



Programming with Constraint Solvers

CS294: Program Synthesis for Everyone

Ras Bodik
Emina Torlak

Division of Computer Science
University of California, Berkeley

Today

Today: we describe four programming problems that a satisfiability constraint solver can mechanize once the program is translated to a logical formula.

Next lecture: translation of programs to formulas.

Subsequent lecture: solver algorithms.

Z3 files for this lecture can be found in <http://www.cs.berkeley.edu/~bodik/cs294/fa12/Lectures/L2/z3-encodings/>

Outline

Recap: the versatile solver

- solver as interpreter, inverter, synthesizer

Specifications, briefly revisited

- from ϕ to pre- and post-conditions

Four programming problems solvable with solvers

- verification; fault localization; synthesis; angelic programming
- constructing formulas for the four problems
- decomposing programs into assertions

Advanced challenge

Is this lecture familiar material? Entertain yourself by thinking through how to carry out this style of program reasoning for programming models other than functional, eg:

- imperative
- Datalog
- Prolog
- attribute grammars
- distributed and concurrent programming
- combination of the above, eg concurrent Prolog

Recall L1: program as a formula

Assume a formula $S_P(x,y)$ which holds iff program $P(x)$ outputs value y

program: $f(x) \{ \text{return } x + x \}$

formula: $S_f(x,y): y = x + x$

With program as a formula, solver is versatile

Solver as an **interpreter**: given x , evaluate $f(x)$

$$S(x, y) \wedge x = 3 \quad \text{solve for } y \quad y \mapsto 6$$

Solver as a execution **inverter**: given $f(x)$, find x

$$S(x, y) \wedge y = 6 \quad \text{solve for } x \quad x \mapsto 3$$

This solver “bidirectionality” enables **synthesis**

Search of candidates as constraint solving

$S_P(x, h, y)$ holds iff sketch $P[h](x)$ outputs y .

`spec(x) { return x + x }`

`sketch(x) { return x << ?? }` $S_{\text{sketch}}(x, y, h): y = x * 2^h$

The solver computes h , thus synthesizing a program correct for the given x (here, $x=2$)

$S_{\text{sketch}}(x, y, h) \wedge x = 2 \wedge y = 4$ solve for h $h \mapsto 1$

Sometimes h must be constrained on several inputs

$S(x_1, y_1, h) \wedge x_1 = 0 \wedge y_1 = 0 \wedge$

$S(x_2, y_2, h) \wedge x_2 = 3 \wedge y_2 = 6$ solve for h $h \mapsto 1$

Specifications

From ϕ to pre- and post-conditions:

A precondition (denoted $pre(x)$) of a procedure f is a predicate (Boolean-valued function) over f 's parameters x that always holds when f is called.

f can assume that pre holds

A postcondition ($post(x, y)$) is a predicate over parameters of f and its return value y that holds when f returns

f ensures that $post$ holds

pre and post conditions

Facilitate modular reasoning

- so called “assume/ guarantee”

Pre/postconditions can express multimodal specs

- invariants,
- input/output pairs,
- traces,
- equivalence to another program

modern programming

write spec,
then
implement!

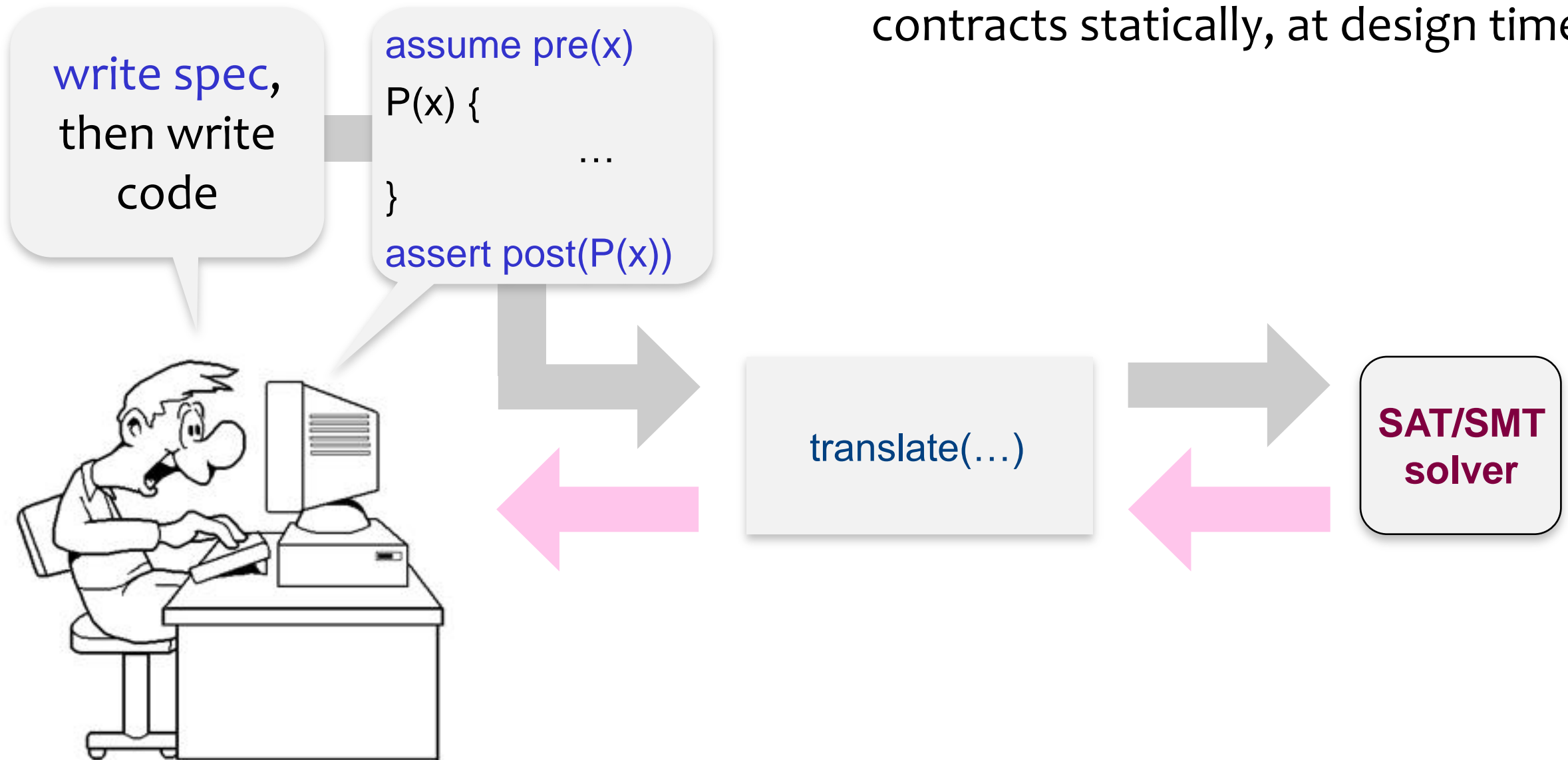
```
assume pre(x)  
P(x) {  
    ...  
}  
assert post(P(x))
```



pre- and post-conditions are known as *contracts*. They are supported by modern languages and libraries, including Racket. Usually, these contracts are tested (ie, evaluated dynamically, during execution).

