Syntax-Guided Synthesis

Rajeev Alur

Program Verification

- Does a program $P$ meet its specification $\varphi$?

- Historical roots: Hoare logic for formalizing correctness of structured programs (late 1960s)

- Early examples: sorting, graph algorithms

- Provides calculus for pre/post conditions of structured programs
Sample Proof: Selection Sort

SelectionSort(int A[], n) {
    i1 := 0;
    while (i1 < n - 1) {
        v1 := i1;
        i2 := i1 + 1;
        while (i2 < n) {
            if (A[i2] < A[v1])
                v1 := i2;
            i2++;
        }
        swap(A[i1], A[v1]);
        i1++;
    }
    return A;
}

Invariant:
∀ k1, k2. 0 ≤ k1 < k2 < n ∧
    k1 < i1 ⇒ A[k1] ≤ A[k2]

Invariant:
    i1 < i2 ∧
    i1 ≤ v1 < n ∧
    (∀ k1, k2. 0 ≤ k1 < k2 < n ∧
     k1 < i1 ⇒ A[k1] ≤ A[k2]) ∧
    (∀ k. i1 ≤ k < i2 ∧
     k ≥ 0 ⇒ A[v1] ≤ A[k])

post: ∀ k. 0 ≤ k < n ⇒ A[k] ≤ A[k + 1]
Towards Practical Program Verification

1. Focus on simpler verification tasks:
   - Not full functional correctness, just absence of specific errors
   - Success story: Array accesses are within bounds

2. Provide automation as much as possible
   - Program verification is undecidable
   - Programmer asked to give annotations when absolutely needed
   - Consistency of annotations checked by SMT solvers

3. Use verification technology for synergistic tasks
   - Directed testing
   - Bug localization
Selection Sort: Array Access Correctness

```
SelectionSort(int A[], n) {
    i1 := 0;
    while (i1 < n-1) {
        v1 := i1;
        i2 := i1 + 1;
        while (i2 < n) {
            assert (0 ≤ i2 < n) & (0 ≤ v1 < n)
            if (A[i2] < A[v1])
                v1 := i2;
            i2++;
        }
        assert (0 ≤ i1 < n) & (0 ≤ v1 < n)
        swap(A[i1], A[v1]);
        i1++;
    }
    return A;
}
```
Selection Sort: Proving Assertions

```c
SelectionSort(int A[], n) {
    i1 := 0;
    while (i1 < n-1) {
        v1 := i1;
        i2 := i1 + 1;
        while (i2 < n) {
            assert 0 ≤ i2 < n & 0 ≤ v1 < n
            if (A[i2] < A[v1])
                v1 := i2;
                i2++;
        }
        assert (0 ≤ i1 < n) & 0 ≤ v1 < n
        swap(A[i1], A[v1]);
        i1++;
    }
    return A;
}
```

Check validity of formula

\((i1 = 0) \& (i1 < n-1) \Rightarrow (0 ≤ i1 < n)\)

And validity of formula

\((0 ≤ i1 < n) \& (i1' = i1+1) \& (i1' < n-1) \Rightarrow (0 ≤ i1' < n)\)
Discharging Verification Conditions

- Check validity of
  \[(i_1 = 0) \& (i_1 < n-1) \Rightarrow (0 \leq i_1 < n)\]

- Reduces to checking satisfiability of
  \[(i_1 = 0) \& (i_1 < n-1) \& \sim(0 \leq i_1 < n)\]

- Core computational problem: checking satisfiability

  - Classical satisfiability: SAT
    - Boolean variables + Logical connectives

  - SMT: Constraints over typed variables
    - \(i_1\) and \(n\) are of type Integer or BitVector[32]
A Brief History of SAT

- **Fundamental Thm of CS:** SAT is NP-complete (Cook, 1971)
  - Canonical computationally intractable problem
  - Driver for theoretical understanding of complexity

