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 [1]. In the PBE paradigm, there is, by definition, an uncertainty regarding the intent of the user. Therefore, it is desirable that a program synthesizer behave predictably with regard to this uncertainty.

Consider an attempt to specify the max function by providing to a program synthesizer the examples $(3, 5) \mapsto 5$, $(−7, 9) \mapsto 9$, and $(−5, −8) \mapsto −8$. In order to synthesize a simpler program, the result will be the program $P(a, b) := \text{return } b;$. The issue that arises here is that neither the program synthesizer, nor the synthesized program, are robust.

The program synthesizer is not robust, as transposing the inputs of each example would result in the program $P(a, b) := \text{return } a;$, while transposing just a single input would result in the correct program $P(a, b) := \text{return } a>b?a:b;$. Additionally, 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 in 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 $P(a, b) := \text{return } a;$ nor $P(a, b) := \text{return } b$ would be viable candidates, and the synthesizer would be forced to return $P(a, b) := \text{return } a>b?a:b;$. Moreover, a synthesizer which returns robust programs will itself be more robust. Let $I_1 = (I_1, O_1)$ and $I_2 = (I_2, O_2)$ be two input-output pairs for a program synthesizer which differ by a small perturbation. If the program $P_1$ returned by the synthesizer on input $I_1$ is robust, then $P_1(I_2)$ will approximate $O_2$ because $I_2$ approximates $I_1$. For this reason, $P_2$, the program returned on input $I_2$, should only differ from $P_1$ 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. Standard program verifiers are unable to verify robustness properties as they are an example of a 2-safety property, a property which requires reasoning about two execution traces simultaneously. Rephrasing the definition, 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. As such, verifying robustness requires reasoning about the relationship between two execution traces.
2 Related Work

2.1 Continuity and Lipschitz Robustness

**Continuity:** Robustness in the setting of numerical perturbations is realized by the property of continuity. Hamlet [7] considered the concept of program continuity, but declared that automating verification of continuity for programs with loops was infeasible. However, Chaudhuri et al. [8] was able to establish a program logic which allowed such automatic verification of continuity.

**Lipschitz Robustness:** In many situations, the result of a program should be relatively stable with respect to uncertainty in the input. That is, if the input is perturbed, the output should vary only slightly, relative to the input perturbation. This is exactly the concept of Lipschitz continuity (see Appendix A.1).

Verifying the property of Lipschitz continuity is considered by Chaudhuri et al. [9]. The approach taken is to verify continuity of the program as a whole using [9] and then verify that each control flow path of the program is piecewise Lipschitz continuous by showing that it is piecewise linear. The continuity verification is performed using a program logic for reasoning about continuity [8]. In order to establish that each control flow path is piecewise linear, an abstract interpretation is applied in which an abstract state, referred to as a robustness matrix, is propagated to determine each rate of change $\frac{\partial x_{\text{out}}}{\partial x_{\text{in}}}$. However, the approach used in [8, 9] is limited to numerical perturbations, and it is not clear how to adapt them to structural changes in the input.

Additionally, Samanta et al. [10] and Henzinger et al. [11] investigate the use of Lipschitz continuity for proving robustness in the context of transducers.

2.2 $k$-Safety Properties

**2-Safety Properties:** In [6], Terauchi & Aiken introduced the term 2-safety property in the study of secure information flow. A general approach to verifying 2-safety properties is the creation of a product program which is then provided as input to a standard program verifier [12]. The product $P_1 \circ P_2$ of two programs $P_1, P_2$ is a program which is equivalent to simultaneously executing both $P_1$ and $P_2$. Because a single program is created with distinct variables corresponding to each variable of $P_1$ and $P_2$, relational properties between $P_1$ and $P_2$ can be expressed as a standard verification property of the single program $P_1 \circ P_2$. As a result, a 2-safety property about $P$, such as robustness, can be established by providing $P \circ P$ to any standard program verifier.