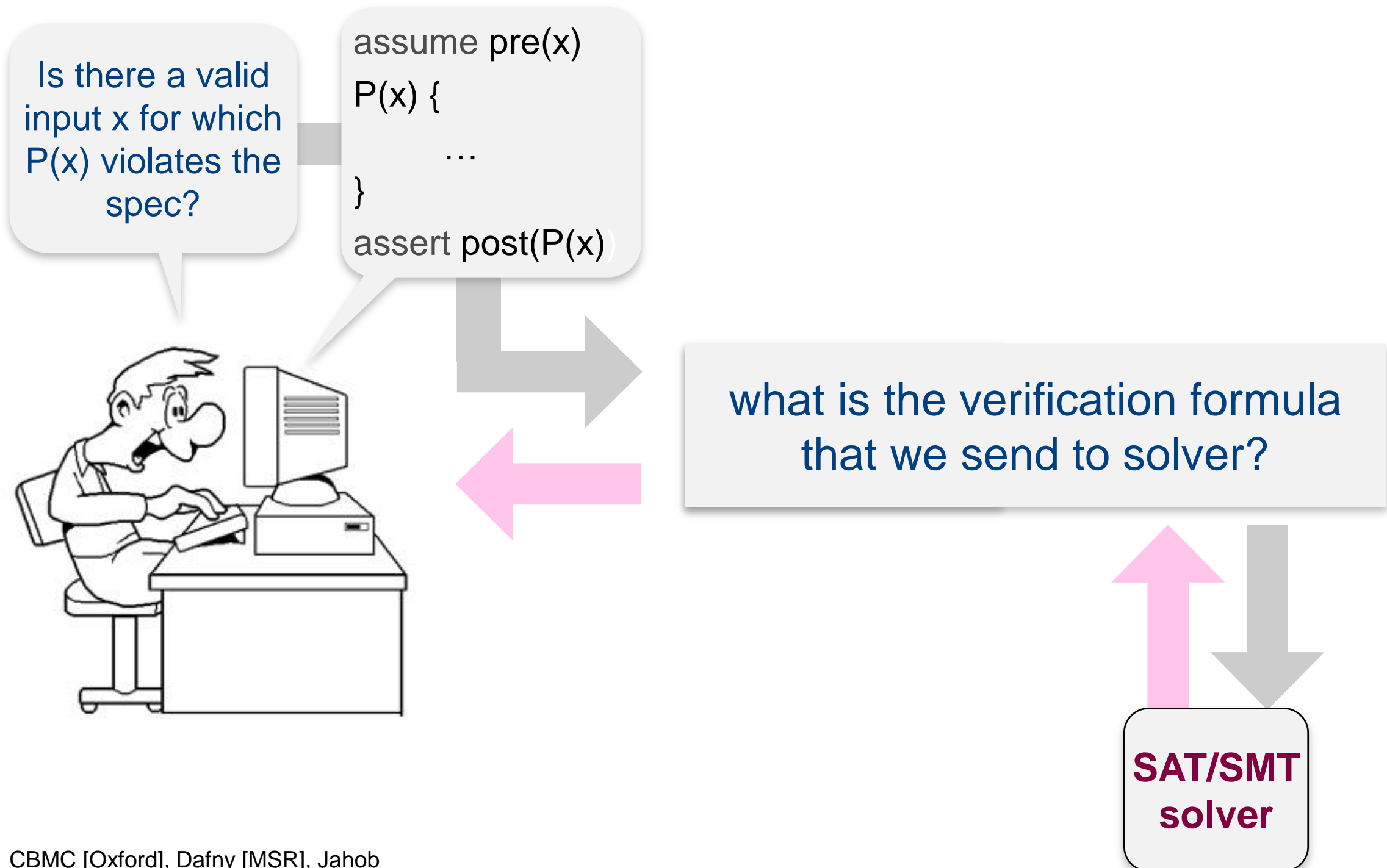
modern programming with a solver

With solvers, we want to test these contracts statically, at design time.



Verification

programming with a solver: **verification**



Background: satisfiability solvers

A satisfiability solver accepts a formula $\phi(x, y, z)$ and checks if ϕ is satisfiable (SAT).

If yes, the solver returns a model m , a valuation of x, y, z that satisfies ϕ , ie, m makes ϕ true.

If the formula is unsatisfiable (UNSAT), some solvers return minimal unsat core of ϕ , a smallest set of clauses of ϕ that cannot be satisfied.

$$\phi_1 \wedge \boxed{\phi_2 \wedge \phi_3} \wedge \dots \wedge \phi_n$$

SAT vs. SMT solvers

SAT solvers accept propositional Boolean formulas
typically in CNF form

SMT (satisfiability modulo theories) solvers accept
formulas in richer logics, eg uninterpreted functions,
linear arithmetic, theory of arrays
more on these in the next lecture

Code checking (verification)

Correctness condition ϕ says that the program is correct for all valid inputs:

$$\forall x . \underbrace{pre(x)}_{\text{valid}} \Rightarrow \underbrace{SP(x, y)}_{\substack{\text{"compute"} \\ y \text{ from } x}} \wedge \underbrace{post(x, y)}_{\text{correct}}$$

How to prove correctness for all inputs x ? Search for *counterexample* x where ϕ does not hold.

