Verifying Robustness of Programs Under Structural Perturbations

Jacob Bond and Clay Thomas

Purdue University

1 Introduction

The question of robustness is fundamental to the subject of programming by example (PBE). Robustness of a program is the property that the program behaves predictably on uncertain inputs [2]. In the PBE paradigm, there is, by definition, an uncertainty about the intent of the user, and therefore, it is desirable that a program synthesizer behave predictably with regards to this uncertainty.

Consider an attempt to specify the max function by providing to a program synthesizer the examples $(13, 15) \mapsto 15$, $(-23, 19) \mapsto 19$, and $(-75, -13) \mapsto -13$. In order to synthesize a simpler program, the result will likely be the program `P(a,b):=return b;`. Two issues are at play here: the program syntehsizer is not robust and the synthesized program is also not robust.

First, the program synthesizer is not robust as transposing the inputs of all of the examples would result in the program `P(a,b):=return a;`, while transposing just a single input would result in the correct program

```
P(a,b):=return a>b?a:b;.
```

Second, the program which is synthesized is not robust as `P(a,b)`$\neq$`P(b,a)`. That is, the program does not behave predictably under uncertainty of the order of the arguments. If the program which is synthesized is required to be robust with respect to uncertainty in the order of the input, neither `P(a,b):=return a;` nor `P(a,b):=return b` would be viable candidates, and the synthesizer would be forced to return

```
P(a,b):=return a>b?a:b;.
```

Moreover, a synthesizer which returns robust programs will itself be more robust. Let $\mathcal{I}_1 = (I_1, O_1)$ and $\mathcal{I}_2 = (I_2, O_2)$ be two input-output pairs for a program synthesizer which differ by a small perturbation. If the program $\mathcal{P}_1$ returned by the synthesizer on input $\mathcal{I}_1$ is robust, then $\mathcal{P}_1(I_2)$ will approximate O_2 because I_2 approximates I_1. For this reason, $\mathcal{P}_2$, the program returned on input $\mathcal{I}_2$, should only differ from $\mathcal{P}_1$ by a small amount.

Thus, the issue of robustness in PBE can be addressed by verifying robustness, either of the synthesized programs or even of the synthesizer as a whole. However, verification of robustness requires the ability to reason about robustness.

2 Related Work

Relational Logics Benton [**?**] established a Relational Hoare Logic in order to reason about relational properties, properties which consider two, usually distinct, programs. Barthe et al. [**?**,**?**] extended Benton's Relational Hoare Logic to probabalistic programs in order to reason about cryptographic protocols and differential privacy.

Self-Composition Barthe et al. [**?**] applied self-composition, sequential running of renamed copies of the original program, to study secure information flow. Terauchi & Aiken [11] built on this work by applying a type-based approach to complement self-composition.

2-*Safety Properties* In [11], Terauchi & Aiken introduce the term 2-safety property to describe a property which requires two execution traces in order to reason about it. A general approach to the verification of 2-safety properties is the creation of a product program [**?**], a program which interleaves two copies of a given program to create a new program. It [**?**], Barthe et al. analyzes various relational program logics, as well as different notions of product programs.

Continuity Hamlet [9] considered the concept of program continuity, but declared that automating verification of continuity for programs with loops was infeasible. Chaudhuri et al. [4, 1] consider the continuity and Lipschitz continuity of programs over the real numbers. Samanta et al. [10] and Henzinger et al. [3] investigate the use of Lipschitz continuity for proving robustness in the context of transducers.

Robustness Robustness for control systems was investigated by Majumdar and Saha [7] and for general programs by [1]. Additionally, the robustness of networked systems was explored by Samanta et al. [8].

k-Safety Properties Clarkson and Schneider [12] introduced k-safety properties as a generalization of this idea. Sousa and Dillig [13] formulated a verification algorithm in order to automate checking of k-safety properties.

