Synthesizing Programs with Constraint Solvers

CAV 2012 invited tutorial

Ras Bodik
Emina Torlak

Division of Computer Science
University of California, Berkeley

Abstract

Classical synthesis derives programs from a specification. We show an alternative approach where programs are obtained through search in a space of candidate programs. Searching for a program that meets a specification frees us from having to develop a sufficiently complete set of derivation rules, a task that is more challenging than merely describing the syntactic shape of the desired program. To make the search for a program efficient, we exploit symbolic constraint solving, lifted to synthesis from the setting of program verification.

We start by describing the interface to the synthesizer, which the programmer uses to specify the space of candidate programs P as well as the desired correctness condition ϕ. The space P is defined by a program template whose missing expressions are described with a grammar. The correctness condition is a multi-modal specification, given as a combination of assertions, input / output pairs, and traces.

Next, we describe several algorithms for solving the synthesis problem $\exists\, P\, \forall\, x\, .\, \phi(x, P(x))$. The key idea is to reduce the problem from 2QBF to SAT by sampling the space of inputs, which eliminates the universal quantification over x.

Finally, we show how to encode the resulting SAT problem in relational logic, and how this encoding can be used to solve a range of related problems that arise in synthesis, from verification to program state repair. We will conclude with open problems on constraint-based synthesis.

What is program synthesis

Find a program P that meets a spec $\phi(\text{input,output})$:

$$\exists P\ \forall x\ .\ \phi(x, P(x))$$

When to use synthesis:

productivity: when writing ϕ is faster than writing P

correctness: when proving ϕ is easier than proving P

What is this tutorial about

Not to learn how to use a synthesizer but to build one

- cf. PLDI'12 tutorial on Sketch programming (Solar-Lezama)
- http://bit.ly/sketch2012

Recipe for building build a lightweight synthesizer

- a semester of work, by building a partial evaluator for a DSL

Concepts and algorithms of solving $\exists P\ \forall x\ .\ \phi(x, P(x))$

- how to solve this second-order formula
- how to write the specification ϕ

Discuss how to make program synthesis more general

- many artifacts are "programs": protocols, invariants, data structures, formulas, compilers, biological cell models, etc

Why this tutorial might be interesting to you

Solvers / decision procedures:

Synthesis is a new application for your solver; internally, one can synthesize inductive invariants, encodings, etc.

Controller synthesis:

View controllers as programs, synthesize them with solvers.

Software model checking:

Solver-based synthesis is to classical synthesis what model checking is to theorem proving. We lift your techniques.

A peek at classical synthesis

Controller synthesis, automata-based (CAV world)

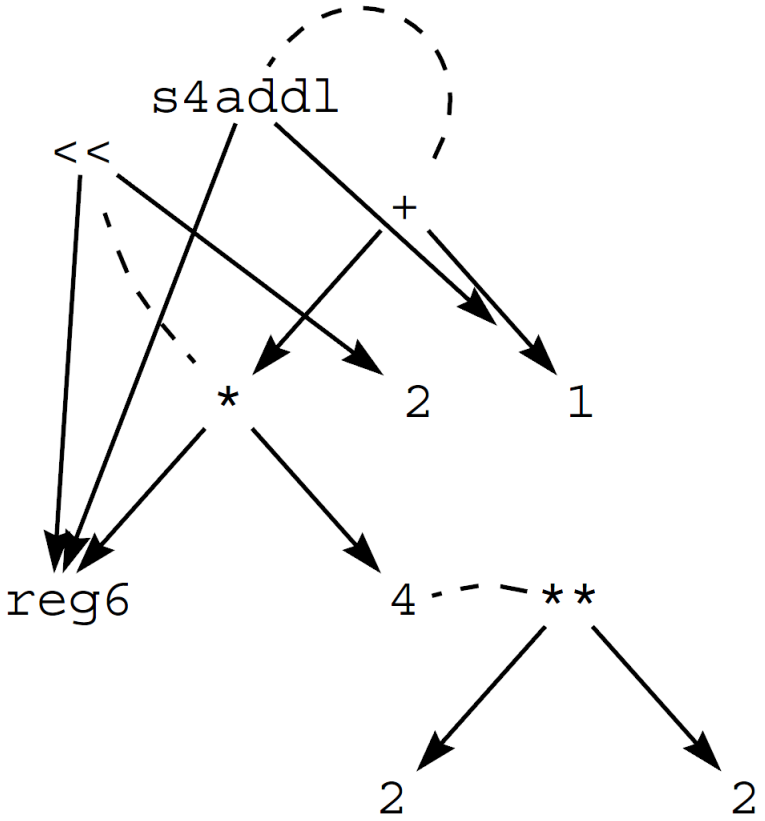
cf. Madhusudan Parthasarathy's talk at SYNT 2012

Program synthesis, deductive (PLDI world)

we will examine key ideas next

Denali: synthesis with axioms and E-graphs

[Joshi, Nelson, Randall PLDI'02]



$\forall\, n\,.\; 2^n = 2{**}n$

$\forall\, k, n\,.\; k * 2^n = k{<<}n$

$\forall k, n::\; k * 4 + n \;=\; \texttt{s4addl}(k, n)$

reg6 $* 4 + 1$ → `s4addl(`**reg6**`,1)`

specification → synthesized program

Two kinds of axioms

Instruction semantics: defines (an interpreter for) the language

$$\forall\, n\,.\; 2^n = 2{**}n$$

$$\forall\, k, n\,.\; k * 2^n = k{<<}n$$

Algebraic properties: associativity of add64, memory modeling, …

$$\forall k, n::\; k * 4 + n \;= \texttt{s4addl}(k, n)$$

$$(\forall\, x, y :: \texttt{add64}(x, y) = \texttt{add64}(y, x))$$

$$(\forall\, x, y, z :: \texttt{add64}(x, \texttt{add64}(y, z)) = \texttt{add64}(\texttt{add64}(x, y), z))$$

$$(\forall\, x :: \texttt{add64}(x, 0) = x)$$

$$(\forall\, a, i, j, x \;:: i = j \vee \texttt{select}(\texttt{store}(a, i, x), j) = \texttt{select}(a, j))$$

Properties of deductive synthesizers

Efficient and provably correct

- thanks to semantics-preserving rules
- only correct programs are explored
- Denali is scalable: prior super-optimizers gen-and-test

Successfully built for axiomatizable domains

- expression equivalence (Denali)
- linear filters (FFTW, Spiral)
- linear algebra (FLAME)
- statistical calculations (AutoBayes)
- data structures as relational DBs (P2; Hawkins et al.)

