# Automating Targeted Property-Based Testing

Andreas Löscher and Konstantinos Sagonas
Department of Information Technology, Uppsala University, Sweden
andreas.loscher@it.uu.se and konstantinos.sagonas@it.uu.se

***Abstract*—Targeted property-based testing is an enhanced form of property-based testing (PBT) where the input generation is guided by a search strategy instead of being random, thereby combining the strengths of QuickCheck-like and search-based testing techniques. To use it, however, the user currently needs to specify a search strategy and also supply all ingredients that the search strategy requires. This is often a laborious process and makes targeted PBT less attractive than its random counterpart. In this paper, we focus on simulated annealing, the default search strategy of our tool, and present a technique that automatically creates all the ingredients that targeted PBT requires starting from only a random generator. Our experiments, comparing the automatically generated ingredients to fine-tuned manually written ones, show that the performance that one obtains is sufficient and quite competitive in practice.**

## I. Introduction

Testing is an integral part of modern software development as it finds errors in programs and systems and gives confidence about their correctness. Random property-based testing (PBT) is a high-level, semi-automatic, black-box testing technique in which, rather than writing many unit tests by hand, one simply specifies general *properties* that the system under test (SUT) is expected to satisfy, and *generators* that produce well-distributed random inputs to the parts of the system that are tested [1], [2]. As with all random testing techniques, the chance of finding an error and the confidence in the SUT increases with the number of generated tests. In cases where the input space is large, even a big number of tests might not be enough to yield a satisfactory confidence if the inputs are generated (semi-)randomly.

Targeted Property-Based Testing (TPBT) [3] is an enhanced form of PBT that, instead of being completely random, uses a search-based component to guide the input generation towards values that have a higher probability of falsifying a property. TPBT explores the input space more effectively and requires less tests to find a bug or achieve a high confidence in the SUT than random PBT. To use TPBT a user has to specify: (1) a *search strategy* that will be used to explore the input space, (2) a component that supports writing *targeted generators*, and (3) *utility values* (UV) for each input that the generation process tries to either maximize or minimize.

The search strategy is typically provided by the PBT tool or a library and can be configured for the task at hand. In order to be able to guide the generation process, the user needs to manually provide some ingredients such as information about how to generate inputs and strategy-specific operators. Simulated Annealing (SA) for example, the default search strategy used by PROPER [4], a QuickCheck-inspired tool with support for targeted PBT, requires the user to specify a neighborhood function that produces a "next" random input which is similar to a current one. Such a function is significantly harder to write than a generator for random PBT.

In this paper, we present a technique for SA that constructs its main ingredient, namely the neighborhood function (NF), automatically from an input generator written for random PBT. This constructed NF can be used as is, and is capable of producing random neighboring input for all input instances that the random generator can produce. By using the presented technique, we can reduce the effort to use TPBT significantly. In addition to the components needed for random PBT, a user effectively now only needs to extract the utility values and specify whether to maximize or minimize them. As we will show on a series of examples, TPBT performs sufficiently well with the automatically constructed NF for most applications. We also describe how random generators can be written so that the construction process can work to its full potential, and how the constructed NFs can be adjusted by the user manually.

The rest of the paper is structured as follows. The next section overviews random PBT, TPBT, and simulated annealing, thus presenting the background that is necessary to understand the rest of the paper. In Section III we describe how automatically constructed NF's can be used, and describe the algorithm for constructing them in Section IV. We evaluate the efficiency of so constructed NF's against fine-tuned hand-written ones in Section V, and finish with two brief sections containing related work and some concluding remarks.

## II. Background

### A. Property-Based Testing

Property-based testing (PBT) is a random testing technique in which the intended system behavior is expressed by a description of valid inputs to the SUT and properties that are expected to hold when the system is subjected to instances of valid inputs. A PBT tool takes these definitions and successively generates a number of random inputs, often with increasing complexity. The tool then subjects the SUT to these inputs and checks if the outputs falsify the properties or not. Following this method, a tester's manual tasks are reduced to correctly specifying the parameters of the SUT and formulating a set of properties that accurately describe its intended behavior.

PBT tools operate on *properties*, which are essentially partial specifications of the SUT, meaning that they are more compact and easier to write and understand than full system specifications. Users can make full use of the host language when writing properties, and thus can accurately describe a wide variety of input-output relations. They may also write their own *custom generators*, should they require greater control over the input generation process. Compared to testing systems with manually-written test cases, testing with properties is a faster and less mundane process. The resulting properties are also much more concise than a long series of unit tests, but, if used properly, can accomplish more thorough testing of the SUT by subjecting it to a much greater variety of inputs than any human tester would be willing or able to write. Moreover, properties can serve as a checkable partial specification of a system, one that is considerably more general than any set of unit tests, and thus one that is much better at exploring a more significant percentage of behaviors of a system and unveiling its errors.

We illustrate PBT and explain the language of PROPER [4], the QuickCheck-inspired tool we use, with a simple example.

```
prop_compress_decompress() ->
  ?FORALL(D, data(), zlib:decompress(zlib:compress(D)) == D).
```

This property expresses that, for any data `D`, if we compress and then decompress `D` with functions provided in a `zlib` module that implements this compression library [5] we end up with the original data. In PROPER, properties begin with `prop_`, anything that begins with a `?` is a macro corresponding to some operation that the PROPER tool provides, and anything that starts with a capital letter (like `D` here) is a variable. Besides macros, like the `?FORALL` macro, PBT tools come with built-in generators for the base types of the host language (e.g., integers, lists, etc.) that can be combined. To further refine these generators it is possible to specify predicates to constrain their generated values using appropriate operations (e.g., `?LET`, `?SUCHTHAT`, etc.).

The paper that introduced targeted PBT [3] showed that it is difficult for random PBT to find counterexamples in cases where the input domain is large and inputs that falsify a property are rare. In these cases, the number of tests that need to be run in order to find a counterexample or have high confidence that the SUT behaves as intended can become unfeasible. With better knowledge of the input domain, the odds of finding a counterexample can be increased. For example, limiting the generator to values where a bug is likely to be found increases the probability of finding an input value that falsifies a property. However, such generators are more complicated to write and less general. Typically one generator is shared by multiple properties and having to write a tailored custom generator for each property makes PBT testing less attractive.