- **Enormous progress in scale of problems that can be solved**
  - Inference: Discover new constraints dynamically
  - Exhaustive search with pruning
  - Algorithm engineering: Exploit architecture for speed-up

- **SAT solvers as the canonical computational hammer!**

---

1952 Quine
≈ 10 var

1960
DP
≈ 10 var

1962 DLL
≈ 10 var

1986 BDDs
≈ 100 var

1988 Socrates
≈ 300 var

1992 GSAT
≈ 300 var

1994 Hannibal
≈ 3k var

1996 Grasp
≈ 1k var

1996 Stålmarck
≈ 1000 var

1996 SATO
≈ 1k var

1996 Berkmin
≈ 10k var

2001 Chaff
≈ 10k var

2002 Berkmin
≈ 10k var

2005 MiniSAT
≈ 20k var
SMT: Satisfiability Modulo Theories

- Computational problem: Find a satisfying assignment to a formula
  - Boolean + Int types, logical connectives, arithmetic operators
  - Bit-vectors + bit-manipulation operations in C
  - Boolean + Int types, logical/arithmetic ops + Uninterpreted functs

- “Modulo Theory”: Interpretation for symbols is fixed
  - Can use specialized algorithms (e.g. for arithmetic constraints)

- Progress in improved SMT solvers

Little Engines of Proof

SAT; Linear arithmetic; Congruence closure
SMT Success Story

SMT Solvers  ↔  Verification Tools

- CBMC
- SAGE
- VCC
- Spec#

SMT-LIB Standardized Interchange Format (smt-lib.org)
- Problem classification + Benchmark repositories
- LIA, LIA_UF, LRA, QF_LIA, ...

+ Annual Competition (smt-competition.org)

- Z3
- Yices
- CVC4
- MathSAT5
Program Synthesis

- Classical: Mapping a high-level (e.g. logical) specification to an executable implementation

- Benefits of synthesis:
  - Make programming easier: Specify “what” and not “how”
  - Eliminate costly gap between programming and verification

- Deductive program synthesis: Constructive proof of Exists f. \( \varphi \)
Verification

Program Verification:
Does P meet spec $\varphi$?

SMT:
Is $\varphi$ satisfiable?

SMT-LIB:
Standard API
Solver competition

Synthesis

Program Synthesis:
Find P that meets spec $\varphi$

Syntax-Guided Synthesis

Plan for SYNTH-LIB
**Superoptimizing Compiler**

- Given a program $P$, find a “better” equivalent program $P'$

```plaintext
multiply (x[1,n], y[1,n]) {
    x1 = x[1,n/2];
    x2 = x[n/2+1, n];
    y1 = y[1, n/2];
    y2 = y[n/2+1, n];
    a = x1 * y1;
    b = shift( x1 * y2, n/2);
    c = shift( x2 * y1, n/2);
    d = shift( x2 * y2, n);
    return ( a + b + c + d)
}
```

Replace with equivalent code with only 3 multiplications
Automatic Invariant Generation

SelectionSort(int A[], n) {
    i1 := 0;
    while (i1 < n - 1) {
        v1 := i1;
        i2 := i1 + 1;
        while (i2 < n) {
            if (A[i2] < A[v1])
                v1 := i2;
            i2++;
        }
        swap(A[i1], A[v1]);
        i1++;
    }
    return A;
}

post: \( \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k + 1] \)
Template-based Automatic Invariant Generation

SelectionSort(int A[], n) {
    i1 := 0;
    while (i1 < n - 1) {
        v1 := i1;
        i2 := i1 + 1;
        while (i2 < n) {
            if (A[i2] < A[v1])
                v1 := i2;
            i2++;
        }
        swap(A[i1], A[v1]);
        i1++;
    }
    return A;
}

post: \( \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k + 1] \)
Template-based Automatic Invariant Generation

