Andreas Löscher

Targeted Property-Based Testing with Applications in Sensor Networks
Dissertation presented at Uppsala University to be publicly examined in Auditorium Minus, Gustavianum, Akademigatan 3, Uppsala, Thursday, 7 June 2018 at 09:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor John Hughes (Chalmers University of Technology).

Abstract

Testing is a fundamental part of modern software development, as it unveils bugs in the system under test and gives confidence in their correctness. Testing is often a laborious task as it typically requires to write by hand a plethora of test cases to test a system thoroughly. This task can be aided by high-level testing techniques such as random property-based testing (PBT) where the testing task is reduced to specifying properties that the system under test is expected to satisfy, and generators that produce well-distributed random inputs to these properties. However, as with all random testing techniques, the confidence in the system and the chances of finding a bug is proportional to the number of tests. If the set of possible inputs is large, even a high number of tests does not yield a satisfactory result. One example is testing sensor networks, where one not only needs to produce the inputs for the software system but also needs to consider the network topology and the systems environment.

This dissertation presents targeted property-based testing, an enhanced form of PBT where the input generation is guided by a search strategy instead of being random, thereby combining the strengths of QuickCheck-like and search-based testing techniques. It furthermore presents an automation for the simulated annealing search strategy that reduces the manual task of using targeted PBT. We present concrete implementations for all presented techniques and a framework for PBT of sensor networks.

Applying PBT to testing sensor networks has allowed us to test relatively complex software and uncover subtle and hard-to-find bugs. We evaluate targeted PBT by comparing it to its random counterpart on a series of case studies. We show that its testing performance is significantly higher than that of random PBT. Furthermore, we demonstrate that the extra effort required to use targeted PBT is limited to specifying a test goal. With these results, we argue that targeted PBT improves the state-of-the-art of software testing and ultimately leads to higher confidence in complex software systems.

Keywords: Software Testing, Search-Based Software Testing, Property-Based Testing

Andreas Löscher, Department of Information Technology, Division of Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Andreas Löscher 2018

ISSN 1104-2516
urn:nbn:se:uu:diva-347623 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-347623)
für Alfred
Testning är en viktig del av mjukvaruutveckling, eftersom det är så buggar (eller programfel) i mjukvaran upptäcks och på så sätt ger testning en högre tillit till systemens korrekthet. Under de senaste decennierna har små inbyggda trådlösa system som sensornätverk blivit populära. Att testa denna typ av system är betydligt mer utmanande än att testa traditionella datorprogram. Det har flera orsaker, till exempel att systemet kan bestå av flera hårdvaru- och mjukvaruarkitekturer, det kan vara flertalet (otillförlitliga) kommunikationskanaler och delsystemens storlek kan variera från små inbyggda system till molntjänster. Kraven vid testning av dessa system är inte annorlunda än kraven vid testning av vanlig mjukvara. Det kan till och med argu


Som med all slumpmässig testning, så är säkerheten för systemets korrekthet och chansen att hitta en bugg proportionell till antalet tester. Om antalet möjliga värden för indata är stort så ger kanske inte ens ett stort antal tester ett tillfredsställande resultat. Ett sätt att hantera en större mängd
möjliga värden för indata är att använda sökbaserte mjukvarutestningstekniker där genereringen av indata vanligtvis är styrd mot ett optimeringsmål, till exempel mot indata som maximerar eller minimerar någon systemparameter.


Acknowledgements

This dissertation would have not been possible without my colleagues, friends, and family that supported me during my time as a PhD student in Uppsala. I would like to thank all of you!

First, I thank Kostis Sagonas, my main advisor. Without you, I would not have been a Ph.D. student and moved my life to Sweden. Through our discussions, I could develop my ideas and you didn't shy away from pointing out weak spots, which I appreciate. My second advisor is Thiemo Voigt. Your positivity and encouragement helped me find my way at the beginning of my studies, and you always had an open ear for me.

I started my Ph.D. around the same time as Stavros Aronis, Kjell Windblad, Stephan Brandauer, David Klaftenegger, and Elias Castegren whom I also shared office with. I enjoyed our trips to the Fika room and our discussions about research and everything else. I am happy to be able to call you my colleagues and friends. I also thank Magnus Lång, Magnus Norgren, Kiko Fernandez, and Albert Mingkun Yang who joined us later.

My life during my Ph.D. was enriched by many people, by going climbing or skiing, going on road trips, exploring the local pub culture, or meeting up for stroller walks. I thank all of you.

I thank my parents Renate and Ulrich for supporting me throughout my whole life. You were the first one that taught me how to program and introduced me to computer science. My sister Anna, you are always there for me and a best friend.

Ida, thank you for fixing my bike that one time. I would have otherwise needed to walk home that day. Thank you for teaching me how to fish, in case of an apocalypse I now know how to survive. Thank you for teaching me how to snowboard, for showing me the north, and for all the times we traveled together. Thank you for being the love of my life and founding a family with me. Thank you for everything.

1 Swedish Coffee Break
Contents

Sammanfattning på svenska ................................................................. vii
Acknowledgements ........................................................................... ix
1 Introduction .................................................................................. 13
2 Background ................................................................................... 19
  2.1 Software Testing .......................................................................... 19
  2.2 Property-Based Testing .............................................................. 21
  2.3 Search-Based Testing ................................................................. 27
3 A Framework for Property-Based Testing of Sensor Networks .......... 31
  3.1 COOJA ...................................................................................... 32
  3.2 NIFTY ....................................................................................... 33
  3.3 Writing Properties for Sensor Networks ...................................... 39
4 Targeted Property-Based Testing .................................................. 43
  4.1 A Motivating Example ............................................................... 43
  4.2 Targeted PBT: Informal Presentation .......................................... 46
  4.3 Implementation .......................................................................... 48
  4.4 Simulated Annealing .................................................................... 52
  4.5 Targeted PBT: More Formal Presentation .................................... 55
5 Automating Targeted Property-Based Testing ............................... 57
  5.1 Using Constructed Neighborhood Functions .............................. 57
  5.2 Construction Algorithm ............................................................ 60
  5.2.1 Caching and Matching ......................................................... 62
6 Targeted Stateful Property-Based Testing ..................................... 71
  6.1 Constructed N for stateful testing ............................................... 71
  6.2 A Conservative Neighborhood Function ................................... 73
  6.3 An Opportunistic Neighborhood Function .................................. 74
7 Evaluation .................................................................................... 81
  7.1 Energy Efficiency of MAC Protocols ......................................... 81
  7.2 Contiki's Socket API ................................................................... 92
  7.3 Routing Tree for Directional Antennas ....................................... 96
  7.4 Noninterference .......................................................................... 99
  7.5 Small User Study ....................................................................... 103
1. Introduction

Testing is a fundamental pillar of modern software development. It is used to unveil bugs in programs and systems and gives confidence in their correctness. During development, testing can help to maintain the integrity of the software system, and prevent the repetition of previous errors when introducing new features. On the other hand, testing takes a considerable amount of effort during development. It is quite common that the test code exceeds the amount of code used for implementation of the software it is testing. In test-driven development, for example, the test code typically is two to five times the amount of the implementation code. An extreme example is SQLite whose the test code, at the time of writing this dissertation, exceeds the code of the library by a factor of 730 [25]. High-level testing techniques can reduce the amount of testing code while maintaining or even increasing the quality of the tests, by generating the test cases and input to them automatically, e.g., from the specification of the software, the code used for the implementation, or similar.

In the last decade, networked embedded wireless devices, such as wireless sensor networks, smart home appliances, and smart wearables, have become popular. Testing this type of systems is significantly more challenging than testing classical computer programs. The reasons are manifold, e.g., a system can be comprised of multiple hardware and software architectures, there might be various (unreliable) communication channels, and the scale of the subsystems may range from small embedded devices to cloud servers. However, the testing requirements for these systems are not different than those of usual software, and arguably it is even more crucial to rigorously test these systems as their commercial applications extend to critical domains such as process control [68] and e-health [63]. Also, testing and fixing software after the systems have been deployed is considerably more difficult and costly. Last but not least, a software bug triggered during operation might not only affect a single device but might affect the whole network.
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 [10, 47, 78, 80]. Part of the reason is that often, in practice, developers create only a few test scenarios to check their software. Creating extensive test suits is a time-consuming and tedious 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. Moreover, thorough testing requires a plethora of test cases that cover even unlikely corner cases.

In this dissertation, we advocate the use of property-based testing (PBT) to improve the state-of-the-art of testing sensor networks. PBT is a high-level, semi-automatic, black-box testing technique. Rather than writing a plethora of test cases by hand, one instead 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 [23, 81]. We developed a framework for PBT of sensor networks based on PROPER [72], an open source QuickCheck-inspired tool for Erlang, and present it in Chapter 3. Our framework is capable of testing a wide variety of properties for sensor networks ranging from network global properties, like the energy efficiency of communication protocols, to node local properties, like the correct behavior of a C-API of some library that is running on one of the nodes, and combinations of local and global properties.

Random PBT generates inputs to properties randomly and, 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 significant 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. During our experiments with random PBT of sensor networks, we experienced that, due to the simulation overhead and the size of the input space, it can take many hours to find a counterexample to some properties.
Search-based software testing techniques (cf. the survey article by Harman, Mansouri, and Zhang [40]) apply search techniques to the input generation processes and in doing so are able to handle even large input domains efficiently. The input generation is typically guided towards an optimization goal, e.g., towards inputs that maximize or minimize some quantity of the system.

To make PBT of complex applications such as sensor networks more effective, we developed targeted property-based testing (TPBT), an enhanced form of PBT, where the input generation process is guided by a search strategy, instead of being random. In Chapter 4, we present TPBT and a system that provides a concrete implementation of it, which is fully integrated into PROPER. Our implementation extends the high-level language of PROPER for specifying properties with supporting infrastructure that allows the user to employ some built-in search strategy or specify a new custom search strategy for input generation in a succinct and flexible way. In doing so, TPBT increases the probability that input that falsifies a property is generated, and ultimately the tester's confidence in the SUT.

As we will see, targeted PBT explores the input space more effectively and requires fewer tests to find a property violation or achieve high confidence in the SUT than random PBT. However, to use targeted PBT, a user has to specify: (1) a search strategy that will be used to explore the input space, (2) a targeted generator that is under the control of the search strategy, and (3) utility values (UV) for each input that the generation process will try to either maximize or minimize.

The search strategy is typically provided by the TPBT tool or a library and can be configured for the task at hand. To be able to guide the generation process, the user needs to manually provide some ingredients such as information on how to generate inputs and strategy-specific operators. For example Simulated Annealing (SA), the default search strategy of PROPER, requires a neighborhood function (Nf) that produces a “next” random input which is similar to a current one. Such a function is significantly harder to write than a generator for random PBT.

To make TPBT easier and less time consuming to use, we developed a technique for simulated annealing that constructs its main ingredient, namely
the NF, automatically from an input generator written for random PBT and present it in Chapter 5. By using this construction technique, we can reduce the effort to use TPBT significantly. In addition to the components needed for random PBT, a user effectively now only needs to extract the utility values and specify whether to maximize or minimize them.

As we will show on a series of examples, TPBT performs sufficiently well with the automatically constructed NFs for most applications. We also describe how random generators can be written so that the construction process can work to its full potential, and how the constructed NFs can be adjusted by the user manually.

The contributions of this dissertation are as follows:

1. We introduce the idea for PBT of sensor networks and present a framework in Chapter 3 to apply this testing method. We show how the framework provides support to specify a wide range of properties, starting from properties of individual functions to network-global properties, and infrastructure to automatically test these properties. We applied our framework to properties that test the correct behavior of C-APIs of libraries running on the sensor nodes, the correctness of distributed applications such as communication protocols, and global properties like the energy consumption of the network.

2. 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. In Chapter 4 we describe the conditions under which targeted property-based testing is applicable and the ingredients it requires. Additionally, we present an implementation of our technique for the PBT tool Proper.

3. We present a technique that automatically constructs the ingredients that TPBT needs for simulated annealing, namely the neighborhood function, from a random generator in Chapter 5. This constructed neighborhood function is capable of producing random neighboring input for all input instances that the random generator can produce and is competitive to a neighborhood function that one would write
by hand. By using the construction technique, we can reduce the
effort to use TPBT significantly.
4. We present targeted stateful testing, the extension of TPBT to testing
software systems that follow the transitions of some state machine
model, in Chapter 6. We show our two approaches, a conservative
one that maintains the existing interface of PROPER, and a more pow-
erful opportunistic one that changes this interface slightly while main-
taining most of its expressiveness.
5. We evaluate our techniques on a series of case studies (Chapter 7)
and demonstrate the ease of use and effectiveness of our frameworks
and techniques.

