Targeted Property-Based Testing

Andreas Löscher
Department of Information Technology
Uppsala University
Uppsala 751 05, Sweden
andreas.loscher@it.uu.se

Konstantinos Sagonas
Department of Information Technology
Uppsala University
Uppsala 751 05, Sweden
kostis@it.uu.se

ABSTRACT

We introduce *targeted property-based testing*, an enhanced form of property-based testing that aims to make the input generation component of a property-based testing tool guided by a search strategy rather than being completely random. Thus, this testing technique combines the advantages of both search-based and property-based testing. We demonstrate the technique with the framework we have built, called Target, and show its effectiveness on three case studies. The first of them demonstrates how Target can employ simulated annealing to generate sensor network topologies that form configurations with high energy consumption. The second case study shows how the generation of routing trees for a wireless network equipped with directional antennas can be guided to fulfill different energy metrics. The third case study employs Target to test the noninterference property of information-flow control abstract machine designs, and compares it with a sophisticated hand-written generator for programs of these abstract machines.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Search-based software engineering*; • **Theory of computation →** *Simulated annealing*;

KEYWORDS

Property-based testing, Search-based testing, QuickCheck, PropEr

ACM Reference format:
Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
https://doi.org/10.1145/3092703.3092711

1 INTRODUCTION

Testing is an integral part of modern software development as it finds errors in software systems and gives confidence in their correctness. Random property-based testing (PBT) is a high-level, semi-automatic, black-box testing technique in which, rather than writing a plethora of test cases 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 [11, 27].

Testing should unveil faults as effectively as possible within the available resources. As with all random testing techniques, the chance of finding a bug and the confidence in the correctness of the SUT increases with the number of generated tests. With each additional test, the input space gets covered a bit more. This works especially well in situations where a bug is triggered by a high percentage of the input space. However, if the input space is large then even a big number of tests per property will not yield satisfactory confidence. Moreover, if the percentage of failing input is relatively small, it can be hard to find counterexamples to properties that the SUT does not fulfill.

Search-based software testing techniques (cf. the survey article by Harman et al. [14]) try to resolve such issues by applying search techniques to the input generation. The input generation is then typically guided towards an optimization goal, e.g., towards inputs that maximize or minimize some quantity of the system.

This paper introduces *targeted property-based testing*, an enhanced form of PBT that makes its input generator component of a PBT tool guided by a search strategy instead of being random. We describe the conditions under which targeted property-based testing is applicable and the ingredients it requires. Moreover, we present Target, a system that provides a concrete implementation of targeted property-based testing. Target, which nowadays is fully integrated in the property-based testing tool PropEr [26], extends the high-level language of PropEr for specifying properties with supporting infrastructure that allows the user to employ some built-in or specify a new custom search strategy for input generation in a succinct and flexible way. In doing so, Target increases the probability that input that falsifies a property is generated, and ultimately the tester's confidence in the SUT. In all examples we have tried so far, targeted PBT outperforms random PBT.

The rest of the paper is structured as follows. The next section overviews random PBT and presents examples that show its shortcomings when dealing with large and complicated input domains. Section 3 introduces the idea of targeted PBT and the Target framework in an informal way. Since Target is parametrized by a search strategy, we describe the implementations of two built-in such strategies in Section 4. We then present a more formal description of targeted PBT in Section 5 and demonstrate its effectiveness on three case studies in Section 6. The paper ends with two sections containing related work and concluding remarks.

2 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

ISSTA'17, July 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092711

inputs to the SUT and properties that are expected to hold when the system is subjected to instances of valid inputs. A property-based testing tool takes these definitions and successively generates inputs with increasing complexity. The tool then subjects the SUT to these inputs and checks if the outputs falsify the properties or not. Following this methodology, 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 behaviour.

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 test data *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 larger percentage of behaviours of a system and unveiling its errors.

In PBT tools the testing process can typically be configured in various ways through options. For example, users can control the number of tests to run, the size of generated inputs, etc.

Let us illustrate PBT and explain the language of PropEr [26], the tool we use, with a simple example.

```
prop_list_reverse() ->
  ?FORALL(L, proper_types:list(), lists:reverse(lists:reverse(L)) == L).
```

This property states that, for any list `L`, if we reverse `L` twice we get back the original list. 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 begins with a capital letter (like `L` 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 (like the `list()` generator, which is available from the `proper_types` module). Using the built-in generators and appropriate macros, like the `?LET` and `?SUCHTHAT` macros that we will use below, users can define custom generators which are tailored to the inputs of properties they want to test. However, writing such custom generators is often quite challenging. Occasionally, it can be extremely difficult to come up with generators that are effective.

A Motivating Example

Let us present a more involved example. Suppose we want to test whether a system of network 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 amount 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. (This type of situation often occurs in sensor networks where leaf nodes collect data and send them to a sink for further processing [2].)

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). Obviously we can easily construct a counterexample for this property by hand. But our aim, of course, is to find a counterexample automatically. We can specify such a property in PropEr as follows:

```
prop_length() ->
  ?FORALL(G, graph(42), lists:max(distance_from_sink(G)) < 21).

graph(N) ->
  Vs = lists:seq(1, N),
  ?LET(Es, proper_types:list(edge(Vs)), {Vs, lists:usort(Es)}).

edge(Vs) ->
 ?SUCHTHAT({N1, N2}, {oneof(Vs), oneof(Vs)}, N1 < N2).
```

Assume that the function call `distance_from_sink(G)` returns a list with the lengths of the shortest path from each node of `G` to the sink. The maximum of these lengths is calculated by `lists:max()`. The property `prop_length` uses the `graph(N)` generator which produces a list of vertices `Vs` and picks a list of edges as defined by the `edge(Vs)` generator. This generator picks two vertices from `Vs` such that the number of the first vertex is strictly smaller than the second one. This makes it possible to filter out duplicate edges in the `graph(N)` generator. Property-based testing makes it easy to describe structured data like the graphs as input to the system. This property can easily be tested using PropEr:

```
1> proper:quickcheck(example:prop_length(), 4711).
.................. 4711 dots ......................
OK: Passed 4711 test(s).
```

We see that the property is not falsified after running the specified number of tests. PropEr randomly generated 4,711 graphs, each with 42 vertices, and none of these graphs had a shortest path from the sink to another node that was longer than 21 hops.

This property is hard to falsify without writing a significantly more involved custom generator for graphs of some number of vertices. The topology has to have a particular shape that is only found in a relatively small percentage of the inputs. For example, more complex input (more edges in the topology) will not automatically lead to a higher probability in finding a counterexample since adding edges can introduce additional shorter paths between the sink and another vertex.