SelectionSort(int A[], n) {
    i1 := 0;
    while (i1 < n - 1) {
        v1 := i1;
        i2 := i1 + 1;
        while (i2 < n) {
            if (A[i2] < A[v1])
                v1 := i2;
            i2++;
        }
        swap(A[i1], A[v1]);
        i1++;
    }
    return A;
}

post: \( \forall k: 0 \leq k < n \Rightarrow A[k] \leq A[k + 1] \)

Invariant:
\( \forall k_1, k_2. 0 \leq k_1 < k_2 < n \land k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2] \)

Invariant:
i_1 < i_2 \land i_1 \leq v_1 < n \land (\forall k_1, k_2. 0 \leq k_1 < k_2 < n \land k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2]) \land (\forall k. i_1 \leq k < i_2 \land k \geq 0 \Rightarrow A[v_1] \leq A[k]) \)
Err = 0.0;
for(t = 0; t<T; t+=dT){
    if(stage==STRAIGHT){
        if(t > ??) stage= INTURN;
    }
    if(stage==INTURN){
        car.ang = car.ang - ??;
        if(t > ??) stage= OUTTURN;
    }
    if(stage==OUTTURN){
        car.ang = car.ang + ??;
        if(t > ??) break;
    }
    simulate_car(car);
    Err += check_collision(car);
}
Err += check_destination(car);
The program requires 3 changes:

- In the return statement `return deriv` in line 5, replace `deriv` by `[0].`
- In the comparison expression `(poly[e] == 0)` in line 7, change `(poly[e] == 0)` to False.
- In the expression `range(0, len(poly))` in line 6, replace 0 by 1.
FlashFill: Programming by Examples

Ref: Gulwani (POPL 2011)

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(425)-706-7709</td>
<td>425-706-7709</td>
</tr>
<tr>
<td>510.220.5586</td>
<td>510-220-5586</td>
</tr>
<tr>
<td>1 425 235 7654</td>
<td>425-235-7654</td>
</tr>
<tr>
<td>425 745-8139</td>
<td>425-745-8139</td>
</tr>
</tbody>
</table>

- Infers desired Excel macro program
- Iterative: user gives examples and corrections
- Being incorporated in next version of Microsoft Excel
Syntax-Guided Program Synthesis

- Core computational problem: Find a program \( P \) such that
  1. \( P \) is in a set \( E \) of programs (syntactic constraint)
  2. \( P \) satisfies \( \text{spec} \ \varphi \) (semantic constraint)

- Common theme to many recent efforts
  - Sketch (Bodik, Solar-Lezama et al)
  - FlashFill (Gulwani et al)
  - Super-optimization (Schkufza et al)
  - Invariant generation (Many recent efforts...)
  - TRANSIT for protocol synthesis (Udupa et al)
  - Oracle-guided program synthesis (Jha et al)
  - Implicit programming: Scala^Z3 (Kuncak et al)
  - Auto-grader (Singh et al)

But no way to share benchmarks and/or compare solutions
Syntax-Guided Synthesis (SyGuS) Problem

- Fix a background theory \( T \): fixes types and operations

- Function to be synthesized: name \( f \) along with its type
  - General case: multiple functions to be synthesized

- Inputs to SyGuS problem:
  - Specification \( \varphi \)
    - Typed formula using symbols in \( T \) + symbol \( f \)
  - Set \( E \) of expressions given by a context-free grammar
    - Set of candidate expressions that use symbols in \( T \)

- Computational problem:
  - Output \( e \) in \( E \) such that \( \varphi[f/e] \) is valid (in theory \( T \))
SyGuS Example

- Theory QF-LIA
  Types: Integers and Booleans
  Logical connectives, Conditionals, and Linear arithmetic
  Quantifier-free formulas

- Function to be synthesized: \( f(\text{int } x, \text{int } y) : \text{int} \)

- Specification: \((x \leq f(x,y)) \land (y \leq f(x,y)) \land (f(x,y) = x \lor f(x,y) = y)\)