Downsides of deductive optimizers

Completeness hinges on sufficient axioms

- some domains hard to axiomatize (e.g., sparse matrices)

Specification must be complete, to seed derivation

- we want partial and multi-modal specs (more on this later)

Control over the "shape" of the synthesized program

- we often want predictable, human-readable programs

Solver-based Inductive synthesis achieves these

Some of the challenges ahead of us

Completeness: want to synthesize any program

while teaching the synthesizer minimal rules. Which ones?

Ideally, just the (operational) semantics of the language!

Efficiency: $\exists P\ \forall x\ .\ \phi\big(x, P(x)\big)$ is second-order formula

but first-order solvers (SAT, SMT) generally more efficient

How to encode this SO problem for these solvers?

Outline

Part I: in a nutshell

- **partial programs**
- **inductive synthesis**
- **a minimalistic synthesizer**

Part II: in depth

- a more expressive synthesis language: specs and holes
- a DIY program-to-formula translator
- turning solvers into “synthesis engines”

Preparing your language for synthesis

Extend the language with two constructs

spec:

```
int foo (int x) {
    return x + x;
}
```

$\phi(x, y)$: $y =$ **foo**(x)

sketch:

```
int bar (int x) implements foo {
    return x << ??;
}
```

?? substituted with an int constant meeting ϕ

result:

```
int bar (int x) implements foo {
    return x << 1;
}
```

instead of **implements**, assertions over safety properties can be used

Synthesis as search over candidate programs

Partial program (sketch) defines a candidate space

we search this space for a program that meets ϕ

Usually can't search this space by enumeration

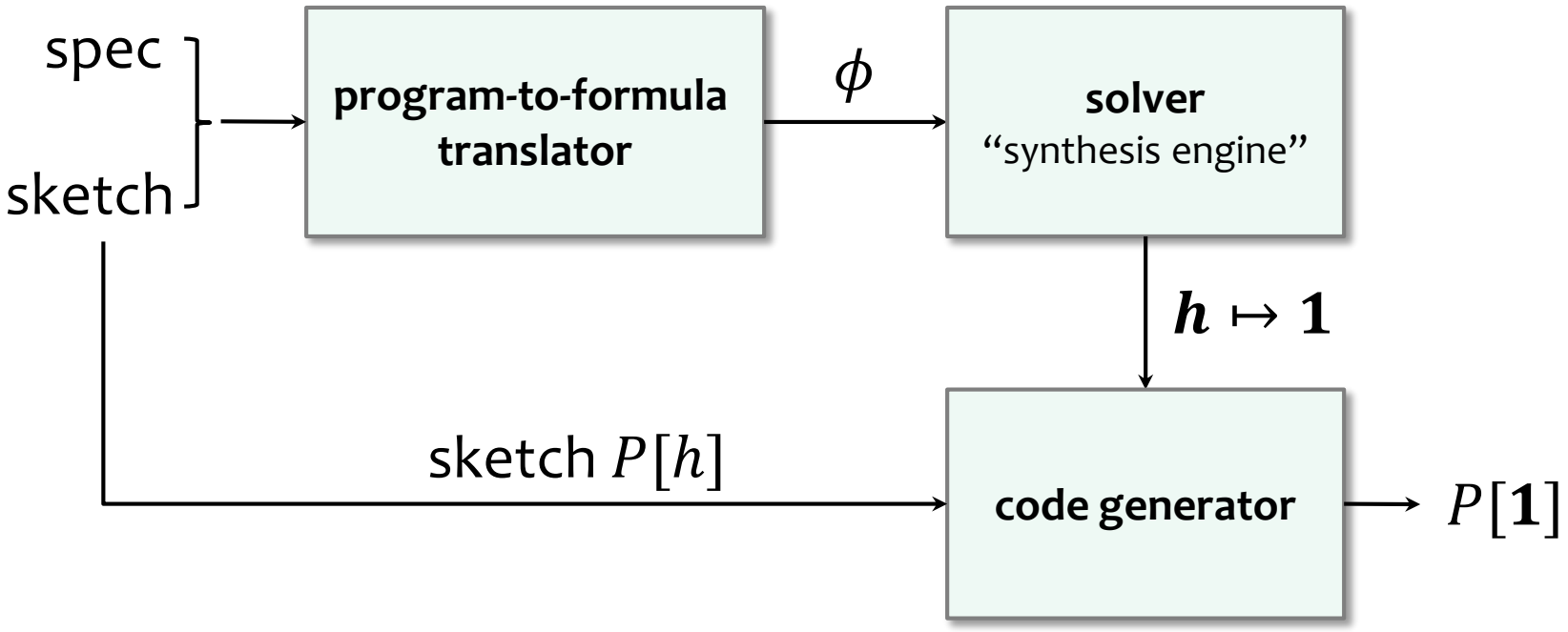
space too large ($\gg 10^{10}$)

Describe the space **symbolically**

solution to constraints encoded in a logical formula gives values of holes, indirectly identifying a correct program

What constraints? We'll cover this shortly.

Synthesis from partial programs



Example: Parallel Matrix Transpose

Example: 4x4-matrix transpose with SIMD

a functional (executable) specification:

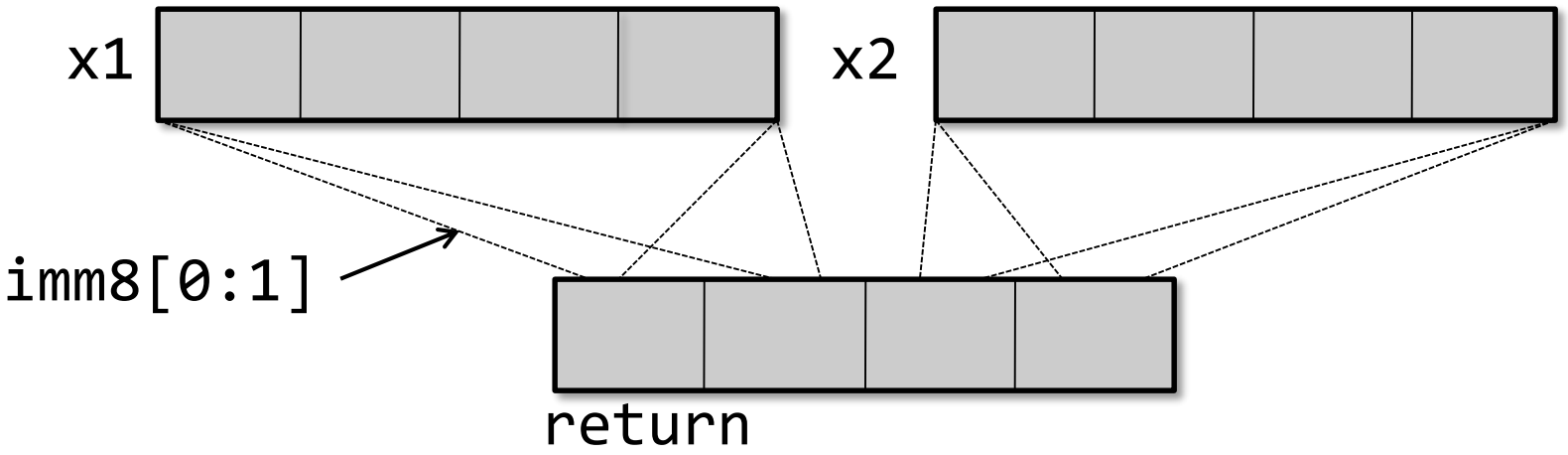
```
int[16] transpose(int[16] M) {
  int[16] T = 0;
  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 4; j++)
      T[4 * i + j] = M[4 * j + i];
  return T;
}
```

