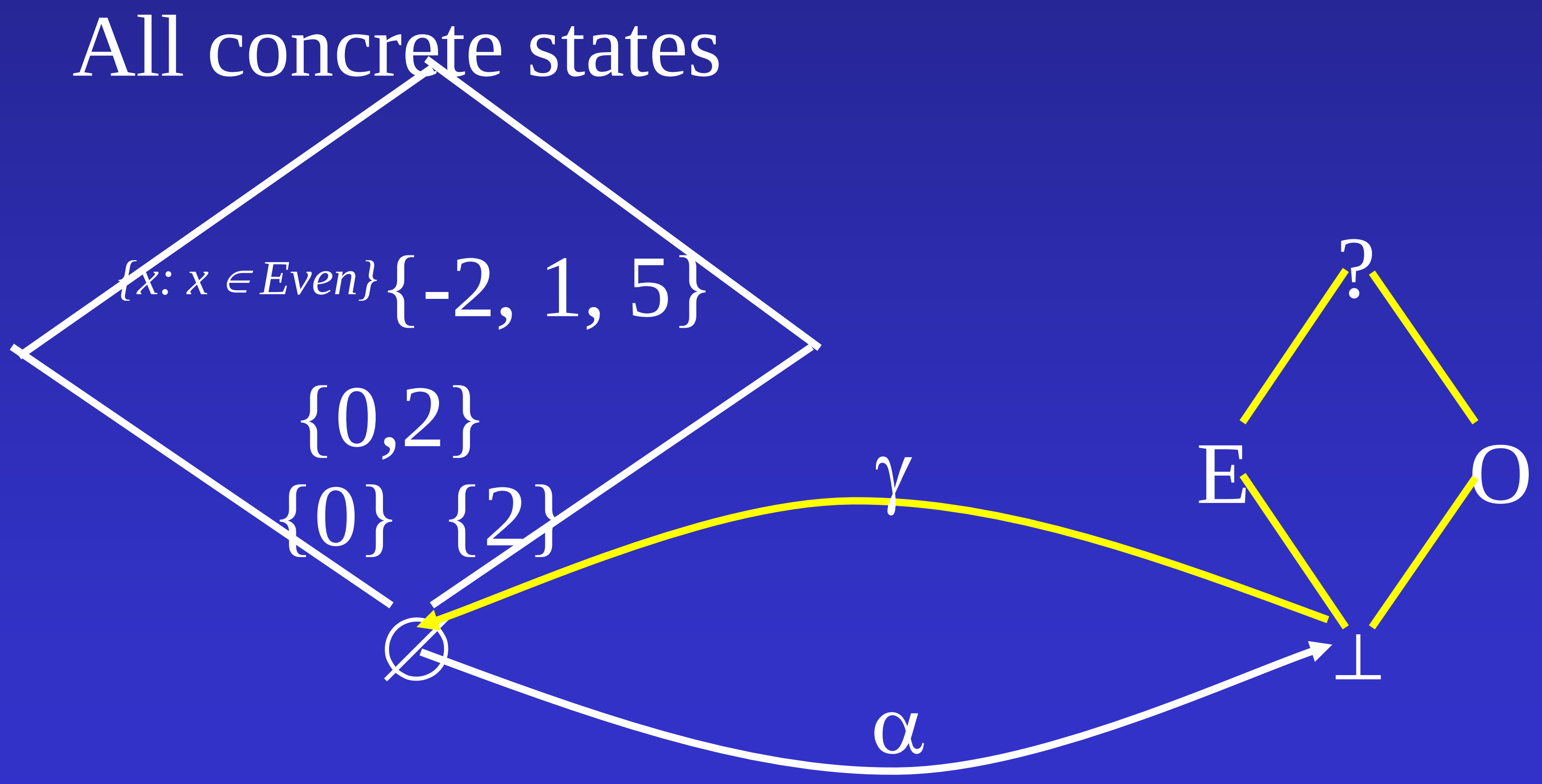
```
1: while  (x !=1)   do {
          2: if   (x %2) == 0
              { 3: x := x / 2; }
               else
             { 4 : x := x * 3 + 1;
           5: assert (x %2 ==0); }
6: }
```