## *B. Targeted Property-Based Testing*

Targeted property-based testing [3] (TPBT) is an enhanced form of PBT which is applicable in cases where the property to test can be expressed as an optimization problem (maximizing or minimizing a certain measure). In such cases, TPBT guides the input generation with search strategies instead of generating them only randomly. It aims to make test outcomes more consistent and reduce the number of required test runs to find bugs or to achieve higher confidence in the SUT compared to random PBT. It uses information gathered during test execution in the form of *utility values* (UVs) that specify how close input came to falsifying a property. TPBT consists of three main components: (1) the *search strategy* that is used to explore the input space, (2) the component that supports writing *targeted generators*, and (3) *UVs* that we want to maximize or minimize. The UVs are paired with the input to the property. If an input has a UV beyond the property-specific threshold, then the property will fail. The difference between this threshold and the UV is effectively the *distance* between the input and a potential counterexample for the property.

The general structure of properties that can be tested with TPBT, called *targeted properties*, looks as follows:

```
prop_Target() ->                           % Try to check a property
  ?STRATEGY(SearchStrategy,                % for some Search Strategy
    ?FORALL(Input, ?TARGET(Params),        % and for some Parameters
            begin                          % for the input generation.
              UV = SUT:run(Input),         % Do so by running SUT with Input
              ?MAXIMIZE(UV),               % and maximize its Utility Value
              UV < Threshold               % up to some Threshold.
            end)).
```

The search strategy generates input for each run and tests the property with it. Besides running the test with the current input, the `SUT:run()` function needs to return the utility value. This UV is then fed to the search strategy component which uses this information to produce the next input with an increased UV, thereby also increasing the chance of falsifying the property.

Most search strategies require additional information about how the inputs are generated. Simulated Annealing (SA) for example, the default strategy provided by PROPER for TPBT, requires the user to specify a generator for the first input and a *neighborhood function* (NF).

Let us consider an example [3]. Suppose we want to test whether a network of nodes performs as expected regardless of its topology. The input to such a property would be graphs of a fixed number of vertices (network nodes). Many performance criteria of networks, like energy consumption or message latency, are influenced by the number of hops messages need to take to reach their destination. In our example, let us suppose that the majority of the messages are going to one dedicated node, the sink. For simplicity, let us also assume that the SUT returns the lengths of all shortest paths between the sink and the other nodes. We can formulate a property that states that the longest of those paths should not exceed 21 hops (for a network with 42 nodes). Note that it is very complicated to write a random generator for graphs that has a good chance of finding a counterexample in a reasonable amount of time. The topology of the graph has to have a particular shape which is only found in a relatively small percentage of the inputs. Using TPBT however, we can use the length of the longest path to the sink as UV, and instruct the search strategy to generate topologies that maximize it. Following the general structure for targeted properties, we can write the code in Fig. 1. The `?FORALL_SA` macro is a targeted variant of `?FORALL`, which uses simulated annealing as search strategy, and thus avoids having to write an explicit `?STRATEGY` line. The `?TARGET` macro is used to mark generators that are under the control of the search strategy. Simulated annealing requires as ingredients a first input and a NF, a function that produces input that is in the neighborhood of the current input (close in the input space). We use an easy

```
prop_length() ->  % targeted property
  ?FORALL_SA(G, ?TARGET(graph_sa(42)),
             begin
               UV = lists:max(distance_from_sink(G)),
               ?MAXIMIZE(UV),
               UV < 21
             end).

graph_sa(N) ->    % targeted generator for simulated annealing
  #{first => graph(N),
    next => fun graph_next/2}.  % see Fig. 2

graph(N) ->       % a simple, general generator for graphs of size N
  Vs = lists:seq(1, N),
  ?LET(Es, proper_types:list(edge(Vs)), {Vs, lists:usort(Es)}).

edge(Vs) ->      % auxiliary generator for edges of the graph
  ?SUCHTHAT({N1, N2}, {oneof(Vs), oneof(Vs)}, N1 < N2).
```

Fig. 1. Targeted property for graphs of a certain size and its generators.

to write and general generator for random graphs of a certain number of nodes $N$ (`graph(N)`) to obtain a random first input and the function `graph_next()`, shown in Fig. 2, as NF. With these ingredients, such a targeted property can find a failing input after only a few thousand tests and in all runs, whereas random PBT is unable to find a counterexample even with hundreds of thousands of tests using the same random graph generator [3].

## C. Simulated Annealing

Simulated Annealing (SA) is a well-studied local search meta-heuristic [6]–[8] that can be used to address discrete and continuous optimization problems. The key feature of SA is a mechanism that enables escaping local optima by accepting search steps to worse solutions in the hope to find a global optimum. SA also has another nice property, namely that it does not depend on the type of data it is operating on. This allows us to apply SA as a strategy to most types of input.

SA operates on the input space $I$ (the set of all possible inputs) and a fitness function $F : I \rightarrow f$. The fitness of an input is equivalent to its utility value. Furthermore, SA defines a NF $n$ that produces random input in the neighborhood $\mathcal{N}$ of a given base input $b$. $n$ is also dependent on the temperature $t$:

$$n(b,t) = v \text{ where } v \sim U(\mathcal{N})$$

SA starts with random initial input from the input space. It then produces a neighboring input and accepts it as new input if its associated fitness is higher than that of the current input. It also accepts worse inputs with a probability that is dependent on the current temperature $t$. The higher the temperature, the higher the probability that a worse solution is accepted. The *acceptance probability* is defined as follows:

$$\Pr_{\text{accept}}(i_{n+1}, t_{n+1}) = \begin{cases} e^{-\frac{(u_n - u_{n+1})}{t_{n+1}}} & \text{if } u_n > u_{n+1} \\ 1 & \text{otherwise} \end{cases}$$

When choosing SA as search strategy, we need to provide a generator for $I$ and a NF for the input space we want to use. It is possible to define this generator and NF using PROPER's generator language and pass them as parameters to

```
graph_next(G, _T) ->
  Size = graph_size(G),
  ?LET(NewSize, neighboring_integer(Size),
       ?LET(Additional, neighboring_integer(Size div 10),
            begin
              {Removals, Additions} =
                case NewSize < Size of
                  true -> {Additional + (Size - NewSize), Additional};
                  false -> {Additional, Additional + (NewSize - Size)}
                end,
              ?LET(G_Del, remove_n_edges(G, Removals),
                   add_n_edges(G_Del, Additions))
            end)).

graph_size({_, E}) ->
  length(E).

%% generator for neighboring integer
neighboring_integer(Base) ->
  Offset = trunc(0.05 * Base) + 1,
  ?LET(X, proper_types:integer(Base - Offset, Base + Offset), max(0,X)).

add_n_edges({V, E}, N) ->
  ?LET(NewEdges, proper_types:vector(N, edge(V)),
       {V, lists:usort(E ++ NewEdges)}).

remove_n_edges({V, E}, 0) -> {V, E};
remove_n_edges({V, []}, _) -> {V, []};
remove_n_edges({V, E}, N) ->
  ?LET(Edge, proper_types:oneof(E),
       ?LAZY(remove_n_edges({V, lists:delete(Edge, E)}, N - 1))).
```

Fig. 2. Manually-written neighborhood function for graphs of a certain size.

the `?TARGET` macro. For property `prop_length()`, the initial element is sampled from the `graph(N)` generator, and the manually-written neighborhood function `graph_next()` is as shown in Fig. 2. To produce neighboring graphs, the code first decides on a new graph size and then removes and adds a random number of edges that achieves the new graph size. A NF for SA as implemented by PROPER has as first argument the *base value* from which a neighbor value will be generated. The second argument is the current temperature of the SA algorithm.

