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 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. 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. In particular, 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

Relational Logics Benton [2] established a Relational Hoare Logic in order to reason about relational properties, properties which consider two, usually distinct, programs. Barthe et al. [3, 4] extended Benton's Relational Hoare Logic to probabalistic programs in order to reason about cryptographic protocols and differential privacy.

Self-Composition Barthe et al. [5] applied self-composition, sequential running of renamed copies of the original program, to study secure information flow. Terauchi & Aiken [6] built on this work by applying a type-based approach to complement self-composition.

2-*Safety Properties* In [6], Terauchi & Aiken introduced the term 2-safety property. A general approach to the verification of 2-safety properties is the creation of a product program [7], a program which interleaves two copies of a given program to create a new program. It [8], 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. [10, 11] consider the continuity and Lipschitz continuity of programs over the real numbers. Samanta et al. [12] and Henzinger et al. [13] 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 [14] and for general programs by [11]. Additionally, the robustness of networked systems was explored by Samanta et al. [15].

k-*Safety Properties* Clarkson and Schneider [16] introduced k-safety properties as a generalization of this idea. Sousa and Dillig [17] formulated a verification algorithm in order to automate checking of k-safety properties.

3 Preliminaries

3.1 Continuity and Lipschitz Continuity

A program $\mathcal{P}$ is continuous at a state σ with respect to input variables In and observable variables Obs if for all $\varepsilon \in \mathbb{R}^+$, there is a $\delta \in \mathbb{R}^+$ so that if σ' satisfies

1. for all $x \in In$, $d\big(\sigma(x), \sigma'(x)\big) < \varepsilon$ and

2. for all $y \notin In$, $\sigma(y) = \sigma'(y)$,

then for all $x \in Obs$, $d\big(\mathcal{P}(\sigma)(x), \mathcal{P}(\sigma')(x)\big)$ [10].

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, $d\big(f(x_1), f(x_2)\big) \leq K\, d(x_1, x_2)$. Similarly, a program $\mathcal{P}$ is robust at a state σ with respect to the input variable x_{in} and output variable x_{out} if for all $\varepsilon \in \mathbb{R}^+$,

1. $d\big(\sigma(x_{in}), \sigma'(x_{in})\big) < \varepsilon$ and
2. for all $y \neq x$, $\sigma(y) = \sigma'(y)$

implies that $d\big(\mathcal{P}(\sigma)(x_{out}), \mathcal{P}(\sigma')(x_{out})\big) < K\varepsilon$, where K is a real-valued function of the size of x_{in} [11].

3.2 Permutations

A permutation is a bijection from a set Ω to itself [18]. 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

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

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 Information on permutation groups can be found in §1.3 of [18].

The permutations on N elements, S_N, can all be expressed as compositions of the permutations [Exercises 3-4, §3.5, 18]

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

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 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 $|S_A||S_T|$ 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 $.

4 Robustness Properties

The uncertainty with respect to which a program is robust can take many different forms. Some common perturbations include

- numerical perturbations [13, 10, 11],
- permutations of arrays and matrices,
- simultaneous permutations of arrays.

4.1 Continuity

Robustness in the setting of numerical perturbations is realized by the property of continuity. In many situations, the result of a program should be relatively stable with respect to any 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. if it is possible that the input has been slightly perturbed, the result of the program should not be drastically different. As an example, sorting algorithms are 1-Lipschitz. If each element of the input array is perturbed by an amount no more than 1, then each element of the output array will be changed by no more than 1.

4.2 Permutations

Let F and G be the formulas

$$F : \forall i. 0 \leq i < \operatorname{length}(a_1) \rightarrow a_1[i] = a_2[(i+1)\% \operatorname{length}(a_1)],$$
$$G : a_1[0] = a_2[1] \wedge a_1[1] = a_2[0] \wedge \forall i. \big(2 \leq i < \operatorname{length}(a_1) \rightarrow a_1[i] = a_2[i]\big).$$

Then invariace of a program which takes an array as input can be specified by

$$\| \operatorname{length}(a_1) = \operatorname{length}(a_2) \wedge (F \vee G) \| \mathcal{P}(a) \| \operatorname{ret}_1 = \operatorname{ret}_2 \|.$$

Sorting Sorting is one 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 `max` function is a special case of this invariance of sorting algorithms under permutation, which is the root of the incorrectness of the attempts to synthesize the `max` function in Section 1.