In principle, it is possible to find a counterexample with PropEr. However, the odds of doing so are low and the number of runs that is needed is very high. For this property, we were not able to find a counterexample using PropEr after 100,000 tests, even after repeating the same experiment a thousand times.

With better knowledge of the input domain, the odds of finding the right input can be increased. For example, limiting the generator to graphs with longer paths between two nodes certainly increases the probability of finding a failing input. Such a generator however is complicated to write. In other properties, the relation between the network topology and the observed performance metric might not be as clear as in this example and it may be unclear what structure the generated graphs should have. Last but not least, having to

write a tailored custom generator for each different property makes property-based testing less attractive; we are much better off if we can use the same graph generator to test all properties of the network.

The property `prop_length` demonstrates that it can take many runs to find a counterexample if the amount of possible inputs that falsify a property is small compared to the size of the input domain. It is however possible to use search strategies to generate inputs that have a higher probability to falsify a particular property by observing the relation between the input value `G` and the output from `lists:max(distance_from_sink(G))`.

In our example, the output of `lists:max(distance_from_sink(G))` is growing monotonically towards its maximum value. This means that it is always possible to increase the output value by either adding edges to introduce a new longest path, or by removing edges to prevent "short-cuts." This fact can be exploited in the generators by using the previously generated input and the associated output of `lists:max(distance_from_sink(G))` to generate the next input. One possibility to do so is to employ a search strategy such as *Hill Climbing* in the input generation process.

3 TARGET: INFORMAL PRESENTATION

Targeted property-based testing is a variation of property-based testing that aims to make test outcomes more consistent and reduce the amount of required test runs to find bugs or achieve the same confidence in the SUT compared to random PBT. It achieves this by guiding the input generation with search techniques towards values that have a higher probability of falsifying a property. Doing so results in a more efficient exploration of the input space. The testing framework we developed that implements targeted PBT, called Target, uses information gathered during test execution in the form of *utility values* (UVs) that specify how close input came to falsifying a property.

The need for these UVs to exist naturally limits the type of properties that Target can be applied to. Still, Target's application domain is quite large. As we will also see in Section 6, tests where timing, resource consumption, or performance properties of a SUT are checked against a threshold are ideal candidates for targeted PBT. Target provides aid for writing such properties, built-in search strategies to guide the input generation, and support for extending the framework with new user-specified search strategies.

Target consists of three main components: (1) the *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 value(s) and a potential counterexample for the property.

The Target framework currently comes with an implementation of Hill Climbing and Simulated Annealing as built-in search strategies. However, the infrastructure that Target provides is general enough to be applicable to other search strategies, e.g., based on genetic algorithms or linear regression.

The general structure of a property that can be tested with Target looks as follows:

```
prop_Target() ->                    % Try to check a property
  ?TARGET_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 (UV). This UV is then reported to the search strategy component of Target which uses this information to produce the next input value. The search strategy then tries to explore the input space effectively and produce a new input that maximizes the UV, thus increasing the chance of falsifying the property.

The implementation of the search strategy is mostly independent from the property that is tested. The user of Target does not necessarily need to know how a certain search strategy is implemented in order to use it.

Most strategies require additional information about how the inputs are generated. (This information is the argument of the `?TARGET` macro passed in the form of an association map.) Simulated Annealing for example, as provided by Target, requires the user to specify a generator for the first element and a neighborhood function; see Section 4.2. However the user does not need to implement how utility values and inputs are handled by the search strategy. Moreover, the property itself stays mostly unchanged and can be expressed in typical PBT manner.

4 SEARCH STRATEGIES

We now describe two search strategies (Hill Climbing and Simulated Annealing) that are built-in in the Target system.

4.1 Hill Climbing

Let us begin by describing the Hill Climbing (HC) strategy for graph inputs as used in our `prop_length` example. The strategy starts with an initial and a random neighboring input value and picks the best of the two as new best input. This best input is then compared to a random neighboring input until no better input can be found. A neighboring input is an input that is similar to the current one.

If we apply the HC strategy to `prop_length` we want to achieve a higher probability of finding an input that falsifies the property. To apply the strategy, we need to connect the utility values produced by `lists:max(distance_from_sink(G))` with the property and the generator for the input.

A rewritten `prop_length` that makes use of the macros provided by Target and the HC strategy looks like follows:

```
prop_length_hc() ->
  ?TARGET_STRATEGY(hill_climbing,
    ?FORALL(X, ?TARGET(graph_hc(42)),
            begin
              UV = lists:max(distance_from_sink(G)),
              ?MAXIMIZE(UV),
              UV < 21
            end)).

graph_hc(N) ->
  #{first => graph(N), next => fun graph_next/1}.
```

The `?TARGET_STRATEGY` macro informs Target about the search strategy we want to use (Hill Climbing in this case). The `?TARGET` macro defines generators that are under the control of Target. HC needs an initial input and the ability to produce a random neighboring input. We use the `graph(N)` generator to obtain a random initial solution and forward it together with the neighborhood function `graph_next(G)` to Target. Finally `?MAXIMIZE` tells Target which variable should be maximized.

Usually PropEr controls how the input is produced from the generators. With Target we want to hand control over the input generation process from PropEr to a search strategy. This is achieved by tagging the generators with the `?TARGET()` macro. This macro is defined as follows:

```
-define(TARGET(Params), targeted(make_ref(), Params)).

targeted(Key, Params) ->
  ?LAZY(targeted_gen(Key, Params)).

targeted_gen(Key, Params) ->
  {State, NextFunc, _UpdateFunc} = get_target(Key, Params),
  {NewState, NextValue} = NextFunc(State),
  update_target(Key, NewState),
  NextValue.
```

The `?TARGET` macro has one argument that allows to pass information about the type or the size of the input or similar to the search strategy. We use the `?LAZY()` construct of PropEr. `?LAZY` creates a generator that evaluates the enclosed expression each time a new value needs to be generated. This means that for each generation step PropEr calls `targeted_gen()` with the same arguments. The first argument to `targeted_gen()` is a unique reference[1] that is used as key to the generator.

The `targeted_gen()` function calls `get_target()` to get the target-triple consisting of a *target state*, a *next function* and a *state-update function*. The next function is called with the current state to produce a new instance of the input. This input generation can change the target state, thus `update_target()` is called to store this new state. When `get_target()` is called for the first time it creates a new target according to the used search strategies and the passed options.

We also want to associate each input value with a utility value. Each time a utility value `UV` has been extracted from the SUT the `?MAXIMIZE(UV)` macro can be used to tell Target which value should be increased in the next input. Its implementation is similar to the `?TARGET()` macro:

```
-define(MAXIMIZE(UV), update_target_uvs(UV)).

update_target_uvs(UV) ->
  [update_target_uv(Key, UV) || Key <- get_target_keys()].

update_target_uv(Key, UV) ->
  {State, _NextFunc, UpdateFunc} = get_target(Key, []),
  NewState = UpdateFunc(State, UV),
  update_target(Key, NewState).
```

The functions `update_target()` and `get_target()` are stateful and preserve the target state in-between test runs. This enables Target to reason over previously generated inputs and their associated utility values when generating the next input(s).

A search strategy is mainly defined by how the next function and the state-update function are implemented as well as the initial

[1] More information at http://erlang.org/doc/man/erlang.html#make_ref-0.