$$\exists x . \neg (pre(x) \Rightarrow SP(x, y) \wedge post(x, y))$$

Verification condition

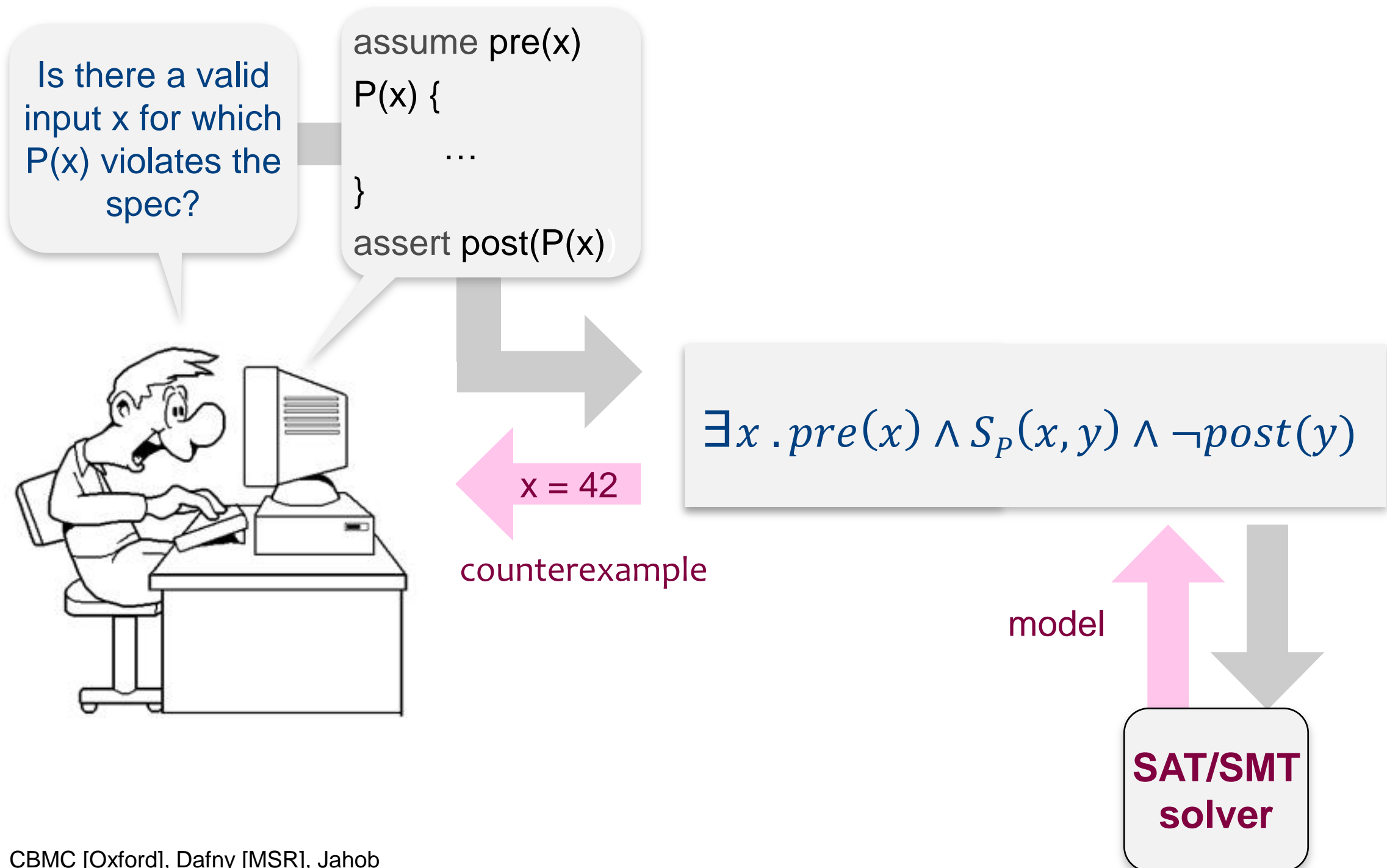
Some simplifications:

$$\begin{aligned} & \exists x . \neg(pre(x) \Rightarrow SP(x, y) \wedge post(x, y)) \\ & \exists x . pre(x) \wedge \neg(SP(x, y) \wedge post(x, y)) \end{aligned}$$

S_p always holds (we can always find y given x since S_p encodes program execution), so the verification formula is:

$$\exists x . pre(x) \wedge S_p(x, y) \wedge \neg post(x, y)$$

programming with a solver: code checking



Example: verifying a triangle classifier

Triangle classifier in Rosette (using the Racket lang):

```
(define (classify a b c)
  (if (and (>= a b) (>= b c))
      (if (or (= a c) (= b c))
          (if (and (= a b) (= a c))
              'EQUILATERAL
              'ISOSCELES)
          (if (not (= (* a a) (+ (* b b) (* c c))))
              (if (< (* a a) (+ (* b b) (* c c)))
                  'ACUTE
                  'OBTUSE)
              'RIGHT))
      'ILLEGAL))
```

This classifier contains a bug.

Specification for classify

pre(*a*, *b*, *c*):

$$a, b, c > 0 \wedge a < b + c$$

post(*a*, *b*, *c*, *y*):

- where *y* is return value from `classify(a, b, c)`
- we'll specify *post* functionally, with a correct implementation of `classify`. Think of alternative ways to specify the classifier.

Verification formula for Z3 (and other solvers for SMT2 standard)

```
(declare-datatypes () ((TriangleType EQUILATERAL ISOSCELES ACUTE  
OBTUSE RIGHT ILLEGAL)))
```

; this is the formula buggy triangle classifier

```
(define-fun classify ((a Int)(b Int)(c Int)) TriangleType  
  (if (and (>= a b) (>= b c))  
    (if (or (= a c) (= b c))  
      (if (and (= a b) (= a c))  
        EQUILATERAL  
        ISOSCELES)  
      (if (not (= (* a a) (+ (* b b) (* c c))))  
        (if (< (* a a) (+ (* b b) (* c c)))  
          ACUTE  
          OBTUSE)  
        RIGHT))  
    ILLEGAL))
```

Continued

; precondition: triangle sides must be positive and
; must observe the triangular inequality

```
(define-fun pre ((a Int)(b Int)(c Int)) Bool
  (and (> a 0)
        (> b 0)
        (> c 0)
        (< a (+ b c))))
```

; our postcondition is based on a debugged version of classify

```
(define-fun spec ((a Int)(b Int)(c Int)) TriangleType
  ... ; a correct implementation comes here
)
```

```
(define-fun post ((a Int)(b Int)(c Int)(y TriangleType)) Bool
  (= y (spec a b c)))
```

Continued

; the verification condition

```
(declare-const x Int)
```

```
(declare-const y Int)
```

```
(declare-const z Int)
```

```
(assert (and (pre x y z)
```

```
            (not (post x y z (classify x y z))))))
```

```
(check-sat)
```

```
(get-model)
```

See file classifier-verification.smt2 in the Lecture 2 directory.

