Property-Based Testing of Sensor Networks

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

***Abstract*—We advocate the use of property-based testing in the area of sensor networks and present a framework to apply this testing methodology. Our framework provides an expressive high-level language to specify a wide range of properties, starting from properties of individual functions to network-global properties, and infrastructure to automatically test these properties in COOJA, the network simulator of the CONTIKI operating system. We demonstrate the ease of use and effectiveness of our framework by two case studies. In the first, we test whether the energy consumption of the radio duty-cycle protocol X-MAC is within some specific bound. Property-based testing finds minimal network configurations where a small number of nodes violate the property. Property-based testing also reveals that the same property is not violated when ContikiMAC is used instead, but finds cases where ContikiMAC has higher energy consumption than X-MAC. In the second case study, we test the C API of CONTIKI's TCP socket library and find bugs in its event system that would be very hard to detect with other methods.**

I. INTRODUCTION

Testing is an integral part of modern software development. It increases the confidence in the correctness of software and helps in uncovering bugs and avoids repeating them. The needs of the development process for software of sensor networks are no different in this regard. In fact, arguably it is even more crucial to rigorously test these systems as they are rapidly becoming commercial these days and used in critical domains such as e.g. process control [1] and e-health [2]. Also, testing and fixing their software is considerably more difficult and costly after their deployment. Last but not least, a software bug triggered during operation might not only affect a single node, but might bring down the whole network.

Still, it is not uncommon to deploy sensor networks that, although tested beforehand using traditional testing techniques such as unit and regression testing, contain serious bugs that prevent them from operating as expected [3]–[6]. Part of the reason for this is that often in practice developers create only a few test scenarios to check their software. Creating multiple tests is a time-consuming and boring task, since on top of writing the test case for the application software one must specify the network topology and find a way to control the used network simulator or testbed. Thorough testing requires using a plethora of test cases that cover even unlikely corners.

To improve on the state-of-the-art of testing sensor networks, in this paper we advocate the use of property-based testing in their development process and present a framework to apply this testing methodology. Our framework comes with an expressive high-level language to specify a wide range of properties, starting from properties of individual functions to network-global properties, and infrastructure to automatically test these properties in COOJA, the network simulator of the CONTIKI operating system [7]. A component of our framework, which we developed from scratch, allows us to test the sensor node software at the level of individual C functions.

We show that our framework is effective and easy to use, by presenting two case studies that test various aspects of CONTIKI's code base. The first of them illustrates testing the energy efficiency of sensor network protocols in general, and the implementation of two MAC protocols in particular. The second case study tests the newly introduced socket API of CONTIKI, and exposes three subtle bugs in its event system.

The rest of the paper is structured as follows. We start by presenting property-based testing and some aspects of the tool we will employ. The next two sections form the main part of this paper: Section III presents the design and implementation of our testing framework, and Section IV the results we have obtained in two case studies. The paper ends with discussing practical aspects of our framework (Section V), comparison with related work (Section VI), and some concluding remarks.

II. PROPERTY-BASED TESTING

Property-based testing (PBT) is a novel approach to testing, where one only needs to specify the generic structure of valid inputs to the system under test (SUT), along with a number of properties of the system's behaviour that are expected to hold for every valid input. A PBT tool, when supplied with this information, will automatically produce progressively more complex random valid inputs, then apply those inputs to the (program implementing the) SUT while monitoring its execution, to test that it behaves as expected. By 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 easy to write and understand than full 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 test cases, 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 bugs.

Because test inputs are generated randomly in PBT, the part of a failing test case that is actually responsible for the falsification of a property can easily become lost inside a lot of irrelevant data. Thus, PBT tools often aid the programmer in extracting that part by simplifying the counterexample, through an automated process called *shrinking*. In most cases, shrinking works in the same way as a human tester would approach debugging: by consecutively removing parts of the failing input until no more can be removed without making the test pass. This "minimal" test case will serve as a good starting point for the debugging process. The shrinking process can often be fine-tuned through user-defined shrinking strategies.

A test run of a single property typically happens as follows:

1) Randomly generate valid instances for each universally quantified variable in the property, using the generator that the user has provided for each such variable.
2) Call the property code with this input.
3) If the property evaluated to `'true'`, repeat from 1. Else:
4) While it is possible to shrink the test case to one that is simpler and fails the property in the same way, shrink it.
5) Report the failing test case and the shrunk input.

In PBT tools, such as the one we employ, this process can typically be configured in various ways through options. For example, users can control the number of tests to run, the size of produced inputs, the number of shrinking attempts, etc.

Let us illustrate PBT and PropEr [8], the tool we use, with an example. Suppose we want to test the implementation of a network protocol that provides functions for encoding and decoding. A natural property that we may be interested in checking is that for all valid inputs I, if one encodes I and then decodes its encoded version, one ends up with the original input I. In the language of PropEr, this can be specified as:

```
prop_encode_decode() ->
  ?FORALL(I, input(), I == protocol:decode(protocol:encode(I))).
```

This code snippet, written in the high-level functional language Erlang, assumes that the implementation of the protocol is provided in a module named `protocol` that provides `encode` and `decode` functions. `I` is a variable that takes values from the *generator* `input()`, a function that generates random inputs which in this case are protocol-specific. For simplicity, let us assume that the protocol operates over strings of ASCII printable characters. In this case the input generator is:[1]

```
input() ->
  list(range(32, 127)).
```

Furthermore, assume that the implementation of the protocol is buggy for strings whose length is in the range [17..23]; i.e. for such strings the property does not hold. Below we show the actual output generated by PropEr when checking this property in the Erlang shell:

```
Eshell V6.3  (abort with ^G)
1> proper:quickcheck(protocol_test:prop_encode_decode()).
...............................................................!
Failed: After 64 test(s).
[45,80,58,119,94,62,118,71,71,119,114,123,75,67,62,84,99,60,61,86,67]

Shrinking ...................(19 time(s))
[32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32]
false
```

Here we see that PropEr ran a total of 63 successful random tests before generating an input, a string of length 21, that falsifies the property. Subsequently, PropEr shrank this input, both in length and in "size" of its elements, down to a minimal input of length 17 for which the property does not hold. It is important to realize that the testing process shown above is completely automatic. (Also, in this case, it is very fast.)

