Flexible and Automatic Targeted Property-Based Testing

LIAM DEVOE, University of Maryland, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA

Property-based testing is a popular testing methodology where executable properties are checked against random inputs. Rather than randomly sampling the space of such inputs, *targeted* property-based testing guides a search process through that space using a user-provided *utility function*: a metric that signifies the desirability of each input. While such utility functions provide users with a lot of flexibility to guide the search, they only look at inputs in isolation, which means they cannot encode fuzzing-like approaches where inputs become desirable as a result of leading the program down a previously unexercised path. At the same time, this flexibility comes at the cost of user effort: utility functions must be written by the users each time. While this is usually reasonable—utility functions often leverage domain-specific information—users already provide a lot of information by writing the properties themselves, which can be leveraged for automation purposes.

In this work, we make targeted property-based testing more *expressive*, by generalizing it to take the history of executed tests into account, allowing—amongst other things—for encoding coverage-guided fuzzing. We also make targeted property-based testing more *automatic*, providing useful default utility functions that target property preconditions, significantly increasing the percentage of inputs that satisfy such preconditions without user effort. We implement our approach on top of Python's Hypothesis property-based testing framework and evaluate our tool on three distinct case studies, demonstrating its potential.

Additional Key Words and Phrases: Test generation, Property-based testing, Targeted property-based testing, Coverage guided fuzzing, Hypothesis

1 INTRODUCTION

Property-based testing (PBT) is a popular methodology for establishing the correctness of a system, by repeatedly executing said system on (usually random) inputs and checking if they satisfy user-defined *properties*: predicates specifying the system's expected behavior on arbitrary inputs. Originating with Haskell's QuickCheck (Claessen and Hughes 2000), property-based testing has recently enjoyed an explosion in popularity, with PBT frameworks being prominently featured in many language ecosystems (Arts et al. 2008; Bulwahn 2012a; Cruanes 2017; Dolan 2017; Lampropoulos and Pierce 2018; Nilsson 2019; Papadakis and Sagonas 2011).

For concreteness, consider how users of one of the most widely used PBT tools, the Hypothesis framework for Python (MacIver et al. 2019), can write a property that asserts a `sort` function always yields a sorted list:

```
@given(lists(integers()))
def sorted_correct(l):
    assert is_sorted(sort(l))
```

The property itself is a function that operates on an arbitrary input list `l` and `asserts` the correctness of sorting it. Hypothesis users must also specify the test generation *strategy*: the `@given` annotation instructs Hypothesis to generate random test data using the predefined strategy `lists(integers())` for generating lists of integers.[1]

[1]There is one additional underappreciated component, *shrinking*, which allows for minimizing counterexamples once they're found. We'll return to this point in Section 7.

Authors' addresses: Liam DeVoe, lidevoe@umd.edu, University of Maryland, USA; Leonidas Lampropoulos, leonidas@umd.edu, University of Maryland, USA.

2022. 2475-1421/2022/1-ART1 $15.00
https://doi.org/

Unsurprisingly, the effectiveness of property-based testing greatly relies on the test generation strategy selected. In many cases, such as the simple sorting property above, users can simply use one of the many strategies bundled with a PBT framework. Unfortunately, in many more situations, effectively testing a system requires generating a suite of test inputs satisfying additional constraints, which can be either implicit (e.g. ensuring a wide range of program behaviors are exercised) or explicit (e.g. satisfying a precondition of the property).

Going back to our sorting example, consider the following slightly different property that checks that if we properly insert an element into an already sorted list, then the result remains sorted:

```
@given(lists(integers()), integers())
def insort_correct(l, x):
    assume(is_sorted(l)) # <- Precondition
    assert is_sorted(insort(l, x))
```

In this example, the property contains a *precondition* that dictates that the `assertion` of the property need only hold if the `assume` statement holds. If not, the property holds vacuously and any test that fails to satisfy this precondition is essentially discarded.

If we were to test this property with the `@given` annotation in the snippet above, we would generate random pairs of a list of integers `l` and an integer `x`, and then *check* if `is_sorted` holds after inserting `x` into `l`. Sadly, even in a simple example such as this, generating random lists of integers and hoping they are sorted is not going to lead to effective testing: sorted lists of length more than 2 or 3 are quite unlikely to be generated at random. Worse, if we were to try and mitigate this issue by only generating empty or singleton lists (which would trivially satisfy the precondition), we would probably fail to exercise several interesting execution paths.

So what can a property-based testing user do in such a case?

Solution 1: Generate and Test. One solution is to just ignore the problem! In particular when the preconditions are not too sparse, and one is looking for shallow feedback about a prototype implementation, a *generate-and-test* approach of filtering out invalid inputs might be sufficient. Unfortunately, this is not often the case.

Solution 2: Handwritten Generators. Instead, property-based testing users often resort to manually writing generators which produce inputs that are specially constructed to satisfy the preconditions by default. For instance, the particular example of testing `insort` has a remarkably simple solution:

```
@composite
def sorted_lists(draw):
    l = draw(lists(integers()))
    return sort(l)
```

That is, we can use `@composite`—a Hypothesis annotation that allows for running arbitrary code inside strategies—to define a `sorted_lists` strategy which samples a random list of integers via `draw` and then sorts it. In fact, if `sort` is not defined in terms of `insort`, then this is a perfectly effective way of testing the latter!

However, this approach doesn't scale well either. In particular, as the complexity of the requirements on test inputs grows, writing such a `@composite` strategy grows exceedingly more so. In fact, such strategies have served as the major of contribution of full research papers: for instance, generating C programs that don't exhibit undefined behaviors to test C compilers (Yang et al. 2011) or well-typed lambda terms to test functional compilers (Pałka et al. 2011).

Solution 3: Automatically Derived Generators. An alternative approach is to attempt to automatically derive a generator based on the code of the precondition itself. Such approaches already exist for domain-specific assertion languages for simple context-sensitive (Steinhöfel and Zeller 2022) or arbitrary (Claessen et al. 2015; Fetscher et al. 2015; Lampropoulos et al. 2017) constraints, or for constraints written in the form of inductive relations (Bulwahn 2012b; Lampropoulos et al. 2018; Paraskevopoulou et al. 2022; Prinz and Lampropoulos 2023), but none exist for a mainstream language such as Python that allows for handling `assume` statements encountered in the wild.

Solution 4: Search-Based Generation. A final alternative is to turn to search-based generation techniques, which aim to systematically explore the vast space of potential test inputs by guiding a randomized search process using dynamic feedback. In particular, coverage guided fuzzing techniques have recently receiving enormous amounts of attention due to their prominent success (Böhme et al. 2019; Lemieux and Sen 2018; Mathis et al. 2019; Pham et al. 2021; Rawat et al. 2017; Vikram et al. 2021; Zeller et al. 2023). Such approaches begin by executing the system under test on an initial set of *seed* inputs, while keeping track of the branches exercised throughout execution. Afterwards, inputs that led the system down previously unexplored paths are saved, and one of them is selected for mutation to produce new inputs. The intuition behind such a choice is that inputs that are "close" (i.e. a small mutation away) to the ones that exercised previously unseen execution paths might lead to yet newer paths being explored.

This fuzzing loop lies at the core of multiple frameworks, such as AFL (AFL 2017), libFuzzer (lib 2019), and Atheris for Python (Ath 2023), that have been used in a plethora of domains with astounding success. However, this loop is also quite inflexible: fuzzers only target programs that receive as inputs arbitrary binary data, and only look for crashes. This comes in stark contrast with the expressive power of a property-based testing tool like Hypothesis that can encode arbitrary properties—for example, fuzzing can in principle be encoded as such a property: for all binary inputs, the program doesn't crash!