Output from the verifier is a of formula

Model of verification formula = counterexample input

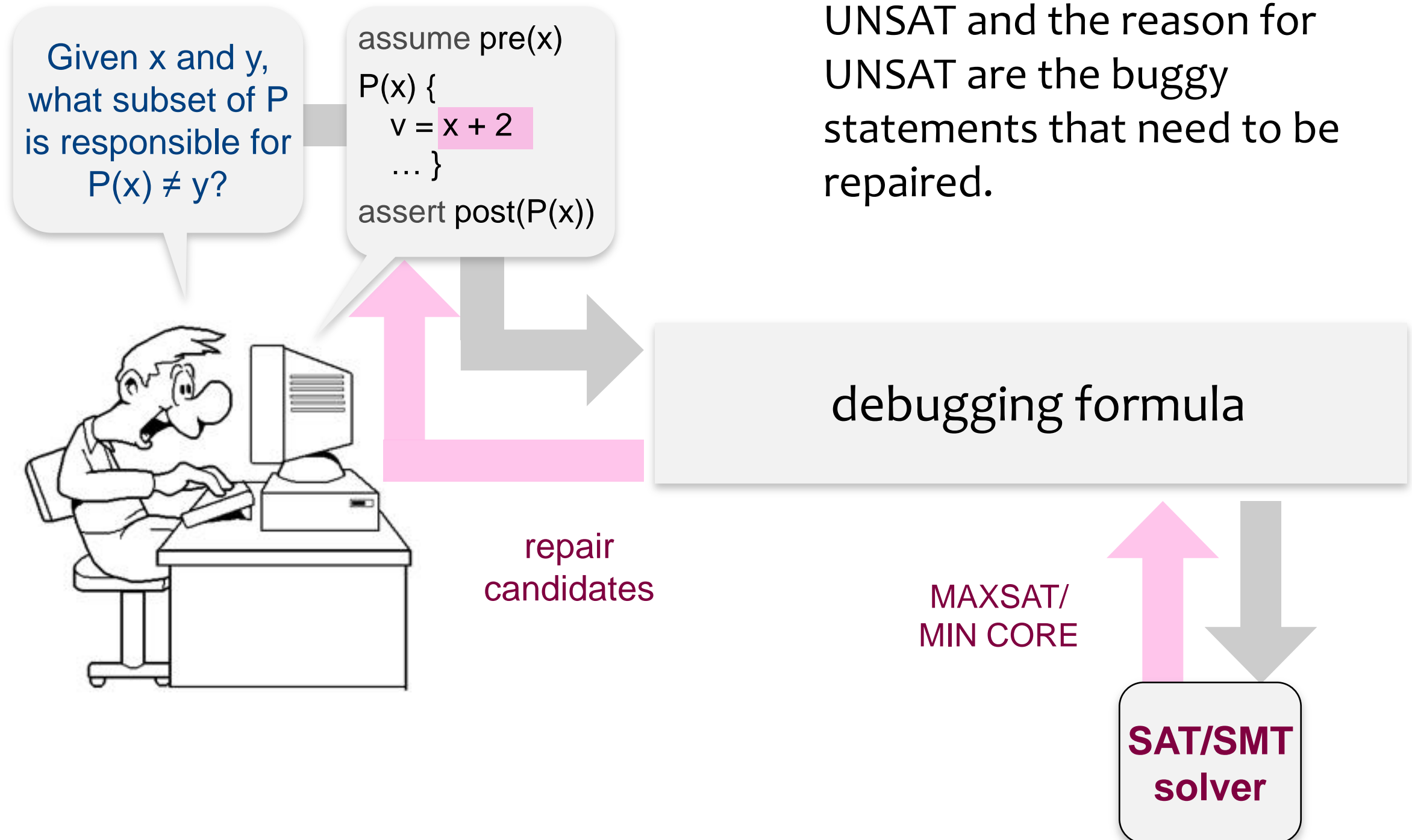
```
sat
(model
  (define-fun z () Int
    1)
  (define-fun y () Int
    2)
  (define-fun x () Int
    2)
)
```

