Program Analysis and Verification

0368-4479

Noam Rinetzky

Lecture 1: Introduction & Overview

Slides credit: Tom Ball, Dawson Engler, Roman Manevich, Erik Poll, Mooly Sagiv, Jean Souyris, Eran Tromer, Avishai Wool, Eran Yahav

Admin

- Lecturer: Noam Rinetzky
  - *maon@cs.tau.ac.il*
  - *http://www.cs.tau.ac.il/~maon*

- 13 Lessons
  - Thursday, 13:00-16:00, Schreiber 8

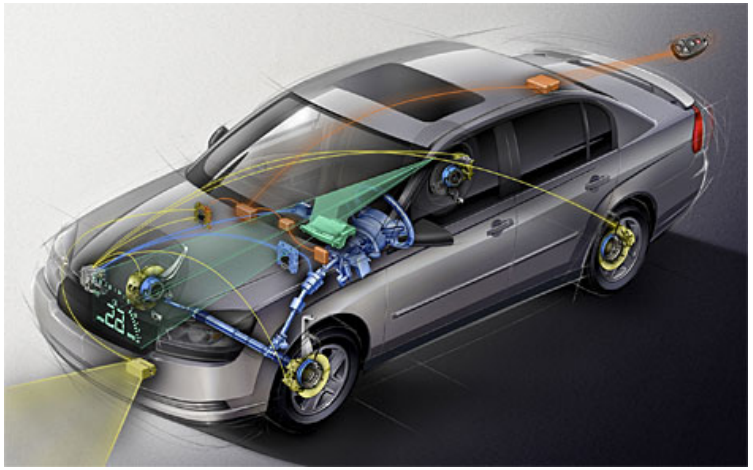
Grades

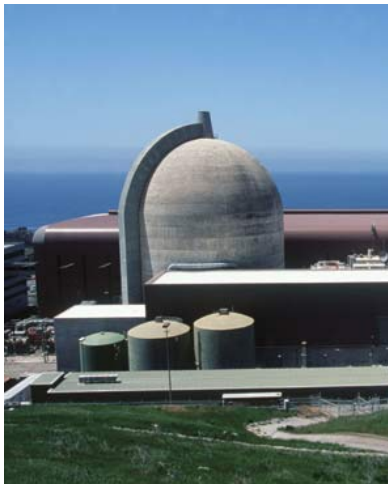
- 2-3 theoretical assignments (35%)
- 1 practical assignment (15%)
- Final project (50%)
  - In groups of 1-2

Today

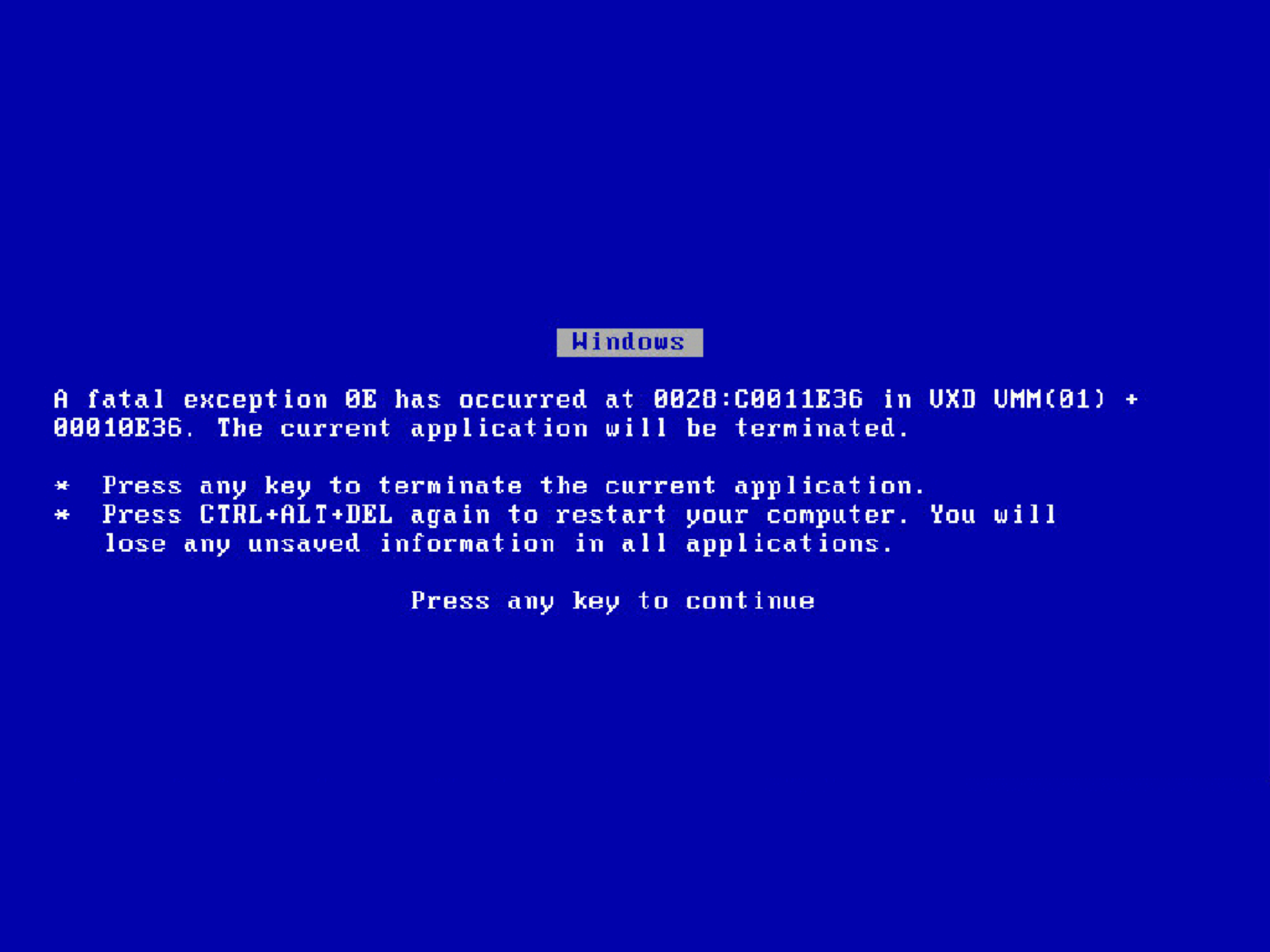
- Motivation
- Introduction

Software is Everywhere





Unreliable Software is Everywhere


Windows
A fatal exception 0E has occurred at 0028:C0011E36 in VXD VMM(01) +
00010E36. The current application will be terminated.
* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will
lose any unsaved information in all applications.
Press any key to continue

30GB Zunes all over the world fail en masse

December 31, 2008

Zune bug

```
 1 while (days > 365) {
 2   if (IsLeapYear(year)) {
 3     if (days > 366) {
 4       days -= 366;
 5       year += 1;
 6     }
 7   } else {
 8     days -= 365;
 9     year += 1;
10   }
11 }
```

December 31, 2008

Zune bug

```
 1 while (366 > 365) {
 2   if (IsLeapYear(2008)) {
 3     if (366 > 366) {
 4       days -= 366;
 5       year += 1;
 6     }
 7   } else {
 8     days -= 365;
 9     year += 1;
10   }
11 }
```

December 31, 2008

Suggested solution: wait for tomorrow

Patriot missile failure

On the night of the 25th of February, 1991, a Patriot missile system operating in Dhahran, Saudi Arabia, failed to track and intercept an incoming Scud. The Iraqi missile impacted into an army barracks, killing 28 U.S. soldiers and injuring another 98.

February 25, 1991

Patriot bug – rounding error

- Time measured in 1/10 seconds
- Binary expansion of 1/10:
  0.0001100110011001100110011001100....
- 24-bit register

  0.00011001100110011001100
- error of
  - 0.0000000000000000000011001100... binary, or ~0.000000095 decimal
- After 100 hours of operation error is
  0.000000095×100×3600×10=0.34
- A Scud travels at about 1,676 meters per second, and so travels more than half a kilometer in this time

Suggested solution: reboot every 10 hours

Toyota recalls 160,000 Prius hybrid vehicles

Programming error can activate all warning lights, causing the car to think its engine has failed

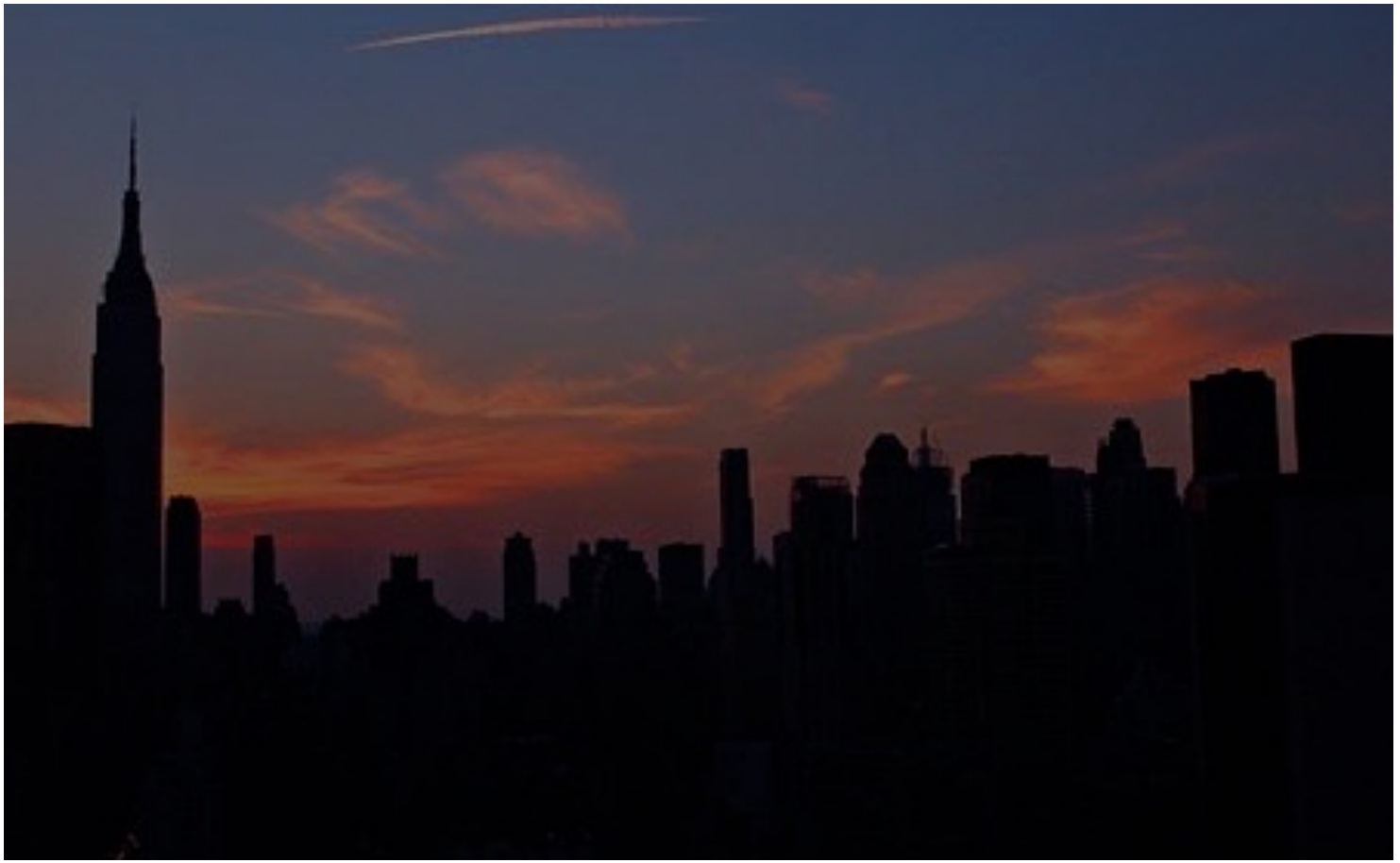
October 2005

Therac-25 leads to 3 deaths and 3 injuries

Software error exposes patients to radiation overdose (100X of intended dose)

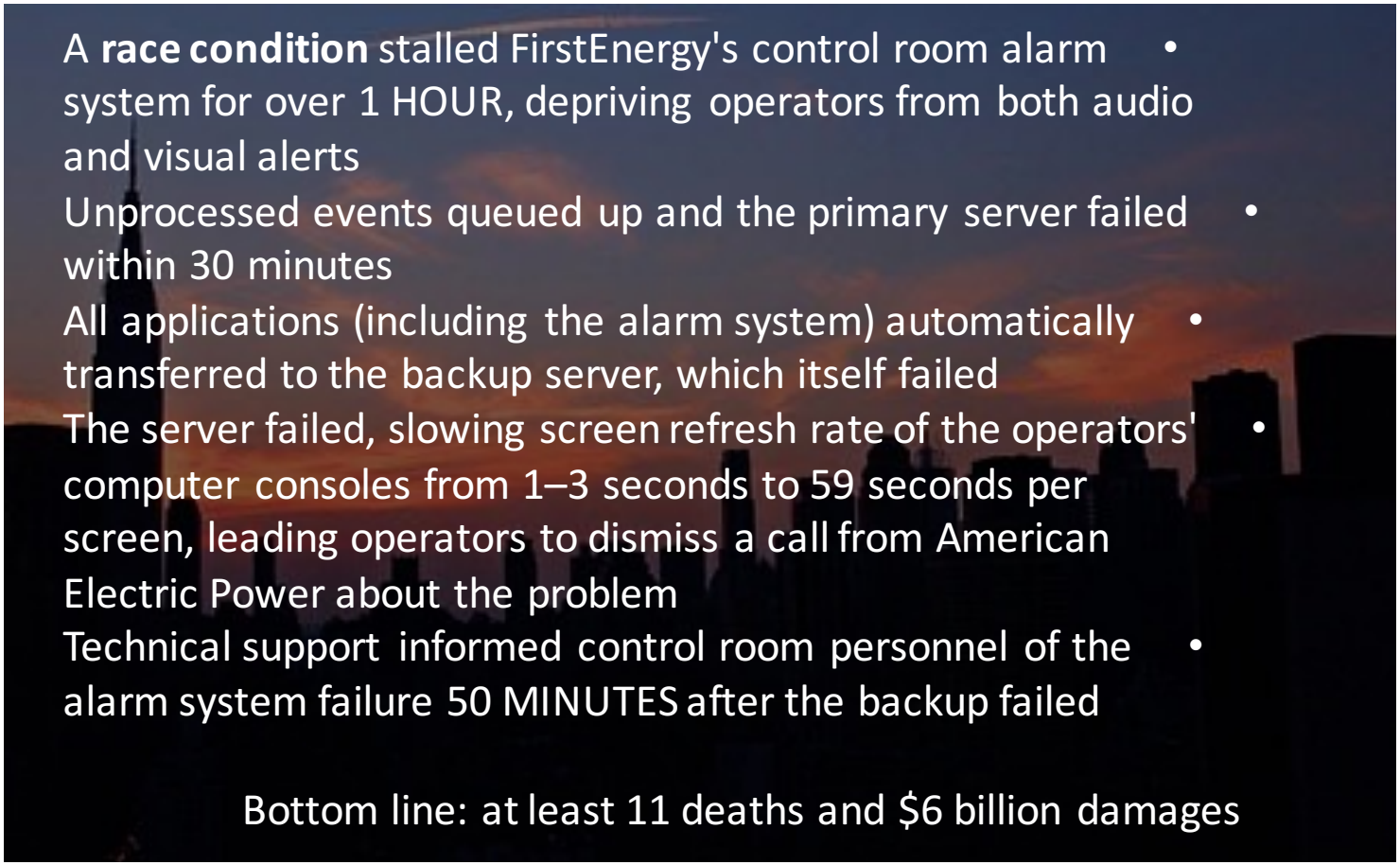
1985 to 1987

Northeast Blackout



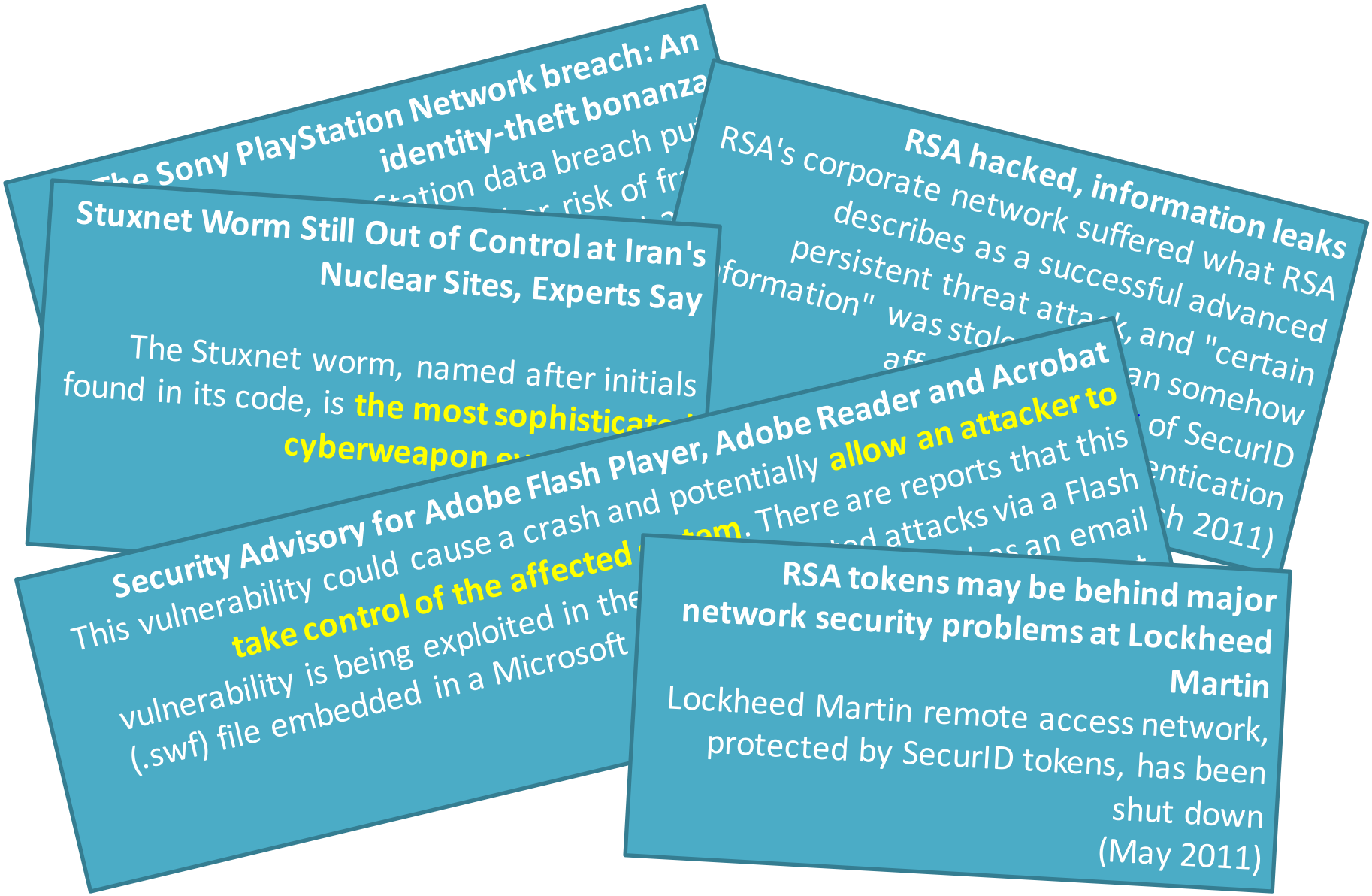
14 August, 2003

Northeast Blackout



14 August, 2003

Unreliable Software is Exploitable



Billy Gates why do you make this possible ? Stop making money and fix your software!!