Could the high effectiveness of coverage-guided generation in fuzzing be leveraged in tandem with the high expressivity of property-based testing? Of course! Recent work by Padhye et al. (2019b) and Lampropoulos et al. (2019) showcase how coverage information can be used as feedback to improve property-based test generation in Java and Coq respectively, while at the same time, Löscher and Sagonas (2018; 2017) devised a yet more flexible way of incorporating feedback: *targeted property-based testing*. In targeted property-based testing, users provide a utility function over possible inputs that the testing framework tries to maximize during generation. For example, when generating random programs, one could try to maximize the execution length of the program. As Löscher and Sagonas showed, such an approach could automatically match the performance of painstakingly handcrafted generation strategies (Hritcu et al. 2013).

Targeted property-based testing offers a lot of flexibility in allowing programmatically defined utility functions, much in line with the flexibility provided by operating on user-defined properties. As a result, targeted property-based testing has been adopted for a number of property-based testing frameworks, including Hypothesis. However, despite its flexibility, it is not flexible enough. In particular, coverage cannot be used as feedback as it cannot be defined as a function of a single input—rather, coverage is a function of the behavior of the program throughout the *history* of inputs it has been executed on. Moreover, targeted property-based testing, just like property-based testing itself, sometimes offers users *too much* of a choice, without providing a sensible default to serve as an on-ramp for onboarding new practitioners.

In this paper we show how we can get the best of both worlds, offering users maximum flexibility in specifying both arbitrary properties and utility functions that can take test history into account, including aggregate feedback such as coverage. Simultaneously, we offer the ability to automatically

derive targeting functions for `assume` preconditions, relieving the user of the tedium of writing one themselves while providing a sensible default when no `target` function is specified.

Concretely, our contributions are:

- We devise a novel automatic way of developing targeting functions for preconditions, by defining a mapping from terms in an expression calculus to integers, such that, for a given term, valuations of its free variables that lead to higher values in the mapping are "more true". We formalize the mapping in Section 3 and prove the correctness of this intuition using the Coq proof assistant.
- We generalize targeted property-based testing, allowing the target function to rely on the entire history of already executed tests rather than just a single input, opening the door for encoding more generation paradigms within a property-based testing framework. In particular, this allows for marrying the specification expressivity of property-based testing with the testing effectiveness of coverage-guided fuzzing (Section 4).
- We implement both advances on top of Python's Hypothesis, significantly improving both its built-in targeted support and providing additional capabilities (Section 6). We evaluate our tool on three case studies:
  (1) We show that our implementation improves upon the current implementation of targeted property-based testing in Hypothesis by replicating the original case study of (Löscher and Sagonas 2017). We find that our implementation uncovers 11 of the 14 bugs in the original case study; 3 more than Hypothesis. Additionally, we take 39% less time on average to uncover any given bug, with up to 4× improvement for non-trivial cases.
  (2) We apply our automatic targeting for preconditions to actual Hypothesis properties that have `assume` statements in popular Python repositories, demonstrating that our mechanism can greatly improve the throughput of valid inputs generated, sometimes increasing as much as from 16% (without targeting) to 70% (with targeting).
  (3) We pit a prototype encoding of coverage guided fuzzing in our framework against Google's Atheris fuzzer (Ath 2023). We find that our fully-in-Python encoding is only 45% slower than Atheris while using 21% fewer tests to find the same bug, showing the potential of this integrated approach.

2 BACKGROUND

Before we improve upon targeted property-based testing, we need to first introduce a precise characterization of both regular and targeted property-based testing in a way that will allow us to generalize it later on, leveraging the example of the introduction for concreteness. We focus here on two key components of property-based testing: the property, which determines whether an input has uncovered a bug, and the strategy, which generates inputs to test the property.

At a high level, a property $P : I \to \mathbb{B}$ is a function from a set of inputs I to a boolean $\mathbb{B}$, which returns *true* when the property is satisfied and $false$ otherwise. Given a property P which takes inputs I, a generator g is defined as a *distribution* $g : G\ I$ over I. In the `sorted_correct` example of the introduction, I is lists of integers, the return value $\mathbb{B}$ is `is_sorted(sort(l))`, and g is `lists(integers())` which is built by composing the built-in `lists` and `integers` Hypothesis strategies. Don't be confused by the distinction between a *distribution* over I and I itself - in the example above, there are an infinite number of possible generators which produce lists of integers. The `lists(integers())` strategy in Hypothesis is just one such choice, which has been carefully constructed to provide a desirable distribution of inputs.

Targeted property-based testing works really well for a multitude of cases—as evidenced by Hypothesis' success as a tool in the Python ecosystem—but sometimes more guidance of the input generation is needed. To see when this might be the case, we'll turn to an equivalent variant of the

`insort_correct` example from the introduction (with `is_sorted` unfolded into its concrete definition that asserts elements are pairwise sorted):

```
@given(lists(integers()), integers())
def insort_correct(l, x):
    zipped = zip(l, l[1:])
    assume(all(a <= b for (a, b) in zipped))
    assert is_sorted(insort(l, x))
```

As before, random generation is unlikely to generate sorted lists and is therefore unlikely to satisfy the precondition. However, there is a natural *target*, a metric that will guide inputs towards sorted lists: the percentage of sorted pairwise elements! With only a small amount of guidance, we can greatly increase the odds of satisfying the precondition. Indeed, the following targeted-enabled variant of `insort_correct` will - after a short exploratory phase - start generating sorted lists as input:

```
@given(lists(integers()), integers())
def insort_correct(l, x):
    zipped = zip(l, l[1:])
    target(sum(a <= b for (a, b) in zipped) / len(l)) # <- New
    assume(all(a <= b for (a, b) in zipped))
    assert is_sorted(insort(l, x))
```

Intuitively, this works because maximizing the percentage of pairwise sorted elements eventually leads to *all* elements being pairwise sorted, and therefore the overall list being sorted.

Formally, targeted property-based testing is an augmentation of property-based testing with the addition of a utility function $U : I \to \mathbb{R}$ (`target` above). The generator now takes into account a mapping of previously seen inputs to their utility values before returning a value: $g : Map\ I\ \mathbb{R} \to G\ I$. Unsurprisingly, the way previously encountered inputs are taken into account by the generator matters. For the simple `insort_correct` example, even a straightforward hill-climbing strategy (as introduced in the original paper by Löscher and Sagonas) would work, keeping track of only the list with the highest percentage of pairwise sorted elements seen far. In most cases, however, to better explore the search space, we would either need to follow a more sophisticated approach such as simulated annealing (Löcher and Sagonas 2018) or to keep track of previously encountered inputs such as in fuzzing approaches (Zeller et al. 2023).

3 AUTOMATIC TARGETING OF PRECONDITIONS

Targeted property-based testing in the style of Löscher and Sagonas asks users to provide a utility function on inputs that serves as a form of feedback to guide the search over those inputs. It offers an excellent way of incorporating certain kinds of otherwise implicit requirements—such as requiring that generated programs execute for long enough to lead a system to an interesting state, or ensuring that test data exercise a wide variety of execution paths (which we will be able to tackle with our generalization in the next section). However, many times there are *explicitly stated* requirements—preconditions—that test inputs should satisfy; in Python's Hypothesis, these come in the form of `assume` statements. Could we relieve users of the burden of writing a targeting function in addition to already specifying preconditions explicitly? That is the purpose of this section.

For a concrete example, Figure 1 depicts a test taken directly from PyTorch (PyT 2023), a popular machine learning framework for Python, that uses `assume` to assert that several preconditions hold before running any test code, where all of the generation strategies are `integers` with user-specified ranges. The particular details of the test are not important—it simply checks the correctness of a

```
@given(integers(5, 64), ...)
def test_conv2d_transpose(height, width, kernel_h, kernel_w,
                          stride_h, stride_w, pad_h, pad_w,
                          output_pad_h, output_pad_w, dilation):
  kernels = (kernel_h, kernel_w)
  paddings = (pad_h, pad_w)
  dilations = (dilation, dilation)
  assume(height + 2 * paddings[0] >= dilations[0] * (kernels[0] - 1) + 1)
  assume(width + 2 * paddings[1] >= dilations[1] * (kernels[1] - 1) + 1)
  assume((output_pad_h < stride_h) and (output_pad_h < dilation))
  assume((output_pad_w < stride_w) and (output_pad_w < dilation))
  # Test code follows.
  ...
  assert ...
```

Fig. 1. Example PyTorch test with assume statements.

two-dimensional convolution. What is important for our purposes is that the test imposes specific constraints on the relative values of the inputs. Despite the ranges of each integer variable being hand-specified, only 22% of the Hypothesis-generated tests actually satisfy all of these assumptions, leading to a lot of wasted generation effort.

