Lecture 01 - Introduction

# PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav

# Goal

- Understand program analysis & synthesis
  - apply these techniques in your research
  - understand jargon/papers
  - conduct research in this area
- We will cover some areas in more depth than others
- What will help us
  - TA: **Nimrod Partush**
  - lecture summaries
  - 3-5 homework assignments
  - Small lightweight project
  - No exam


December 31, 2008

# Zune Bug

```
while (days > 365) {
 if (IsLeapYear(year)) {
   if (days > 366) {
     days -= 366;
     year += 1;
   }
 } else {
   days -= 365;
   year += 1;
 }
}
```

# Zune Bug

```
while (366 > 365) {
 if (IsLeapYear(2008)) {
   if (366 > 366) {
     days -= 366;
     year += 1;
   }
 } else {
   days -= 365;
   year += 1;
 }
}
```


Suggested solution: wait for tomorrow

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
  - 0.0000000000000000000000011001100… 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

*I just want to say LOVE YOU SAN!!*

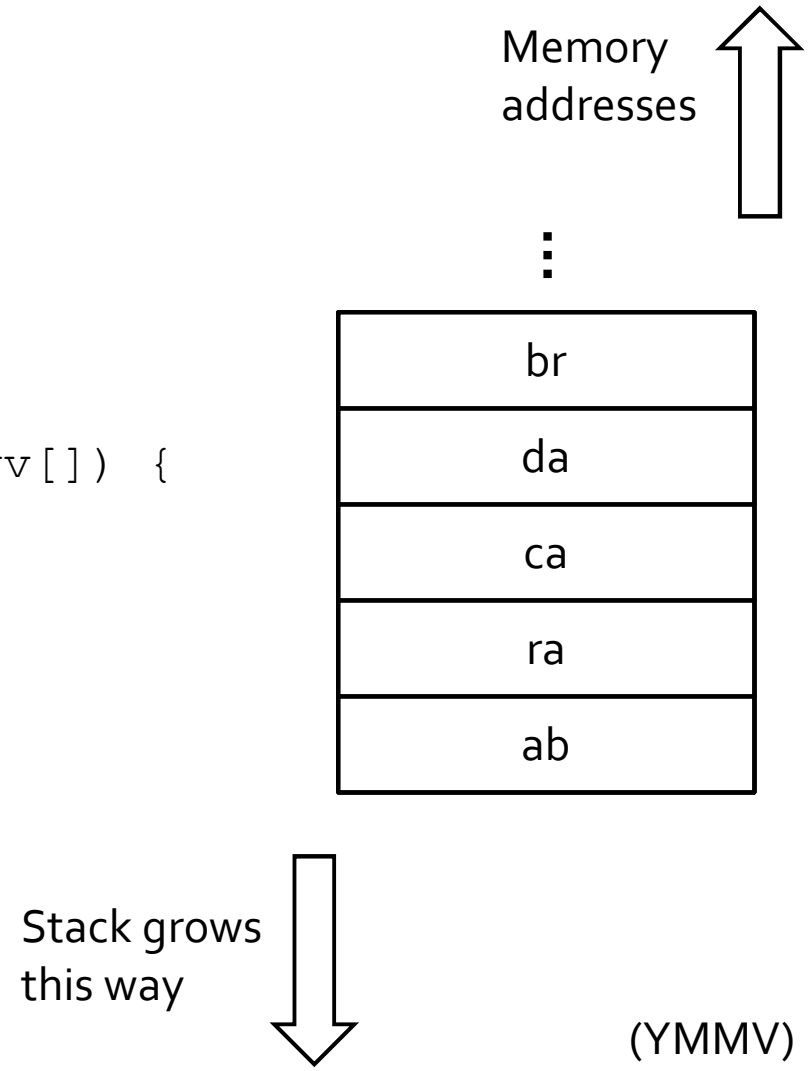
(W32.Blaster.Worm)

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
```

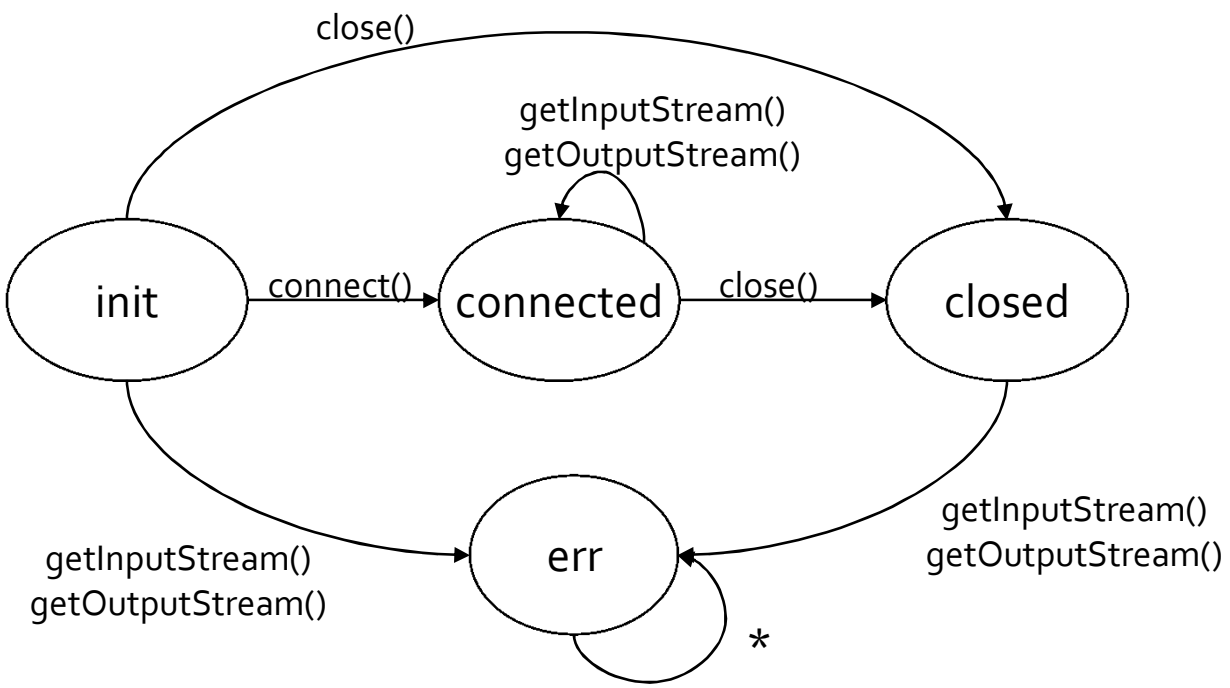



(YMMV)

# (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"**



# 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);
   }
}
```