The language of PropEr is very powerful, offering significantly more than what is shown in this simple example. In particular, the user can specify much more involved properties consisting of several input variables, write generators that have more complex underlying structure (e.g., are balanced trees of some sort, have the format of an IP packet, etc.). More importantly, PropEr also comes with support for testing of *stateful* systems, i.e., systems whose operation follows some (finite) state machine model, where states and transitions between them are associated with pre- and post-conditions. In Section IV we will employ some of this support, but, for lack of space, we refer the reader to the manual and tutorials in PropEr's website [9] for more information about the above.

III. Our Framework: Design and Implementation

Let us now describe our framework for property-based testing of sensor networks. As shown in Fig. 1, it consists of several components. Besides PropEr, to automatically test properties of interest, our framework uses Cooja to simulate the SUTs. Since PropEr can not directly control Cooja, we implemented a control layer to perform this task. To enable function-level testing, we created Nifty, an interface generator to call C functions from Erlang. Nifty generates a Contiki application that must be compiled together with the firmware, and a library in Erlang that forwards the function call interface on the sensor nodes using a Cooja plugin.

A. Cooja

We use Cooja, the network simulator of the Contiki operating system to simulate the systems we test. Cooja is a cross-platform simulator and uses different hardware emulators to simulate a variety of sensor nodes like TMote Sky, Zolertia Z1 and MicaZ. Simulating the hardware of the nodes has the advantage that the firmware that is compiled for real hardware can be used in the simulation without modification. Cooja also contains multiple radio models. These radio models range from topology-based ones, where each connection between nodes is explicitly defined, to more complex ones that model signal loss over distance, interference and packet corruption according to the ongoing radio traffic.

[1] In fact, one does not even need to write such a definition, if the program defines a type named `input`. In this case, the generator and all infrastructure required for shrinking input values is created automatically by PropEr.

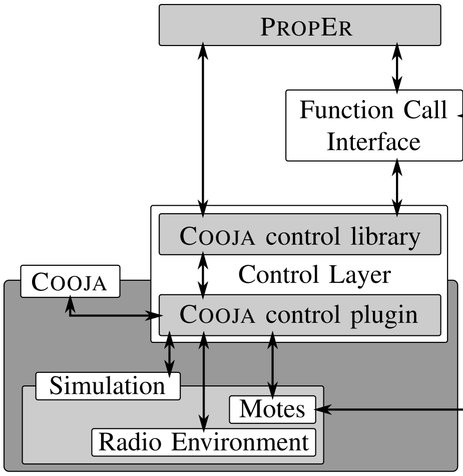

Fig. 1. Our testing framework: overview of the architecture.

As mentioned, PROPER can not directly control the simulation and we need a control layer to perform this task. Since COOJA and PROPER are implemented in different languages (Java and Erlang) our control layer consists of two components as well: an Erlang library to be used together with PROPER, and a Java plugin that directly interacts with COOJA. The Java plugin operates a distributed Erlang node which allows both components to communicate over the Erlang distribution mechanism using TCP/IP. The Erlang library of our control layer provides high-level functions for all of the functionality that the plugin offers. When we call a function in the Erlang library, a message representing the requested operation is constructed and sent to the COOJA plugin. The plugin deconstructs this message and triggers the requested action. If a return value is required, then a message containing this value is sent back to the library.

The control layer is thus able to control most aspects of the simulation. On a network-global level, we can control the configuration of the radio environment and change the topology of the network. The radio configuration depends on the used radio medium. For the graph-based radio medium we can specify the links between nodes and the corresponding link quality. For the radio medium that models loss over distance (Unit Disk Graph Medium: Distance Loss) we can specify the transmission range and how link quality decreases over distance. It is also possible to record all messages that are sent over the radio medium. These messages can be further analyzed to reason over the correctness of the used communication protocol. Our control layer provides a parser for IPv6 packets, making their analysis easier.

The control layer provides the functionality to add and remove sensor nodes from the simulation at any time. Removing a node can for example be used to simulate node failure. It is also possible to change the position of the nodes. This can be used to simulate mobile nodes.

It is possible to send and read messages from the serial line of the nodes. Messages on the serial line are delimited by newline characters and stored in a FIFO queue that is accessible from the control layer.

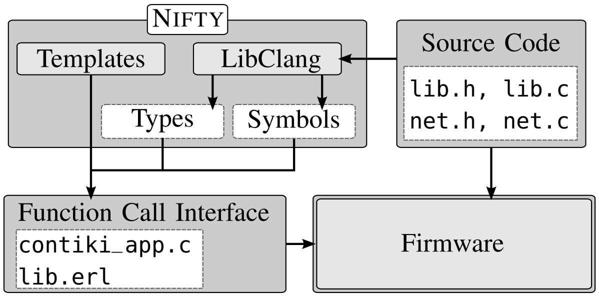

Fig. 2. Overview of the interface generation process.