How, then, could targeted property-based testing help in this case? Let's simplify the example a bit and look at only a single inequality:

```
assume(height + 2 * paddings[0] >= dilations[0] * (kernels[0] - 1) + 1)
```

This assumption contains multiple variables that are generated at random by Hypothesis. A user could, in principle, write a strategy that ensures this statement is generated by construction. But doing so for even one of the assumptions of the test above requires a nontrivial amount of effort and understanding! However, there is a simple quantity that we can try maximize in order to make the statement hold—the difference of the left and right hand side:

```
(height + 2 * paddings[0]) - (dilations[0] * (kernels[0] - 1) + 1))
```

Indeed, if we use targeted property-based to maximize this difference we will incrementally arrive at a satisfying valuation of the generated variables. Namely, when the difference is nonnegative, the precondition is satisfied.

In this section, we show how targeted property-based testing can be used to target a wide range of Hypothesis preconditions, including the one above, by providing a mapping from terms in a language of boolean expressions into integers such that, given such a term that might contain free variables, valuations of these variables that lead to a higher utility value are "more true" than valuations that lead to lower ones. We implement this functionality on top of Python's Hypothesis, allowing users to automatically derive a utility function from the bodies of `assume` statements that our framework will subsequently use to facilitate test generation. Intuitively, such a function is a perfect candidate for targeting, as it will guide the search towards inputs satisfying the precondition.

Targeting Preconditions, Formally. Let's start by defining a small language of arithmetic and boolean expressions, similar to the one available in Software Foundations (Pierce et al. 2023). Arithmetic expressions range over variables, numbers, addition, subtraction, and multiplication,

$$
\begin{array}{rcl}
a & := & X \mid n \\
& \mid & a + a \mid a - a \mid a * a \\
\\
b & := & \mathit{true} \mid \mathit{false} \\
& \mid & a = a \mid a < a \mid a \leq a \\
& \mid & b \vee b \mid b \wedge b \mid \neg b
\end{array}
$$

Fig. 2. A simple language of boolean expressions.

$$
\begin{array}{lcl}
(\!|\mathit{true}|\!)_\sigma & = & 1 \\
(\!|\mathit{false}|\!)_\sigma & = & -1 \\
(\!|a_1 \leq a_2|\!)_\sigma & = & [\![a_2]\!]_\sigma - [\![a_1]\!]_\sigma \\
(\!|a_1 < a_2|\!)_\sigma & = & [\![a_2]\!]_\sigma - [\![a_1]\!]_\sigma - 1 \\
(\!|a_1 = a_2|\!)_\sigma & = & -|[\![a_1]\!]_\sigma - [\![a_2]\!]_\sigma| \\
(\!|b_1 \vee b_2|\!)_\sigma & = & \mathit{max}([\![b_1]\!]_\sigma, [\![b_2]\!]_\sigma) \\
(\!|b_1 \wedge b_2|\!)_\sigma & = & \mathit{min}([\![b_1]\!]_\sigma, [\![b_2]\!]_\sigma) \\
(\!|\neg b|\!)_\sigma & = & \mathit{if}\ (\!|b|\!)_\sigma = 0\ \mathit{then}\ (\!|\mathit{false}|\!)_\sigma\ \mathit{else}\ (-(\!|b|\!)_\sigma)
\end{array}
$$

Fig. 3. Targeting Function for Boolean Expressions

while boolean expressions range over equalities and inequalities over arithmetic expressions, as well as negation, conjunction, and disjunction. The syntax can be seen in Figure 2.

While this language is small, it suffices as the core of our formalization. In fact, the arithmetic expression sublanguage can be expanded arbitrarily to include other features such as functions, lists, or other datatypes. The only thing that we assume from the sublanguage is that there exists some standard semantics for it, which computes the value corresponding to an expression e in a given state σ (which we denote as $[\![e]\!]_\sigma$). We elide the semantics of the arithmetic sublanguage, as it is entirely standard and unsurprising.

What *is* important is the treatment of the comparison operators (such as equalities and inequalities for arithmetic expressions or membership and boolean folds over lists), as well as the treatment of the standard boolean logical operators. For each such operator we need to provide an (in this case integer valued) function that would guide generation towards a satisfying assignment for the free variables of the expression, which we denote as $(\!|e|\!)_\sigma$. This function is defined in Figure 3.

- **Cases** *true*, *false*: For the constants, we need to pick some value that corresponds to *true* and some value that corresponds to *false*. We (somewhat arbitrarily) pick 1 and -1 respectively, and this choice will inform the choices for the rest of the operators.
- **Case** $a_1 \leq a_2$: As we saw earlier in this section, the quantity to target when comparing two arithmetic expressions is simply their difference, $[\![a_2]\!]_\sigma - [\![a_1]\!]_\sigma$. This quantity increases as the gap between a_1 and a_2 grows.
- **Case** $a_1 < a_2$: Strict inequalities are handled similarly, with the exception that we subtract 1 from the result to ensure that when $a_1 = a_2$ the utility value still corresponds to that of *false*.
- **Case** $b_1 \vee b_2$: For disjunction, we only care about maximizing the utility value of *one* of the disjuncts, so we'll keep only the highest—closest to being true—utility value of the two disjuncts.
- **Case** $b_1 \wedge b_2$: Dually, for conjunction, we care about maximizing the utility value of *all* of the conjuncts, so we'll keep the lowest—farthest away from being true—utility value of the two conjuncts.

- **Case** $a_1 = a_2$: For equality, we rely on the utility value that combining conjunctions and inequalities gives rise to, by treating $a_1 = a_2$ as $a_1 \leq a_2 \wedge a_2 \leq a_1$. Following our rules, this is taking the minimum of the two differences, which is equivalent to the negation of the absolute value of the difference of the two expressions.
- **Case** $\neg b$: Intuitively, if $(\!|b|\!)_\sigma$ is low (far from being true), then $(\!|\neg b|\!)_\sigma$ should be high (close to being true), and vice versa. This intuition suggest using the negation of the utility value of the inner expression $-(\!|b|\!)_\sigma$. However, there exists an edge case: some true states have a utility value of 0, and we would like negating these states to result in a lower utility value instead of also being 0. Here, we use that of *false*. This does have the unfortunate side-effect that $[\![\neg\neg b]\!]_\sigma$ is not necessarily equal to $[\![b]\!]_\sigma$, but such a double negation expression would be very unlikely to naturally occur in a Hypothesis test.

Correctness. Armed with this mapping, we can now formally state what we intuitively described earlier as expressions with higher utility value being "more true": Given an expression b and two states σ_1 and σ_2, if the utility value of b in σ_2 is higher than its value in σ_1, then the truth value obtained by evaluating b in σ_1 implies that of b in σ_2.

Theorem 3.1. $\forall\ b\ \sigma_1\ \sigma_2.(\!|b|\!)_{\sigma_1} \leq (\!|b|\!)_{\sigma_2} \rightarrow [\![b]\!]_{\sigma_1} \rightarrow [\![b]\!]_{\sigma_2}$

We formalize and prove this entire development in the Coq proof assistant.

The key insight behind the proof is that states with negative utility values are always false, and states with nonnegative utility values are always true. Formally, as a lemma:

Lemma 3.2. $\forall\ b\ \sigma.((\!|b|\!)_\sigma < 0 \leftrightarrow \neg[\![b]\!]_\sigma) \wedge ((\!|b|\!)_\sigma \geq 0 \leftrightarrow [\![b]\!]_\sigma)$

This "inflection point" of 0 is no accident, and is a result of our choices for the targeting function $(\!|b|\!)_\sigma$. In particular, this justifies the definition of $(\!|\neg b|\!)_\sigma$: if $(\!|b|\!)_\sigma$ is positive, $[\![b]\!]_\sigma$ must be true, so our only choice is to negate $(\!|b|\!)_\sigma$ to match $[\![\neg b]\!]_\sigma$ being false. In fact, one can show that this theorem is a sufficient requirement that a targeting function $(\!|b|\!)_\sigma$ must satisfy for Theorem 3.1 to hold.