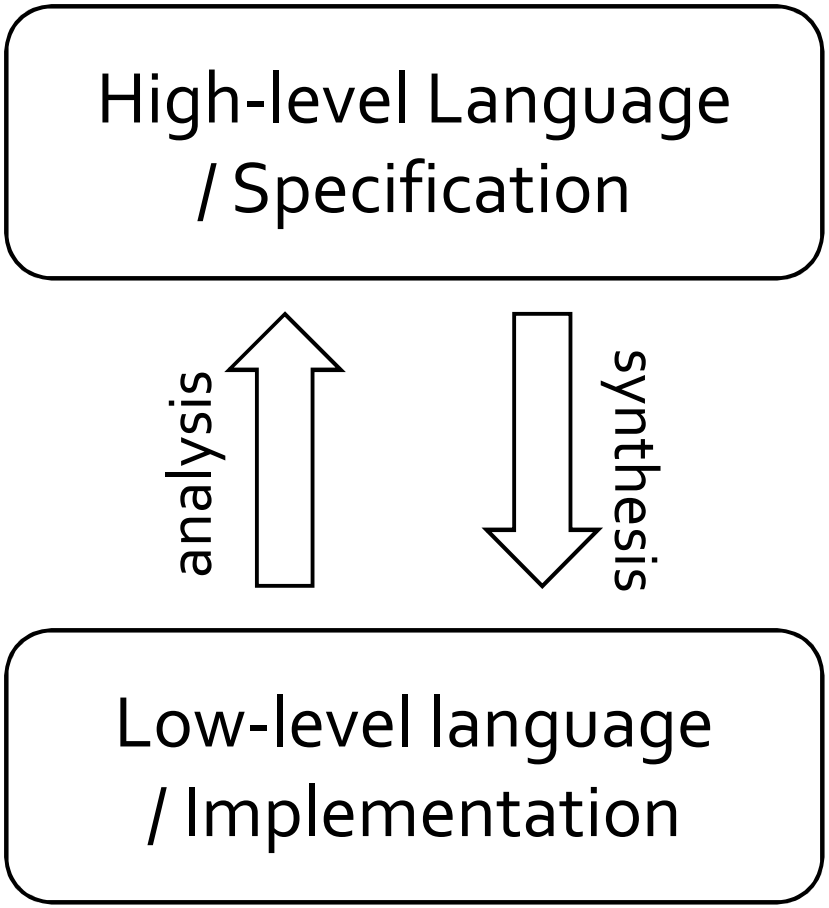
# Testing is Not Enough

- 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

# Program Analysis & Synthesis*




* informally speaking

# Static Analysis

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

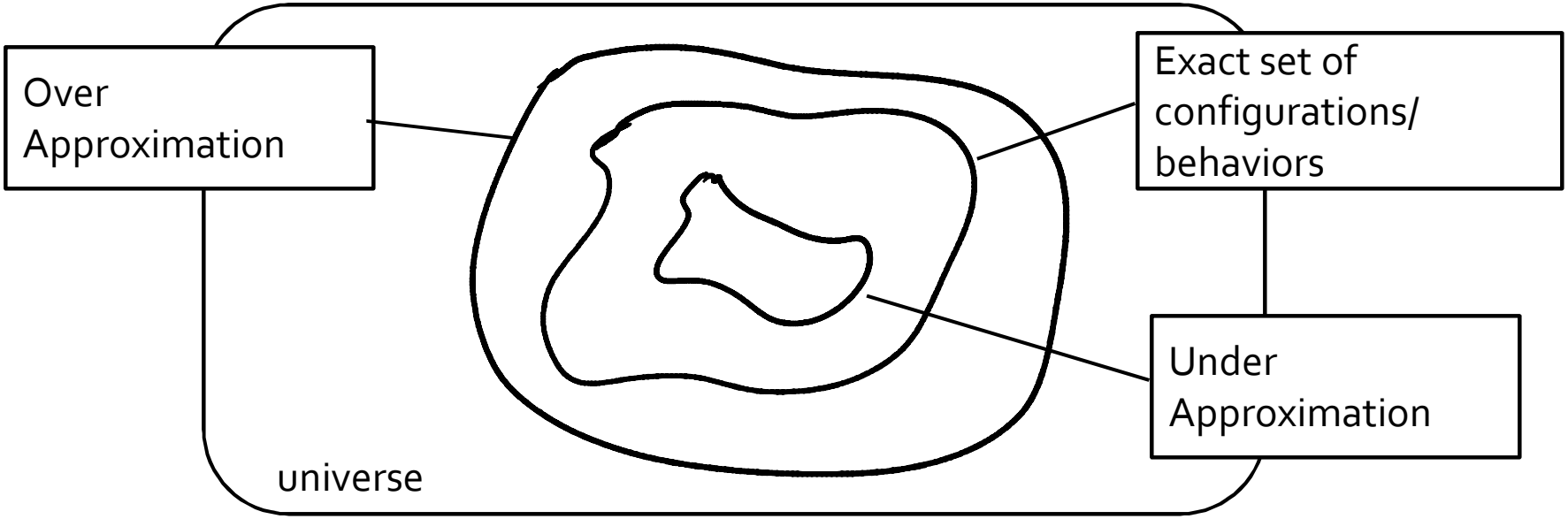
# Static Analysis

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

- Bad news: problem is generally undecidable

# Static Analysis

- Central idea: use approximation



# Over Approximation

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

- Over approximation: assertion may be violated

# Precision

```
main(…) {
 printf("assertion may be violated\n");
}
```

- Lose precision only when required
- Understand where precision is lost

# Static Analysis

- Formalize software behavior in a mathematical model (semantics)
- Prove properties of the mathematical model
  - Automatically, typically with approximation of the formal semantics
- Develop theory and tools for program correctness and robustness

# Static Analysis

- Spans a wide range
  - type checking … up to full functional verification

- General safety specifications
- Security properties (e.g., information flow)
- Concurrency correctness conditions (e.g., progress, linearizability)
- Correct use of libraries (e.g., typestate)

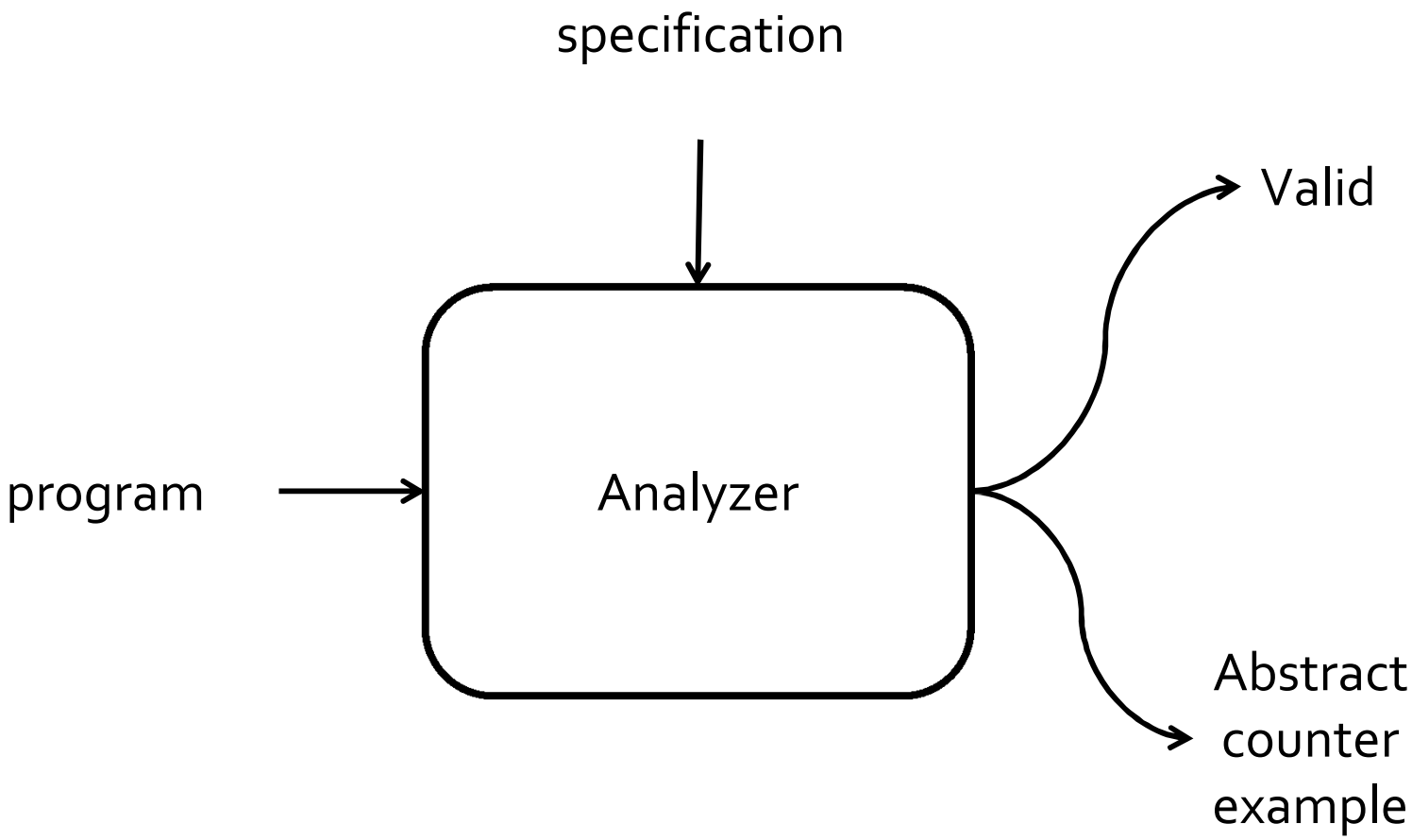
- Under-approximations useful for bug-finding, test-case generation,…

