Learning Good Generators for Property-Based Testing in Deductive Program Verifiers

Joseph Cutler

January 12, 2022

1 Motivation

Deductive Program Verifiers like Dafny and Frama-C have proven themselves both useful and powerful tools in the fight against un-verified software. However, their adoption is hindered by the fact that graduate-level expertise in verification is required in order to verify any non-trivial program. One approach to lowering the barrier to entry for deductive verifiers is to incorporate lighter-weight formal methods into the mix. While SMT based verification is very powerful, its nuances lead to much of the difficulty in using verifiers based on it, and so selectively substituting this verification technique with a less precise but easier-to-use formal method would lower the difficulty while retaining the same workflow and some of the theoretical guarantees.

In this report, we investigate the use of *Property-Based Testing* in place of SMT solving as a core for program verification tools. Property-Based Testing (PBT) [2] is a lightweight formal method wherein the specifications for a program are checked by randomly generating thousands of inputs, and ensuring that the program meets its specification by executing it at those inputs. While PBT doesn't give full correctness guarantees like SMT solving does, it brings a level of confidence in a manner similar to Model Checking. Moreover, PBT is fully modular, and so it can be used selectively: randomly-tested code and SMT-verified code can coexist in the same file, enabling users to take a *gradual verification* approach by gradually replacing tested code with verified code.

Further, PBT has significant usability benefits over SMT-based verification. A common complaint while using SMT-based verifiers is that it is often difficult to interpret the SMT output. When an SMT solver fails to verify a specification, it can be for one of two reasons: either (a) the specification is incorrect, or (b) the solver was simply unable to find a proof. These two results are often indistinguishable, as the solver may not return a counter-model indicating that the specification is falsifiable — in this case, the only course of action is to attempt to provide more hints to the solver in the hopes that it will eventually prove the property. With PBT, the only failure mode is through counter-examples, and so no interpretation is required. Moreover, the counter-examples in a PBT approach are more easily interpretable. With SMT, counter-examples are encoded in an SMT data format and are not easily lifted back to program values. With testing, the inputs were concrete to begin with, and so counter-examples can be presented as-is. Finally, PBT can actually aid in the development of SMT-verified programs by allowing the user to test specifications to gain confidence that they are true before spending costly time attempting to develop tricky loop invariants.

In this report, we develop a simple deductive program verifier based on Dafny which is backed by PBT instead of SMT. In Section 2, we show how PBT can be used in place of an SMT solver to do deductive verification. This process is conceptually simple, but we will see that it hides a major challenge: generating thousands of independent random input sets which satisfy a function's precondition is NP-hard. In fact, tackling a small variant of problem is the bulk of the content of this report, and its main technical contribution. We give a sketch of this contribution in this section. In brief, our technique first infers a set of “candidate" example-generators from the precondition in question, and then uses a reinforcement-learning-based online learning algorithm to find the best generator among the set.

In Section 3, we introduce a subset of a domain-specific language called `ALuck` based on `Luck` [5], whose programs are generators. We discuss the semantics of these generators, and give some examples of good and bad generators.

In Section 4, we present our algorithm for inferring candidate generators for a precondition. In Section 5, we discuss the online learning technique we employ to discover the best generator among our candidates. Finally, in Section 6, we discuss our implementation of this algorithm, present benchmarks, and discuss future work.

2 PBT, the Generator Problem, and our Approach

At the core of all deductive program verifiers is a decision procedure for Hoare triples $\{\phi\}\,C\,\{\psi\}$. Such a triple is valid when for all heaps σ satisfying ϕ, we have that ψ holds of the heap which results after C is run on σ. In symbols:

$$\forall \sigma . \phi(\sigma) \implies \psi(C(\sigma)) \tag{$*$}$$

Figure 1: Syntax of Preconditions

$$\begin{array}{llll} \text{Expressions} & e & ::= & x \mid e+e \mid -e \mid e*e \\ \text{Relations} & R & ::= & = \mid \leq \mid < \\ \text{Propositions} & P & ::= & P \wedge P \mid eRe \end{array}$$

With traditional SMT-based verifiers, this is decided by compting the weakest precondition of ψ with respect to C, $\mathtt{wp}(C,\psi)$ and using SMT to prove that ϕ implies it. With property-based random testing, we can skip the weakest precondition computation and directly test $(*)$. PBT tests formulae like $\forall x.P(x)$ by randomly generating values a (of a specified type), and then evaluating the concrete boolean expression $P(a)$. When the property P is an implication $Q \implies R$, as is the case for $(*)$, this is not very effective. Intuitively, we'd like to test that R holds for a bunch of examples satisfying Q, but what actually ends up happening is that the vast majority of our random examples satisfy $Q \implies R$ simply because they don't satisfy $Q(x)$. To fix this, PBT relies on *user-written* generators which generate values *satisfying* Q, and then checking that R holds.