This raises a difficult question: if satisfying Theorem 3.2 is sufficient, then why are we choosing such a complicated targeting function? In particular, consider the following simple targeting function, which trivially satisfies Theorem 3.2:

$$(\!|b|\!)_\sigma = \mathit{if}\ [\![b]\!]_\sigma\ \mathit{then}\ 0\ \mathit{else}\ -1$$

The perhaps unsatisfying answer is that the latter targeting function simply does not give rise to good behavior in targeted property-based testing: there is no incremental increase in utility value that can be targeted. In contrast, our evaluation in Section 6 shows that our choice of a targeting function can have a significant positive impact in generating inputs that satisfy their precondition.

Extensions. To tackle more realistic properties and preconditions, we need to extend the language supported. The first observation to take into account when extending this core is that the only requirement for expressions is that they can be evaluated. As a result, we can arbitrarily extend the grammar of arithmetic terms a with features such as arrays and lists, or functions and function calls, without any change to our targeting mechanism, and our implementation supports arbitrary Python expressions for operands as a result.

The other observation is precisely the fact that Theorem 3.2 is sufficient. For example, to extend our core to handle real-valued comparisons, we can simply use floating-point difference wherever integer subtraction was used. For another example, if we wanted to allow for boolean folds over lists, such as `all` or `any`, we can define them by simple structural recursion over the list arguments:

$$
\begin{array}{lcl}
(\!|all\ p\ []|\!)_\sigma & = & (\!|true|\!)_\sigma \\
(\!|all\ p\ (h :: t)|\!)_\sigma & = & (\!|p\ h \wedge all\ p\ t|\!)_\sigma \\
(\!|any\ p\ []|\!)_\sigma & = & (\!|true|\!)_\sigma \\
(\!|any\ p\ (h :: t)|\!)_\sigma & = & (\!|p\ h \vee any\ p\ t|\!)_\sigma
\end{array}
$$

4 GENERALIZED TARGETED PROPERTY-BASED TESTING

Let's turn now to a case where targeted property-based testing, in its current form, is not equipped to handle. Suppose there is a deeply nested bug, which is only uncovered by a specific input. For a concrete example, consider the following simplified property:

```
@given(text())
def buggy(s):
    if s[0] == "b":
        if s[1] == "u":
            if s[2] == "g":
                raise ValueError("bug␣found!")
            else:
                ... # More code
        else:
            ... # More code
    else:
        ... # More code
```

In this example, the `buggy` function takes as input a string `s`; performs a sequence of checks against characters in the string; and finally, if the string starts with "bug", raises an error. While this example is contrived, we can still use it for presentation purposes to discuss what coverage-guided fuzzing brings to the mix.

First of all, how does completely random generation of strings fare here? Unfortunately, not very well. A quick back-of-the-envelope calculation would show that if we're generating (long enough) sequences of ASCII characters, a fault-inducing string only has a $\frac{1}{256}^3$ chance of being generated, which is less than 1 in 16,000,000. Worse, as the sequence of choices that needs to be made to reach a particular code block grows, the likelihood of triggering that block at random grows exponentially. As a result, purely random testing has almost no chance of uncovering such bugs.

On the other hand, if we had a way of making incremental progress towards strings with "bug" as a prefix, there is (significant) hope. Suppose we had a way of identifying a string starting with "b" as "interesting": that is, it leads execution down a new path. Such a string has a $\frac{1}{256}$ chance of being generated, which is very likely to occur when executing thousands of tests. Moreover, if we start from such a string and mutate it by changing a random character, we have a $\frac{1}{256*l}$ chance of changing its second character into a `u` (where l is its length). In other words, by remembering inputs that exercised new program behavior and mutating them, the chance of reaching a particular code block no longer scales exponentially with the length of the sequence of choices that need to be made to reach it!

Coverage-guided fuzzing techniques leverage this intuition to guide a search process, in the hopes of covering a wide array of program behaviors. Could we then use targeted property-based testing to guide the search in a similar fashion? If so, what should the utility function be?

A first naive attempt would be to try to maximize a quantity like "number of lines executed" or "number of branches taken". Unfortunately, that doesn't quite work. While we could make

PBT	Targeted	Generalized
I	I	I, X, S
$p : I \to \mathbb{B}$	$p : I \to \mathbb{B}$	$p : I \to \mathbb{B} \times X$
$g : G\ I$	$g : Map\ I\ \mathbb{R} \to G\ I$	$g : S \to G\ I$
	$U : I \to \mathbb{R}$	$f : S \times I \times X \to S$

Fig. 4. Property-based testing, targeted property-based testing, and generalized targeted property-based testing abstractions.

incremental progress towards our goal, if *any* of the other branch bodies in our example (marked `# More code`) contain respectively more lines or more branches than our buggy path, our search will be guided towards exploring those instead. That is, it is not the number of lines or branches executed that is interesting—rather, it is hitting a line or a branch *that hasn't been encountered before*. How then could we incorporate that idea into targeted property-based testing? By simply allowing targeting functions to access (an abstraction of) the entire history of previously seen inputs.

Formally, we introduce two objects: a state S, representing the history of previous tests, and some arbitrary data X produced during execution of a test, representing (runtime) information gathered. As a result, properties become functions that return such data: $p : I \to \mathbb{B} \times X$. Finally, we introduce a function $f : S \times I \times X \to S$ which updates a state S given an input I whose execution produced new data X. If an input leads to an updated state, we deem it interesting for further mutation. This abstraction is contrasted with regular and targeted property-based testing in Figure 4.

Crucially, both targeted property-based testing and coverage-guided fuzzing are straightforwardly encodable in our generalization. For targeted property-based testing, we simply let both X and S be real numbers $\mathbb{R}$, with X being computed as the utility function of an input. Then we take the update function to be $f(s, i, x) = max(s, x)$. Since max is monotonic, this update function will only update the state if an input has a higher utility value than previously seen, reconstructing targeted property-based testing.

For a simple encoding of coverage-guided fuzzing, we let the state S be the set of all possible jumps of a program, and X be the set of branches actually taken during execution. The update function is simply the union of these sets: $f(s, i, x) = s \cup x$, meaning that an input will update the state if it led execution down a previously unseen path.

It is also straightforward to encode more nuanced fuzzing strategies. For instance, AFL's state is not a set, but a map from branches to natural numbers—corresponding to the log of the number of times each branch is taken. That way following a loop twice instead of once counts as "interesting", whereas following a loop ten times instead of nine does not.

Finally, we could further refine our abstraction, to expose to users finer details of the search process—such as the power schedule of seeds (how long to mutate a seed for). In our implementation, we opted to simply follow standard fuzzing practices without further burden to users (Section 6).

The Example, Revisited. Going back to our `buggy` example, our state will track the set of lines which have been exercised by any input. We write the function which updates state accordingly:

```
def update_lines(state, data):
    for file, fdata in data['files'].items():
        state[file] |= set(fdata['executed_lines'])
    return state
```

```
@given(
  text(),
  update_state=update_lines, # <- New
  state=defaultdict(set)     # <- New
)
def buggy(s):
    cov = Coverage()  # <- New
    cov.start()       # <- New

    if s[0] == "b":
        if s[1] == "u":
            if s[2] == "g":
                raise ValueError("bug found!")
            else:
                ... # More code
        else:
            ... # More code
    else:
        ... # More code

    data = cov.get_data()  # <- New
    update_state(data)     # <- New
```

Fig. 5. Instrumented version of buggy.