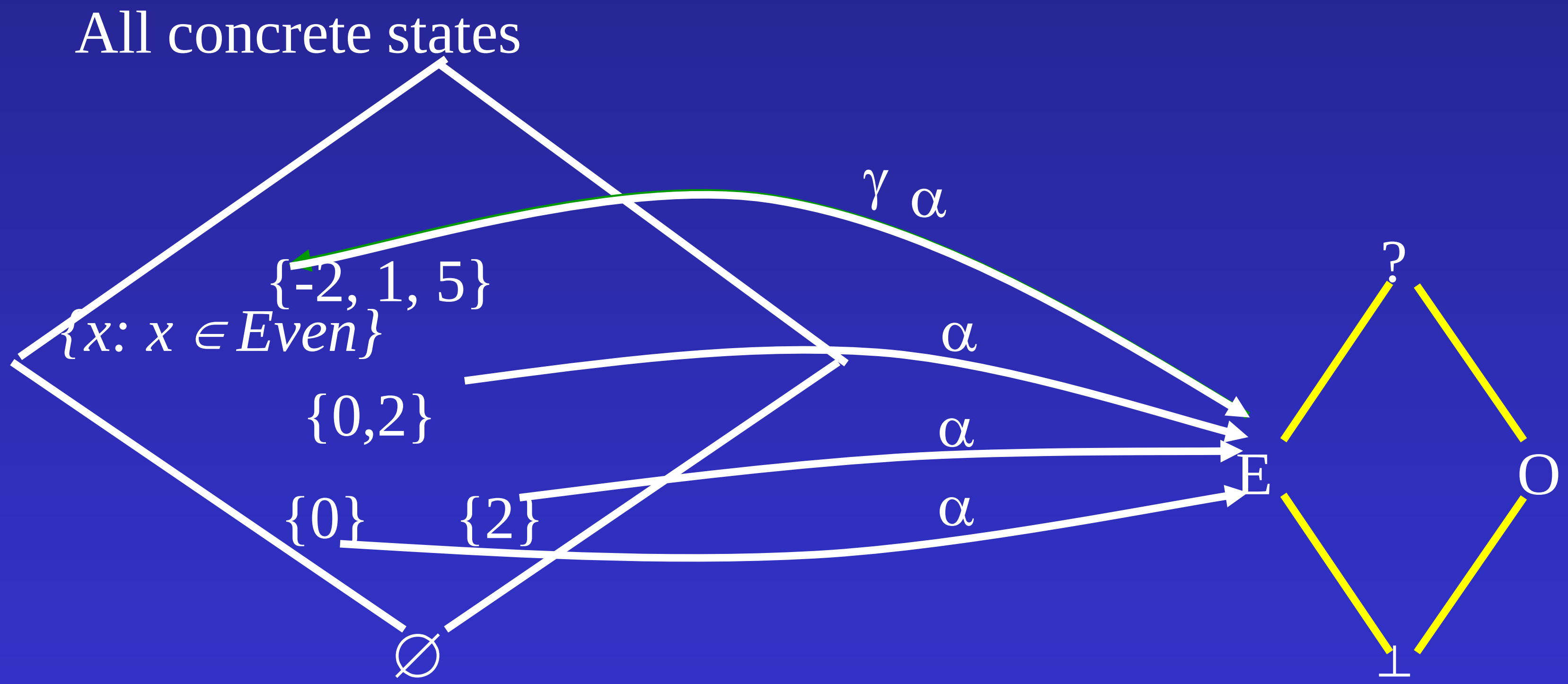
Abstract Interpretation



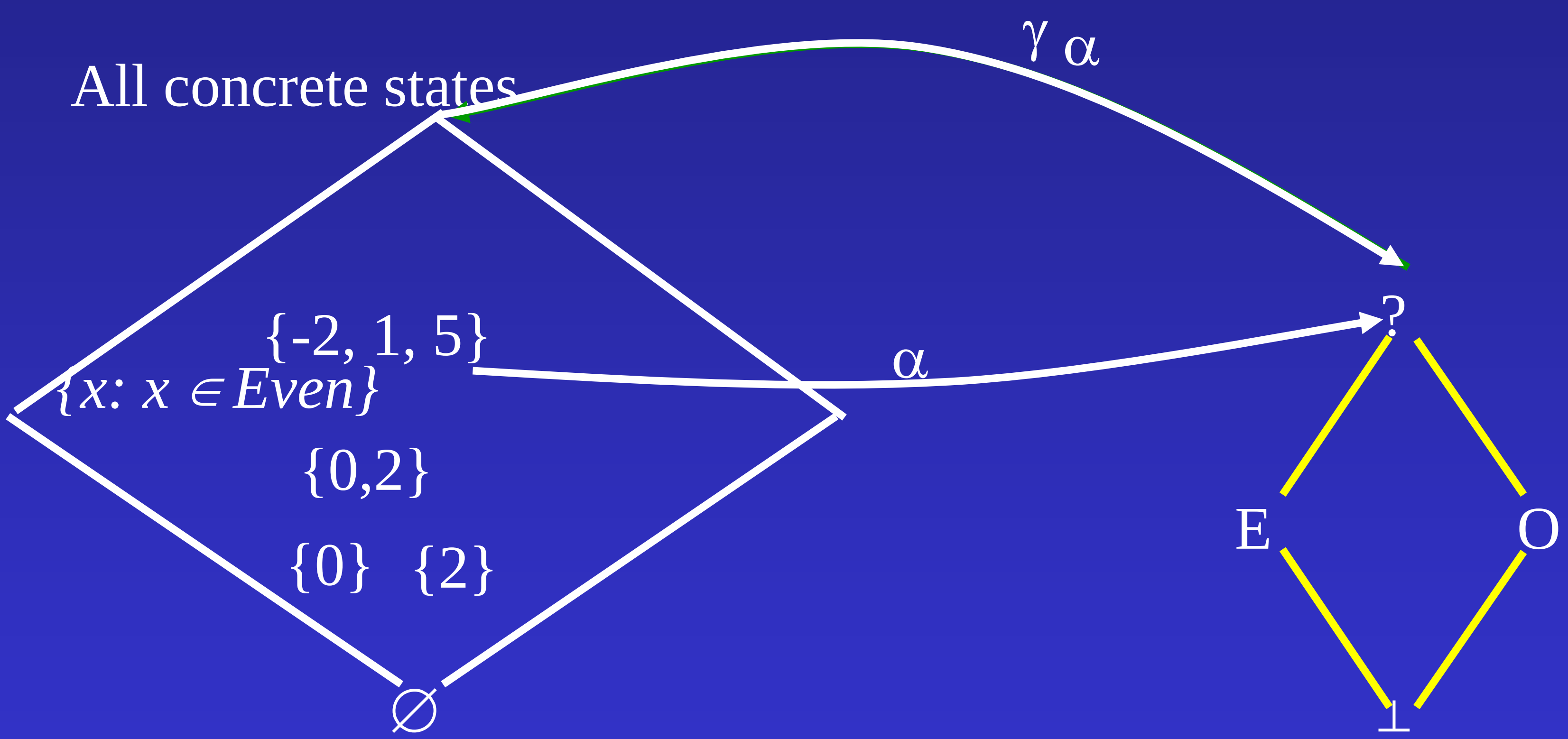
Odd/Even Abstract Interpretation