(W32.Blaster.Worm)

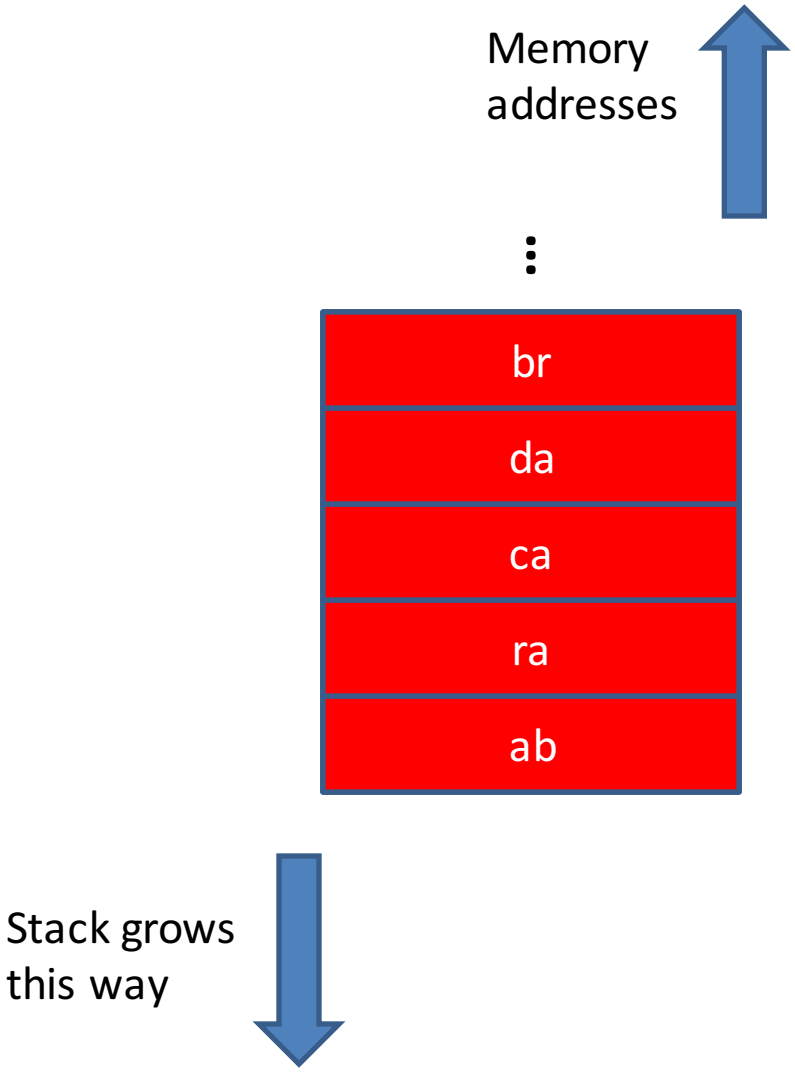
August 13, 2003

Windows exploit(s)

Buffer Overflow

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

 ./a.out abracadabra
Segmentation fault
```



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");
}
```

Input Validation

evil input

Application

1234567890123456

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

Access Granted.

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

Boeing's 787 Vulnerable to Hacker Attack

security vulnerability in onboard computer networks could allow passengers to access the plane's control systems

January 2008

Apple's SSL/TLS bug (22 Feb 2014)

- Affects iOS (probably OSX too)

```
static OSStatus
SSLVerifySignedServerKeyExchange(…)
{
    OSStatus        err;

    ...
    if ((err = SSLHashSHA1.f1(...)) != 0)
goto fail;
    if ((err = SSLHashSHA1.f2(...)) != 0)
goto fail;
goto fail;
    if ((err = SSLHashSHA1.f3(...)) != 0)
goto fail;
    ...

fail:
SSLFreeBuffer(&signedHashes);
SSLFreeBuffer(&hashCtx);
return err;
                                    }
```

(Quoted from http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c)

Shellshock bug (24/Sep/2014)

GNU Bash through 4.3 processes trailing strings after function definitions in the values of environment variables, which allows remote attackers to execute arbitrary code via a crafted environment

What can we do about it?

What can we do about it?

I just want to say LOVE YOU SAN!!soo much

Billy Gates why do you make this possible ? Stop making money and fix your software!!

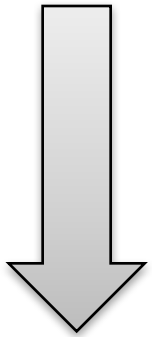
(W32.Blaster.Worm / *Lovesan worm*)

August 13, 2003

What can we do about it?

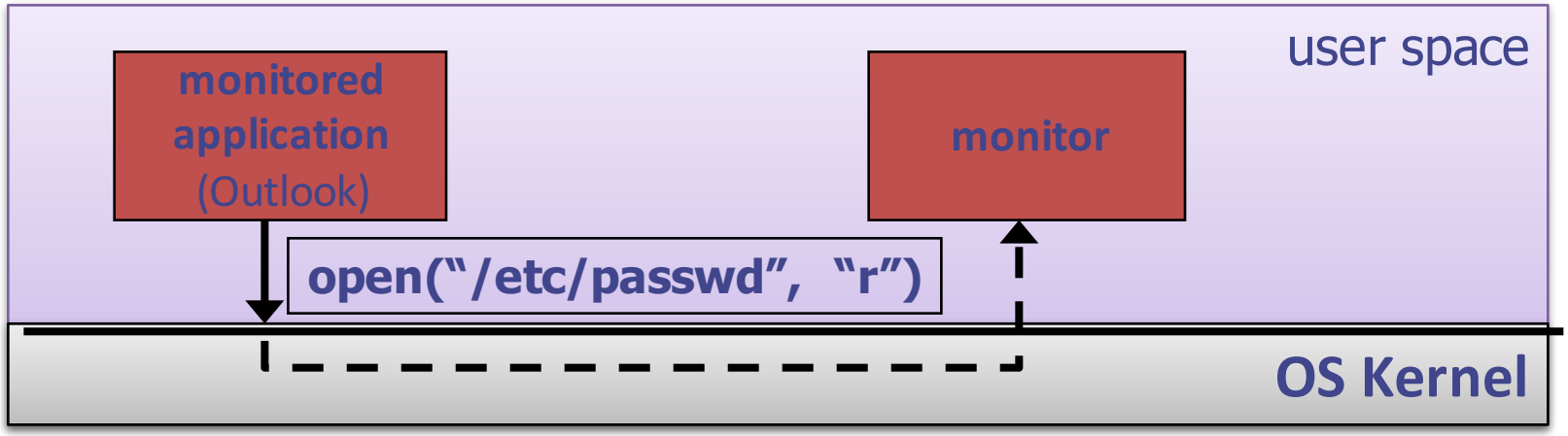
- Monitoring
- Testing
- Static Analysis
- Formal Verification
- Specification

Run time



Design Time

Monitoring (e.g., for security)



- StackGuard
- ProPolice
- PointGuard
- Security monitors (ptrace)

Testing

- build it

Testing

- build it; try it on a some inputs
  - printf (“x == 0 => should not get that!”)

Testing

- Valgrind memory errors, race conditions, taint analysis
  - Simulated CPU
  - Shadow memory

```
Invalid read of size 4
   at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40B07FF4: read_png_image(QImageIO *) (kernel/qpngio.cpp:326)
   by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
 Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

Testing

- Valgrind memory errors, race conditions
- Parasoft Jtest/Insure++ memory errors + visualizer, race conditions, exceptions …
- IBM Rational Purify memory errors
- IBM PureCoverage detect untested paths
- Daikon dynamic invariant detection

Testing

- Useful and challenging
  - Random inputs
  - Guided testing (coverage)
  - Bug reproducing

- But …
  - Observe some program behaviors
  - What can you say about other behaviors?

Testing is not enough

- Observe some program behaviors
- What can you say about other behaviors?

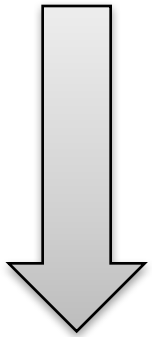
- Concurrency makes things worse

- Smart testing is useful
  - requires the techniques that we will see in the course

What can we do about it?

- Monitoring
- Testing
- Static Analysis
- Formal Verification
- Specification

Run time



Design Time

Program Analysis & Verification

```
x = ?
if (x > 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

?

- Is assertion true?

Program Analysis & Verification

```
y = ?; x = y * 2
if (x % 2 == 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

?

- Is assertion true? Can we prove this? Automatically?
- Bad news: problem is generally undecidable

Formal verification

- Mathematical model of software
  - ρ: Var $\rightarrow \mathbb{Z}$
  - ρ= [x$\rightarrow$0, y$\rightarrow$1]
- Logical specification
  - $\{ 0 < x \} = \{\rho \in \text{State} \mid 0 < \rho(x) \}$
- Machine checked formal proofs

$$\{ 0 < x \wedge y = x \} \rightarrow \{ 0 < y \}$$

$$\frac{\{ 0 < x \}\ \text{y:= x}\ \{ 0 < x \wedge y = x \} \qquad \{ 0 < y \}\ \text{y:= y+1}\ \{ 1 < y \}}{\{\ ?\ \}\ \text{y:= x ; y:=y+1}\ \{ 1 < y \}}$$

Formal verification

- Mathematical model of software
  - State = Var ➔ Integer
  - S = [x➔0, y➔1]
- Logical specification
  - { *0 < x* } = { S ε State | 0 < S(x) }
- Machine checked formal proofs

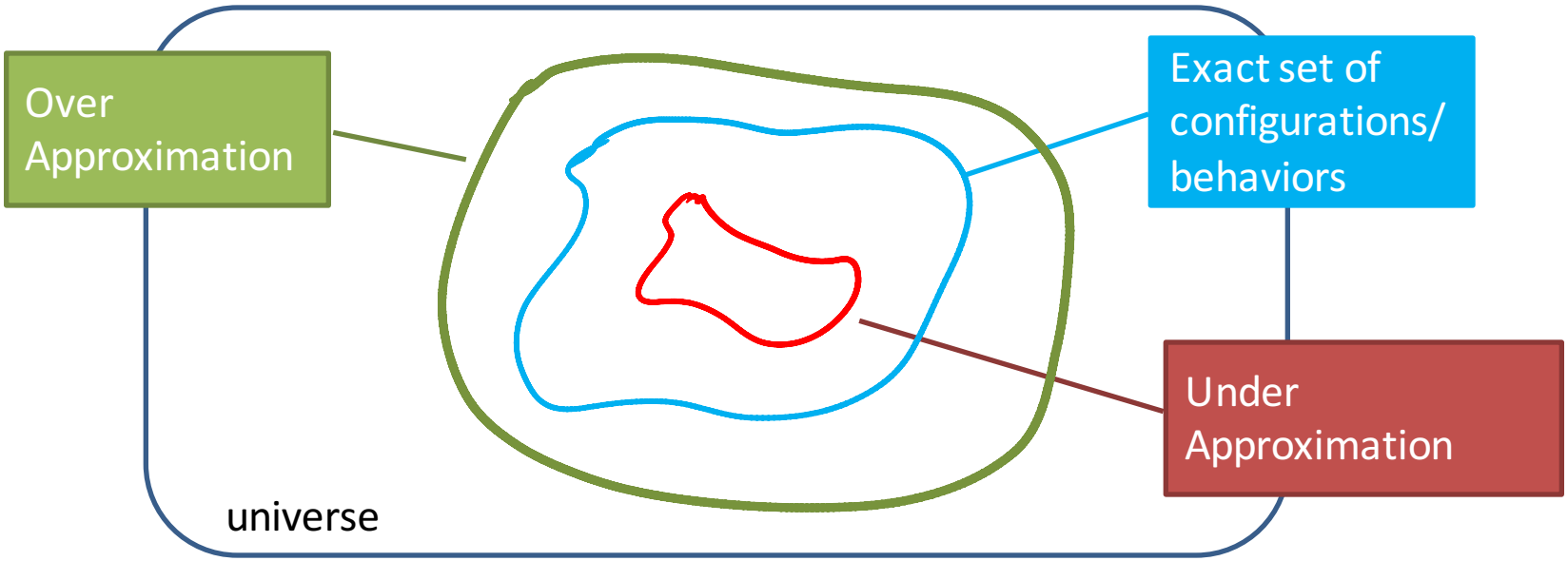
$$\frac{\{ P \}\ \text{stmt1}\ \{ Q' \} \qquad Q' \rightarrow P' \qquad \{ P' \}\ \text{stmt2}\ \{ Q \}}{\{ P \}\ \text{stmt1; stmt2}\ \{ Q \}}$$

Program Verification

```
{true}
y = ?; x = 2 * y;
{x = 2 * y}
if (x % 2 == 0) {
  {x = 2 * y}
  y = 42;
  { ∃z. x = 2 * z ∧ y = 42 }
}  else {
   { false }
   y  = 73;
   foo();
   { false }
}
{ ∃z. x = 2 * z ∧ y = 42 }
assert (y == 42);
```

- Is assertion true? Can we prove this? Automatically?
- Can we prove this manually?

Central idea: use approximation



Program Verification

```
{true}
y = ?; x = 2 * y;
{x = 2 * y}
if (x % 2 == 0) {
  {x = 2 * y}
  y = 42;
  { ∃z. x = 2 * z ∧ y = 42 }
}  else {
   { false }
   y  = 73;
   foo();
   { false }
}
{ ∃z. x = 2 * z ∧ y = 42 } {x = ? ∧ y = 42 } { x = 4 ∧ y = 42 }
assert (y == 42);
```

- Is assertion true? Can we prove this? Automatically?
- Can we prove this manually?

L4.verified [Klein+,'09]

- Microkernel
  - IPC, Threads, Scheduling, Memory management
- Functional correctness (using Isabelle/HOL)
  - + No null pointer de-references.
  - + No memory leaks.
  - + No buffer overflows.
  - + No unchecked user arguments
  - + …
- Kernel/proof co-design
  - Implementation - 2.5 py (8,700 LOC)
  - Proof – 20 py (200,000 LOP)

Static Analysis

- Lightweight formal verification
- Formalize software behavior in a mathematical model (semantics)
- Prove (selected) properties of the mathematical model
  - Automatically, typically with approximation of the formal semantics

Why static analysis?

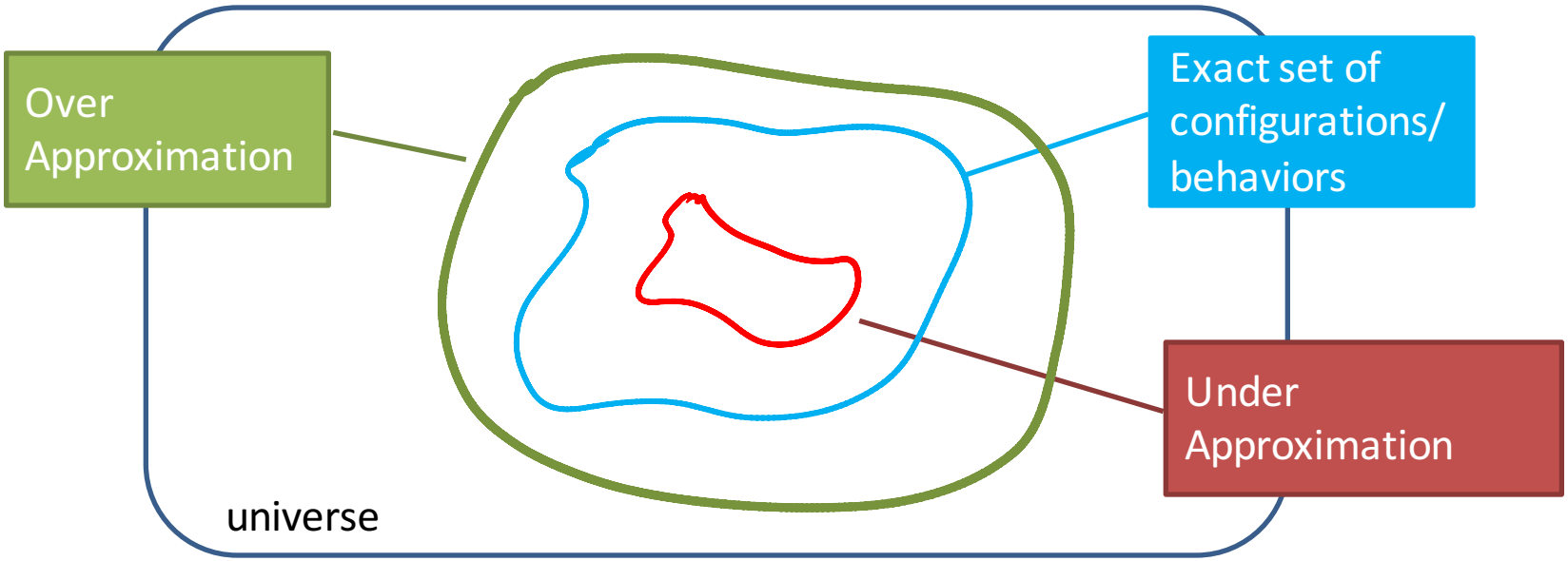
- Some errors are hard to find by testing
  - arise in unusual circumstances/uncommon execution paths
    - buffer overruns, unvalidated input, exceptions, ...
  - involve non-determinism
    - race conditions
- Full-blown formal verification too expensive

Is it at all doable?

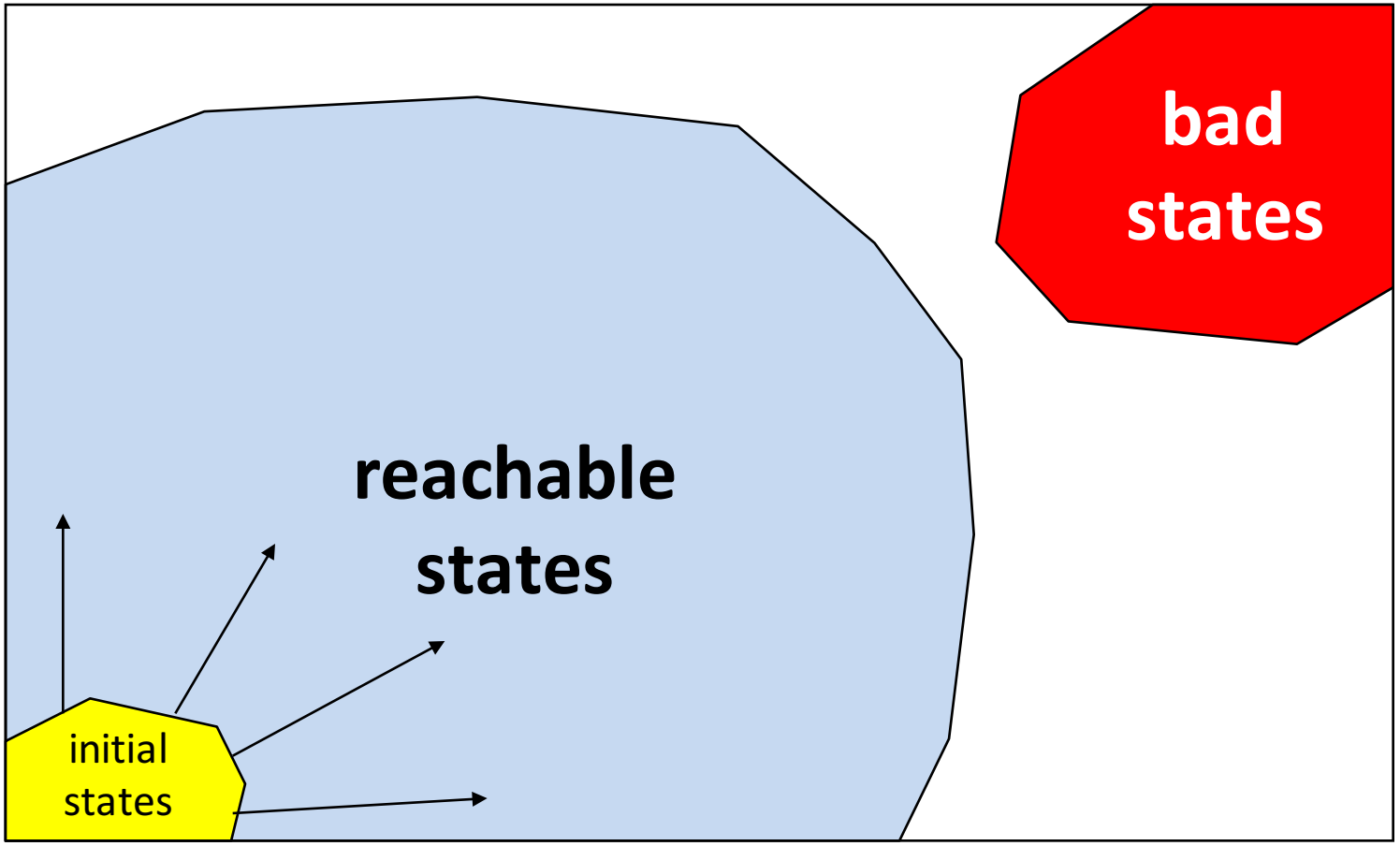
```
x = ?
if (x > 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