With these contributions this dissertation defends the following thesis:

**Thesis**

*Combining search-based input generation techniques with property-based
testing can significantly improve the testing performance for many
application domains and of sensor networks in particular.*
2. Background

2.1 Software Testing

Software testing is the process of finding discrepancies (i.e., bugs) between the requirements of a system under test (SUT) and their current implementation [42]. Testing is usually done by running a set of test cases and checking whether they pass or fail. A test case typically consists of a set of instructions that subject the SUT with fixed input and checks if the returned output is correct according to the specification. If the output is as expected, the test case passes, otherwise the test case fails and a bug is revealed. We call the input of the test case that triggers a bug witness or counterexample. Writing good test cases by hand is a laborious and expensive task. A tester has to consider all corner cases of the software she is testing and write test cases for all of them.

Testing is probably the most frequently used tool for developers to check if their code is behaving correctly. However, while testing is an excellent tool for finding bugs, it “is hopelessly inadequate for showing their absence” [27]. Nevertheless, in practice, software testing is used by most software companies to improve the quality of their developed software. The confidence in the correctness of the SUT increases with the quality of the test suite, which can only be increased by adding more and more (hopefully even better) tests to it. Doing this by hand, however, is a mundane and laborious task.

Testing is done on multiple levels of an application, starting from testing single functions independent from each other (unit testing), over testing multiple components together (integration testing), to testing the whole system (system testing). Each of those stages targets a different aspect for the total correctness of the SUT. During unit testing we check if the functional implementation of each software component is correct. During integration testing we test if the components we tested during unit-testing interact with
each other as expected, e.g., if the common protocols between these components are implemented correct. Finally, during system testing we test if the application or system as a whole is behaving according to its specification.

Depending on whether the internal structure or the implementation of the SUT is known or accessible during testing, we can divide testing techniques into black-box testing and white-box testing. In black-box testing, the internal structure and implementation of the SUT are ignored. We can only check if the outputs of the SUT are correct to a given input according to the software’s specification. It is not possible to, e.g., inspect the source code.

In contrast to black-box testing, in white-box testing the internal structure and implementation of the SUT are known and accessible. White-box testing techniques typically derive tests from an examination of the structure of the SUT, e.g., by inspecting or instrumenting the source code [43]. White-box testing techniques such as symbolic execution have become popular in the last four decades [9, 44]. These testing techniques systematically explore many execution paths of a program by instrumenting its implementation and executing it with symbolic input. Typically, a constraint or SMT solver is used to producing concrete input values for a specific program path if necessary. While symbolic execution techniques are very powerful, they rely on an accurate model for the language or architecture they are targeting and powerful constraint solvers. To test sensor networks, for example, a symbolic execution tool would not only need to consider the sensor node software and hardware, but also the node environment such as sensor input, and the network communication. Furthermore, symbolic execution techniques can suffer from path explosion, where the number of program paths is becoming too large to be covered in a reasonable amount of time. The interested reader is referred to the recent survey article by Baldoni et al. [9].
2.2 Property-Based Testing

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 is expected to satisfy, and generators that produce well-distributed random inputs to the parts of the system that are tested [23, 81]. 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. Following this method, a tester’s manual tasks are reduced to correctly specifying the parameters of the SUT and formulating a set of properties that accurately describe its intended behavior.

PBT tools operate on properties, which are essentially partial specifications of the SUT, meaning that they are more compact and 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 behaviors of a system and unveiling its bugs.

Because test inputs are generated randomly in PBT, the part of a failing test case that is causally 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
Algorithm 1: Overview of checking a single property

1 begin
2     repeat
3         Input ← generate valid instances for each input variable of
4             the property;
5         Result ← call the property with Input;
6         if Result == true then
7             continue; /* try next input */
8         else
9             shrink the input;
10            Report the failing test case and the shrunk input;
11             exit; /* we are done */
12         end
13     until max number of test reached;
14 end

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. Algorithm 1 shows a typical test run of a single property.

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 [72], 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 property can be specified as:

```erlang
prop_encode_decode() \rightarrow
\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:  

```plaintext
1  input() ->
2  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 shrinked 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 a more complex underlying structure (e.g., are balanced trees of some sort, have the format of an IP packet, etc.).

---

1In fact, in the tool we use, 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.
Similar to other PBT tools for Erlang (e.g., QuViq Quickcheck and Triq), PROPER also comes with support for testing stateful systems [8], i.e., system whose operations follow some (finite) state machine model, where states and transitions between them are associated with preconditions and postconditions. PROPER’s approach to stateful testing implements that of model-based testing (MBT) where the SUT is tested against an abstract model that the user needs to provide. The testing process still follows that of Algorithm 1 but now in Line 3 a test case in the form of a sequence of commands is generated. PROPER divides stateful testing into two phases. In the generation phase the test case is generated from the abstract model. The state that is preserved in between the command calls and the return values of the calls are symbolic in this phase. During the execution phase of the test case, PROPER checks if the behavior of the real system is correct with respect to the model by executing each command that was generated in the generation phase. Figure 2.1 shows the two phases of PROPER’s stateful testing.

The abstract model is described in the form of a generator for the commands that can be executed along with preconditions and postconditions for each of these commands. We will now show how to create a (simple) model of a car:
The `command(S)` generator specifies the interactions with the system which are executed by calling a function. The interactions are represented as tuples, where the first element is always the atom `call`, the second element is the Erlang module containing this code, followed by the function name, and finally the arguments of the function as a list of `PROPER` generators. These generators for the arguments are later used to generate random input to the functions. The `command(S)` generator is dependent on the state `S` of the test execution. In our model, besides other information, the state stores if the car has been started. The `command(S)` generator only produces commands that make sense in the current state, e.g., it is not possible to accelerate or break if the car is switched off.

The model of our car only contains the following interactions: we can start, stop, accelerate, break, or just drive the car. A car is, of course, a more complex system, but MBT and in extension stateful testing with `PROPER` allows us to test properties of the system that we can observe on a simpler model.

Stateful testing makes it possible to keep some information in the form of a state between the individual commands. This bookkeeping of the state is done by a call to a function `next_state(S, Res, Call)`. Note, that this function is called twice: (1) when the sequence of commands is generated and (2) when the commands are executed. The state `S` and the result `Res` of the call can, therefore, be both symbolic and concrete. The initial state is specified by the function `initial_state()`. We specify that the car is off, the driven distance is 0.0, and that the available fuel is 5.0:

```erlang
command(S) ->
  command(S) ->
    case S of
      {_, _, off} ->
        {call, ?MODULE, start, [{}];
      _ ->
        oneof([{call, ?MODULE, stop, []},
               {call, ?MODULE, accelerate, [acceleration_force(S)]},
               {call, ?MODULE, break, [break_force(S)]},
               {call, ?MODULE, drive_along, [td()]
        ]})
    end.
```
initial_state() ->
{0.0, 5.0, off}.

Additionally, we can specify preconditions and postconditions for each command. If a command does not meet its precondition, then it is not scheduled for execution and the next command is selected instead. If the postcondition is violated, however, we have found a discrepancy between the SUT and the model and therefore a bug. The precondition is checked twice, one time in the generation phase and one time in the execution phase. The postcondition is checked after the execution of a command.

Besides the information if the car is on or off, we store the driven distance and the leftover fuel in the state of our system and formulate a postcondition that states that the fuel and the driven distance can never be negative:

postcondition({D, F, _OnOff}, {call, _, _, _, _}) ->
F >= 0.0 andalso D >= 0.0.

Let us assume that we complete our car model and want to check that after the fuel is consumed, we drove at least a distance of 100km. A stateful property that checks this behavior for our car model looks as follows:

prop_fuel_consumption() ->
?FORALL(Commands, proper_statem:commands(?MODULE),
begin
{_History, {D, F, _}, Result} = proper_statem:run_commands(?MODULE, Cmds),
cleanup(),
GoodConsumption = F > 0.0 orelse D > 100.0,
Result =:= ok andalso GoodConsumption
end)

In Line 2 we use PROPER’s proper_statem:commands() generator for stateful testing to generate a sequence of commands. The argument to this generator is a module that exports the command() generator, the next_state() function, the precondition(), and the postcondition(). In Line 4 the commands get executed. PROPER returns a three-tuple with the history of the internal state, the final state, and the result from the command execution. The result indicates if the execution of the commands was successful or if an exception occurred during the execution of one of the scheduled com-
mands, e.g., if a postcondition has been violated. Typically some form of cleanup is performed afterwards (Line 5). In Lines 6 and 7 we check that the command execution was successful and that our fuel consumption was as specified. The property can now be tested in the same way as all other properties:

Eshell V6.3 (abort with ^G)
1> proper:quickcheck(car:prop_fuel_consumption(), 1000).

................... 1000 dots ....................
OK: Passed 1000 test(s).
true

The property holds for all 1000 randomly generated command sequences.

2.3 Search-Based Testing

Search-based software testing (SBST) is a subfield of search-based software engineering, in which search techniques are used to automate or aid the testing task, such as test data generation, test case minimization, or test selection. The concepts of SBST have first been described by Miller and Spooner in 1976 [64]. Since then, SBST has been applied to many areas of software testing, and a short overview can be found in the related work chapter (Chapter 8).

SBST is often applied in areas where random search has a low probability of succeeding, e.g., if the input domain is large and only a small percentage of the input satisfies the test goal. To better find inputs that are close to the test goal a meta-heuristic is employed to guide the input generation process. The meta-heuristic or search strategy requires some guidance usually given as fitness values or utility values of the input data. The search strategy then tries to find input values that are near-optimal, i.e., maximal (or minimal, depending on the problem). The search strategies that are used are commonly non-deterministic and not problem-specific, and range from simple local search procedures to complex learning processes [12].

We illustrate the concepts of SBST on an example where we want to find input to a test case so that the test case fails. The input is a $x$ and a $y$ value
Figure 2.2. Comparison of Random Testing vs. SBST with Simulated Annealing; The red area is the part of the input space that triggers a bug and the grey dots the generated input of both methods. The right plot additionally shows the contour lines of the fitness function and the direction of the search starting from $x = 0, y = 0$.

between 0.0 and 10.0. Furthermore, only a small size of this input space triggers a bug that we want to find. Figure 2.2 illustrates this problem. The left plot shows 2000 randomly generated points on the input space and the area (red) of the input space that would trigger a bug. We can see that none of these 2000 randomly generated inputs is inside the area that reveals a bug. However, if we have a fitness function to the input space, so that the fitness of inputs that reveal the bug is high, we can apply SBST. The right plot in Fig. 2.2 shows how SBST with simulated annealing as search heuristic would generate inputs when starting the search from $x = 0.0, y = 0.0$. The arrow shows the path the search takes towards the maximum of the fitness function.

Random generation can in theory find the failing input much faster, the chances of that happening are however slim. Search-based input generation requires on average fewer tests to finds the desired input compared to random generation.

To use search techniques for testing, we need to have a representation of our problem that is compatible with the search strategy we use. Harman argues that this requirement is easily satisfied because typically a software engineer will have a suitable representation in for their problem [38]. Fur-
thermore, in the case of test data generation, this requirement is automatically fulfilled the search can be applied to the input of our test cases more or less directly [61]. Another requirement is the existence of fitness or utility value that guides the search. Harman and Clark argue that many software engineers naturally work with metrics and metrics act as projections of qualities of interest of software systems [39]. Therefore, these metrics make excellent candidates for fitness values.
3. A Framework for Property-Based Testing of Sensor Networks

This chapter is based on:

Andreas Löscher, Konstantinos Sagonas, and Thiemo Voigt.
**Property-Based Testing of Sensor Networks.**
*12th Annual IEEE International Conference on Sensing, Communication, and Networking* [56].

Andreas Löscher and Konstantinos Sagonas.
**The Nifty Way to Call Hell from Heaven.**
*Proceedings of the 15th International Workshop on Erlang* [55].

In this chapter, we present a framework based on PROPER for PBT of sensor networks. The framework uses 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 and infrastructure to automatically test these properties in COOJA, the network simulator of the CONTIKI operating system [31].

We show the components of our framework and how they interact in Fig. 3.1. PROPER, our PBT tool, cannot directly interact with COOJA. We, therefore, implemented a control layer to perform this task. Furthermore, we present NIFTY [55], a component of our framework, which we developed from scratch, that allows us to test the sensor node software at the level of individual C functions. 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. We already described PROPER in Section 2.2 and present the two other main components of our framework, COOJA and NIFTY, in the following sections.
3.1 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.

As mentioned, PROPER cannot directly control the simulation. Therefore, we created 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, it is possible to 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. Furthermore, we provide a parser for IPv6 packets, making their analysis easier.