This counterexample input *refutes* correctness of classify

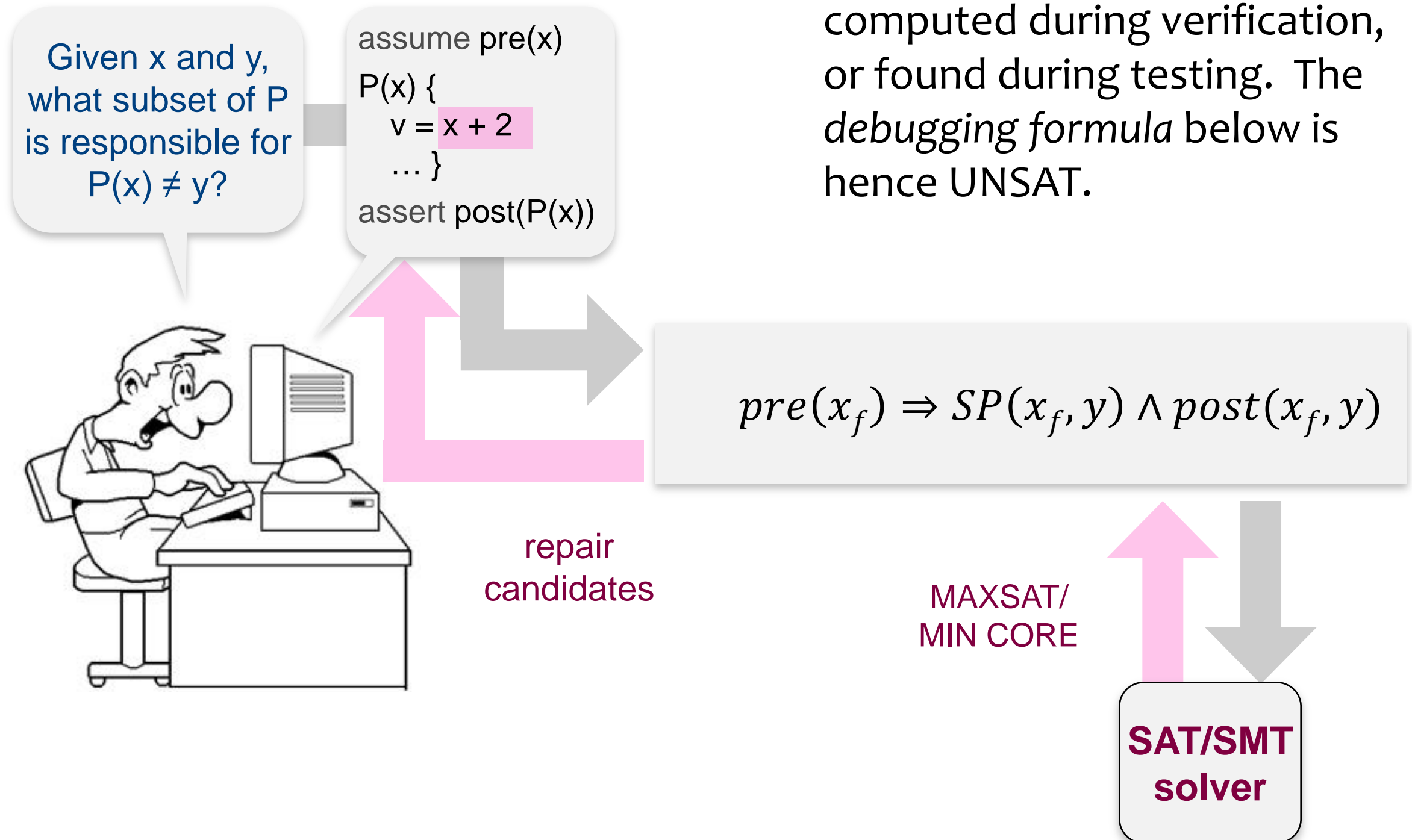
Debugging

programming with a solver: debugging

We need a formula that is UNSAT and the reason for UNSAT are the buggy statements that need to be repaired.



programming with a solver: debugging



Computing unsat core in Z3

We can give names to top-level assertions

```
(assert (! (EXPR) :named NAME))
```

Z3 gives the unsat core as a subset of named assertions. Dropping any of these assertions makes the formula satisfiable.

Debugging formula in Z3 (Step 1)

We need to decompose the function `classify` into small parts to see which of them are in the unsat core. Each “part” will be one assertion. First, we inline the function call by assigning values to “globals” `a`, `b`, `c`. This make the formula a top-level assertion.

```
(set-option :produce-unsat-cores true)

(declare-const a Int) (assert (= a 2)) ; a, b, c are the failing input
(declare-const b Int) (assert (= b 2)) ; this input was computed during
(declare-const c Int) (assert (= c 1)) ; verification

(assert (! (= ISOSCELES ; ISOSCELES is the expected output for 2,2,1
  (if (and (>= a b) (>= b c))
    (if (! (or (= a c) (= b c)) :named a2)
      (if (! (and (= a b) (= a c)) :named a3)
        EQUILATERAL
        ISOSCELES)
      (if (not (= (* a a) (+ (* b b) (* c c))))
        (if (< (* a a) (+ (* b b) (* c c))) ACUTE OBTUSE)
        RIGHT))
    ILLEGAL)) :named a1))

(check-sat)
(get-unsat-core) ; for details, see file classifier-unsat-core-1-course-grain.smt2
```

why are we not naming these assertions?

Debugging formula in Z3 (Step 2)

We now break the large expression into smaller assertions using temporary variables t1 and t2.

```
(declare-const a Int) (assert (= a 26))
(declare-const b Int) (assert (= b 26))
(declare-const c Int) (assert (= c 7))

(declare-const t1 Bool) (assert (! (= t1 (or (= a c) (= b c))) :named a1))
(declare-const t2 Bool) (assert (! (= t2 (and (= a b) (= a c))) :named a2))

(assert (= ISOSCELES
  (if (and (>= a b) (>= b c))
    (if t1
      (if t2
        EQUILATERAL
        ISOSCELES)
      (if (not (= (* a a) (+ (* b b) (* c c))))
        (if (< (* a a) (+ (* b b) (* c c))) ACUTE OBTUSE) RIGHT))
    ILLEGAL)))

(check-sat)
(get-unsat-core) ; -> Unsat core is (a1), the list of one assertion named a1.
```

Discussion

Unsat core comprises of the sole assertion a1.

Commenting out the assertion a1 makes the formula SAT. (Try it!)

In other words, the program becomes correct on the failing input used in computing the core (2,2,1).

The execution on (2,2,2) became correct because commenting out a1 makes t1 unconstrained, which allows the solver to pick any value for t1. It picks a value that makes the program correct on this execution.

Assertion is a repair candidate because we want to change the code that computes the value of t1.

Buggy classifier bug identified via unsat core

This code is in the source language (Racket):

```
(define (classify a b c)
  (if (and (>= a b) (>= b c))
      (if (or (= a c) (= b c))
          (if (and (= a b) (= a c))
              'EQUILATERAL
              'ISOSCELES)
          (if (not (= (* a a) (+ (* b b) (* c c))))
              (if (< (* a a) (+ (* b b) (* c c)))
                  'ACUTE
                  'OBTUSE)
              'RIGHT))
      'ILLEGAL))
```