Bad news: problem is generally undecidable

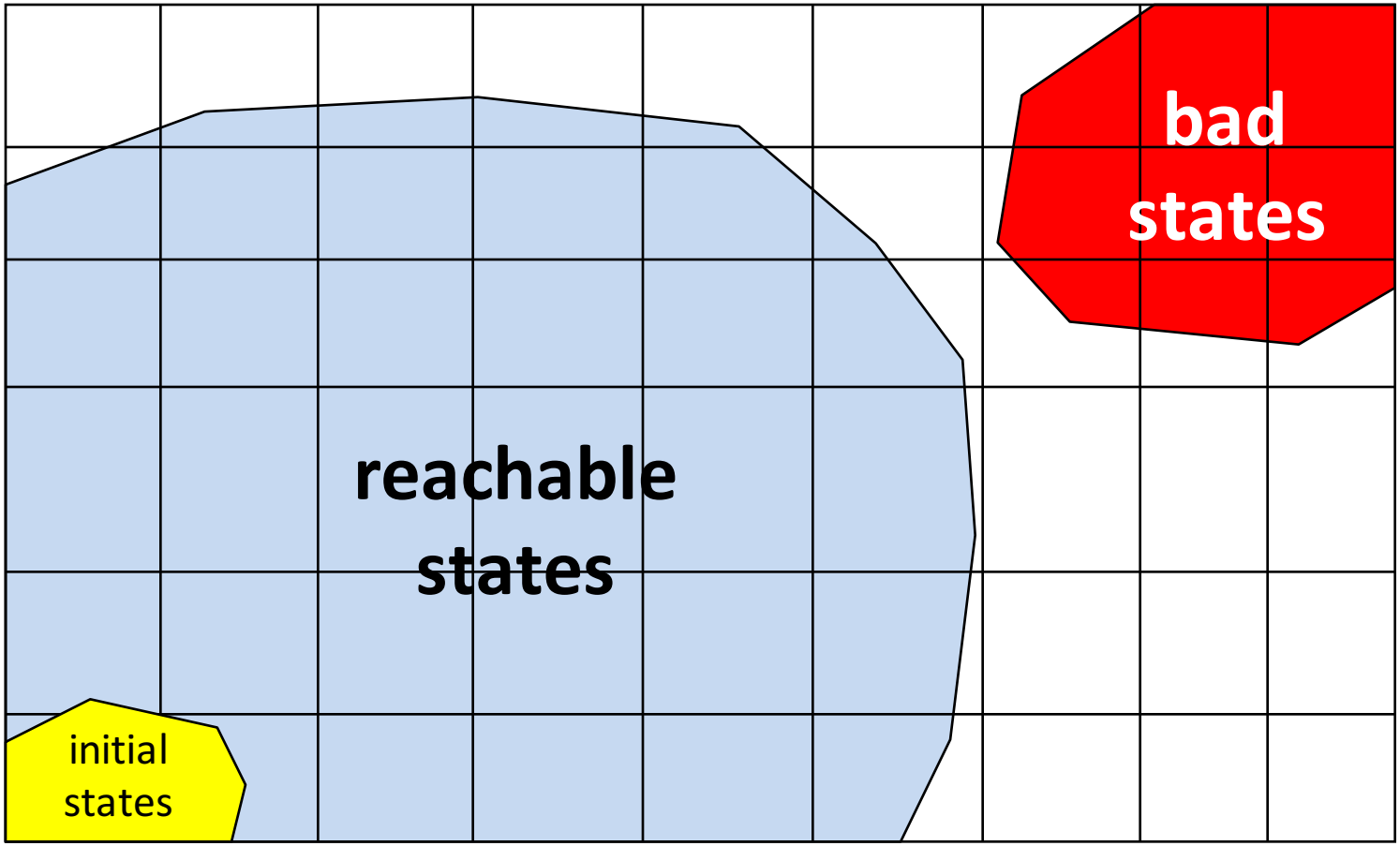
Central idea: use approximation



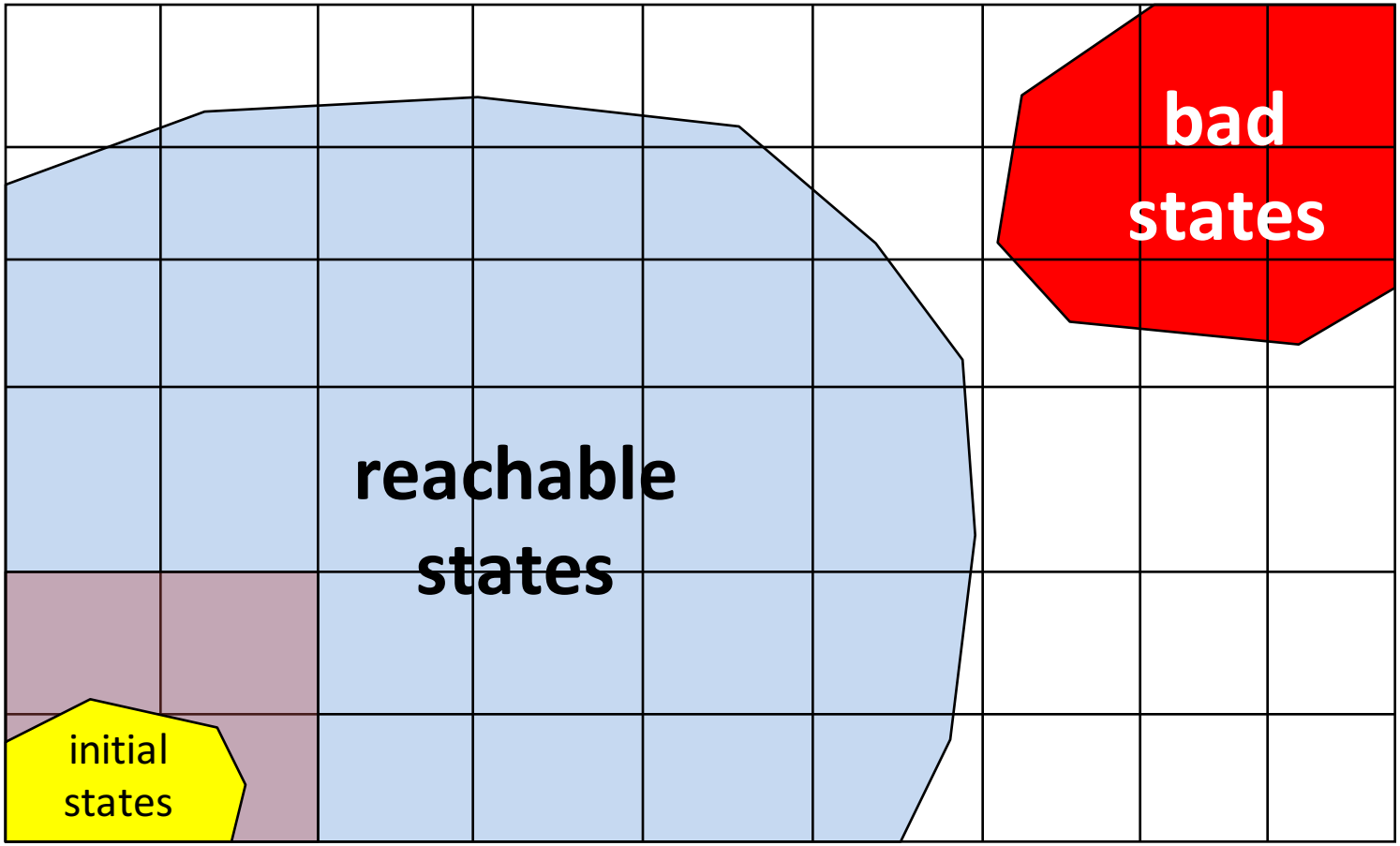
Goal: exploring program states



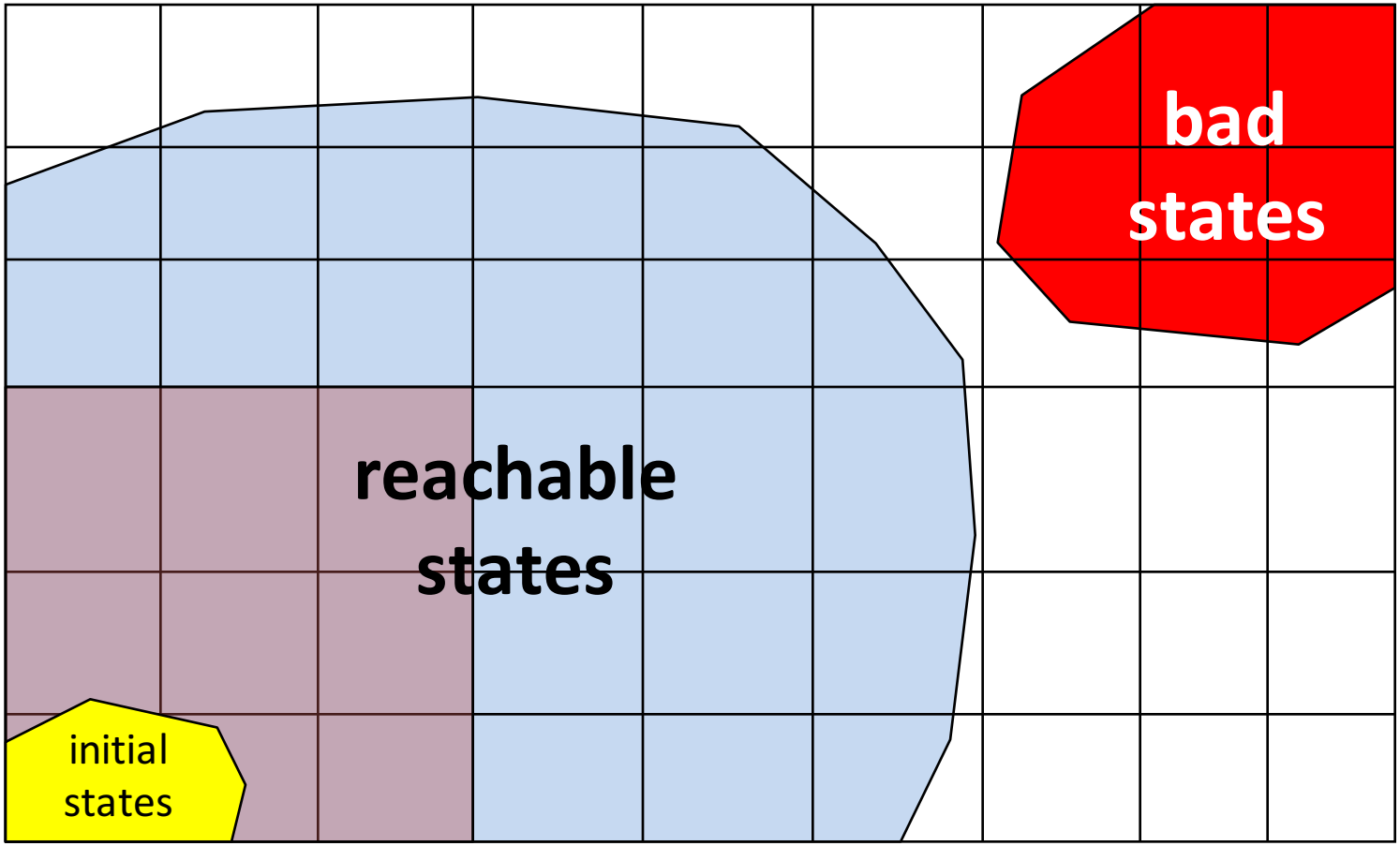
Technique: explore abstract states



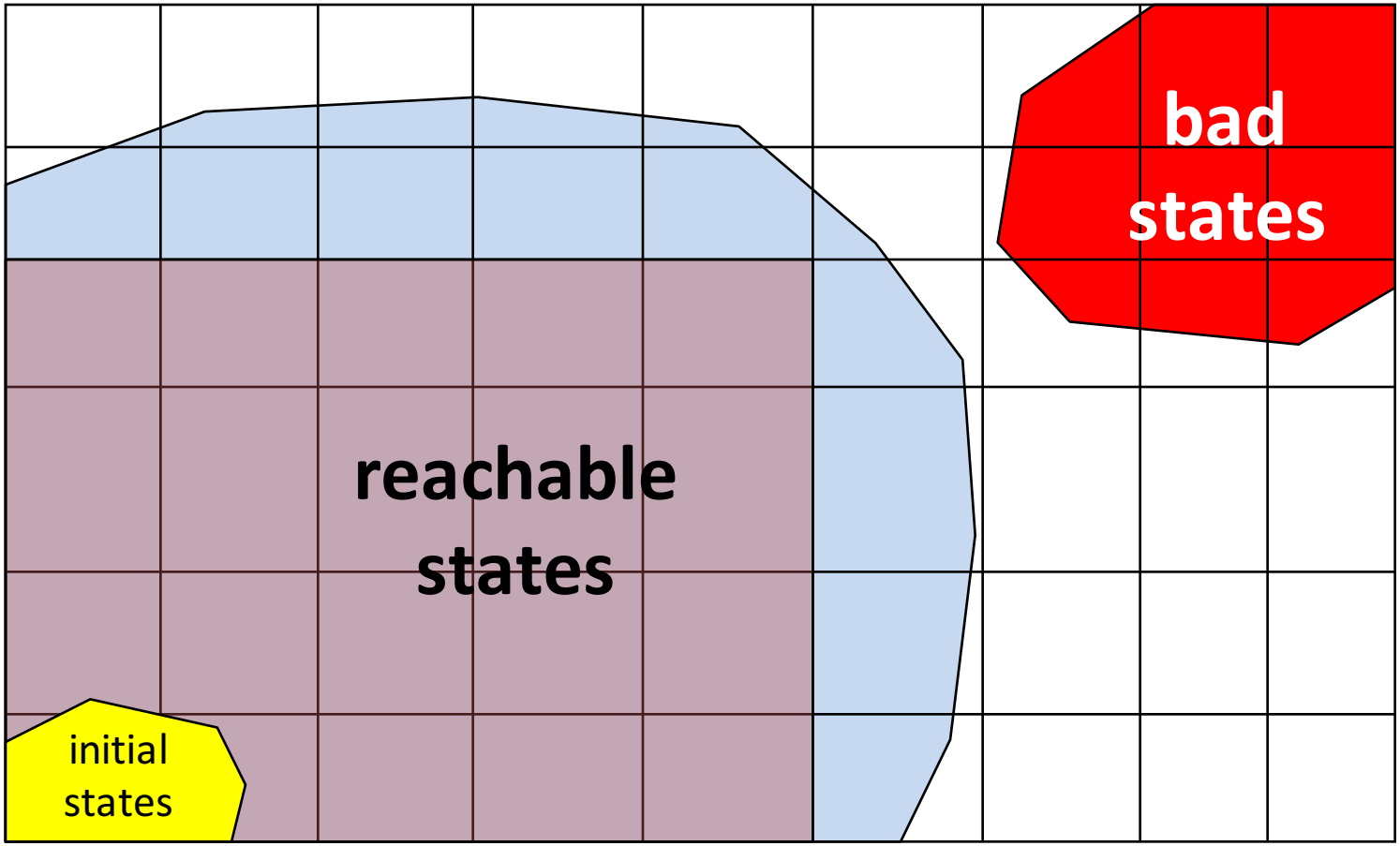
Technique: explore abstract states



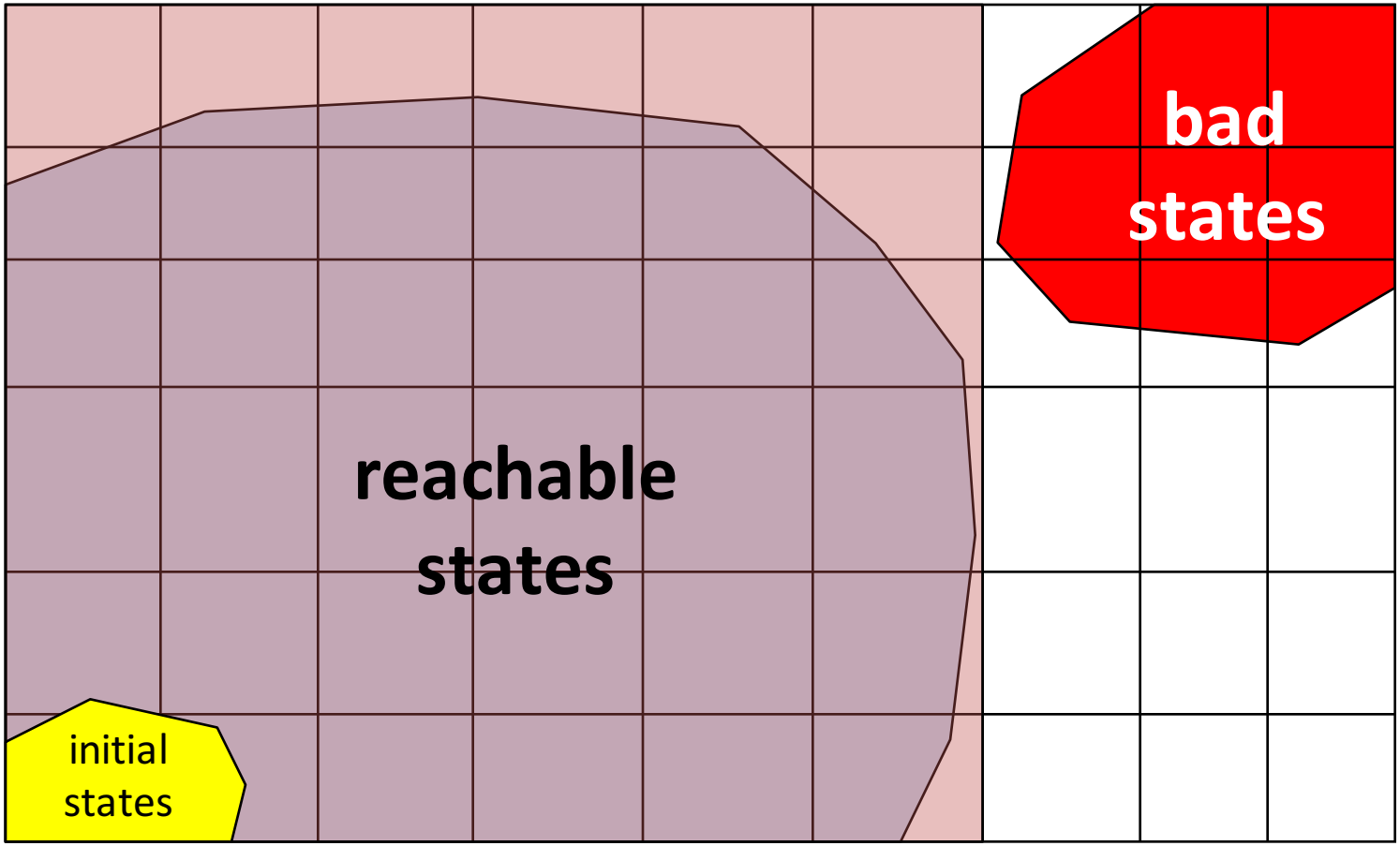
Technique: explore abstract states



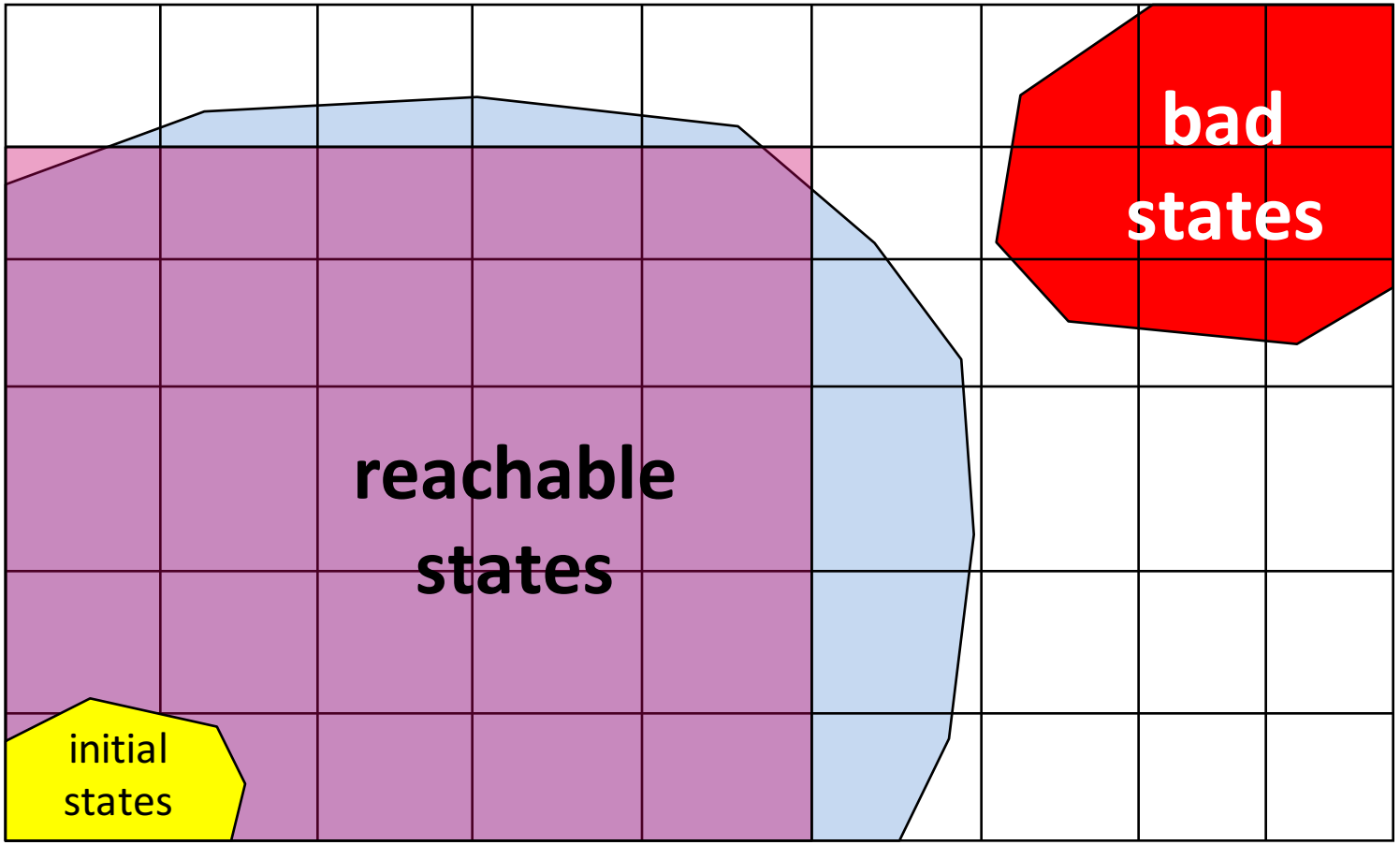
Technique: explore abstract states



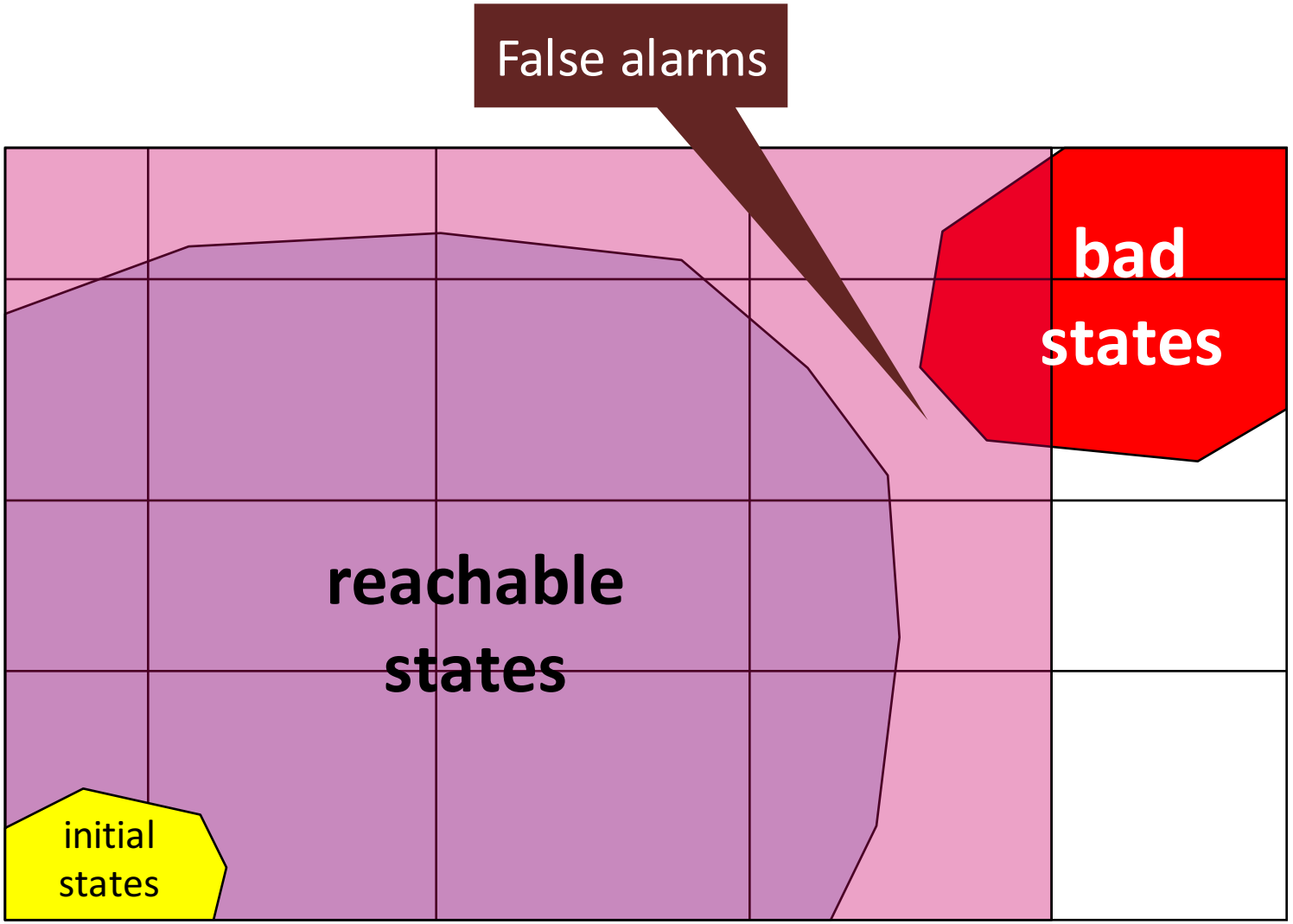
Sound: cover all reachable states



Unsound: miss some reachable states



Imprecise abstraction



55

A sound message

```
x = ?
if (x > 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

Assertion **may** be violated

Precision

- Avoid useless result

```
UselessAnalysis(Program p) {
 printf(“assertion may be violated\n”);
}
```

- Low false alarm rate
- Understand where precision is lost

A sound message

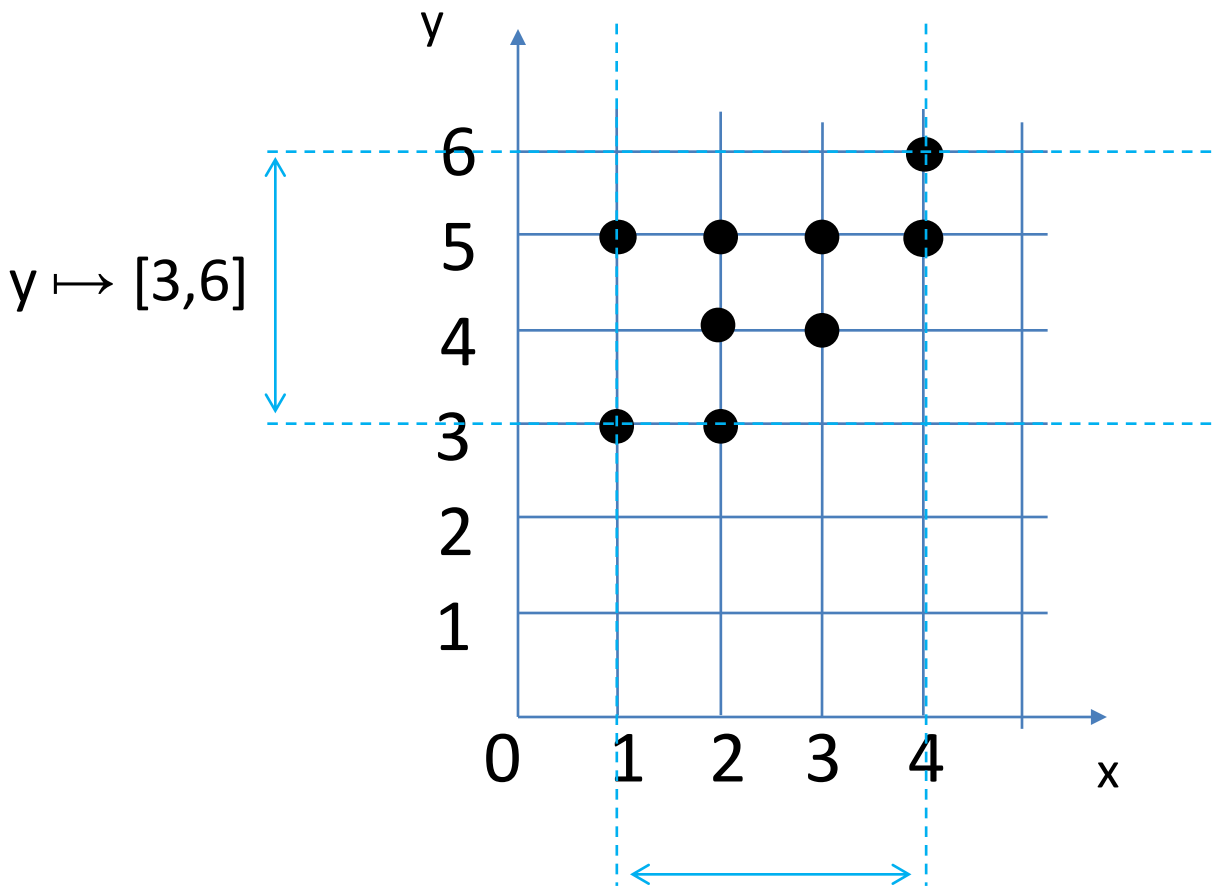
```
y = ?; x = y * 2