Similar to the recorded radio messages of the radio medium, it is possible to record all hardware events from the nodes. These hardware events are usually “on”/“off” events for the simulated hardware. The radio hardware yields events that indicate packet reception and transmission. This includes events where the radio hardware detects packet interference.

B. NIFTY

To effectively test the software that controls a sensor network, we need to be able to test the sensor nodes' implementation. Software for the CONTIKI operating system is usually written in C. Thus, we need to be able to call single C functions on the sensor nodes. This way we can directly test protocol implementations and other core components. For this reason we developed NIFTY, a function call interface generator.

NIFTY processes a C header file that typically contains type and function declarations and creates a call interface for each function defined in the header file. If we want to call the functions in a generic way, we need to be able to tell the call interface which function should be called with which arguments. The best way to communicate with the sensor nodes from our control layer is by using the serial line of the nodes. NIFTY therefore generates an interface that operates a function call protocol over the serial line.

To call a function, NIFTY sends a string representing the function call with the arguments over the serial line to the node that should execute the call. The call interface of the node deconstructs the string and calls the function with the right arguments. The return value of the called function is sent back over the serial line in a similar manner. Additionally, NIFTY generates an Erlang library that performs these steps automatically and hides them behind regular library functions. Calling a function on a simulated sensor node is not different than calling any library function.

Fig. 2 illustrates the interface generation process. NIFTY uses LibClang [10], a high-level C interface for the Clang compiler, to parse the given header files. This way, we can extract the relevant type and function information from the abstract syntax tree provided by LibClang. Anything that involves the C preprocessor, like defines and includes, are resolved by the compiler library. Type information is stored into a type table that contains all relevant information needed to construct the call interface. A symbol table stores information for functions, such as the types of their arguments and their return.

After processing the input and creating the type and symbol table, Nifty uses a template engine to generate the source files of the call interface. This template engine contains the static parts of the output, combined with template tags that encode the dynamic parts. When the actual output is generated, the template tags are rendered with the information stored in the type table and the symbol table, to create the corresponding call interface. Finally, the generated interface has to be compiled together with the other source files to build a usable firmware.

The created call interface is implemented as a Contiki application that operates a node's serial line and listens for messages. If a message is received, it is parsed by the call interface.

A message that represents a function call starts with the number of the function followed by the values of the arguments. The function number is assigned at the time the interface is generated. Positive function numbers are used for the functions that are defined in the header file that Nifty used to generate the interface. Negative function numbers are used for utility functions. These utility functions provide basic support for dynamic memory management and are equivalents to the standard C functions `malloc`, `free`, and `sizeof`. Additionally the interface provides functions to read and write memory. These utility functions are always part of the call interface.

In addition to these synchronous function calls, we found it necessary to provide a mechanism that allows it to get feedback from asynchronous function calls like callback functions. Nifty provides an event message type. Event messages are messages prefixed with `"EVENT:"` and put in an dedicated FIFO queue. These messages are otherwise ignored by the call protocol. A callback function can send event messages to yield values to the control layer.

C. Using the Framework

Fig. 3 shows how we use our framework for property-based testing of sensor networks. A sensor network is defined by its sensor nodes, their firmware, and its network topology. The topology defines how the sensor nodes are connected with each other. The behavior of the node software, especially of the network components, depends on the network topology.

We can define the topology in a file that the simulator loads when it is started. The sensor nodes and the network topology are automatically loaded by the simulator. A much more interesting option however is to test our property with many different topologies, created dynamically.

Sensor networks change their state over time. Performing one action can change the result of all subsequent actions. In fact just progressing the simulation in time can alter the state of the sensor network. For example, timers can be triggered or messages can be forwarded. To ensure we have self-contained test cases, it is therefore necessary to restart the simulator before each test case. This way we always start from a known state. Additionally, it is necessary to control in which timesteps the simulation progresses, since the timings of the inputs can alter the result of a test case. We can progress the simulation explicitly by forwarding it a certain amount of time. Some functions of the control layer and all functions of the generated call interfaces forward the simulation implicitly.

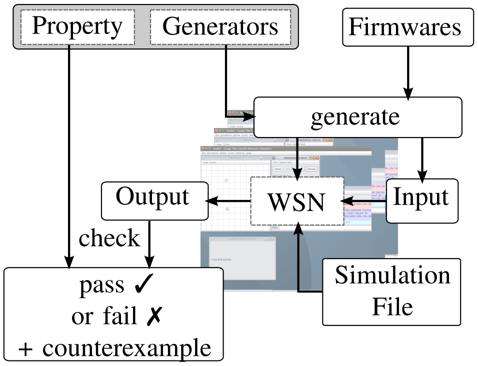

Fig. 3. Overview of the the testing process

Finally, in order to retrieve data from the simulation we need to subscribe to the corresponding data sources. These data sources are the global radio medium, the hardware events of a sensor node, and the serial line of a sensor node.

IV. Case Studies

We now evaluate our framework using two case studies. The first of them illustrates how we can employ property-based testing to test the energy efficiency of sensor network protocols in general, and MAC protocols in particular. We start from this case study because the support it requires from the PBT tool has already been presented in Section II. A second case study shows how we can employ PropEr's support for stateful testing and discover scenarios that show bugs in the socket API of Contiki.