Notice that the neighborhood function of Fig. 2 is quite simplistic, as it completely ignores its temperature argument. Still, its code is about six times as long and, arguably, significantly more complicated than the code for `graph(N)`. A random generator produces the whole input data at once while a neighborhood function usually has to select some part of the input, alter it slightly, and put everything back together while preserving all constraints and invariants of the input. This process can become complicated especially with more complex input domains. Once written, however, a NF is mostly independent from the property to test and can be used for all properties that have the same type of input.

Requiring the programmer to provide a NF for each generator makes TPBT hard to use and therefore less appealing. Not only need the users know how to use PROPER's language to write custom generators, but they also need to understand how SA works and how to produce random neighboring inputs for their tests. It also makes migrating properties to targeted ones harder. In the next section, we will introduce a technique that constructs a NF for graphs from a generator like `graph(N)`.

## III. Using Constructed Neighborhood Functions

Our aim is to reduce the additional tasks to use TPBT to, ideally, only specifying the utility values and whether simulated annealing —or some other search strategy that requires a neighborhood function— should maximize or minimize them. To achieve this, the main ingredient we need is a technique that can construct a NF automatically. We will present such a technique in the next section. Let us first see how one could use a constructed NF in targeted properties.

In our tool, the simplest way to specify that we want to transform a random generator into a NF is to simply mention the generator we want to use wrapped in a map with a `gen` key when defining the targeted property. Using `prop_length()` as an example, the code becomes:

```
prop_length() ->
  ?FORALL_SA(G, ?TARGET(#{gen => graph(42)}), % the only change is here
             begin
               UV = lists:max(distance_from_sink(G)),
               ?MAXIMIZE(UV),
               UV < 21
             end).
```

Of course, we still need to provide the `graph(N)` generator, whose code however is very simple (Fig. 1), but we do not need to supply the complicated code of Fig. 2. PropEr takes the generator definition and constructs a `graph_nf(G,T)` function that produces neighboring graphs according to the specifications in `graph(N)`. The random generator that was used as input for the construction is utilized by SA to obtain an initial input. Note that the constructed `graph_nf()` takes the temperature into account, in contrast to the hand-written `graph_next()` of Fig. 2 which ignored that argument.

On the other hand, the constructed NF is a general one and can be inferior to a carefully crafted hand-written one that is fitted to the input domain. What is important, however, is that the automatically obtained NF performs reasonably well so that SA can work as intended. If we test the above `prop_length()` property now with the constructed NF we find a counterexample after $4,060$ tests on average (measured over $100$ runs with a maximum of $100,000$ tests). While this is worse than the result achieved with the hand-written NF ($1,548$ tests on average) the performance we get is sufficient for finding a counterexample in a reasonable time and, most importantly, in all runs. Recall that random PBT was not able to find a counterexample at all for this property using the same setting.

The effort needed to use TPBT in this example is reduced significantly. Instead of writing 30 lines of complicated code for the neighborhood function, a user only needs to specify which generator to use. We mentioned earlier that it would be possible to also find a counterexample with random PBT if we were to write a more complicated random generator. Using TPBT and an automatically constructed NF allows us to use a simpler and easier to understand generator instead.

In some cases, the user might have special knowledge about the input domain that she wants to use when implementing the NF. However, writing the whole NF by hand can be intricate, because it is necessary to implement the neighborhood relation for all parts of the generator. The task becomes even more complicated if the temperature should be taken into account when producing the next input. Therefore, our tool provides an interface to inspect the constructed NF so that it can be further refined and/or used as part of a hand-written NF.

In Fig. 2, the code of `graph_next()` uses the function `neighboring_integer()` to produce non-negative integers neighboring a given base integer. We can translate the built-in generator `non_neg_integer()` of PropEr and use it instead of the hand-written function as follows:

```
graph_next(G, T) ->
  Size = graph_size(G),
  NfInt = proper_sa:get_neighborhood_function(non_neg_integer()),
  ?LET(NewSize, NfInt(Size, T),
       ?LET(Additional, NfInt(Size div 10, T),
            begin
            ...
            end).
```

As a positive side effect, we can now scale the size of the neighborhood for the integers with the temperature in the NF.

Alternatively, it is also possible to adjust the construction process by overwriting the construction rules for some of the inner generators (e.g., the element generator of a list) with a user-defined one. This is done by annotating a random generator with a hand-written NF. If the constructor needs to build a NF for the annotated generator, the manually written one is used instead, which allows us to change the behavior for parts of the resulting NF. Let us assume we want to write a NF for lists of integers where the elements should only change by $\pm 1$ during each search step. We can easily write a NF for integers that behaves in that way:

```
integer_pm_one(B, _) ->
  oneof([B+1, B-1]).
```

Writing by hand a NF for the list part of the input is more complicated, and we want to use the constructor instead. We use a random generator for lists of integers as basis and, using the `?USERNF` macro, we annotate the element generator with our manually written NF `integer_pm_one()`. We then use the constructor to produce a NF with the desired behavior:

```
integer_list() ->
  list(?USERNF(integer(), fun integer_pm_one/2)).

prop_sth() ->
  ?FORALL_SA(L, ?TARGET(#{gen => integer_list()}), ...).
```

The property `prop_sth()` passes the annotated random generator `integer_list()` to the constructor to build the NF. The resulting `integer_list_nf()`, which is constructed in memory, will produce neighboring integer lists with the property that if list elements are modified, they are passed to the manually written `integer_pm_one()` that offsets them by $\pm 1$. The ability to adjust the construction to the requirements of the input type provides the user with additional control and most of the time makes it unnecessary to write the entire NF by hand.

## IV. Construction Algorithm

In this section, we present the algorithm for constructing a NF from a PropEr generator for input of the same type.

**Algorithm 1:** Top-level NF construction algorithm.

```
$construct\_nf(g)$:
begin
    $type \leftarrow get\_type(g)$;
    $nf_{raw} \leftarrow n(b,t)$ for $type$ according to Tables I and II;
    $nf \leftarrow apply\_constraints(nf_{raw}, g)$;
    return $nf$;
end
```

The basic idea of the algorithm is to reenact the decisions made by the generator while generating an input value. Instead of deciding which value a variable should hold randomly, we choose values in the neighborhood of the previously generated value, called the *base value*, of that variable. By doing so for all random variables of the generator, we assume that the resulting value will be in the neighborhood of the base one.