if (x % 2 == 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

Assertion **is true**

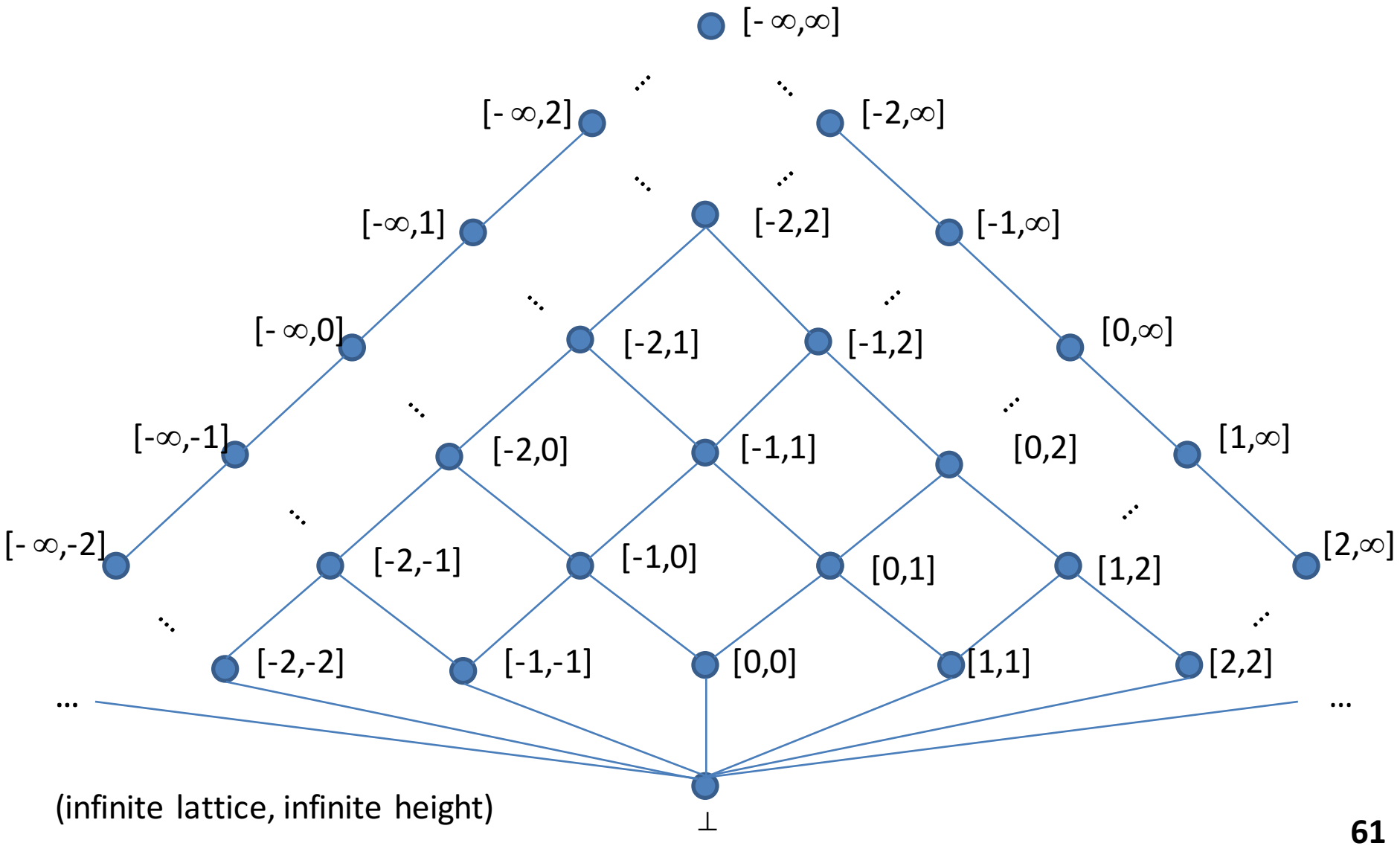
How to find “the right” abstraction?

- Pick an abstract domain suited for your property
  - Numerical domains
  - Domains for reasoning about the heap
  - …
- Combination of abstract domains

Intervals Abstraction



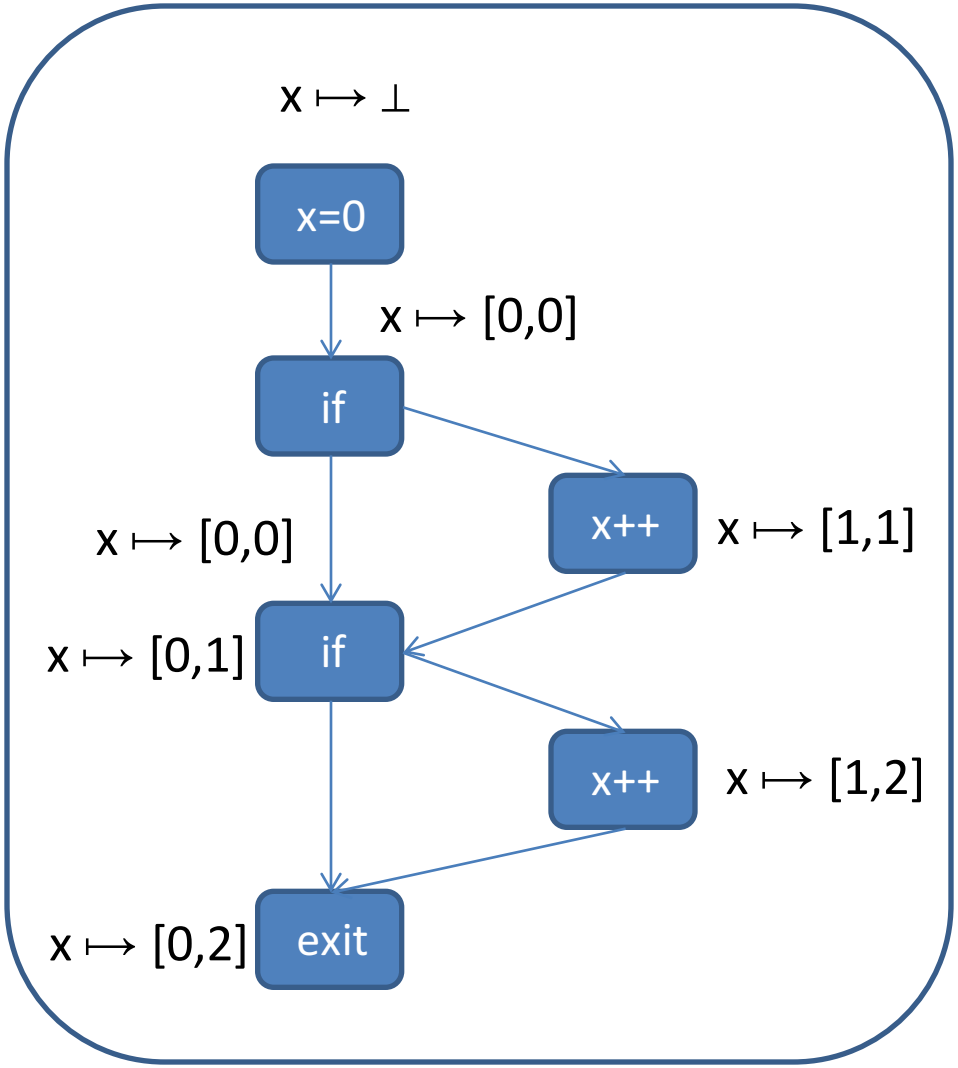
Interval Lattice



(infinite lattice, infinite height)

Example

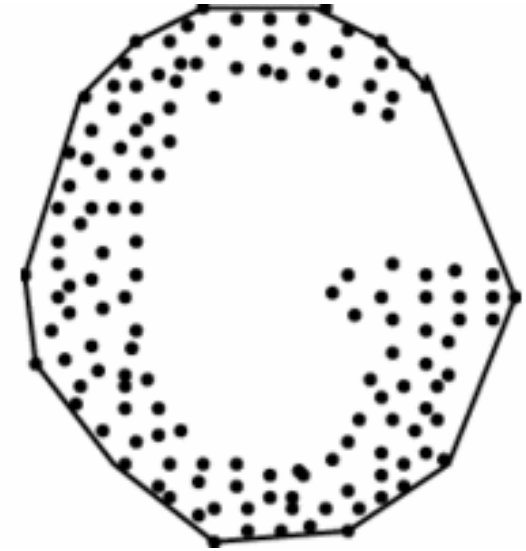
```
int x = 0;
if (?) x++;
if (?) x++;
```



[a1,a2] ⊔ [b1,b2] = [min(a1,b1), max(a2,b2)]

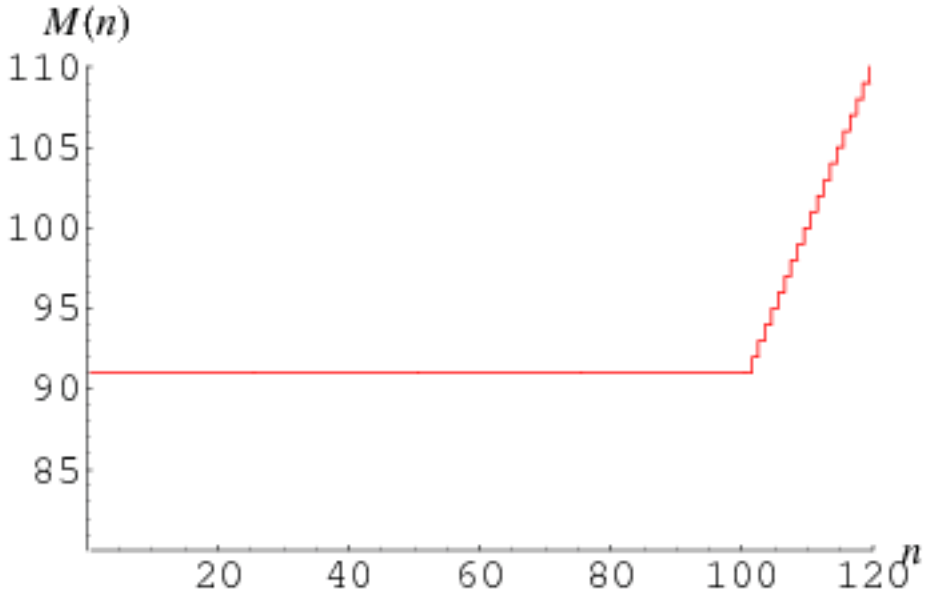
Polyhedral Abstraction

- abstract state is an intersection of linear inequalities of the form $a_1x_2+a_2x_2+\ldots a_nx_n \leq c$
- represent a set of points by their convex hull



(image from http://www.cs.sunysb.edu/~algorith/files/convex-hull.shtml)

McCarthy 91 function



$$M(n) = \begin{cases} M(M(n+11)) & \text{for } n \leq 100 \\ n - 10 & \text{for } n > 100. \end{cases}$$