It is possible to add and remove sensor nodes from the simulation at any time. Deleting a node can, e.g., be used to simulate node failure. Furthermore, the position of the nodes can be altered, which can be used to simulate mobile nodes. We can 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. 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.

3.2 NIFTY

To adequately 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 pro-
tocol 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. Therefore, NIFTY generates an interface that operates a function call protocol over the serial line.

NIFTY generates interfaces in three steps:

- In a preprocessing step, NIFTY reads the given C header file and collects information about the defined types and functions. This information is stored in a type table and a symbol table.
- The second step uses the type and symbol table to generate the files for the interface.
- In the last step, the generated files are compiled together with the rest of the sensor node software.

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. Figure 3.2 illustrates the interface generation process.

Let us assume that we want to create an interface to the API shown in Listing 3.1, which shows a part of the function definitions from CONTIKI’s socket API. The interface uses a typedef, pointers, and integer types.

### Preprocessing

NIFTY uses LibClang [29] to parse the C header files. LibClang is a high-level C interface to the Clang compiler [28]. It invokes a C preprocessor resolving macros and generates an abstract syntax tree (AST). The library provides an interface to iterate over the tree and collect the information about the definitions.

NIFTY iterates over the AST and collects relevant information about all defined functions and types. During this process, we can decide if we want to descend deeper into the tree or if we want to continue with the next node. By descending, we can collect more information about the current node, e.g., in case of a node describing a function we can gather information about the function’s arguments.

NIFTY interfaces with LibClang using a NIF module and processes the C header into a symbol table, a list of the used types, and a constructor table. The symbol table stores for each function definition the types of the argu-

---

**Listing 3.1.** Part of CONTIKI’s socket API from socket_interface.h.
Table 3.1. Symbol table for socket_interface.h.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;create_socket&quot;</td>
<td>[{return,&quot;int&quot;}]</td>
</tr>
<tr>
<td></td>
<td>[{return,&quot;int&quot;},</td>
</tr>
<tr>
<td></td>
<td>{argument,&quot;0&quot;,&quot;int&quot;},</td>
</tr>
<tr>
<td></td>
<td>{argument,&quot;1&quot;,&quot;uint16_t&quot;}]</td>
</tr>
<tr>
<td>&quot;listen&quot;</td>
<td>[{return,&quot;int&quot;},</td>
</tr>
<tr>
<td></td>
<td>{argument,&quot;0&quot;,&quot;int&quot;},</td>
</tr>
<tr>
<td></td>
<td>{argument,&quot;1&quot;,&quot;uint16_t&quot;}]</td>
</tr>
<tr>
<td>&quot;connect&quot;</td>
<td>[{return,&quot;int&quot;},</td>
</tr>
<tr>
<td></td>
<td>{argument,&quot;0&quot;,&quot;int&quot;},</td>
</tr>
<tr>
<td></td>
<td>{argument,&quot;1&quot;,&quot;uip_ip6addr_t *&quot;},</td>
</tr>
<tr>
<td></td>
<td>{argument,&quot;2&quot;,&quot;uint16_t&quot;}]</td>
</tr>
<tr>
<td>&quot;build_ip_addr&quot;</td>
<td>[{return,&quot;uip_ip6addr_t *&quot;},</td>
</tr>
<tr>
<td></td>
<td>{argument,&quot;0&quot;,&quot;uint16_t&quot;},</td>
</tr>
<tr>
<td></td>
<td>{argument,&quot;1&quot;,&quot;uint16_t&quot;},</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>{argument,&quot;7&quot;,&quot;uint16_t&quot;}]</td>
</tr>
</tbody>
</table>

Table 3.2. Constructor table for socket_interface.h.

<table>
<thead>
<tr>
<th>Type</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>{typedef,&quot;uint8_t&quot;}</td>
<td>&quot;unsigned char&quot;</td>
</tr>
<tr>
<td>{typedef,&quot;uint16_t&quot;}</td>
<td>&quot;unsigned short&quot;</td>
</tr>
<tr>
<td>{typedef,&quot;uip_ip6addr_t&quot;}</td>
<td>&quot;union uip_ip6addr_t&quot;</td>
</tr>
</tbody>
</table>

The constructor table stores additional information for the supported user defined types. For structs it is the type information of the fields, for enums the association between names and values, and for typedefs the underlying type. Table 3.2 shows the constructor table for socket_interface.h.

The next step of the preprocessing phase is the construction of a type table that contains detailed information about all used types. Table 3.3 shows the type table built from parsing socket_interface.h as shown in Listing 3.1.
<table>
<thead>
<tr>
<th>Type</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;int&quot;</td>
<td>{base,[&quot;int&quot;,&quot;signed&quot;,&quot;none&quot;]},</td>
</tr>
<tr>
<td>&quot;unsigned short&quot;</td>
<td>{base,[&quot;int&quot;,&quot;unsigned&quot;,&quot;short&quot;]},</td>
</tr>
<tr>
<td>&quot;unsigned char&quot;</td>
<td>{base,[&quot;char&quot;,&quot;unsigned&quot;,&quot;none&quot;]},</td>
</tr>
<tr>
<td>&quot;uint8_t&quot;</td>
<td>{typedef, &quot;unsigned char&quot;},</td>
</tr>
<tr>
<td>&quot;uint16_t&quot;</td>
<td>{typedef, &quot;unsigned short&quot;},</td>
</tr>
</tbody>
</table>

Table 3.3. Type table for socket_interface.h.

The type table associates a type name with a type definition. The type definition can be either a base type definition, a user-defined type, a type alias, or a struct definition.

A base type definition contains the arithmetic type specifier and the optional specifiers. Arithmetic type specifiers are char, int, float, and double. For char and int types the optional type specifiers signed, unsigned, short, and long are used. This information is necessary since it determines how the base types are represented and how we need to translate them from Erlang to C. For float and double those fields are ignored. For char the size specifier is ignored. A pointer to a base type is also considered a base type.

Type aliases store the name of the type they are an alias of. A user-defined type is a type that is defined in the header file and not a base type. Pointers to user-defined types and type aliases are also user-defined types. struct types are user-defined types. Their type entries contain a reference to a struct definition which is stored in the constructor table.

After the initial generation of the type table, the table may contain incomplete or invalid types. These can be, e.g., type aliases that do not end with a basic or user-defined type, or unsupported types such as union types. A type checker checks the validity of all types in the type table and removes invalid types. After the type table has been checked, all functions that rely on invalid types are removed from the symbol table. Note that there is no type entry for the type union uip_ip6addr_t, the associated typedef, or the pointer type uip_ipaddr_t* in the type table. NIFTY does not support unions as of right now and cannot construct them directly. These types are therefore filtered out from the type table. However, NIFTY can always handle any pointer type without knowing what pointer is pointing to. It is therefore
possible to construct `uip_ipaddr_t*` with `build_ip_addr()` and use it within the socket API.

**Generation**

**NIFTY** uses the ErlyDTL [24] template engine to generate the code for the call interface. ErlyDTL creates BEAM code from templates written in the Django [33] Template Language (DTL). Using a template engine allows us to describe static and dynamic parts of the render output in one domain specific language.

The DTL is a text-based template language. A template is a text file that contains static text, variables, and tags. If the template engine encounters a variable during rendering, it is evaluated and exchanged by the value. Tags are used to perform complex operations in the template like producing text output, loading additional information, or as control structures like loops. In Fig. 3.3 we show examples of how the template engine renders variables and loops. We can modify the value of the variable from inside the template using filters. Filters can also have one argument:

```
{{variable|filter:argument}}
```

ErlyDTL provides the possibility to extend the tags and filters of the template language by custom ones. **NIFTY** uses custom filters and tags that support basic operations on dictionaries, type tables, and types. The generated files form an application for **CONTIKI** that if included in the software for a sensor node operates its 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 and does not correspond to the name of the function in any way. 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 a 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.

Using templates for the generation process allows us to change the target architecture by merely exchanging the set of templates that NIFTY uses. For example, we developed a separate version of NIFTY [55] that, instead of producing call interfaces for CONTIKI, creates a call interface in the form of an Erlang NIF and can be used to interface between Erlang code and C-libraries.

3.3 Writing Properties for Sensor Networks

In this section, we describe how properties of sensor networks are written using our framework. Figure 3.4 shows an overview of the testing process using the framework. Besides the generator and the property, an initial simulation file for COOJA is required. This simulation file can contain an already setup network with different nodes and connections between them. More importantly, however, is that the simulation file includes the control plugin that enables us to control the simulation and to read data from the network. Additionally, the property needs to start up the simulation, con-
Figure 3.4. Overview of the testing process

Control its progress and quit it after the network has been simulated. The structure of a property using our framework is shown in Listing 3.2.

A sensor network is defined by its sensor nodes, their firmware (the software of the node), and its network topology. The topology defines how the sensor nodes are connected to each other. The behavior of the node software, especially of the network components, depends on the network topology. As mentioned, we can define the topology in the simulation file simulator loads when it is started. The sensor nodes and the network topology are then automatically loaded by the simulator. A much more intriguing 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. To retrieve data from the simulation we need to subscribe to the corresponding data sources. These
prop_SensorNetwork() ->

`FORALL(I, input(),

  begin
    %% start simulation
    nifty_cooja:start("< PathToCooja >",
    "< PathToSimulationFile >",
    [])). % Options, e.g., gui mode
    %% check simulation state and get simulation handler
    {running, Handler} = nifty_cooja:state(),

    %%
    %% actual property
    %%

    %% exit the simulation
    ok = nifty_cooja:exit()
    end).

Listing 3.2. The general structure of a property that tests sensor networks with our framework.

data sources are the global radio medium, the hardware events of a sensor node, and the serial line of a sensor node.

Our framework provides more ways to control or manipulate the simulation, then we have mentioned so far. It is for example possible, to change the speed individual nodes are progressed to simulate time drift, to directly read the memory of any global variable defined in C, or to change the properties of the communication environment by, e.g., increasing the noise level, etc.

In Chapter 7 we present two case studies that demonstrate our framework on concrete examples. The first case study (Section 7.1) shows how to test a global property such as the energy consumption by generating random sensor networks. The second case study demonstrates how the framework can be used to test the C-API of libraries or operating system components on the sensor node. We do this by testing the C-API of CONTIKI’s socket API for which we created a call interface using NIFTY.
This chapter is based on:

Andreas Löscher and Konstantinos Sagonas.

**Targeted Property-Based Testing.**
Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis [54].

This chapter 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. Additionally, we present an implementation of our technique as an extension to PROPER.

### 4.1 A Motivating 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 number of hops messages need to take to reach their destination. In our example let us suppose that the majority of the messages are going to one dedicated node, the sink. This type of situation often occurs in sensor networks where leaf nodes collect data and send them to a sink for further processing [4].