```
init_target(#{first := First, next := Next}) ->
  {%% 1st element: initial state
   {generate_sample(First), unknown, none},
   %% 2nd element: next function
   fun ({LastAcc, AccUtility, _LastGen}) ->
     NewValue = generate_sample(Next(LastAcc)),
     {{LastAcc, AccUtility, NewValue}, NewValue}
   end,
   %% 3rd element: state-update function
   fun ({LastAcc, AccUtility, LastGen}, GenUtility) ->
     case AccUtility =:= unknown orelse GenUtility > AccUtility of
       true ->   % accept new solution
         {LastGen, GenUtility, LastGen};
       false ->  % continue with old solution
         {LastAcc, AccUtility, LastGen}
     end
   end
  }.
```

Figure 1: A function initializing Target for hill climbing.

state of Target. For the strategy `hill_climbing` these definitions are shown in Fig. 1. The function `init_target()` is called with the options passed when a target is initially created. These options consist of a generator `First` of the first input and a generator `Next` that produces neighboring input. In our example, we initialized Target with `graph_hc`, so we will use the Hill Climbing strategy to generate graphs.

The *state* consists of the currently best input, the associated utility value, and the last generated input. The initial best input is a random sample from the `graph` generator we used in the property `prop_length` in Section 2. Since we do not know the utility value of the initial input without testing it, we set it to `unknown` in the initial state. Similarly, we initially set the last generated value to `none`.

The *next function* generates a random value in the neighborhood of the last accepted input. This value is stored in the new state and returned as the next input for the property. We sample the neighboring solution from the generator `Next` which is parameterized by the last accepted solution. This way we can use PropEr's language for defining generators to describe the neighborhood function for the graph.

The *state-update function* compares the utility values of the last generated and last accepted solution and stores the best value and utility value as the new best solution.

The generator `Next` as used in `prop_length_hc` is implemented with `graph_next(G)` as seen in Fig. 2. To produce neighboring graphs, the code first decides on a new graph size and then removes and adds a random amount of edges such that the new size of the graph is as decided.

If we test the property `prop_length_hc` now it fails after an average of 17,666 tests (measured over 1,000 runs with each time running a maximum of 100,000 tests). More importantly, a counterexample is found in all runs.

Hill Climbing as presented here has a series of shortcomings. Finding a graph with maximum degree is a convex problem (local optima are also global optima). Hill Climbing is a local optimization strategy that performs very well on this class of problems. In practice however, we need strategies that allow us to escape local optima. We therefore propose the use of more powerful strategies such as the one we present next.

```
graph_next(G) ->
  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))).
```

Figure 2: Implementation of the neighborhood function for the HC strategy.

4.2 Simulated Annealing

Simulated Annealing (SA) is a well-studied local search metaheuristic [22] 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 has also another favorable property in 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 a solution space Ω (the set of all possible solutions) and an objective function $f : \Omega \to \mathbb{R}$. In our framework Ω is equivalent with the input space I. The objective function f is equivalent to the property function p if we ignore whether the property holds or fails. Furthermore, SA defines a neighborhood function $\mathcal{N} : \Omega \to \Omega$ that produces random neighboring solutions (solutions that are close in the solution space) to a given solution.

SA starts with a random initial solution from the solution space. It then produces a neighboring solution and accepts it as new solution if the associated UV is higher than the one of the current solution. It also accepts worse solutions 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 temperature usually decreases over time following a temperature function (TF). 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 implementing SA as search strategy we need to provide a generator for I and the neighborhood function $\mathcal{N}$ for the input space we want to use. Similar to the HC strategy, it is possible to define this generator and neighborhood function using PropEr's generator language and to pass them as parameters to the `?TARGET` macro. The default integer generator for SA is the one below:

```
integer(Low, High) ->
  #{first => proper_types:integer(Low, High),
    next  => integer_next(Low, High)}.
```

The `first` element is implemented by using PropEr's default integer generator. The `next` element returns a generator that given an instance from the input space and a temperature value between 0.0 and 1.0 will produce a neighboring element to the given instance. The distance between the neighboring instance and the original instance is determined by the given temperature. In Target, `integer_next()` is implemented as follows:

```
integer_next(Low, High) ->
  fun (OldInstance, Temperature) ->
    Offset = trunc(abs(Low - High) * Temperature * 0.1) + 1,
    ?LET(X, proper_types:integer(-Offset, Offset),
         ensure_range(X + OldInstance, Low, High))
  end.
```

The `ensure_range()` function bounds the newly generated value to the allowed interval [`Low`,`High`]. This neighborhood function produces new integers that are at most 10% of the total interval range apart from the given integer. This distance is also scaled by the temperature. This scaling makes it possible to find the local optima point faster in low temperature conditions.

In a similar manner, we can use the `graph(N)` and `graph_next(G)` generators to specify an SA-capable generator for graphs:

```
graph_sa(N) ->
  #{first => graph(N), next => fun (Base, _T) -> graph_next(Base) end}.
```

Here, the next generator ignores the temperature argument, which means that temperature scaling is not used and the temperature only affects the acceptance probability $\Pr_{\text{accept}}$.

The temperature is controlled by a temperature function (TF). We provide four different temperature functions: a linear decreasing TF and three versions of re-heating ones. The linear decreasing TF decreases the temperature linearly from 1.0 to 0.0. This works well for most applications. An issue with this approach however is that the temperature is high for a long time in the beginning. This means that bad solutions are accepted with a high probability and the distance between tested solutions is also high. In such a situation, a very good solution might not be pursued because one of the next-worse solutions is accepted.

Re-heating TFs try to address this issue by decreasing the temperature much faster. Since this can easily result in getting a situation where SA is getting stuck in a local optima, re-heating strategies also increase the temperature if no new solution has been accepted in a certain number of attempts. Re-heating also has the advantage in that it lets the search escape local optima that have larger extends [5]. Target provides three different temperature functions (fast, very fast, and delayed re-heating) that utilize re-heating. It is also possible to use a user-defined temperature function instead.

Target comes with a library containing SA-capable generators (initial generator and neighborhood function) for some basic data types. For more complicated input the user of Target has to provide these generators. Using Target with the SA search strategy reduces

the user's responsibility of having to provide a fitting neighborhood function to a generator. Implementing such a neighborhood function for more complex input can occasionally be difficult. We are however confident that this process can be automated.

5 TARGET: MORE FORMAL PRESENTATION

We now define targeted property-based testing and strategies more formally. In PBT we can define the input as $i \in I$, where I is the set of all possible inputs. If we test the property p with the input i the property either holds or fails, i.e., $p : I \to \{true, false\}$.

The Target framework extends this notion by adding the utility value $u \in U$ to the result of p:

$$p : I \to U \times \{true, false\}$$

Further, we define $i^n \in I^n$ as a vector containing all n previously tested inputs and $u^n \in U^n$ as the vector containing the associated utility values. We define a *Target* as:

$$\mathcal{T} : I^n \times U^n \to I$$

Usually it is not possible to obtain an explicit version of the utility function (UF), especially if we cannot model the SUT precisely. Given a sufficient sample set of UVs, it however possible to approximate the underlying UF.

It is up to the tester to extract the UVs from the running code that tests a specific property. We assume that inputs with higher UVs have a higher probability of being a counterexample than inputs with lower UVs.

Using Target, the property has to be expressed as follows:

$$\forall i \in \text{Input}, f(i) < t$$

The parameter t is a fixed threshold. The function f should run the SUT with the input and extract the UV that is to be maximized and is in fact the utility function.

A specific implementation of the target function $\mathcal{T}$ is called a *strategy*. The targeted property-based testing framework we describe here is not limited to a single strategy but is designed to be general enough to support a wider variety of methods to fit different application requirements.

When using the simulated annealing strategy in Target, the target function $\mathcal{T}_{sa}$ is implemented by the neighborhood function $\mathcal{N}$. The input for the neighborhood function is the last solution that was accepted by the simulated annealing algorithm:

$$\mathcal{T}_{sa}(i_n, u_n) = \begin{cases} \text{select a random } i \in I & \text{if } n = 0 \\ \text{last accepted } i \text{ from } i_n \to \mathcal{N}(i) & \text{otherwise} \end{cases}$$

Similarly, the hill climbing strategy $\mathcal{T}_{hc}$ can be defined as follows:

$$\mathcal{T}_{hc}(i_n, u_n) = \begin{cases} \texttt{generate\_sample(First(N))} & \text{if } n = 0 \\ k : u_k^n = \max_{j=0}^{n} u_j^n \to \mathcal{R}(i_k^n) & \text{otherwise} \end{cases}$$

$$\mathcal{R}(x) = \texttt{Next(}x\texttt{)}$$

6 CASE STUDIES

We now evaluate targeted property-based testing and Target by comparing its effectiveness to guide the input generation with random property-based testing in three case studies. All tests were run on a laptop with an Intel Core i7-4600U CPU (2.10GHz) and 8 GB of RAM. However, only one core of the machine was used.

6.1 Energy Efficiency of MAC Protocols

Energy efficiency is an important part of sensor network applications. During deployment, it is crucial to preserve the battery of the nodes in order to maximize their lifetime.

In prior work [18], we described how random property-based testing can be used to test the energy efficiency of network configurations using different implementations of MAC protocols. In a similar spirit, we generate random sensor networks consisting of UDP server and client nodes. The client nodes periodically send messages to the server nodes. We then record the duty cycle for each node in the network over a fixed period of time. The property we want to hold is that for a given network, no node has a duty cycle more than a threshold. (A more detailed description of the experimental setup using Contiki OS and the Cooja simulator is given in the paper by Löscher et al. [18].)

Testing sensor networks in a simulator can be costly since besides the software that we want to test also the hardware of the sensor nodes and the (radio) environment for communication needs to be simulated. Finding a counterexample that achieves a duty cycle of more than 10% for at least one node required only between 10 and 40 test runs. The property is much harder to falsify if the threshold is higher. The problem here is similar to the longest path problem we saw before. Additional connections in the network topology can allow messages to take a more efficient route to the server, relieving the other nodes in the system.

The more efficient exploration of the input space as done by Target leads to a higher confidence in the system's correctness for a lower number of runs. Moreover Target is capable of finding configurations that falsify the property with a smaller number of runs whereas a completely random input generation requires a much higher amount of runs. This issue gets amplified in performance by the high cost for testing each run in the simulator.

Let us describe how we used Target to guide the generation of the random sensor networks towards one with a high duty cycle. We chose a duty cycle of 25% as threshold. This means that input that achieves a duty cycle of more than 25% for at least one of the nodes will falsify the property. We then compare the performance with random property-based testing.

The property used to test the duty cycle with Target is based on the original property; see Fig. 3 that shows in red color the differences from the original code of the property. To use Target, all we need to do is to modify the property in order to specify the search strategy and the utility value of the input generation. This is done by wrapping the property by `?TARGET_STRATEGY(simulated_annealing, ...)`. The utility value is the maximum duty cycle since that is the value we want to maximize. We can specify this by adding `?MAXIMIZE(MaxDutyCycle)`. Finally, we need to specify the target generator according to the search strategy. Simulated annealing, as used here, needs a generator for the initial graph along with a specification of the neighborhood function for graphs. A graph is a complex data structure which results in a rather complex neighborhood function. Let us describe this function as used in the property.

Let $G = (V, E)$ be a graph where V is a set of vertices and $E = V \times V$ are the edges. The *order* of the graph is $|V|$ and the *size* of the graph is $|E|$. We define a graph $G_{\text{next}} = \mathcal{N}(G)$ as a graph $(V_{\text{next}}, E_{\text{next}})$ that has an altered set of vertices and edges. The order

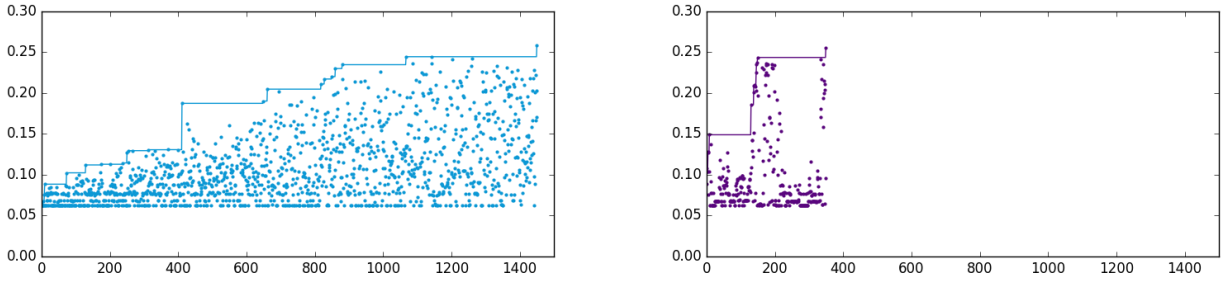

Figure 4: The achieved duty cycle (y-axis) with random PBT (left) and TARGET with Simulated Annealing (right) varying the number of tests (x-axis). The graphs also show that TARGET requires significantly less tests to find a counterexample.