McCarthy 91 function

```
proc MC (n : int) returns (r : int) var t1 : int, t2 : int;
begin

  if n > 100 then

     r = n - 10;
  else

    t1 = n + 11;
    t2 = MC(t1);

    r = MC(t2);


  endif;
end

var a : int, b : int;
begin /* top */
  b = MC(a);
end
```

if (n>=101) then n-10 else 91

McCarthy 91 function

```
proc MC                                                  : int;
begin
                    /* top */
  if n > 100 then
                    /* [|n-101>=0|] */
     r = n - 10;    /* [|-n+r+10=0; n-101>=0|] */
  else
                    /* [|-n+100>=0|] */
    t1 = n + 11;    /* [|-n+t1-11=0; -n+100>=0|] */
    t2 = MC(t1);    /* [|-n+t1-11=0; -n+100>=0;
                       -n+t2-1>=0; t2-91>=0|] */
    r = MC(t2);     /* [|-n+t1-11=0; -n+100>=0;
                          -n+t2-1>=0; t2-91>=0; r-t2+10>=0;
                          r-91>=0|] */
  endif;            /* [|-n+r+10>=0; r-91>=0|] */
end

var a : int, b : int;
begin /* top */
  b = MC(a);        /* [|-a+b+10>=0; b-91>=0|] */
end
```

if (n>=101) then n-10 else 91

if (n>=101) then n-10 else 91

McCarthy 91 function

if (n>=101) then n-10 else 91

```
proc MC (n : int) returns (r : int) var t1 : int, t2 : int;
begin
                    /* (L6 C5) top */
  if n > 100 then
                    /* (L7 C17) [|n-101>=0|] */
     r = n - 10;    /* (L8 C14) [|-n+r+10=0; n-101>=0|] */
  else
                    /* (L9 C6) [|-n+100>=0|] */
    t1 = n + 11;    /* (L10 C17) [|-n+t1-11=0; -n+100>=0|] */
    t2 = MC(t1);    /* (L11 C17) [|-n+t1-11=0; -n+100>=0;
                       -n+t2-1>=0; t2-91>=0|] */
    r = MC(t2);     /* (L12 C16) [|-n+t1-11=0; -n+100>=0;
                          -n+t2-1>=0; t2-91>=0; r-t2+10>=0;
                          r-91>=0|] */
  endif;            /* (L13 C8) [|-n+r+10>=0; r-91>=0|] */
end

var a : int, b : int;
begin
                    /* (L18 C5) top */
  b = MC(a);        /* (L19 C12) [|-a+b+10>=0; b-91>=0|] */
end
```

What is Static analysis

- Develop theory and tools for program correctness and robustness
- Reason statically (at compile time) about the possible runtime behaviors of a program

"The algorithmic discovery of properties of a program by inspection of its source text[1]"

-- Manna, Pnueli

1 Does not have to literally be the source text, just means w/o running it

Static analysis definition

Reason statically (at compile time) about the possible runtime behaviors of a program

"The algorithmic discovery of properties of a program by inspection of its source text[1]"
-- Manna, Pnueli

1 Does not have to literally be the source text, just means w/o running it

Some automatic tools

Challenges

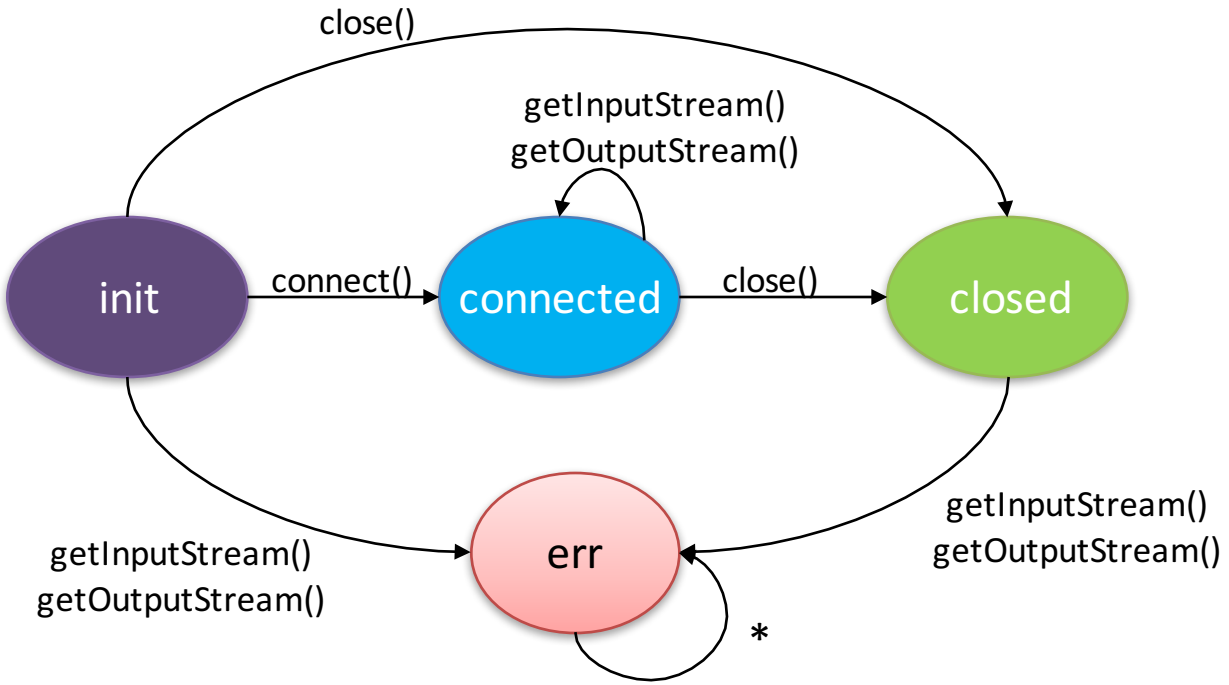
```
class SocketHolder {  Socket s;  }
Socket makeSocket() { return new Socket(); // A }
open(Socket l) { l.connect(); }
talk(Socket s) { s.getOutputStream()).write("hello"); }

main() {
  Set<SocketHolder> set = new HashSet<SocketHolder>();
  while(…) {
     SocketHolder h = new SocketHolder();
     h.s = makeSocket();
     set.add(h);
  }
  for (Iterator<SocketHolder> it = set.iterator(); …) {
     Socket g = it.next().s;
     open(g);
     talk(g);
   }
}
```

(In)correct usage of APIs

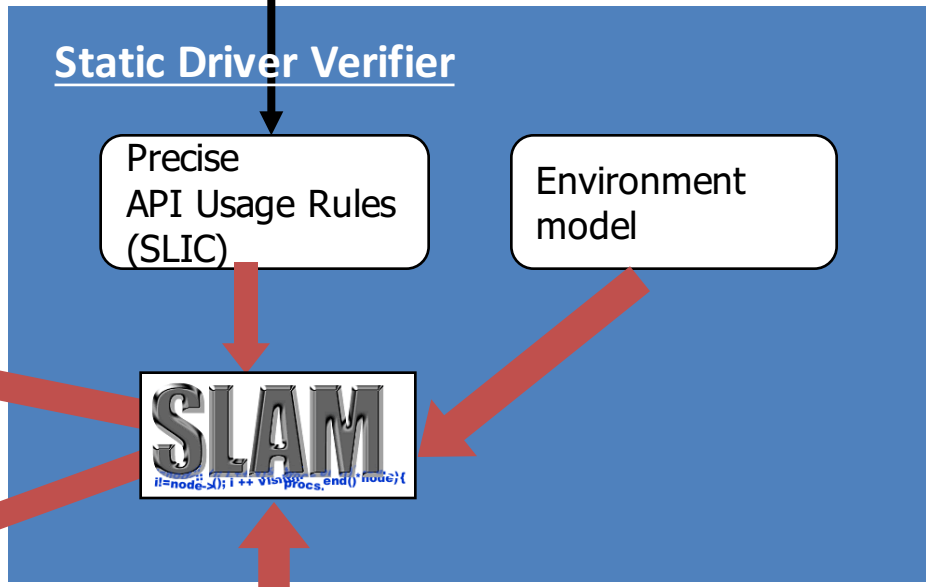
- **Application trend: Increasing number of libraries and APIs**
  - Non-trivial restrictions on permitted sequences of operations
- **Typestate:** Temporal safety properties
  - What sequence of operations are permitted on an object?
  - Encoded as DFA

e.g. "Don't use a Socket unless it is connected"



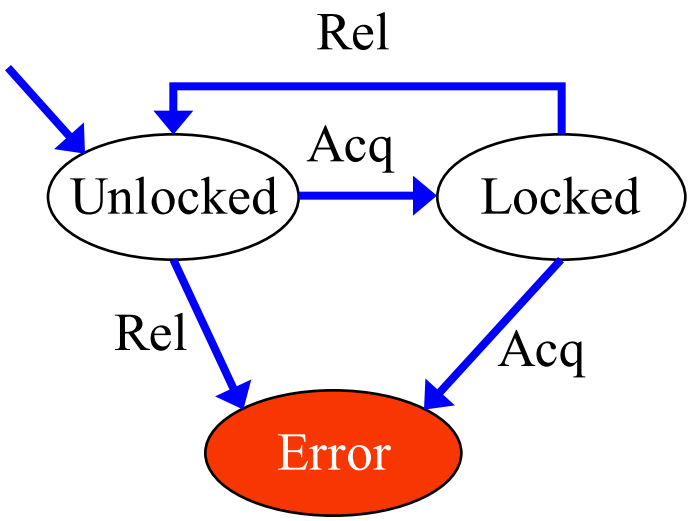
Static Driver Verifier

Rules



SLAM

State machine for locking



Locking rule in SLIC

```
state {
  enum {Locked,Unlocked}
s = Unlocked;
}

KeAcquireSpinLock.entry {
  if (s==Locked) abort;
  else s = Locked;
}

KeReleaseSpinLock.entry {
  if (s==Unlocked) abort;
  else s = Unlocked;
}
```

SLAM (now SDV) [Ball+,'11]

- 100 drivers and 80 SLIC rules.
  - The largest driver ~ 30,000 LOC
  - Total size ~450,000 LOC
- The total runtime for the 8,000 runs (driver x rule)
  - 30 hours on an 8-core machine
  - 20 mins. Timeout
- Useful results (bug / pass) on over 97% of the runs
- Caveats: pointers (imprecise) & concurrency (ignores)

The **Astrée** Static Analyzer

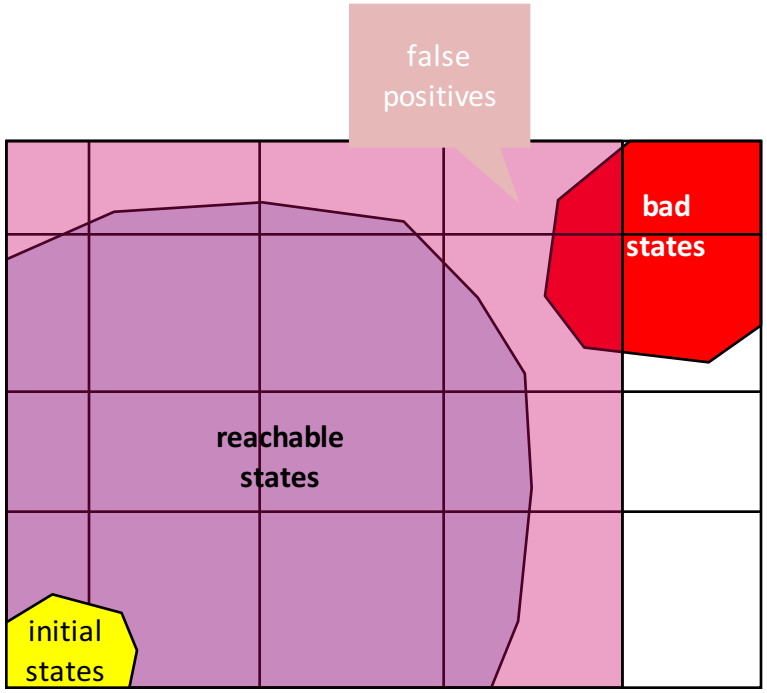
Patrick Cousot
Radhia Cousot
Jérôme Feret
Laurent Mauborgne
Antoine Miné
Xavier Rival

ENS France

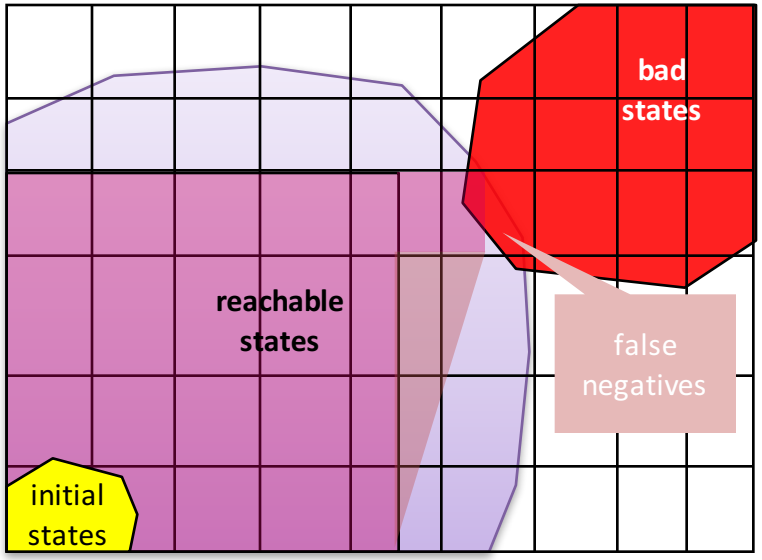
Objectives of **Astrée**

- Prove absence of errors in safety critical C code
- ASTRÉE was able to prove completely automatically the absence of any RTE in the primary flight control software of the Airbus A340 fly-by-wire system
  - a program of 132,000 lines of C analyzed

Scaling

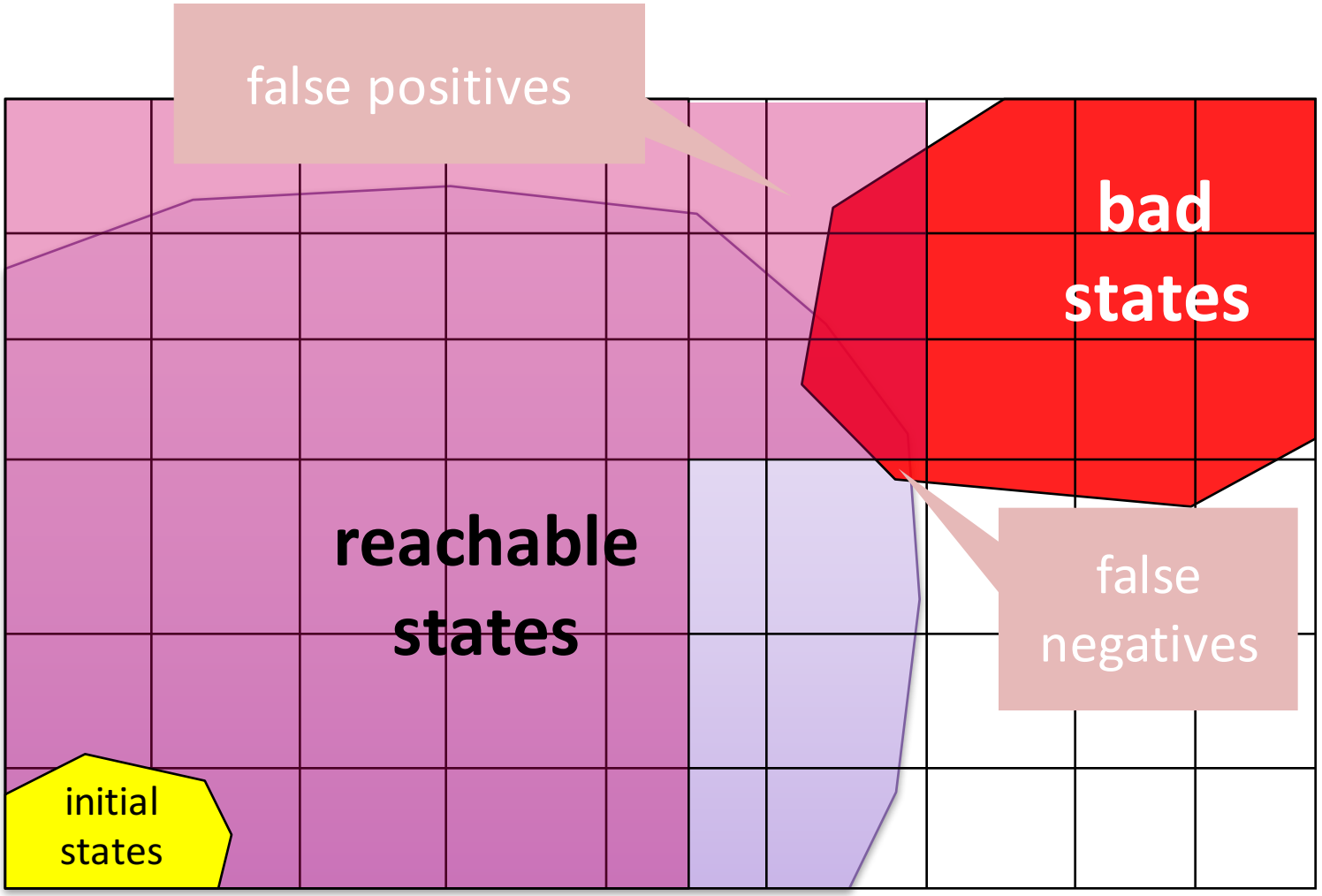


Sound



Complete

Unsound static analysis



Unsound static analysis

- Static analysis
  - No code execution
- Trade soundness for scalability
  - Do not cover all execution paths
  - But cover “many”

FindBugs [Pugh+,'04]

- Analyze Java programs (bytecode)
- Looks for "bug patterns"

- Bug patterns
  - Method() vs method()
  - Override equal(…) but not hashCode()
  - Unchecked return values
  - Null pointer dereference

App17	KLOC	NP bugs	Other Bugs	Bad Practice	Dodgy
Sun JDK 1.7	597	68	180	594	654
Eclipse 3.3	1447	146	259	1079	653
Netbeans 6	1022	189	305	3010	1112
glassfish	2176	146	154	964	1222
jboss	178	30	57	263	214

PREfix [Pincus+,'00]

- Developed by Pinucs, purchased by Microsoft
- Automatic analysis of C/C++ code
  - Memory errors, divide by zero
  - Inter-procedural bottom-up analysis
  - Heuristic - choose "100" paths
  - Minimize effect of false positive

Program	KLOC	Time
Mozilla browser	540	11h
Apache	49	15m

2-5 warnings per KLOC

PREfast

- Analyze Microsoft kernel code + device drivers
  - Memory errors, races, …
- Part of Microsoft visual studio
- Intra-procedural analysis
- User annotations

```
memcpy( __out_bcount( length ) dest, __in_bcount( length ) src, length );
```

PREfix + PREfast found 1/6 of bugs fixed in Windows Server'03

Coverity [Engler+, '04]

- Looks for bug patterns
  - Enable/disable interrupts, double locking, double locking, buffer overflow, …
- Learns patterns from common
- Robust & scalable
  - 150 open source program -6,000 bugs
  - Unintended acceleration in Toyota

Sound SA vs. Testing

Sound SA	Unsound SA	Testing
• Can find rare errors Can raise false alarms	▪ Can miss errors Can raise false alarms	• Can miss errors Finds real errors
• Cost ~ program's complexity	▪ Cost ~ program's complexity	• Cost ~ program's execution
• Can handle limited classes of programs and still be useful	▪ No need to efficiently handle rare cases	• No need to efficiently handle rare cases

Sound SA vs. Formal verification

Sound Static Analysis

- Fully automatic
- Applicable to a programming language
- Can be very imprecise
- May yield false alarms

Formal verification

- Requires specification and loop invariants
- Program specific
- Relatively complete
- Provides counter examples
- Provides useful documentation
- Can be mechanized using theorem provers

Operational Semantics

Agenda

- What does semantics mean?
  - Why do we need it?
  - How is it related to analysis/verification?
- Operational semantics
  - Natural operational semantics
  - Structural operational semantics