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:

```prolog
prop_length(N) ->
  ?FORALL(G, graph(N), lists:max(distance_from_sink(G)) < (N div 2)).
```

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:

```prolog
1> proper:quickcheck(example:prop_length(42), 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 find-
ing a failing input. Such a generator, however, is complicated to write. In
other properties, the relation between the network topology and the ob-
served 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 net-
work.

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. However, it is possible to
use search strategies to generate inputs that have a higher probability of
falsifying 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.
4.2 Targeted PBT: Informal Presentation

Targeted property-based testing (TPBT) is a variation of property-based testing that aims to make test outcomes more consistent and reduce the number 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 in PROPER, 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 targeted PBT can be applied to. Still, its application domain is quite large. As we will also see in Chapter 7, tests where timing, resource consumption, or performance properties of a SUT are checked against a threshold are ideal candidates for TPBT. The TPBT component of PROPER provides macros 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.

TPBT 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.

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

The general structure of properties that can be tested with TPBT, called targeted properties, looks as follows:
prop_Target() -> % Try to check a property
  ?FORALL_TARGETED(Input, Generator, % by using a search strategy
    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 \texttt{SUT:run()} function needs to return the utility value. This UV is then fed to the search strategy component which uses this information to produce the next input with an increased UV, thereby also increasing the chance of falsifying the property.

The implementation of the search strategy is mostly independent of the property that is tested. The user of TPBT 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 typically added to the generator with macros additional macros. Simulated Annealing (SA) for example, the default strategy provided by PROPER for TPBT, requires the user to specify a generator for the first input and a \textit{neighborhood function} (\texttt{Nf}); see Section 4.4. 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.

Accompanying the macros for writing targeted properties mentioned above, we also introduce two additional macros: \texttt{?EXISTS} and \texttt{?NOT_EXISTS}. Both macros specify a targeted property. \texttt{?NOT_EXISTS(I, targeted_gen(), check(I))} defines a property that says that no \texttt{I} exists so that \texttt{check(I)} is true; vice versa \texttt{?EXISTS(I, targeted_gen(), check(I))} specifies that such a value must exist. If a value that satisfies the property for \texttt{?EXISTS} is found, then the property holds, and at that point, no new test is generated. If the property fails after the maximum number of tests, \texttt{?EXISTS} does not produce a counterexample. The \texttt{?NOT_EXISTS} macro behaves like a \texttt{?FORALL} with the difference that the property condition is negated.
4.3 Implementation

In this section, we describe the implementation of TPBT in Proper on the example of the Hill Climbing (HC) strategy that we want to use to guide the generation of graph inputs as used in our prop_length example. HC is a relatively simple search strategy. It starts with a random initial and 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 Proper and the HC strategy looks like follows:

```prolog
prop_length_targeted() ->
\(\forall X, \forall NF\) (graph(42), fun graph_next/1),
begin
UV = lists:max(distance_from_sink(G)),
\(\maximizes\) (UV),
UV < 21
end).
```

The ?forall_targeted macro informs Proper to use a search strategy to guide the input generation. 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. We use the ?usernf macro to add the function graph_next(G) (see Listing 4.1) to the generator which produces neighboring graphs. Finally ?maximize tells HC which variable should be maximized. When we test the property, we have to tell Proper which search strategy we want to use. For the Hill Climbing strategy we can test the property with the following call:

```prolog
> proper:quickcheck(prop_length_targeted, [{search_strategy, hill_climbing},
{search_steps, 10000}]).
```

48
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(_, E, N) ->
  ?LET(NewEdges, proper_types:vector(N, edge(V)), {V, lists:usort(E ++ NewEdges)}).

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

Listing 4.1. Implementation of the neighborhood function for the HC strategy.

Usually, PROPER controls how the input is produced from the generators. With TPTB, we want to hand control over the input generation to a search strategy. This is achieved by using the \texttt{FORALL\_TARGETED} macro. We call a generator that is controlled by a search strategy a \emph{targeted generator}. PROPER extracts the information needed by the search strategy and produces such a generator with \texttt{targeted()}:

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.
For the Hill Climbing strategy, the parameters that get extracted are the random generator itself and the added function for generating neighboring input. We use the \texttt{LAZY()} construct of PROPER. \texttt{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 \texttt{targeted_gen()} with the same arguments. The first argument to \texttt{targeted_gen()} is a unique reference$^1$ that is used as key to the generator. Note, that the definition of the macro is independent of the used search strategy. How the input is generated and which parameters are accepted are defined by the used search strategy.

The \texttt{targeted_gen()} function calls \texttt{get_target()} to get the target-triple consisting of a \textit{target state}, a \textit{next function} and a \textit{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 \texttt{update_target()} is called to store this new state. When \texttt{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 \texttt{UV} has been extracted from the SUT the \texttt{MAXIMIZE(UV)} macro can be used to tell PROPER which value should be increased in the next input:

\begin{verbatim}
1  -define(MAXIMIZE(UV), update_target_uvs(UV)).
2
3  update_target_uvs(UV) ->
4      [update_target_uv(Key, UV) || Key <- get_target_keys()].
5
6  update_target_uv(Key, UV) ->
7      {State, _NextFunc, UpdateFunc} = get_target(Key, []),
8      NewState = UpdateFunc(State, UV),
9      update_target(Key, NewState).
\end{verbatim}

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

$^1$More information at http://erlang.org/doc/man/erlang.html#make_ref-0.
A search strategy is mainly defined by how the next function and the state-update function are implemented as well as the initial state. For the strategy `hill_climbing` these definitions are shown in Listing 4.2. 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 the targeted generator with `graph_hc`, which provides all ingredients needed for 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(N)` generator we used in the property `prop_length()` in Section 2.2. 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 \texttt{Next}
as used in \texttt{prop.length.targeted} is implemented with \texttt{graph.next(G)} as seen
in Listing 4.1. To produce neighboring graphs, the function 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 \texttt{prop.length.targeted} now it fails after an average of
17,666 tests (measured over 1,000 runs with each time running a maxi-
umum of 100,000 tests). More importantly, a counterexample is found in
all runs. Hill Climbing as presented here has a series of shortcomings. Find-
ing a graph as for \texttt{prop.length.targeted} is a convex problem (local optima
are also global optima). Hill Climbing is a local optimization strategy that
performs very well for convex problems. In practice however, we need
strategies that allow us to escape local optima. Therefore, we propose the
use of more powerful strategies such as the one we present next.

### 4.4 Simulated Annealing

Simulated Annealing (SA) is a well-studied local search meta-heuristic [66]
that can be used to address discrete and continuous optimization problems.
The key feature of SA is a mechanism that enables escaping local optima
by accepting search steps to worse solutions in the hope to find a global
optimum. SA also has another 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 \(\Omega\) (the set of all possible solutions) and
an objective function \(f : \Omega \rightarrow \mathbb{R}\). In our framework, \(\Omega\) is equivalent to
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 \(N : \Omega \rightarrow \Omega\) 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 the 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:

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

When implementing SA as a 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 add them to the generator using the \texttt{USERNF} macro. We could, e.g., define a generator for integers that has the neighborhood function information as follows:

```prolog
integer(Low, High) ->
  \texttt{USERNF}(proper.types:integer(Low, High),
    integer.next(Low, High)).

integer.next(Low, High) ->
  fun (OldInstance, Temperature) ->
    Offset = \texttt{trunc}(abs(Low - High) * Temperature * 0.1) + 1,
    \texttt{LET}(X, proper.types:integer(-Offset, Offset),
      ensure.range(X + OldInstance, Low, High))
  end.
```

The first element is implemented by using PROPER’s default integer generator. The \texttt{integer.next(Low, High)} function 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. The \texttt{ensure.range()} function bounds the newly generated value to the allowed interval $[\text{Low}, \text{High}]$. 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 tempera-
ture. This scaling makes it possible to find the local optima point faster in low-temperature conditions.

Similarly, we can use the \texttt{graph\( (N) \)} and \texttt{graph\_next\( (G) \)} generators to specify an SA-capable generator for graphs:

\begin{verbatim}
1 graph\_sa\( (N) \) ->
2 \{first = graph\( (N) \), next = fun \( (Base, _T) \) -> graph\_next\( (Base) \) end\}.
\end{verbatim}

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). \texttt{PROPER} provides 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. However, an issue with this approach is that the temperature is high for a long time in the beginning. This means that poor 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 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 \cite{11}. \texttt{PROPER} 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.

The implementation of TPBT in \texttt{PROPER} 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 TPBT with \texttt{PROPER} has to provide these generators. However, implementing a neighborhood function for more complex input is significantly harder than writing a random generator for the same input. Therefore, we present in Chapter 5 an automation that automatically constructs such a neighborhood function.
from a PROPER generator and reduces the task to basically specifying the utility values.

4.5 Targeted PBT: 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.,

$$ p : I \rightarrow \{ \text{true}, \text{false} \} $$

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

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

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 targeted generator as:

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

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 TPBT, the property has to be expressed as follows:

$$ \forall i \in I, f(i) < t $$
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 $T$ 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, the target function $T_{sa}$ is implemented by the neighborhood function $N$. The input for the neighborhood function is the last solution that was accepted by the simulated annealing algorithm:

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

Similarly, the hill climbing strategy $T_{hc}$ can be defined as follows:

$$T_{hc}(i_n, u_n) = \begin{cases} 
\text{generate_sample(First(N))} & \text{if } n = 0 \\
k : u_k^n = \max_{j=0}^n u_j^n \rightarrow R(i_k^n) & \text{otherwise}
\end{cases}$$

$$R(x) = \text{Next}(x)$$
5. Automating Targeted Property-Based Testing

This chapter is based on:

Andreas Löscher and Konstantinos Sagonas.  
**Towards Automated Targeted Property-Based Testing.**  
11th IEEE Conference on Software Testing, Verification and Validation [53].

Targeted property-based testing as we have presented it in Chapter 4 requires the user to supply the ingredients that the search strategy needs to guide the input generation. For the default strategy of PROPER, Simulated Annealing (SA), the user not only needs to specify fitting utility values but also a neighborhood function (NF). A NF generates a value in the neighborhood to a given base value, i.e., a value that is similar. Such a neighborhood function has to be provided for all input generators that we want to use with TPBT. Furthermore, such a function is significantly harder to write than a random generator. In Chapter 4 we introduced a graph(N) generator that generated random graphs of size N. The generator itself is written in only 5 lines, but the neighborhood function that we used required 29 lines of code to implement (see Listing 4.1). In this chapter, we present a technique that constructs a neighborhood function automatically from a random generator. In the next section we describe how such a constructed NF can be used by the user and in Section 5.2 we will present the algorithm itself.

5.1 Using Constructed Neighborhood Functions

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

In our tool, the simplest way to specify that we want to transform a random generator into a NF is to just give the generator using one of the targeted property specification macros (?FORALL_TARGETED, ?EXISTS, or ?NOT_EXISTS):

```
prop_length() ->
  ?FORALL_TARGETED(G, graph(42),
      begin
        UV = lists:max(distance_from_sink(G)),
        ?MAXIMIZE(UV),
        UV < 21
      end).
```

Besides specifying the utility value for TPBT we only need to specify a random generator. PROPER takes the generator definition and constructs a graph_nf(G, T) function that produces neighboring graphs according to the specifications in graph(N). The random generator that was used as input for the construction is utilized by SA to obtain an initial input.

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

The effort needed to use TPBT in this example is reduced significantly. Instead of writing 29 (see Listing 4.1) lines of complicated code for the neighborhood function, a user only needs to specify which generator to use. We mentioned earlier that it would be possible to also find a counterexample with random PBT if we were to write a more complicated random generator. Using TPBT and an automatically constructed \( \mathcal{N}_F \) allows us to use a simple and easy to understand generator instead.

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

In Listing 4.1, the code of \texttt{graph\_next()} uses the function \texttt{neighboring\_integer()} to produce non-negative integers neighboring a given base integer. We can translate the built-in generator \texttt{non\_neg\_integer()} of PROPER and use it instead of the hand-written function as follows:

```plaintext
graph\_next(G, T) ->
  Size = graph\_size(G),
  \( \mathcal{N}_F \)Int = proper\_sa:get\_neighborhood\_function(non\_neg\_integer()),
  ?LET(NewSize, \( \mathcal{N}_F \)Int(Size, T),
    ?LET(Additional, \( \mathcal{N}_F \)Int(Size div 10, T),
      begin
        ...
      end).
```

As a positive side effect, we can now scale the size of the neighborhood for the integers with the temperature in the \( \mathcal{N}_F \).

Alternatively, it is also possible to adjust the construction process by overwriting the construction rules for some of the inner generators (e.g., the
element generator of a list) with a user-defined one. This is done by annotating a random generator with a hand-written \( \text{NF} \). If the constructor needs to build a \( \text{NF} \) for the annotated generator, the manually written one is used instead, which allows us to change the behavior for parts of the resulting \( \text{NF} \). Let us assume we want to write a \( \text{NF} \) for lists of integers where the elements should only change by \( \pm1 \) during each search step. We can easily write a \( \text{NF} \) for integers that behaves in that way:

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

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

1. integer_list() ->
2. list(?USERNF(integer(), fun integer_pm_one/2)).
3. prop_sth() ->
4. ?FORALL_TARGETED(L, integer_list(), ...).

The property \( \text{prop_sth}() \) passes the annotated random generator integer_list() to the constructor to build the \( \text{NF} \). The resulting integer_list_nf(), which is constructed in memory, will produce neighboring integer lists with the property that if list elements are modified, they are passed to the manually written integer_pm_one() that offsets them by \( \pm1 \). The ability to adjust the construction to the requirements of the input type provides the user with additional control and most of the time makes it unnecessary to write the entire neighborhood function by hand.

5.2 Construction Algorithm

In this section, we present the algorithm for constructing a \( \text{NF} \) from a \texttt{PROCER} generator for input of the same type.
Algorithm 2: Top-level NF construction algorithm.

1. `construct_nf(g):`
2. `begin`
3. `type ← get_type(g);`
4. `nf_{raw} ← n(b, t) for type according to Table 5.1;`
5. `nf ← apply_constraints(nf_{raw}, g);`
6. `return nf;`
7. `end`

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

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

If a custom generator has constraints (as defined by `?SUCHTHAT`) then these are checked after a neighboring candidate has been produced. If all constraints are fulfilled, then the candidate is returned as new input. Otherwise, the constructed NF is used to produce another neighboring input from the base value. If a valid neighboring solution cannot be produced within a certain number of attempts, a random element is generated from the originating random generator instead. The top-level construction algorithm is given in Algorithm 2.

For most built-in generators and combinations thereof, a NF can be constructed statically for the whole generator at once. This is often not possible for more complicated user-defined generators (e.g., generators that use `?LET`) since values that are instantiated early in the generation process might influence the structure of the generators dynamically.
In Table 5.1 (see page 67), we list the translation rules for most of PROPER's built-in “basic” generators and some additional rules. The constructed NF is written as \( n(b, t) \) where \( b \) is the base input and \( t \) the current temperature. We define a NF that is obtained from a generator \( g \) as \( \text{construct}_n(g) \). We use the notation \( v \sim U(\text{Set}) \) to uniformly sample a value \( v \) from \( \text{Set} \) and \( x \sim g \) to sample a value \( x \) from generator \( g \).

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

The so constructed NF scales the size of the neighborhood from which the next element gets selected by the temperature. This means that under low temperature the neighborhood is smaller than under high temperature. The idea is that at the beginning of the testing it is easier to explore larger parts of the input space if the neighbors are further apart from each other. In contrast, towards the end of the testing, smaller steps can help narrowing down good input more efficiently [82].

5.2.1 Caching and Matching

User-defined generators that use a \(?\text{LET}(x, g_{in}, f_{in})\) construct are comprised of an inner generator \( g_{in} \) which produces a random value for \( x \), and a function \( f_{in} \) that constructs the value generated by the entire \(?\text{LET}.\) Following the idea of the construction algorithm, we construct a NF so that \( g_{in} \) produces a value in the neighborhood of the previous inner value and then applies \( f_{in} \) afterward. The last entry of Table 5.1 shows the general NF our construction algorithm uses for \(?\text{LET}\) generators.

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

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

We approach the second issue by structurally matching the prior value to the return value of the $f_{in}$ function: an immediate value is not matched and used directly as new value of the generator; if the return value is only a generator then it is matched against the whole prior value; if the return value and the prior value is a list then the list members are matched against each other if the lengths of the lists are the same. The same applies to tuples. If at some point the algorithm cannot match a prior value against the newly constructed value $v_{new}$ anymore, then $v_{new}$ is used as the new value. The matching algorithm is shown in Algorithm 3.

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

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