PROPER comes with a set of built-in generators with which all generators including user-defined ones are built. During construction, we traverse the generator definitions and build a NF where the random decisions of these built-in generators are replaced by general NFs for the respective generator type.

If a custom generator has constraints (as defined by `?SUCHTHAT`) then these are checked after a neighboring candidate has been produced. If all constraints are fulfilled then the candidate is returned as new input. Otherwise, the constructed NF is used to produce another neighboring input from the base value. If a valid neighboring solution cannot be produced within a certain number of attempts, a random element is generated from the originating random generator instead. The top-level construction algorithm is given in Algorithm 1.

For most built-in generators and combinations thereof a NF can be constructed statically for the whole generator at once. This is often not possible for more complicated user-defined generators (e.g., generators that use `?LET`) since values that are instantiated early in the generation process might influence the structure of the generators dynamically.

In Table I, we list the translation rules for most of PROPER's built-in "basic" generators; some additional rules appear in Table II. The constructed NF is written as $n(b, t)$ where $b$ is the base input and $t$ the current temperature. We define a NF that is obtained from a generator $g$ as $construct\_nf(g)$. We use the notation $v \sim U(Set)$ to uniformly sample a value $v$ from $Set$ and $x \sim g$ to sample a value $x$ from generator $g$.

As an example, we define the neighborhood of numeric values as an interval that is 10% the size of the total sample space around the base value. For structural types, the algorithm decides for each sub-structure if it stays unchanged or if it will be modified. For lists that can change their length, elements can additionally be deleted or inserted. If an element is chosen to be modified, then it is exchanged to one in the neighborhood of the current element, depending on the element's generator.

The so constructed NF scales the size of the neighborhood from which the next element gets selected by the temperature. This means that under low temperature the neighborhood is smaller than under high temperature. The idea is that in the

TABLE I
ALGORITHMS FOR CONSTRUCTING NFS FOR THE MAIN TYPES.

$\boxed{n(b,t) \text{ for atom()}}$ — the algorithm for `binary()` is similar

```
$g_{chars} \leftarrow$ list(integer(0, 255));
$nf_{chars} \leftarrow construct\_nf(g_{chars})$;
$next_{chars} \leftarrow nf_{chars}$(atom_to_list($b$), $t$);
return list_to_atom($next_{chars}$);
```

$\boxed{n(b,t) \text{ for exactly}(Value)}$

```
return $Value$;
```

$\boxed{n(b,t) \text{ for integer}(l,\ h)}$

```
$r \leftarrow (h - l) * 0.05 * t$;
$o \sim U([-r, r])$;
return max($l$, min($\lfloor b + o \rfloor$, $h$));
```

$\boxed{n(b,t) \text{ for list}(g_{element})}$

```
$growthCoeff \sim U([0.1, 0.9])$;
$nf_{element} \leftarrow construct\_nf(g_{element})$;
foreach element $e$ in $b$ do
    $operation \sim$ List.getOperation($growthCoeff$, $t$);
    switch $operation$ do
        case $add$ do
            $v \sim g_{element}$;
            insert $v$ after $e$ in $b$
        case $del$ do
            delete $e$ from $b$
        case $modify$ do
            $next_e \leftarrow nf_{element}(e, t)$;
            exchange $e$ with $next_e$ in $b$
        otherwise do
            -
        end
    end
end
return updated $b$;
```

$\boxed{n(b,t) \text{ for tuple}([g_1, g_2, \ldots])}$

```
foreach element $e_i$ in $b$ do
    $operation \sim$ Tuple.getOperation($t$);
    switch $operation$ do
        case $modify$ do
            $nf_i = construct\_nf(g_i)$;
            $Pnext_{e_i} \leftarrow nf_i(e_i, t)$;
            exchange $e_i$ with $next_{e_i}$ in $b$
        otherwise do
            -
        end
    end
end
return updated $b$;
```

$\boxed{n(b,t) \text{ for union}([g_1, g_2, \ldots])}$

```
$operation \sim$ getOperation($t$);
switch $operation$ do
    case $change$ do
        $g \sim U([g_1, g_2, \ldots])$;
        $next_g \sim g$;
        return $next_g$;
    case $modify$ do
        $Gs \leftarrow$ all $g_i$ where $b$ is an instance;
        $g \sim U(Gs)$;
        $nf_g \leftarrow construct\_nf(g)$;
        return $nf_g(b, t)$;
    otherwise do
        return $b$;
    end
end
```

TABLE II
MORE ALGORITHMS FOR CONSTRUCTING NEIGHBORHOOD FUNCTIONS.

```
n(b,t) for float(l, h)

$r \leftarrow (h - l) * 0.05 * t$;
$o \sim U([-r, r])$;
return max(l, min(b + o, h));
```

```
n(b,t) for fixed_list([$t_1, t_2, \ldots$]) — see tuple([$t_1, t_2, \ldots$])
```

```
n(b,t) for vector(n, element_type)

  — similar to list(element_type) but without add and del cases
```

```
n(b,t) for ?USERNF(g, $nf_{user}$)

$retval \leftarrow nf_{user}(b, t)$;
return $match(b, retval, t)$;
```

```
n(b,t) for g = ?LET($x, g_{in}, f_{in}$)

$nf' \leftarrow construct\_nf(g_{in})$;
if (($prev_x \leftarrow cache\_lookup(\{g, b\})$) exists) then
    $next_x \leftarrow nf'(prev_x, t)$;
else
    $next_x \sim g_{in}$;
end
$next \leftarrow match(b, f_{in}(next_x), t)$;
$cache\_store(\langle g, next\rangle, next_x)$;
return $next$;
```

**Algorithm 2:** Match the base value $b$ against the intermediate result $m$ with temperature $t$.

```
$match(b, m, t)$:
begin
    switch m do
        case m is just a generator do
            $nf_m \leftarrow construct\_nf(m)$;
            $b' \leftarrow nf_m(b, zero)$;
            return $nf_m(b', t)$;
        case m and b are lists of same length do
            foreach $e_m$ and corresponding $e_b$ in m and b do
                replace $e_m$ with $match(e_b, e_m, t)$;
            end
            return updated $m$;
        case m and b are tuples do
            /* similar to the list case */
        otherwise do
            return $\sim m$; /* return a sample from m */
        end
    end
end
```

beginning of the testing it is easier to explore larger parts of the input space if the neighbors are further apart from each other. In contrast, towards the end of the testing, smaller steps can help narrowing down good input more efficiently [9].