Program analysis & verification

```
y = ?; x = ?;
x = y * 2
if (x % 2 == 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

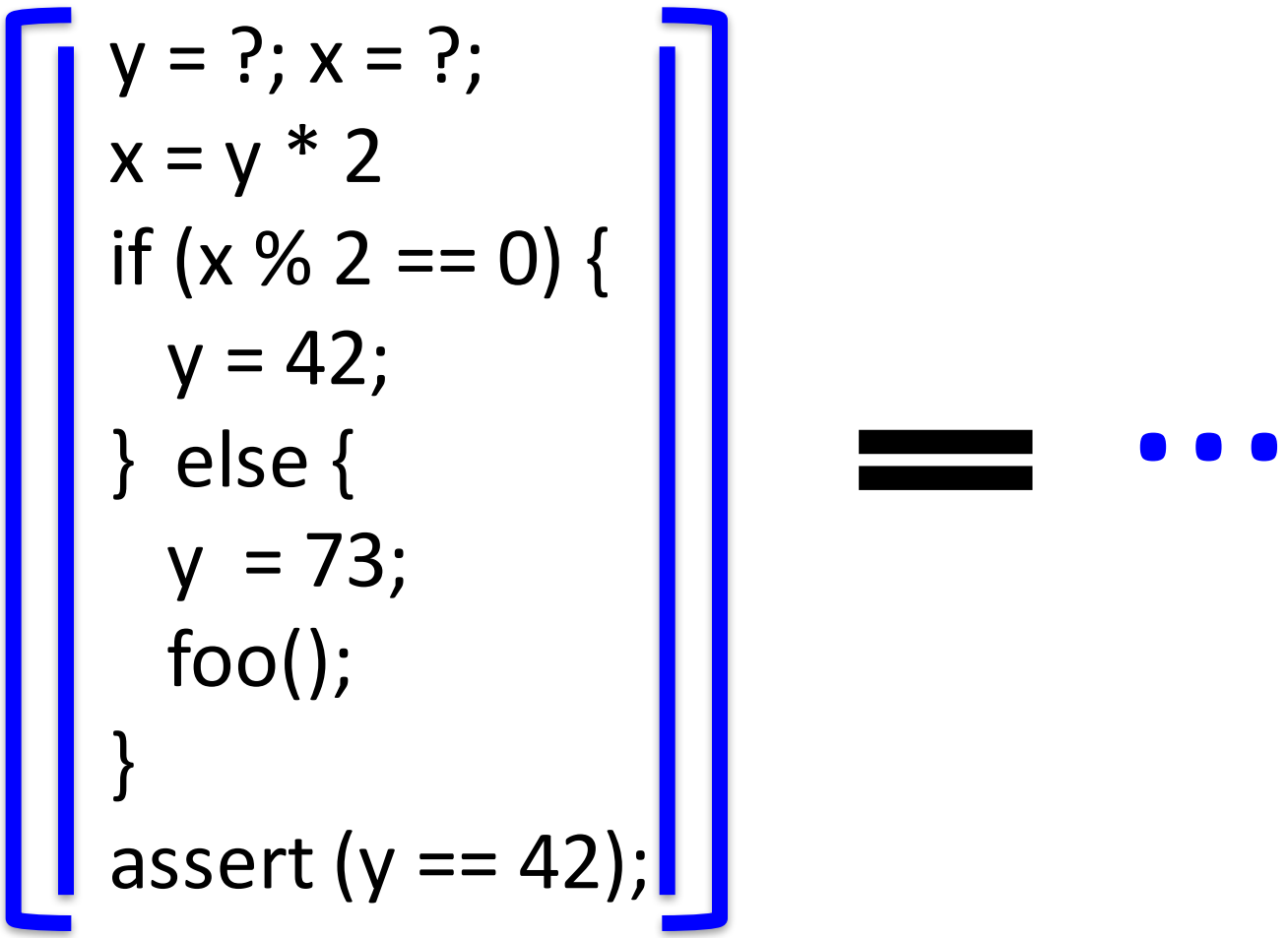
?

What does P do?

```
y = ?; x = ?;
x = y * 2
if (x % 2 == 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

?

What does P mean?



“Standard” semantics

```
y = ?;
x = y * 2
if (x % 2 == 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

…-1,0,1, … …-1,0,1,…

y **x**

“Standard” semantics

(“state transformer”)

```
y = ?;
x = y * 2
if (x % 2 == 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

…-1,0,1, … …-1,0,1,…

y **x**

“Standard” semantics

(“state transformer”)

```
y = ?;
x = y * 2
if (x % 2 == 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

y=3, x=9

…-1,0,1, … …-1,0,1,…

y **x**

“Standard” semantics

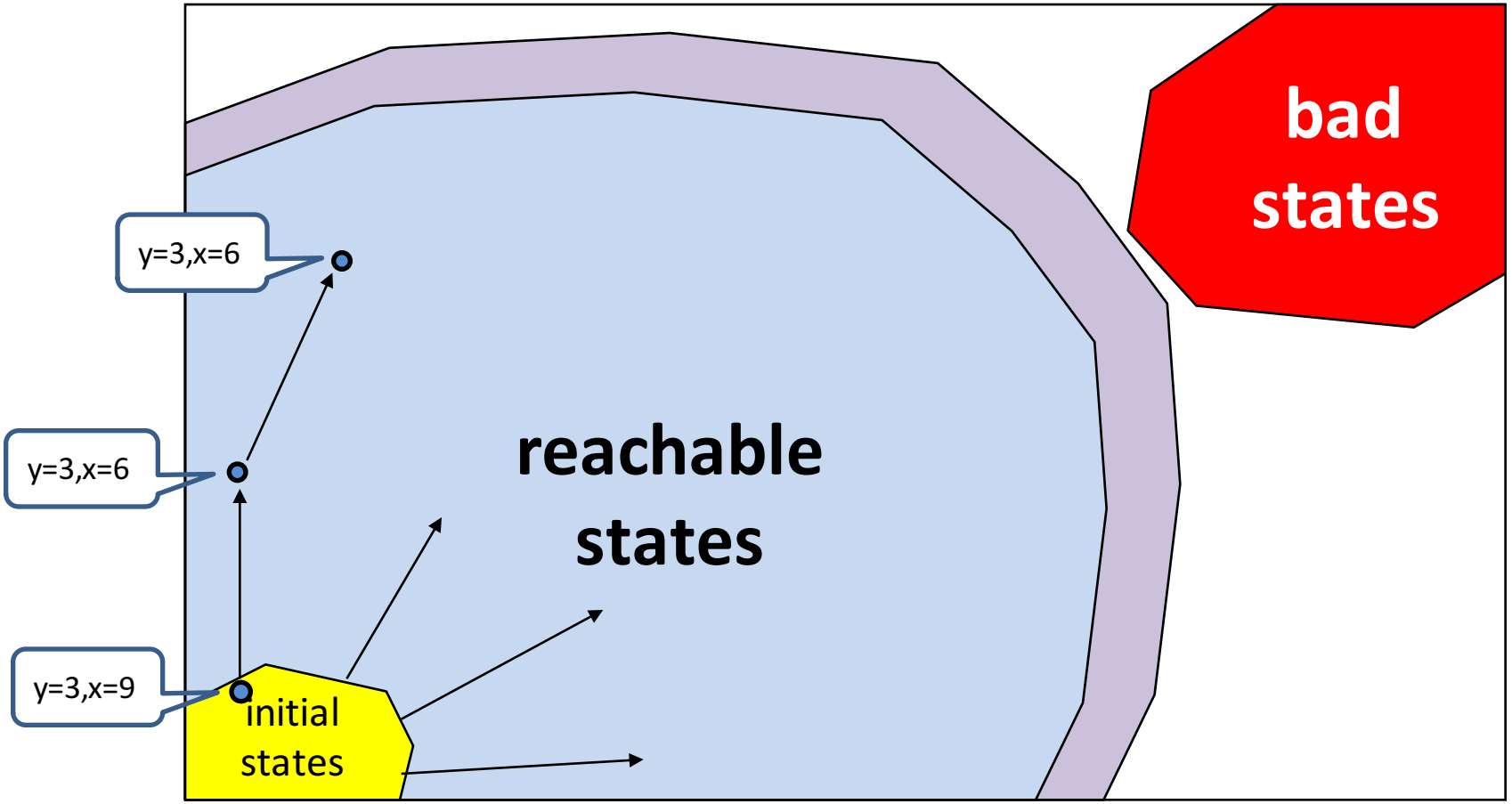
(“state transformer”)

```
y = ?;                    y=3, x=9
x = y * 2                 y=3, x=6
if (x % 2 == 0) {         y=3, x=6
   y = 42;                y=42, x=6
}  else {
   y  = 73;               …
   foo();                 …
}
assert (y == 42); y=42, x=6
```

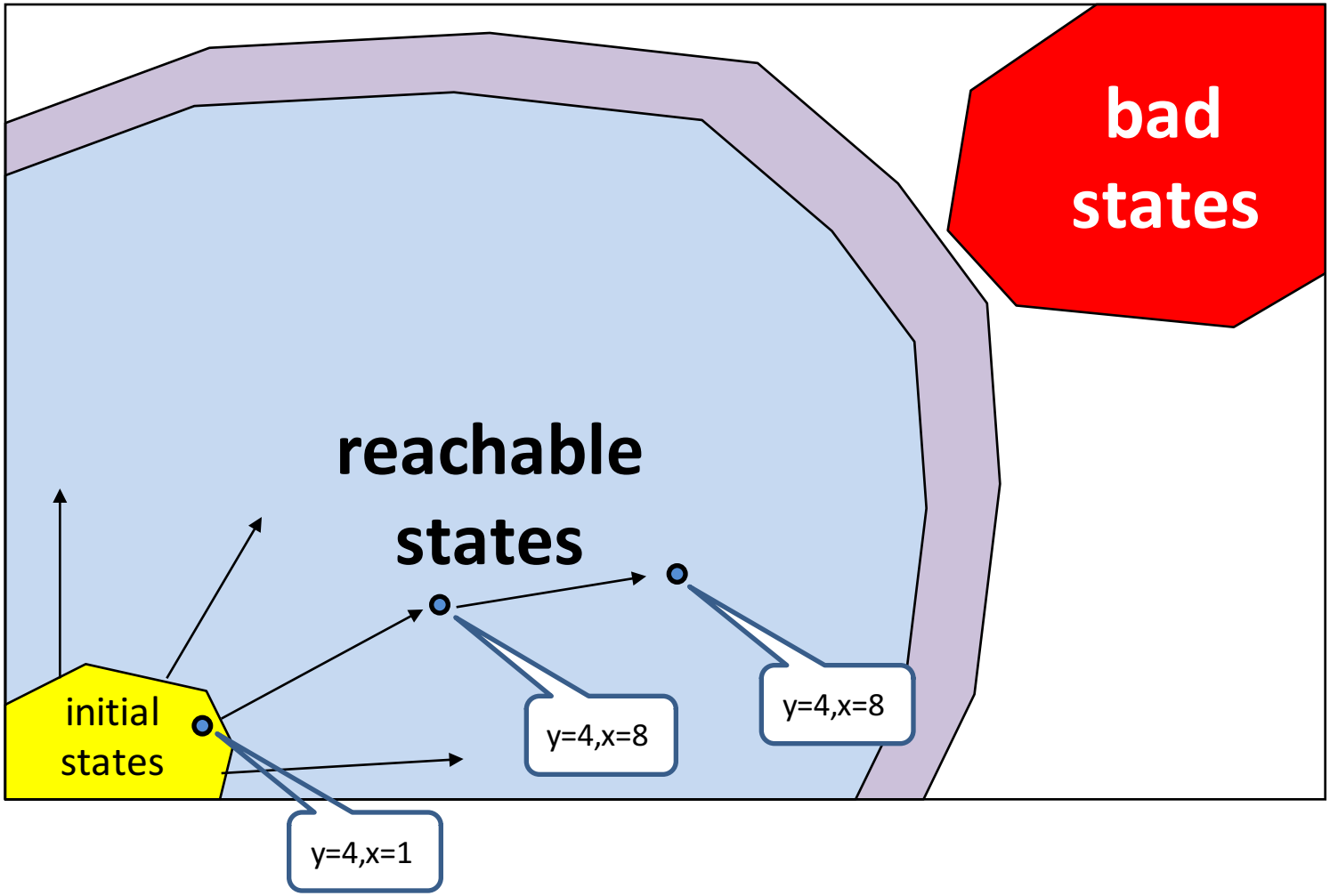
…-1,0,1, … …-1,0,1,…

y **x**

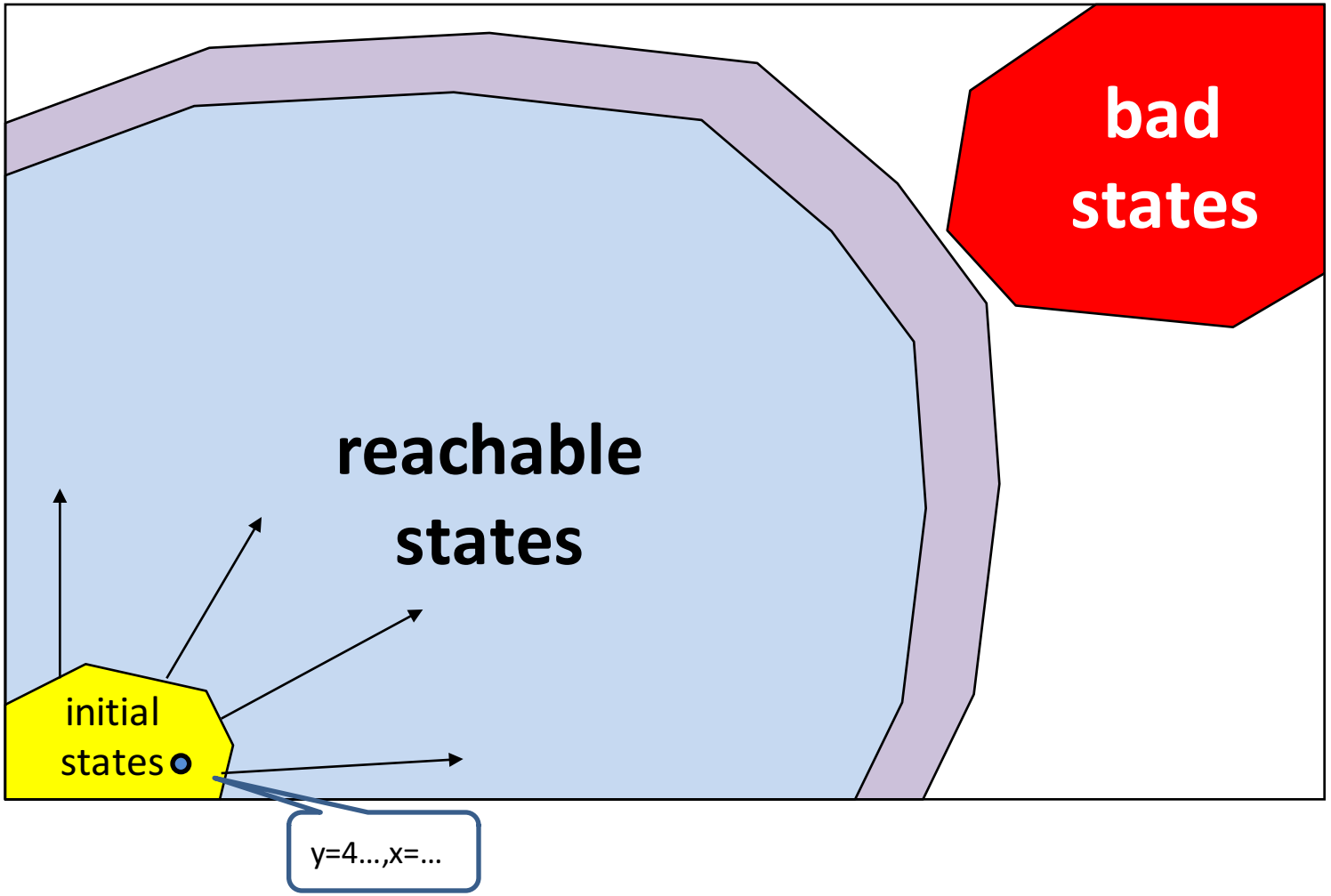
“State transformer” semantics



“State transformer” semantics

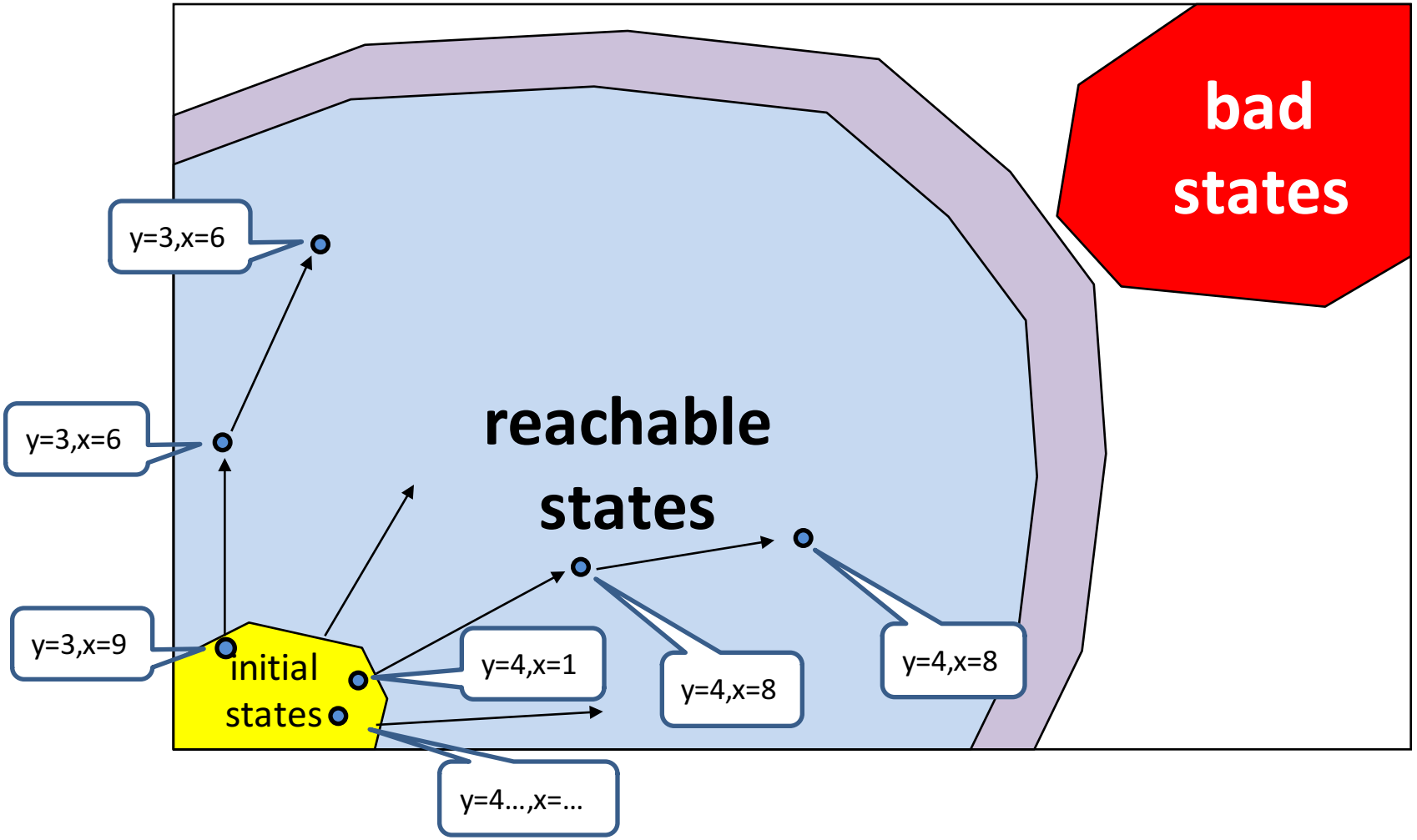


“State transformer” semantics



“State transformer” semantics

Main idea: find (properties of) all reachable states*



“Standard” (collecting) semantics

(“sets-of states-transformer”)

```
y = ?; x = ?;
x = y * 2
if (x % 2 == 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