A. Energy Efficiency of MAC Protocols

Energy efficiency is important for sensor networks. During deployments, it is crucial to preserve the battery of the nodes to maximize their lifetime. We therefore test the duty cycle of the nodes' radio component, the component that usually consumes the most energy [11]. Using our PBT framework we examine if there are network configurations for which the duty cycle is above a certain threshold.

Our setting uses two node types: UDP server and UDP client nodes. Client nodes send periodically messages to server nodes. The node types that we use are taken from the `rpl-udp` example of the Contiki distribution, which uses IPv6 and RPL. As a MAC layer, the network layer that performs duty cycling, we use Contiki's implementation of X-MAC [12].

We list the property for this scenario in Fig. 5. This property checks that for all nodes of the network the duty cycle is below 10%. We check this property for randomly generated network topologies. In line 4 of our property, the simulator is started with no nodes. We add the nodes for each test case to the simulator (line 6) and progress the simulation of the generated WSN by two minutes (120 seconds). After the simulation finishes, we calculate the duty cycle of each node and check

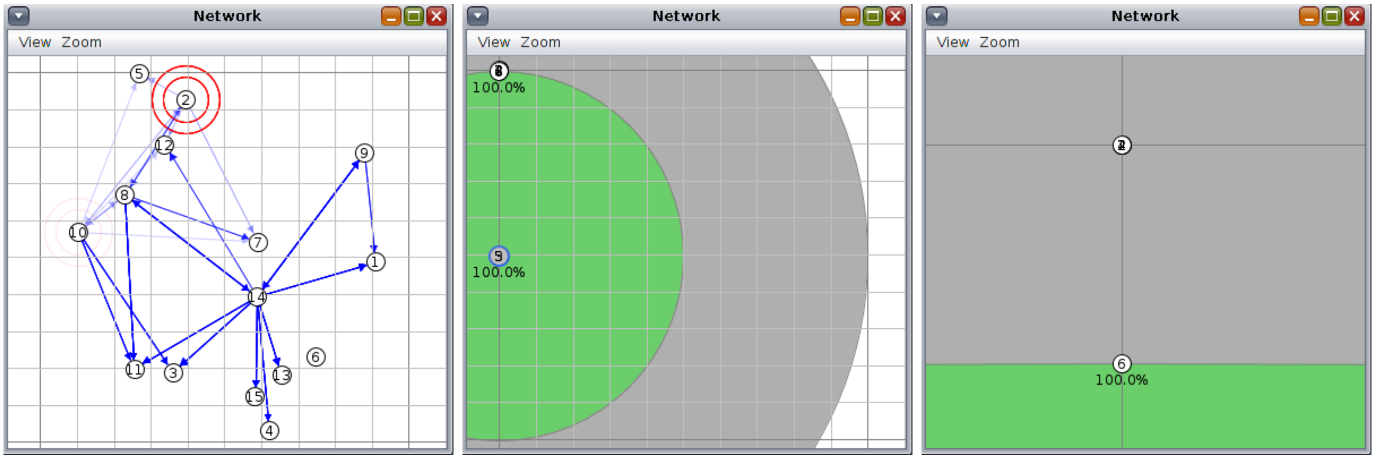

Fig. 4. Shrinking can reduce the size of test cases significantly. The left window shows an automatically generated WSN, consisting of 15 nodes, for which the duty cycle property fails. This network is automatically shrunk down to only six nodes (shown in the middle). The window on the right is a close-up of the shrunk test case, and reveals that all messages between the two cluster of nodes need to be forwarded by node 6, which therefore has duty cycle of 10%.

```
prop_duty_cycle_below_threshold() ->
  ?FORALL(Motes, configuration(),
         begin
           ok = setup()
           {running, Handler} = nifty_cooja:state(),
           Mote_IDs = add_motes(Handler, Motes),
           SimTime = 120 * 1000,  % simulate for 2 minutes (120 secs)
           ok = nifty_cooja:simulation_step(Handler, SimTime),
           MaxDutyCycle = max_duty_cycle(Handler, Mote_IDs, Motes),
           ok = nifty_cooja:exit(),
           MaxDutyCycle < 0.1     % check that is below 10%
         end).
```

Fig. 5. Property to check the duty cycling of a random WSN.

that its maximum is below our 10% threshold. We extract this information from the hardware events by accumulating the difference between all `RADIO_ON` and `RADIO_OFF` events, to get the total time the radio of each node was on, and divide it by the total simulation time.

To generate network topologies, we implemented a generator `configuration()` that creates a list with elements of type `mote()`. A `mote()` is a tuple consisting of a `coordinates()` and a `mote_type()`. Coordinates are 3-tuples of floats and we set their x and y values to be between 0.0 and 100.0 and their z value to 0.0. This creates nodes at positions that are uniformly distributed in a plane of 100×100 units. The transmission range of all nodes is 50 units. Mote types are either UDP clients (`"sky1"`) or UDP servers (`"sky2"`). The following code snippet lists the generators that create network configurations:[2]

```
configuration() ->
  list(mote()).
mote() ->
  tuple([coordinates(), mote_type()]).
coordinates() ->
  tuple([float(0.0, 100.0), float(0.0, 100.0), 0.0]).
mote_type() ->
  weighted_union([{50, "sky1"}, {50, "sky2"}]).
```