In terms of $(*)$, this means that a user of a verifier backed by PBT would have to provide programs which generate random heaps σ to satisfy the preconditions of every function they plan to test. This is labor-intensive, error prone, and diminishes the benefits of PBT over SMT-based verification. To make the former practical, we need to *automatically* create generators from the “source code" of a specification. Unfortunately, this problem is NP-Hard in even simple cases. If we limit ourselves to preconditions ϕ which consist only of arithmetic constraints (which we will in fact do), the problem of generating hundreds of sets of inputs which satisfy ϕ is as hard as generating *one* set of inputs, which is equivalent to SMT solving.

In the rest of this report, we will focus in on this generation problem, and present a solution to a specific instance of it. The main cause of the generator problem described above is that propositions are *declarative*: they simply state what must hold in order for them to be satisfied, and provide no instructions on how to generate values which satisfy them. In classical PBT, the user is required to write *generators* which are programs that emit random values satisfying a property. Ideally, we would like to automatically translate declarative propositions into generators, but as stated earlier, this is arbitrarily hard. Instead, the core of our approach is a translation of propositions into a large set of *potentially failing* generators. By relaxing the problem from inferring one *perfect* generator to a few carefully-chosen generators of unknown (but hopefully high) quality, we make the problem tractable. Empirically, it seems that among a large enough set of these automatically-created generator candidates, there is often one which approximates a generator which a human would have written for the same proposition. With this in mind, we can then run an online learning algorithm to learn which among the candidates is the best, while simultaneously generating a sequence of random inputs for testing purposes.

3 Syntax of Propositions and `ALuck`

Before we discuss the generator learning algorithm, we must discuss the class of propositions for which we will be learning generators, as well as the class of generators to be learned.

The syntax of preconditions that we plan to handle are shown in Figure 1. These conditions are conjunctions and negations of arithmetical equalities and inequalities with variables ranging over the function's arguments. The allowable expressions in these inequalities are essentially multi-linear functions, where each variable occurs with degree at most one. While this form is restrictive, we have found empirically that the numerical parts most preconditions of normal functions fall within this class. This class of preconditions does not represent the upper limit of what our technique can handle, and we believe that it can be extended to handle constraints with other numerical operators (division, mod). Moreover, there appears to be no inherent difficulty to extending the technique to handle constraints over structured types like lists: we present the algorithm as-is to simplify the discussion.

In order to make the generator inference/synthesis problem tractable, we fix the syntactic form of the generators we plan to consider. Our generators are written in a language called `ALuck` (for *arithmetic* `Luck`) inspired by `Luck` [5], a domain-specific language for writing generators which tries to mirror the declarative form of standard propositions. Generators written `ALuck` run by sequentially *constraining* and *concretizing* variables. Every variable in an `ALuck` generator begins as a symbolic variable. Constraints over these variables are then added. Variables can then be concretized, wherein they are replaced by a random value satisfying the constraints on that variable that have been accumulated thusfar. The final result of the generator is a map from variables to their (randomly) chosen values.

More concretely, generators in `ALuck` are sequences of “concretize" operations, written $!x$, and “constrain" operations, written simply as the constraint to be added. The syntax of `ALuck` is shown in Figure 2. These sequences are then evaluated from left to right while maintaining a pair of mappings: one from concretized variables to their values, and the other from yet-unconcretized variables to the set of constraints that have accumulated on them. When a “constrain" operation c is encountered, the constraint c is added to the constraint sets of all of the variables it men-

Figure 2: Generator Script Syntax

$$\begin{array}{lccl} \text{Generator Actions} & a & ::= & !x \mid eRe \\ \text{Generator Scripts} & s & ::= & [] \mid a :: s \end{array}$$

Figure 3: Generation Semantics

$$\frac{v \sim \mathcal{U}\left(\llbracket C(x) \rrbracket\right) \qquad V\{x \mapsto v\}, C \vdash s \Downarrow V', C'}{V, C \vdash !x :: s \Downarrow V', C'} \qquad \frac{C'(x) = C(x) \cup \{c\} \qquad V, C' \vdash s \Downarrow V', C''}{V, C \vdash c(x) :: s \Downarrow V', C''} \qquad \frac{V \vDash c \qquad V, C \vdash s \Downarrow V', C'}{V, C \vdash c :: s \Downarrow V', C'}$$

tions. If a constraint mentions no variables, it is checked for valdity: if the constraint is not valid, the generator fails. When a “concretize" operation $!x$ is encountered, a value is randomly sampled from the uniform distribution on the set of possible values[1] denoted by the constraints on x. This semantics is shown in Figure 3, where the judgment $V, C \vdash s \Downarrow V', C'$ means that the script s evaluates under concretized-variable-map V and constraint map C, and returns updated variable and constraint maps V' and C'.