$\{(y,x) \mid y,x \in Nat\}$

“Standard” (collecting) semantics

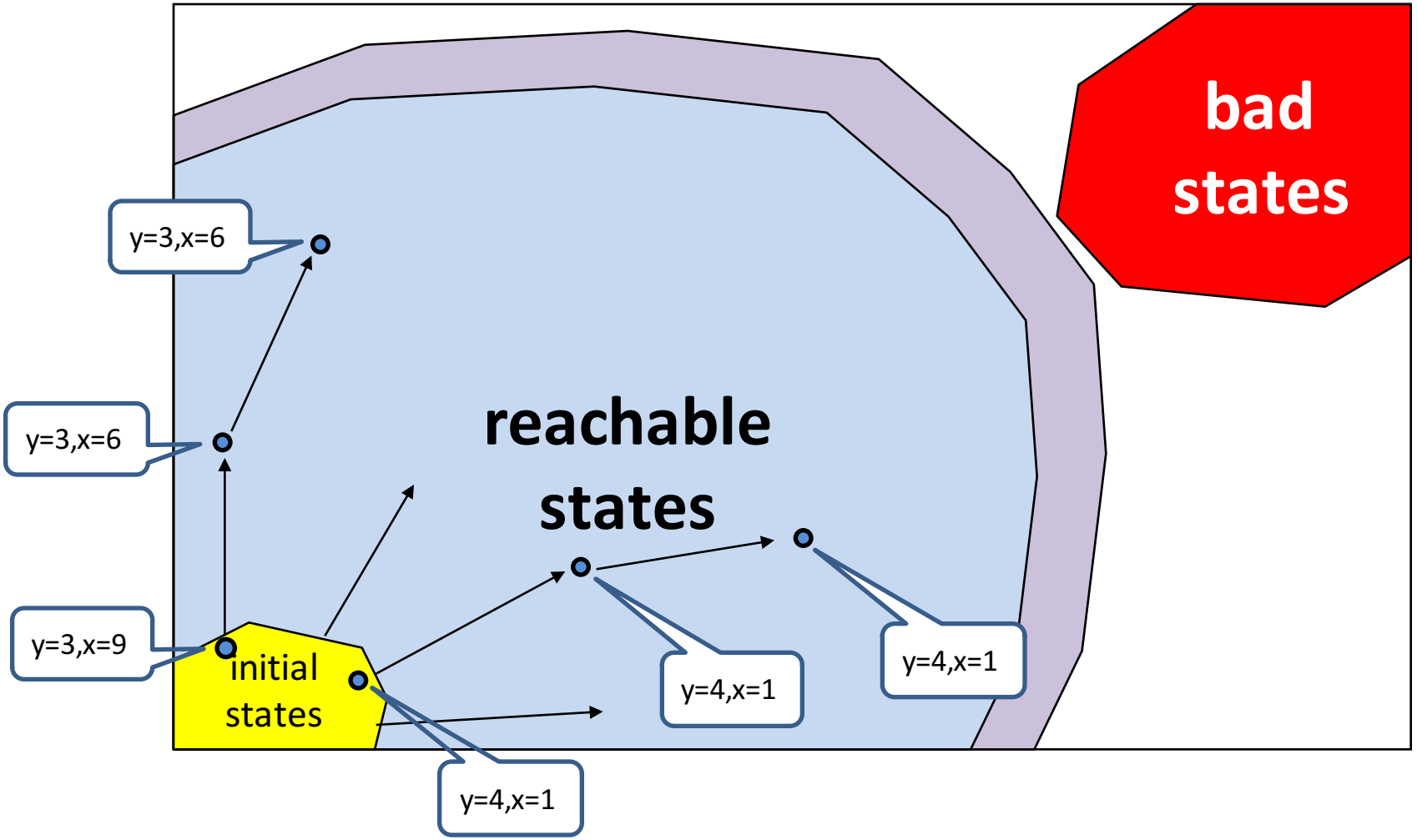
(“sets-of states-transformer”)

```
y = ?;                  {(y=3, x=9),(y=4,x=1),(y=…, x=…)}
x = y * 2               {(y=3, x=6),(y=4,x=8),(y=…, x=…)}
if (x % 2 == 0) {       {(y=3, x=6),(y=4,x=8),(y=…, x=…)}
   y = 42;              {(y=42, x=6),(y=42,x=8),(y=42, x=…)}

} else {
   y  = 73;             { }
   foo();               { }
}
assert (y == 42);       {(y=42, x=6),(y=42,x=8),(y=42, x=…)}
```

Yes

“Set-of-states transformer” semantics



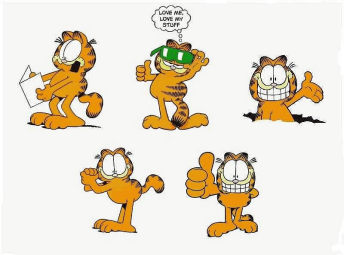
Program semantics

- State-transformer
  - Set-of-states transformer
  - Trace transformer
- Predicate-transformer
- Functions

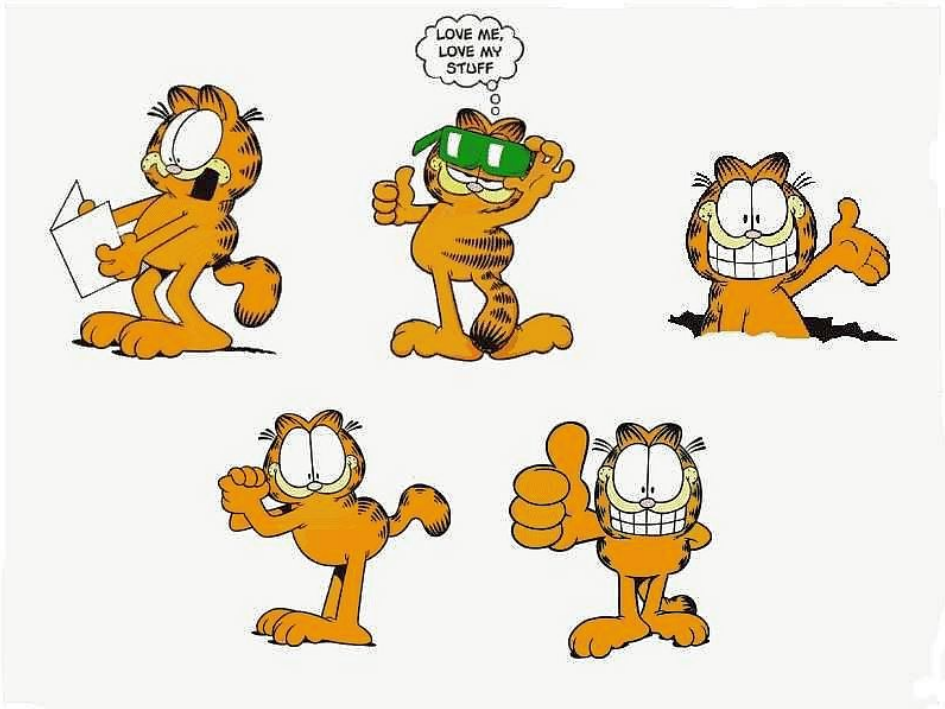
Program semantics

- State-transformer
  - Set-of-states transformer
  - Trace transformer
- Predicate-transformer
- Functions

- Cat-transformer



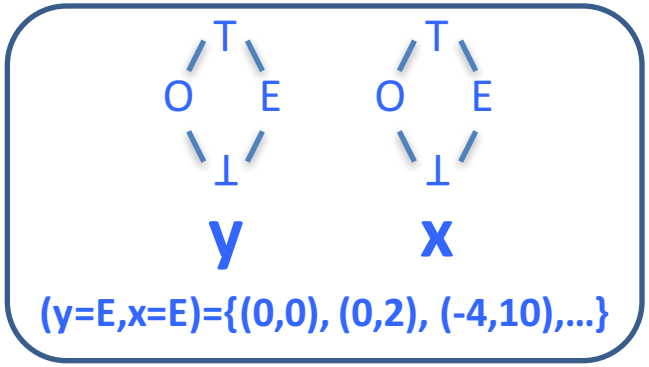
Program semantics & verification



“Abstract-state transformer” semantics

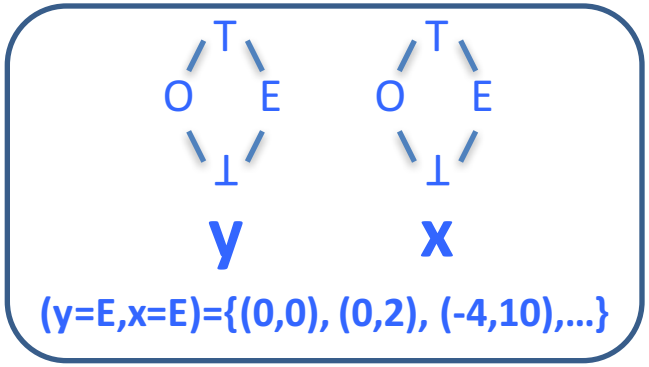
```
y = ?;
x = y * 2
if (x % 2 == 0) {
   y = 42;
}  else {
   y  = 73;
   foo();
}
assert (y == 42);
```

y=T, x=T



“Abstract-state transformer” semantics

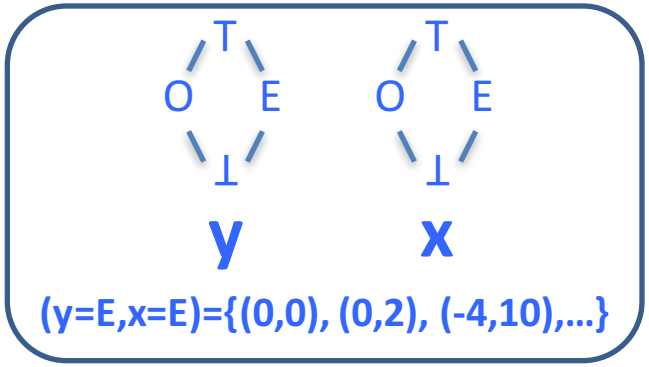
```
y = ?;                    y=T, x=T
x = y * 2                 y=T, x=E
if (x % 2 == 0) {         y=T, x=E
   y = 42;                y=T, x=E
}  else {
   y  = 73;               …
   foo();                 …
}
assert (y == 42);         y=E, x=E
```



Yes/?/No

“Abstract-state transformer” semantics

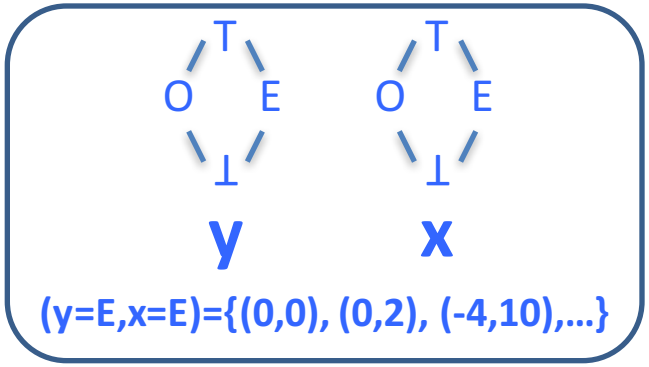
```
y = ?;                    y=T, x=T
x = y * 2                 y=T, x=E
if (x % 2 == 0) {         y=T, x=E
   y = 42;                y=T, x=E
}  else {
   y  = 73;               …
   foo();                 …
}
assert (y == 42);         y=E, x=E
```



Yes/?/No

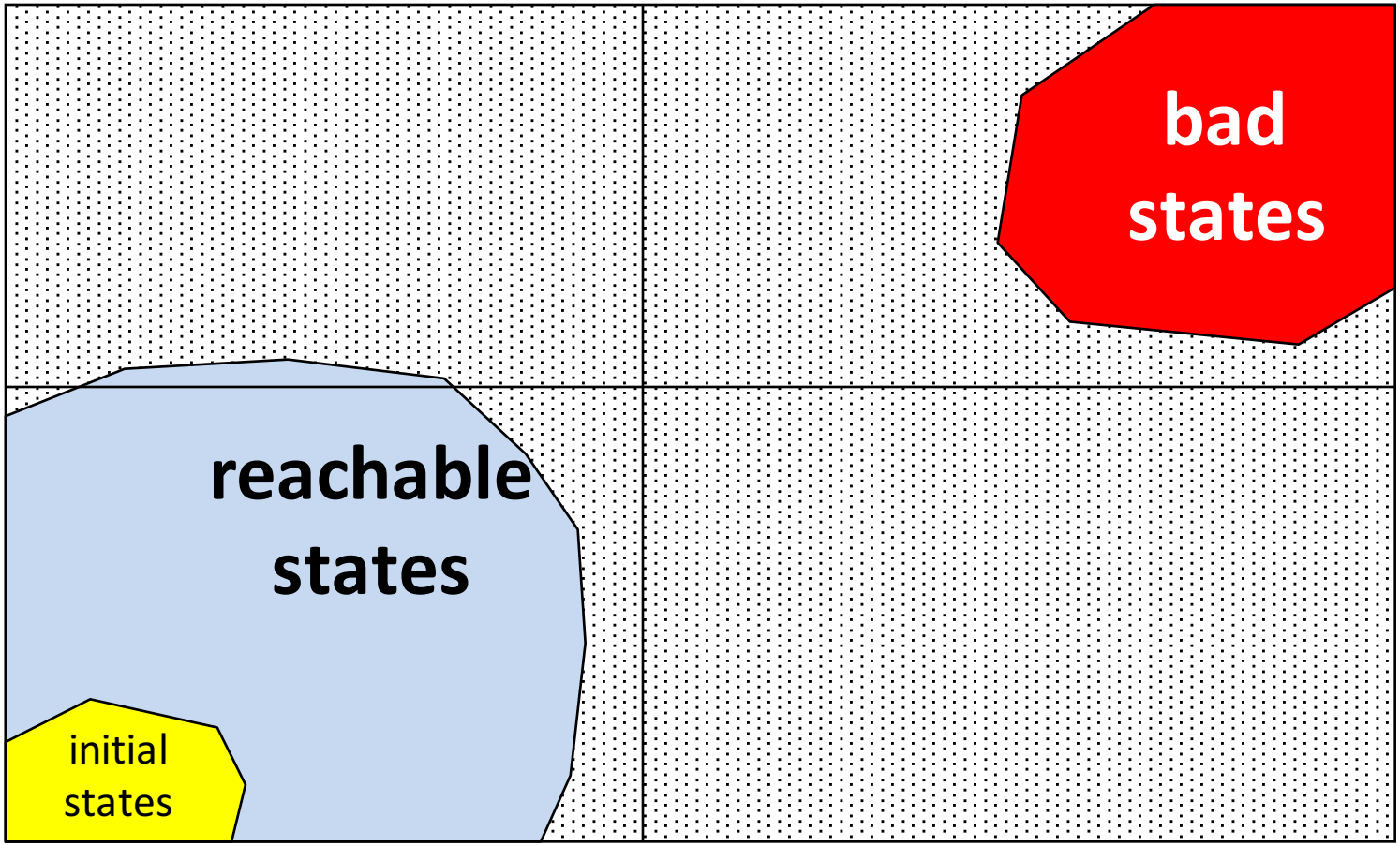
“Abstract-state transformer” semantics

```
y = ?;                    y=T, x=T
x = y * 2                 y=T, x=E
if (x % 2 == 0) {         y=T, x=E
   y = 42;                y=E, x=E
}  else {
   y  = 73;               …
   foo();                 …
}
assert (y%2 == 0)         y=E, x=E
```

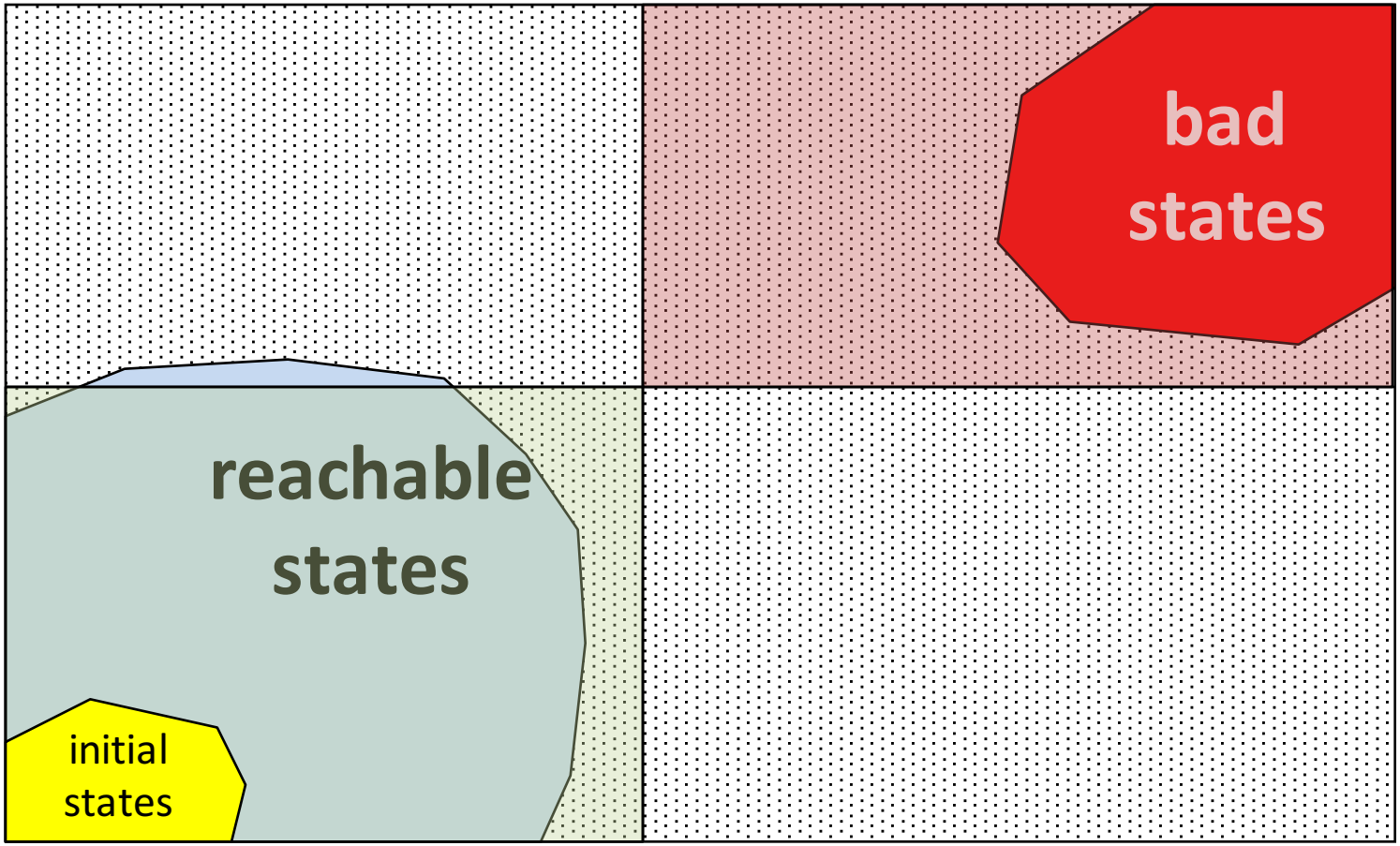


?