# Static Analysis: Techniques

- Abstract Interpretation
- Dataflow analysis
- Constraint-based analysis
- Type and effect systems

- (we will not be able to cover all in depth)

# Static Analysis for Verification



# Verification Challenge I

```
main(int i) {
 int x=3,y=1;

 do {
  y = y + 1;
 } while(--i > 0)
 assert 0 < x + y
}
```

Determine what states can arise during any execution

Challenge: set of states is unbounded

# Abstract Interpretation

```
main(int i) {
 int x=3,y=1;

 do {
  y = y + 1;
 } while(--i > 0)
 assert 0 < x + y
}
```

Recipe

1) Abstraction
2) Transformers
3) Determine what states can arise during any execution

Challenge: set of states is unbounded

Solution: compute a bounded representation of (a superset) of program states

# 1) Abstraction

```
main(int i) {
 int x=3,y=1;

 do {
  y = y + 1;
 } while(--i > 0)
 assert 0 < x + y
}
```

- concrete state

  $\rho$: Var $\rightarrow \mathcal{Z}$

- abstract state (sign)

  $\rho^{\#}$: Var $\rightarrow$ {+, o, -, ?}

| x | y | i |
|---|---|---|
| 3 | 1 | 7 |

| x | y | i |
|---|---|---|
| 3 | 2 | 6 |

...

| x | y | i |
|---|---|---|
| + | + | + |

# 2) Transformers

```
main(int i) {
 int x=3,y=1;

 do {
  y = y + 1;
 } while(--i > 0)
 assert 0 < x + y
}
```

- concrete transformer

| x | y | i |
|---|---|---|
| 3 | 1 | 0 |

`y = y + 1`

| x | y | i |
|---|---|---|
| 3 | 2 | 0 |

- abstract transformer

| x | y | i |
|---|---|---|
| + | + | 0 |
| + | - | 0 |
| + | 0 | 0 |
| + | ? | 0 |

`y = y + 1`

| x | y | i |
|---|---|---|
| + | + | 0 |
| + | ? | 0 |
| + | + | 0 |
| + | ? | 0 |

# 3) Exploration

```
main(int i) {
 int x=3,y=1;

 do {
  y = y + 1;
 } while(--i > 0)
 assert 0 < x + y
}
```

| x | y | i |
|---|---|---|
| ? | ? | ? |
| + | + | ? |

| x | y | i |
|---|---|---|
| + | + | ? |
| + | + | ? |
| + | + | ? |

| x | y | i |
|---|---|---|
| + | + | ? |
| + | + | ? |
| + | + | ? |

✓

# Incompleteness

```
main(int i) {
 int x=3,y=1;

 do {
  y = y - 2;
  y = y + 3;
 } while(--i > 0)
 assert 0 < x + y
}
```

| x | y | i |
|---|---|---|
| ? | ? | ? |
| + | + | ? |

| x | y | i |
|---|---|---|
| + | ? | ? |
| + | ? | ? |
| + | ? | ? |

| x | y | i |
|---|---|---|
| + | ? | ? |
| + | ? | ? |
| + | ? | ? |

✗

# Parity Abstraction

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

challenge: how to find “the right” abstraction

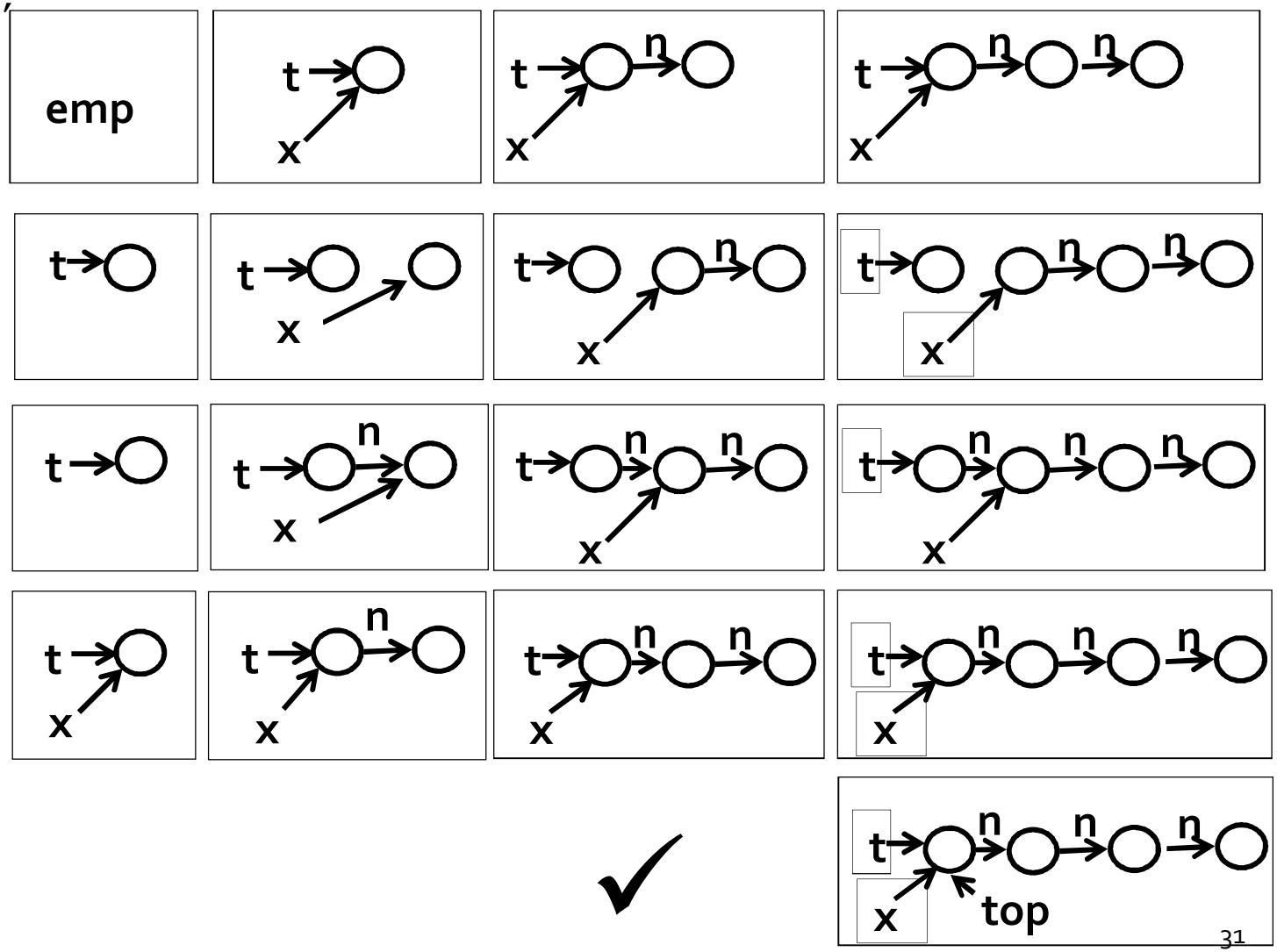
# Finding “the right” abstraction?

