UPPSALA UNIVERSITET

AUTOMATING TARGETED PROPERTY-BASED TESTING

Andreas Löscher, Konstantinos Sagonas

andreas.loscher@it.uu.se, kostis@it.uu.se

Department of Information Technology

Uppsala University

Sweden

UPPSALA UNIVERSITET

Property-Based Testing

- High-level, semi-automatic, black-box testing technique.
- Testing user-specified properties of the SUT.
- Examples:
  - QuickCheck (Haskell)
  - ScalaCheck (Scala)
  - PropEr (Erlang)
  - …

PropEr

A QuickCheck-Inspired Property-Based Testing Tool for Erlang

Random Property-Based Testing

- PBT tool provides:
  - Random generators for basic types.
  - Language to write more complex generators.

- PBT tool automatically tests these properties:
  - Generates wide range of random inputs.
  - Runs the SUT with these inputs.
  - Checks if the properties hold.

Random Property-Based Testing

Generator

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

Property should hold for all L

Property

Random Property-Based Testing

```
L=[]
L=[2]
L=[-5,-1,-8,1]
L=[16,3,-23]
L=[38,29,-28,12,-11,-3,-28,-6,9,-16,4,4]
```

…

```
1> proper:quickcheck(example:prop_list_reverse(), 1000).
..................... 1000 dots .......................
OK: Passed 1000 test(s).
```

UPPSALA UNIVERSITET

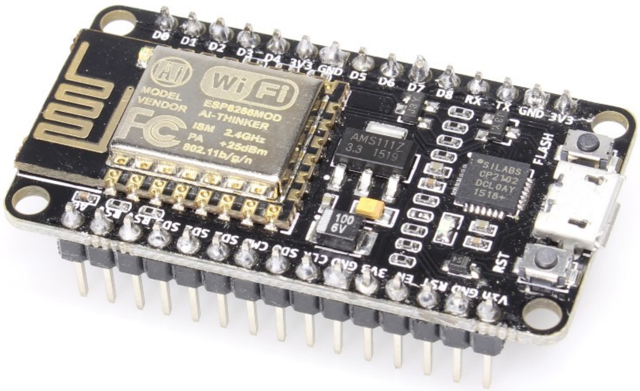
PBT of Sensor Networks

Setup:

- Sensor network
- Random distribution of UDP server and client nodes
- Client node periodically sends messages to server node

Test:

- Has X-MAC for any network a duty-cycle > 25%?



(duty-cycle ::= % time the radio is on)

User-defined Generators

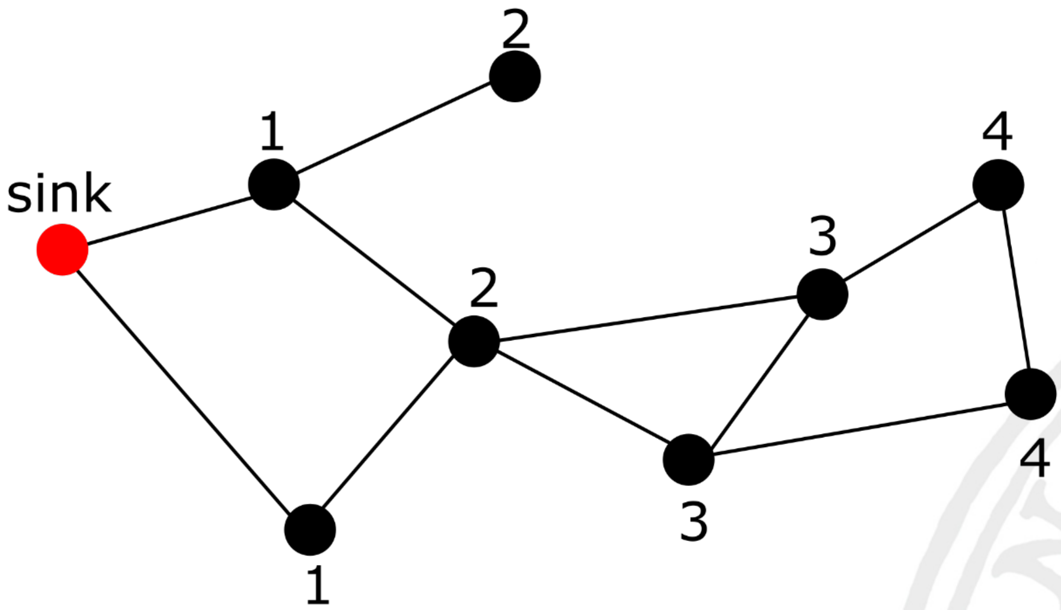
A generator for random graphs with **N** nodes:

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

edge(Vs) ->
  ?SUCHTHAT({V1, V2}, {oneof(Vs), oneof(Vs)},
            V1 < V2).
```

Great: We can easily generate random sensor networks!

Distance from Sink



On this graph, the maximum distance to sink is 4

Is there a large network with **N** nodes where the maximum distance to the sink > **N/2**?

Distance from Sink

```
prop_max_distance(N) ->
  ?FORALL(G, graph(N),
    begin
      D = lists:max(distance_from_sink(G)),
      D < (N div 2)
    end).
```

```
2> proper:quickcheck(demo:prop_max_distance(42)).
.......... 100 dots ..........
OK: Passed 100 tests
true
3> proper:quickcheck(demo:prop_max_distance(42), 100000).
.......... 100000 dots ..........
OK: Passed 100000 tests
true
```

Possible Solutions

- Write more involved (custom) generators?
- Guide the input generation:

  use a search strategy, and

  introduce a feedback-loop in the testing.

Targeted Property-Based Testing

- Combines Search Techniques with Property-Based Testing.
- Automatically guides input generation towards inputs with high probability of failing.

- Gather information during test execution in the form of **utility values** (**UVs**).
- UVs capture how close input came to falsifying a property.

Targeted Property-Based Testing

Utility Value

```
prop_max_distance(N) ->
  ?FORALL_TARGETED(G, graph(N),
    begin
      D = lists:max(distance_from_sink(G)),
      ?MAXIMIZE(D),
      D < (N div 2)
    end).
```

Now `prop_max_distance(42)` fails after 1,548 tests (on average).

Targeted Property-Based Testing

Simulated Annealing requires a Neighborhood Function (NF)

nf(Base, Temperature)

- Returns a neighbor (similar value) to a given *Base* value
- Neighbor distance can be scaled by the *Temperature*

Hand-written NF

```
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)).
```

Hand-written NF

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

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))).
```

Neighborhood Function

- Neighborhood functions are significantly harder to write than random generators
  31 vs **5** lines of code
- Must preserve all constraints and invariants of the input
- Makes TPBT difficult to use

Automating Targeted Property-Based Testing

- Construct the neighborhood function automatically from a random generator:
  - Random generator is problem-specific.

- Idea:
  - Reenact the decisions of the random generator.
  - Instead of choosing variables randomly, we choose values in the neighborhood of the previously generated one.

Example: Edge Generator

Constraint

Tuple Generator

```
edge(Vs) ->
    ?SUCHTHAT({V1, V2},
              V1 < V2).  {oneof(Vs), oneof(Vs)},
```

```
Vs = [1,2,3,4,5,6,7,8,9,10]
```

```
Base Value: {4,8}
```

Rule - for each element:

- Generate a random new element
- Generate a neighbor, or
- Leave as it is

Example: Edge Generator

Tuple Generator

```
edge(Vs) ->
  ?SUCHTHAT({V1, V2}, {oneof(Vs), oneof(Vs)},
            V1 < V2).
```

```
Vs = [1,2,3,4,5,6,7,8,9,10]
```

```
Base Value: {4,8}
```

Leave as it is

Change to a neighbor

Example: Edge Generator

oneof Generator

```
edge(Vs) ->
  ?SUCHTHAT({V1, V2}, {oneof(Vs), oneof(Vs)},
           V1 < V2).
```

```
Vs = [1,2,3,4,5,6,7,8,9,10]
```

```
Base Value: 8   Neighbor: 5
```

Rule - exchange to a random element

Example: Edge Generator

Constraint

Tuple Generator

```
edge(Vs) ->
    ?SUCHTHAT({V1, V2},  {oneof(Vs), oneof(Vs)},
              V1 < V2).
```

```
Vs = [1,2,3,4,5,6,7,8,9,10]
```

```
Base Value: {4,8}  Neighbor: {4,5}
```

- Check constraint:
  - If fullfilled: done
  - Else: retry

Automating Targeted Property-Based Testing

- Rules for all built-in types of PropEr.
- More complex types (constructed with `?LET`) require some additional techniques:
  - Matching
  - Caching
- It is possible to adjust the generation process by overwriting rules with own ones.