We then instrument buggy, as seen in Figure 5 to collect coverage information. As we will see in the evaluation section that follows, this setup can almost match the performance of Google's optimized Atheris fuzzer (Ath 2023), while still retaining the full expressive power of property-based testing, showing the potential of our unified approach.

5 EVALUATION

We implemented our advances on top of the Hypothesis property-based testing tool for Python, revamping its targeted infrastructure to accommodate generalized targeting functions and adding automatic precondition targeting. We discuss this implementation in detail in Section 6 and its limitations in Section 7. In this section, we evaluate our tool on three case studies, one for each of our contributions.

First, we focus on our implementation of targeted property-based testing itself compared to the prototype implementation that currently exists in Hypothesis. To evaluate it, we replicate a case study from the original paper (Löscher and Sagonas 2017) on testing information flow control monitors. Then, we evaluate our tool's ability to target preconditions, by targeting multiple existing tests from popular Python repositories that use Hypothesis. Finally, we demonstrate the potential of generalized targeted PBT by pitting our tool against Google's Atheris framework in one of their own example fuzz targets.

5.1 Improving TPBT for Python

For our first case study, we turn to the work of Löscher and Sagonas that introduced Targeted Property-Based Testing, and in particular their one freely available case study on testing the

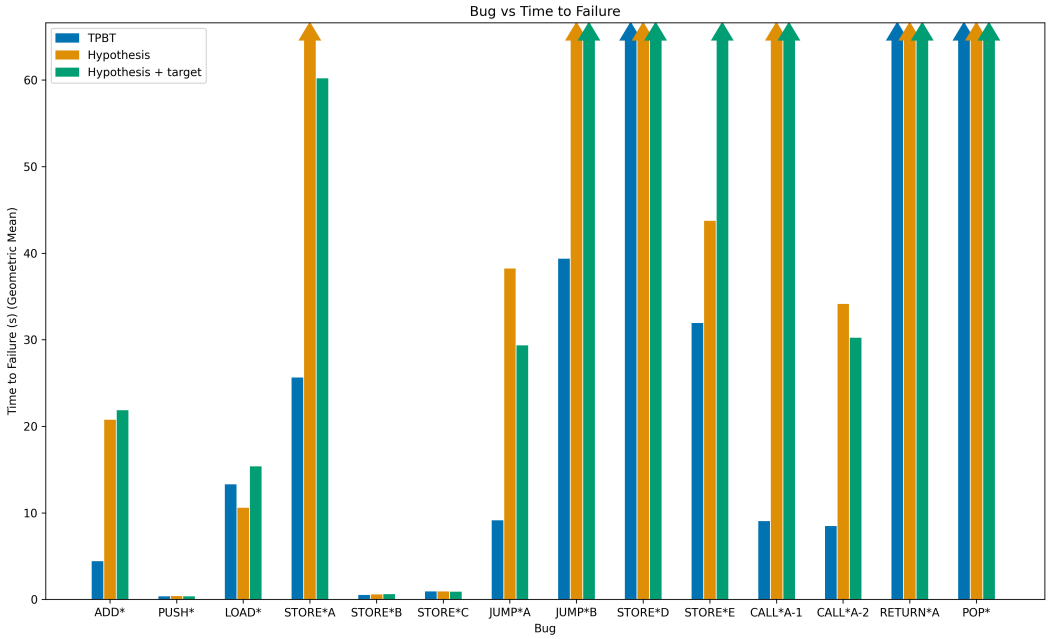

Fig. 6. Mean time to failure for each of our implementation (blue), Hypothesis (orange), and Hypothesis' version of targeted PBT (green). Each cluster of bars on the x axis corresponds to a different injected fault. Bars with upwards arrows indicate that the bug, while theoretical possible to uncover, could not be uncovered in any of our tests within a time limit of 5 minutes.

information flow control monitors of Hritcu et al. (2013, 2016). In their work, Hritcu et al. describe an implementation of abstract stack-based machines where each piece of data is augmented with a *label* that is public ($\bot$) or secret ($\top$), representing data that respectively is or isn't visible to an external observer without special clearance. The machines come with a built in dynamic monitor that tracks and propagates these labels throughout the execution of the machine, ensuring *noninterference*: differences in secret data should never influence the values of public data (Goguen and Meseguer 1982).

One focus of Hritcu et al. is testing whether variants of these monitors enforce this *end-to-end noninterference* (EENI) property, using increasingly smarter generators for abstract machines. The baseline generator produces a pair of *indistinguishable* abstract machines—that is, two machines which differ only in the values of data labeled secret. However, the instruction sequence in these machines is randomly generated, which means it's unlikely the machine will run for long enough to exercise interesting execution paths of the control flow monitors. In contrast, the smarter generator produces instruction sequences *by execution*: it generates a single instruction that can be executed from the current state, takes a step, and repeats this process from the resulting state. The abstract machines generated this way are therefore much more likely to execute for longer and reach interesting execution paths, but at the cost of significant user effort in writing the generator. Löscher and Sagonas realized that targeted property-based testing can be used to simulate this process when the targeting function is to *maximize the number of steps in an execution of the abstract machines*.

Tab. 1. Assignment of IFC weights to instructions.

Instruction	Weight
NO-OP	0
PUSH	$\frac{1}{2^n}$
CALL	$\frac{5}{2^n}$
RETURN	0
ADD	$\frac{1}{2^n}$
LOAD	$\frac{1}{2^n}$
STORE	$\frac{1.5}{2^n}$
JUMP	$\frac{1}{2^n}$
POP	$\frac{1}{2^n}$
HALT	0

To replicate this case study, we first re-implemented the abstract stack machine of Hritcu et al. in Python, including all of their artificially crafted faults that could be programmatically injected, so that Hypothesis can be directly used to test the different buggy variants of it. The baseline generator we use for Hypothesis and TPBT is a direct Python encoding of the baseline generator from Löscher and Sagonas (2017), augmented with the *weighted*, *sequence*, and *smart integers* modifications described therein.

There is a slight question in what metric to target. As mentioned above, the most obvious metric is *execution length*. However, without further guidance, this metric guides generation towards machines filled with only PUSH instructions—one of the few instructions that imposes no restrictions on the state of the stack or memory, as instructions such as LOAD or STORE can cause

Tab. 2. IFC Test results. Time is geometric mean, in seconds. DNF (Did Not Finish) denotes the number of trials which did not uncover the bug.

Bug	**Our Tool**		**Hypothesis**		**Hypothesis Targeted**	
	Time	**DNF**	**Time**	**DNF**	**Time**	**DNF**
ADD*	4.476	0	20.831	26	21.907	27
PUSH*	0.403	0	0.446	0	0.42	0
LOAD*	13.344	8	10.657	96	15.42	98
STORE*A	25.701	51	N/A	100	60.254	99
STORE*B	0.568	0	0.642	0	0.66	0
STORE*C	0.996	0	0.981	0	0.96	0
JUMP*A	9.192	78	38.304	71	29.415	74
JUMP*B	39.411	99	N/A	100	N/A	100
STORE*D	N/A	100	N/A	100	N/A	100
STORE*E	31.993	98	43.785	99	N/A	100
CALL*A-1	9.09	95	N/A	100	N/A	100
CALL*A-2	8.023	43	34.205	74	30.282	72
RETURN*A	N/A	100	N/A	100	N/A	100
POP*	N/A	100	N/A	100	N/A	100

stack underflows. While targeting execution length is the right intuition, we want to also take into account the *diversity* of executed instructions. A machine that executes one of each instruction is more likely to uncover a bug than a machine that executes a single instruction many times.

One way to achieve this is with a decaying weight: each successive instruction executed of a particular type is weighted half as much as the previous one. The metric we target is then the sum of the weights of all executed instructions. The weights we use are summarized in table 1, where n refers to how many of that instruction have been executed so far. Note that instructions do not necessarily share the same starting weight: STORE is the only instruction which can store values to the memory relied upon by other instructions, so we generate those slightly more often. CALL and RETURN impose stringent conditions on the state of the stack and memory, and a higher weight here ensures we hang onto them, should they execute successfully. The weight of RETURN is folded into CALL as both must be generated together to execute. NO-OP and HALT have no impact on program state—except the latter to finish execution—so we do not want to excessively promote their execution in our weighting.