```
prop_duty_cycle_below_threshold() ->
  ?TARGET_STRATEGY(simulated_annealing,
    ?FORALL({Motes, Links}, configuration(),
      begin
        ok = setup()
        {running, Handler} = nifty_cooja:state(),
        initialize_simulation(Handler, Motes, Links),
        SimTime = 120 * 1000,   % simulate for 2 minutes (120 secs)
        ok = nifty_cooja:simulation_step(Handler, SimTime),
        MaxDutyCycle = max_duty_cycle(Handler, Motes),
        ok = nifty_cooja:exit(),
        ?MAXIMIZE(MaxDutyCycle),
        MaxDutyCycle < 0.25     % check that is below 25%
      end)).

configuration() ->
  ?LET({V, E}, ?TARGET(sa_graph()), {motes(V), links(E)}).

sa_graph() ->
  #{first => graph(), next => sa_graph_next()}.
```

Figure 3: Property that checks whether the duty cycle of a sensor network is below a certain threshold (here 25%). The red parts show the additional code required to use the TARGET framework in PROPER compared to random PBT.

of the altered graph is

$$order_{\text{next}} = \mathcal{N}(order) \tag{1}$$

and $v_{\text{add}}, v_{\text{del}} \in N$ are the amounts of vertices we add and delete from the original graph to achieve the new order. Thus:

$$order_{\text{next}} = order + v_{\text{add}} - v_{\text{del}} \tag{2}$$

And now we can define the set of next vertices as follows:

$$V_{\text{next}} = (V \cup A) \setminus D \tag{3}$$

where $v_{\text{add}} = |A|$, $v_{\text{del}} = |D|$ and $D \subseteq V \cup A$.

By removing some of the vertices we have to adjust the edge set to contain only valid edges. Furthermore we want the edge set to also contain edges for the newly added vertices:

$$\begin{aligned} E_{\text{add}} &\subseteq (V \cup A) \times A \\ E_{\text{del}} &= \{(v_i, v_j) \in (E \cup E_{\text{add}}, v_i \in D \vee v_j \in D)\} \\ E_{\text{intermediate}} &= (E \cup E_{\text{add}}) \setminus E_{\text{del}} \end{aligned} \tag{4}$$

The transition from $E_{\text{intermediate}}$ to E_{next} works like the transition from V to V_{next}.

```
sa_graph_next() ->
  fun ({V, E} = _OldInstance, Temperature) ->
    %% generate new order according to Eq. (1)
    ?LET(NewSize, tinteger(length(V), Temperature),
    %% generate add and del operations as in Eq. (2)
    ?LET({Adds, Dels}, get_op_count(NewSize, length(V), Temperature),
    %% perform operations according to Eqs. (3) and (4)
    ?LET({V_Add, E_Add}, add_vs(V, E, Adds, Temperature),
    ?LET({V_Del, E_Del}, del_vs(V_Add, E_Add, Dels),
    %% change edges
    ?LET(NewEdgeSize, tinteger(length(E_Del), Temperature),
    ?LET({EAdds, EDels}, get_op_count(NewEdgeSize, length(E_Del), Temperature),
    ?LET(E_EAdd, add_es(V_Del, E_Del, EAdds),
    ?LET(E_EDel, del_es(E_EAdd, EDels), {V_Del, E_EDel}))))))))
  end.