- Candidate Implementations: Linear expressions
  \( \text{LinExp} := x \mid y \mid \text{Const} \mid \text{LinExp} + \text{LinExp} \mid \text{LinExp} - \text{LinExp} \)

- No solution exists
SyGuS Example

- Theory QF-LIA

- Function to be synthesized: \( f(\text{int } x, \text{int } y) : \text{int} \)

- Specification: \((x \leq f(x,y)) \land (y \leq f(x,y)) \land (f(x,y) = x \lor f(x,y) = y)\)

- Candidate Implementations: Conditional expressions without +

\[
\text{Term} := x \mid y \mid \text{Const} \mid \text{If-Then-Else} (\text{Cond}, \text{Term}, \text{Term}) \\
\text{Cond} := \text{Term} \leq \text{Term} \mid \text{Cond} \land \text{Cond} \mid \sim \text{Cond} \mid (\text{Cond})
\]

- Possible solution:

\(\text{If-Then-Else} (x \leq y, y, x)\)
Let Expressions and Auxiliary Variables

- Synthesized expression maps directly to a straight-line program
- Grammar derivations correspond to expression parse-trees
- How to capture common subexpressions (which map to aux vars)?
- Solution: Allow “let” expressions

Candidate-expressions for a function \( f(int x, int y) : int \)

\[
T := (let [z = U] in \ z + z) \\
U := x | y | Const | (U) | U + U | U*U
\]
Optimality

- Specification for $f(int \ x) : int$
  
  \[ x \leq f(x) \land -x \leq f(x) \]

- Set $E$ of implementations: Conditional linear expressions

- Multiple solutions are possible
  
  If-Then-Else \( (0 \leq x, x, 0) \)
  
  If-Then-Else \( (0 \leq x, x, -x) \)

- Which solution should we prefer?
  
  Need a way to rank solutions (e.g. size of parse tree)
Invariant Generation as SyGuS

- Goal: Find inductive loop invariant automatically

- Function to be synthesized
  \[ \text{Inv} (\text{bool} \ x, \text{bool} \ z, \text{int} \ a, \text{int} \ b) : \text{bool} \]

- Compile loop-body into a logical predicate
  \[ \text{Body}(x,y,z,a,b,c, x',y',z',a',b',c') \]

- Specification:
  \[ \text{Inv} \& \text{Body} \& \text{Test}' \Rightarrow \text{Inv}' \]

- Template for set of candidate invariants
  \[ \text{Term} := a \mid b \mid \text{Const} \mid \text{Term} + \text{Term} \mid \text{If-Then-Else} (\text{Cond}, \text{Term}, \text{Term}) \]
  \[ \text{Cond} := x \mid z \mid \text{Cond} \& \text{Cond} \mid \sim \text{Cond} \mid (\text{Cond}) \]
Program Optimization as SyGuS

- Type matrix: 2x2 Matrix with Bit-vector[32] entries
  Theory: Bit-vectors with arithmetic

- Function to be synthesized \( f(\text{matrix } A, B) : \text{matrix} \)

- Specification: \( f(A,B) \) is matrix product
  \[
  \ldots
  \]

- Set of candidate implementations
  Expressions with at most 7 occurrences of *
  Unrestricted use of +
  let expressions allowed
Program Sketching as SyGuS

- Sketch programming system
  - C program P with ?? (holes)
  - Find expressions for holes so as to satisfy assertions

- Each hole corresponds to a separate function symbol

- Specification: P with holes filled in satisfies assertions
  - Loops/recursive calls in P need to be unrolled fixed no of times

- Set of candidate implementations for each hole:
  - All type-consistent expressions

- Not yet explored:
  - How to exploit flexibility of separation betw syntactic and semantic constraints for computational benefits?
Solving SyGuS

- Is SyGuS same as solving SMT formulas with quantifier alternation?