- pick an abstract domain suited for your property
  - numerical domains
  - domains for reasoning about the heap
  - …
- combination of abstract domains

- another approach
  - abstraction refinement

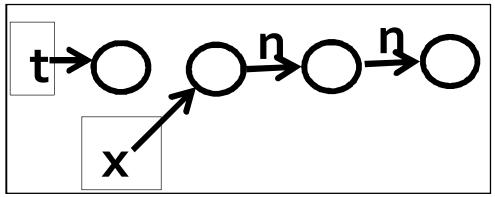
# Example: Shape (Heap) Analysis

```
void stack-init(int i) {
 Node* x = null;

 do {

 Node *t =
   malloc(…)

  t->n = x;

  x = t;
 } while(--i>0)

Top = x;
assert(acyclic(Top))
}
```

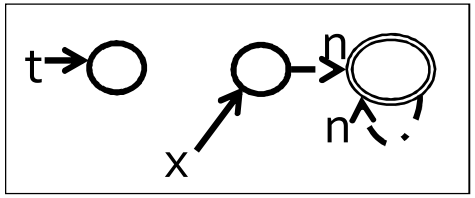


# Following the Recipe (In a Nutshell)

## 1) Abstraction

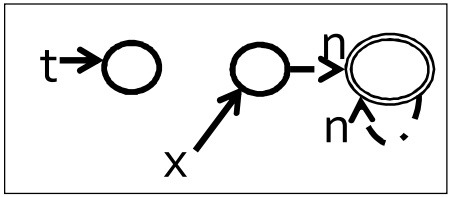



Concrete state

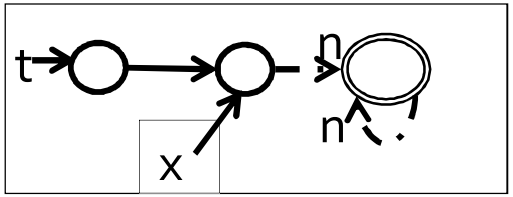



Abstract state

## 2) Transformers

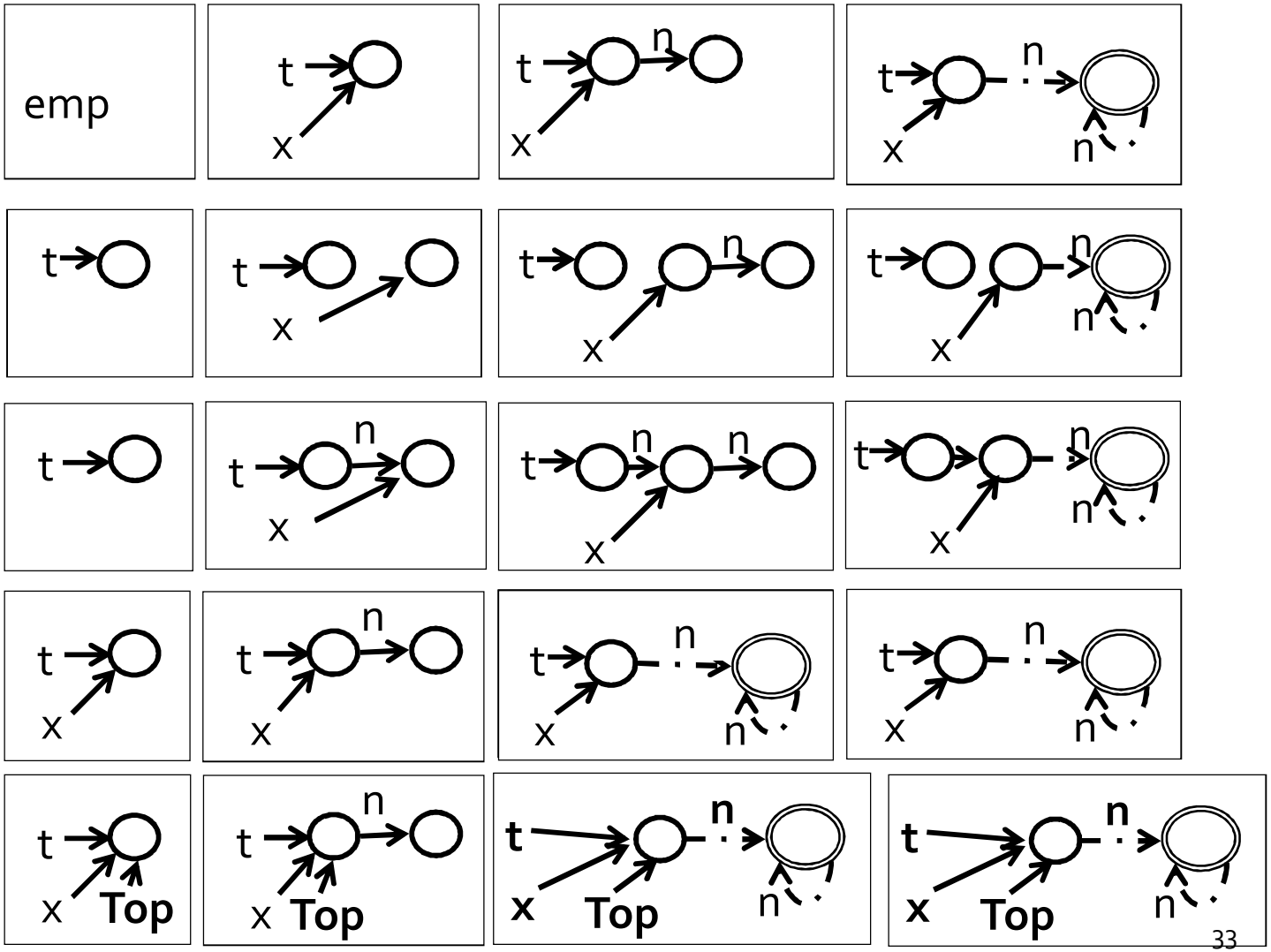



```
t->n = x
```



# 3) Exploration

```
void stack-init (int i)
   {
 Node* x = null;

 do {

 Node *t =
   malloc(…)

  t->n = x;

  x = t;
 } while(--i>0)

Top = x;
assert(acyclic(Top))

}
```



# Example: Polyhedra (Numerical) Domain

```
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
b = MC(a);
end
```

What is the result of this program?