tinteger(Base, Temperature) ->
  Offset = trunc(0.5 * Base * Temperature) + 1,
  proper_types:integer(Base - Offset, Base + Offset).
```

Figure 5: Code to generate the next graph when using the SA strategy of TARGET. The generation steps of the neighborhood function correspond to Eqs. (1) to (4).

It is important to note that the size of the parameters involved in the manipulation of the graph are scaled by the current temperature of the SA process. This means that under high temperature the next graph can be more different from the previous one than under low temperature.

The implementation for the graph generator is similar to the one used by `prop_length_hc` which was shown in Fig. 2 with the difference that the order of the graph is not fixed. Figure 5 shows the implementation of these definitions as a graph generator for SA strategy of TARGET.

To evaluate the effectiveness of the strategy compared to random PBT we tested the property with and without using the strategy.

Figure 4 shows the achieved duty cycle for one test of the property with random property-based testing (the graph on the left) and with TARGET using Simulated Annealing (graph on the right). Random PBT progressively produces more complex input but the input generation is not guided otherwise. We can see that the achieved duty cycle for more complex networks is potentially higher than the duty cycle for smaller ones. The duty cycle values are spread, which results in a less thorough exploration of the input space that potentially yields a higher duty cycle. We can see that already after around 400 tests random PBT achieves a high duty cycle of

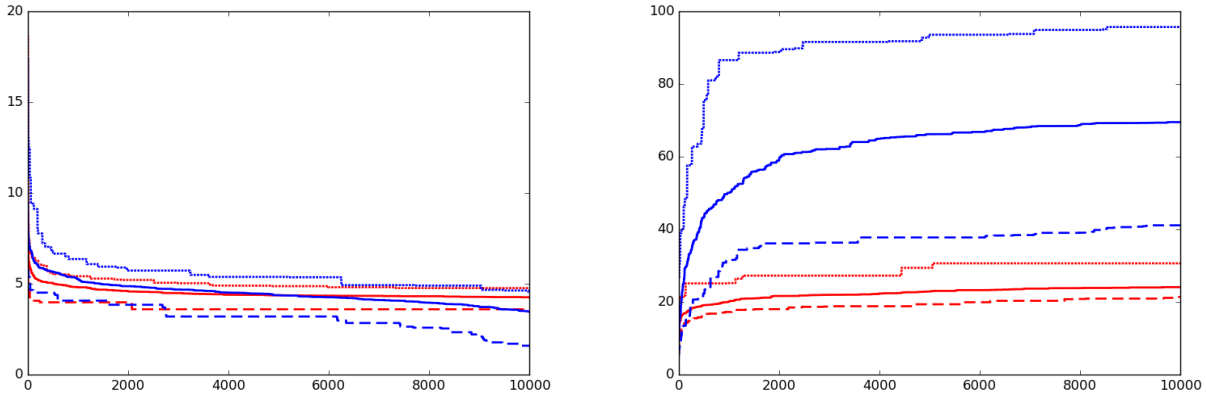

Figure 6: The y-axis shows a metric of the energy consumption over tests (x-axis). The lines show the minimum (dotted), median (solid), and minimum (lines) of random PBT (red) in comparison to TARGET (blue) of the achieved minimums (left graph) and maximums (right graph) of 1000 runs with 10,000 tests.

around 20%. The graph also shows that this area is not densely populated by samples. This means that the probability of finding such an example so early is not high.

When using TARGET, the search strategy quickly follows the path of good solutions to produce inputs that yield a high duty cycle; see the graph on the right of Fig. 4. The more promising input areas are explored more thoroughly. We can also observe that after around 300 tests a worse solution was most likely accepted, which ultimately leads to a duty cycle of more than 25% after 350 tests.

When using random PBT, it took an average of 1188 tests to find a counterexample and the mean time for each test was 23.5s. Thus, a counterexample was found only after 7h46m. (These numbers are the mean over ten runs.) In contrast, TARGET was able to produce a counterexample on average after around 200 tests and the mean time for each test was 40.6s. (We note that the time per test using TARGET is higher mainly because TARGET guides the generation towards larger networks which require more resources to simulate.) Thus, TARGET needed on average only 2h12m to find a counterexample; i.e., it was about 3.5 times faster than random PBT in this case study.

6.2 Routing Tree for Directional Antennas

With this second case study we wanted to see how well TARGET performs when handling complex structured input.

In our experiment we generate the routing tree for a network of nodes that are connected via a radio connection. The distinct feature of the radios that the nodes use is that they are SPIDA smart antennas [23, 25]. In contrast to regular antennas which are omnidirectional (i.e., they have an equal gain in all directions), smart antennas are antennas that can dynamically control the gain as a function of the direction. Thus, this type of smart antennas can be configured to increase the communication range by concentrating the transmit power in a specific direction. This also results in less radio interference since devices that are not part of the communication are less effected. The radios used by SPIDA antennas can be configured to send or receive radio messages from six different directions. This means that besides generating a routing tree we also need sender and receiver directions for each link. Two nodes can both send and receive on each of their six directions with no additional energy cost. For the communication between two nodes we therefore have 36 (six sender directions, six receiver directions) possible combinations of antenna directions.

Prior to running our tests, we collected experimentally the average package delivery rate pdr_{avg} for each link (and for each of the 36 possible direction pairs) in a deployment of 40 sensor nodes equipped with the SPIDA smart radio.

To evaluate how TARGET performs in comparison with random PBT we wrote the property to test so that we generate routing trees with directions for each link. The more total hops the routing tree has, the more packages need to be forwarded to the root on average. A low pdr_{avg} will result in more re-transmissions. Therefore we calculate a metric for the energy consumption based on these values. The metric is calculated by multiplying the sum of hops from each node to the root of the routing tree, with $1-pdr_{avg}$. We then instruct TARGET to minimize this value:

```
?TARGET_STRATEGY(simulated_annealing,
  ?FORALL(Tree, ?TARGET(tree()),
          begin
            HOPS = get_total_hops(Tree),
            PDR_AVG = get_avg_pdr(Tree),
            Metric = (1 - PDR_AVG) * HOPS,
            ?MAXIMIZE(-Metric),
            ... % condition of the property here
          end)).
```

Here we use TARGET with the Simulated Annealing strategy to optimize the tree towards low energy consumption. We also compare the number of test runs against random PBT.

The left graph of Fig. 6 plots the maximum, median, and minimum of the achieved minimum energy metrics for 1,000 runs of 10,000 tests using random PBT in comparison to using TARGET with SA. While random PBT is able to find decent inputs, TARGET is ultimately able to find routing trees with a much lower energy metric and it does so more often than random PBT.

Let us now change the `?MAXIMIZE(-Metric)` expression to `?MAXIMIZE(Metric)` to maximize the metric instead of minimizing it. This means that we try to produce routing trees and directions that have a potentially high energy consumption. The generator stays unchanged. The right graph of Fig. 6 shows the result of 10,000 runs of the property compared to random PBT. We can see that Target is able to produce routing trees with an energy consumption that is much higher much faster. After 1,000 tests the worst run with Target is already better than the best end result achieved with random PBT after 10,000 tests. Both generation techniques need 150 seconds to run 10,000 tests on average.

It is of course possible for random PBT to find a solution much faster since generators produce random instances of input. The probability for this to happen is however low and the time it takes to convergence towards a specific goal value is much more consistent using the Target framework.

6.3 Noninterference

In this last case study, we use Target to test for noninterference. Hriţcu et al. [15] explored how PBT can be used to aid in the design of secure information-flow control (IFC) abstract machines. They showed that by specifying strong properties and a good generation strategy it is possible to efficiently generate programs that expose violations of noninterference in simple IFC machines.

A *Naive* generation, where programs are generated by choosing a random sequence of instructions independently and uniformly, does not discover bugs in the definition of IFC machines quickly and reliably enough. Therefore, Hriţcu et al. [15] presented a *generation-by-execution* (*ByExec*) strategy that generates programs step-by-step by adding one machine instruction at a time with the goal that the newly added instruction will not crash the IFC machine. The idea behind this is that long running programs explore more interesting machine states and are more likely to discover bugs in the machine's definition. In addition to this step-by-step generation, the generator picks the instructions with a weighted distribution so that for example `push` instructions are more likely to be generated than `noop` instructions. (For a more detailed description of the generator and the IFC machines refer to the article of Hriţcu et al. [15].) This generation theme fits well into the framework of Target.

We can easily implement the *ByExec* technique by writing generators for Target's simulated annealing strategy. The initial value is an empty program and a neighboring program is one with an added instruction. The new instruction is chosen so that it does not crash or terminate the program, outgoing from the last non-halting instruction of the prior program. We then only need to maximize the execution length of the so generated program. The problem with this is that the resulting programs become very long and their execution costly. We alleviate this problem by resetting the search after we reach an execution of 50 instructions. The noteworthy difference with the original *ByExec* generator is that not the whole program is generated at once and then tested. The Target generator incrementally generates the program by adding one instruction at a time and testing each intermediate step. Other than that, we use the same distribution for the instructions as the original *ByExec* generator [15].

Table 1: Average times (in msecs) to find a counterexample for bugs injected to the stack machine of Hriţcu et al. [15].

	PBT		Target	
	Naive	*ByExec*	*List*	*ByExec*
ADD	2234.08	312.97	319.86	68.49
PUSH	70.18	9.79	2.75	0.78
LOAD	324028.34	987.91	287.23	135.52
STORE A	–	4668.04	1388.09	263.94
STORE B	226.85	8.19	4.24	1.33
STORE C	130.22	10.01	2.76	0.79
MTTF (arithmetic)	–	999.48	334.16	78.48
MTTF (geometric)	–	102.44	40.05	11.25

As a second Target generator we use a generic *List* generator with simulated annealing. This generator is based on creating neighboring lists by adding and deleting elements to it. This generator is built-in in Target and can be used "out-of-the-box" simply by supplying it some information about the type of the list elements. Similar to the Target *ByExec* generator, we maximize the execution length, use the same weighted distribution for the instructions, and reset the search after generating a program with an execution of 50 or more instructions.

To compare the generators for Target, we re-implemented the original *ByExec* and a *Naive* generation strategy in Erlang instead of in Haskell and tested them on all bugs of the simple stack machine of Hriţcu et al. [15]. All bugs were subjected to 1,000 runs of the property, each run with a maximum of 1,000,000 tests. We recorded the average time to find each bug in milliseconds. The *Naive* generator was unable to find a counterexample for the "STORE A" bug.

Table 1 shows the measured times along with the arithmetic and geometric mean times to failure (MTTF). We can see that the Target *ByExec* generator outperforms random PBT using the *ByExec* generator. This is not surprising as it implements the same generation strategy as the regular *ByExec* generator but tests more intermediate steps. This has similar advantages as strengthening the test property as discussed in the journal article [15]. Both implementations of *ByExec* required only around 30 lines of generator code.

Even more noteworthy is that the Target *List* generator performs surprisingly well. It is slower than the specialized Target *ByExec* generator but faster than random PBT using *ByExec*, which is more sophisticated generation strategy but hard-coded instead of being guided by a search strategy. In contrast to the*ByExec* generator, the *List* generator needs very little effort to write: it is just a modified version of the *Naive* generator where the standard generator for lists is exchanged with the one from the Target library.

This case study shows that targeted PBT can be efficiently applied to a complex problem domain like the generation of programs with certain properties. It also shows that not all domains need a specialized SA-capable generator. Generic configurable generators like Target's *List* generator can significantly reduce the implementation effort that is required by the programmer and also offer good testing effectiveness.

7 RELATED WORK

The idea of using a high-level language for writing properties and generators for property-based testing originated by the pioneering work of Claessen and Hughes [11] on the QuickCheck library for the lazy functional language Haskell back in 2000. Nowadays, a wide variety of programming languages come with similar PBT tools [27]. In the context of Erlang alone, the high-level programming language that we use in this paper to express the properties to test and their inputs, three such tools are available: a commercial one by QuviQ [4] and two open-source ones: Triq [28] and PropEr [26], on which Target is based. These tools have been successfully applied to test a wide variety of systems such as telecom systems [4], web services [17], protocols used in cars, and recently sensor networks [18]. Our Target framework is directly applicable to all these areas. Note that we use PropEr (and Erlang) only as a language to write properties and generators. The SUT need not be written in Erlang; for example, the sensor network applications that we used as case studies are written in C.

Search-based testing also has a long history. Its first ideas can be traced back to a 1976 paper by Miller and Spooner [21]. Since then search-based testing techniques have been applied to a wide variety of testing areas [1, 14, 20, 24] including structural testing, model-based testing, stress testing, non-functional testing, integration testing, and test-suite generation, among others. However, to the best of our knowledge, our work is the first one that tries to embed search-based testing ideas into a general environment for property-based testing.

EvoSuite [12] is a framework for automated unit test generation that is based on search-based software testing. The framework generates and optimizes whole test suites towards coverage criteria. EvoSuite suggests test oracles by adding assertions to the generated test cases that capture the current behavior of the SUT. The developer can then detect deviations from the intended system behavior to the current one by inspecting these assertions. EvoSuite is fully automated and operates on Java byte-code level. In contrast to EvoSuite, targeted property-based testing is a black-box testing technique. It is focused on efficient input generation for high-level properties, which can be written in a language different than that of the SUT, and uses user-defined optimization criteria (the utility values). This allows Target to guide the input generation efficiently even for very complex input domains like network topologies or programs. Furthermore, Target can be used to test non-functional properties like timing, performance, or resource consumption.