3 Preliminaries

3.1 Continuity and Lipschitz-Continuity

3.2 Permutations

A permutation is a bijection from a set Ω to itself [6]. When Ω is a finite set, a permutation can be specified using two-line notation by placing the elements of Ω on one line, and their images under σ beneath them:

$$\sigma\colon \begin{matrix} 0\,1\,2\,3\,4\,5 \\ 5\,2\,3\,1\,4\,0 \end{matrix} \qquad\qquad \sigma\colon \begin{matrix} \mathrm{a\ b\ c} \\ \mathrm{a\ c\ b} \end{matrix}$$

Similar to two-line notation, a permutation on $\{0, \ldots, N-1\}$ can be encoded in an array a of length N by placing the image of i in $a[i]$. The example σ above is encoded in an array as $\sigma = [5, 2, 3, 1, 4, 0]$, so that $\sigma(i) = \sigma[i]$.

The inverse σ^{-1} of a permutation σ is the permutation defined by

$$\sigma(i) = j \iff \sigma^{-1}(j) = i.$$

For the permutation σ above,

$$\sigma^{-1} : \begin{matrix} 0\ 1\ 2\ 3\ 4\ 5 \\ 5\ 3\ 1\ 2\ 4\ 0 \end{matrix}.$$

First-Order Formula for a Permutation The property of being a permutation on $\{0, \ldots, N-1\}$ is encoded in the theory of arrays as

$$\begin{aligned} \text{Perm}(a, N) : N = \text{length}(a) \wedge (\forall i.0 \leq i < N \rightarrow 0 \leq a[i] < N) \wedge \\ (\forall i, j.0 \leq i < j < N \rightarrow a[i] \neq a[j]). \end{aligned}$$

Action on an Array Given a permutation σ, σ can *act* on an array a as follows. Let $a = [37, 25, 19, 49, 81, 21]$ and σ be as above. Then $\sigma(3) = \sigma[3] = 1$ and applying σ to a results in the element of a at index 3, 49, being moved to index 1 in σa. That is, $\sigma a[1] = 49$ and $a[3] = \sigma a[1] = \sigma a[\sigma[3]]$. More generally,

$$\sigma a[\sigma[i]] = a[i] \iff \sigma a[i] = a[\sigma^{-1}[i]]. \tag{1}$$

Generating Permutations For information on permutation groups, see §1.3 in [6].

The permutations on N elements, S_N, can all be expressed as compositions of the permutations [Exercises 3-4, §3.5, 6]

$$\begin{matrix} 0\ 1\ 2 \cdots N-1 \\ 1\ 0\ 2 \cdots N-1 \end{matrix} \quad \text{and} \quad \begin{matrix} 0\ 1\ 2 \cdots N-1 \\ 1\ 2\ 3 \cdots \quad 0 \end{matrix},$$

where the first permutation is a transposition of 0 and 1 and the second permutation is a rotation of all of the elements by one position.

3.3 Automata

We consider two classes of automata: finite state machines (FSM) and finite state transducers (FST). All automata we consider are deterministic. Given an automata A, we denote by $A(s)$ the output of A on input s. If A is an FSM, we represent "accept" by 1 and "reject" by 0. Otherwise, A is a finite state transducer (FST), and $A(s)$ is a string. For a FSM A, let $L(A)$ denote the set of strings accepted by A. (We do not define the language of an FST).

Recall that for any FSMs A and B, the problem of determining whether $L(A) = L(B)$ is decidable. We can compose an FST T and a FSM A, denoted $A \circ T$, to get another FSM.

We will assume that all input strings are terminated by a special end-of-input character \$.

4 Robustness Properties

Some settings in which programs should be robust:

- numerical perturbations [3, 4, 1],
- permutations of arrays and matrices,
- simultaneous permutations of arrays,

4.1 Continuity

4.2 Permutations

Sorting

Searching Consider the function `Find(a, x)` which returns the index of the element `x` in the array `a` or -1 if `x` is not an element of `a`. Let σ be a permutation and suppose that `a[i]=x`. Then from (1)

$$\sigma\texttt{a}[\sigma(\texttt{i})] = \texttt{a[i]} = \texttt{x},$$

so that `Find(`σ`a, x)=`σ`(i)`. Considering σ as a permutation on nonnegative integers, $\sigma(-1) = -1$ and `Find(`σ`a, x)=`σ`(i)` holds even in the case that `x` is not an element of `a`. Thus, `Find` is robust in that perturbing the input array by a permutation σ perturbs the output of `Find` by the same permutation σ.