Unsat core depends on how we name asserts

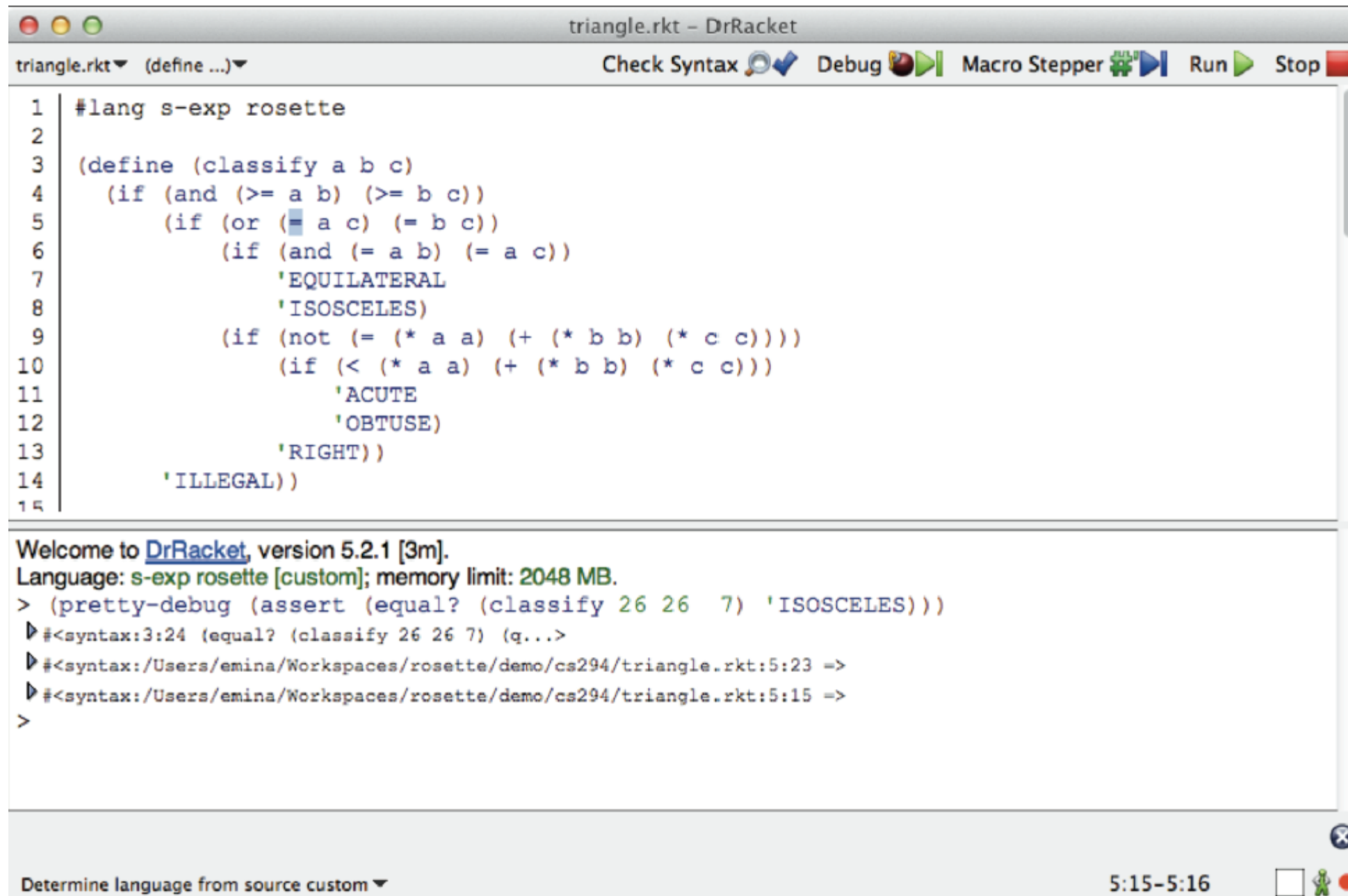
Note: If we broke the assertion a1 further, the core would contain three assertions, underlined:

```
(define (classify a b c)
  (if (and (>= a b) (>= b c))
      (if (or (= a c) (= b c))
          (if (and (= a b) (= a c))
              'EQUILATERAL
              'ISOSCELES)
          (if (not (= (* a a) (+ (* b b) (* c c))))
              (if (< (* a a) (+ (* b b) (* c c)))
                  'ACUTE 'OBTUSE) 'RIGHT))
      'ILLEGAL))
```

Changing the value of the or expression, or either of the equalities, can rescue this failing run.

Mapping unsat core back to source code

This is how Rosette maps the unsat to src.



The screenshot shows the DrRacket IDE with a file named 'triangle.rkt'. The editor contains a Rosette program that defines a 'classify' function. The function takes three arguments 'a', 'b', and 'c' and returns a string representing the type of triangle based on the lengths of its sides. The output window shows the execution of the program, including a welcome message and the execution of an assertion that the triangle with sides 26, 26, and 7 is classified as 'ISOSCELES'.

```
triangle.rkt - DrRacket
triangle.rkt (define ...)
Check Syntax Debug Macro Stepper Run Stop

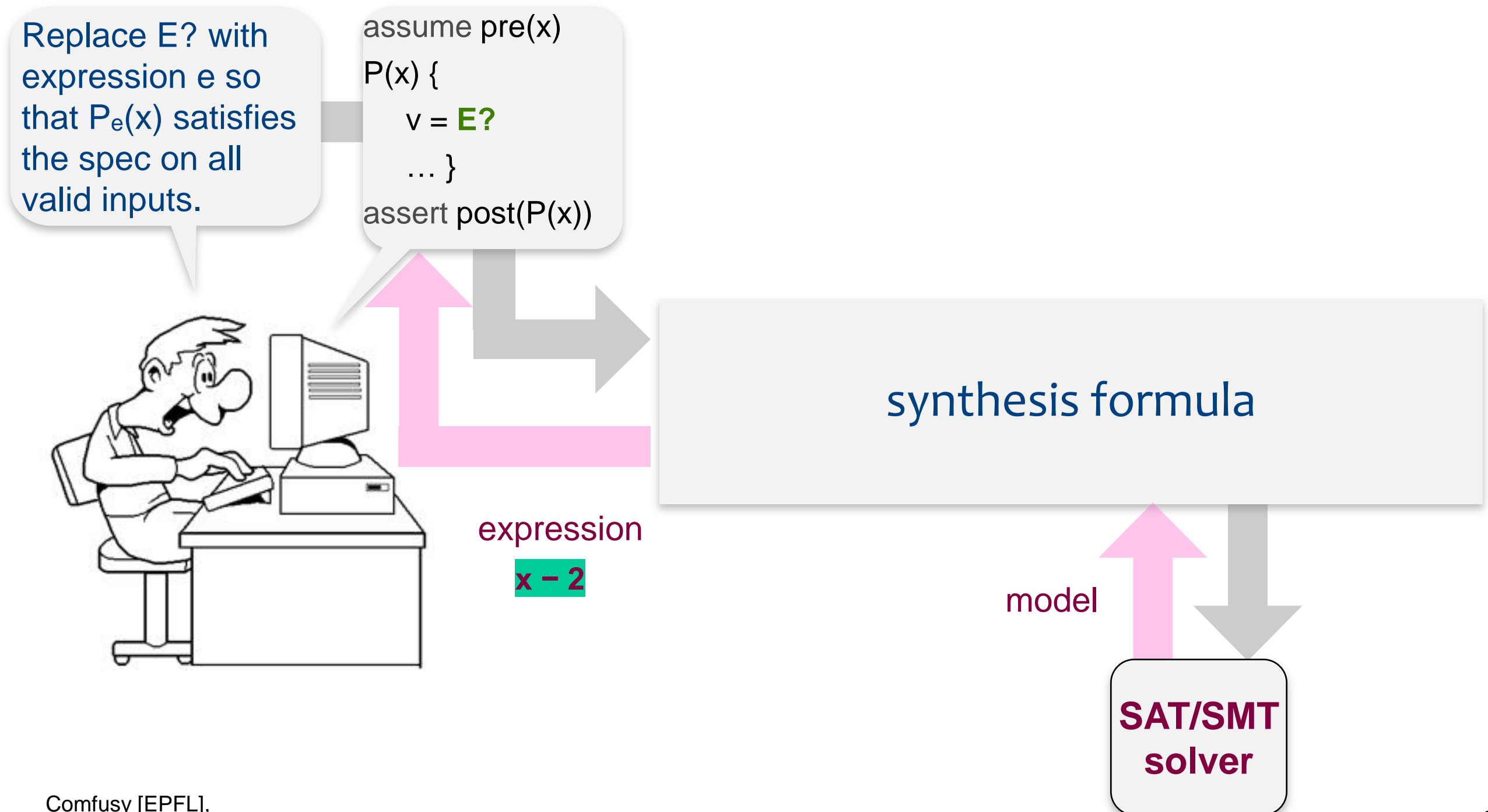
1 #lang s-exp rosette
2
3 (define (classify a b c)
4   (if (and (>= a b) (>= b c))
5     (if (or (= a c) (= b c))
6       (if (and (= a b) (= a c))
7         'EQUILATERAL
8         'ISOSCELES)
9       (if (not (= (* a a) (+ (* b b) (* c c))))
10        (if (< (* a a) (+ (* b b) (* c c)))
11          'ACUTE
12          'OBTUSE)
13        'RIGHT))
14   'ILLEGAL))
15

Welcome to DrRacket, version 5.2.1 [3m].
Language: s-exp rosette [custom]; memory limit: 2048 MB.
> (pretty-debug (assert (equal? (classify 26 26 7) 'ISOSCELES)))
  ▶ #<syntax:3:24 (equal? (classify 26 26 7) (q...>
  ▶ #<syntax:/Users/emina/Workspaces/rosette/demo/cs294/triangle.rkt:5:23 =>
  ▶ #<syntax:/Users/emina/Workspaces/rosette/demo/cs294/triangle.rkt:5:15 =>
>

Determine language from source custom 5:15-5:16
```