We use the control layer of our framework to add the generated nodes to the simulation, and to record the hardware events:

```
add_motes(Handler, Motes) ->
  [add_mote(Handler, Mote) || Mote <- Motes].

add_mote(Handler, {Pos, Type}) ->
  {ok, ID} = nifty_cooja:mote_add(Handler, Type),
  ok = nifty_cooja:mote_set_pos(Handler, ID, Pos),
  ok = nifty_cooja:mote_hw_listen(Handler, ID),
  ID.
```

In our first experiment we test the property of Fig. 5. The property fails on a network configuration of around fifteen nodes. The left window of Fig. 4 shows one configuration that falsifies the property. When this configuration is found, PropEr automatically shrinks it down to a smaller configuration that also falsifies the property. As in the protocol example of Section II that reduced the length of the string, shrinking in this case tries to reduce the amount of nodes to a minimum. It also tries to shrink their x and y coordinates to small values. The shrinking process finds the configuration shown in the middle window of Fig. 4. The shrunken test case contains only six nodes. Three of them form a cluster at position $(x = 0, y = 0)$ and two more nodes form a cluster at position $(x = 0, y = 50.25)$. Both clusters are just barely out of radio range from each other and cannot communicate directly. Therefore, all traffic between those two clusters has to go through node 6 at position $(x = 0, y = 0.25)$, resulting in a duty cycle of 10.07% for this node.

Having a configuration with only six nodes that falsifies this property is an indication that we made a bad choice when choosing our MAC layer. ContikiMAC [13] is generally regarded as more energy efficient than X-MAC [14]. Therefore, our second experiment is to switch the MAC layer to ContikiMAC and rerun the test on this network configuration.[3] Now, node 6, which failed the property for X-MAC, has a duty cycle of only 1.5% with ContikiMAC. We continue this experiment by

[2] In `mode_type()`, we specify that the mote types are selected with equal 50% probability only to show how different probabilities could be specified.

[3] PropEr provides support for saving and reusing a failing test case [9].

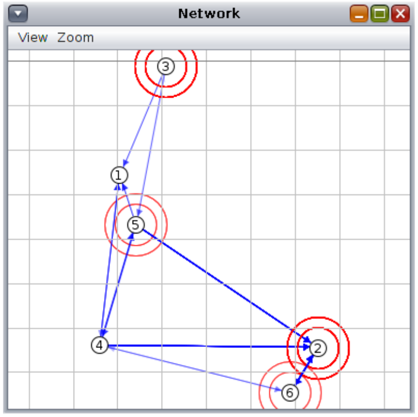

Fig. 6. A sensor network topology, consisting of only six nodes, for which some node(s) have lower duty cycle using X-MAC rather than ContikiMAC.

testing $1,000$ randomly generated configurations. For all of them, PBT finds that the property holds. This is not a proof, but it increases our confidence that the property is actually true in this particular setting.

This second experiment might suggest that ContikiMAC has a lower energy consumption than X-MAC. Is this always the case? We do not have to do a complicated analysis! Using our framework, we can simply write a property that tests that this is true for all topologies. The relevant code is shown below:

```
prop_ContikiMAC_more_efficient() ->
  ?FORALL(Motes, configuration(),
    begin
      Contiki_MAC = duty_cycles(Motes, contiki),
      XMAC = duty_cycles(Motes, xmac),
      lists:all(fun ({C,X}) -> C < X end, lists:zip(Contiki_MAC,XMAC))
    end).
```

This property tests that, for all randomly generated sensor network configurations, all their nodes have a lower duty cycle when using ContikiMAC than when using X-MAC. Testing this property, however, quickly reveals that it is not true. Our framework finds a sensor network with only six nodes, shown in Fig. 6, where at least one node has a higher duty cycle when using ContikiMAC.

This third experiment makes a more general point. With property-based testing it is very easy to compare the performance of two "similar" implementations for a wide variety of different networks. Moreover, a PBT tool's shrinking ability provides us with test cases that are much easier to analyze than the generated test cases that are found to falsify the property.

Note however that in a PBT tool there is a trade-off between specifying simple generators purely based on types without any domain knowledge and the time required for failing test cases to shrink. For example, during our experiments we experienced that while finding counterexamples (network configurations that falsify the stated properties) required only 10 to 40 test runs, their shrinking took much longer. In our first example (duty-cycling of X-MAC) around 2000 additional simulations were run for the shrinking.

The problem is in how we generate the network layout by randomly placing the network nodes in a square. We let the network topology automatically be determined by the simulation: two nodes can communicate if and only if they are within radio range. The shrinking algorithm cannot take into account this topology information. It shrinks the counterexample according to the generator used, which means that the numerical values of the node positions are shrunk towards 0. This results in many shrinking steps that either have no effect or alter the network topology in an unforeseen way.

PROPER comes with support for writing generators that represent the application domain much better. For example, in this case study we can specify the network configuration as an undirected graph, where the graph nodes represent the network nodes and the edges represent the ability of two nodes to communicate with each other. A generator that creates such a graphs according to the Erdős–Rényi model [15] can be implemented as follows:

```
er_graph({Nominator, Denominator}) ->
  WeightLink = Nominator,
  WeightNoLink = Denominator-Nominator,
  ?SIZED(Size, er_graph(Size, WeightLink, WeightNoLink, [], [])).

er_graph(0, _, _, V, E) ->
  {V, E};
er_graph(Size, WeightLink, WeightNoLink, Vertices, Edges) ->
  ?LET(LinksDef, vector(length(Vertices), edge(W1, W2)),
      begin
        NewVertex = length(Vertices),
        NewEdges = build_edges(LinksDef, Vertices, NewVertex),
        ?LAZY(er_graph(Size-1, WeightEdge, WeightNoEdge,
                       [NewVertex|Vertices], NewEdges++Edges))
      end).

edge(W1, W2) ->
  ?LAZY(weighted_union([{W1, true}, {W2, false}])).
```

This generator produces random graphs by recursively adding one vertex to the graph at a time and creating the edges between the newly added vertex and all other vertices with the given probability. (Refer to PROPER's manual [9] for the explanation of `?SIZED`, `?LET` and `?LAZY`).