# McCarthy 91 function

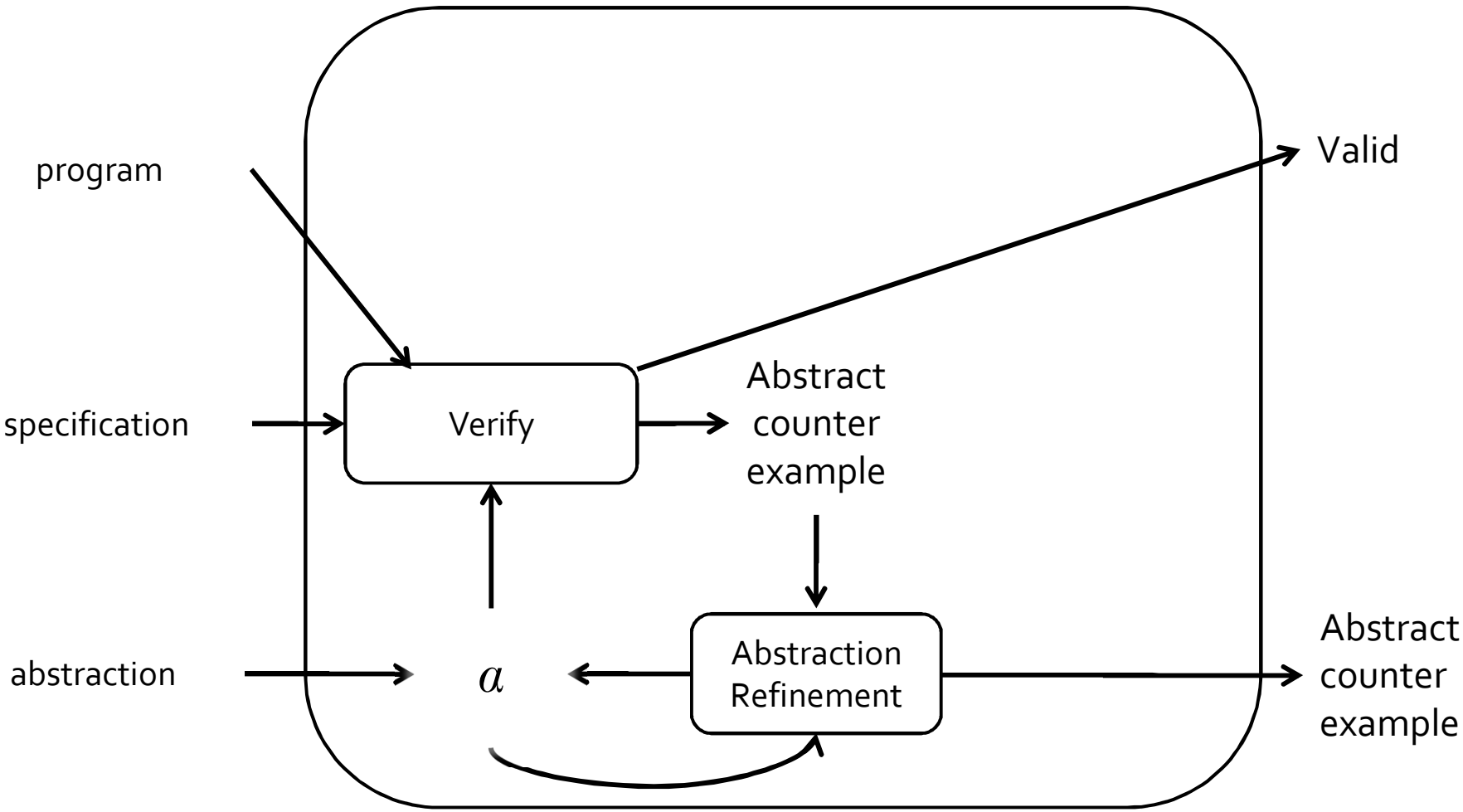
```
proc MC (n : int) returns (r : int) var t1 : int, t2 : int;
begin
                    /* (L6 C5) top */
  if n > 100 then
                    /* (L7 C17) [|n-101>=0|] */
     r = n - 10;    /* (L8 C14) [|-n+r+10=0; n-101>=0|] */
  else
                    /* (L9 C6) [|-n+100>=0|] */
    t1 = n + 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
```

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

# Some things that should trouble you

- does  a result always exist?
- does the recipe always converge?
- is the result always “the best”?
- how do I pick my abstraction?
- how do come up with abstract transformers?

# Abstraction Refinement



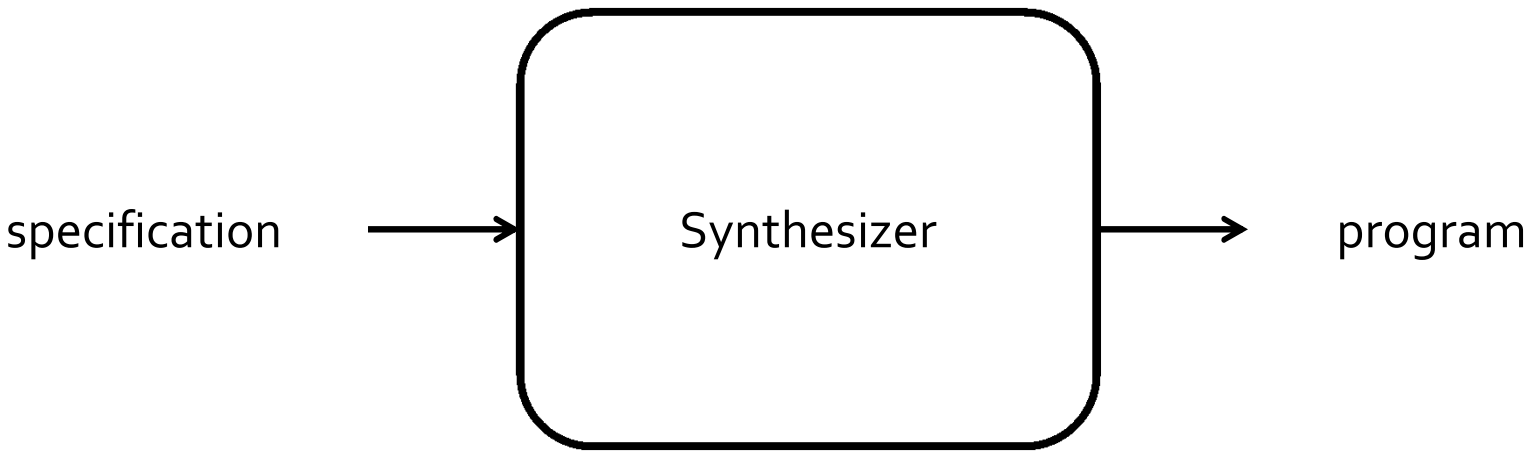

Change the **abstraction** to match the **program**

# Recap: program analysis

- Reason statically (at compile time) about the possible runtime behaviors of a program
- use sound over-approximation of program behavior
- abstract interpretation
  - abstract domain
  - transformers
  - exploration (fixed-point computation)
- finding the right abstraction?