For our tests, we injected each bug of the original paper one at a time and test for 100 trials of 10,000 tests per trial, per bug, for each of Hypothesis' base strategy, Hypothesis' implementation of targeted property-based testing, and our own implementation. The targeting function used for both our tool and Hypothesis targeted is the weighting function described in table 1. The results are depicted in Figure 6, while the full details can be found in Table 2.

We find that our tool (in blue) is consistently better than both: every bug that is found by either usage mode of Hypothesis is also found by our implementation, more frequently, and faster (up to 4x for the not trivial-to-find bugs).

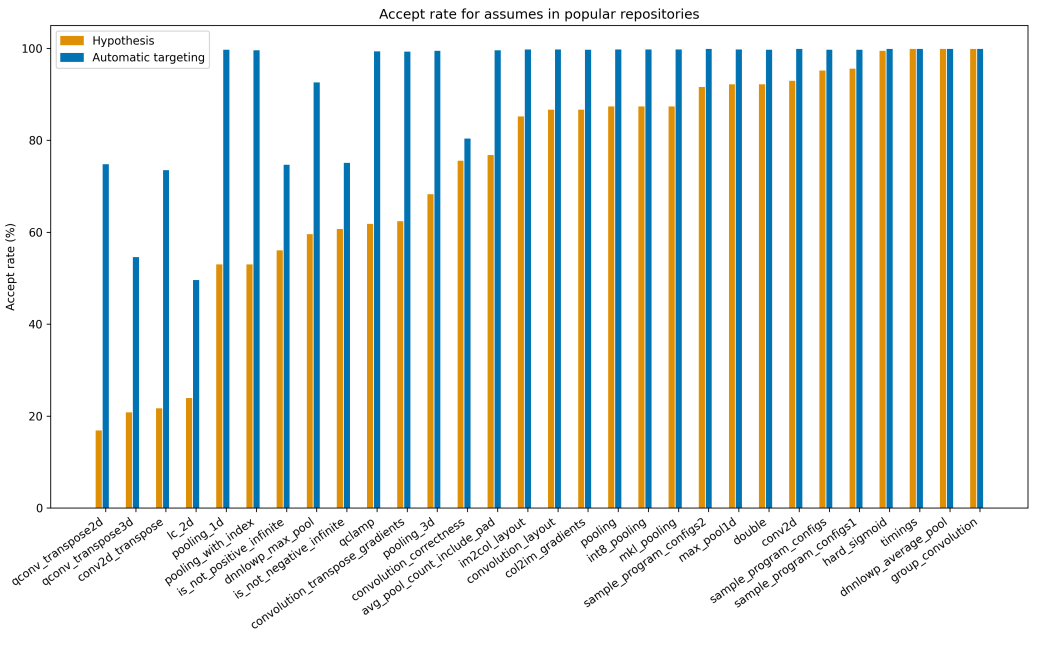

Fig. 7. Input accept rate for Hypothesis compared to TPBT.

Tab. 3. Summary statistics for automatic assume targeting evaluation.

	% Increase in Accept Rate
Median	19.29
Mean	13.06

5.2 Automatic Targeting of Preconditions

For our second case study, we collected a dataset of 30 functions from four of the most popular GitHub repositories that use Hypothesis. First, we identified the most starred GitHub repositories that list Hypothesis as a dependency. For each repository, we collected all tests which use at least one `assume` statement for use in our dataset.

From the 81 tests collected this way, 10 used `assume` statements that fell outside of the language we described in Figure 2. For instance, our language does not include strings, so `assume` statements of e.g. the form `assume(x == "somestring")` could not be targeted except as a simple boolean. Separately, a further 41 functions fit inside our language, but relied heavily on repository-specific helpers or classes, and would have required installing complicated chains of requirements in order to run. We discarded functions with either of these two issues from the dataset. After applying these filters, 30 of the original 81 functions remained.

To evaluate this dataset, we compare the accept rate—that is, the percentage of inputs which satisfy all preconditions—for each function with both Hypothesis and our tool with automated targeting of preconditions. All test functions were run for 10,000 inputs. As shown in Figure 7 and Table 3, the accept rate of our tool is an improvement over random generation for all functions tested.

5.3 Coverage-Guided Fuzzing with Hypothesis

For our final case study, we evaluate our generalization of targeted PBT by comparing against Atheris, a coverage-guided fuzzer for Python. The Atheris GitHub repository contains three sample fuzzers where their tool found bugs. Two of these—an IDNA and JSON fuzzer—use c extensions and could not be instrumented by Slipcover, the Python library we're using to provide coverage (Pizzorno and Berger 2023). As a result, we turn to the third sample given by Atheris, a YAML fuzzer, for this case study.

Our strategy is an encoding of the test harness provided by Atheris: where Atheris uses `ConsumeUnicode`, we use `@given(text(min_codepoint=0, max_codepoint=128))`, a strategy which restricts inputs to ASCII. As Atheris itself provides ASCII values 50% of the time in `ConsumeUnicode`, this was the closest appropriate encoding in our tool. For our target function, we use the coverage function described in Section 4, which considers an input "interesting" if it uncovers *any* line which was previously unexecuted. We test for 100 trials of both our tool and Atheris, and run until a bug is uncovered. The average time and test to failure can be seen in Table 4. We find that our implementation of targeted property-based testing on top of Hypothesis is 1.44× slower than Atheris—despite the fact that Atheris is a heavily engineered and optimized fuzzer.

6 IMPLEMENTATION

In this section we discuss the various design decisions that we had to make during the implementation of our generalization on top of Hypothesis. Before we do that, however, it will be useful to have a base understanding of Hypothesis' internals.

6.1 Hypothesis Background

Hypothesis is a property-based testing framework which provides a number of predefined *strategies* for generating random inputs of various types. These strategies define a distribution over all elements of that type. Of course, not all distributions are equally useful: if `integers` only produced even numbers, this would be a poor distribution indeed. The default strategies in Hypothesis have been written with the goal of providing "good" output distributions in mind.

In practice, strategies in Hypothesis are a mapping from *randomness* to outputs: to produce a value, Hypothesis generates a random string of bits. These bits are then interpreted by the strategy as a series of choices and returned as a value. The details of how the strategy interprets the bits determines the underlying distribution.

Hypothesis has also carefully constructed their default strategies to interpret these bits in a particularly desirable fashion. Namely, that strings of bits which are *lexicographically smaller* correspond to simpler values. This is the key behind shrinking in Hypothesis: to shrink a counterexample, start from the original string of bits and try to reproduce the counterexample with increasingly lexicographically smaller strings.

As we saw in the introduction, Hypothesis provides a `@composite` annotation which allows users to define strategies that run arbitrary Python code, including drawing random values from other strategies. In particular, this means strategies can be *non-deterministic*. For example, here is a valid strategy which flips a coin by `draw`ing a boolean, and then generates either a string or an integer based on the result of the flip:

```
@composite
def int_or_text(draw):
    b = draw(booleans())
    if b:
        return draw(integers())
    else:
        return draw(text())
```

Finally, Hypothesis provides an implementation of targeted property-based testing, wherein values are mutated via changes to the underlying randomness bit string that produced them. This is in contrast to Löscher and Sagonas (2017), which defines mutations on the *value* itself, not on the randomness which generated it.

6.2 Targeted Property-Based Testing

Our implementation of targeted property-based testing is built on top of, and is separate from, Hypothesis, and completely supersedes its targeted PBT implementation. As mentioned, Hypothesis performs mutation by operating on the random bit string. Our tool instead performs mutation by operating on the *output value* of strategies, following Löscher and Sagonas (2017). This has important implications for shrinking, as we will discuss in Section 7.

Tab. 4. Mean time and tests to failure for fuzzing a YAML parser with Atheris (left) and a coverage-guided targeting function in Hypothesis (right).