This example comes from a Sketch grad-student contest

Implementation idea: parallelize with SIMD

Intel SHUFP (shuffle parallel scalars) SIMD instruction:

```
return = shufps(x1, x2, imm8 :: bitvector8)
```



High-level insight of the algorithm designer

Matrix M transposed in two shuffle phases

Phase 1: shuffle M into an intermediate matrix S with some number of `shufps` instructions

Phase 2: shuffle S into an result matrix T with some number of `shufps` instructions

Synthesis with partial programs helps one to complete their insight. Or prove it wrong.

The SIMD matrix transpose, sketched

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;

  S[??::4] = shufps(M[??::4], M[??::4], ??);
  S[??::4] = shufps(M[??::4], M[??::4], ??);
  …
  S[??::4] = shufps(M[??::4], M[??::4], ??);

  T[??::4] = shufps(S[??::4], S[??::4], ??);
  T[??::4] = shufps(S[??::4], S[??::4], ??);
  …
  T[??::4] = shufps(S[??::4], S[??::4], ??);

  return T;
}
```

Phase 1

Phase 2

The SIMD matrix transpose, sketched

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;
  repeat (??) S[??::4] = shufps(M[??::4], M[??::4], ??);
  repeat (??) T[??::4] = shufps(S[??::4], S[??::4], ??);
  return T;
}
int[16] trans_sse(int[16] M) implements trans { // synthesized code
  S[4::4]   = shufps(M[6::4],   M[2::4],  11001000b);
  S[0::4]   = shufps(M[11::4],  M[6::4],  10010110b);
  S[12::4]  = shufps(M[0::4],   M[2::4],  10001101b);
  S[8::4]   = shufps(M[8::4],   M[12::4], 11010111b);
  T[4::4]   = shufps(S[11::4],  S[1::4],  10111100b);
  T[12::4]  = shufps(S[3
  T[8::4]   = shufps(S[4
  T[0::4]   = shufps(S[1
}
```

From the contestant email:

Over the summer, I spent about 1/2 a day manually figuring it out.

Synthesis time: <5 minutes.

Demo: transpose on Sketch

Try Sketch online at http://bit.ly/sketch-language

Demo notes (1)

In the demo, we accelerated synthesis by changing

```
repeat(??) loop body
repeat(??) loop body
```

to

```
int steps = ??
repeat(steps) loop body
repeat(steps) loop body
```

→ can improve efficiency by adding more "insight"

here, the "insight" constraints state that both loops have same (unknown) number of iterations

Demo notes (2)

How did the student come up with the insight that two phases are sufficient?

- We don't know but the synthesizer can prove that one phase is insufficient (a one-phase sketch has no solution)

Inductive Synthesis, Phrased as Constraint Solving

What to do with a program as a formula?

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

program: `f(x) { return x + x }`

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

This formula is created as in program verification with concrete semantics [CMBC, Java Pathfinder, …]

With program as a formula, solver is versatile

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

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

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

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

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 << ?? }
```

$S_{sketch}(x, y, h)$: $y = x * 2^h$

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

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

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 $\boldsymbol{h \mapsto 1}$

Inductive synthesis

Our constraints encode **inductive synthesis:**

- We ask for a program P correct on a few inputs.
- We hope (or test, verify) that P is correct on rest of inputs.

Part II will describe how to select suitable inputs

We do inductive synthesis with **concrete inputs**

- important note: concrete inputs eliminate ∀x from $\exists h\ \forall x\ .\ \phi\big(x, P[h](x)\big)$, turning 2QBF into a SAT problem.

Translate program to a formula

It remains to show how to create a formula

Steps:

- make synthesis constructs (??) explicit in host language
- unroll program into a bounded acyclic program
- make it functional (get rid of side effects)
- "read off" the formula from this functional program

In Part I, we'll translate transpose to formula manually.

In Part II, we'll build a program-to-formula compiler.

Example of how a program is translated

```
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;
  S[0::4]  = shufps(M[??::4], M[??::4], ??);
  S[4::4]  = shufps(M[??::4], M[??::4], ??);
  S[8::4]  = shufps(M[??::4], M[??::4], ??);
  S[12::4] = shufps(M[??::4], M[??::4], ??);
  T[0::4]  = shufps(S[??::4], S[??::4], ??);
  T[4::4]  = shufps(S[??::4], S[??::4], ??);
  T[8::4]  = shufps(S[??::4], S[??::4], ??);
  T[12::4] = shufps(S[??::4], S[??::4], ??);
  return T;
}
```

Example of how a program is translated

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S[0::4]  = shufps(M[??::4], M[??::4], ??);
  S[4::4]  = shufps(M[??::4], M[??::4], ??);
  S[8::4]  = shufps(M[??::4], M[??::4], ??);
  S[12::4] = shufps(M[??::4], M[??::4], ??);
  T[0::4]  = shufps(S[??::4], S[??::4], ??);
  T[4::4]  = shufps(S[??::4], S[??::4], ??);
  T[8::4]  = shufps(S[??::4], S[??::4], ??);
  T[12::4] = shufps(S[??::4], S[??::4], ??);
  assert equals(T, trans(M));
  return T;
}
```

Make the correctness condition explicit: trans_sse implements trans

Example of how a program is translated

Name the holes: each corresponds to a fresh symbolic variable.