In [13], Barthe et al. analyzes different notions of product programs, as well as various relational program logics and relationships between them.

**$k$-Safety Properties:** Clarkson & Schneider [14] introduced $k$-safety properties as a generalization of these ideas. Sousa & Dillig [15] developed a program logic based on product programs to reason about $k$-safety properties. Their approach is more efficient than using product programs as they avoid creation of an actual product program, while still being able to reason about properties of such a
program. Moreover, rather than considering a single product program, their approach considers the equivalence class of programs which are equivalent to the product program $P_1 \circ \cdots \circ P_k$ and attempts to find an element of this equivalence class which is particularly easy to reason about.

However, the result of these improvements is that their method is not compatible with a standard program verifier. Sousa & Dillig developed an implementation of their algorithm, though it is limited to the verification of Java comparators. Discussion with the authors indicated that many structural invariants could be verified with this framework, but that the existing implementation is heavily specialized and would require new features in order to handle objects like arrays or trees.

3 Example Robustness Properties

The uncertainty with respect to which a program is robust can take many different forms. Some common perturbations include numerical perturbations [11, 8, 9] and permutations of arrays and matrices. As the case of continuous robustness is handled in [8, 9], the focus will be discrete perturbations. In the context of discrete perturbations, it makes sense to consider perturbations of the input which leave the output invariant, as this situation is more common than in the case of continuous perturbations.

3.1 Integer Perturbations

A first example of uncertainty under a discrete perturbation is the presence of noise in a program operating on integer arrays. However, because Lipschitz robustness with respect to $\mathbb{R}$ implies Lipschitz robustness with respect to $\mathbb{Z}$, this problem is a special case of the approach in [8, 9]. As an example, sorting algorithms are 1-Lipschitz robust by [8, 9]. If each element of the input array is perturbed by at most 1, then each element of the output array will be perturbed by at most 1.

3.2 Permutations

Sorting Sorting gives an example of a procedure which is robust with respect to permuting the input. Even in the face of uncertainty regarding the order of the input array, the result of procedure will not be altered. The $\text{max}$ function is a special case of this invariance of sorting algorithms under permutation, which is the root of the failures to synthesize the $\text{max}$ function in Section 1.

Searching Consider the function $\text{Find}(a, x)$ which returns the index of the element $x$ in the array $a$ or $-1$ if $x$ is not an element of $a$. Let $\sigma$ be a permutation and suppose that $a[i]=x$. As discussed in Appendix A.2, we have $\sigma a[\sigma(i)] = a[i] = x$, so that $\text{Find}(\sigma a, x) = \sigma(i)$. Thus, $\text{Find}$ is robust in that perturbing the input array by a permutation $\sigma$ perturbs the output of $\text{Find}$ by the same permutation.
Adjacency Matrices The effect of permuting the rows and columns of an adjacency matrix by the same permutation is simply a relabelling of the vertices. This leads to the following two robustness properties. If the program computes a property of the graph, such as the existence of a Hamiltonian cycle, the program should be invariant under such a relabelling. If the program computes a result which depends on the labelling, such as finding an explicit Hamiltonian cycle, the program should satisfy the that the output will be perturbed by the same permutation as the input.

4 Invariance under permuting lists

As discussed above, many programs are invariant under the reordering of a list. Results from group theory suggest a powerful reduction from checking invariance of a list under all permutations to invariance under a set of just two permutations. Some background is discussed in Appendix A.2. In the languages discussed there, we want to verify that

\[ P(\sigma a) = P(a) \]

for every \( \sigma \in S_n \).

For any \( n \), define

\[
\alpha = 0 1 2 \ldots n-2 n-1 \\
1 2 3 \ldots n-1 0
\]

\[
\beta = 0 1 2 3 \ldots n-1 \\
1 0 2 3 \ldots n-1
\]

Lemma 1. Every permutation is a composition of some sequence of \( \alpha \) and \( \beta \).

