Verifying Robustness of Programs Under Structural Perturbations

Clay Thomas and Jacob Bond

December 7, 2017
Motivation

- An attempt to synthesize the max function using PBE:
  - $(13, 15) \mapsto 15$
  - $(-23, 19) \mapsto 19$
  - $(-75, -13) \mapsto -13$

Neither synthesized program, nor synthesizer are robust.
An attempt to synthesize the max function using PBE:
- \((13, 15) \mapsto 15\)
- \((-23, 19) \mapsto 19\)
- \((-75, -13) \mapsto -13\)

Synthesized program: \(P(a, b) := \text{return } b\)
• An attempt to synthesize the max function using PBE:
  • \((13, 15) \mapsto 15\)
  • \((-23, 19) \mapsto 19\)
  • \((-75, -13) \mapsto -13\)
• Synthesized program: \(P(a, b) := \text{return } b\)
• Neither synthesized program, nor synthesizer are robust
Robustness

- Robustness: behaving predictably on uncertain inputs [2]
Robustness

- Robustness: behaving predictably on uncertain inputs [2]
- $P(13, 15) \neq P(15, 13)$
Robustness

- Robustness: behaving predictably on uncertain inputs [2]
- $P(13, 15) \neq P(15, 13)$
  - $(15, 13) \mapsto 15$
  - $(19, -23) \mapsto 19$
  - $(-13, -75) \mapsto -13$

would synthesize very different program
Robustness

- Robustness: behaving predictably on uncertain inputs [2]
- \( P(13,15) \neq P(15,13) \)
  - \( (15,13) \mapsto 15 \)
  - \( (19,-23) \mapsto 19 \)
  - \( (-13,-75) \mapsto -13 \)

  would synthesize very different program

- Synthesize a robust program or develop robust synthesizer
Robustness Properties

- **Continuity**: small change to input \(\Rightarrow\) small change to output
  \[
  \text{Sort([1,4,3,6])} = [1,3,4,6] \\
  \text{Sort([2,3,3,5])} = [2,3,3,5]
  \]
Robustness Properties

- **Continuity**: small change to input ⇒ small change to output
  \[
  \text{Sort}([1,4,3,6]) = [1,3,4,6] \\
  \text{Sort}([2,3,3,5]) = [2,3,3,5]
  \]

- **Permutation**: permuting input leaves output invariant
  \[
  \text{Sort}([1,4,3,6]) = [1,3,4,6] \\
  \text{Sort}([6,3,1,4]) = [1,3,4,6]
  \]
Robustness Properties

- **Continuity**: small change to input ⇒ small change to output
  
  \[
  \text{Sort}([1,4,3,6]) = [1,3,4,6] \\
  \text{Sort}([2,3,3,5]) = [2,3,3,5]
  \]

- **Permutation**: permuting input leaves output invariant
  
  \[
  \text{Sort}([1,4,3,6]) = [1,3,4,6] \\
  \text{Sort}([6,3,1,4]) = [1,3,4,6]
  \]

- **Simultaneous Permutation**: permuting all inputs leaves output invariant (Grade(responses, answers))
  
  \[
  \text{Grade}([\sqrt{x^2}, 1/e, 6.5], [\abs{x}, e^{-1}, 13/2]) = 1 \\
  \text{rearrange problem parts} \\
  \text{Grade}([1/e, 6.5, \sqrt{x^2}], [e^{-1}, 13/2, \abs{x}]) = 1
  \]
Verifying Continuity [1]

• Consider

1: \textbf{if} \ x \geq 0 \ \textbf{then}
2: \qquad r := y
3: \textbf{else}
4: \qquad r := z

• If \( y \neq z \), discontinuous at \( x = 0 \)

• Proof rule:

\[ c \vdash \text{Cont}(P_1, \text{In}, \text{Out}) \]
\[ c \vdash \text{Cont}(P_2, \text{In}, \text{Out}) \]
\[ c' \vdash \text{Cont}(b, \text{Var}(b)) \quad (c \land \neg c') \vdash \text{Out}P_1 = \text{Out}P_2 \]

• Only applicable to numerical perturbations
Verifying Continuity [1]

• Consider

1: if \( x \geq 0 \) then
2: \( r := y \)
3: else
4: \( r := z \)

• If \( y \neq z \), discontinuous at \( x = 0 \)
Verifying Continuity [1]

- Consider
  
  1. `if x ≥ 0 then`
  2. \( r := y \)
  3. `else`
  4. \( r := z \)

- If \( y \neq z \), discontinuous at \( x = 0 \)