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S[0::4]  = shufps(M[mx1_0::4], M[mx2_0::4], mi_0);
  S[4::4]  = shufps(M[mx1_1::4], M[mx2_1::4], mi_1);
  S[8::4]  = shufps(M[mx1_2::4], M[mx2_2::4], mi_2);
  S[12::4] = shufps(M[mx1_3::4], M[mx2_3::4], mi_3);
  T[0::4]  = shufps(S[sx1_0::4], S[sx2_0::4], si_0);
  T[4::4]  = shufps(S[sx1_1::4], S[sx2_1::4], si_1);
  T[8::4]  = shufps(S[sx1_2::4], S[sx2_2::4], si_2);
  T[12::4] = shufps(S[sx1_3::4], S[sx2_3::4], si_3);
  assert equals(T, trans(M));
  return T;
}
```

Example of how a program is translated

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S[0::4]  = shufps(rd4(M, mx1_0), rd4(M, mx2_0), mi_0);
  S[4::4]  = shufps(rd4(M, mx1_1), rd4(M, mx2_1), mi_1);
  S[8::4]  = shufps(rd4(M, mx1_2), rd4(M, mx2_2), mi_2);
  S[12::4] = shufps(rd4(M, mx1_3), rd4(M, mx2_3), mi_3);
  T[0::4]  = shufps(rd4(S, sx1_0), rd4(S, sx2_0), si_0);
  T[0::4]  = shufps(rd4(S, sx1_1), rd4(S, sx2_1), si_1);
  T[0::4]  = shufps(rd4(S, sx1_2), rd4(S, sx2_2), si_2);
  T[0::4]  = shufps(rd4(S, sx1_3), rd4(S, sx2_3), si_3);
  assert equals(T, trans(M));
  return T;
}
```

Turn bulk array accesses into explicit calls to a read function.

rd4(A, i) returns a new 4-element array consisting of A[i], ..., A[i+3].

Example of how a program is translated

```
int[16] trans_sse(int[16] M) {
  int[16] S = 0, T = 0;
  S0 = wr4(S , shufps(rd4(M, mx1_0), rd4(M, mx2_0), mi_0), 0);
  S1 = wr4(S0, shufps(rd4(M, mx1_1), rd4(M, mx2_1), mi_1), 4);
  S2 = wr4(S1, shufps(rd4(M, mx1_2), rd4(M, mx2_2), mi_2), 8);
  S3 = wr4(S2, shufps(rd4(M, mx1_3), rd4(M, mx2_3), mi_3), 12);
  T0 = wr4(T , shufps(rd4(S3, sx1_0), rd4(S3, sx2_0), si_0), 0);
  T1 = wr4(T0, shufps(rd4(S3, sx1_1), rd4(S3, sx2_1), si_1), 4);
  T2 = wr4(T1, shufps(rd4(S3, sx1_2), rd4(S3, sx2_2), si_2), 8);
  T3 = wr4(T2, shufps(rd4(S3, sx1_3), rd4(S3, sx2_3), si_3), 12);
  assert equals(T3, trans(M));
  return T3;
}
```

Convert to SSA by replacing bulk array writes with functional writes.

wr4(A, Delta, i) returns a copy of A, but with Delta[0::4] at positions i, ..., i+3.

Read out the formula

Once the program is functional, turn it into a formula.

- Many encodings of programs as formulas are possible.
- Solver solve some encodings faster than others.
- Times from our experiments with encoding transpose:

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

Why Kodkod?

a SAT-based solver optimized for reasoning over finite domains (as used in inductive synthesis)

high level input logic

- FOL with relational algebra, transitive closure, bitvector arithmetic and partial models

model finder and minimal UNSAT core extractor for this logic

- enables both synthesis and diagnosis of synthesis failures

Some applications of Kodkod

lightweight formal methods

Alloy4 (Alloy), Nitpick (Isabelle/HOL), ProB (B, Even-B, L (UML)

checking code & memory models

Forge, Karun, Miniatur, TACO, MemSAT

declarative programming

Squander, PBnJ, Tarmeem, Cobbler

declarative configuration

ConfigAssure (networks), Margrave (policies)

test-case generation

Kesit, Whispec

... and many more

alloy.mit.edu/kodkod

MemSAT

Example of how a statement is translated

```
int[16] S = 0
mx1_0
rd4(A, i)
wr4(A, Delta, i)
```

Everything is a relation: a set of tuples of equal length, drawn from a finite universe.

- universe: { 0, 1, 2, …, 15 }
- tuples: [0], [1, 2], [5, 3, 4]
- three relations:

  { [0] }, // scalar value 0

  { [0], [3], [4] }, // a set

  { [1, 2], [2, 3], [3, 4] }

Example of how a statement is translated

```
int[16] S = 0
mx1_0   // was ??
rd4(A, i)
wr4(A, D, i)
```

let S := { [0, 0], [1, 0] , ..., [15, 0] }

Constant binary relation from 0 to 15, inclusive, to 0.

Example of how a statement is translated

```
int[16] S = 0
mx1_0   // was ??
rd4(A, i)
wr4(A, D, i)
```

let S := { [0, 0], [1, 0] , ..., [15, 0] }

mx1_0 ⊆ { [0], [1] , ..., [12] } ∧ **one** mx1_0

Unary relational variable that may take any *one* of the values in { [0], ..., [12] }; i.e., a scalar.

Example of how a statement is translated

```
int[16] S = 0
mx1_0
rd4(A, i)
wr4(A, D, i)
```

let S := { [0, 0], [1, 0] , ..., [15, 0] }

mx1_0 ⊆ { [0], [1] , ..., [12] } ∧ **one** mx1_0

{[0]} × i.A ∪ ... ∪ {[3]} × add(i, 3).A

A = {[0,a],[1,b],[2,c],[3,d],[4,e]}
i = {[1]}
rd4(A,i,) = {[0,b],[1,c],[2,d],[3,e]}

Relational join ("map get"):

{ [a] } . { [a, c], [b, d] } = { [c] }

Example of how a statement is translated

```
int[16] S = 0
mx1_0
rd4(A, i)
wr4(A, D, i)
```

let S := { [0, 0], [1, 0] , ..., [15, 0] }

mx1_0 ⊆ { [0], [1] , ..., [12] } ∧ **one** mx1_0

{[0]} × i.A ∪ ... ∪ {[3]} × add(i, 3).A

A = {[0,a],[1,b],[2,c],[3,d],[4,e]}
i = {[1]}
rd4(A,i,) = {[0,b],[1,c],[2,d],[3,e]}

Relational product ("concat"):