Of course, the step of “sampling from constraints specified by inequalities on variables" for concretization operations is exactly what we're trying to solve! To make this step tractable, we enforce that our `ALuck` programs be “well-concretized": when a variable is concretized, all variables which occur in constraints with it must also already have been concretized. This requirement is strict, but it (a) makes it possible to run generators, and (b) simplifies the generator inference process by shrinking the class of generators under consideration.

To give some intuition about how these generator scripts work, we consider running the example script $s = [0 \le x, !x, x \le y, !y]$, which is a generator script for $P(x, y) = 0 \le x \le y$. When we run s, the constraint $0 \le x$ is added to x. Next, x is concretized: a value a is sampled from $\mathcal{U}\left[0, 2^{64} - 1\right]$. Then, the constraint $a \le y$ is added to y. Finally, a value b is sampled from $\mathcal{U}\left[a, 2^{64} - 1\right]$, and the result $\{x \mapsto a, y \mapsto b\}$ is returned.

It is crucial to note that generators can fail. For example, the generator $[!x, !z, x \le y \le z, !y]$ will fail if the generated x and z are such that $z < x$: the sample domain for y will be empty. Some generators like this one fail much all the time, and some such as the generator s above never fail.

Definition 1 (p-Goodness)**.** *We say that a generator script s is p-good if it succeeds with probability at least p.*

4 Generator Candidate Inference

The first step in our algorithm is to infer a set of generators from a given predicate. We begin by noting that every ordering of the variables in a property immediately determines a well-concretized generator: this procedure is shown below in Algorithm 1. In essence, the procedure works by placing all of the constraints that could possibly appear before a concretization immediately before it.

Algorithm 1 Generator from an ordering

```
function CONSTRUCTGENERATOR($xs$,$P$)
    $P_{const} \leftarrow$ conjuncts in $P$ mentioning only one variable
    $P \leftarrow P$ without $P_{const}$
    $s \leftarrow P_{const}$
    $ys \leftarrow []$
    for $x \in xs$ do
        $P' \leftarrow$ conjuncts in $P$ mentioning $x$ and variables in $ys$
        $s \leftarrow$ append($s$,append($c'$,$!x$))
        $P \leftarrow P$ without $P'$
        $ys \leftarrow$ append($ys$,$x$)
    end for
    $s \leftarrow$ append($s$,$P$)
      return $s$
end function
```

Because of this, to infer good generators for a property P, it suffices to generate good orderings of its variables. Unfortunately, for a property P with n free variables, there are $n!$ such orderings: a very large search space. Fortunately,

[1] Because integers are bounded by machine word lengths, the “uniform distribution" does make sense here, even for unconstrained variables.

we can prune this search space by only looking for “relevant" orderings. To illustrate, consider the property $x \leq y \wedge u \leq v$. There are 24 different variable orderings on 4 variables, but there are really only four useful concretization orders: the two possible orders of x and y, combined with the two possible orders of u and v. Because there are no interactions between x and u or y and v, the two orderings x, u, y, v and x, y, u, v will give the same generator. This quotienting allows us to shrink the space of orderings significantly.

To operationalize this, we create an undirected graph from a proposition to encode the important inter-relationships that a concretization order should capture.

Definition 2 (Proposition Graph)**.** *For a proposition P, define $G(P)$ to be the graph whose nodes are variables, with an edge (x, y) when x and y both occur in one of the conjuncts of P.*

Then, to generate a concretization ordering, we randomly depth-first search $G(P)$, and list variables in the order that they're visited in the graph. Pseudocode for this is shown in Algorithm 2.

Algorithm 2 Generate a Random Concretization Ordering

```
function RANDOMCONCR($P$)
    $G \leftarrow G(P)$
    $X \leftarrow \{\}$
    $S \leftarrow$ shuffle(V($G$))
    $xs \leftarrow []$
    while $S \neq []$ do
        $x \leftarrow$ Pop($S$)
        if $x \notin X$ then
            $X \leftarrow X \cup \{x\}$
            $xs \leftarrow$ append($xs$,[$x$])
            $S \leftarrow$ append(shuffle(N($x$)),$S$)
        end if
    end while
    return $xs$
end function
```

Then, to generate our set of generator candidates, we repeatedly run this function. This may give repeated generators, and so we filter the result for uniqueness. It's worth noting that getting repeated results is good: it means that we have successfully pruned the search space sufficiently to get a small number of possible generators. The number of generators in our set requires a careful balance. Too few and the set may not contain a generator which succeeds often enough to rapidly generate our desired number of unique inputs. Too many and the learning algorithm will converge too slowly to the best generator in the set. Empirically, we have found that n^2 (where n is the number of variables in P) is a good number of generators to take.