Luck [16] and UDITA [13] are languages for test data generation. In Luck generators are automatically derived from predicates using a hybrid approach that combines narrowing based techniques with constraint solving. The predicates are decorated with annotations that allow control over the amount of constraint solving that happens and the distribution of the generated values. UDITA is a Java-based language with non-deterministic choices for test generation. UDITA provides bound-exhaustive testing (e.g., all trees with up to n nodes) and offers different exploration strategies like random, depth-first, and breath-first. In contrast to Luck and UDITA, Target provides search strategies that guide the generation process towards promising values. It would however be possible to combine both approaches and use a language like Luck or UDITA to specify the neighborhood function for Target's SA strategy.

Adaptive random testing (ART) [8, 9], restricted random testing (RRT) [6], and quasi-random testing (QRT) [10] are forms of enhanced random testing that seek to distribute test cases more evenly within the input space to maximize test case diversity. They are based on the assumption that non-point failure types are easier to detect by an even spread of test cases. ART distributes the test cases based on a distance measurement and favors test cases that are *far away* from all previously generated ones. RRT defines exclusion zones around previously generated test cases and generates new test cases outside these zones. QRT utilizes low-discrepancy sequences in an n-dimensional hyper-cube that spread more evenly in underpopulated areas of the cube. There have been various improvements of these techniques especially of ART [7, 19]. On the other hand, a more recent study by Arcuri and Briand [3] has pointed out several shortcomings (high cost, only effective under high failure rates, etc.) of the ART technique.

Compared to targeted property-based testing, ART focuses entirely on the input domain without taking into account feedback from the test execution. We also point out that ART (and its RRT and QRT variants) can be implemented as search strategies in Target. Such strategies would however ignore the utility values.

8 CONCLUDING REMARKS

We introduced targeted property-based testing, a testing technique that extends property-based testing with a search-based component for more effective generation of inputs when the properties to be checked have a form that involve a utility value that we seek to maximize or minimize. Moreover, we showed the ability of our Target framework to explore the input space effectively while maintaining a high-level and expressive language for specifying properties. Through three different case studies, two from the domain of networks and one related to secure information-flow control, we demonstrated that Target is able to generate complex structured input and showed how complicated properties of network configurations and security can be tested.

As mentioned, Target is an integrated component of PropEr and currently supports two built-in search strategies. However, it is also an extensible framework where users can easily define their own search strategies if they wish or need to.

ACKNOWLEDGMENTS

We thank Leonidas Lampropoulos for comments that have improved the presentation, for suggesting to us the IFC case study, and for helping us resolve some problems that we encountered in the original Haskell version of the code.

REFERENCES

[1] Wasif Afzal, Richard Torkar, and Robert Feldt. 2009. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology* 51, 6 (2009), 957–976. https://doi.org/10.1016/j.infsof.2008.12.005

[2] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdai Cayirci. 2002. Wireless sensor networks: a survey. *Computer Networks* 38, 4 (2002), 393–422. https://doi.org/10.1016/S1389-1286(01)00302-4

[3] Andrea Arcuri and Lionel Briand. 2011. Adaptive Random Testing: An Illusion of Effectiveness?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 265–275. https://doi.org/10.1145/2001420.2001452

[4] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing Telecoms Software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. ACM, New York, NY, USA, 2–10. https://doi.org/10.1145/1159789.1159792

[5] Dimitris Bertsimas and Omid Nohadani. 2010. Robust Optimization with Simulated Annealing. *J. of Global Optimization* 48, 2 (Oct. 2010), 323–334. https://doi.org/10.1007/s10898-009-9496-x

[6] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. 2002. Restricted Random Testing. In *Proceedings of the 7th European Conference on Software Quality*. Springer, Berlin, Heidelberg, 321–330. https://doi.org/10.1007/3-540-47984-8_35

[7] Tsong Yueh Chen, F.-C. Kuo, Robert G. Merkel, and S. P. Ng. 2004. Mirror adaptive random testing. *Information and Software Technology* 46, 15 (2004), 1001–1010. https://doi.org/10.1016/j.infsof.2004.07.004

[8] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. 2010. Adaptive Random Testing: The ART of Test Case Diversity. *J. Syst. Softw.* 83, 1 (Jan. 2010), 60–66. https://doi.org/10.1016/j.jss.2009.02.022

[9] Tsong Yueh Chen, Hing Leung, and I. K. Mak. 2005. Adaptive Random Testing. In *Proceedings of the 9th Asian Computing Science Conference*, Michael J. Maher (Ed.). Springer, Berlin, Heidelberg, 320–329. https://doi.org/10.1007/978-3-540-30502-6_23

[10] Tsong Yueh Chen and Robert Merkel. 2007. Quasi-Random Testing. *IEEE Transactions on Reliability* 56, 3 (Sept. 2007), 562–568. https://doi.org/10.1109/TR.2007.903293

[11] Koen Claessen and John Hughes. 2000. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/1988042.1988046

[12] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In *Proceedings of the 11th International Conference on Quality Software (QSIC '11)*. IEEE Computer Society, Washington, DC, USA, 31–40. https://doi.org/10.1109/QSIC.2011.19

[13] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test Generation Through Programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*. ACM, New York, NY, USA, 225–234. https://doi.org/10.1145/1806799.1806835

[14] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based Software Engineering: Trends, Techniques and Applications. *ACM Comput. Surv.* 45, 1, Article 11 (Dec. 2012), 61 pages. https://doi.org/10.1145/2379776.2379787

[15] Cătălin Hriţcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo De Amorim, Maxime Dénès, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. 2016. Testing Noninterference, Quickly. *Journal of Functional Programming* 26 (2016), e4. https://doi.org/10.1017/S0956796816000058

[16] Leonidas Lampropoulos, Benjamin C. Pierce, Cătălin Hriţcu, John Hughes, Zoe Paraskevopoulou, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-based Generators. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 114–129. https://doi.org/10.1145/3009837.3009868

[17] Leonidas Lampropoulos and Konstantinos Sagonas. 2012. 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*. 3–16. https://doi.org/10.4204/EPTCS.98.3

[18] Andreas Löscher, Konstantinos Sagonas, and Thiemo Voigt. 2015. Property-based Testing of Sensor Networks. In *Sensing, Communication, and Networking, 12th Annual IEEE International Conference on*. IEEE, 100–108. https://doi.org/10.1109/SAHCN.2015.7338296

[19] Johannes Mayer. 2005. Lattice-based Adaptive Random Testing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 333–336. https://doi.org/10.1145/1101908.1101963

[20] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Softw. Test. Verif. Reliab.* 14, 2 (June 2004), 105–156. https://doi.org/10.1002/stvr.v14:2

[21] Webb Miller and David L. Spooner. 1976. Automatic Generation of Floating-Point Test Data. *IEEE Transactions on Software Engineering* SE-2, 3 (Sept. 1976), 223–226. https://doi.org/10.1109/TSE.1976.233818

[22] Alexander G. Nikolaev and Sheldon H. Jacobson. 2010. Simulated annealing. In *Handbook of Metaheuristics*. Springer, 1–39.

[23] Martin Nilsson. 2009. Directional antennas for wireless sensor networks. In *Proc. 9th Scandinavian Workshop on Wireless Adhoc Networks (Adhoc'09)*.

[24] Alessandro Orso and Gregg Rothermel. 2014. Software Testing: A Research Travelogue (2000–2014). In *Proceedings of the Future of Software Engineering*. ACM, New York, NY, USA, 117–132. https://doi.org/10.1145/2593882.2593885

[25] Erik Öström, Luca Mottola, Martin Nilsson, and Thiemo Voigt. 2010. Smart Antennas Made Practical: The SPIDA Way. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '10)*. ACM, New York, NY, USA, 438–439. https://doi.org/10.1145/1791212.1791294

[26] Manolis Papadakis and Konstantinos Sagonas. 2011. A PropEr Integration of Types and Function Specifications with Property-based Testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2034654.2034663

[27] QuickCheck 2016. QuickCheck. (2016). https://en.wikipedia.org/wiki/QuickCheck

[28] Triq 2016. Trifork QuickCheck for Erlang. (2016). https://github.com/krestenkrab/triq