{ [a], [b] } × { [c] } = { [a, c], [b, c] }

Example of how a statement is translated

```
int[16] S = 0
mx1_0
rd4(A, i)
wr4(A, D, i)
```

A = {[0,a],[1,b],[2,c],[3,d],[4,e]}
i = {[1]}
D = {[0,f],[1,g],[2,h],[3,j]}
wr4(A,D,i,) = {[0,a],[1,f],[2,g],[3,h],[4,j]}

let S :={ [0, 0], [1, 0] , ..., [15, 0] }
mx1_0 ⊆ { [0], [1] , ..., [12] } ∧ **one** mx1_0
{[0]} × i.A ∪ ... ∪ {[3]} × add(i, 3).A
A ++ (i × {[0]}.D ∪ ... ∪ add(i, 3) × {[3]}.D)

Relational override ("map put"):

{ [a, c], [b, d], [e, f] } ++ { [e, g] } =
{ [a, c], [b, d], [e, g] }

Demo of how a program is translated

KODKOD

Summary of Part I

Partial programs define space of candidates

- most of the candidates are wrong; find a correct one

Solver-based inductive synthesis finds the program

1. lift holes in partial programs to symbolic variables
2. translate the program to a formula,
3. carefully select sample inputs (aka observations)
4. find values of holes s.t. program is correct on these inputs

What semantics did we use: concrete, not abstract

- turned the solver into bidirectional interpreter that maps input and output to holes

Outline

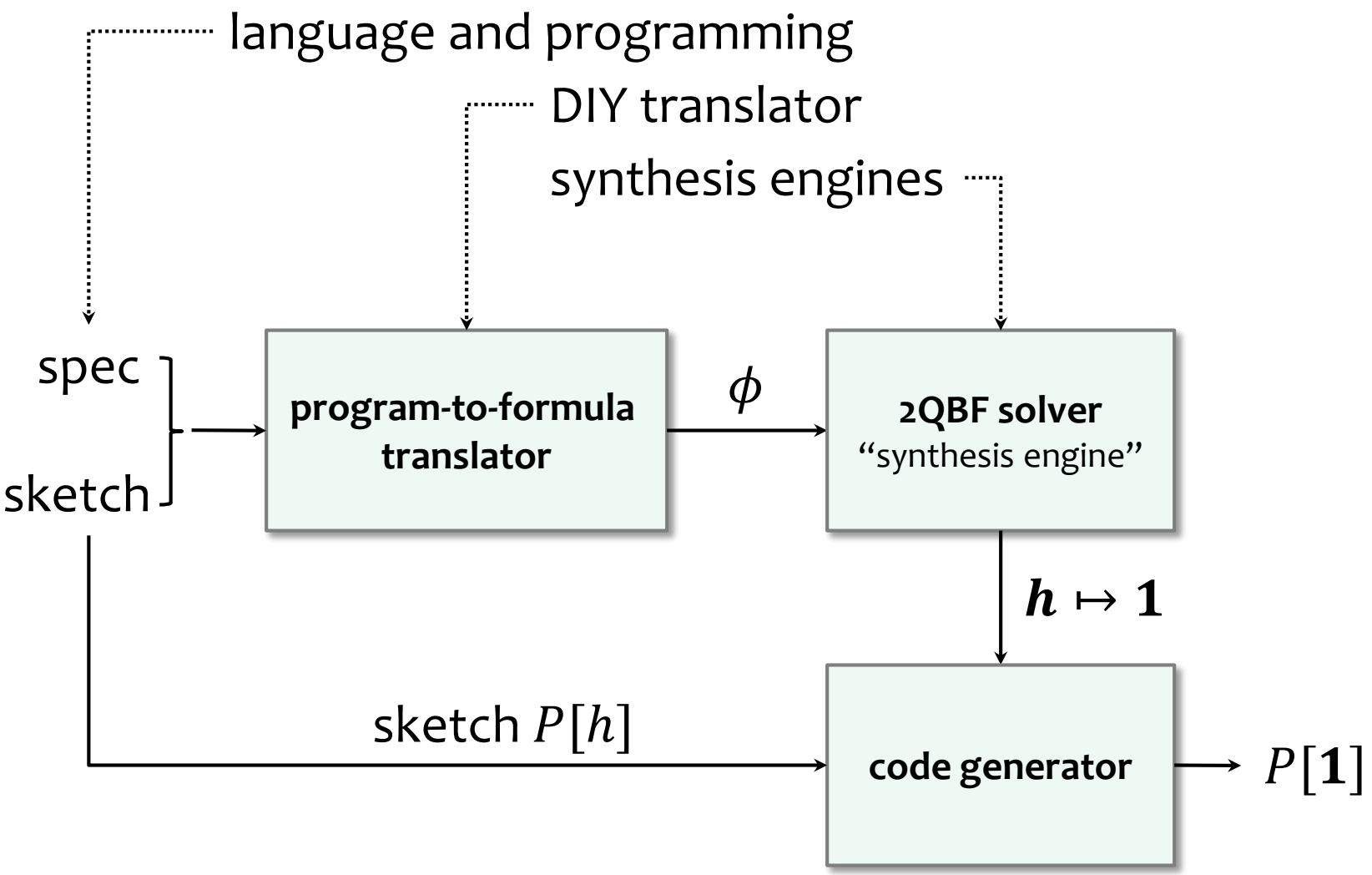
Part I: in a nutshell

- partial programs
- inductive synthesis
- a minimalistic synthesizer

Part II: in depth

- **a more expressive synthesis language: specs and holes**
- **a DIY program-to-formula translator**
- **turning solvers into "synthesis engines"**

Outline of Part II



Specs are partial and multi-modal

Why multi-modal specifications?

During specification writing, you run may into:

I find it very hard to write a full behavioral spec!

I find it very hard to write a declarative spec!

My synthesizer runs forever on this problem!