# Program Synthesis

- Automatically synthesize a program that is correct-by-construction from a (higher-level) specification



# Program Synthesis: Techniques

- Gen/Test
- Theorem Proving
- Games
- SAT/SMT Solvers
- Transformational Synthesis
- Abstract Interpretation
- …
- (we will not be able to cover all in depth)

# Synthesis Challenge I

```
signum(int x) {
   if (x>0) return 1;
   else if (x<0) return -1;
   else return 0;
}
```

Challenge: Generate efficient assembly code for "signum"

```
# x in d0
add.l d0, d0   | add d0 to itself
subx.l d1,d1   | subtract (d1+carry) from d1
negx.l d0      | put (0-d0-carry) into d0
addx.l d1, d1  | add (d1+carry) to d1
# signum(x) is now in d1
```

# Superoptimizer

[Massalin, 1987]

- exhaustive search over assembly programs
- order search by increasing program length
- check input/output "equivalence" with original code
  - boolean test – construct boolean formula for functions and compare them
    - not practical
  - probabilistic test – run many times on some inputs and check if the outputs of both programs are the same
- expensive, only applied to critical pieces of code (e.g., common libraries)

# Denali Superoptimizer

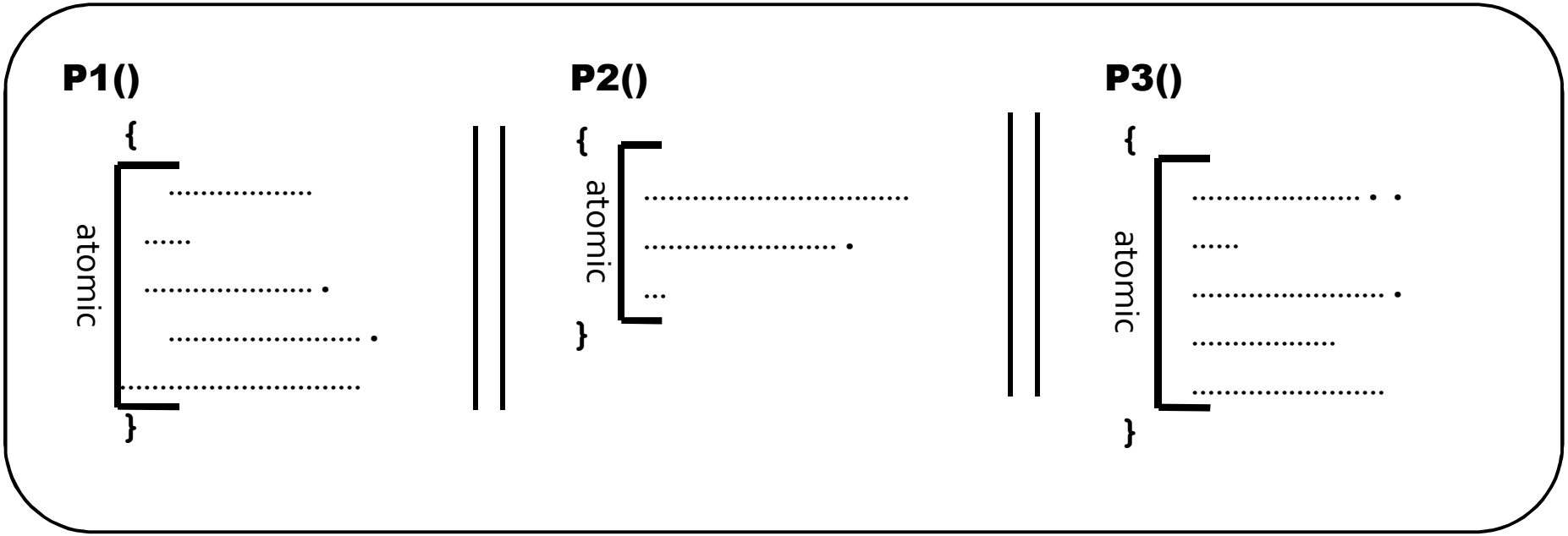
[Joshi, Nelson, Randall, 2001]

*"a refutation-based automatic theorem-prover is in fact a general-purpose goal-directed search engine, which can perform a goal-directed search for anything that can be specified in its declarative input language. Successful proofs correspond to unsuccessful searches, and vice versa."*

- Turn the search of a program into a search of counter-example in a theorem prover

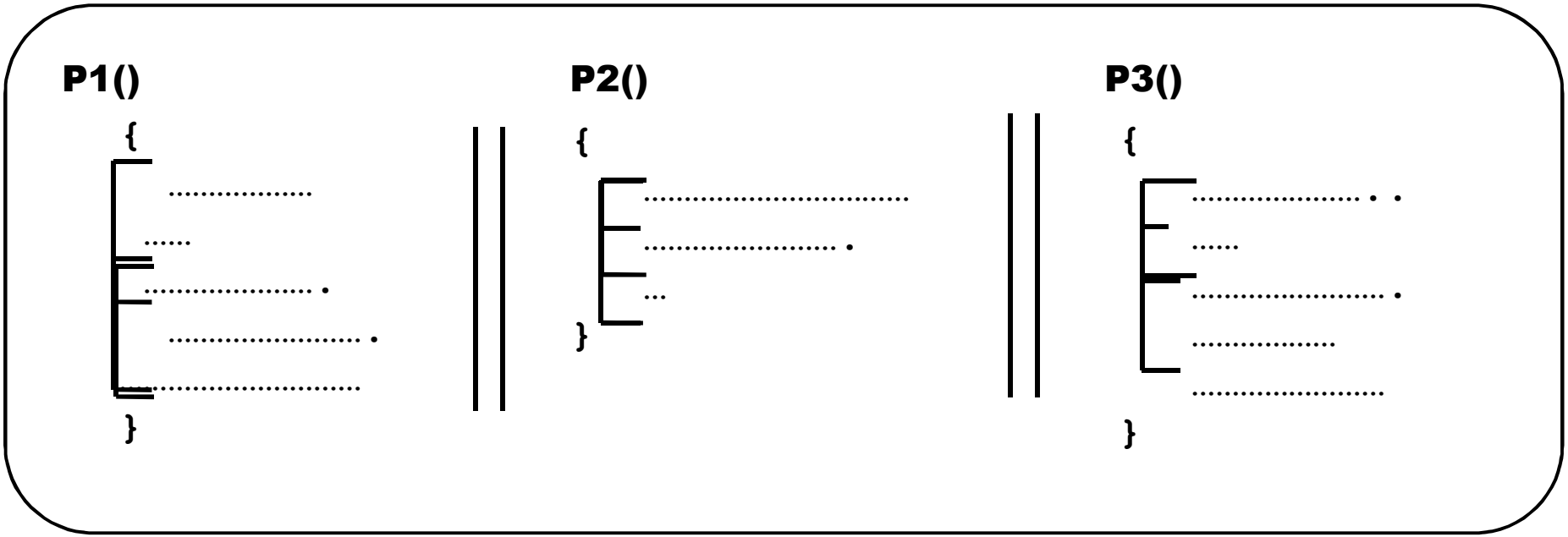
(more details later in the course…)