- SyGuS can sometimes be reduced to Quantified-SMT, but not always
  - Set E is all linear expressions over input vars x, y
    - SyGuS reduces to Exists a,b,c. Forall X. $\varphi$ [ f/ ax+by+c]
  - Set E is all conditional expressions
    - SyGuS cannot be reduced to deciding a formula in LIA

- Syntactic structure of the set E of candidate implementations can be used effectively by a solver

- Existing work on solving Quantified-SMT formulas suggests solution strategies for SyGuS
SyGuS as Active Learning

Initial examples I

Learning Algorithm

Verification Oracle

Candidate Expression

Counterexample

Fail

Success

Concept class: Set $E$ of expressions

Examples: Concrete input values
Counter-Example Guided Inductive Synthesis

- Concrete inputs $I$ for learning $f(x,y) = \{(x=a,y=b), (x=a',y=b'), \ldots\}$

- Learning algorithm proposes candidate expression $e$ such that $\varphi[f/e]$ holds for all values in $I$

- Check if $\varphi[f/e]$ is valid for all values using SMT solver

- If valid, then stop and return $e$

- If not, let $(x=\alpha, y=\beta, \ldots)$ be a counter-example (satisfies $\sim \varphi[f/e]$)

- Add $(x=\alpha, y=\beta)$ to tests $I$ for next iteration
CEGIS Example

- Specification: \((x \leq f(x,y)) \& (y \leq f(x,y)) \& (f(x,y) = x \mid f(x,y) = y)\)

- Set E: All expressions built from \(x, y, 0, 1,\) Comparison, +, If-Then-Else

Examples = \{\}

Candidate
\(f(x,y) = x\)

Example
\((x=0, y=1)\)
CEGIS Example

- Specification: \((x \leq f(x,y)) \& (y \leq f(x,y)) \& (f(x,y) = x \mid f(x,y) = y)\)

- Set \(E\): All expressions built from \(x, y, 0, 1, \text{Comparison, +, If-Then-Else}\)

Examples = \{(x=0, y=1)\}

Candidate 
\(f(x,y) = y\)

Example 
\((x=1, y=0)\)
CEGIS Example

- Specification: \((x \leq f(x,y)) \land (y \leq f(x,y)) \land (f(x,y) = x \lor f(x,y) = y)\)

- Set E: All expressions built from \(x, y, 0, 1, \text{Comparison}, +, \text{If-Then-Else}\)

Examples =
{\((x=0, y=1)\), \((x=1, y=0)\), \((x=0, y=0)\), \((x=1, y=1)\)}
SyGuS Solutions

- CEGIS approach (Solar-Lezama, Seshia et al)

- Similar strategies for solving quantified formulas and invariant generation

- Learning strategies based on:
  - Enumerative (search with pruning): Udupa et al (PLDI’13)
  - Symbolic (solving constraints): Gulwani et al (PLDI’11)
  - Stochastic (probabilistic walk): Schkufza et al (ASPLOS’13)
Enumerative Learning

- Find an expression consistent with a given set of concrete examples

- Enumerate expressions in increasing size, and evaluate each expression on all concrete inputs to check consistency

- Key optimization for efficient pruning of search space:
  Expressions $e_1$ and $e_2$ are equivalent if $e_1(a,b)=e_2(a,b)$ on all concrete values ($x=a,y=b$) in Examples
  Only one representative among equivalent subexpressions needs to be considered for building larger expressions

- Fast and robust for learning expressions with ~ 15 nodes
Symbolic Learning

- Suppose we know upper bound on no. of occurrences of each symbol

- Variables encode edges in desired expression tree
  E.g. l9, r9 : {n1, ... n10} give left and right children of node n9

- Constraints:
  Types are consistent, Shape is a DAG
  Spec $\varphi[f/e]$ is satisfied on every concrete input values in I

- Use an SMT solver to find a satisfying solution

- If unsatisfied, then bounds need to be increased in outer loop
Stochastic Learning

- Idea: Find desired expression $e$ by probabilistic walk on graph where nodes are expressions and edges capture single-edits.