Multi-modal specifications

executable specification (a ref implementation)

```
spec  :: Input → Output
```

declarative specification (a checker)

```
check :: Input x Output → Boolean
```

safety assertions (anywhere in the program)

- example: x==0 and p must be of type Foo
- implicit language assertions, eg bounds checks

input-output pairs

manually provided or computed from other specs

All these specs can be translated to constraints

Specifications constrain two aspects

Observable behavior: an input/output relation

- **WHAT** the program computes
- behavior spec could be **full** (executable spec)
- or **partial** (io pairs or safety property)

Structural properties: internal computational steps

- **HOW** the computation proceeds
- expressed in a sketch
- further constrained by assertions such as

```
assert number_of_iterations < 3 * log(input_size)
assert number_of_cache_misses < n
```

Lessons

I find it very hard to write a full behavioral spec!

a sketch constrains what programs synthesizable from it can compute, thus compensating partial behavioral spec

I find it very hard to write a declarative spec!

don't need to write FO spec → use ref implementations

My synthesizer runs forever on this problem!

assert that program agrees with a trace: asserting values at intermediate programs points decompose the synthesis

Holes need not be constants

Synthesis of expressions

Unspecified parts of a sketch may be expressions, too:

Array index expressions: `A[ ??*i+??*j+?? ]`

Polynomial of degree 2 over x: `??*x*x + ??*x + ??`

Reusable expression "generators"

Following function synthesizes to one of

`a, b, a+b, a-b, a+b+a, …, a-a+b-b, …`

```
inline int expr(int a, int b){  // generator
   switch(??) {
      case 0: return a;
      case 1: return b;
      case 2: return expr(a,b) + expr(a,b);
      case 3: return expr(a,b) - expr(a,b);
   }
}
```

Synthesizing polynomials

```
int spec (int x) {
  return 2*x*x*x*x + 3*x*x*x + 7*x*x + 10;
}

int p (int x) implements spec {
  return (x+1)*(x+2)*poly(3,x);
}

inline int poly(int n, int x) {
  if (n==0) return ??;
  else return x * poly(n-1, x) + ??;
}
```

Notice the absence of any meta-variables. The generator poly() is an ordinary function.

Here, SKETCH performs polynomial division. Result of division is what poly(3,x) is synthesized into.

Programs with unbounded inputs

Unbounded input size

If desired program is to handle inputs of arbitrary size:

- synthesize it on a sample of small inputs, then
- check that it works on a set of larger inputs

Small-world hypothesis for verification:

- if a program is correct on all inputs of to size *k*,
  then it is also correct on all inputs greater than *k*

Corollary for synthesis:

- synthesizing a program on small inputs produces program correct on large inputs

Of course, there are no formal guarantees

- our correctness is only as good as bounded model checking

DIY Synthesis

DIY synthesis: desiderata

may want a general-purpose language with "holes"

- loops, ADTs, etc., with basic integer/boolean holes
- support to grow the language to list-valued holes, etc

automatic synthesis of values/expressions for holes

- a compiler that translates partial programs to formulas
- inductive synthesizer: correct for given inputs
- DIY and do it fast!

DIY synthesis: problem statement

Given a program $P :: (X, H) \rightarrow O$ with two input kinds

X: regular inputs

H: holes

Solve the inductive synthesis problem

$$\exists h \,.\, \phi(P(x_1, h), o_1) \wedge \ldots \wedge \phi(P(x_n, h), on)$$

By building a partial evaluator/compiler to formulas

- $\text{SynthTranslator} :: (P, X, O) \rightarrow \text{Formula}(H)$
- hardcodes (partially evaluates) the i/o pairs x, o
- produces a formula where holes h are free variables

DIY synthesis in 3 steps

1. Define a small interpreted language: E ::= n | E + E | ...
   - e.g., expressions over booleans, integers and lists; conditionals and loops; function definition and application; mutation

2. Add holes to the language: E ::= n | ??_int | E + E | ...

3. Extend the interpreter into SynthTranslate
   - recall we have concrete values for the input and output
   - interpret each hole as a fresh symbolic value: ?? --> **x**
   - operations on concrete values remain the same: 1 + 1 --> 2
   - operations on symbolic values return ASTs: 1 + ?? --> **1 + x**
   - at the end, compile the resulting AST to a formula

DIY synthesis, step 1: *steal* a language

```
top-level-form = general-top-level-form
               | (#%expression expr)
               | (module id name-id
                   (#%plain-module-begin
                    module-level-form ...))
               | (begin top-level-form ...)
               | (begin-for-syntax top-level-form ...)

module-level-form = general-top-level-form
               | (#%provide raw-provide-spec ...)
               | (begin-for-syntax module-level-form ...)

general-top-level-form = expr
               | (define-values (id ...) expr)
               | (define-syntaxes (id ...) expr)
               | (#%require raw-require-spec ...)

expr = id
     | (#%plain-lambda formals expr ...+)
     | (case-lambda (formals expr ...+) ...)
     | (if expr expr expr)
     | (begin expr ...+)
     | (begin0 expr expr ...)
     | (let-values ([(id ...) expr] ...)
         expr ...+)
     | (letrec-values ([(id ...) expr] ...)
         expr ...+)
     | (set! id expr)
     | (quote datum)
     | (quote-syntax datum)
     | (with-continuation-mark expr expr expr)
     | (#%plain-app expr ...+)
     | (#%top . id)
     | (#%variable-reference id)
     | (#%variable-reference (#%top . id))
     | (#%variable-reference)

formals = (id ...)
        | (id ...+ . id)
        | id
```

Racket

with an interpreter, a small core, and metaprogramming

quickly build SynthTranslate by partially hijacking the host's own interpreter

grow the synthesizer by taking over the interpretation of more host constructs (eg to allow list-valued holes)

you always have a working system to experiment with!

DIY synthesis, step 2: add **named** holes …

```
top-level-form = general-top-level-form
               | (#%expression expr)
               | (module id name-id
                   (#%plain-module-begin
                    module-level-form ...))
               | (begin top-level-form ...)
               | (begin-for-syntax top-level-form ...)

module-level-form = general-top-level-form
               | (#%provide raw-provide-spec ...)
               | (begin-for-syntax module-level-form ...)

general-top-level-form = expr
               | (define-values (id ...) expr)
               | (define-syntaxes (id ...) expr)
               | (#%require raw-require-spec ...)

expr = id
               | (#%plain-lambda formals expr ...+)
               | (case-lambda (formals expr ...+) ...)
               | (if expr expr expr)
               | (begin expr ...+)
               | (begin0 expr expr ...)
               | (let-values ([(id ...) expr] ...)
                  expr ...+)
               | (letrec-values ([(id ...) expr] ...)
                  expr ...+)
               | (set! id expr)
               | (quote datum)
               | (quote-syntax datum)
               | (with-continuation-mark expr expr expr)
               | (#%plain-app expr ...+)
               | (#%top . id)
               | (#%variable-reference id)
               | (#%variable-reference (#%top . id))
               | (#%variable-reference)

formals = (id ...)
               | (id ...+ . id)
               | id
```

```
(define-symbolic id kind)

kind = number?
     | boolean?
```

Example usage of Rosette named holes

```
#lang racket

(define z 0)

(define (sign y)
  (if (< y z)
      -1
      (if (< z y)
          1
          0)))
```

```
#lang rosette

(define-symbolic z number?)

(define (sign y)
  (if (< y z)
      -1
      (if (< z y)
          1
          0)))
```