# Synthesis of Atomic Sections




Safety Specification: S

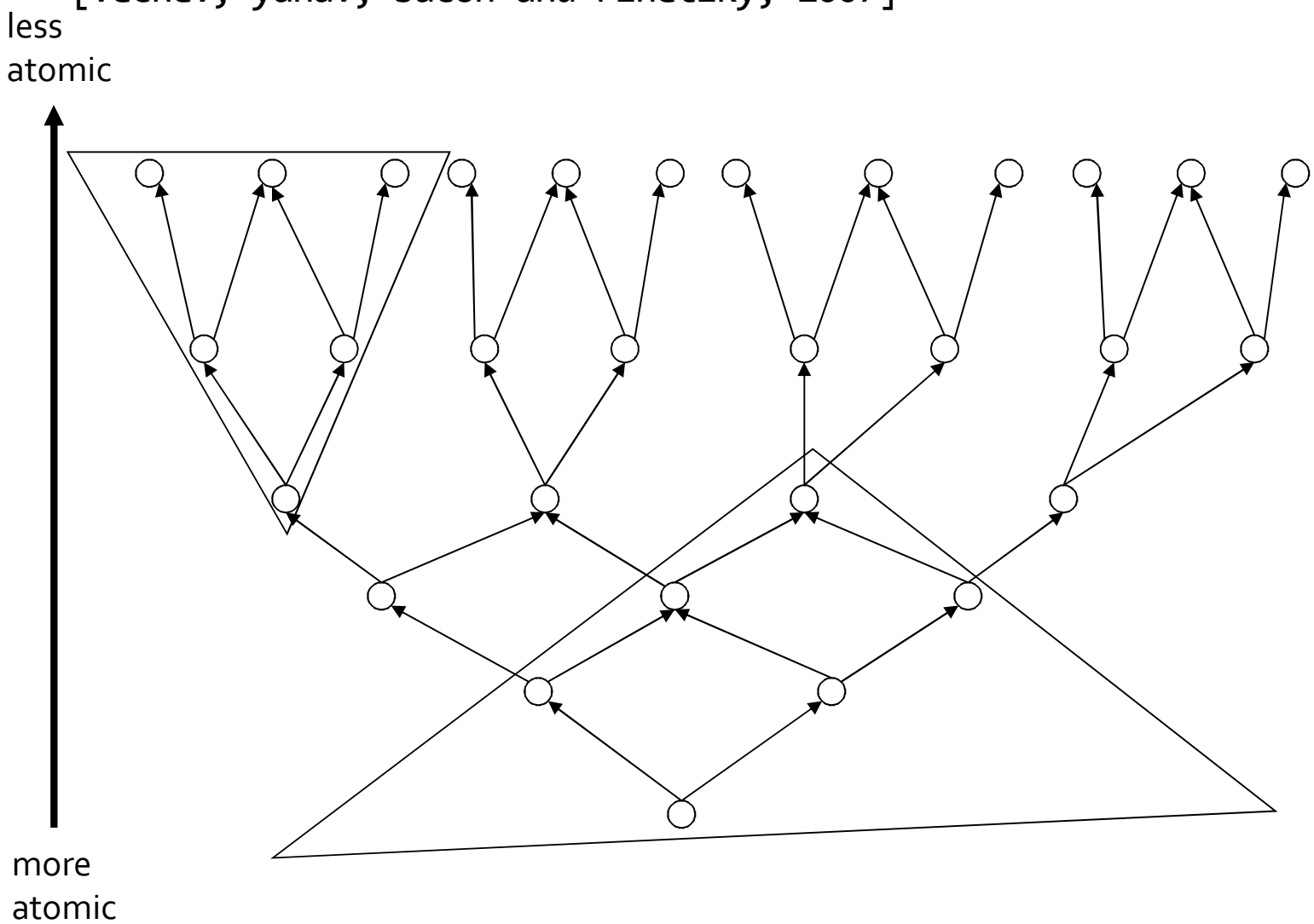
# Synthesis of Atomic Sections




Safety Specification: S

# Semantic Optimized Search

[vechev, yahav, bacon and rinetzky, 2007]



# Program Repair as a Game

[Jobstmann et. al. 2005]

```
   unsigned int got_lock = 0;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
4:     got_lock++;
     }
     ...
5:   if (got_lock != 0){
6:     unlock();
     }
7:   got_lock--;
     ...
   }
```

```
lock() {
lock: LOCK:=1;}
unlock(){
unlock: LOCK:=0;}
```

**Specification**

P1: do not acquire a lock twice

P2: do not call unlock without holding the lock

**P1: always**( line=lock **implies**
  **next**( line!=lock **w-until** line=unlock ))

**P2:** ( line!=unlock **w-until** line=lock )) **and**
  **always**( line=unlock **implies**
    **next**( line!=unlock **w-until** line=lock ))

(slide adapted with permission from Barbara Jobstmann)

# How to Repair a Reactive System?

1. Add freedom
   - choice for the system, space of permitted modifications to the system
2. Source code → transition system (**game**)
   - non-determinism in the program (demonic)
   - non-determinism in permitted modification (angelic)
3. Specification → monitor acceptance
4. Check if we can find system choices s.t. model is accepted by monitor
   - product of trans. system and monitor
   - search for winning strategy in game

(slide adapted with permission from Barbara Jobstmann)

# Repaired Program

```
   unsigned int got_lock = 0;
   ...
1: while(*) {
     ...
2:   if (*) {
3:     lock();
4:     got_lock = 1;
     }
     ...
5:   if (got_lock != 0){
6:     unlock();
     }
7:   got_lock = 0;
     ...
   }
```

```
lock() {
lock: LOCK:=1;}
unlock(){
unlock: LOCK:=0;}
```

**Specification**

P1: do not acquire a lock twice

P2: do not call unlock without holding the lock

**P1: always**( line=lock **implies**
**next**( line!=lock **w-until** line=unlock ))

**P2:** ( line!=unlock **w-until** line=lock )) **and**
**always**( line=unlock **implies**
**next**( line!=unlock **w-until** line=lock ))

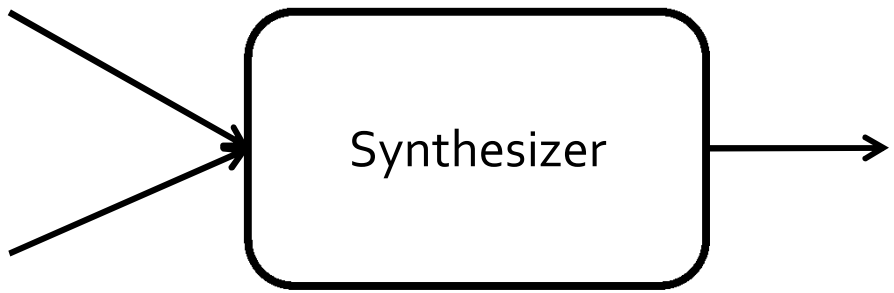
(slide adapted with permission from Barbara Jobstmann)

# Partial Programs and SKETCH

[aLisp: Andre et al 2002, Sketch: Solar-Lezama et al 2006]

- partial program ≈ freedom in games
  - defines a space of program
- Given a partial program P with control variables C ("holes"), a specification S, the goal is to find an assignment for C such that $P[C] \models S$

```
double(x) {
return ?? * x;
}

double(x) {
return x + x;
}
```




```
double(x) {
return 2 * x;
}
```

# SKETCH: isolate rightmost 0

```
bit[W] isolate0 (bit[W] x) { // W: word size
  bit[W] ret=0;
  for (int i = 0; i < W; i++)
    if (!x[i]) { ret[i] = 1; break; }
  return ret;
}

bit[W] isolate0Sketched(bit[W] x) implements isolate0 {
  return ~(x + ??) &  (x  + ??);
}


bit[W] isolate0Fast (bit[W] x) implements isolate0 {
  return ~x & (x+1);
}
```

(Hacker's Delight, H.S. Warren)