	Time to Failure (s)	Tests to Failure
Atheris	6.08	35050
TPBT	8.799	27830

Unlike Löscher and Sagonas, however, we allow our implementation to keep track of multiple past inputs as starting points for further mutation. Such inputs were deemed sufficiently interesting and worth exploring more, but exploring them might still turn out to be a fruitless endeavor focusing on an uninteresting part of the search space. In Löscher and Sagonas, such dead ends are mitigated by using simulated annealing; we instead take a page out of the fuzzing playbook and keep track of multiple seeds for mutation in a *seed pool*.

At a high level, the core loop of our tool is:

- If the seed pool is empty, generate a random input x from Hypothesis;
- Otherwise, pick the best seed from the pool to mutate in order to obtain an input x;
- Call the function under test with x as input;
- If the input x is deemed *interesting* (in the sense of either a higher utility value in targeted PBT or updating the state in generalized targeted PBT), add it to the seed pool.

The power schedule—how long each seed is mutated for—we're using is also inspired by standard fuzzing practice. We imbue each seed with an energy budget of 5,000 to start. That is, if they are mutated a total of 5,000 times without generating a new seed, they are marked as exhausted and discarded. If a seed does uncover something interesting, its energy budget is doubled. The intuition here is that if a seed is uncovering lots of new seeds, we would like to keep it around for longer to ensure we're not missing anything. If the seed pool has been exhausted, we alternate between generating completely random inputs or mutating a random exhausted seed, until the seed pool is populated again.

6.3 Automatic Targeting of Preconditions

Once we have the machinery of targeted property-based testing in place, the implementation of automatic targeting of preconditions is relatively straightforward. We traverse the AST of the test body, looking for `assume` statements that fall within the scope of our precondition language from Section 3. We then compute the corresponding `target` statement and insert it into the AST.

One possible complication arises when we combine our extensions and Pythons overloading. In the small core presented in Section 3, whenever we see an `a < b` node, we can safely assume that `a` and `b` are integers, and use the appropriate `target` rule. However, once we add other ordered types to the mix, we might need to disambiguate what targeting function to use. In some cases, such as when using real numbers instead of integers, the targeting function stays the same. However, in other cases, such as when dealing with sets ordered by the subset relation (as in, $\{1\} < \{1, 2\}$) it is not difficult to define a natural targeting rule for subset—it would simply be the cardinality of their difference. Nonetheless, in such cases, it is not obvious which rule to use for an `a < b` node without the help of type information. Our solution assumes types don't change across multiple executions of the same test and simply check which of the handled types a concrete execution yields. A different, perhaps more principled solution, would be to integrate with mypy, or a similar optional static typing framework for Python.

6.4 Generalized Targeted Property-Based Testing

Generalizing targeted property-based testing is also not particularly difficult to implement. Our core loop remains the same. The interesting bit is our treatment of coverage. We rely on recent work by Pizzorno and Berger (2023) that provides a more efficient way of obtaining runtime coverage information, as it periodically de-instruments branches and lines that have already been covered. In our experiments with the YAML case study, Slipcover led to noticeable speedups in performance, as shown in Table 5.

6.5 Mutation

We have, until now, glossed over the details of how we perform mutation on seed inputs. This is largely because the details are unsurprising: following Löscher and Sagonas, each strategy in our tool defines how to mutate a value drawn from it, potentially invoking the mutation scheme of one or more component strategies in the process. For instance, to mutate `lists(integers())`, we first mutate `lists` by adding, removing, and/or mutating some number of elements. Then, if elements are selected for mutation, we use the mutation scheme defined by `integers`—the component strategy inside `lists(integers())`—to perform that mutation.

However, there is one complication. Recall from Section 6.1 that Hypothesis allows for *non-deterministic* strategies. How does one mutate a value generated by a non-determinstc strategy? Doing so requires knowing precisely which strategy was exercised during generation. This motivates the following implementation detail: we assign a unique *primary key* (PK) to each strategy at the source code level, allowing for tracking which strategy was drawn from and when. These primary keys are automatically inserted at run-time by an AST transformation and require no user intervention.

For instance, the `int_or_text` example of Section 6.1 is automatically transformed to the following before being run:

```
@composite
def int_or_text(draw):
    b = draw(booleans(pk=1))
    if b:
        return draw(integers(pk=2))
    else:
        return draw(text(pk=3))
```

This is reminiscent of very recent work on reflective generators for targeted property-based testing (Goldstein et al. 2023), where different choices a generator makes are annotated with a label in order to allow for recovering a sequence of choices that could have produced a generated value. Interestingly, such a feature could be used to mitigate the one major drawback of our implementation (not our conceptual approach), which we discuss immediately below.

7 LIMITATIONS

Shrinking. We have discussed the *generation* of inputs to properties at great length, but this is only one key component of property-based testing. Other than properties themselves, the final major aspect of property-based testing is *shrinking*: once a counterexample is found, how can we find the simplest counterexample which still reproduces the error in order to present it to the user?

Hypothesis takes a relatively unique stance on shrinking (MacIver and Donaldson 2020), as users don't need to provide an external shrinking function for generated data. Instead, Hypothesis keeps track of the relationship between a generated input and the string of bits that provided the randomness to produce it. When a counterexample is found, Hypothesis tries lexicographically

Tab. 5. Time spent executing and in coverage using Slipcover and coverage.py.

	Total execution time (s)	Time spent in coverage(s)
Slipcover	20.8	7.77
Coverage.py	47.8	26.7

smaller bit strings using the same generator, with the hopes of finding progressively simpler counterexamples.

Unfortunately, our tool severs this link between the source of randomness and generated counterexamples, as we mutate the values coming out of strategies, not the randomness coming into them. We made that choice to keep as close to the original targeted property-based paper, but it is largely orthogonal from our contributions. Both automatic targeting and the generalization would work perfectly well with an internal mutation scheme.

Alternatively, we could leverage the reflective scheme of Goldstein et al. (2023), in order to recover a bit string that *could* have produced a potentially mutated counterexample. Then the default shrinking approach of Hypothesis could proceed as normal. Our handling of non-determinism with primary keys is fortunately very much in line with such an approach, but we leave such an effort for future work.

8 RELATED WORK

The literature on fuzzing and property-based testing is vast. In this section, we're covering the most closely related work—other than targeted property-based testing (Löcher and Sagonas 2018; Löscher and Sagonas 2017), which we're generalizing and have extensively discussed throughout the paper.

Combining Property-Based Testing and Fuzzing. Arguably the most closely related work is formed by independent lines of research aiming to bring together property-based testing and coverage-guided fuzzing. On the one hand, Lampropoulos et al. (2019) introduce FuzzChick, an extension of QuickChick (Lampropoulos and Pierce 2018), a traditional yet powerful property-based tool for the Coq proof assistant. FuzzChick can leverage the full expressive power of property-based testing and enhances it by bringing coverage-guided test generation into the mix. In particular, Lampropoulos et al. introduce a notion of *semantic mutations* that take into account available type information in order to only generate structurally valid inputs. However, branch coverage for the functional specifications that QuickChick usually targets appears to be less useful—as FuzzChick's test generation appears to still be an order of magnitude less effective than handwritten generators.

On the other hand, Padhye et al. (2019a) tackle the problem from the fuzzing side, introducing JQF, a property-based testing platform for Java. A big focus of their work is on *extensible generation*, where different fuzzing strategies can be implemented by defining a new *guidance*—a way for the search generation process to take runtime information into account. For example, they implement *semantic fuzzing* (Padhye et al. 2019b), where in addition to coverage, input generation is skewed towards valid inputs that satisfy structural or semantic constraints of the system under test.

Further along the configurable search axis, Padhye et al. (2019c) allowed users to specify domain-specific notions of feedback, as well as how to aggregate it. This allows for rapid prototyping of strategies as both the search strategy as well as the mutation heuristics can be reused. Much like targeted property-based testing, FuzzFactory offers a flexible mechanism to guide the generation process towards more "interesting" inputs, but (generalized) targeted property-based testing adds to that the additional flexibility of targeting arbitrary user-defined properties.