### *Caching and Matching*

User-defined generators that use a ?LET($x, g_{in}, f_{in}$) construct are comprised of an inner generator $g_{in}$ which produces a random value for $x$, and a function $f_{in}$ that constructs the value generated by the entire ?LET. Following the idea of the construction algorithm, we construct a NF so that $g_{in}$ produces a value in the neighborhood of the previous inner value and then applies $f_{in}$ afterward. The last entry of Table II shows the general NF our construction algorithm uses for ?LET generators.

In general, the $f_{in}$ function of a ?LET is not reversible. This means that it is not possible to calculate the inner value that was used to construct a specific value by using $f_{in}$. This inner base value is however needed to generate a value in its neighborhood for the next generation round. Additionally, a constructed generator can generate three different types of results when evaluating $f_{in}$: an immediate value, a generator, or a mix of both.

We solve the first issue partially by caching the constructed inner values. After each construction, we store that the generator $g$ produced the combined value $v$ with the generated inner value. We do this by using the pair $\langle g, v\rangle$ as key to cache the inner value. During the next iteration we try to restore this inner value. This time we use as key the generator and the base value (which is the previous combined value). If a cached inner value exists, we use it as the base value for the internal generator. If no cached value is found, we generate a new random element instead.

We approach the second issue by structurally matching the prior value to the return value of the $f_{in}$ function: an immediate value is not matched and used directly as new value of the generator; if the return value is only a generator then it is matched against the whole prior value; if the return value and the prior value is a list then the list members are matched against each other if the lengths of the lists are the same. The same applies to tuples. If at some point the algorithm cannot match a prior value against the newly constructed value $v_{new}$ anymore, then $v_{new}$ is used as new value. The matching algorithm is shown in Algorithm 2.

Matching allows us to construct neighborhood functions for nested ?LET generators where the $f_{in}$ function uses other generators. Without matching, the process would stop after resolving the first ?LET construction and a new random value would be generated at this point. This means that in such cases only the values from the inner generator of the first ?LET would be modified by the neighborhood function.

We will demonstrate how matching works in practice on a variation of the graph(N) generator that generates graphs with a variable number of vertices:

```
graph_duplicated_edges() ->
  ?LET(Vn, integer(2, 42),
       begin
         Vs = lists:seq(1, Vn),
         {Vs, list(edge(Vs))}
       end).
```

After generating a Vn value, this generator returns a tuple containing the list of vertices and a basic generator for the list of edges. During random PBT, PROPER would resolve this return value by generating a fresh list of edges for every test. The constructed NF retrieves the base value of Vn from the cache and constructs a NF from the generator integer(2, 42). This inner NF is used to produce the next Vn in the neighborhood of the cached previous `Vn`. The $f_{in}$ function then returns a tuple containing a list of vertices as immediate value and a generator for the list of edges. The constructed NF then matches the previous list of edges to the generator for the new list of edges. `list(edge(Vs))` is a basic generator and the neighborhood relation can be resolved with the matched previous value. Matching tries to optimistically match the base value against the intermediate values of the generation. Because of this, some base values may be matched that are not valid anymore. In our `graph_duplicate_edges()` generator, it can for example happen that some of the edges in the base value for the list of edges are not valid edges anymore because the set of vertices got smaller. Therefore it is necessary to check that the elements of the list are valid according to their generator.

To preserve the integrity in such cases, we replace during matching each list element with a neighbor that is constructed with a special *zero* temperature. At *zero* temperature, the NF does not alter the base value if it is a valid input value. If the input value is not valid, a new random input value is generated. The advantage of using the constructed NF is that the base value is traversed in the same way as if every element would be modified. Only the part of the base value that is invalid is constructed again. In our example, this results in more similar list elements than if we would only check whether whole elements are valid input and generate new elements if not.

Matching and caching do not always work and it is important to write more involved generators with the limitations of the approaches in mind. Our `graph_duplicated_edges()` generator does not filter duplicates from the list of edges. The following generator adds such filtering but also prevents that matching and caching work appropriately:

```
graph_no_caching() ->
  ?LET(Vn, integer(2, 42),
       begin
         Vs = lists:seq(1, Vn),
         ?LET(Es, list(simple_edge(Vs)), {Vs, lists:usort(Es)})
       end).
```

For this generator, the constructed NF will generate a `Vn` that is in the neighborhood of the previous one. The list of edges will be generated randomly for each generated input value because matching and caching does not work properly in this case. This happens because: (1) The first `?LET` returns the inner `?LET` generator (the second `?LET`) which is matched with the whole base value. In the next step, the second `?LET` generates an immediate value which requires no matching since the generation process is finished at this point. (2) Since `Vn` differs in each generation, the second `?LET` is parameterized differently and effectively constitutes a new generator. This means that this part of the neighborhood function is constructed anew for each run and we cannot retrieve the base value for the list of edges, meaning that the only possible action left is to generate a new list of edges. It is however possible to modify the `graph_no_caching()` generator so that it contains no duplicate edges. By reversing the order of the `?LET` constructs as follows matching and caching work again:

```
graph() ->
  ?LET({Vs, Es}, graph_duplicated_edges(), {Vs, lists:usort(Es)}).
```

In this example, caching works since both generators are static and do not depend on values that are generated, and matching works as in the `graph_duplicated_edges()` example. Matching requires that the base value and the intermediate are structurally compatible. If the structure of the values that require matching is depending on values that are generated earlier then the matching might not work correctly like in the following example:

```
graph_no_matching() ->
  ?LET({Vs, Es}, graph_duplicated_edges2(), {Vs, lists:usort(Es)}).

graph_duplicated_edges2() ->
  ?LET({Vn, En}, {integer(2, 42), integer(0, inf)},
       begin
         Vs = lists:seq(1, Vn),
         {Vs, edges(Vs, En)}
       end).

edges(_V, 0) -> [];
edges(V, N) -> [edge(V)|edges(V, N-1)].
```

The list of edges is now generated as a list of generators, one for each edge. The length of this list is still in the neighborhood of the previous list but, since the length can change, the matching can no longer match the previous list of edges to the list of generators since only lists of equal length are matched. The three generators `graph()`, `graph_no_caching()`, and `graph_no_matching()` define the same type of input with a similar distribution of values. The construction algorithm however produces neiborhood functions with quite different quality.

The restrictions of the NF constructor are also stemming from the assumptions of the algorithm. We assume that if we slightly change the decisions for the values we make during generation we end up with some input value that is similar to the previous value. The set of value decisions in user-defined generators can either be static or dynamic, depending on early decisions. In the later case, this means that we cannot match completely the previously generated value to the available decisions because their type can differ from the ones that were made earlier.