Adjacency Matrices

4.3 Simultaneous Permutation

Consider an algorithm for grading a multipart homework problem, which takes as input three arrays: the student's responses, the correct answers, and the credit to be awarded for each part. A reordering of the parts of the problem should not affect the credit which a student is awarded. As reordering the parts of the problem corresponds to simultaneously permuting the three input arrays, the grading algorithm should be invariant under simultaneous permutations of the input arrays. Using ideas from Section 3.2, this property can be expressed as the conjunction of the following two formulas:

$$\begin{aligned} x_2[1] = x_1[0] \wedge y_2[1] = y_1[0] \wedge z_2[1] = z_1[0] \wedge \\ x_2[0] = x_1[1] \wedge y_2[0] = y_1[1] \wedge z_2[0] = z_1[1] \wedge \\ \forall i.\, 2 \le i < \text{length}(y_1) \rightarrow (x_2[i] = x_1[i] \wedge y_2[i] = y_1[i] \wedge z_2[i] = z_1[i]) \end{aligned}$$

$$\begin{aligned} \text{length}(y_1) = N \wedge (x_2[0] = x_1[N-1] \wedge y_2[0] = y_1[N-1] \wedge z_2[0] = z_1[N-1]) \wedge \\ \forall i.\, 1 \le i < N \rightarrow (x_2[i] = x_1[i-1] \wedge y_2[i] = y_1[i-1] \wedge z_2[i] = z_1[i-1]) \end{aligned}$$

The first formula specifies that the first two parts of the problem are transposed, while the rest of the parts remain unchanged. The second specifies a rotation by one of the parts of the problem.

```
function GRADE(responses, answers, credits)
    points ← 0
    for 0 ≤ i ≤ length(answers) do
        if responses[i] − answers[i] = 0 then
            points ← points + credits[i]
    return points
```

4.4 Verifying Robustness

A 2-safety property is one which requires reasoning about two execution traces simultaneously [11]. Robustness is the property that given two inputs which are related by some form of uncertainty, the outputs should be related in a predictable way. Thus, robustness is a 2-safety property.

4.5 Shortcomings Regarding Lipschitz-Continuity

5 Invariance under permuting lists

A different problem we have been thinking about is more discrete and algebraic in nature. We want to verify that programs are invariant under permutations of their input arrays.

5.1 Automata

Given a program P which takes a linear data type as input. We may wish to verify that this program is invariant under reordering of the array, in other words, it is invariant under the action of S_n. Denote the output of P on input s by $P(s)$. Then we want to test whether $P(\sigma s) = P$ for all $\sigma \in S_n$.

As a first simplification, observe that is suffices to check our condition on a set of generators of S_n. Concretely, let $\alpha = (1\ 2\ 3\ ...\ n)$ and let $\beta = (1\ 2)$. Then any $\sigma \in S_n$ can be written as a product of elements of $\{\alpha, \beta\}$, say $\sigma = u_1 u_2 \dots u_m$ with $u_i \in \{\alpha, \beta\}$. So if we know that $P(\alpha s) = P(s)$ and $P(\beta s) = P(s)$, then $P((u_1 u_2 \dots u_m)s) = P((u_2 \dots u_m)s) = \dots = P(u_m s) = P(s)$.

For a fixed n and $\sigma \in S_n$, we can construct a finite state transducer T such that $T(s) = \sigma s$ for $|s| = n$. However, this does not give us any reasonable way of checking that a given finite state machine is invariant under permutation of the input string. Instead, we will construct finite state transducers T_α and T_β such that for any n and any string s with $|s| = n$, we have $T_\alpha(s) = (1\ 2\ \dots\ n)s$ and $T_\beta(s) = (1\ 2)s$. Then, determining whether an FSM M is invariant under permutation of its inputs is equivalent to determining whether

$$L(M \circ T_\alpha) = L(M) = L(M \circ T_\beta)$$

We now construct T_α. For each symbol $a \in \Sigma$, T_α has a transition from its start state s_0 to s_a, while reading in put a and writing ϵ output. For each a and

each $b \neq \$$, there is a transition from s_a to s_a, reading b and writing b. Then for each a, there is a transition from s_a to s_1, reading \$ and writing $a\$$.

We now construct T_β. For each symbol $a \in \Sigma$, T_α has a transition from its start state s_0 to s_a, while reading in put a and writing ϵ output. Each s_a has a transition to s_1 while reading input b (for any $b \in \Sigma$) and writing output ba. Then s_1 simply has transitions to s_1, reading any $a \in \Sigma$ and writing back a.