While such a generator is more complex than the original one, it defines the network topology explicitly. The shrinking will now decrease the number of nodes and links between them. Each shrinking step is a meaningful one and will produce a smaller and less connected network, which makes the shrinking much faster. This graph model shrinks in only 50 to 150 steps.

B. CONTIKI's *Socket API*

Recently, CONTIKI got a new API for TCP and UDP sockets to replace the old proto-socket interface. In this API, functions that require network communication are non-blocking. For example, the function `tcp_socket_connect` will return immediately. When the connection has been established successfully, an `"established"` event will be triggered via an event callback function. We wanted to test that the new API behaves correctly when it establishes and terminates connections.

Something to note regarding testing this socket API is that its functions have to be called in a specific order. For example, it does not make sense to try to connect to a socket that is not even listening. In such situations, PROPER's support for *finite state machine* testing [9] comes in handy.

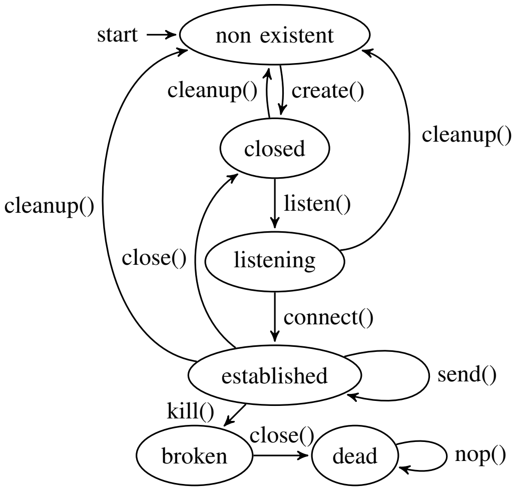

Fig. 7. The FSM that we used to generate commands to test the socket API.

In contrast to the previous set of experiments, here we will use a fixed network configuration. The setup is a simple multi-hop sensor network consisting of four Zolertia Z1 nodes. These nodes are positioned in a circle so that each node can only communicate with its two neighbors. We use CONTIKI's IPv6 network layer together with RPL for routing. The simulator's radio environment is configured to have no loss. That means that no packets are lost due to a decreasing signal quality over distance.

In order to expose the socket API, we generated a NIFTY function call interface for it. Additionally, we created interfaces for some auxiliary functions like accessing the IP address of the nodes.

We tested the interface by executing a randomly generated sequence of API functions. Our property was quite simple:

```
prop_socketAPI() ->
  ?FORALL(Cmds, proper_fsm:commands(?MODULE),
    begin
      ok = setup(),
      {_H, {_S,_SD}, Result} = proper_fsm:run_commands(?MODULE, Cmds),
      ok = nifty_cooja:exit(),
      Result == ok
    end).
```

In words: for all random sequences of commands specified in this test module, set up the simulation environment (the code for this function is not shown), run the sequence of commands on the simulator getting back a result, exit the simulator, and consider the property successful if the result was 'ok'.

To generate command sequences, we used the support of the proper_fsm module to specify a finite state machine (FSM) that dictates the order in which the API functions can be called. The states of this FSM and transitions between them are shown in Fig. 7. In the language of PROPER, the transitions of this FSM are specified as follows:

```
closed(S) ->
  [{listening, {call, ?MODULE, listen, [sockets(S), tcp_port()]}},
   {non_existent, {call, ?MODULE, cleanup, [sockets(S)]}}].
```

This specifies the transitions out of the closed state of the FSM. It states that from this state we can transition to the listening state by executing the command listen with the listed symbolic arguments sockets(S) and tcp_port(). We can also transition into the state non_existent by executing the command cleanup with the argument sockets(S). When a command is executed, the arguments to the command are generated as well. So the tcp_port() call generates a suitable port number. The commands() generator of the proper_fsm module creates random sequences of commands that respect the transitions of the specified FSM.

To check that the result of executing the commands is OK, each of the commands has to satisfy a set of post-conditions, which PROPER also allows us to specify. For example, a post-condition for closing the connection looks like this:

```
postcondition(_, _, _, {call, _, close, _}, Result) ->
  {M1, M2} = Result#socket_state.motes,
  T = ?TIMEOUT_CLOSE,
  check_event(M1, "closed", T) andalso check_event(M2, "closed", T).
```

This post-condition evaluates to true if the two nodes of a TCP connection trigger a "closed" event before the given timeout. If at least one of them does not trigger such an event, the post-condition evaluates to false. If this happens we have found a test case that falsifies our property.

When run in our framework, the property fails quickly after establishing a connection with the following command sequence:

```
create -> listen  -> connect -> close   -> listen -> connect
       -> cleanup -> create  -> cleanup -> create -> listen
```

Shrinking in this case means reducing the length of the command sequence as well as shrinking the "size" of any values in arguments of the commands. (This is not shown here.) After shrinking the failing test case, we end up with a minimal test case of just four commands:

```
create -> listen -> connect -> close
```

This command sequence fails because the post-condition of the close command fails. When we rerun the test case in the simulator and observe the output of the nodes, we can see that the cause of the property failing is a supposedly received message of length 0. Receiving a message triggers a "received" event. However we have not been sending a message. This bug occurs on every successful connection.