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)

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

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 The effect of permuting the rows and columns of an adjacency matrix by the same permutation is simply a relabelling of the vertices. As the labelling of the vertices is arbitrary, a program which takes an adjacency matrix as input should likely satisfy one of 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 same property as in the case of searching, that the output should be perturbed by the same permutation as the input.

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. This property can be expressed using ideas from Section 3.2 by applying a transposition and a rotation simultaneously to each array.

```
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
```

5 Verifying Robustness

5.1 Lipschitz Continuity

Verifying the property of Lipschitz continuity is considered by Chaudhuri et al. [10, 11]. The approach taken is to verify continuity of the program as a whole 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 [10]. 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 used to determine each rate of change $\partial x_i^{out}/\partial x_j^{in}$. One advantage of this approach, similar to other relational program logics, is that branches of `if` blocks can be reasoned about independently and then it is verified that the branches are compatible. As this compatibility is transitive, this prevents have to make a pairwise comparison of each branch as in approaches based on product programs. However, the approach used in [10, 11] is limited to numerical perturbations.

5.2 Product Programs

A more general approach to verifying robustness properties is the creation of a product program which is then provided as input to a standard program verifier [7]. The product $\mathcal{P}_1 \circledast \mathcal{P}_2$ of two programs $\mathcal{P}_1, \mathcal{P}_2$ is a program which is equivalent to simultaneously executing both $\mathcal{P}_1$ and $\mathcal{P}_2$. Because a single program is created with distinct variables corresponding to each variable of $\mathcal{P}_1$ and $\mathcal{P}_2$, relational properties between $\mathcal{P}_1$ and $\mathcal{P}_2$ can be expressed as a standard verification property of the single program $\mathcal{P}_1 \circledast \mathcal{P}_2$. As a result, a 2-safety property about $\mathcal{P}$, such as robustness, can be established by providing $\mathcal{P} \circledast \mathcal{P}$ to any standard program verifier.

5.3 Cartesian Hoare Logic

After discussing with the authors of [17], it seems that with a lot of time and effort, their implementation of Descartes could be used to verify many robustness properties.

6 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.

6.1 Automata

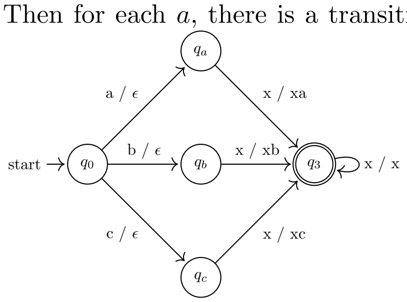
Given a program P which takes a linear data type as input. We may wish to verify that this program is invariant under reordering of the array, in other words, it is invariant under the action of S_n. Denote the output of P on input s by $P(s)$. Then we want to test whether $P(\sigma s) = P$ for all $\sigma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

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 $P((u_1 u_2 \dots u_m)s) = P((u_2 \dots u_m)s) = \dots = P(u_m s) = P(s)$.

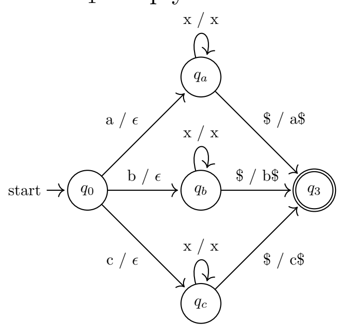
For a fixed n and $\sigma \in S_n$, we can construct a finite state transducer T such that $T(s) = \sigma s$ for $|s| = n$. However, this does not give us any reasonable way of checking that a given finite state machine is invariant under permutation of the input string. Instead, we will construct finite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

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

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.



6.2 Programs

The previous section indicates a method for verifying robustness of more general programs under permutation of inputs. Given a procedure F taking an array as argument, consider the functions F_α, which first swaps the first two elements of the array, then computes F, and F_β, which moves the first element of the array to the back, then computes F. Then F is invariant under permutation of its input if and only if F is functionally equivalent to F_α and F_β.

Certainly this method is more efficient than checking invariance under a larger class of permutations. For DFAs, this simplification made the problem of permutation invariance solvable. This indicates that this simplification may aid in our analysis in a profound way. For example, this may be a much easier way of verifying this invariance compared to expressing permutations as a general 2-safety property.

7 Invariance under permuting binary search trees

In the previous section 6, 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).