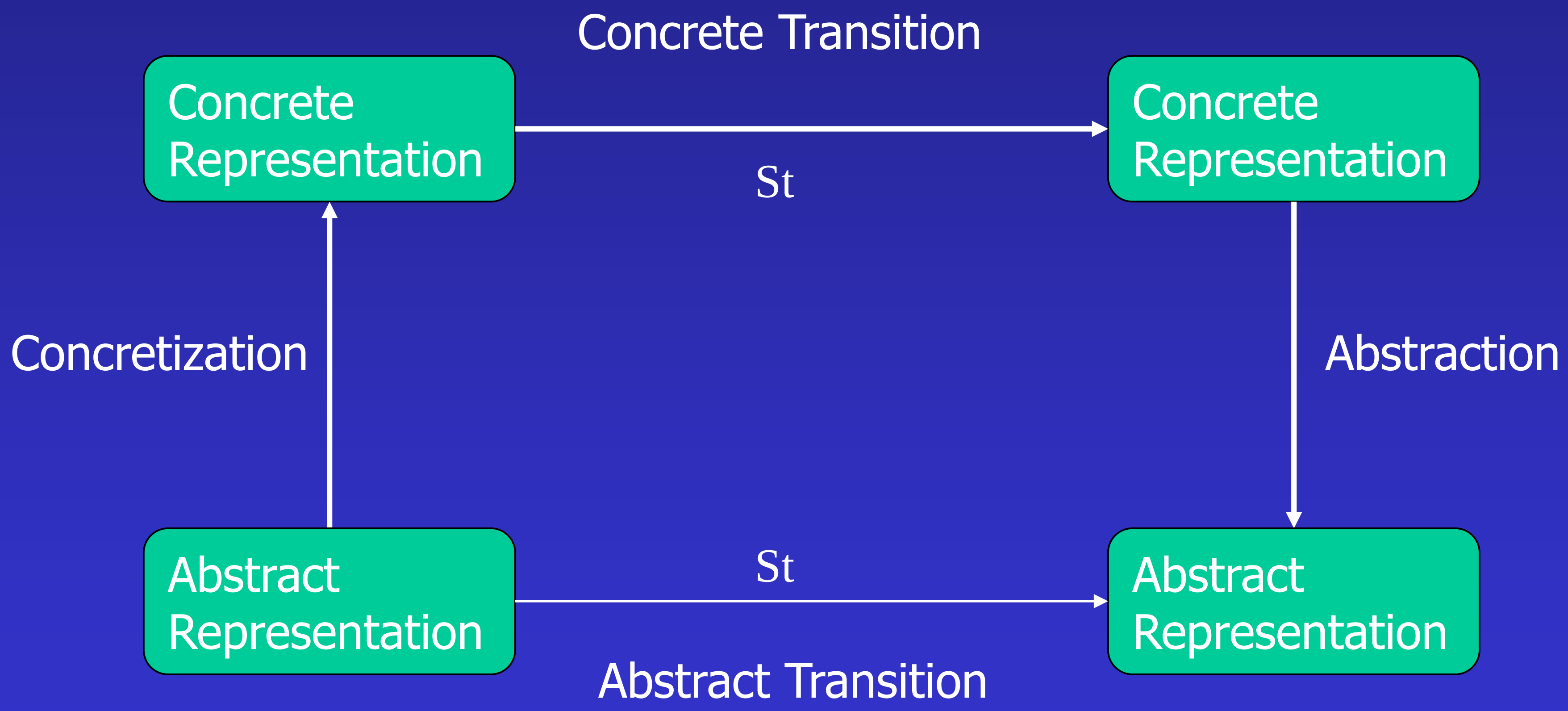
Odd/Even Abstract Interpretation



Odd/Even Abstract Interpretation

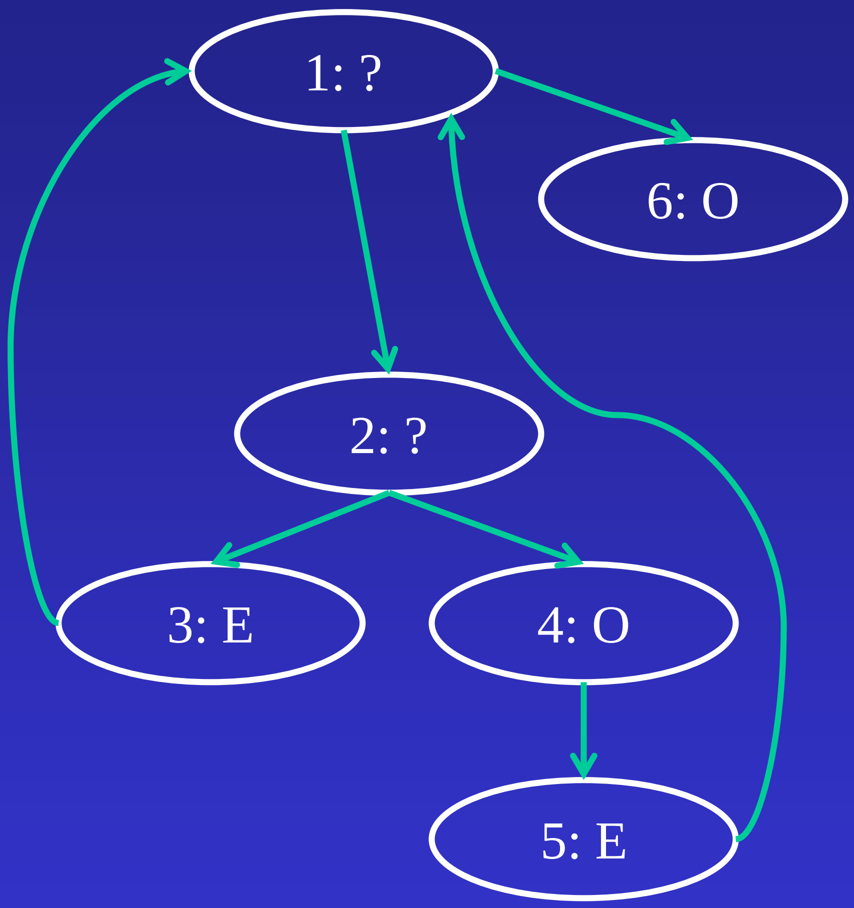