Proof. This is a standard exercise in courses on group theory, where it is expressed by saying “\( \{\alpha, \beta\} \) generates the group \( S_n \)”. See, for example, [Exercises 3-4, §3.5, [16]].

Thus, each \( \sigma \in S_n \) is of the form \( \sigma = u_1 \circ u_2 \circ \ldots \circ u_m \) with \( u_i \in \{\alpha, \beta\} \). So if we know that \( P(\alpha s) = P(s) \) and \( P(\beta s) = P(s) \), then

\[ P(\sigma s) = P((u_1 \circ u_2 \circ \ldots \circ u_m) s) = P(u_2 \circ \ldots \circ u_m) s = \ldots = P(u_m s) = P(s) \]

4.1 Application to programs

Let \( F \) and \( G \) be the formulas

\[
F : a_1[\text{length}(a_1) - 1] = a_2[0] \land \forall i.0 \leq i < \text{length}(a_1) - 1 \rightarrow a_1[i] = a_2[i + 1]
\]

\[
G : a_1[0] = a_2[1] \land a_1[1] = a_2[0] \land \forall i.(2 \leq i < \text{length}(a_1) \rightarrow a_1[i] = a_2[i])
\]

\( F \) expresses that \( a_2 = \alpha a_1 \) and \( G \) expresses that \( a_2 = \beta a_1 \). Thus, lemma 1 tells us that invariance of a program with respect to permuting its input can be expressed in the language of [15] as

\[ \| \text{length}(a_1) = \text{length}(a_2) \land (F \lor G) \| \ P(a) \ | \ ret_1 = ret_2 \|. \]
4.2 Automata

In this subsection, we show that this reduction is especially powerful for analyzing deterministic finite automata.

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 provide any reasonable way of checking that a given finite state machine is invariant under any permutation of the input string. Instead, we will construct finite state transducers $T_\alpha$ and $T_\beta$ such that for any $n$ and any string $s$ with $|s| = n$, we have $T_\alpha(s) = \alpha s$ and $T_\beta(s) = \beta s$. Then, determining whether a FSM $M$ is invariant under permutation of its inputs is equivalent to determining whether

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

**Theorem 1.** There is an algorithm for determining whether a deterministic finite state machine $M$ is invariant under permuting it input. The algorithm runs in time $O(m|\Sigma|^2 \log(m|\Sigma|))$, where $m$ is the number of states of $M$ and $\Sigma$ is the alphabet of the machine $M$.

As discussed, it suffices to construct $T_\alpha$ and $T_\beta$ and check language equality.

We first construct $T_\alpha$. For each symbol $a \in \Sigma$, $T_\alpha$ has a transition from its start state $s_0$ to $s_a$, while reading input $a$ and writing output $\epsilon$. For each $a \in \Sigma$ and each $b \in \Sigma$ with $b \neq \$$, 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 $T_\beta$. For each symbol $a \in \Sigma$, $T_\beta$ has a transition from its start state $s_0$ to $s_a$, while reading input $a$ and writing output $\epsilon$. 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 $a$.

![Automata Diagram](image)

(a) The automata $T_\alpha$

(b) The automata $T_\beta$

Fig. 1: Examples of our transducers when $\Sigma = \{a, b, c\}$

The automata $T_\alpha$ and $T_\beta$ each have $O(|\Sigma|)$ states. Now, as discussed in Appendix A.2, the automata $M \circ T_\alpha$ and $M \circ T_\beta$ have $O(m|\Sigma|)$ states. Determining language equality can then be done by minimizing the number of
states of each of these FSMs, because minimal FSMs are unique [17]. This can be done, for example, by Hopcroft’s algorithm [17], yielding a total runtime of $O(m|\Sigma|^2 \log(m|\Sigma|))$.

5 Invariance under permuting binary search trees

In the previous section, 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. trees representing equivalent ordered lists).