We decided to bypass this bug and continue testing the interface in order to perhaps find more interesting bugs. We can bypass this bug by adjusting the post-condition for the command connect to be as follows:

```
postcondition(_, _, _, {call, _, connect, _}, Result) ->
  {M1, M2} = Result#socket_state.motes,
  T = ?TIMEOUT_CONNECT,
  check_event(M1, "established", T) andalso
    check_event(M1, "received", T) andalso
      check_event(M2, "established", T).
```

After this change, the property fails again. After shrinking, we end up with sequence of six commands:

```
create -> listen -> connect -> close -> listen -> connect
```

This command sequence fails because the post-condition of the connect command evaluates to false. Rerunning the test case reveals a new problem. The close command triggers the "closed" event twice on the socket that calls the tcp_socket_close() method of the API. We observe the first "closed" event in the post-condition of the close command and continue executing commands. The next time we check for events, we will observe the second "closed" event and our property fails.

At this point, we should probably have stopped testing and have tried to fix the bugs we found. Both of them indicate problems in the implemented event system. However we did not do that. As an experiment, we ignored the double "closed" event in our property, again by appropriately modifying some post-condition, and reran the tests.

The property failed again! In fact, it now failed with a sequence of 14 commands, which was shrunk down to the following sequence:

```
create -> listen -> connect -> cleanup -> create ->
          listen -> connect -> close (on socket that listened)
```

Executing this command sequence, we can not observe any "closed" event on the socket that did not execute the close command, although the other socket triggers such an event.

The first two bugs we found occur every time a connection is established or closed and are easy to catch with traditional testing methods. This final bug, however, is only triggered by a particular sequence of commands. Property-based testing is especially effective in finding these kind of bugs. It is very easy to generate a large number of test cases after specifying a property. Eventually the bug will be triggered.

We reported all three bugs to the issue tracker of the Contiki developer repository [16], [17]. At the time of this writing, the double close event bug is already confirmed by another user; we expect the other two bugs to be confirmed by the Contiki developers.

V. Discussion

The property specification language that PropEr offers is Turing complete. In principle, it allows us to express any possible property. We can test for functional correctness as well as non-functional properties (e.g. timing, energy consumption, etc.). In practice, the testing is limited by the capabilities of the simulator, since we can only test behavior that we can observe. A notable limitation in this regard is the simulation of the radio medium. While Cooja offers multiple radio models, the behavior of real world radio environments is more complex.

The call interfaces that Nifty generates occupy the serial line of the sensor nodes. This means that other applications are not able to use it anymore without interfering with the call interface. In practice this is not a problem since deployments usually do not use the serial line since it contradicts with the basic notion of going wireless.

For APIs with a high number of functions Nifty interfaces can have a large memory footprint. Firmwares can become too big for very resource constrained devices. We are alleviating this problem by offering options to switch off specific memory demanding features in the generated interface. Using the serial line for the call interface can have an effect on tightly timed systems. Contiki is a cooperative multi-tasking system, which means that while we are executing a function call all other tasks are blocked from execution. In practice long running function calls should be avoided when testing those systems.

Scalability-wise, our approach is currently limited by the performance of the simulator which is used; not the PBT tool.

VI. Related Work

The idea of property-based testing and using a high-level language for testing purposes is not new. In the context of programming languages has been pioneered by the QuickCheck library [18] for the lazy functional language Haskell. Similar tools have been developed for other languages, including the commercial Erlang QuickCheck tool and the open-source PropEr tool, and these tools have been applied to test telecom software [19] and Web Services [20]. However, to the best of our knowledge, our work is the first one to apply property-based testing in the area of wireless sensor networks (WSNs).

Still, a variety of other testing and verification techniques have been explored before for sensor networks. KleeNet [21] is a debugging environment that extends the technique of KLEE [22] to WSNs. It executes unmodified sensor network applications on symbolic input and automatically injects non-deterministic failures like packet corruption or packet duplication. KleeNet is independent of the underlying operating system, but each OS has to provide a front-end to KLEE, which abstracts from the sensor node hardware. T-Check [23] is an explicit model checker for TinyOS applications built upon the TOSSIM [24] simulator. The model checker emulates the hardware on the level of TinyOS interfaces which abstracts for low-level interrupt driven concurrency for the sake of scalability. Bucur and Kwiatkowska present a tool for software verification of single MSP430-based wireless sensor nodes [25] . Their tool translates embedded C into standard C, which is then checked with CBMC, a software verifier for ANSI C. Anquiro [26] is a domain-specific model checker for statically verifying the correctness of sensor network software. It abstracts from the low-level functionality of the sensor node hardware and radio communication to be able to check larger sensor networks.

All the above techniques are good in finding errors since the inherent method of exploring all states or execution paths will find these bugs with certainty. These tools however operate on a model of the sensor network (Anquiro) or abstract away many aspects from the actual sensor node hardware (KleeNet and T-Check). The verification framework of Bucur *et al.* has an accurate model of the MSP430 CPU but verifies only one node. In contrast, besides being easier to use, our framework uses the real firmware of the nodes that could be uploaded to a physical node without any modifications. Additionally, the hardware of the sensor node is simulated in Cooja, which means that our framework is able to also find bugs that involve the nodes' hardware. Software verification, model checking, and symbolic execution do not scale well with increasing network sizes and complexity of the software of the nodes. The scalability of our framework depends only on the performance of Cooja, which means, that we can test larger systems which run for a longer time. For example, we are able to test systems with 50 sensor nodes without any problems.