Limitations

- Certain nested generators with multiple `?LET`
  - if the constraints for the inner generators depend on values generated by the outer generators.
- Recursive generators.

UPPSALA UNIVERSITET

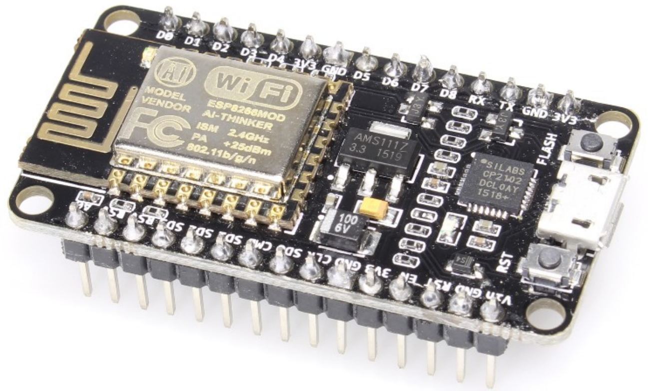
PBT of Sensor Networks

Setup:

- Sensor network.
- Random distribution of UDP server and client nodes.
- Client node periodically sends messages to server node.

Test:

- Has X-MAC for any network a duty-cycle > 25%?



(duty-cycle ::= % time the radio is on)

Case Study 1

Random PBT

- **Mean Time to Failure: 7h46m**

Targeted PBT with hand written NF (100 loc)

- **Mean Time to Failure : 2h12m**

Targeted PBT with constructed NF

- **Mean Time to Failure : 2h19m**

Case Study 2

$$\frac{i(pc) = Noop}{\begin{array}{|c|c|c|}\hline pc & s & m \\ \hline\end{array} \Rightarrow \begin{array}{|c|c|c|}\hline pc{+}1 & s & m \\ \hline\end{array}} \qquad (\textsc{Noop})$$

$$\frac{i(pc) = Push\ v}{\begin{array}{|c|c|c|}\hline pc & s & m \\ \hline\end{array} \Rightarrow \begin{array}{|c|c|c|}\hline pc{+}1 & v : s & m \\ \hline\end{array}} \qquad (\textsc{Push})$$

$$\frac{i(pc) = Pop}{\begin{array}{|c|c|c|}\hline pc & v : s & m \\ \hline\end{array} \Rightarrow \begin{array}{|c|c|c|}\hline pc{+}1 & s & m \\ \hline\end{array}} \qquad (\textsc{Pop})$$

- Definitions for an abstract machine.
- Test: Do these definitions fulfill a certain security criteria?

  (**Noninterference**)

Cătălin Hriţcu, et al. "Testing noninterference, quickly." *Journal of Functional Programming*, 26 (2016).