- Proof rule:

\[
\begin{align*}
    c \vdash \text{Cont}(P_1, \text{In}, \text{Out}) & \quad c \vdash \text{Cont}(P_2, \text{In}, \text{Out}) \quad c' \vdash \text{Cont}(b, \text{Var}(b)) \\
    (c \land \neg c') \vdash \text{Out}_{P_1} = \text{Out}_{P_2}
\end{align*}
\]

\[c \vdash \text{Cont}(\text{if} \ b \ \text{then} \ P_1 \ \text{else} \ P_2, \text{In}, \text{Out})\]
Verifying Continuity [1]

• Consider

1: \textbf{if} \ x \geq 0 \ \textbf{then}
2: \quad r := y
3: \textbf{else}
4: \quad r := z

• If \ y \neq z, \ discontinuous \ at \ x = 0

• Proof rule:

\[
\begin{align*}
&c \vdash \text{Cont}(P_1, \text{In}, \text{Out}) \\
&c' \vdash \text{Cont}(b, \text{Var}(b)) \\
&c \land \neg c' \vdash \text{Out}_{P_1} = \text{Out}_{P_2} \\
\end{align*}
\]

\[
\begin{align*}
&c \vdash \text{Cont}(\textbf{if} \ b \ \textbf{then} \ P_1 \ \textbf{else} \ P_2, \text{In}, \text{Out})
\end{align*}
\]

• Only applicable to numerical perturbations
Cartesian Hoare Logic [5]

- Robustness requires two executions

Cartesian Hoare Triple examples:
- Determinism: $\parallel \vec{x}_1 = \vec{x}_2 \parallel f(\vec{x}) \parallel \text{ret}_1 = \text{ret}_2$
- Symmetry: $\parallel x_1 = y_2 \land x_2 = y_1 \parallel f(x_1, y_2) \parallel \text{ret}_1 = \text{ret}_2$

Requires specifying property in first-order logic

Not optimized for 2-safety properties
Cartesian Hoare Logic [5]

- Robustness requires two executions
- Verified using product program
Cartesian Hoare Logic [5]

- Robustness requires two executions
- Verified using product program
  - $P_1 \otimes P_2$ is simultaneous execution
Cartesian Hoare Logic [5]

- Robustness requires two executions
- Verified using product program
  - $P_1 \otimes P_2$ is simultaneous execution
- Cartesian Hoare Logic reasons about product programs
Cartesian Hoare Logic [5]

- Robustness requires two executions
- Verified using product program
  - $P_1 \otimes P_2$ is simultaneous execution
- Cartesian Hoare Logic reasons about product programs
- Cartesian Hoare Triple examples:

### Determinism

\[ \parallel \vec{x}_1 = \vec{x}_2 \parallel f(\vec{x}) \parallel \text{ret}_1 = \text{ret}_2 \]

### Symmetry

\[ \parallel x_1 = y_2 \land x_2 = y_1 \parallel f(x, y) \parallel \text{ret}_1 = \text{ret}_2 \]
Cartesian Hoare Logic [5]

- Robustness requires two executions
- Verified using product program
  - $P_1 \otimes P_2$ is simultaneous execution
- Cartesian Hoare Logic reasons about product programs
- Cartesian Hoare Triple examples:
  - Determinism:
    $$\parallel \vec{x}_1 = \vec{x}_2 \parallel f(\vec{x}) \parallel ret_1 = ret_2 \parallel$$
  - Symmetry:
    $$\parallel x_1 = y_2 \land x_2 = y_1 \parallel f(x, y) \parallel ret_1 = ret_2 \parallel$$

Requires specifying property in first-order logic
Not optimized for 2-safety properties
Cartesian Hoare Logic [5]

- Robustness requires two executions
- Verified using product program
  - $P_1 \otimes P_2$ is simultaneous execution
- Cartesian Hoare Logic reasons about product programs
- Cartesian Hoare Triple examples:
  - Determinism:
    $\| x_1 = x_2 \| f(x) \| ret_1 = ret_2 \|
  - Symmetry:
    $\| x_1 = y_2 \land x_2 = y_1 \| f(x, y) \| ret_1 = ret_2 \|$
Cartesian Hoare Logic [5]

- Robustness requires two executions
- Verified using product program
  - $P_1 \otimes P_2$ is simultaneous execution
- Cartesian Hoare Logic reasons about product programs
- Cartesian Hoare Triple examples:
  - Determinism:
    $$\|\vec{x}_1 = \vec{x}_2\| f(\vec{x})\| ret_1 = ret_2\|
  - Symmetry:
    $$\|x_1 = y_2 \land x_2 = y_1\| f(x, y)\| ret_1 = ret_2\|
  - Requires specifying property in first-order logic
Cartesian Hoare Logic [5]