We define binary search trees recursively as either being null, or having exactly two children which are binary search trees. Let list be the function from a binary search trees to its corresponding list, i.e. the function giving the in-order traversal of the tree’s nodes.

Define two (partial) operations $\rho$ and $\theta$ on binary search trees as in Figure 2.

![Fig. 2: The tree operations $\rho$ and $\theta$](image)

One can verify that $\rho$ and $\theta$ preserve list($t$) by observing that the subtrees in the above diagrams remain in the same order before and after applying the transformations. We will show in the next theorem that $\rho$ and $\theta$ suffice to generate all transformations of one tree into any other tree with the same underlying list.

**Theorem 2.** If a program $P$ on binary search trees is invariant under $\rho$ and $\theta$, then whenever list($t_1$) = list($t_2$), we have $P(t_1) = P(t_2)$.

**Proof.** Let $t_1$ and $t_2$ satisfy list($t_1$) = list($t_2$). Observe that if $P$ is invariant under $\rho$ and $\theta$, then $P$ is also invariant under $\rho^{-1}$ and $\theta^{-1}$. Thus, it suffices to show that $t_1$ and $t_2$ can both be transformed via $\rho$, $\theta$, $\rho^{-1}$, and $\theta^{-1}$ into another tree $t_3$, because then we can transform $t_1$ into $t_3$, then apply the inverse transformation of $t_2 \rightarrow t_3$. We proceed by providing an algorithm for transforming to a $t_3$ in the following form, as demonstrated in figure 3(a):
Definition 1. A tree \( t \) is in degenerate list form if either

1. \( t \) is null, or
2. \( t \) has a null left child, and \( t \)'s right child is in degenerate list form.

It is clear that if \( t_1 \) and \( t_2 \) are in degenerate list form, then \( \text{list}(t_1) = \text{list}(t_2) \) if and only if \( t_1 = t_2 \). Thus, if we can show that \( t_1 \) and \( t_2 \) can both be transformed into degenerate list form, say \( t'_1 \) and \( t'_2 \), then we will have \( t'_1 = t'_2 \) and we will be done.

Consider the following algorithm for adjusting a tree via \( \rho \) and \( \theta \), where invariants and assertions are written inside \{braces\}:

1: function Listify\((t)\)
2: while \( t \) has a right child do
3: apply \( \rho^{-1} \) to \( t \)
4: \{\( t \) has a null right child\}
5: while \( t \) has a left child do \( \triangleright \{ t \)'s right child is in degenerate list form\}
6: while \( t \)'s left child has a right child do
7: apply \( \theta \) to \( t \)
8: \{\( t \)'s left child has a null right child\}
9: apply \( \rho \) to \( t \)
10: \{\( t \) is in degenerate list form\}

To verify that this algorithm terminates with \( t \) in degenerate list form, we need only verify that each of the assertions hold and that the loop invariant is inductive.

The invariant on line 4 follows from the negation of the condition of the loop before it. The loop invariant holds from line 4 simply because the null tree is in degenerate list form. The invariant on line 8 also follows from the negation of the condition of the loop before it.

Next we show that the loop invariant is inductive. The inner loop on line 6 does not change \( t \)'s right child, so it remain in degenerate list form. Thus, when we apply \( \rho \) to \( t \), its left child has a null right child, and \( t \)'s right child is in degenerate list form. As shown in figure 3(b), this leave \( \rho(t) \)'s right subtree in degenerate list form. Thus, the invariant is inductive.

\[
\begin{align*}
\text{(a) A tree in degenerate list form} & & \text{(b) Applying } \rho \text{ to a tree as in line 9} \\
\end{align*}
\]

Fig. 3: Tree operations. The symbol \( \emptyset \) represents a null tree.