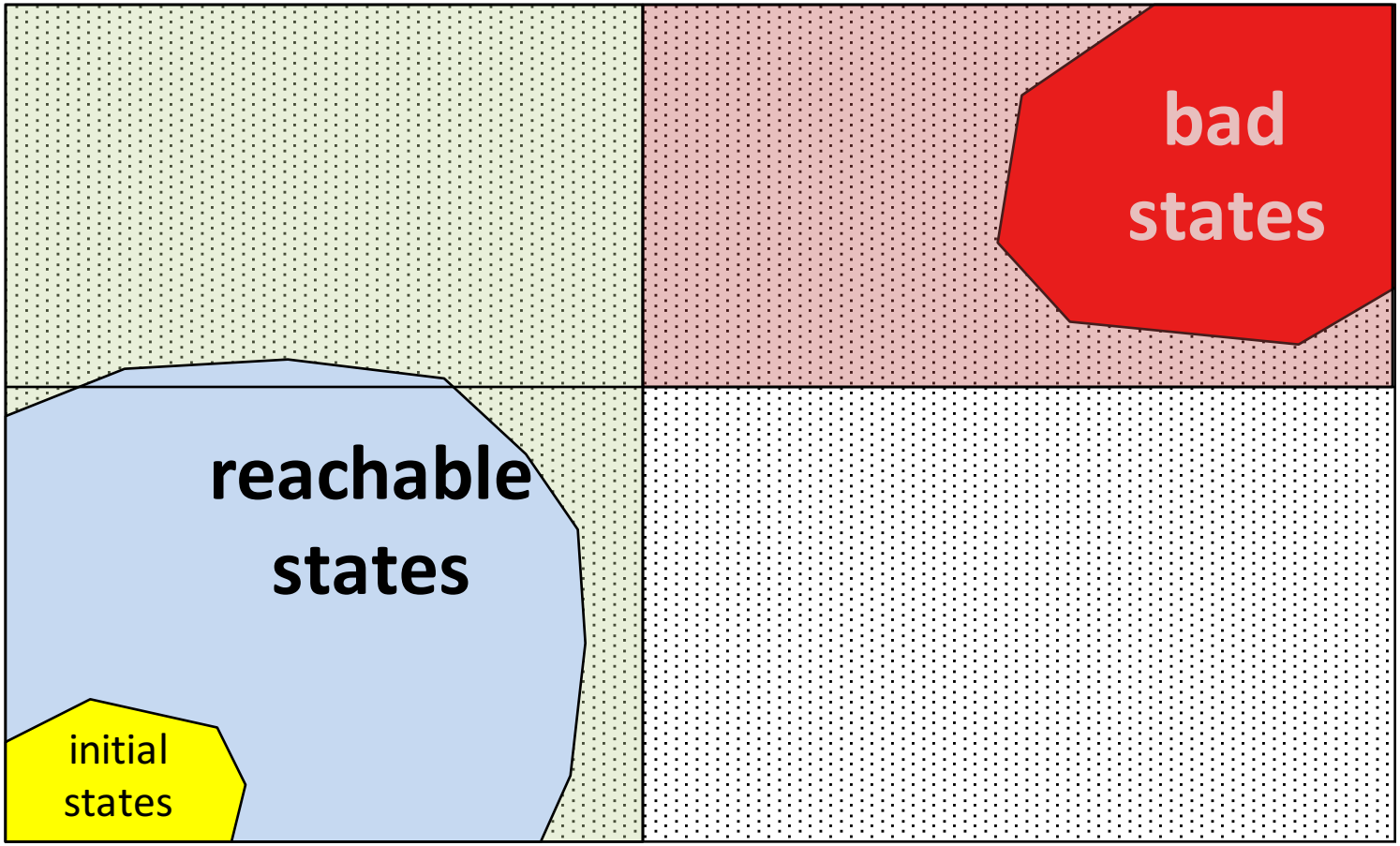
“Abstract-state transformer” semantics



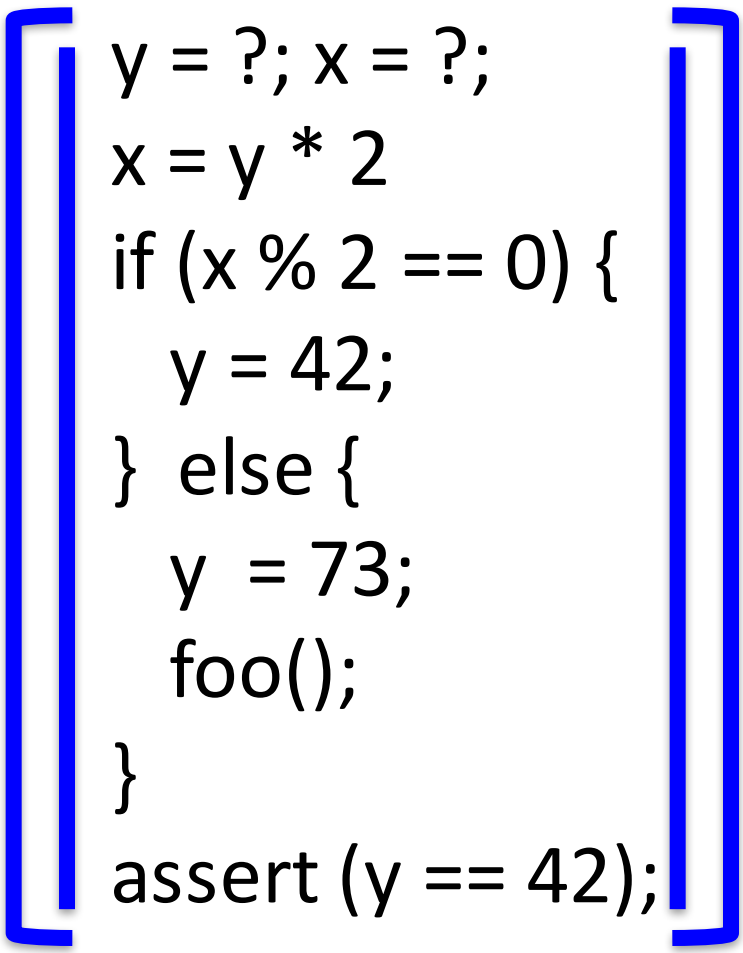
“Abstract-state transformer” semantics



“Abstract-state transformer” semantics



How do we say what P mean?



=

...

syntax

semantics

Agenda

- Operational semantics
  - Natural operational semantics
  - Structural operational semantics

Programming Languages

- Syntax
  - "how do I write a program?"
  - BNF
  - "Parsing"
- Semantics
  - "What does my program mean?"
  - …

Program semantics

- State-transformer
  - Set-of-states transformer
  - Trace transformer
- Predicate-transformer
- Functions

Program semantics

- State-transformer
  - Set-of-states transformer
  - Trace transformer
- Predicate-transformer
- Functions

What semantics do we want?

- Captures the aspects of computations we care about
  - "adequate"
- Hides irrelevant details
  - "fully abstract"
- Compositional

What semantics do we want?

- Captures the aspects of computations we care about
  - “adequate”
- Hides irrelevant details
  - “fully abstract”
- Compositional

Formal semantics

"Formal semantics is concerned with rigorously specifying the meaning, or behavior, of programs, pieces of hardware, etc."

Semantics with Applications – a Formal Introduction (Page 1)
Nielsen & Nielsen

Formal semantics

"This theory allows a program to be manipulated like a formula –
that is to say, its properties can be calculated."

Gérard Huet & Philippe Flajolet homage to Gilles Kahn

Why formal semantics?

- Implementation-independent definition of a programming language

- Automatically generating interpreters
  - and some day maybe full fledged compilers

- Verification and debugging
  - if you don't know what it does, how do you know its incorrect?

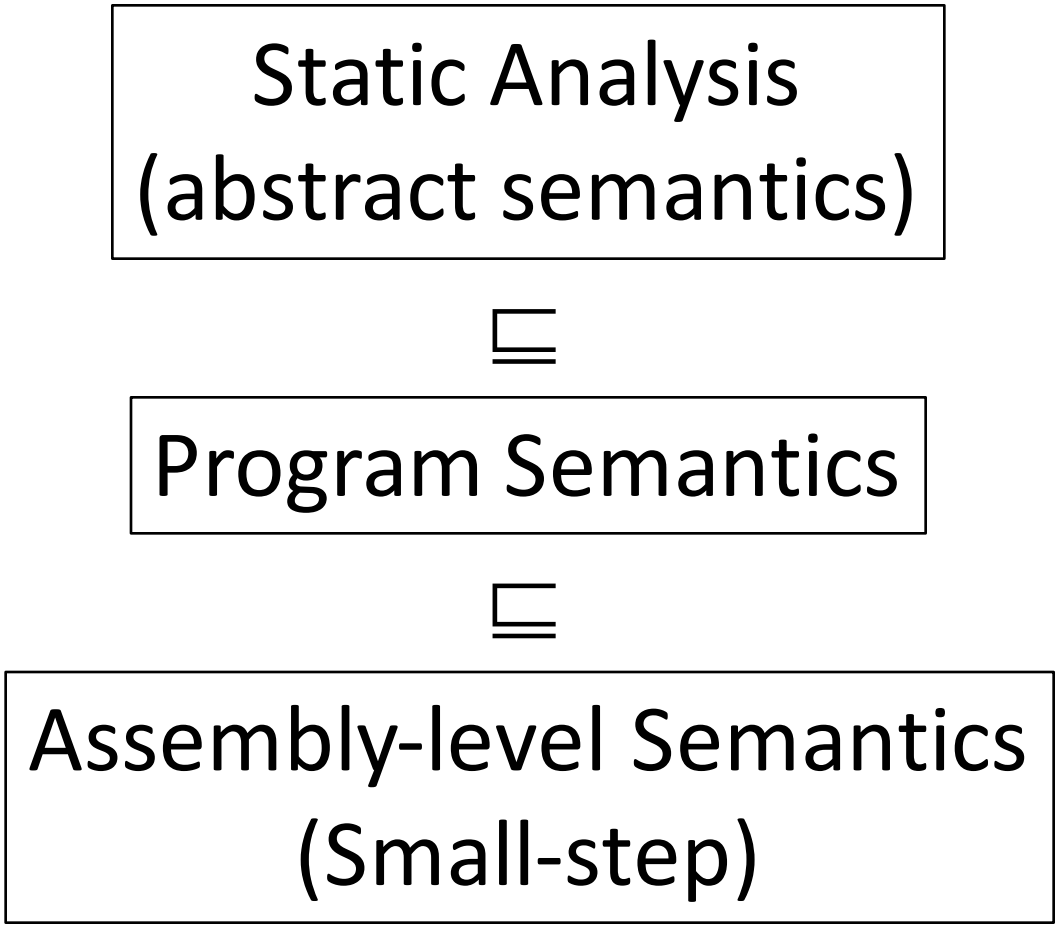
Why formal semantics?

- Implementation-independent definition of a programming language

- Automatically generating interpreters
  - and some day maybe full fledged compilers

- Verification and debugging
  - if you don't know what it does, how do you know its incorrect?

Levels of abstractions and applications

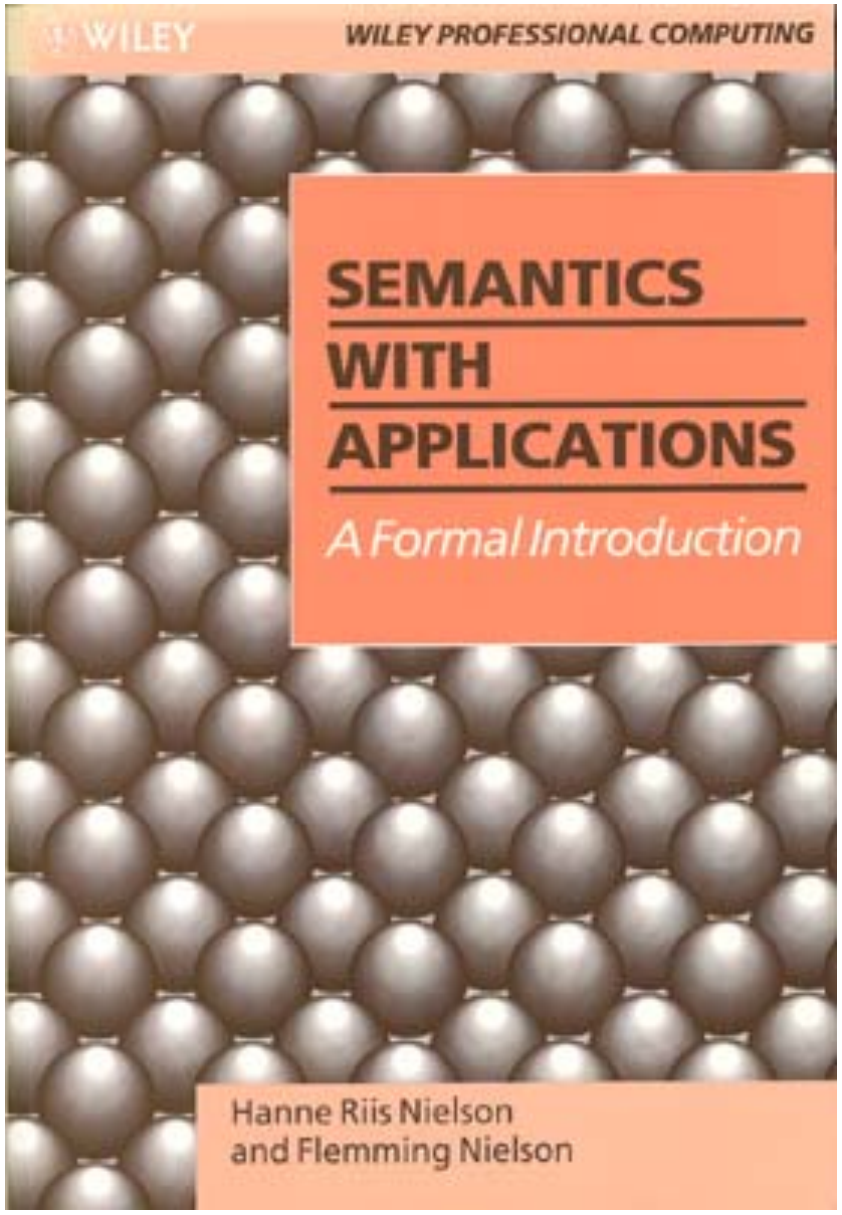


Semantic description methods

- Operational semantics
  - Natural semantics (big step) [G. Kahn]
  - Structural semantics (small step) [G. Plotkin]
    - *Trace semantics*
    - *Collecting semantics*
    - *[Instrumented semantics]*
- Denotational semantics [D. Scott, C. Strachy]
- Axiomatic semantics [C. A. R. Hoare, R. Floyd]

Operational Semantics

http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html



A simple imperative language: **While**

Abstract syntax:

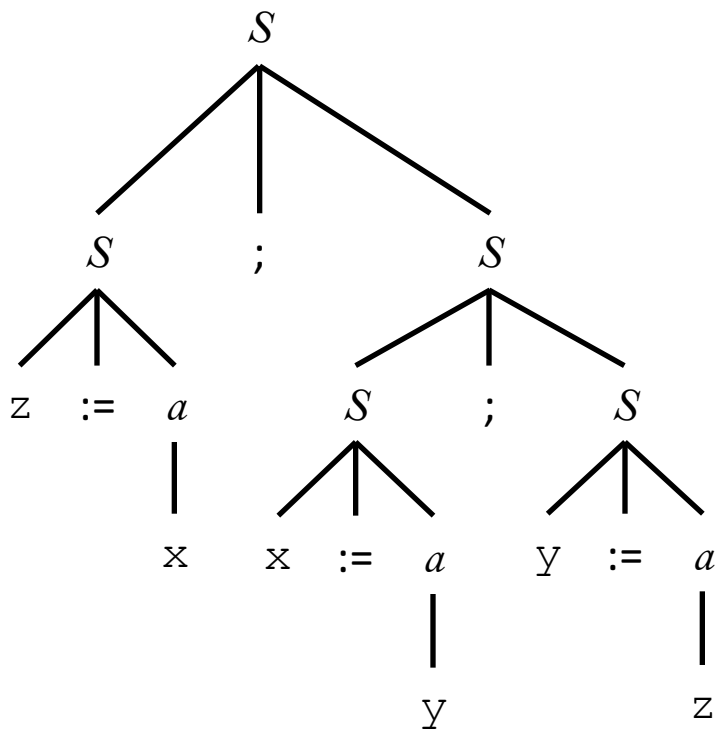
$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2$$

$$b ::= \texttt{true} \mid \texttt{false}$$
$$\mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

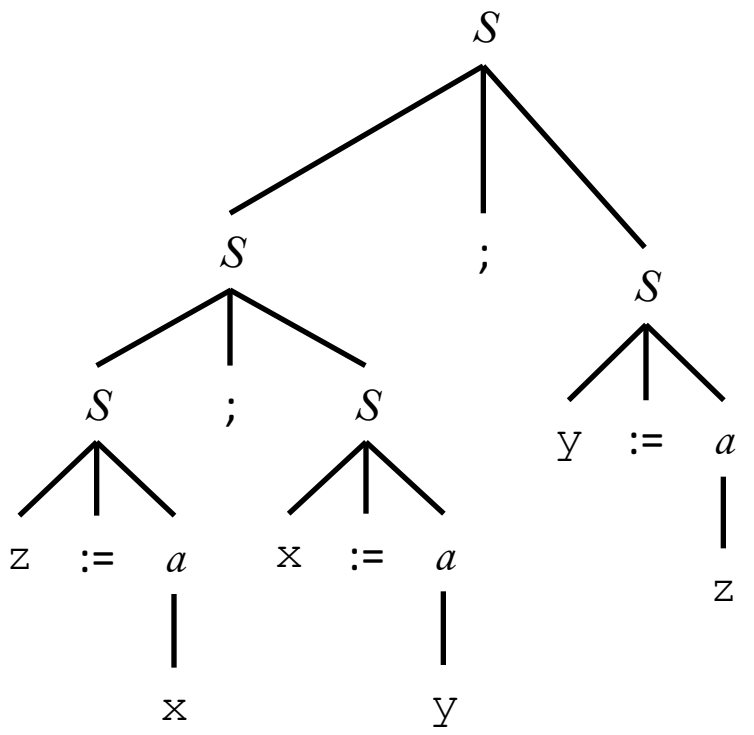
$$S ::= x := a \mid \texttt{skip} \mid S_1 ; S_2$$
$$\mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$$
$$\mid \texttt{while } b \texttt{ do } S$$

Concrete syntax vs. abstract syntax

z:=x; x:=y; y:=z



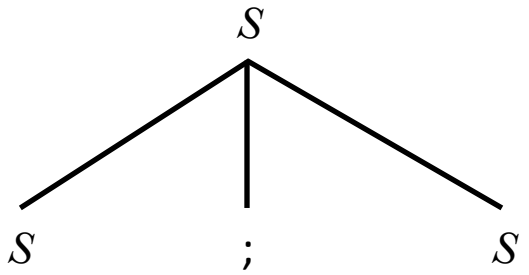
z:=x; (x:=y; y:=z)



(z:=x; x:=y); y:=z

Exercise: draw an AST

`y:=1; while ¬(x=1) do (y:=y*x; x:=x-1)`



Syntactic categories

$n \in$ **Num**	numerals
$x \in$ **Var**	program variables
$a \in$ **Aexp**	arithmetic expressions
$b \in$ **Bexp**	boolean expressions
$S \in$ **Stm**	statements

Semantic categories

Z	Integers {0, 1, -1, 2, -2, …}
T	Truth values {**ff**, **tt**}
State	**Var** → **Z**
Example state:	s=[x↦5, y↦7, z↦0]
Lookup:	s x = 5
Update:	s[x↦6] = [x↦6, y↦7, z↦0]

Example state manipulations

- $[x \mapsto 1, y \mapsto 7, z \mapsto 16]\ y$ =
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16]\ t$ =
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16][x \mapsto 5]$ =
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16][x \mapsto 5]\ x$ =
- $[x \mapsto 1, y \mapsto 7, z \mapsto 16][x \mapsto 5]\ y$ =

Semantics of arithmetic expressions

- Arithmetic expressions are side-effect free
- Semantic function $\mathcal{A}[\![\ \textbf{Aexp}\ ]\!] : \textbf{State} \rightarrow \textbf{Z}$
- Defined by induction on the syntax tree

  $\mathcal{A}[\![\ n\ ]\!]\ s = n$

  $\mathcal{A}[\![\ x\ ]\!]\ s = s\ x$

  $\mathcal{A}[\![\ a_1 + a_2\ ]\!]\ s = \mathcal{A}[\![\ a_1\ ]\!]\ s\ +\ \mathcal{A}[\![\ a_2\ ]\!]\ s$

  $\mathcal{A}[\![\ a_1 - a_2\ ]\!]\ s = \mathcal{A}[\![\ a_1\ ]\!]\ s\ -\ \mathcal{A}[\![\ a_2\ ]\!]\ s$

  $\mathcal{A}[\![\ a_1 * a_2\ ]\!]\ s = \mathcal{A}[\![\ a_1\ ]\!]\ s \times \mathcal{A}[\![\ a_2\ ]\!]\ s$

  $\mathcal{A}[\![\ (a_1)\ ]\!]\ s = \mathcal{A}[\![\ a_1\ ]\!]\ s$ --- not needed

  $\mathcal{A}[\![\ -a\ ]\!]\ s = 0 - \mathcal{A}[\![\ a_1\ ]\!]\ s$
- Compositional
- Properties can be proved by structural induction

Arithmetic expression exercise

Suppose s x = 3

Evaluate $\mathcal{A}\ [\![\texttt{x+1}]\!]$ s

Semantics of boolean expressions

- Boolean expressions are side-effect free
- Semantic function $\mathcal{B}[\![\ \textbf{Bexp}\ ]\!] : \textbf{State} \rightarrow \textbf{T}$
- Defined by induction on the syntax tree

  $\mathcal{B}[\![\ \text{true}\ ]\!]\ s = tt$

  $\mathcal{B}[\![\ \text{false}\ ]\!]\ s = ff$

  $\mathcal{B}[\![\ a_1 = a_2]\!]\ s =$

  $\mathcal{B}[\![\ a_1 \leq a_2\ ]\!]\ s =$

  $\mathcal{B}[\![\ b_1 \wedge b_2\ ]\!]\ s =$

  $\mathcal{B}[\![\ \neg\ b\ ]\!]\ s =$

Operational semantics

- Concerned with **how** to execute programs
  - How statements modify **state**
  - Define transition relation between configurations
- Two flavors
  - Natural semantics: describes how the **overall** results of executions are obtained
    - So-called “big-step” semantics
  - Structural operational semantics: describes how the **individual steps** of a computations take place
    - So-called “small-step” semantics

The End