(Best) Abstract Transformer



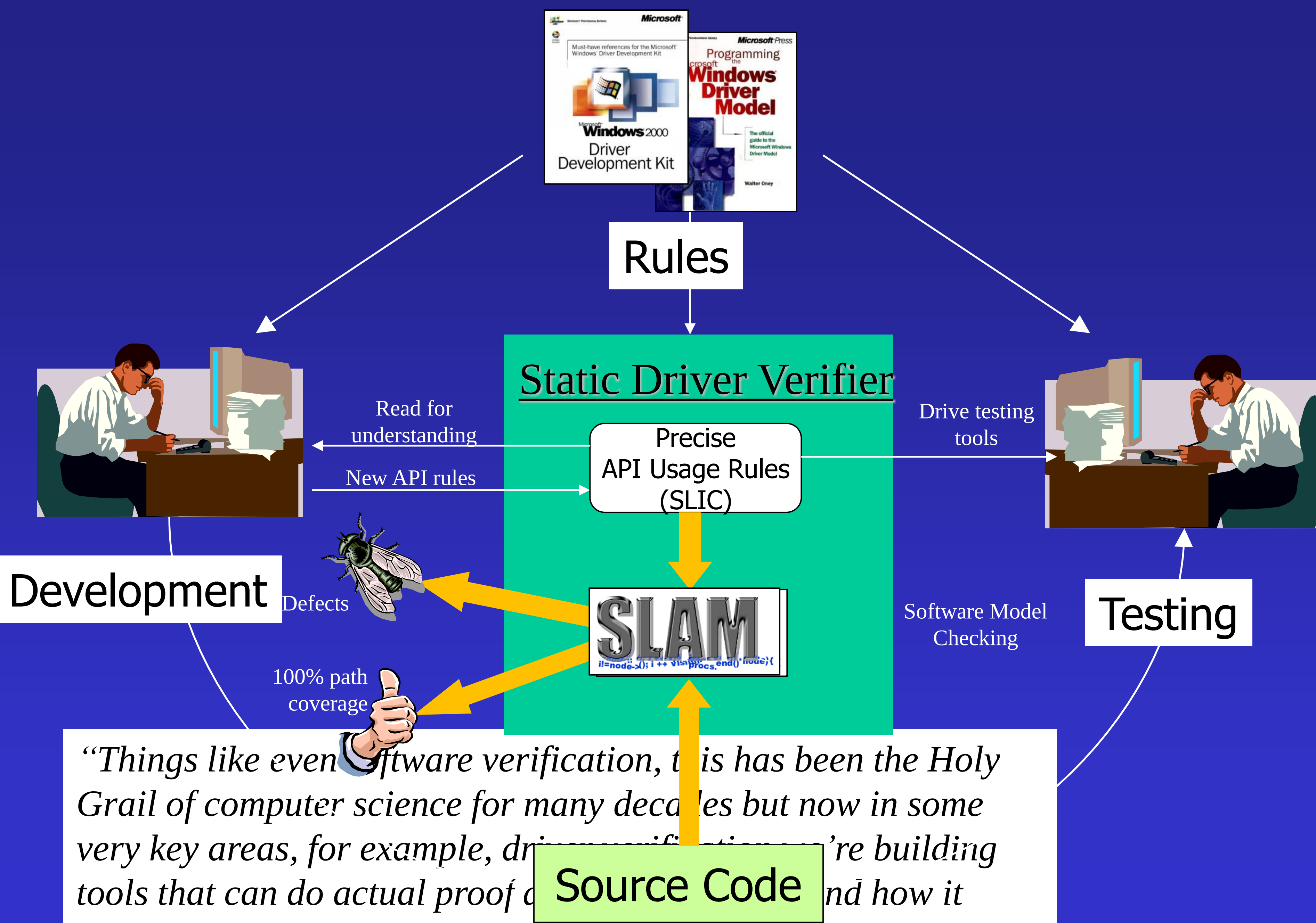
Odd/Even Abstract Interpretation

```
1: while  (x !=1)   do {
        2: if   (x %2) == 0
            { 3: x := x / 2; }
             else
           { 4 : x := x * 3 + 1;
          5: assert (x %2 ==0); }
6: }
```



