Verifying Robustness of Programs Under Algebraic Perturbations

Jacob Bond and Clay Thomas

Purdue University

1 Introduction

It is desirable for programs to respond smoothly to changes in input, in the sense that if a small change is made to the input, the change in output is also small. Many classical algorithms are indeed robust with respect to continuity [1]. In the setting of PBE, a user would likely prefer, and expect, a program synthesizer to return a robust program. For this reason, the ability to efficiently verify robustness would be useful in the determination of which program the synthesizer should return.

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.

Finally, if a sufficiently efficient method for verifying program robustness is developed, it could be applied to a program synthesizer in order to verify directly that the synthesizer is indeed robust.

1.1 Motivating Example

Consider an attempt to specify the max function by providing to the synthesizer $\Big((13, 15), 15\Big)$, $\Big((-23, 19), 19\Big)$, and $\Big((-75, -13), -13\Big)$. In order to synthesize a simpler program, the result would likely be a program which returns the second argument. However, if instructed to return a program which is invariant under permuting the arguments, this would no longer be a viable program, and it is likely that the synthesizer would return a function for finding the max of two elements.

Moreover, by reversing the inputs from the input-output pairs given above, the synthesizer would now be likely to construct a program which returns the first input. In this way, the synthesizer is sensitive to small changes in input, and would not be robust itself.

1.2 Problem Definition

Our goal is to formulate robustness conditions inside of a first-order theory, in order to utilize an SMT solver in checking robustness. In particular, we will

explore various robustness properties for arrays of integers, as well as automated methods for checking these properties with an SMT solver. We will perform an experiment to determine the improvements that result from a synthesis engine being able to reason about robustness properties.

2 Robustness Properties

Robustness of a program is the property that the program behaves predictably on uncertain inputs [2]. In our context, the uncertainty derives from the fact that the user's input only specifies a very small number of potential arguments to the desired function and that the returned program should not change drastically as a result of specifying input-output pairs which are slightly perturbed.

Some settings in which programs should be robust:

- numerical perturbations [3, 4, 1],
- permutations of arrays,
- simultaneous permutations of arrays,
- relabeling of inputs [5].

Robustness as a 2-Safety Property A 2-safety property is one which requires reasoning about two execution traces simultaneously. As robustness requires reasoning about uncertain inputs, this can be achieved by considering a given input along with a second input which may be considered to vary over the range of uncertainty. In this way, robustness is a 2-safety property.

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:

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

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,

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

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]

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

4 Examples of Robust Programs

4.1 Continuity

4.2 Permutations

Suppose $\mathcal{P}$ is a program with takes an array as input. Assume further that it is desired that $\mathcal{P}$ be permutation invariant.

Sorting

Searching

Adjacency Matrices

4.3 Further Examples of Robustness Properties

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 amounts that each part is worth. A reordering of the parts of the problem should not affect the points which a student earns. As reordering the parts of the problem corresponds to simultaneously permuting the

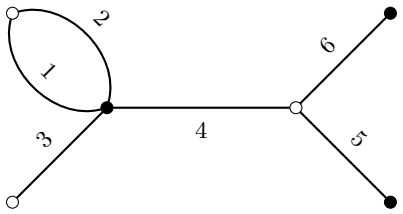
three input arrays, the grading algorithm should be invariant under simultaneous permutations of the input arrays. Using ideas from Section 3.2, this property can be expressed as the conjunction of the following two formulas:

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

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

```
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
```

Relabeling Consider a bipartite graph drawn without edge-crossings. Labeling the edges results in a cyclic ordering of the edges around each vertex, which can then be viewed as a pair of permutations $\sigma_0, \sigma_1 \in S_N$. However, relabeling the edges does not change the structure of the graph, and as such, many computations performed on the graph should be invariant under a relabeling of the graph.