Finally, we see that our conclusion on line 10 follows from the loop invariant and the negation of the condition, by the definition of degenerate list form.
6 Class 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, especially if the programmer is not formally trained. Here is one type of invariant that can be expressed through code:

**Definition 2.** We say that a program $P$ is class invariant with respect to a function $f$ (or that $P$ is $f$-class invariant) if whenever $f(x) = f(y)$, we have $P(x) = P(y)$.

Invariance under permutations of lists and binary search trees are special cases of this definition. For lists, the function $f$ is the sort, while for binary search trees, $f$ can be taken to be the list function, which returns the underlying list in sorted order. Nearly every function written on a tree or heap should be invariant with respect to a list function, as these data types are meant to represent the lists faithfully (while allowing for faster algorithms).

We regard this problem as very difficult, because $f$ can be an arbitrary program and thus encode more complicated properties than first order logic. One approach comes from the following observation:

**Lemma 2.** Let $P : S \to Z$ and $f : S \to T$. Then $P$ is $f$-class invariant if and only if there exists a program $\tilde{P} : T \to Z$ such that $P = \tilde{P} \circ f$

\[ S \xrightarrow{f} T \xrightarrow{\tilde{P}} Z \]


Proof. If $P$ is $f$-class invariant, then the function

\[ \tilde{P}(t) = \begin{cases} P(s), & t = f(s) \text{ for some } s \in S \\ z_0, & \text{otherwise} \end{cases} \]

for any $z_0 \in Z$ is well defined and satisfies $P = \tilde{P} \circ f$.

Conversely, if $\tilde{P}$ exists, then whenever $f(s) = f(s')$, we have $P(s) = \tilde{P}(f(s)) = \tilde{P}(f(s')) = P(s')$, so $P$ is class invariant with respect to $f$.

Now, the key observation is that $P$ and $f$ together give a full functional specification for $\tilde{P}$. We want to either construct such a $\tilde{P}$ in order to prove our property, or provide the programmer with a counterexample to the property in order to help them debug. Thus, a counterexample guided synthesis loop, similar to that present in [18], seems like a natural candidate for this problem. We build up more examples $E \subseteq S$, and try to synthesize $\tilde{P}$ using $E$ or find a counterexample among $E$:

It turns out this algorithm is sound, assuming a sound equivalence-of-programs verifier. If we assume a sound and complete synthesizer (meaning that there is always some set of examples that will result in a successful synthesis) and verifier, the algorithm is relatively complete. However, we need to assume one additional condition on the equivalence-of-programs verifier. Namely, we suppose there is some total order on the elements in $S$, and that if $\tilde{P} \circ f$ is not equivalent to $P$, the verifier outputs the smallest counterexample with respect to this ordering.
Input $P$ and $f$. Let $E = \{\}$. Synthesize a program $\tilde{P} : T \rightarrow Z$ given examples $\{(f(s), P(s)) | s \in E\}$. Verify whether $\tilde{P} \circ f$ is equivalent to $P$. If no $\tilde{P}$ exists, then some pair $s, s_0$ exist such that $f(s) = f(s_0)$ yet $P(s) \neq P(s_0)$. Because the verifier outputs counterexamples in some order, $s$ and $s_0$ will eventually be found, and the algorithm will output “YES”.

Fig. 4: Decision procedure for verifying class invariance with respect to $f$

**Theorem 3.** The algorithm in Figure 4 is sound and relatively complete, relative to a perfect synthesizer and a perfect equivalence-of-programs verifier.

**Proof.** If $\tilde{P}$ exists, then a perfect synthesizer will eventually synthesize $\tilde{P}$ given some set of examples, say $E$. Because the verifier gives back counterexamples in a fixed order, a superset of $E$ will eventually be fed into the synthesizer, resulting in $\tilde{P}$. Then the idealized verifier will output “YES”.

If no $\tilde{P}$ exists, then some pair $s, s_0$ exist such that $f(s) = f(s_0)$ yet $P(s) \neq P(s_0)$. Because the verifier outputs counterexamples in some order, $s$ and $s_0$ will eventually be found, and the algorithm will output “YES”.