```
graph_duplicated_edges() ->
  ?LET(Vn, integer(2, 42),
  begin
    Vs = lists:seq(1, Vn),
    \{Vs, list(edge(Vs))\}
  end).
```
Algorithm 3: Match the base value $b$ against the intermediate result $m$ with temperature $t$.

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

After generating a $v_n$ value, this generator returns a tuple containing the list of vertices and a basic generator for the list of edges. During random PBT, PROPER would process this return value by generating a fresh list of edges for every test. The constructed $N_F$ retrieves the base value of $v_n$ from the cache and constructs a $N_F$ from the generator integer(2, 42). This inner $N_F$ is used to produce the next $v_n$ in the neighborhood of the cached previous $v_n$. The $f_{in}$ function then returns a tuple containing a list of vertices as immediate value and a generator for the list of edges. The constructed $N_F$ then matches the previous list of edges to the generator for the new list of edges. $\text{list}(\text{edge}(V_s))$ is a basic generator and the neighborhood relation can be resolved with the matched previous value. Matching tries to optimistically match the base value against the intermediate values of the generation. Because of this, some base values may be matched that are not valid anymore. In our graph_duplicate_edges() generator, it can for example happen that some of the edges in the base value for the list of edges are not valid edges anymore because the set of vertices got smaller. Therefore it is necessary to check that the elements of the list are valid according to their generator.
To preserve the integrity in such cases, we replace during matching each list element with a neighbor that is constructed with a special zero temperature. At zero temperature, the N\textsuperscript{F} does not alter the base value if it is a valid input value. If the input value is not valid, a new random input value is generated. The advantage of using the constructed N\textsuperscript{F} is that the base value is traversed in the same way as if every element would be modified. Only the part of the base value that is invalid is constructed again. In our example, this results in more similar list elements than if we would only check whether whole elements are valid input and generate new elements if not.

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

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

For this generator, the constructed N\textsuperscript{F} will generate a \(V_n\) that is in the neighborhood of the previous one. The list of edges will be generated randomly for each generated input value because matching and caching do not work properly in this case. This happens because:

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

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

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

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

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

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

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

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

### Table 5.1. Algorithms for constructing Nfs.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n(b, t)$ for atom()</td>
<td>— the algorithm for binary() is similar</td>
</tr>
<tr>
<td>1 $g_{\text{chars}} \leftarrow \text{list}(\text{integer}(0, 255))$;</td>
<td></td>
</tr>
<tr>
<td>2 $n_{\text{chars}} \leftarrow \text{construct}<em>{\text{nf}}(g</em>{\text{chars}})$;</td>
<td></td>
</tr>
<tr>
<td>3 $\text{next}<em>{\text{chars}} \leftarrow n</em>{\text{chars}}(\text{atom_to_list}(b), t)$;</td>
<td></td>
</tr>
<tr>
<td>4 return $\text{list_to_atom}(\text{next}_{\text{chars}})$;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n(b, t)$ for exactly$(Value)$</td>
<td></td>
</tr>
<tr>
<td>1 return $Value$;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n(b, t)$ for integer$(l, h)$</td>
<td></td>
</tr>
<tr>
<td>1 $r \leftarrow (h - l) \times 0.05 \times t$;</td>
<td></td>
</tr>
<tr>
<td>2 $o \sim U([-r, r])$;</td>
<td></td>
</tr>
<tr>
<td>3 return $\text{max}(l, \text{min}([b + o], h))$;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n(b, t)$ for list$(g_{\text{element}})$</td>
<td></td>
</tr>
<tr>
<td>1 $\text{growthCoeff} \sim U([0.1, 0.9])$;</td>
<td></td>
</tr>
<tr>
<td>2 $n_{\text{element}} \leftarrow \text{construct}<em>{\text{nf}}(g</em>{\text{element}})$;</td>
<td></td>
</tr>
<tr>
<td>3 foreach element $e$ in $b$ do</td>
<td></td>
</tr>
<tr>
<td>4 operation $\sim \text{List_getOperation}(\text{growthCoeff}, t)$;</td>
<td></td>
</tr>
<tr>
<td>5 switch operation do</td>
<td></td>
</tr>
<tr>
<td>6 case add do</td>
<td></td>
</tr>
<tr>
<td>7 $v \sim g_{\text{element}}$;</td>
<td></td>
</tr>
<tr>
<td>8 insert $v$ after $e$ in $b$</td>
<td></td>
</tr>
<tr>
<td>9 case del do</td>
<td></td>
</tr>
<tr>
<td>10 delete $e$ from $b$</td>
<td></td>
</tr>
<tr>
<td>11 case modify do</td>
<td></td>
</tr>
<tr>
<td>12 $\text{next}<em>e \leftarrow n</em>{\text{element}}(e, t)$;</td>
<td></td>
</tr>
<tr>
<td>13 exchange $e$ with $\text{next}_e$ in $b$</td>
<td></td>
</tr>
<tr>
<td>14 otherwise do</td>
<td></td>
</tr>
<tr>
<td>15 -</td>
<td></td>
</tr>
<tr>
<td>16 end</td>
<td></td>
</tr>
<tr>
<td>17 end</td>
<td></td>
</tr>
<tr>
<td>18 end</td>
<td></td>
</tr>
<tr>
<td>19 return updated $b$;</td>
<td></td>
</tr>
</tbody>
</table>

[ continued on next page ]
\[ n(b, t) \text{ for tuple}[[g_1, g_2, \ldots]] \]

1. **foreach** element \( e_i \) in \( b \) **do**
2. \( \text{operation} \sim \text{Tuple.getOperation}(t); \)
3. **switch** operation **do**
4. **case** modify **do**
5. \( n_f_i = \text{construct}_n(f)(g_i); \)
6. \( P_{next, e_i} \leftarrow n_f_i(e_i, t); \)
7. exchange \( e_i \) with next \( e_i \) in \( b \)
8. otherwise **do**
9. **end**
10. **end**
11. **return** updated \( b \);

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

1. \( \text{operation} \sim \text{getOperation}(t); \)
2. **switch** operation **do**
3. **case** change **do**
4. \( g \sim U([g_1, g_2, \ldots]); \)
5. \( \text{next}_g \sim g; \)
6. **return** \( \text{next}_g; \)
7. **case** modify **do**
8. \( G_s \leftarrow \text{all } g_i \text{ where } b \text{ is an instance}; \)
9. \( g \sim U(G_s); \)
10. \( n_f_g \leftarrow \text{construct}_n(f)(g); \)
11. **return** \( n_f_g(b, t); \)
12. otherwise **do**
13. **return** \( b; \)
14. **end**
15. **end**

\[ n(b, t) \text{ for float}([l, h]) \]

1. \( r \leftarrow (h - l) \ast 0.05 \ast t; \)
2. \( o \sim U([-r, r]); \)
3. **return** \( \max(l, \min(b + o, h)); \)

\[ n(b, t) \text{ for fixed_list}([t_1, t_2, \ldots]) \text{ --- see tuple}([t_1, t_2, \ldots]) \]

\[ n(b, t) \text{ for vector}(n, \text{element_type}) \]

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

[ continued on next page ]
\[ n(b,t) \text{ for } \texttt{?USERNF}(g, n_{f_{user}}) \]

1. \( \text{retval} \leftarrow n_{f_{user}}(b,t) \);
2. \textbf{return} match\((b, \text{retval}, t)\);

\[ n(b,t) \text{ for } g = \texttt{?LET}(x, g_{in}, f_{in}) \]

1. \( n' \leftarrow \text{construct\_nf}(g_{in}) \);
2. \textbf{if} \((\text{prev}_x \leftarrow \text{cache\_lookup}((g,b))) \text{ exists} \) \textbf{then}
3. \hspace{1em} \text{next}_x \leftarrow n'(\text{prev}_x, t);
4. \textbf{else}
5. \hspace{1em} \text{next}_x \sim g_{in} ;
6. \textbf{end}
7. \text{next} \leftarrow \text{match}(b, f_{in}(\text{next}_x), t);
8. \text{cache\_store}((g, \text{next}), \text{next}_x);
9. \textbf{return} \text{next} ;

69
6. Targeted Stateful Property-Based Testing

The testing of stateful systems is a core component of property-based testing tools. In this chapter, we extend targeted PBT (TPBT) to the domain of stateful testing. Fortunately, TPBT can directly be applied to stateful testing since the general structure of properties for stateful testing is the same as for any other property: A \textsc{proper} generator produces input in the form of a sequence of commands to a property that executes them and checks that the behavior of the system is as expected. TPBT only needs the ingredients required by the used search strategy to guide the generation of the command sequences. For the simulated annealing (SA) search strategy, the default search strategy of \textsc{proper}, we need to provide a neighborhood function (\textsc{nf}) for the input generator. Properties for stateful as provided by \textsc{proper}'s \texttt{proper_statem} module uses a commands generator for input generation. In the previous section, we presented a way to construct \textsc{nf}'s automatically from generators. Unfortunately, this technique will not produce an adequate \textsc{nf} for stateful testing. We analyze the commands generator in Section 6.1 and discuss why the construction algorithm does not perform well. In the following sections, we describe two \textsc{nf} for the commands generation: a conservative \textsc{nf} that operates on the existing interface for stateful testing and an opportunistic \textsc{nf} that changes this interface slightly to provide a more powerful neighbor generation.

6.1 Constructed \textsc{nf} for stateful testing

Ideally, we want to construct the \textsc{nf} with the algorithm that we presented in Chapter 5. The restrictions of the construction algorithm and the unique interactions between the generator for the command sequence, the state update function, preconditions, and postconditions make it unfortunately not a suitable technique to apply for the generator that \textsc{proper} uses to
generate command sequences for stateful testing. To get a better understanding of the issue, we will first analyze the command generator which is currently implemented as follows:

```
1 commands(Size, Mod, State, Count) ->
2    proper_types:frequency(
3        [{1, []}],
4        {Size, ?LET(Call, ?SUCHTHAT(X, Mod:command(State), Mod:precondition(State, X)),
5           begin
6              Var = {var,Count},
7              NextState = Mod:next_state(State, Var, Call),
8              ?LET(Cmds,commands(Size-1, Mod, NextState, Count+1),
9                  [{set,Var,Call}|Cmds])
10           end)})).
```

The Generator is a recursive one that builds the commands list up by randomly generating a command from the `command()` generator. This command is generated as a symbolic call such that its precondition is true. After we have a valid command, we create a new symbolic variable for the return value of the symbolic call. In the next step, the symbolic state is updated, and the generator recursively calls itself to generate the rest of the commands. In Section 2.2, Fig. 2.1 illustrates in detail how the command generation works.

In Line 4 the generation of the command and the validity of its precondition is dependent on the current symbolic state. The symbolic state, however, changes after each command gets selected. This means that the commands we can choose in each step can, in theory, be completely different depending on which commands we have chosen earlier. Due to the limitations of the construction algorithm that we presented in Chapter 5, we will not be able to generate a NF that produces good neighbors automatically.

However, since the generator of stateful testing is a fixed component that is shared by all stateful properties, we can construct a general NF which then can be used by all targeted stateful properties. In the following sections we will present two NF for targeted stateful PBT: first, we will present a hand-written conservative NF that works with the existing interface that is provided by `proper_statem`, and a more opportunistic NF that changes this interface slightly to provides a more powerful NF.
6.2 A Conservative Neighborhood Function

The conservative neighborhood function preserves the interface that `proper_state` provides for stateful testing. The existing random commands (Mod) generator produces command sequences such that the commands that get chosen (and their call arguments) are dependent on the current symbolic state. After a candidate for such a command has been generated, PROPER checks if its precondition, which in general is also dependent on the state, holds and if so adds it to the list of generated commands. If the precondition fails, the command is discarded, and a new command is generated instead. This restricts the degree of freedom we have when writing a NF for these commands. Let us consider the following generated command sequence that PROPER would generate for the model of the car we defined in Section 2.2.