Summary Abstract Interpretation

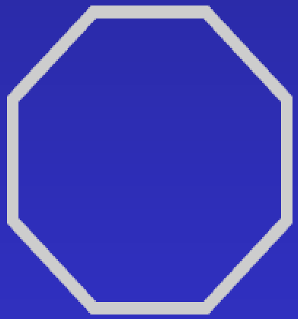
- Conceptual method for building static analyzers
- A lot of techniques:
  - join, meet, widening, narrowing, procedures
- Can be combined with theorem provers



"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof [illegible] and how it works in order to guarantee the reliability" Bill Gates

Success Story: Astrée

- Developed at ENS
- A tool for checking the absence of runtime errors in Airbus flight software



[CC’00] R. Shaham, E.K. Kolodner, S. Sagiv:
Automatic Removal of Array Memory Leaks in Java
[WCRE’2001] A. Miné: The Octagon Abstract Domain
[PLDI’03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival: A static analyzer for large safety-critical software

Success: Panaya
Making ERP easy

- Static analysis to detect the impact of a change for ERP professionals (slicing)
- Developed by N. Dor and Y. Cohen
- Acquired by Infosys

[ISSTA'08] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, D. Weiss: Customization change impact analysis for erp professionals via program slicing

[FSE'10] S. Litvak, N. Dor, R. Bodík, N. Rinetzky, M. Sagiv: Field-sensitive program dependence analysi

Exciting Times for Formal Methods

- Adapted by the Network and System's communities
- The beginning of industry adaption
- New applications
  - Networks
  - Biology
  - Education

Tentative Schedule

Week	Lecture	Recitation	Exercise
1	Overview	No Recitation	No assignment
2	SAT and SMT Solvers	Z3	Graph algorithms with Z3
3	Bounded Model Checking	CBMC	CBMC
4	Concolic Testing	KLEE	KLEE
5	Deductive Verification 1	No Recitation	No assignment
6	Deductive Verification 2	Dafny	Dafny
7	Static Analysis	Apron, Absint	Apron and AbsInt
8	Random Testing	Quickcheck, Randoop, Simullant, Autotest, YETI, GramTest	Use the tools
9	Fuzz Testing	TBD	TBD
10	Mutation Testing	TBD	TBD
11	Unit Testing	TBD	TBD
12	Delta Debugging	TBD	TBD
13	Program Synthesis	TBD	TBD
14	System's Code	TBD	TBD
15	Network and Cloud	TBD	TBD

Course Benefits

- Learn about research which is becoming mature
- Understand the limits of formal methods