Case Study 2

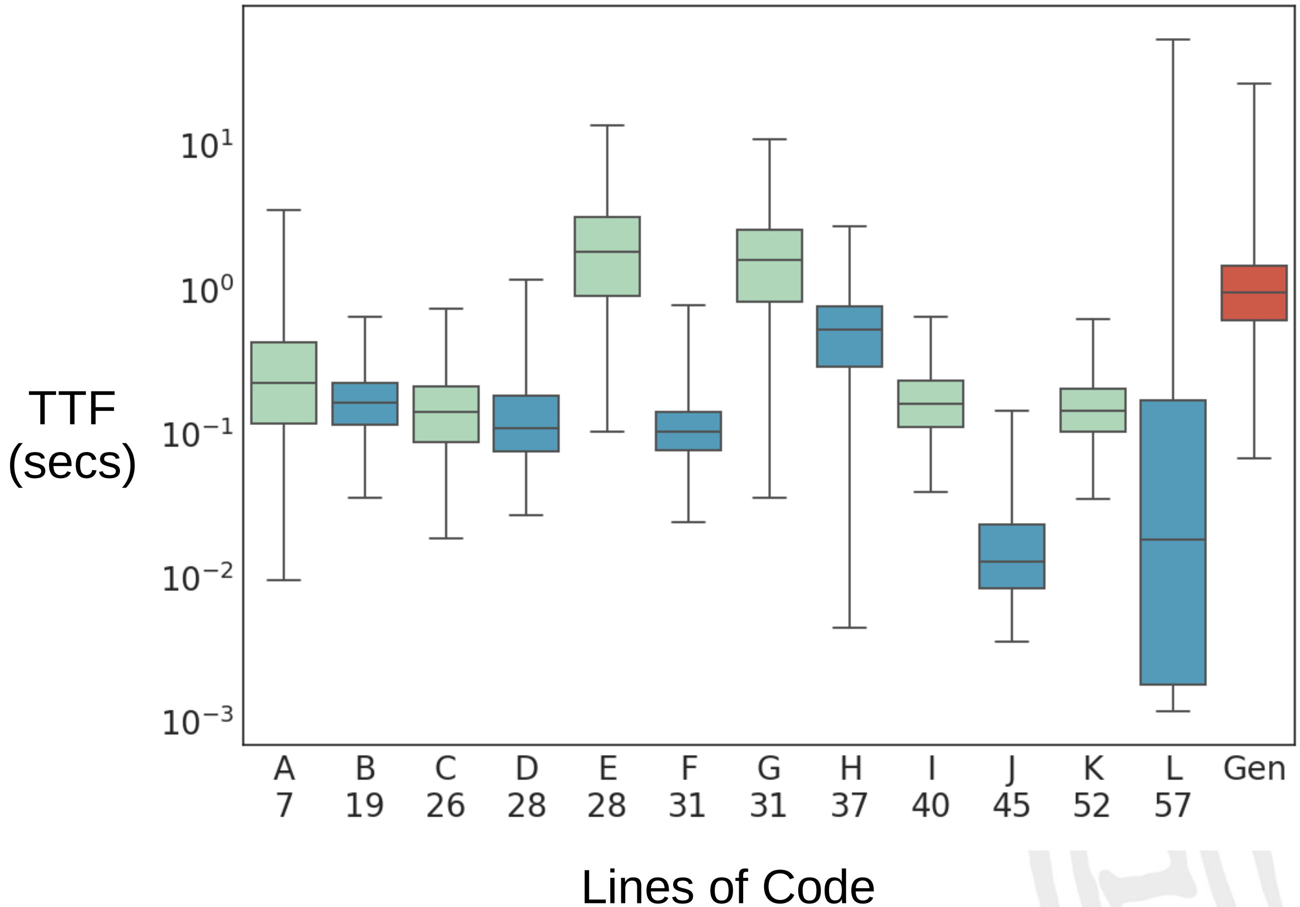
- **Random PBT - Sequence**: programs are a random list of instructions chosen with a fine-tuned weighted distribution
- **Targeted PBT - List**: hand written NF for lists; list elements are either added new or removed
- **Targeted PBT – Constructed:** constructed NF from the Sequence generator

	Random PBT	**Targeted PBT**	**Targeted PBT**
	Sequence	*List*	*Constructed*
ADD	5800,57	271,68	489,93
LOAD	7764,15	341,30	447,25
STORE A	16997.81	2634,80	3685,32

User Study

- We asked students from an advanced functional programming course (M.Sc.) to program a NF for testing a targeted property.
- All students were familiar with random and targeted PBT.
- Compared the hand-written NFs to the constructed one.

User Study



TTF (secs)
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
A 7
B 19
C 26
D 28
E 28
F 31
G 31
H 37
I 40
J 45
K 52
L 57
Gen
Lines of Code

Targeted Property-Based Testing

```
prop_max_distance(N) ->
  ?FORALL_TARGETED(G, graph(N)),
    begin
      D = lists:max
      ?MAXIMIZE(D)
      D < (N div 2)
    end)).
```

Utility Value

Now **prop_max_distanc**
(on average).

Hand-written NF

```
graph_next(G, _T) ->
  Size = graph_size(G),
  ?LET(NewSize, neighboring integer(Size),
                                  _integer(Size div 10),

                                   =
                                   of

                                  Size - NewSize), Additional};

                                  ditional + (NewSize - Size)}

                                  edges(G, Removals),
                                  l, Additions))
```

PropEr

A QuickCheck-Inspired Property-Based Testing Tool for Erlang

http://proper.softlab.ntua.gr/

ICST * Artifact * AEC * Evaluated
Consistent * Complete * Well Documented * Easy to Reuse *

Examp

```
edge(Vs) ->
  ?SUCHTHAT({V1, V2}, {one
            V1 < V2).

Vs = [1,2,3,4,5,6,7

Base Value: {4,8}
```

Leave as it is

Change to a neighbor

User Study