Synthesis

programming with a solver: synthesis



Let's correct the classifier bug with synthesis

We ask the synthesizer to replace the buggy expression, `(or (= a c))(= b c)`, with a suitable expression from this grammar

```
hole --> e and e | e or e
e     --> var op var
var   --> a | b | c
op    --> = | <= | < | > | >=
```

We want to write a partial program (sketch) that syntactically looks roughly as follows:

```
(define (classify a b c)
  (if (and (>= a b) (>= b c))
      (if hole ; this used to be (or (= a c))(= b c)
          (if (and (= a b) (= a c))
```

The sketch in Z3: Part 1, derivations for the hole grammar

First we define the “elementary” holes.

These are the values computed by the solver.

These elementary holes determine which expression we will derive from the grammar (see next slides):

(declare-const h0 Int)

(declare-const h1 Int)

(declare-const h2 Int)

(declare-const h3 Int)

(declare-const h4 Int)

(declare-const h5 Int)

(declare-const h6 Int)

part 2: encoding the hole grammar

The call to function hole expands into an expression determined by the values of h_0, \dots, h_6 , which are under solver's control.

```
(define-fun hole((a Int)(b Int)(c Int)) Bool
  (synth-connective h0
    (synth-comparator h1
      (synth-var h2 a b c)
      (synth-var h3 a b c))
    (synth-comparator h4
      (synth-var h5 a b c)
      (synth-var h6 a b c))))
```

Part 3: the rest of the hole grammar

```
(define-fun synth-var ((h Int)(a Int)
                       (b Int)(c Int)) Int
  (if (= h 0)
      a
      (if (= h 1) b c)))
```

```
(define-fun synth-connective ((h Int)(v1 Bool)
                              (v2 Bool)) Bool
  (if (= h 0)
      (and v1 v2)
      (or v1 v2)))
```

You can find synth-comparator in `classifier-synthesis.smt2`.

Part 4: replace the buggy assertion with the hole

The hole expands to an expression from the grammar that will make the program correct (if one exists).

The expression is over variables a,b,c, hence the arguments to the call to hole.

```
(define-fun classify ((a Int)(b Int)(c Int))
                    TriangleType
  (if (and (>= a b) (>= b c))
      (if hole a b c
          (if (and (= a b) (= a c))
              ...
              ...
          ...
      ...
  ...)
```

The synthesis formula

The partial program is now translated to a formula.

Q: how many parameters does the formula have?

A: h_0, \dots, h_6, a, b, c , (and, technically, also the return value)

We are now ready to formulate the synthesis formula to be solved. It suffices to add i/o pair constraints:

```
(assert (= (classify 2 12 27) ILLEGAL))  
(assert (= (classify 5 4 3) RIGHT))  
(assert (= (classify 26 14 14) ISOSCELES))  
(assert (= (classify 19 19 19) EQUILATERAL))  
(assert (= (classify 9 6 4) OBTUSE))  
... ; we have 8 input/output pairs in total
```

The result of synthesis

These i/o pairs sufficed to obtain a program correct on all inputs. The program

h0 -> 1

h1 -> 0

h2 -> 0

h3 -> 1

h4 -> 0

h5 -> 1

h6 -> 2

which means the hole is

(or (= a b) (= b c))

programming with a solver: synthesis

Q: Why doesn't the inductive synthesis variant say
 $\exists e . \bigwedge_i pre(x) \Rightarrow S_p(x, y, e) \wedge post(x, y)$?

A: Because pre- and postconditions on pairs x_i, y_i have been checked when these pairs were selected.

Replace E? with expression e so that $P_e(x)$ satisfies the spec on all valid inputs.

```
assume pre(x)
P(x) {
  v = E?
  ... }
assert post(P(x))
```

We want to solve:

$$\exists e . \forall x . pre(x) \Rightarrow S_p(x, y, e) \wedge post(x, y)$$

We instead solve the I.S. variant:

$$\exists e . \bigwedge_i S_p(x_i, y_i, e)$$

expression
x - 2

model

**SAT/SMT
solver**

Why is this an incorrect synthesis formula?