- For a given set $I$ of concrete inputs, $\text{Score}(e) = \exp(-0.5 \ \text{Wrong}(e))$, where $\text{Wrong}(e) = \text{No of examples in I for which } \varphi [f/e]$.

- Fix $n$ and consider $E_n$ to be set of all expressions in $E$ of size $n$.

- Initialize: Choose $e$ by uniform sampling of $E_n$.

- If $\text{Score}(e)=1$ then return $e$, else:
  - Choose a node $v$ in parse-tree of $e$ at random.
  - Replace subtree at $v$ by a random subtree of same size to get $e'$.
  - Update $e$ to $e'$ with probability $\min\{1, \text{Score}(e')/\text{Score}(e)\}$.

- Outer loop responsible for updating expression size $n$. 
Benchmarks and Implementation

- Prototype implementation of Enumerative/Symbolic/Stochastic CEGIS

- Benchmarks:
  - Bit-manipulation programs from Hacker’s delight
  - Integer arithmetic: Find max, search in sorted array
  - Challenge problems such as computing Morton’s number

- Multiple variants of each benchmark by varying grammar

- Results are not conclusive as implementations are unoptimized, but offers first opportunity to compare solution strategies
Evaluation

- Enumerative CEGIS has best performance, and solves many benchmarks within seconds
  Potential problem: Synthesis of complex constants

- Symbolic CEGIS is unable to find answers on most benchmarks
  Caveat: Sketch succeeds on many of these

- Choice of grammar has impact on synthesis time
  When E is set of all possible expressions, solvers struggle

- None of the solvers succeed on some benchmarks
  Morton constants, Search in integer arrays of size > 4

- Bottomline: Improving solvers is a great opportunity for research!
SyGuS Recap

- **Contribution:** Formalization of syntax-guided synthesis problem
  - Not language specific such as Sketch, Scala^Z3,…
  - Not as low-level as (quantified) SMT

- **Advantages compared to classical synthesis**
  1. Set E can be used to restrict search (computational benefits)
  2. Programmer flexibility: Mix of specification styles
  3. Set E can restrict implementation for resource optimization
  4. Beyond deductive solution strategies: Search, inductive inference

- Prototype implementation of 3 solution strategies

- Initial set of benchmarks and evaluation
From SMT-LIB to SYNTH-LIB

(set-logic LIA)
(synth-fun max2 ((x Int) (y Int)) Int
  ((Start Int (x y 0 1
      (+ Start Start)
      (- Start Start)
    (ite StartBool Start Start)))
  (StartBool Bool ((and StartBool StartBool)
    (or StartBool StartBool)
    (not StartBool)
    (<= Start Start))))

(declare-var x Int)
(declare-var y Int)
(constraint (>= (max2 x y) x))
(constraint (>= (max2 x y) y))
(constraint (or (= x (max2 x y)) (= y (max2 x y))))
(check-synth)
Plan for Synth-Comp

- Proposed competition of SyGuS solvers at FLoC, July 2014

- Organizers: Alur, Fisman (Penn) and Singh, Solar-Lezama (MIT)

- Website: excape.cis.upenn.edu/Synth-Comp.html

- Mailing list: synthlib@cis.upenn.edu

- Call for participation:
  - Join discussion to finalize synth-lib format and competition format
  - Contribute benchmarks
  - Build a SyGuS solver
SyGuS Solvers ↔ Synthesis Tools

- Program optimization
- Program sketching
- Programming by examples
- Invariant generation

SYNTH-LIB Standardized Interchange Format
  Problem classification + Benchmark repository
+ Solvers competition

Potential Techniques for Solvers:
  Learning, Constraint solvers, Enumerative/stochastic search

Little engines of synthesis?