- Robustness requires two executions
- Verified using product program
  - $P_1 \otimes P_2$ is simultaneous execution
- Cartesian Hoare Logic reasons about product programs
- Cartesian Hoare Triple examples:
  - Determinism:
    $$\parallel \vec{x}_1 = \vec{x}_2 \parallel f(\vec{x}) \parallel \text{ret}_1 = \text{ret}_2 \parallel$$
  - Symmetry:
    $$\parallel x_1 = y_2 \land x_2 = y_1 \parallel f(x, y) \parallel \text{ret}_1 = \text{ret}_2 \parallel$$

- Requires specifying property in first-order logic
- Not optimized for 2-safety properties
Our Contributions

Goals:

• Reason about invariance under discrete perturbations
• Want to optimize for our specific problem

Results:

• Small sets of perturbations that “generate” all perturbations
  • Lists, binary search trees
• Formulate “invariance with respect to a function”
  • General, sound procedure
• Sanity checks and bug finding
Lists – Invariance under order

Given an array $a$

- Let $a_{swap}$ be $a$ with its first and second entry swapped
  - $[a[1], a[0], a[2], a[3], \ldots, a[n]]$
- Let $a_{rot}$ be $a$ rotated by 1
  - $[a[1], a[2], a[3], \ldots a[n], a[0]]$

Lemma: If for any $a$, $P(a) = P(a_{swap}) = P(a_{rot})$, then for any permutation $a'$ of $a$, we have $P(a) = P(a')$.
Proof: Math [3]
Programs – Invariance under order

- \( \text{maxList([x])} = x \)
- \( \text{maxList([x, ...xs...])} = \text{max}(x, \text{maxList(xs)}) \)

- Verifying \( \text{maxList(a)} = \text{maxList(a}_{\text{swap}}) \) has one case:

\[
\text{maxList([x, y, ...xs...])} \equiv \text{maxList([y, x, ...xs...])} \\
\text{max}(x, \text{maxList([y, ...xs...]))} \equiv \text{max}(y, \text{maxList([x, ...xs...]))} \\
\text{max}(x, \text{max}(y, \text{maxList(xs))}) \equiv \text{max}(y, \text{max}(x, \text{maxList(xs))}) \\
\text{max}(x, \text{max}(y, z)) \equiv \text{max}(y, \text{max}(x, z))
\]
Binary Search Trees

- For lists, two simple permutations generated all permutations
- Goal: similar permutations for BSTs
Binary Search Trees

```
Binary Search Trees

a
   
   b
   
   LL
   
   LR

rotate

LL

a
   
   b
   
   LR
   
   R

flatten

a
   
   b
   
   LRL
   
   LRR

flatten

a
   
   b
   
   LL
   
   LRL

```
Binary Search Trees

It suffices to show

- Every tree can be transformed into a “normal form” (i.e. list)
  - “flatten” straightens out the tree
  - “rotate” lets you straighten all the parts
- Every operation is reversible
Lists and Binary Search Trees

- Can check robustness under ALL permutations by checking just TWO permutations
More General Procedure

- Sets of permutations are case-by-case
- Goal: formulation of invariance
  - Useful
  - Easy to code/express
  - Checkable
More General Procedure

Invariance of a program $P : T \rightarrow Z$ relative to a function $f : T \rightarrow T'$

- $f(t)$ gives a “canonical representative” of $t$
- For concreteness, $f = \text{list} : \text{BST} \rightarrow \text{List}$

Observation: The following are equivalent:

- $\text{list}(x) = \text{list}(y) \implies P(x) = P(y)$
- There exists a program $\tilde{P} : \text{Lists} \rightarrow Z$ such that $P(t) = \tilde{P}(\text{list}(t))$
More General Procedure

- Idea: Synthesize a witness to the invariance
  - A function $\tilde{P} : Lists \rightarrow Z$
- $P$ and list provide a full specification of $\tilde{P}$
- Counterexample guided inductive synthesis [4]

\[ \begin{align*}
\text{BSTs} & \quad \xrightarrow{P} \quad Z \\
& \quad \xrightarrow{\text{list}} \quad \tilde{P} \\
& \quad \xrightarrow{\text{Lists}}
\end{align*} \]
More General Procedure

Input $P$. Let $E = \{\}$. Synthesize a program $\tilde{P} : Lists \rightarrow Z$ given examples $\{(list(t), P(t))| t \in E\}$.

Verify whether $\tilde{P}(list(t))$ is equivalent to $P(t)$.

Check if there are any $t \in E$ such that $list(t) = list(t_0)$, yet $P(t) \neq P(t_0)$.

Add $t_0$ to the example set $E$.

return YES

Yes, equivalent

No, with counterexample $t_0 \in BSTs$

return NO

Yes, $t$ exists

No, $t$ does not exist.
Future Directions

• Develop proof rules for discrete perturbations
• Improved handling of branching programs by Cartesian Hoare Logic
• Working implementation of Cartesian Hoare Logic
• Find more data structures with small perturbation sets
• Speed up our general procedure
• Synthesis for verification?
• Implement!