6 Invariance under permuting binary search trees

In the previous section 5, we found a reduction from checking *all* permutations of a list to checking a small set of permutations. It is natural to ask if there are other data types for which we can do this. Binary search trees are one such case, where just like lists, two permutations suffice to generate all equivalent binary search trees (i.e. tree representing equivalent ordered lists).

Represent binary search trees by the algebraic data type Then define two (partial) operations ρ and θ on binary search trees as follows:

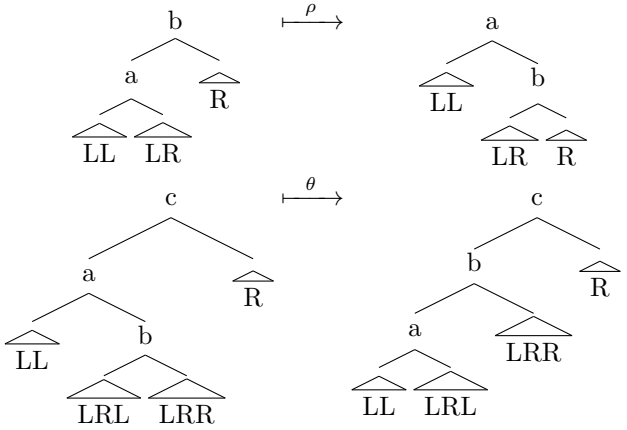
---

data Tree = Nil | Branch Tree Int Tree

list Nil = [ ]
list (Branch left a right) = list(left) ++ [a] ++ list(right)

ρ(Branch (Branch LL a LR) b R)
  = Branch LL a (Branch LR b R)

θ(Branch (Branch LL a (Branch LRL b LRR)) c R)
  = Branch LL a (Branch LR b R)



Theorem 1. *If a program P on binary search trees is invariant under ρ and θ, then whenever* $\text{list}(t_1) = \text{list}(t_2)$*, we have* $P(t_1) = P(t_2)$.

Proof. Observe that if P is invariant under ρ and θ, then P is also invariant under ρ^{-1} and θ^{-1}. Thus, we need to show that any two binary search trees representing the same ordered list can be transformed into one another using only ρ, ρ^{-1}, θ, and θ^{-1}. Because these operations are invertible, it suffices to demonstrate that you can use them to transform any binary search tree into a "degenerate list structure", in which no subtree has a nonnull left child.

Consider the following algorithm for adjusting a tree via ρ and θ:

```
1: function LISTIFY(t)
2:     while t has a right child do
3:         apply ρ^-1 to t
4:     while t has a left child do
5:         while t's left child has a right child do
6:             apply θ to t
7:         apply ρ to t
```

I claim that this algorithm terminates with t in degenerate list form. Proof: the first loop clearly terminates, and leave t with a null right child. Thus, after the first loop, t's right child is in degenerate list form.

Consider the inner loop on line 5. This loop clearly always terminates, and leave the right subtree of t unchanged. When this loop finishes, t's left child has a null right child. Observe that if ρ is applied to t when t's left child has a null right child AND t's right child is in degenerate list form, then the resulting tree's right subchild remains in degenerate list form. Thus, at each iteration of the second loop starting at line 4, we have t's right child in degenerate list form.

Furthermore, upon each iteration of the loop starting at line 4, we move one more node into the right subtree of t. Thus, this loop eventually halts, and t is left with a null left child. As t's right subtree is also in degenerate list form, t is left in degenerate list form.

Clearly, if t_1 and t_2 are both in degenerate list form, then $\text{list}(t_1) = \text{list}(t_2)$ if and only if $t_1 = t_2$. Thus, if $\text{list}(t_1) = \text{list}(t_2)$, then t_1 and t_2 can both be transformed into the same degenerate list structure useing the ρ and θ. Thus, t_1 can be transformed into t_2.

7 Invariance with respect to a function

All previous work on these topics share one downside: they require programmers to express their invariants in formal logic. This could potentially be difficult. One class of invariants that can be expressed through codes is *invariance with respect to a function*.

Definition 1. *We say that a program P is invariant with respect to a function f if whenever $f(x) = f(y)$, we have $P(x) = P(y)$.*