Finally, Crowbar (Dolan 2017) is an OCaml framework powered by AFL that brings the power of coverage-guided fuzzing to OCaml users. Much like Hypothesis' existing implementation of targeted property-based testing, Crowbar uses AFL to mutate the randomness going into the generators, rather than the structures produced by them. Crowbar has successfully discovered errors in a multitude of widely used OCaml packages, including DCHP library implementations, distributed databases, etc. Still, as it's powered by AFL under the hood, there is no way for users

to provide their own domain-specific forms of feedback to target or to automatically deal with property preconditions.

Property-Based Testing Preconditions. As discussed in the introduction, the last decade has seen a concentrated effort in dealing with properties that have preconditions of a particular form that can be extremely effective in their application domains, but none of which can account for preconditions that appear as boolean expressions in `assume` statements such as the ones Hypothesis users frequently use.

Bulwahn (2012b), as part of Isabelle's QuickCheck (Bulwahn 2012a), developed a method for automatically enumerating inputs that satisfy constraints in the form of inductive relations. A similar line of work exists for the QuickChick property-based testing framework for Coq (Lampropoulos et al. 2018; Paraskevopoulou et al. 2022; Prinz and Lampropoulos 2023), where inputs are sampled randomly rather than enumerated. Rather than targeting inductive preconditions, a different take on the problem has been to generate inputs lazily while attempting to check whether a precondition holds, in a process that resembles narrowing in functional logic programming (Antoy 2000). Claessen et al. (2015) do so for Haskell preconditions that return a special form of "concurrent booleans", Fetscher et al. (2015) extend their approach for typing judgments in PLT Redex, while Lampropoulos et al. (2017) introduce their own Haskell-like domains specific language of preconditions to allow for user-control of the resulting distribution. Finally, Mathis et al. (2019) target syntactic constraints that are imposed by input parsers, while Steinhöfel and Zeller (2022) also introduces their own declarative specification language for context-sensitive preconditions that can be used to produce inputs for fuzzing.

Other Coverage-Guided Fuzzing Approaches. Plenty of recent work aims to improve upon the traditional coverage-guided fuzzing loop. Some examples include approaches like AFLFast (Böhme et al. 2019) which focus their efforts on the power schedule of the fuzzer to concentrate generation time on inputs that are heuristically more likely to provoke previously unseen paths. Pham et al. (2021) and Wang et al. (2019) focus on improving the mutations that are used to produce new inputs, using learned or user-provided grammars. FairFuzz (Lemieux and Sen 2018) and Vuzzer (Rawat et al. 2017) use additional dynamic or static information respectively to target their search towards particular unexplored or rarely explored execution paths. Vikram et al. (2021) aim to grow more concise inputs that are easier for users to debug compared to post-hoc minimized traditional fuzzer counterexamples. All such approaches improve upon aspects of fuzzing that are largely orthogonal to the flexibility provided by the extended the scope of specifications that generalized targeted property-based testing can tackle.

9 CONCLUSION AND FUTURE WORK

In this paper, we showed how targeted property-based testing can be made more expressive, encompassing techniques such as coverage-guided fuzzing, and more automatic, by targeting explicitly specified preconditions at no user effort. In the future, we would like to further improve upon our approach, incorporating more orthogonal ideas from the fuzzing world and ensuring that our mutation-based approach plays well with Hypothesis' sophisticated integrated shrinking.

REFERENCES

2017. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl.

2019. libFuzzer. https://llvm.org/docs/LibFuzzer.html.

2023. Atheris fuzzer for Python. https://github.com/google/atheris.

2023. PyTorch. https://pytorch.org.

Sergio Antoy. 2000. A Needed Narrowing Strategy. In *Journal of the ACM*, Vol. 47. ACM Press, 776–822. https://www.informatik.uni-kiel.de/~mh/papers/JACM00.pdf

Thomas Arts, Laura M. Castro, and John Hughes. 2008. Testing Erlang Data Types with QuviQ QuickCheck. In *7th ACM SIGPLAN Workshop on Erlang* (Victoria, BC, Canada). ACM, 1–8. https://doi.org/10.1145/1411273.1411275

Lukas Bulwahn. 2012a. The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof. In *2nd International Conference on Certified Programs and Proofs (CPP) (Lecture Notes in Computer Science, Vol. 7679)*. Springer, 92–108. https://www.irisa.fr/celtique/genet/ACF/BiblioIsabelle/quickcheckNew.pdf

Lukas Bulwahn. 2012b. Smart Testing of Functional Programs in Isabelle. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (Lecture Notes in Computer Science, Vol. 7180)*. Springer, 153–167. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.229.1307&rep=rep1&type=pdf

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841

Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000143

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 268–279. http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf

Simon Cruanes. 2017. QuickCheck Inspired Property-Based Testing for OCaml. https://github.com/c-cube/qcheck/.

Stephen Dolan. 2017. Property Fuzzing for OCaml. https://github.com/stedolan/crowbar.

Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *24th European Symposium on Programming (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 383–405. http://users.eecs.northwestern.edu/~baf111/random-judgments/

J. A. Goguen and Jose Meseguer. 1982. SECURITY POLICIES AND SECURITY MODELS. *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy* (1982), 11–20.

Harrison Goldstein, Samantha Frohlich, Meng Wang, and Benjamin C. Pierce. 2023. Reflecting on Random Generation. In *Proceedings of ACM Programming Languages*. Seattle, WA, USA. https://doi.org/10.1145/3607842

Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Catalin Hritcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo de Amorim, Maxime Denes, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. 2016. Testing Noninterference, Quickly. In *Journal of Functional Programming (JFP)*.

Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li yao Xia. 2017. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, (POPL)*.

Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*.

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating Good Generators for Inductive Relations. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*.

Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickCHick: Property-Based Testing In Coq*. Electronic textbook. http://www.cis.upenn.edu/~bcpierce/sf

Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

Andreas Löcher and Konstantinos Sagonas. 2018. Automating Targeted Property-Based Testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 70–80. https://doi.org/10.1109/ICST.2018.00017

Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 46–56. https://doi.org/10.1145/3092703.3092711

David MacIver, Zac Hatfield-Dodds, and Many Contributors. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (21 11 2019), 1891. https://doi.org/10.21105/joss.01891

David R. MacIver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13:1–13:27. https://doi.org/10.4230/LIPIcs.ECOOP.2020.13

Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 548–560. https://doi.org/10.1145/3314221.3314651

Rickard Nilsson. 2019. ScalaCheck: Property-Based Testing for Scala. https://scalacheck.org/.

Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019a. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019b. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019c. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (oct 2019), 29 pages. https://doi.org/10.1145/3360600

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

Manolis Papadakis and Konstantinos F. Sagonas. 2011. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*. 39–50. https://doi.org/10.1145/2034654.2034663

Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing Correctly with Inductive Relations. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*.

Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hritcu, Vilhelm Sjoberg, and Brent Yorgey. 2023. *Logical Foundations*. Electronic textbook. http://www.cis.upenn.edu/~bcpierce/sf

Juan Altmayer Pizzorno and Emery D. Berger. 2023. SlipCover: Near Zero-Overhead Code Coverage for Python. *ISTTA* (2023). https://doi.org/10.48550/arXiv.2305.02886

Jacob Prinz and Leonidas Lampropoulos. 2023. Merging Inductive Relations. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*.

Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. https://doi.org/10.14722/ndss.2017.23404

Dominic Steinhöfel and Andreas Zeller. 2022. Input Invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 583–594. https://doi.org/10.1145/3540250.3549139

Vasudev Vikram, Rohan Padhye, and Koushik Sen. 2021. Growing A Test Corpus with Bonsai Fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 723–735. https://doi.org/10.1109/ICSE43902.2021.00072

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 724–735. https://doi.org/10.1109/ICSE.2019.00081

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 283–294. https://doi.org/10.1145/1993498.1993532

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2023. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/