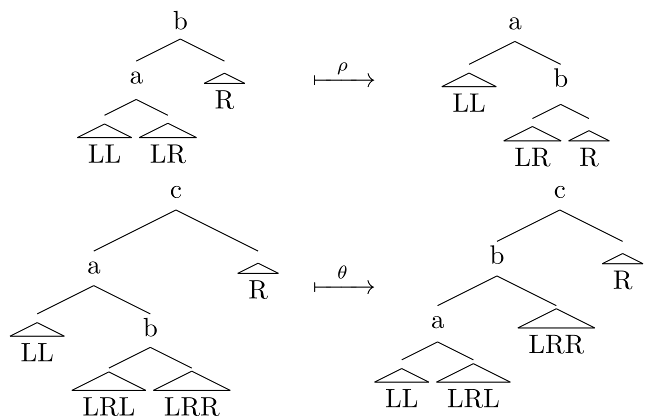
Represent binary search trees by the algebraic data type Then define two (partial) operations ρ and θ on binary search trees as follows:

```
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)
```



You can verify that ρ and θ 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 ρ and θ additionally suffice to generate all transformations of any tree into another tree with an equal underlying list.

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

Proof. Let t_1 and t_2 satisfy $\mathrm{list}(t_1) = \mathrm{list}(t_2)$. Observe that if P is invariant under ρ and θ, then P is also invariant under ρ^{-1} and θ^{-1}. Thus, it suffices to show that t_1 and t_2 can both be transformed via ρ and θ into another tree t_3, because then we can transform t_1 into t_3, then apply the inverse transformation to get to t_2. We proceed by providing an algorithm for transforming to a t_3 in the following form:

Definition 1. *A tree t is in degenerate list form if either*

1. *t is Nil, or*
2. *t has a Nil left child, and t's right child is in degenerate list form.*