DIY synthesis, step 2: add holes using macros

```
(define-syntax (define-symbolic stx)
  (syntax-case stx (number? boolean?)
    [(_ id number?)  #`(define id (sym-var #'id number?))]
    [(_ id boolean?) #`(define id (sym-var #'id boolean?))]))
```

```
(define-symbolic z number?)
```

```
(define z (sym-var #'z number?))
```

z is bound to a symbolic integer variable called “z”

DIY synthesis, step 3: hijack the interpreter

```
(provide (rename-out [< sym/<]))

(define (sym/< x y)
  (if (and (number? x) (number? y))
      (< x y)
      (sym-expr sym/< x y)))

(define (sym-expr op . args)
  (sym op args))

(struct sym (op args))
```

operations on symbolic values produce symbolic ASTs/terms that are compiled to formulas

DIY synthesis, step 3: hijack the interpreter

```
(provide (rename-out [if sym/if]))

; very simple conditional handling, assuming that both
; branches are possibly symbolic integer expressions
; with no side-effects (e.g., assertions or mutation)
(define-syntax-rule (sym/if testExpr thenExpr elseExpr)
  (let ([test (! (sym/equal? testExpr #f))])
     (match test
       [#t thenExpr]
       [#f elseExpr]
       [_ (sym/phi test thenExpr elseExpr)])))
```

DIY synthesis: demo

Synthesis “Engine”

Synthesis is a 2QBF problem

So far, we did not synthesize programs that were actually correct for all inputs x. Instead of

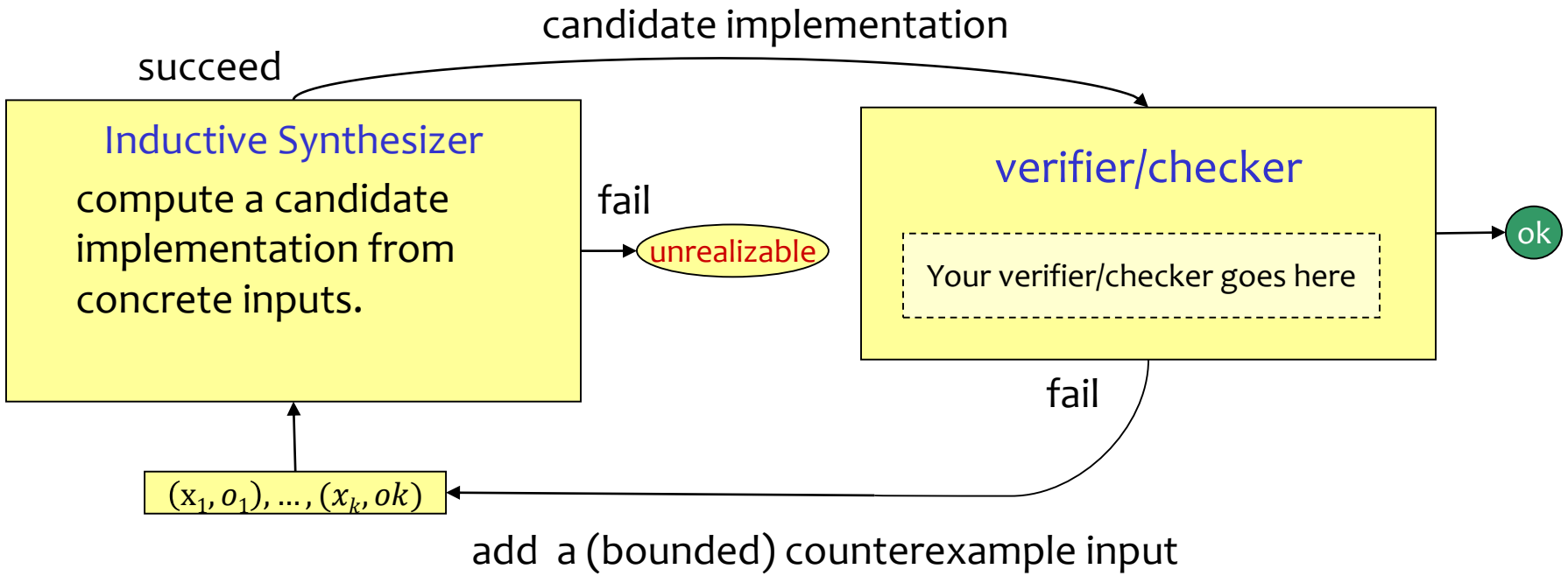
$$\exists h \, \forall x \, . \, \phi(x, P(x, h))$$

we solved the

$$\exists h \, . \, \phi(P(x_1, h), o_1) \; \wedge \; \ldots \wedge \phi(P(x_n, h), on)$$