$\sigma_0 = (1\,3\,4\,2)(5)(6)$
$\sigma_1 = (1\,2)(3)(4\,5\,6)$

Fig. 1. A dessin d'enfant and its corresponding pair of permutations

A relabeling of the graph corresponds to simultaneously conjugating the permutations σ_0, σ_1 by and element of S_N. This can be expressed in a first-order theory as follows:

1. Let the property that c conjugates a into b be given by

$$\text{Conj}(a, b, c) : \forall i.\, 0 \leq i < \text{length}(a) \to a[c[i]] = c[b[i]].$$

2. Then being simultaneously conjugate is the property that

$$\text{SimultConj}([x_1, y_1], [x_2, y_2]) : \exists c.\ \text{Perm}(c) \wedge \text{Conj}(x_1, x_2, c) \wedge \text{Conj}(y_1, y_2, c)$$

An example of such an algorithm is to compute a canonical pair of permutations for any equivalence class of graphs.

```
1: function CanonicalPair(s0, s1)
2:     CandidateReps ← []
3:     for 0 ≤ i < N do
4:         Q ← Deque([i]), perm ← [i]
5:         while length(perm) < N do
6:             x ← PopLeft(Q)
7:             for s in (s0, s1) do
8:                 if s(x) not in perm then
9:                     Append(perm, s(x))
10:                    PushRight(Q, s(x))
11:        Append(CandidateReps,[perm^-1*s0*perm, perm^-1*s1*perm])
       return Min(CandidateReps)
```

5 Verifying Robustness Using Cartesian Hoare Logic

5.1 Shortcomings Regarding Lipschitz-Continuity

6 Preliminaries

6.1 The symetric group

The symetric group S_n is the group of bijective functions on the set $\{1, 2, \ldots, n\}$, which we call permutations. We will write elements of S_n using "cycle notation", where $(i_1\ i_2\ \ldots\ i_m)$ represents the permutation that sends i_j to i_{j+1} for $j = 1, \ldots, m-1$ and sends i_m to i_1.

We will often talk about the symmetric group acting on linear data types (namely, arrays, lists, and strings). For any $\sigma \in S_n$ and any array a, this action is given by the formula

$$(\sigma a)[i] = a[\sigma(i)]$$

where we index our arrays starting at 1. This action extends naturally to lists and strings. For an example, the action of the full cycle $(1\ 2\ 3\ \ldots\ n)$ just "rotates" a string by one, e.g. $(1\ 2\ 3\ 4)$"$abcd''$ = "$bcda''$.

6.2 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 \$.

7 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.

7.1 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.

8 Invariance under permuting binary search trees

In the previous section 7, we

```
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
```

9 Invariance with respect to a function

10 Related Work

10.1 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].

10.2 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.

10.3 *k*-safety Properties

The concept of a 2-safety property was introduced by Terauchi and Aiken [11] in the study of secure information flow. Clarkson and Schneider [12] introduced k-safety properties as a generalization of this idea. Finally, Sousa and Dillig [13] formulated a verification algorithm in order to automate checking of k-safety properties.

References

1. 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
2. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and robustness of programs. Commun. ACM **55**(8) (August 2012) 107–115
3. 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
4. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity analysis of programs. POPL '10, New York, NY, USA, ACM (2010) 57–70
5. Zapponi, L.: What is . . . a dessin d'enfant? Notices Amer. Math. Soc. **50**(7) (2003) 788–789
6. Dummit, D.S., Foote, R.M.: Abstract Algebra. 3rd edn. John Wiley & Sons (2004)
7. Majumdar, R., Saha, I.: Symbolic robustness analysis. In: 2009 30th IEEE Real-Time Systems Symposium. (Dec 2009) 355–363
8. Samanta, R., Deshmukh, J.V., Chaudhuri, S. In: Robustness Analysis of Networked Systems. Springer Berlin Heidelberg, Berlin, Heidelberg (2013) 229–247
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. Samanta, R., Deshmukh, J.V., Chaudhuri, S. In: Robustness Analysis of String Transducers. Springer International Publishing, Cham (2013) 427–441
11. Terauchi, T., Aiken, A. In: Secure Information Flow as a Safety Problem. Springer Berlin Heidelberg, Berlin, Heidelberg (2005) 352–367
12. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: 2008 21st IEEE Computer Security Foundations Symposium. (June 2008) 51–65
13. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. PLDI '16, New York, NY, USA, ACM (2016) 57–69