## V. Evaluation

The quality of the neighborhood function is a key factor for the effectiveness of the simulated annealing search strategy. Therefore, in this section, we compare how the hand-written and tuned NFs of our previous work in the TPBT paper [3] perform against automatically constructed ones with respect to testing effectiveness and performance. Furthermore, we try to quantify the trade-offs between ease of use and testing performance in a small user study.

### A. Energy Efficiency of MAC Protocols

The first case study concerns testing a property about the energy efficiency of network configurations using different implementations of MAC protocols [10]. The setup of the test experiment [3] generates sensor networks consisting of UDP server and client nodes where the client nodes periodically send messages to the server node. The experiment then records the duty cycle for each node and checks that no node has a

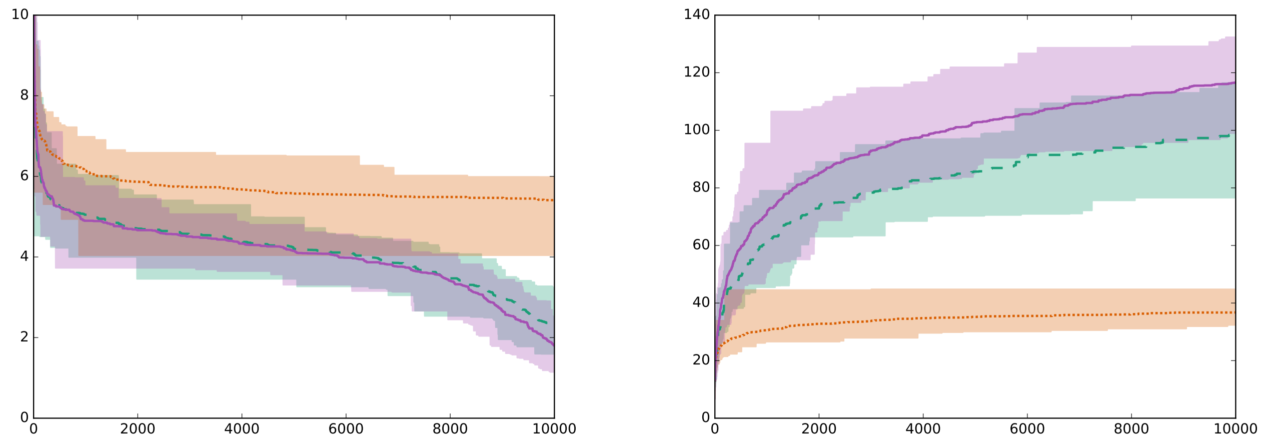

Fig. 3. The y-axis shows a metric of the energy consumption over tests (x-axis). The graph shows the minimum, median, and maximum of random PBT (orange, dots) in comparison to targeted PBT with a hand-written (green, lines) and a generated NF (blue, solid) of the achieved minimums (left graph) and maximums (right graph) of 100 runs with $10,000$ tests.

duty cycle above 25%. (A more detailed description of the setup is given in [3].)

The input to the targeted property is a network topology with a variable number of sensor nodes and links between them. The hand-written NF alters in each step the nodes and links of the topology (scaled by the temperature) and is tuned to work well with the tested property. With this hand-written NF the time to find a counterexample is on average 2h12m (down from 7h46m using random PBT). However, the hand-written NF requires around 100 lines of complex code.

We exchanged the hand-written NF with an automatically constructed one using the `graph()` generator from Section IV and tested the otherwise unchanged property. On average the property failed after 2h19m, which is very similar to the time achieved by the hand-written neighborhood function, showing that the automatic construction of NFs is quite good in this case.

## B. Routing Trees for Directional Antennas

The second case study [3] showed that TPBT can guide the input generation for complex data structures like routing trees for directional antennas towards configurations with a high or low estimated energy consumption. The routing trees for this type of antennas are special in the sense that, besides needing to generate links that connect nodes of the network, we also need to generate sending and receiving directions for each link.

The experimental setup [3] used a recursive random generator and a complicated hand-written NF. This random generator produces a random tree with directions, recursively choosing one link at a time. The NF alters the tree by moving subtrees to different parents. Because of the limitations of our construction algorithm, it is hard to obtain good performance when using this recursive generator as the basis for an automatically constructed NF.

We therefore implemented a different random generator based on random minimum spanning trees [11]. This generator produces random sending directions and a random weight for each possible link. A spanning tree with minimum weights is then deterministically calculated from these weights. We implemented a hand-written NF that switches the weights on a subset of all possible links. This alters the order in which the links get chosen when building the minimum spanning tree. After changing the weights, the hand-written NF generates new sending and receiving directions for some of the links.

In Fig. 3 we plot the achieved minimum, median, and maximum of the estimated energy consumption when instructing the search strategy to optimize it in either direction. The two graphs show the median line and the range for all runs illustrating the worst case and the best case. We also show how random PBT performs compared to targeted PBT.

Targeted PBT outperforms random PBT with either NF. After less than $1,000$ tests, the worst case run of TPBT produces inputs with a higher estimated energy consumption than the best case of random PBT when instructed to maximize the measure (right graph). The constructed NF performs even slightly better than the hand-written one. When minimizing (left graph), both neighborhood functions have very similar performance.

This experiment shows that the automatically constructed neighborhood functions work very well when they are based on the right generators. The random generator that we use as input to the construction is important for the quality of the resulting NF and the testing performance. We were not able to use the original recursive generator and had to come up with one that generates routing trees in a non-recursive way. While it took some work to implement a new generator that is better suited for our construction algorithm, the effort of writing a NF by hand is much bigger. The new non-recursive generator is only 10 lines, while the corresponding hand-written NF requires around 40 lines of code.

TABLE III
AVERAGE TIMES (IN MSECS) TO FIND A COUNTEREXAMPLE FOR BUGS INJECTED TO THE STACK MACHINE OF HRIŢCU *et al.* [12].

| | PBT | Targeted PBT | |
|---|---|---|---|
| | *Random* | *Handwritten* | *Constructed* |
| ADD | 5800.57 | 274.68 | 489.93 |
| PUSH | 338.90 | 3.72 | 2.74 |
| LOAD | 7764.15 | 341.30 | 447.25 |
| STORE A | 16997.81 | 2634.80 | 3685.32 |
| STORE B | 341.02 | 5.51 | 3.84 |
| STORE C | 336.45 | 4.60 | 3.29 |
| *MTTF (arithmetic)* | 5263.15 | 544.10 | 772.06 |
| *MTTF (geometric)* | 1760.45 | 53.44 | 55.10 |
| Average tests per second | 9479 | 7431 | 2595 |

## C. Noninterference

Hriţcu *et al.* [12] explored how random PBT can be employed to quickly test formal specifications for mistakes and help in the design of secure information-flow control (IFC) abstract machines. The third case study of the TPBT paper [3] showed that targeted PBT can be used to efficiently test these properties by guiding the generation towards long-running programs. Long-running programs discover more interesting states in the abstract machines and have a higher chance of unveiling errors in their definitions. Here, we want to evaluate how well a constructed NF performs in finding the injected bugs of the IFC machine and compare its effectiveness against the hand-written NF and random PBT.