Passive Distributed Assertions (PDA) [27] is a mechanism that allows the sensor network programmer to specify assertions over multiple sensor nodes. The sensor nodes automatically generate traces that later can be evaluated to check if the distributed assertions held. PDAs can be used to find distributed bugs in deployed sensor networks. Property-based testing aims to test sensor networks thoroughly before deployment. The properties are specified in an external language and do not require the modification of the SUT. It is possible to combine PDAs with our property-based testing framework to test existing PDA specifications with generated input.

VII. Concluding Remarks

We have argued for the use of property-based testing in the area of sensor network programming and presented an effective and easy to use framework to apply this testing methodology. The use of a high-level and expressive language for specifying properties and generators, combined with the underlying infrastructure for being able to manipulate low-level C and sensor network simulator code from it, has allowed us to test relatively complex software and uncover subtle and hard-to-find bugs in it. Moreover, the shrinking ability of our PBT tool has managed to produce test cases that make it easier, if not very easy, to reason about the source of these bugs.

In the near future, we plan to apply our framework to more sensor network code; not only to test other components of Contiki's implementation but also actual WSN applications. We encourage others to do alike and hope that this work will pave the way in developing more dependable and reliable sensor networks.

Acknowledgements

This research was conducted under the aegis of the Swedish Foundation for Strategic Research (SSF) project *ProFuN: A Programming Platform for Future Wireless Sensor Networks*.

References

[1] T. O'donovan, J. Brown, F. Büsching, A. Cardoso, J. Cecílio, P. Furtado, P. Gil, A. Jugel, W.-B. Pöttner, U. Roedig, J. S. Silva, R. Silva, C. J. Sreenan, V. Vassiliou, T. Voigt, L. Wolf, and Z. Zinonos, "The GINSENG system for wireless monitoring and control: Design and deployment experiences," *ACM Transactions on Sensor Networks (TOSN)*, vol. 10, no. 1, pp. 4:1–4:40, Dec. 2013.

[2] A. Milenković, C. Otto, and E. Jovanov, "Wireless sensor networks for personal health monitoring: Issues and an implementation," *Computer communications*, vol. 29, no. 13, pp. 2521–2533, 2006.

[3] K. Langendoen, A. Baggio, and O. Visser, "Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture," in *Parallel and Distributed Processing Symposium, International*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, p. 155.

[4] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 381–396. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298491

[5] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong, "A macroscope in the redwoods," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '05. New York, NY, USA: ACM, 2005, pp. 51–63. [Online]. Available: http://doi.acm.org/10.1145/1098918.1098925

[6] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, "The hitchhiker's guide to successful wireless sensor network deployments," in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '08. New York, NY, USA: ACM, 2008, pp. 43–56. [Online]. Available: http://doi.acm.org/10.1145/1460412.1460418

[7] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki—a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. IEEE, 2004, pp. 455–462.

[8] M. Papadakis and K. Sagonas, "A PropEr integration of types and function specifications with property-based testing," in *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*. ACM, 2011, pp. 39–50.

[9] "Property-based testing for Erlang," 2011, http://proper.softlab.ntua.gr/.

[10] "libclang," http://clang.llvm.org/doxygen/group__CINDEX.html, Apr. 2014.

[11] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Information Processing in Sensor Networks. IPSN 2005. Fourth International Symposium on*. IEEE, 2005, pp. 364–369.

[12] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-MAC: A short preamble MAC protocol for duty-cycled wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 2006, pp. 307–320.

[13] A. Dunkels, "The ContikiMAC radio duty cycling protocol," Swedish Institute of Computer Science, Tech. Rep., 2011.

[14] M. Michel and B. Quoitin, "Technical report: ContikiMAC vs X-MAC performance analysis," 2014, arXiv preprint arXiv:1404.3589.

[15] P. Erdös and A. Rényi, "On random graphs i." *Publ. Math. Debrecen*, vol. 6, pp. 290–297, 1959.

[16] "Closed connection event issued twice #621," https://github.com/contiki-os/contiki/issues/621, Dec. 2014.

[17] "SocketAPI closed event is not triggered in some cases #867," https://github.com/contiki-os/contiki/issues/867, Dec. 2014.

[18] 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*. ACM, 2000, pp. 268–279.

[19] 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*. ACM, 2006, pp. 2–10.

[20] 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

[21] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 2010, pp. 186–196.

[22] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, ser. OSDI '08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[23] P. Li and J. Regehr, "T-check: bug finding for sensor networks," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 2010, pp. 174–185.

[24] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*. ACM, 2003, pp. 126–137.

[25] D. Bucur and M. Kwiatkowska, "On software verification for sensor nodes," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1693–1707, 2011.

[26] L. Mottola, T. Voigt, F. Österlind, J. Eriksson, L. Baresi, and C. Ghezzi, "Anquiro: Enabling efficient static verification of sensor network software," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*. ACM, 2010, pp. 32–37.

[27] K. Romer and J. Ma, "PDA: Passive distributed assertions for sensor networks," in *Information Processing in Sensor Networks, 2009. International Conference on*. IEEE, 2009, pp. 337–348.