7 Conclusion

While PBE suffers from some amount of uncertainty in a user’s intent, synthesizing a robust program can mitigate some of this uncertainty. To this end, we explored various approaches of verifying the robustness of a program, as well as different robustness properties a program can possess. We discovered a reduction for the problem of verifying permutation invariance in lists and binary search trees, and proposed a counterexample-guided synthesis approach to verifying more general invariance properties.

Directions for future research include implementing methods for verifying robustness in order to determine their usability as part of a synthesis procedure. Specifically, we would like to implement the product program approach of [12], the Cartesian Hoare Logic method of [15], and our approach of synthesizing a program $\tilde{P}$ to verify invariance.
A Appendix: Background

A.1 Lipschitz Robustness

Lipschitz continuity of a mathematical function \( f \) is the property that there exists a real number \( K \in \mathbb{R}^+ \) so that for all \( x_1, x_2 \), \( d(f(x_1), f(x_2)) \leq K d(x_1, x_2) \).

Similarly, a program \( P \) is robust at a state \( \sigma \) with respect to the input variable \( x_{\text{in}} \) and output variable \( x_{\text{out}} \) if for all \( \varepsilon \in \mathbb{R}^+ \), whenever \( \sigma' \) satisfies \( d(\sigma(x_{\text{in}}), \sigma'(x_{\text{in}})) < \varepsilon \) and for all \( y \neq x_{\text{in}} \), \( \sigma(y) = \sigma'(y) \), we have \( d(P(\sigma)(x_{\text{out}}), P(\sigma')(x_{\text{out}})) < K \varepsilon \), where \( K \) is a real-valued function of the size of \( x_{\text{in}} \) [9].

A.2 Permutations

A permutation is a bijection from a set \( \Omega \) to itself [16]. When \( \Omega \) is a finite set, a permutation can be specified using two-line notation by placing the elements of \( \Omega \) on one line, and their images under \( \sigma \) beneath them:

\[
\begin{array}{cccccc}
\sigma: & 0 & 1 & 2 & 3 & 4 & 5 \\
& 5 & 2 & 3 & 1 & 4 & 0 \\
\tau: & a & b & c \\
& a & c & b \\
\end{array}
\]

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 \( \sigma \) above is encoded in an array as \( \sigma = [5, 2, 3, 1, 4, 0] \), so that \( \sigma(i) = \sigma[a[i]] \). The inverse of a permutation is defined as for general inverse functions, so \( \sigma^{-1} \) sends \( \sigma(i) \) to \( i \).

**Action on an Array:** Given a permutation \( \sigma \), \( \sigma \) can act on an array or list \( a \) by permuting the entries of \( a \). For example, let \( a = [37, 25, 19, 49, 81, 21] \) and \( \sigma \) be as above. Then \( \sigma(3) = \sigma[3] = 1 \) and applying \( \sigma \) to \( a \) results in the element of \( a \) at index 3, 49, being moved to index 1 in \( \sigma a \). That is, \( \sigma a[1] = 49 \) and \( a[3] = \sigma a[1] = \sigma a[\sigma[3]] \). More generally,

\[
\sigma a[\sigma[i]] = a[i] \quad \iff \quad \sigma a[i] = a[\sigma^{-1}[i]] \quad (1)
\]

A.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 a 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 a FST).

Recall that for any FSMS \( A \) and \( B \), the problem of determining whether \( L(A) = L(B) \) is decidable. We can compose a FST \( T \) and a FSM \( A \), denoted \( A \circ T \), to get another FSM such that \( (A \circ T)(s) = A(T(s)) \). Furthermore, \( A \circ T \) can be constructed with \( |S_A| \cdot |S_T| \) states, where \( S_A \) and \( S_T \) are the states of \( A \) and \( T \). We will assume that all input strings are terminated by a special end-of-input character \$. 

10
References