# Synthesis as generalized SAT

- The sketch synthesis problem

$$\exists c\ \forall x\ \mathrm{spec}(x) = \mathrm{sketch}(x,c)$$

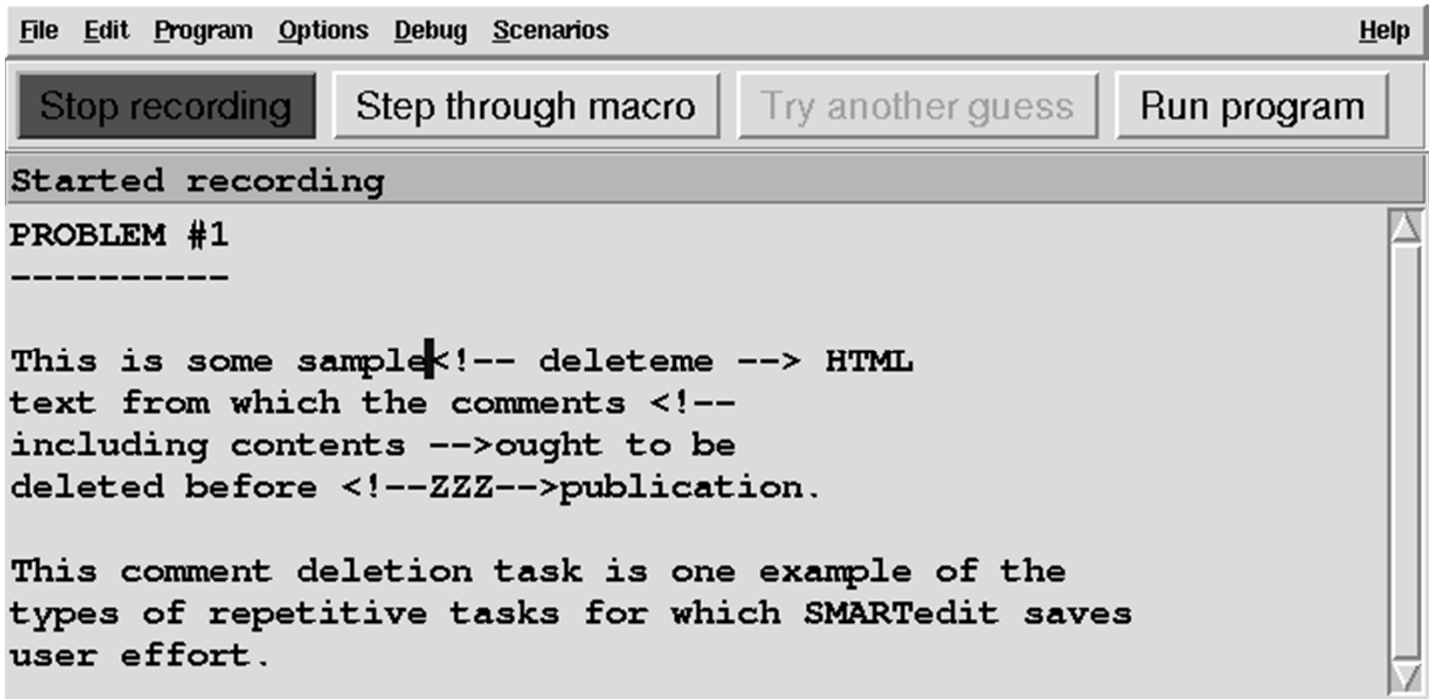
- Counter-example driven solver

```
I = ∅
x = random-input()
do
 I = I ∪ {x}
 find c such that ⋀i∈I (spec(i)=sketch(c,i))
 if cannot find c then exit("non-satisfiable sketch")
 find x such that spec(x) ≠ sketch(x,c)
while x != nil
return c
```

# SMARTEdit

[Lau et al, 2000]

- synthesize editor macros (programs) from examples
- behind the scenes: machine learning techniques



File Edit Program Options Debug Scenarios Help

Stop recording | Step through macro | Try another guess | Run program

Started recording

```
PROBLEM #1
----------

This is some sample HTML
text from which the comments <!--
including contents -->ought to be
deleted before <!--ZZZ-->publication.

This comment deletion task is one example of the
types of repetitive tasks for which SMARTedit saves
user effort.
```

File Edit Program Options Debug Scenarios Help

Record iteration | Step through macro | Try another guess | Run program

Step 1 of 2 is Move to the end of 'ample' [36% prob.]

```
PROBLEM #1
----------

This is some sample HTML
text from which the comments <!--
including contents -->ought to be
deleted before <!--ZZZ-->publication.

This comment deletion task is one example of the
types of repetitive tasks for which SMARTedit saves
user effort.
```

File Edit Program Options Debug Scenarios Help

Record iteration | Step through macro | Try another guess | Run program

```
Step 1 of 2 is Move to the start of '<!--' [26% prob.]
PROBLEM #1
----------

This is some sample HTML
text from which the comments <!--
including contents -->ought to be
deleted before <!--ZZZ-->publication.

This comment deletion task is one example of the
types of repetitive tasks for which SMARTedit saves
user effort.
```

File Edit Program Options Debug Scenarios Help

Record iteration | Step through macro | Try another guess | Run program

```
Step 2 of 2 is Delete from here to the end of ' -->' [26% ...
PROBLEM #1
----------

This is some sample HTML
text from which the comments <!--
including contents -->ought to be
deleted before <!--ZZZ-->publication.

This comment deletion task is one example of the
types of repetitive tasks for which SMARTedit saves
user effort.
```

File Edit Program Options Debug Scenarios Help

Record iteration | Step through macro | Try another guess | Run program

```
Step 1 of 2 is Move to the start of '<!--' [93% prob.]
PROBLEM #1
----------

This is some sample HTML
text from which the comments ought to be
deleted before <!--ZZZ-->publication.

This comment deletion task is one example of the
types of repetitive tasks for which SMARTedit saves
user effort.
```

File Edit Program Options Debug Scenarios Help

Record iteration | Step through macro | Try another guess | Run program

```
Ending program.
PROBLEM #1
----------

This is some sample HTML
text from which the comments ought to be
deleted before publication.

This comment deletion task is one example of the
types of repetitive tasks for which SMARTedit saves
user effort.
```

# Recap: program synthesis

- Automatically synthesize a program that is correct-by-construction from a (higher-level) specification
- many techniques
  - games
  - games with abstraction (abstract interpretation)

# Coming up

(extremely optimistic! more likely, we'll cover half of it)

- principle of program analysis
  - overview of dataflow
  - why dataflow works?
  - abstract interpretation basics
  - a taste of operational semantics
  - numerical domains
  - heap domains
  - shape analysis
- approaches to program synthesis
  - program synthesis using games
  - abstraction-guided synthesis
  - games with abstraction
  - synthesis with machine learning techniques
  - a tiny bit on SAT/SMT based synthesis

# References

- Patriot bug:
  - http://www.cs.usyd.edu.au/~alum/patriot_bug.html
  - Patrick Cousot's NYU lecture notes
- Zune bug:
  - http://www.crunchgear.com/2008/12/31/zune-bug-explained-in-detail/
- Blaster worm:
  - http://www.sans.org/security-resources/malwarefaq/w32_blasterworm.php
- Interesting CACM article
  - http://cacm.acm.org/magazines/2010/2/69354-a-few-billion-lines-of-code-later/fulltext
- http://journals.cambridge.org/download.php?file=%2FMSC%2FMSC19_05%2FS0960129509990041a.pdf&code=d5af66869c1881e31339879b90c07d0c