5 Generator Learning with Bandits Algorithms

With our bag of generators in hand, we now need to find the best one. The approach we take will be inspired by the Multi-Armed Bandits [3] problem [2] from the theory of reinforcement learning. In short, the multi armed bandits problem describes a situation where an algorithm is repeatedly presented a fixed set of choices. Each choice gives a different (random) reward, and the goal of the game is to maximize the total reward by learning which choice or set of choices gives the best rewards. In our setting, the “choices" are our generator candidates, and the rewards are given by success or failure of a generator to yield a value. Under this analogy, an algorithm for the Multi-Armed Bandits problem will let us learn which generators give the best results *while simultaneously* generating a stream of valid inputs for the function.

To make the discussion more concrete, the Multi-Armed Bandits problem is described as the following repeated game: at each round t, the player plays an action $a_t \in [K]$, and receives a reward $X_{a_t,t}$, which is a $\{0,1\}$-valued random variable. The random variables $X_{i,t}$ are IID for fixed i, and independent for fixed t. We write the mean of the ith reward variable (for all t) μ_i. The goal of the game is to maximize one's reward, and so the goal of a bandits learning algorithm is to learn an adaptive policy, which takes a history of play up until state t (all actions $a_{t'}$ and received rewards $X_{a_{t'},t'}$ for $t' < t$), and produces a new action a_t. The metric by which we compare bandits algorthms is *regret*: how much worse they do than the best policy in hindsight.

Definition 3 (Regret)**.** *Define $i_\star = \arg\max_i \mu_i$, and write $\mu_\star = \mu_{i_\star}$. The regret $R(A)$ of an algorithm A over T rounds is defined as*

$$R(A) = T\mu_\star - \mathbb{E}\left[\sum_{t=1}^{T} X_{A(t),t}\right]$$

[2] More precisely, Stochastic Bernoulli Bandits

The algorithm we will use is called UCB1 [1]. UCB1 achieves a regret bound of $O(\sqrt{KT \log K})$. While algorithms are know which achieve $O(\sqrt{KT})$ regret, UCB1 is simple to implement and understand, and performs more than well enough for our setting. In Algorithm 3, we present the UCB1 algorithm, modified for use in property-based testing. The algorithm takes generator candidates $g_1, \dots g_k$, the property P, a number of rounds to run for T, and returns a list of generated values satisfying P.

Algorithm 3 Learn a Generator

```
function UCB1(generators $g_1, \dots, g_K$, property $P$, rounds $T$)
    for $i = 1 \dots K$ do
        $x_i \leftarrow \text{sample}(g_i)$
        $\hat{\mu}_i \leftarrow 1$ if $P(x_i)$, 0 otherwise
        $n_i \leftarrow 1$
    end for
    $X \leftarrow []$
    for $t = 1 \dots T$ do
        $j \leftarrow \arg\max_i \hat{\mu}_i + \sqrt{\frac{2 \log t}{n_i}}$
        $x \leftarrow \text{sample}(g_j)$
        if $P(x)$ then
            $r_t \leftarrow 1$
            $X \leftarrow \text{snoc}(X, x)$
        else
            $r_t \leftarrow 0$
        end if
        $\hat{\mu}_j \leftarrow \hat{\mu}_j + r_t$
        $n_j \leftarrow n_j + 1$
    end for
end function
```

Theorem 1. *Fix a property P and generators $g_1, \dots, g_K$ which are p_i-good. Then,*

$$\lim_{T \to \infty} \mathbb{E}\left[\frac{1}{T}\left|\mathit{ucb1}(g_1, \dots, g_K, P, T)\right|\right] \geq \max_i p_i$$

Intuitively, this theorem justifies our intuition that the ucb1 algorithm “finds the best generator". As the number of rounds gets large, the ratio of the number of valid inputs generated by `ucb1` to the number of rounds is no worse than the success probability of the best generator. One can also think of this theorem as saying that, with high probability[3], that the stream of values emitted by an “infinite run" of `ucb1` acts like a $(\max_i p_i)$-good generator for P.