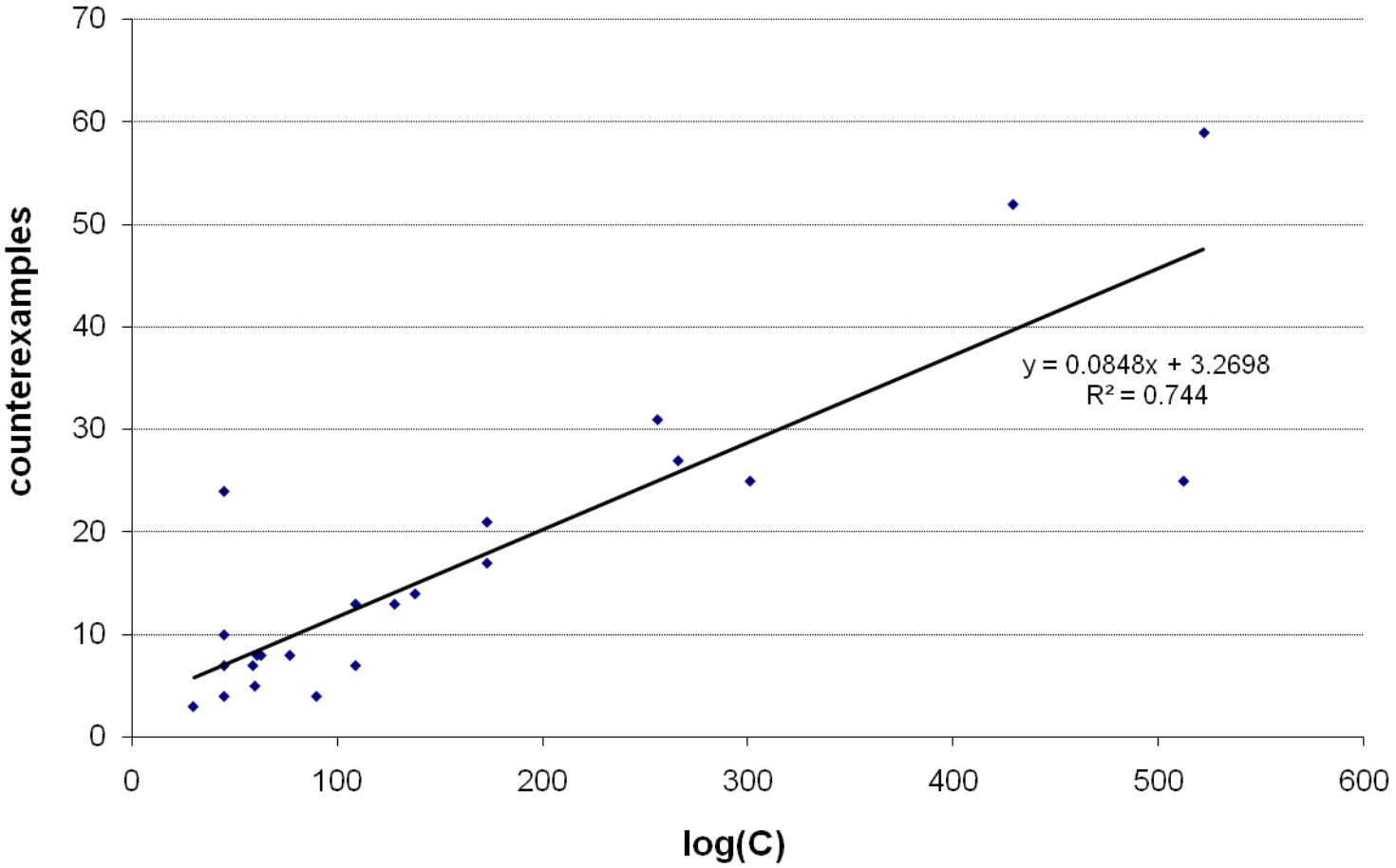
Now we show how to find the suitable inputs $x_1, \ldots, xn$ such that we solve the actual 2QBF problem

1) CounterExample -Guided Inductive Synthesis (CEGIS)



Number of counterexample vs. log(C)

C = size of candidate space = exp(bits of controls)



Space pruning by sketches; input pruning by CEGIS

Grammar space: Explore all programs in the language

- case study: a concurrent list: space of ~10^{30} candidates
- 1sec/validation: synthesis time ≫ age of universe

Sketched space: sketch reduces candidate space

- concurrent list **sketch**: candidate space goes down to 10^9
- synthesis time[1]: ~10-100 days
- sadly, our spaces may be 10^{800}, so this alone not enough

CEGIS: reduce how many inputs you consider

- concurr. list sketch: 1min synth time, 3 CEGIS iterations
- 3 inputs (here, schedules) sufficed

[1]assuming that the space contains 100-1000 correct candidates

2) Disambiguating 2QBF solver

What if verification is impossible?

ex: we cannot translate the reference implementation to a formula (its code too large, complex, or unavailable)

We can't ask a symbolic verifier for a counterexample

but we can execute the specification (on a real machine)

Idea: ask a solver for disagreeing plausible candidates

find two plausible candidates (ie correct wrt observations) that evaluate to a different value on a distinguishing input x? **Problem:** given P_1, find P_2 and x s.t. $P_1(x) \neq P_2(x)$.

Example: synthesize deobfuscation of malware

spec is complex malware [Jha, Gulwani, Seshia, Tiwari, 2010]

3) Generate and Test

Don’t underestimate the simplicity of exhaustively enumerating candidates

Works for small spaces of candidates

Our results: what was synthesized

Block ciphers, [ASPLOS 2006]

- highly optimized matrix code

Stencils [PLDI 2007]

- highly optimized matrix codes

Concurrent Data Structures [PLDI 2008]

- lock free lists and barriers

Deutsch-Shorr-Waite [POPL 2010]

- stack-free depth-first-search

Dynamic Programming Algorithms [OOPSLA 2011]

- O(N) algorithms, including parallel ones

Sample of Results by Others

Instruction sequences [Jha et al, ICSE 2010]

- including de-obfuscation of malware

Program inversion [Srivastava et al., PLDI 2011]

- from compressors to decompressors

Sorting [Srivastava, Gulwani, Foster, POPL 2010]

- lock free lists and barriers

Controller switching logic [Jha, Seshia, Tiwari 2011]

- numerical constraints plus generalization

Geometric constructions [Gulwani et al. PLDI 2011]

- not constraints-based

Recap

Inductive synthesis

Find a program correct on a set of inputs and hope (or verify) that it's correct on rest of inputs.

A **partial program** syntactically defines the candidate space.

Inductive synthesis search phrased as a **constraint problem**.

Program found by (symbolic) interpretation of a (space of) candidates, not by deriving the candidate.

So, to find a program, we need only an interpreter, not a sufficiently set of derivation axioms.

How General is Program Synthesis?

Generality boils down to "what's a program?"

If you can view X as a program, you can synthesize it.

A sample of "programs":

- loop **invariants**
- **controllers**
- logical **formulas** (incl. encodings of other problems)
- network and other **protocols**
- **data structures** (layouts and maintenance methods)
- **incremental** and other algorithms
- biological **models**
- end-users **scripts**

Synthesis with partial programs

Partial programs can communicate programmer insight

> **Once you understand how to write a program,**
> **get someone else to write it.** *Alan Perlis, Epigram #27*

Suitable synthesis algorithm completes the mechanics.

When you see programming or a formalization problem, think whether it is decomposable into a partial program and its completion.

Acknowledgements

UC Berkeley

Gilad Arnold
Shaon Barman
Prof. Ras Bodik
Prof. Bob Brayton
Joel Galenson
Thibaud Hottelier
Sagar Jain
Chris Jones
Ali Sinan Koksal
Leo Meyerovich
Evan Pu
Casey Rodarmor
Prof. Koushik Sen
Prof. Sanjit Seshia
Lexin Shan
Saurabh Srivastava
Liviu Tancau
Nicholas Tung

MIT

Prof. Armando Solar-Lezama
Rishabh Singh
Kuat Yesenov
Jean Yung
Zhiley Xu

IBM

Satish Chandra
Kemal Ebcioglu
Rodric Rabbah
Vijay Saraswat
Vivek Sarkar