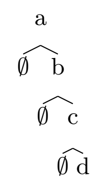


Fig. 1. A tree in degenerate list form

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 ρ and θ, where invariants and assertions are written inside {braces}:

```
 1: function LISTIFY(t)
 2:     while t has a right child do
 3:         apply ρ^{-1} to t
 4:     {t has a null right child}
 5:     while t has a left child do           ▷ {t's right child is in degenerate list form}
 6:         while t's left child has a right child do
 7:             apply θ to t
 8:         {t's left child has a null right child}
 9:         apply ρ 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 ρ to t, it's left child has a null right child, and t's right child is in degenerate list form. As shown in figure 2, this leave $\rho(t)$'s right subtree in degenerate list form. Thus, the invariant is inductive.

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.

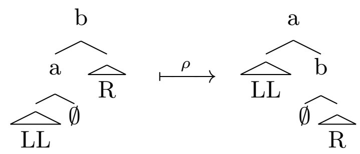


Fig. 2. Applying ρ to a tree as in line 9

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

For example, if a programmer defines a tree or heap data type with a function list mapping the data type to lists, then nearly every function written on this data type should be invariant with respect to the list function, as these data types are meant to represent the lists perfectly (while allowing for faster algorithms).

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

Lemma 1. *Let $P : S \to T$ and $f : T \to Z$. Then P is invariant with respect to f if and only if there exists a program $\widetilde{P} : T \to Z$ such that $P = \widetilde{P} \circ f$*

$$\begin{tikzcd} S \arrow[rr, "P"] \arrow[dr, "f"'] & & Z \\ & T \arrow[ur, dashed, "\widetilde{P}"'] & \end{tikzcd}$$

Proof. If P is invariant with respect to f, then the function

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

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

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

Now, the key observation is that P and f together give a full functional specification for $\widetilde{P}$. We want to either construct such a $\widetilde{P}$ 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 [19], seems like a natural candidate for this problem.

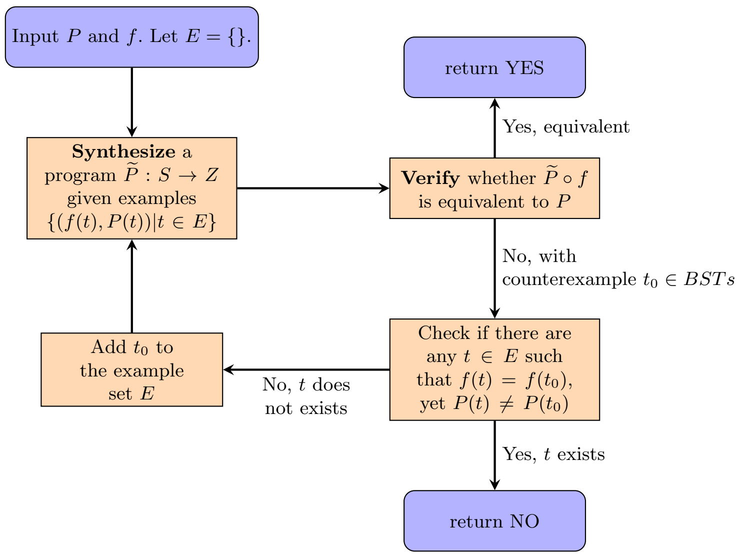


Fig. 3. Decision procedure for verifying invariance with respect to a function

Theorem 2. *The algorithm 3 is sound and relatively complete, relative to a perfect synthesizer and a perfect equivalence-of-programs verifier.*

Proof. In order to prove this formally, we need an even stronger condition on the equivalence-of-programs verifier. Namely, we suppose there is some total order on the elements in T, and that if $\widetilde{P} \circ f$ is not equivalent to P, the verifier outputs the smallest counterexample with respect to this ordering.

If $\widetilde{P}$ exists, then a perfect synthesizer will eventually synthesize $\widetilde{P}$ 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 $\widetilde{P}$. Then the idealized verifier will output "YES".

If no $\widetilde{P}$ exists, then some pair t, t_0 exist such that $f(t) = f(t_0)$ yet $P(t) \neq P(t_0)$. Because the verifier outputs counterexamples in some order, t and t_0 will eventually be found, and the algorithm will output "YES".

References

1. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and robustness of programs. Commun. ACM **55**(8) (August 2012) 107–115

2. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '04, New York, NY, USA, ACM (2004) 14–25
3. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '09, New York, NY, USA, ACM (2009) 90–101
4. Barthe, G., Köpf, B., Olmedo, F., Zanella-Béguelin, S.: Probabilistic relational reasoning for differential privacy. ACM Trans. Program. Lang. Syst. **35**(3) (November 2013) 9:1–9:49
5. Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: Proceedings of the Computer Security Foundations Workshop. Volume 17. (2004) 100–114
6. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In Hankin, C., Siveroni, I., eds.: Static Analysis: 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005. Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2005) 352–367
7. Barthe, G., Crespo, J.M., Kunz, C.: Product programs and relational program logics. Journal of Logical and Algebraic Methods in Programming **85**(5, Part 2) (2016) 847 – 859 Articles dedicated to Prof. J. N. Oliveira on the occasion of his 60th birthday.
8. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In Butler, M., Schulte, W., eds.: FM 2011: Formal Methods: 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2011) 200–214
9. Hamlet, D.: Continuity in software systems. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA '02, New York, NY, USA, ACM (2002) 196–200
10. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity analysis of programs. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '10, New York, NY, USA, ACM (2010) 57–70
11. Chaudhuri, S., Gulwani, S., Lublinerman, R., Navidpour, S.: Proving programs robust. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ESEC/FSE '11, New York, NY, USA, ACM (2011) 102–112
12. Samanta, R., Deshmukh, J.V., Chaudhuri, S.: Robustness analysis of string transducers. In Van Hung, D., Ogawa, M., eds.: Automated Technology for Verification and Analysis: 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings, Cham, Springer International Publishing (2013) 427–441
13. Henzinger, T.A., Otop, J., Samanta, R.: Lipschitz Robustness of Finite-state Transducers. In Raman, V., Suresh, S.P., eds.: 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014). Volume 29 of Leibniz International Proceedings in Informatics (LIPIcs)., Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2014) 431–443
14. Majumdar, R., Saha, I.: Symbolic robustness analysis. In: 2009 30th IEEE Real-Time Systems Symposium. (Dec 2009) 355–363

15. Samanta, R., Deshmukh, J.V., Chaudhuri, S.: Robustness analysis of networked systems. In Giacobazzi, R., Berdine, J., Mastroeni, I., eds.: Verification, Model Checking, and Abstract Interpretation: 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2013) 229–247
16. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: 2008 21st IEEE Computer Security Foundations Symposium. (June 2008) 51–65
17. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '16, New York, NY, USA, ACM (2016) 57–69
18. Dummit, D.S., Foote, R.M.: Abstract Algebra. 3rd edn. John Wiley & Sons (2004)
19. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XII, New York, NY, USA, ACM (2006) 404–415