Proof. We begin by noting that $r_t = 1$ in an iteration if and only if an element is added to the output list in stage t. Therefore, the length of the output $\left|\texttt{ucb1}(g_1, \dots, g_K, P, T)\right|$ is precisely the sum of the ($\{0,1\}$-valued) r_t, $\sum_{t=1}^{T} r_t$. In the notation of the bandit problem, r_t is the (revealed) value of the random variable $X_{A(t),t}$, and so

$$\mathbb{E}\left[\left|\texttt{ucb1}(g_1, \dots, g_K, P, T)\right|\right] = \mathbb{E}\left[\sum_{t=1}^{T} r_t\right] = \mathbb{E}\left[\sum_{t=1}^{T} X_{A(t),t}\right]$$

.

But then by the definition of regret,

$$\mathbb{E}\left[\sum_{t=1}^{T} X_{A(t),t}\right] = T\mu_\star - R(A)$$

Dividing through by T, using the regret bound for UCB1, and the fact that $p_i \geq \mu_i$, we have that:

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{T}\sum_{t=1}^{T} X_{A(t),t}\right] &= \mu_\star - \frac{R(A)}{T} \\
&\geq \mu_\star - O\left(\sqrt{\frac{K \log K}{T}}\right) \\
&\geq \max_i p_i - O\left(\sqrt{\frac{K \log K}{T}}\right)
\end{aligned}$$

which approaches $\max_i p_i$ as $T \to \infty$. □

[3] One could do an analysis using Chebyshev’s inequality to bound the difference between the empirical frequency of the sequence and its mean

Table 1: Results

Constraint	K	Rejected	Unique
$0 \leq x \leq y \leq 100 \wedge x \leq 100$	2	11	831
$x = y + z \wedge x, y, z \geq 0$	5	506	494
$x = y + z \wedge x \geq y \wedge x, y, z \geq 0$	4	48.6	951.4
$0 \leq x \leq y$	2	24.6	975.4
$0 \leq x \leq y \leq 100$	2	993.4	6.6
$0 \leq x \leq y \leq 100 \wedge x \leq 100$	2	11	831
$x - y \leq 5 \wedge x - y \leq 10 \wedge y - z \leq 2$	3.8	26.2	831

6 Implementation, Results, and Future Work

I have implemented a prototype of a deductive verifier backed by PBT based on the above algorithms, it is available here. All told, it is approximately 1000 lines of Haskell, approximately 300 lines of which are parsing code. The implementation makes use of the QuickCheck library [2] for generation combinators. The verifier implements a variant of the `IMP` language [4] with assignment, conditionals, and iteration. The only novel aspect of the implementation is the generation algorithm described in Sections 3,4, and 5: the testing of Hoare triples is trivial to implement, and is essentially folklore at this point.

Because the generator inference algorithm implemented by the prototype is the first of its kind, there are no existing tools we can compare it to in a benchmark. Instead, we simply present our results as a table (Table 1) of data about runs of the algorithm on different inputs. We leave it to the reader to decide if these results are impressive.

In Table 1, we show the results for some trials of our algorithm. In each trial, we run the generration algorithm on the specified constraint five times for $T = 1000$ iterations. K is the average number of unique scripts discovered by Algorithm 2, Rejected is the average number of failed generator samples that happen during the runs, and Unique is the average number of unique input sets generated. In all cases, the generation takes less than 1ms of wall clock time, as reported by the Haskell library `timeit`. By default, integers are generated in the range $[-10000, 10000]$ instead of the full integer range.

As Table 1 indicates, more work is required in order for this algorithm to be ready for prime time. Most of this work needs to be done on the script inference side of things, to get better generator candidates. One low hanging fruit is a range analysis. This is exemplified by the (huge) performance differences between the runs on the proposition $0 \leq x \leq y \leq 100$ with and without the added constraint $x \leq 100$. This constraint is already implied by the first, but not explicitly including it degrades performance significantly. The “best" generator in for the first proposition samples an arbitrary $x \geq 0$, which is very unlikely to leave room for some y satisfying $x \leq y \leq 100$. Including the added constraint (which can be inferred from the first) improve performance dramatically. Another source of potential improvement is in Algorithm 2, where we could attempt to remove edges from the graph in order to minimize the number of scripts. Finally, we have only scratched the surface of reinforcement learning in Algorithm 3. Further tricks such as a probabilistic policy and dropping particularly bad generators could lead to faster convergence of the learner.

References

[1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[2] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, sep 2000.

[3] John C Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society: Series B (Methodological)*, 41(2):148–164, 1979.

[4] Edgar T. Irons. Experience with an extensible language. *Commun. ACM*, 13(1):31–40, jan 1970.

[5] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner’s Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 114–129, 2017.