; hole grammar defined as previously, including the 7 hole vars

(declare-const h0 Int) ... (declare-const h6 Int)

; the partial program is the same

(define-fun classify ((a Int)(b Int)(c Int)) TriangleType

(if (and (>= a b) (>= b c))

(if (hole a b c)

(if (and (= a b) (= a c))

; now we change things, reusing the formula from the verification problem

(declare-const x Int)

(declare-const y Int)

(declare-const z Int)

(assert (and (pre x y z)

(not (post x y z (classify x y z)))))

(check-sat)

(get-model)

Why is this an incorrect synthesis formula?

What problem did we solve?

$$\exists x, y, z, \vec{h} . pre(x) \Rightarrow S_p(x, y, e) \wedge post(x, y)$$

The solver finds a hole value and just one input on which this hole yields a correct program.

we want holes that are correct on all inputs

Advanced topic: enumerate all solutions

We can ask the solver for alternative programs, by insisting that the solution differs from the first one:

```
; ask for a solution different from “(or (= a b)(= b c))”  
(assert (not (and (= h0 1)(= h1 0)(= h2 0)(= h3 1)  
                  (= h4 0)(= h5 1)(= h6 2))))
```

The second synthesized program may be a simple algebraic variation (eg, `(or (= b a)(= b c))`), so we suppress such variations with lexicographic ordering:

```
; example: a=b is legal but b=a is not  
(assert (and (< h2 h3)(< h5 h6)))
```

```
; (or (= a b)(= b c)) is legal but (or (= b c)(= a b)) is not  
(assert (<= h2 h5))
```

Four alternative solutions

(Manual) enumeration leads to four solutions for the hole:

1. `(or (= a b) (= b c))`
2. `(or (= a b) (<= b c))`
3. `(or (<= a b) (<= b c))`
4. `(or (<= a b) (= b c))`

Some of these solutions may be surprising. Are they all correct on all inputs or only on the small set of eight input/output pairs?

To find out, verify these solutions.

Our verifier says they are all correct.

Angelic Programming

programming with a solver: angelic execution

Given x , choose v at runtime so that $P(x, v)$ satisfies the spec.

```
assume pre(x)
P(x) {
  v = choose()
  ...
}
assert post(P(x))
```



the choose statements may be executed several times during the execution (due to a loop), hence the model is mapped to a trace of choose values.

$$\exists \vec{v} . pre(x) \wedge post(y) \wedge S_P(x, y, \vec{v})$$

trace

$$\vec{v} = 0, 2, \dots$$

model

**SAT/SMT
solver**

Example 1: Angelic Programming

The n-queens problem with angelic programming

```
for i in 1..n
    ; place queen  $i$  in a suitable position in the  $i$ th column.
    ; the position is selected by the oracle from the domain  $\{1, \dots, n\}$ 
    position[i] = choose(1..n)
end for
; now check for absence of conflicts (this is our correctness condition)
for i in 1..n
    for j in 1..i-1
        assert queens  $i, j$  do not conflict
    end for
end for
```

Synthesis vs. constraint solving

Constraint solving: solves for a value

- this value satisfies given constraints
- this is a FO value (Bool, Int, Vector of Int, ...)
 - FO=first order, SO=second order

Synthesis: solves for a program

- this program must meet the specification
- program is a SO value – a function from value to value
- in our synthesis approach, we reduce SO to FO with holes

Angelic programming is runtime constraint solving

- choose'n values must meet the constraint that the program terminates without failing any assertion
 - termination may be optional, depending on the setting

Why the name “angelic programming”?

The **choose** expression is *angelic* nondeterminism

- the oracle chooses the value to meet the spec if possible

Compare with *demonic* nondeterminism

- used to model an adversary in program verification
 - eg, an interleaving of instructions
- here, we want from the oracle a counterexample interleaving, one that breaks the spec

Applications of angelic programming

Choose is used *during program development*

- choose expressions return values to be eventually computed by code (that we haven't yet implemented)
- example: choose is used to construct a binary search tree *data structure* for given data and a repOK procedure that checks if data structure is a bst (see code on next slide)

Choose expressions remain in final code

- in n-queens, the choose expr remains in finished code
- we have no intention to replace it with classical operational code
- that code would anyway just perform search; the code might do it better than our solver, but often the solver suffices even in real applications

Angelic BST insertion procedure

```
Node insertT(Node root, int data) {  
    if (root == null) return new Node(data);  
    Node nn = new Node(data);  
    // ask oracle for Node n, where nn will be inserted  
    Node n = choose(set of Nodes created so far)  
    // oracle tells us whether to insert as left/right child  
    if (choose(Bool)) n.left = nn  
    else n.right = nn  
    return root  
}
```

Other ideas for using angelic execution

Imagine you are writing an interpreter for a dataflow language (the DSL in your project), eg

- actor language
- attribute grammar evaluator

This interpreter must choose an order in which to fire executions of nodes, assignments, etc

Your interpreter can compute this order or it can ask choose to compute it given a spec of what partial order must be met by a correct execution

Summary

Constraint solving solves several problems:

- invert the program execution (yields input, given output)
it's not the same as inverting the program (yields a program)
- verify the program
by asking if a violating input exists
- localize the fault
by asking which assertions need to be relaxed to meet the spec
- synthesize a program fragment
we can synthesize expressions, statements, not just constants
- angelic execution
ask an oracle for a suitable value at runtime

Summary (cont)

A program P is translated into the same formula S_P but the formulas for the various problems are different.

Sometimes it is suitable to translate the program differently for each problem, for performance reasons.

Next lecture overview

Solvers accepts different logics and encoding in these logics has vastly different performance.

Here, a comparison of encoding of SIMD matrix transpose in various solvers and logics.

encoding	solver	time (sec)
QF_AUFLIA	cvc3	>600
	z3	159
QF_AUFBV	boolector	409
	z3	287
	cvc3	119
QF_AUFBV-ne	cvc3	>600
	boolector	>600
	z3	25
	stp	11
REL_BV	rosette	9
REL	kodkod	5

Next lecture (cont)

Intro to the solver logics

Encoding arrays and loops

Using Racket as a formula code generator

Ideas for the semester project

References

SMT2 language guide: <http://rise4fun.com/z3/tutorial/guide>

Practical Z3 questions: <http://stackoverflow.com/questions/tagged/z3>