Our baseline for random PBT is the SEQUENCE generator [12]. This generator produces a random list of instructions that get chosen with a weighted distribution; e.g., it is more likely that a `push` instruction gets generated than an `add` or a `noop`. Additionally, sequences of instructions that make sense together (e.g., a `push` followed by a `load`) are produced.

The hand-written NF produces a new list of instructions by removing some old instructions from the previous list and adding some new ones. Existing instructions are not modified, as we found out that these alterations do not lead to longer sequences of executed instruction fast enough.

In Table III we list the average times it took for each technique to find each of the six injected bugs. As expected from our previous experiment [3], random generation of the programs is slowest in finding counterexamples. TPBT with the hand-written NF performs best in this experiment. However, the automatically constructed NF is quite competitive to it and achieves acceptable performance in all cases.

It should be noted that Hriţcu *et al.* [12] also present a more sophisticated BYEXEC generation strategy in which the instruction list is generated one instruction at a time such that the newly added instruction may not crash the IFC machine. This strategy, both when using random and targeted PBT, finds all injected bugs faster [3]. However, it also requires more effort to implement than the generator we use here.

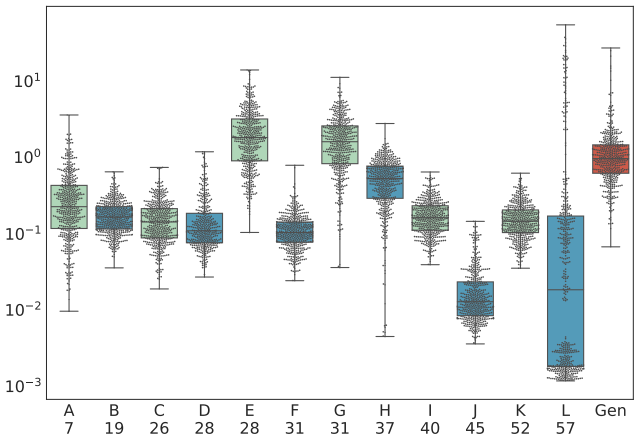

Fig. 4. The graph plots, for each neighborhood function (labeled A–L), the times (in seconds) required to find a counterexample for the property (dots) and their distribution (boxes), repeating the experiment 500 times. The numbers below the labels are the lines of code for each hand-written implementation. The rightmost entry (Gen) corresponds to the generated neighborhood function.

## D. Small User Study

The automated construction of the neighborhood function reduces the manual work required from the user and thus makes targeted PBT more accessible and easier to use. Still, this comes with a possible downgrade in testing performance since the quality of the NF is a critical component of the search strategy. To study the reduction in programming effort and its effects on performance, we conducted a small user study with a group of M.Sc. students from an advanced functional programming course at Uppsala University in the fall of 2017. The students were asked to implement NFs for testing a targeted property. We then analyzed the techniques that the NFs used to generate a neighbor and their overall testing performance. Before taking part in this study, all students were familiar with the concepts of random and targeted PBT. (A previous assignment contained exercises on random PBT.)

The property that we provided the students with tested the spell system component of an imaginary role-playing game for an exploit that would grant the player too much power and is therefore considered a bug. We also supplied the implementation of the spell system and the targeted property. The students were then asked to implement a NF by hand so that the property fails consistently and were given ten days for handing in their submissions to this and the rest of the assignment. For reference, we have put the complete description of the assignment and the associated source code online [13].

We received twelve solutions for this task. It is noteworthy that every student who submitted a NF as solution solved the problem and was able to find a combination of spells that falsified the given property.

To evaluate the performance of the hand-written NF compared to the constructed NF we tested the property 500 times with each NF and recorded the times required to find a counterexample. Figure 4 plots these times for each NF

TABLE IV
STRATEGIES USED BY THE PARTICIPANTS OF THE USER STUDY FOR IMPLEMENTING THE NEIGHBORHOOD FUNCTION.

| Strategy | Description |
|---|---|
| *random expansion* | add new spells at random locations |
| *random reduction* | remove spells from random locations |
| *strategic expansion* | add new spells at "well-chosen" locations |
| *strategic reduction* | remove spells from "well-chosen" locations |
| *shuffling* | change the order of the spells |
| *modification in-place* | alter spells in-place |
| *restart* | discard the current input and generate a new random input |
| *spell selection* | select in a more sophisticated way the spells that are added, removed or modified |

combined with their distribution. A well-implemented NF that is tuned to the problem can achieve a testing performance that is one to two orders of magnitude better than the constructed NF. This difference in performance is expected. The constructed NF is designed to be applicable regardless of the property that is tested. By writing the NF by hand, one can make much stronger assumptions about the solution space, which can then be exploited in the NF. For example, the fastest solving NF (L) contains a set of configuration parameters that were fine-tuned to fit the given problem. Even though most hand-written NFs are faster than the constructed one, we can observe that some hand-written NFs have similar or worse performance, and that the variance of the hand-written NFs is much higher than the one for the constructed NF. This shows that implementing a neigborhood function is not always straightforward and can pose an obstacle to using targeted PBT.

The number of lines of code required for the implementation of the neighborhood functions ranged from 7 to 57. To further analyze their complexity, we identified a total of eight classes of neighbor selection strategies that were used to generate the next neighbor. The names of these strategies and a short description of them appears in Table IV. In all student submissions, multiple strategies were combined. We could identify that at least *random expansion* or *random reduction* strategies were necessary to achieve good results in terms of testing performance. Additionally, we observed that the intensity of the modifications done to generate a neighbor affected the overall performance. Finally, many NFs tuned the number of reductions, expansions, and alterations made to the base value in each generation step.

In a nutshell, this small user study shows that hand-written and fine-tuned implementations of neighborhood functions that are fitted to the property outperform automatically constructed ones. But it also indicates that it is not always straightforward to do so. Implementing a good neighborhood function can require considerable effort. In contrast, a constructed NF provides a baseline that is available immediately and can be adjusted further if needed.

## VI. Related Work

Our work builds upon the paper that introduced targeted property-based testing [3]. Our implementation is on top of the open-source QuickCheck-inspired testing tool PropEr [4]. The component that automatically constructs neighborhood functions from generators is already integrated in the tool.

The QuickCheck library for Haskell by Claessen and Hughes [2] pioneered the idea of using a high-level language for writing properties and generators for property-based testing. Property-based testing has since been applied to a variety of applicationareas [10], [14], [15].

The idea of using search-based techniques for software testing was put forward by Miller and Spooner [16] in 1976. Since then, it has been applied to various testing areas; cf. some survey articles on the subject [17]–[19]. In particular, we mention two testing tools (EvoSuite [20] and Randoop [21]) that use such techniques, and two interesting recent uses [22], [23] of search-based testing.

Languages for test data generation that provide alternative ways of defining random generators (e.g., through predicates as in Luck [24] or with non-deterministic choices as in UDITA [25])) also exist.

The idea to construct a function or a program from a high-level specification is of course not new, and the field of program synthesis has been an active area of research since the 1970s [26] and even more so recently. It applies techniques from constraint programming, machine learning, and stochastic search. The interested reader is referred to the recent article by Gulwani *et al.* [27] for an overview of the field. However, to the best of our knowledge, our work is the first one that constructs a neighborhood function for simulated annealing from a random generator or another high-level description of the input/solution space.

## VII. Concluding Remarks

We presented a technique for targeted property-based testing based on simulated annealing that constructs a neighborhood function automatically from a random generator of inputs. We demonstrated that, by doing so, the effort of using targeted PBT is reduced significantly making it almost as easy as its random counterpart. We furthermore showed that the efficiency of simulated annealing with these neighborhood functions is most of the time sufficient and in some cases competitive to hand-written ones.

As future work, it would be interesting to conduct larger case studies from real applications, and also investigate how ingredients that search strategies for targeted PBT other than simulated annealing can also be generated automatically.

## Acknowledgements

This research has been partially supported by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (Vetenskapsrådet). We thank Simon Olofsson for his help in the second case study (Section V-B) and the students that participated in the user study of Section V-D.

## References

[1] "QuickCheck," 2016. [Online]. Available: https://en.wikipedia.org/wiki/QuickCheck

[2] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM, 2000, pp. 268–279. [Online]. Available: http://doi.acm.org/10.1145/1988042.1988046

[3] A. Löscher and K. Sagonas, "Targeted property-based testing," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 46–56. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092711

[4] M. Papadakis and K. Sagonas, "A PropEr integration of types and function specifications with property-based testing," in *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*. New York, NY, USA: ACM, 2011, pp. 39–50. [Online]. Available: http://doi.acm.org/10.1145/2034654.2034663

[5] J.-l. Gailly and M. Adler, "zlib compression library," 2017. [Online]. Available: http://www.zlib.net/

[6] E. Aarts and J. Korst, *Simulated annealing and boltzmann machines*. New York, NY; John Wiley and Sons Inc., Jan 1988.

[7] D. Henderson, S. H. Jacobson, and A. W. Johnson, *The Theory and Practice of Simulated Annealing*. Boston, MA: Springer US, 2003, pp. 287–319. [Online]. Available: https://doi.org/10.1007/0-306-48056-5_10

[8] K. A. Dowsland and J. M. Thompson, *Simulated Annealing*. Berlin, Heidelberg: Springer, 2012, pp. 1623–1655. [Online]. Available: https://doi.org/10.1007/978-3-540-92910-9_49

[9] X. Yao, "Dynamic neighbourhood size in simulated annealing," in *Proc. of Int'l Joint Conf. on Neural Networks (IJCNN'92)*. IEEE Press, 1992, pp. 411–416.

[10] A. Löscher, K. Sagonas, and T. Voigt, "Property-based testing of sensor networks," in *Sensing, Communication, and Networking, 12th Annual IEEE International Conference on*. IEEE, Jun. 2015, pp. 100–108. [Online]. Available: https://doi.org/10.1109/SAHCN.2015.7338296

[11] A. Frieze, "On the value of a random minimum spanning tree problem," *Discrete Applied Mathematics*, vol. 10, no. 1, pp. 47 – 56, 1985. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0166218X85900587

[12] C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky, A. A. D. Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis, "Testing noninterference, quickly," *Journal of Functional Programming*, vol. 26, p. e4, 2016. [Online]. Available: https://www.cambridge.org/core/article/div-class-title-testing-noninterference-quickly-div/00D792464D8E0CAD7F6B30417F121729

[13] A. Löscher and K. Sagonas, "Targeted property-based testing assignment," 2018. [Online]. Available: https://gist.github.com/TheGeorge/8f3f2b45d501b1d3dd657f57c576b74b

[14] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with Quviq QuickCheck," in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. New York, NY, USA: ACM, 2006, pp. 2–10. [Online]. Available: http://doi.acm.org/10.1145/1159789.1159792

[15] L. Lampropoulos and K. Sagonas, "Automatic WSDL-guided test case generation for PropEr testing of web services," in *Proceedings 8th International Workshop on Automated Specification and Verification of Web Systems*, 2012, pp. 3–16. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.98.3

[16] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Trans. Software Eng.*, vol. 2, no. 3, pp. 223–226, Sep. 1976. [Online]. Available: https://doi.org/10.1109/TSE.1976.233818

[17] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584908001833

[18] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, Dec. 2012. [Online]. Available: http://doi.acm.org/10.1145/2379776.2379787

[19] P. McMinn, "Search-based software test data generation: A survey," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1002/stvr.v14:2

[20] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *Proceedings of the 11th International Conference on Quality Software*, ser. QSIC '11. Washington, DC, USA: IEEE Computer Society, Jul. 2011, pp. 31–40. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2011.19

[21] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.37

[22] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "Combining symbolic execution and search-based testing for programs with complex heap inputs," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 90–101. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092715

[23] S. Mahajan, A. Alameer, P. McMinn, and W. G. J. Halfond, "Automated repair of layout cross browser issues using search-based techniques," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 249–260. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092726

[24] L. Lampropoulos, B. C. Pierce, C. Hriţcu, J. Hughes, Z. Paraskevopoulou, and L.-y. Xia, "Beginner's Luck: A language for property-based generators," in *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: ACM, 2017, pp. 114–129. [Online]. Available: http://doi.acm.org/10.1145/3009837.3009868

[25] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806835

[26] Z. Manna and R. J. Waldinger, "Synthesis: Dreams $\rightarrow$ programs," *IEEE Trans. Software Eng.*, vol. 5, no. 4, pp. 294–328, Jul. 1979. [Online]. Available: https://doi.org/10.1109/TSE.1979.234198

[27] S. Gulwani, O. Polozov, and R. Singh, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1–2, pp. 1–119, 2017. [Online]. Available: http://dx.doi.org/10.1561/2500000010