```plaintext
1 [{set, {var, 1}, {call, car, start, []}},
2 {set, {var, 2}, {call, car, drive_along, [19.553117710657048]}},
3 {set, {var, 3}, {call, car, drive_along, [12.151024131878811]}},
... 
12 {set, {var, 12}, {call, car, stop, []}},
13 {set, {var, 13}, {call, car, start, []}},
14 {set, {var, 14}, {call, car, break, [4.679224474476533]}},
15 {set, {var, 15}, {call, car, accelerate, [5.335291151597263]}}]
```

Each element in the commands list is a three-tuple beginning with the atom `set`, followed by a symbolic variable and a symbolic call. The symbolic variable represents the return value of the symbolic call. We could define a neighbor of this command sequence as a similar sequence where some commands are modified, e.g., exchanged to a different command, deleted or similar. We now consider what happens if the command in Line 2 is modified. By modifying the command, we need to invalidate all commands that are scheduled afterward, namely the commands from Line 3 to Line 15. The commands following the one that we just modified are generated based on the symbolic state that was produced by `next_state()` after the original command. The `next_state()` function will now most likely produce a different state on which the generation of the following commands depends. A similar argument can be made for the preconditions of the following commands that now might return `false`. It is not enough to only invalidate the commands that violate their precondition. The reason for this is that
the \texttt{command(S)} generator implements constraints about which command is generated with which arguments independently from the precondition. In general, the precondition and the commands do not cover the same set of constraints.

It is still possible to formulate a neighborhood function that produces similar command sequences with these limitations in mind. The conservative \texttt{N_F} does exactly this. First, it selects a command from the previously generated list of commands and removes all commands that are scheduled after this command including the command itself to produce a shortened base list. The conservative To construct a neighbor the \texttt{N_F} generates a new list of commands following the algorithm of the commands generator and appends it to the shortened base. Listing 6.1 shows the implementation of a so behaving \texttt{N_F} for commands.

This algorithm produces similar neighbors but has the disadvantage that commands at the beginning of the list are less likely to be altered, especially if the command list grows long due to the search of TPBT. This can be an issue if some of the first commands need to removed to reach the search goal. The search could then be stuck in a local optimum from which it cannot escape. In practice, this might not be a problem. The user can restart the search periodically to explore command sequences with different beginnings. Each of those command sequences would then be guided by TPBT until the search is reset again.

6.3 An Opportunistic Neighborhood Function

The \texttt{proper_statem:commands(Module)} generator as provided by \texttt{PROPER} has two ways of specifying which commands can be added to the list of commands, the \texttt{command(S)} generator and the \texttt{precondition(S, Call)}, which are both dependent on the state \texttt{s}. This is very powerful, and the command selection can follow, e.g., the transitions of a state machine. In practice, this high level of control when writing the model might not be necessary. While it is important to make the arguments of the commands dependent on the state, the selection of which command gets generated does not need to depend on the state. This is especially true if we preserve the filter function of
Listing 6.1. The hand-written conservative NF for the `commands(Mod)` generator.

the precondition and keep the dependence of the precondition on the state: only commands that fulfill the precondition get scheduled for execution. We change the interface for the `command(S)` generator to `static_command()` as follows:

```erythrea
-define(ARGUMENTS(X, Args), fun (X) -> Args end).

static_command() ->
  oneof([  
    {call, ?MODULE, start, ?ARGUMENTS(_, [])},  
    {call, ?MODULE, stop, ?ARGUMENTS(_, [])},  
    {call, ?MODULE, accelerate, ?ARGUMENTS(S, [acceleration_force(S)])},  
    {call, ?MODULE, break, ?ARGUMENTS(S, [break_force(S)])},  
    {call, ?MODULE, drive_along, ?ARGUMENTS(_, [td()])}  
  ]).
```
static_commands(Mod) ->

?LET(RawCmds, list(Mod:static_command()),

begin

{Cmds, _, _} = lists:foldl(

fun Func(Call = {call, _, _, Args}, Acc = {AccCmds, State, Count})

when is_list(Args) ->

  case Mod:precondition(State, Call) of

    true ->
        Var = {var,Count},
        NextState = Mod:next_state(State, Var, Call),
        {{[set,Var,Call]|AccCmds], NextState, Count+1};

    false ->
        Acc
  end

end,

Func({call, M, F, RawArgs}, Acc = {_, State, _}) ->

case RawArgs(State) of

Args when is_list(Args) -> Func({call, M, F, Args}, Acc);

_ -> error(cannot_resolve_args)

end

end,

{{[], Mod:initial_state(), 1}, RawCmds},
lists:reverse(Cmds)
end).

Listing 6.2. The static_commands(Mod) generator, that calculates the states after the full sequence of commands has been generated.

Now, this generator does no longer depend on the symbolic state \( S \). Note that we now have to wrap the arguments for the commands inside the \(?\text{ARGUMENTS}\) macro. The first argument to \(?\text{ARGUMENTS}\) is the name of the symbolic state, and the second argument is the list of arguments which can be written as before. This allows us to delay the evaluation of the arguments generators until the symbolic state for them is available. To generate the list of commands we define a static_commands(Module) generator as shown in Listing 6.2.

The generator instantiates the complete list of commands and then filters the list according to the defined preconditions. If a call fulfills its precondition, the symbolic state is updated with the symbolic call, and the call is added to the result. If the call does not satisfy its precondition, it is skipped and the symbolic state remains as it is. With this infrastructure in place, we can use the constructed N\(_F\) for static_commands(Module) as a starting point:

\[
\begin{align*}
\text{prop_targeted_consumption}() & \rightarrow \\
?\text{NOT_EXISTS}(\text{Commands, \#(\text{gen} \Rightarrow \text{static_commands(?MODULE)}), ...})
\end{align*}
\]
This will work well as long as one does not care about the arguments of the calls. With a constructed $N_F$ like this, the arguments for the call get generated randomly for each neighbor. The default matching that PROPER provides is a structural matching that matches structural identical data structures against each other. For lists, this means that only lists of equal length are matched. If the lists are of different length, a new random value is produced based on the intermediate value. In the case of the `static_commands` generator, this means that the commands itself are in the neighborhood of the previously generated commands. The arguments however are not and are generated randomly for each neighbor. Figure 6.1 illustrates this behavior on our car model.

However, if we track the commands with a unique key we can match them using these keys, instead of matching them structurally. We achieve this by overwriting the default matching function with one that takes the keys into account. We introduce a new annotation `?USERMATCHER` that provides us with the possibility to overwrite the matching algorithm that the construction algorithm uses with one that the user provides. This matching function takes three arguments: the base value, the intermediate value of the generated neighbor, and the temperature. The matcher returns the complete constructed neighbor of the base value. Implementing the full matching for a complex generator is a complex task. For our command generator, for example, we only want to change one matching step, the matching of commands. We want to keep the structural matching for the rest of the input data, especially for the command arguments. We therefore expose part of the matching interface, so that it is possible to hand the matching back to PROPER via the functions `match()` and `structural_match()`. The interface of these functions is the same as the one of the user-defined function, with one difference. If the base value is the atom `no_matching` the matching algorithm will generate a new random value based on the intermediate value given in the second argument.

To keep track of individual commands, we provide a unique key using `make_ref()` for each newly generated command. Additionally, we wrap the generator inside another `?LET` construct that overwrites the matching algorithm with a hand-written one. Finally, we remove the keys after the match-
ing to return a command sequence that is compatible with the interface of
proper_statem:

```
1 cmd_with_key(Mod)
2  ?LET(Cmd, Mod:static_command(), {make_ref(), Cmd}).
3
4 static_cmds_with_keys(Mod) -> % previously static_commands(Mod)
5  ?LET(RawCmds, list(cmd_with_key(Mod)),
6    begin
7     ...
8     end).
9
10 static_commands(Mod) ->
11  ?LET(CmdsWithKey, ?USERMATCHER(static_cmds_with_keys(Mod), fun key_matcher/3),
12    begin
13    {_, Cmds} = lists:unzip(CmdsWithKey),
14    Cmds
15    end).
```

The matching function that we provide matches commands which share
the same key and hands the matching back to PROPER’s structural matching
strategy. The following implementation of the function key_matcher() does
exactly this:

```
1 key_matcher(Base, ToMatch, Temp) ->
2  Pairs = [{safe_keyfind(Key, Base), KeyCmd} || KeyCmd = {Key, _} <- ToMatch],
3    [proper_sa_gen:structural_match(B, RT, Temp) || {B, RT} <- Pairs].

5 safe_keyfind(Key, List) ->
6    case lists:keyfind(Key, 1, List) of
7      false -> no.matching;
8      KeyCall -> KeyCall
9    end.
```

We now have a Nf for stateful testing that can produce neighbors by chang-
ing any element in the sequence of commands of the base value. We had
to limit the functionality of the command(S) generator by removing the argu-
ment to the state. However, we preserved the state argument for the argu-
ments of the commands and the precondition, so that the possibility
to carefully select the commands remains intact. It should be noted, that
one should be careful with strict preconditions. There is no mechanism
for trying to generate more commands if some preconditions fail. If the
Figure 6.1. Example of command sequences as generated by the conservative NF and the opportunistic with and without matching NF. The left side shows the previous sequence of commands and the right side the neighbors as produced by the associated NF's.

preconditions are too strict, then all commands can get filtered out and the command list is empty. The search strategy will not be able to guide the input generation towards, e.g., longer sequences of commands in such cases.

Figure 6.1 shows on the example of the car model, how the NF's produce neighbors. We see that only opportunistic NF with matching can modify the command sequence so that also neighbors of the arguments get generated (orange). The conservative NF and the opportunistic NF without matching only remove (red) or add (green) commands. Additionally, the opportunistic NF without matching produces new instances for all arguments when generating a neighbor.

Writing the NF for the command generator entirely by hand would have been a significant amount of work. With the annotations for the construction algorithm, we reduce this task to implementing the matcher function.
7. Evaluation

This chapter contains a series of case studies that evaluate the frameworks and techniques presented in Chapters 3 to 5. In the first case study, we test whether the energy consumption of the radio duty cycle protocol X-MAC is within some specific bound. We compare the performance of random PBT and TPBT. Furthermore, we compare the performance of the hand-written $N_F$ to that of an automatically constructed $N_F$ from the same generator. 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. The third 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. We also compare the performance of random PBT to that of TPBT. Additionally, we present an alternative approach to generating these routing trees and demonstrate how a constructed $N_F$ compares against a hand-written one. The next case study uses TPBT 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. Finally, we present a small user study in which we try to quantify the trade-offs between ease of use and testing performance when using constructed neighborhood functions.

7.1 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. Therefore we want to test the duty cycle of the nodes' radio component, the component that usually consumes the most energy [74]. Using our PBT framework we examine if there are network configurations for which the duty cycle is above a certain threshold.
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).

Listing 7.1. Property to check the duty cycling of a random WSN.

Our setting uses two node types: User Datagram Protocol (UDP) server and UDP client nodes. Client nodes send periodically messages to server nodes. We use the “rpl-udp” example from the Contiki distribution as a base for our experiment. As the name suggests, the nodes use the Routing Protocol for Low-Power and Lossy Networks (RPL) [79] and UDP with IPv6 to communicate with each other. As Media Access Control (MAC) layer, the network layer that performs duty cycling, we use Contiki’s implementation of X-MAC [15].

We list the property for this scenario in Listing 7.1. 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 an empty network (no nodes). We then 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 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

A sensor node is usually called a mote.
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 $100 \times 100$ units. The transmission range of all nodes is 50 units. The `mote_type()` is either a UDP client ("sky1") or UDP server ("sky2"). The following code snippet lists the generators that create network configurations:

```erlang
1 configuration() ->
2     list(mote()).
3 mote() ->
4     tuple([coordinates(), mote_type()]).
5 coordinates() ->
6     tuple([float(0.0, 100.0), float(0.0, 100.0), 0.0]).
7 mote_type() ->
8     weighted_union([[50, "sky1"], [50, "sky2"]]).
```

In `mode_type()`, we specified that the mote types are selected with equal 50% probability only to show how different probabilities could be specified. We use the control layer of our framework to add the generated nodes to the simulation, and to record the hardware events:

```erlang
1   add_motes(Handler, Motes) ->
2       [add_mote(Handler, Mote) || Mote <- Motes].
3   add_mote(Handler, {Pos, Type}) ->
4       {ok, ID} = nifty_cooja:mote_add(Handler, Type),
5       ok = nifty_cooja:mote_set_pos(Handler, ID, Pos),
6       ok = nifty_cooja:mote_hw_listen(Handler, ID),
7       ID.
```

In our first experiment, we test the property of Listing 7.1. The property fails on a network configuration of around fifteen nodes. The left window of Fig. 7.1 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 2.2 that reduced the length of the string, shrinking in this case tries to reduce the number 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. 7.1. 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 $(x = 0, y = 50.25)$. 

83
Figure 7.1. 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 a duty cycle of 10%.

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 [30] is generally regarded as more energy efficient than X-MAC [62]. Therefore, our second experiment is to switch the MAC layer to ContikiMAC and rerun the test on this network configuration.\(^2\) 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 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 in Listing 7.2.

\(^2\)ProPER provides support for saving and reusing a failing test case [26].
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).

Listing 7.2. A property that compares the duty cycle of X-MAC with ContikiMAC.

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. 7.2, 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
er_graph($\{\text{Nominator, Denominator}\}$) ->
\[\text{WeightLink} = \text{Nominator},\]
\[\text{WeightNoLink} = \text{Denominator-Nominator},\]
\[?\text{SIZED}(\text{Size}, \text{er_graph(}\text{Size}, \text{WeightLink}, \text{WeightNoLink}, [], []).\)]

er_graph(0, _, _, V, E) ->
$\{V, E\};$

er_graph(\text{Size}, \text{WeightLink}, \text{WeightNoLink}, \text{Vertices}, \text{Edges}) ->
?LET(\text{LinksDef}, \text{vector(length(Vertices), edge(W1, W2))},
\begin{align*}
&\text{NewVertex} = \text{length(Vertices)},
&\text{NewEdges} = \text{build_edges(LinksDef, Vertices, NewVertex)},
&?\text{LAZY}(\text{er_graph(}\text{Size-1}, \text{WeightEdge}, \text{WeightNoEdge},
&\text{[NewVertex|Vertices]}, \text{NewEdges++Edges}))
\end{align*}
\)

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

Listing 7.3. Graph Generator according to the Erdős-Rényi model.

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.

\text{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 graphs according to the Erdős-Rényi model [32] can be implemented as shown in Listing 7.3.

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 [26] for the explanation of $\texttt{?SIZED}$, $\texttt{?LET}$ and $\texttt{?LAZY}$.)

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.

**Using Targeted Property-Based Testing**

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 in Section 4.1. 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 with TPBT leads to higher confidence in the system’s correctness for a lower number of runs. Moreover, with TPBT we are 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 PROPER with TPBT to guide the generation of the sensor networks towards one with a high duty cycle. We chose a duty cycle of 25% as the 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 TPBT is based on the original property; see Listing 7.4 that shows in red color the differences from the original code of the property. To use TPBT, 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 using the \texttt{?FORALL\_TARGETED} instead of the \texttt{?FORALL} macro and specifying a search strategy (in this case \texttt{simulated\_annealing}). However, since simulated annealing is the default search strategy of our tool, one can even omit setting it in the test options. The utility value is the maximum duty cycle since that is the value we want to maximize. Can we specify this by adding \texttt{?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. We 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 \textit{order} of the graph is $v$ and the \textit{size} of the graph is $e$. We define a graph $G_{next} = \mathcal{N}(G)$ as a graph $(V_{next}, E_{next})$ that has an altered set of vertices and edges. The \textit{order} of the altered graph is

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

(7.1)
sa_graph_next() ->
  fun {{V, E} = _OldInstance, Temperature} ->
    %% generate new order according to (7.1)
    ?LET(NewSize, tinteger(length(V), Temperature),
    %% generate add and del operations as in (7.2)
    ?LET({Add, Del}, get_op_count(NewSize, length(V), Temperature),
    %% perform operations according to (7.3), and (7.4)
    ?LET({V Add, E Add}, add_vs(V, E, Add, Temperature),
    ?LET({V Del, E Del}, del_vs(V Add, E Add, Del),
    %% change edges
    ?LET(NewEdgeSize, tinteger(length(E Del), Temperature),
    ?LET(E EAdd, add_es(V Del, E Del, E Add),
    ?LET(E EDel, del_es(E EAdd, E Del), {V Del, E EDel}))))))
end).

Listing 7.5. Code to generate the next graph when using the SA strategy. The
generation steps of the neighborhood function correspond to (7.1) - (7.4).

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:

\[
order_{\text{next}} = order + v_{\text{add}} - v_{\text{del}}
\] (7.2)

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

\[
V_{\text{next}} = (V \cup A) \setminus D
\] (7.3)

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:

\[
E_{\text{add}} \subseteq (V \cup A) \times A
\]
\[
E_{\text{del}} = \{(v_i, v_j) \in (E \cup E_{\text{add}}, v_i \in D \lor v_j \in D)\}
\]
\[
E_{\text{intermediate}} = (E \cup E_{\text{add}}) \setminus E_{\text{del}}
\] (7.4)

The transition from \(E_{\text{intermediate}}\) to \(E_{\text{next}}\) works like the transition from \(V\) to
\(V_{\text{next}}\).
It is important to note that the size of the parameters involved in the manipulation of the graph is 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_targeted which was shown in Chapter 4 in Listing 4.1 with the difference that the order of the graph is not fixed. Listing 7.5 shows the implementation of these definitions as a graph generator for SA strategy.

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

Figure 7.3 shows the achieved duty cycle for one test of the property with random PBT (the graph on the left) and with targeted PBT 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 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 targeted PBT, 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. 7.3. 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, with TPBT we were 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 TPBT is higher mainly because
Figure 7.3. The achieved duty cycle (y-axis) with random PBT (left) and targeted PBT with Simulated Annealing (right) varying the number of tests (x-axis). The graphs also show that targeted PBT requires significantly less tests to find a counterexample.

PROPER guides the generation towards larger networks which require more resources to simulate. Thus, PROPER with targeted PBT 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.

Automatically Constructing the Neighborhood Function

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

We exchanged the hand-written NF with an automatically constructed one using the \texttt{graph()} generator from Section 5.2 and tested the otherwise unchanged property. On average the property failed after 2h19m, which is very similar to 2h12m achieved by the hand-written neighborhood function, showing that the automatic construction of NFs is quite good in this case.
7.2 Contiki’s Socket API

In 2014, 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 [26] comes in handy.

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.

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:

```prolog
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).
```

92
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. 7.4. In the language of PROPER, the transitions of this FSM are specified as follows:

```plaintext
1  closed(S) ->
2    [{listening, {call, ?MODULE, listen, [sockets(S), tcp_port()]}}],
3    {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,
```

94
After this change, the property fails again. After shrinking, we end up with a 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 this 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 [51, 52].
prop_directional_antennas() ->

?FORALL_TARGETED(Tree, tree(),

  begin
    HOPS = get_total_hops(Tree),
    PDR_AVG = get_avg_pdr(Tree),
    Metric = (1 - PDR_AVG) * HOPS,
    ?MINIMIZE(Metric),
    ... % condition of the property here
  end).

Listing 7.6. The property for directional antennas minimizing the metric for the energy consumption.

7.3 Routing Tree for Directional Antennas

With this case study, we wanted to see how well targeted PBT 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 [67, 70]. 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 affected. 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 targeted PBT 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 $pdr_{\text{avg}}$ 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 $1 - pdr_{\text{avg}}$. We then specify to minimize this value using the Simulated Annealing strategy as shown in Listing 7.6.

We compare the number of test runs against random PBT. The left graph of Fig. 7.5 plots the maximum, median, and minimum of the achieved minimum energy metrics for 100 runs of 10,000 tests using random PBT in comparison to using targeted PBT with SA. While random PBT is able to find decent inputs, targeted PBT 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 $\text{MINIMIZE(Metric)}$ expression to $\text{MAXIMIZE(Metric)}$ 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. 7.5 shows the result of 10,000 runs of the property compared to random PBT. We can see that targeted PBT can produce routing trees with an energy consumption that is much higher much faster. After 1,000 tests the worst
run with targeted PBT is already better than the best result achieved with random PBT after 10,000 tests. Both 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 targeted PBT.

Automatically Constructing the Neighborhood Function

We now want to compare how well the construction algorithm for N\textsubscript{F} performs when applied to this property. The experimental setup so far used a recursive random generator and a complicated hand-written N\textsubscript{F}. This random generator produces a random tree with directions, recursively choosing one link at a time. The hand-written N\textsubscript{F} alters the tree by moving subtrees to different parents. Because of the limitations of our construction algorithm, it is hard to obtain good performance when using this recursive generator as the basis for an automatically constructed N\textsubscript{F}.

We therefore implemented a different random generator based on random minimum spanning trees [35]. This generator produces random sending directions and a random weight for each possible link. A spanning tree with minimum weights is then deterministically calculated from these weights. We implemented a hand-written N\textsubscript{F} that switches the weights on a subset of all possible links. This alters the order in which the links get chosen when building the minimum spanning tree. After changing the weights, the hand-written N\textsubscript{F} generates new sending and receiving directions for some of the links.

In Fig. 7.6 we plot the achieved minimum, median, and maximum of the estimated energy consumption when instructing the search strategy to optimize it in either direction. The two graphs show the median line and the range for all runs illustrating the worst case and the best case. We also show how random PBT performs compared to TPBT.

Targeted PBT outperforms random PBT with either N\textsubscript{F}. After less than 1,000 tests, the worst case run of TPBT produces inputs with a higher
Figure 7.6. The y-axis shows a metric of the energy consumption over tests (x-axis). The graph shows the minimum, median, and maximum of random PBT (orange, dots) in comparison to targeted PBT with a hand-written (green, lines) and a generated Nf (blue, solid) of the achieved minimums (left graph) and maximums (right graph) of 100 runs with 10,000 tests.

estimated energy consumption than the best case of random PBT when instructed to maximize the measure (right graph). The constructed Nf performs even slightly better than the hand-written one. When minimizing (left graph), both Nf perform similarly.

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

7.4 Noninterference

In this last case study, we use targeted PBT to test for noninterference. Hrițcu et al. [41] 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, Hrițcu et al. [41] 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 Hrițcu et al. [41].) This generation theme fits well into the framework of TPBT.

We can easily implement the ByExec technique by writing targeted generators for the 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 targeted 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 [41].

As a second targeted generator, we use a generic List generator again with simulated annealing. This generator is based on creating neighboring lists by adding and deleting elements to it. This generator is built-in into PROPER.
and can be used “out-of-the-box” simply by supplying it some information about the type of the list elements. Similar to the targeted 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 TPBT, 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 [41]. 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 7.1 shows the measured times along with the arithmetic and geometric mean times to failure (MTTF). We can see that the targeted 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 of Hrițcu et al. [41]. Both implementations of ByExec required only around 30 lines of generator code.

Even more noteworthy is that the targeted List generator performs surprisingly well. It is slower than the specialized targeted ByExec generator but faster than random PBT using ByExec, which is more sophisticated genera-

<table>
<thead>
<tr>
<th></th>
<th>PBT</th>
<th>Targeted PBT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Naive</td>
<td>ByExec</td>
</tr>
<tr>
<td>ADD</td>
<td>2234.08</td>
<td>312.97</td>
</tr>
<tr>
<td>PUSH</td>
<td>70.18</td>
<td>9.79</td>
</tr>
<tr>
<td>LOAD</td>
<td>324028.34</td>
<td>987.91</td>
</tr>
<tr>
<td>STORE A</td>
<td>—</td>
<td>4668.04</td>
</tr>
<tr>
<td>STORE B</td>
<td>226.85</td>
<td>8.19</td>
</tr>
<tr>
<td>STORE C</td>
<td>130.22</td>
<td>10.01</td>
</tr>
<tr>
<td>MTTF (geometric)</td>
<td>—</td>
<td>102.44</td>
</tr>
</tbody>
</table>

Table 7.1. Average times (in msecs) to find a counterexample for bugs injected to the stack machine of [41].
tion 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 PROPER’s TPBT 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 PROPER’s targeted List generator can significantly reduce the implementation effort that is required by the programmer and also achieve good testing effectiveness.

Automatically Constructing the NF

We want to evaluate how well a constructed NF performs in finding the injected bugs of the IFC machine and compare its performance against the hand-written NF and random PBT.

Our baseline for random PBT is the SEQUENCE generator as described in [41]. This generator produces a random list of instructions where each instruction gets chosen with a weighted distribution. It is for example more likely that a \texttt{push} instruction gets generated than an \texttt{add} or a \texttt{noop}. Additionally, sequences of instructions that make sense together such as a \texttt{push} followed by a \texttt{load} are produced.

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

In Table 7.2 we list the average times it took for each technique to find each of the six injected bugs. As expected, random generation of the programs is slowest in finding counterexamples. TPBT with the hand-written NF performs best in this experiment. However, the automatically constructed NF is quite competitive to it and achieves acceptable performance in all cases.
The **ByEXEC** generation strategy, both when using random and targeted PBT, finds all injected bugs faster but also requires more effort to implement than the generator we use here.

### 7.5 Small User Study

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

The property that we provided the students with tested the spell system component of an imaginary role-playing game for an exploit that would

<table>
<thead>
<tr>
<th>PBT</th>
<th>Targeted PBT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Random</strong></td>
<td><strong>Handwritten</strong></td>
</tr>
<tr>
<td>ADD</td>
<td>5800.57</td>
</tr>
<tr>
<td>PUSH</td>
<td>338.90</td>
</tr>
<tr>
<td>LOAD</td>
<td>7764.15</td>
</tr>
<tr>
<td>STORE A</td>
<td>16997.81</td>
</tr>
<tr>
<td>STORE B</td>
<td>341.02</td>
</tr>
<tr>
<td>STORE C</td>
<td>336.45</td>
</tr>
<tr>
<td><strong>MTTF (arithmetic)</strong></td>
<td>5263.15</td>
</tr>
<tr>
<td><strong>MTTF (geometric)</strong></td>
<td>1760.45</td>
</tr>
<tr>
<td>Average tests per second</td>
<td>9479</td>
</tr>
</tbody>
</table>

**Table 7.2.** *Average times (in msecs) to find a counterexample comparing random PBT with TPBT using a hand-written N\(_F\) and a constructed N\(_F\).*
We received twelve solutions for this task. It is noteworthy that every student who submitted a NF as solution solved the problem and was able to find a combination of spells that falsified the given property.

To evaluate the performance of the hand-written NF compared to the constructed NF, we tested the property 500 times with each NF and recorded the times required to find a counterexample. Figure 7.7 plots these times for each NF combined with their distribution. A well-implemented NF that

---

Figure 7.7. The graph plots, for each neighborhood function (labeled A-L), the times (in seconds) required to find a counterexample for the property (dots) and their distribution (boxes), repeating the experiment 500 times. The numbers below the labels are the lines of code for each hand-written implementation. The rightmost entry (Gen) corresponds to the generated neighborhood function.
Table 7.3. Strategies used by the participants of the user study for implementing the neighborhood function.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random expansion</td>
<td>add new spells at random locations</td>
</tr>
<tr>
<td>random reduction</td>
<td>remove spells from random locations</td>
</tr>
<tr>
<td>strategic expansion</td>
<td>add new spells at “well-chosen” locations</td>
</tr>
<tr>
<td>strategic reduction</td>
<td>remove spells from “well-chosen” locations</td>
</tr>
<tr>
<td>shuffling</td>
<td>change the order of the spells</td>
</tr>
<tr>
<td>modification in-place</td>
<td>alter spells in-place</td>
</tr>
<tr>
<td>restart</td>
<td>discard the current input and generate a new random input</td>
</tr>
<tr>
<td>spell selection</td>
<td>select in a more sophisticated way the spells that are added, removed or modified</td>
</tr>
</tbody>
</table>

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

The number of lines of code required for the implementation of the neighborhood functions ranged from 7 to 57. To further analyze their complexity, we identified a total of eight classes of neighbor selection strategies that were used to generate the next neighbor. The names of these strategies and a short description of them appear in Table 7.3. In all student submissions, multiple strategies were combined. We could identify that at least random expansion or random reduction strategies were necessary to achieve good results in terms of testing performance. Additionally, we observed that the intensity of the modifications done to generate a neighbor affected the overall performance. Finally, many NFs tuned the number
of reductions, expansions, and alterations made to the base value in each generation step.

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

Property-Based Testing

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 [23] 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 [81]. In the context of Erlang alone, the high-level programming language that we use in this dissertation to express the properties to test and their inputs, three such tools are available: a commercial one by QuviQ [1] and two open-source ones: These tools have been successfully applied to test a wide variety of systems such as telecom systems [7], web services [3, 45], compilers [71], and protocols used in cars [6]. 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).

Two things are noteworthy: (1) TPBT is directly applicable to all these areas. (2) 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.

SmallCheck [76] generates all values of a given type up to some limiting depth and increases this depth progressively. In comparison to SmallCheck, TPBT uses a search strategy to guide the input generation. Therefore, TPBT can handle large input domains. We note that it is possible to implement the exhaustive enumeration of inputs that SmallCheck provides as a search strategy in PROPER. However, such a strategy would ignore the utility values.
KleeNet [77] is a debugging environment that extends the technique of KLEE [17] 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 [49] is an explicit model checker for TinyOS applications built upon the TOSSIM [48] 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 [14]. Their tool translates embedded C into standard C, which is then checked with CBMC, a software verifier for ANSI C. Anquiro [65] 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 and Kwiatkowska 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 also to 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 the framework we presented in 3 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.
Peyrard et al. describe an ongoing effort to formally verify the Contiki operating system and present the verification of its AES and CCM* modules, two of its most critical parts responsible for data encryption [73].

Passive Distributed Assertions (PDA) [75] 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 distribution 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.

**Search-Based Software Testing**

Search-based testing also has a long history. Its first ideas can be traced back to a 1976 paper by Miller and Spooner [64]. Since then search-based testing techniques have been applied to a wide variety of testing areas [2, 13, 16, 40, 57, 60, 69] including structural testing, model-based testing, stress testing, functional 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 [34] 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 TPBT to guide the input generation efficiently even for very complex input.
domains like network topologies or programs. Furthermore, TPBT can be used to test non-functional properties like timing, performance, or resource consumption.

Narrowing-Based Test Data Generation

Luck [46] and UDITA [36] 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, PROPER with targeted PBT 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 the SA strategy.

Quasi-Random Test Data Generation

Adaptive random testing (ART) [19, 21], restricted random testing (RRT) [18], and quasi-random testing (QRT) [20] 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 to these techniques especially of ART [22, 59]. On the other hand, a more recent study by Arcuri and Briand [5] has pointed out several
shortcomings (high cost, only effective under high failure rates, etc.) of the ART technique.

Compared to targeted PBT, 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 for TPBT. Such strategies would however ignore the utility values.

*Program Synthesis*

Program Synthesis is the idea to construct a function or a program from a high-level specification, typically given in the form of constraints such as input-output examples, natural language, partial programs, or assertions. In other words, we specify what we want, and the program synthesizer gives us an implementation that fulfills our need. Program Synthesis is an active area of research since the 1970s [58] and even more so recently. It has applications in data processing, code repair, code suggestion, testing and many others. It applies techniques from constraint programming, machine learning, and stochastic search. The interested reader is referred to the recent survey article by Gulwani, Polozov, and Singh [37] for an overview of the field. In principle, program synthesis techniques can be used to construct neighborhood functions for various domains, given their high-level specifications. However, to the best of our knowledge, our work is the first one that constructs a neighborhood function for simulated annealing from a random generator or another high-level description of the input/solution space.
9. Conclusion

The input generation component of a property-based testing tool is a critical part that influences how effective such a tool is in finding potential bugs. This dissertation improves the input generation component of property-based testing by applying search strategies to it to guide the generation towards values that maximize or minimize some metric of the system under test. Thus, we increase the effectiveness in which the input space is explored and the overall testing performance of property-based testing. Additionally, we maintain the high-level expressiveness of the property specification language that PROPER provides.

In this thesis, we began by arguing 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 method. The experiences gained from working with the framework led to the realization that the complexity of sensor networks and similar systems can hinder thorough testing if the input for the properties is generated purely randomly.

We therefore developed 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 involves a utility value that we seek to maximize or minimize. We showed that targeted property-based testing can reduce the time needed to explore the relevant parts of the input space dramatically.

We presented a technique for targeted property-based testing based on simulated annealing that constructs a neighborhood function automatically from a random generator of inputs. We show that this technique reduces the effort of using targeted property-based testing significantly, making it almost as easy as its random counterpart. We furthermore showed that the efficiency of simulated annealing with these neighborhood functions is most of the time sufficient and in some cases competitive to hand-written ones.
Finally, we extended targeted property-based testing to the domain of testing stateful systems. We provided the necessary infrastructure for stateful testing in the form of two neighborhood functions for the simulated annealing search strategy. The first neighborhood function we present is fully compatible with PROPER’s stateful testing interface but limited in how neighbors can be generated. We therefore also provided a neighborhood function that is more powerful but requires a slight change in the interface while maintaining most of its expressiveness.

With these statements we argue that this dissertation defends the following:

**Thesis**

*Combining search-based input generation techniques with property-based testing can significantly improve the testing performance for many application domains and of sensor networks in particular.*
Bibliography


Andreas Löscher. Closed connection event issued twice #621. 2014. URL: https://github.com/contiki-os/contiki/issues/621 (see p. 95).

Andreas Löscher. SocketAPI closed event is not triggered in some cases #867. 2014. URL: https://github.com/contiki-os/contiki/issues/867 (see p. 95).


Appendix A.
Artifacts

Section 3.2 describes the *NIF Interface Generator* NIFTY which can be obtained from the following URL:

http://parapluu.github.io/nifty/

The testing framework that is described in Chapter 3 containing the variation of NIFTY for interfacing with sensor nodes can be obtained from the following URL:

https://github.com/parapluu/nifty-contiki

The implementation of targeted property-based testing presented in Chapter 4 and the automation presented in Chapter 5 were carried out in the testing tool PROPER:

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

Prior to January 1994, the series was called Uppsala Dissertations from the Faculty of Science.

Acta Universitatis Upsaliensis
Uppsala Dissertations from the Faculty of Science and Technology
Editor: The Dean of the Faculty of Science

69. **Fredrik Robelius**: Giant Oil Fields – The Highway to Oil. Giant Oil Fields and their Importance for Future Oil Production. 2007.
70. **Anna Davour**: Search for low mass WIMPs with the AMANDA neutrino telescope. 2007.
71. **Magnus Ågren**: Set Constraints for Local Search. 2007.
77. **Samson Keleta**: Double Pion Production in $\nu$ Reaction. 2008.
82. **Arvid Pohl**: Search for Subrelativistic Particles with the AMANDA Neutrino Telescope. 2009.
84. **Karin Schönning**: Meson Production in pd Collisions. 2009.
85. **Henrik Petrén**: η Meson Production in Proton-Proton Collisions at Excess Energies of 40 and 72 MeV. 2009.
88. **Sophie Grape**: Studies of PWO Crystals and Simulations of the $\bar{p}p \rightarrow \bar{\Lambda} \Lambda, \bar{\Sigma}^0$ Reactions for the PANDA Experiment. 2009.
96. **Liselott Dominicus van den Bussche.** Getting the Picture of University Physics. 2010.
97. **Olle Engdegård.** A Search for Dark Matter in the Sun with AMANDA and IceCube. 2011.
100. **Patrik Adlarson.** Studies of the Decay $\eta \rightarrow \pi^+ \pi^- \pi^0$ with WASA-at-COSY. 2012.
104. **Linda Åmand.** Ammonium Feedback Control in Wastewater Treatment Plants. 2014.
105. **Mikael Laaksoharju.** Designing for Autonomy. 2014.
108. **Mojtaba Soltanalian.** Signal Design for Active Sensing and Communications. 2014.
110. **Andrzej Pyszniak.** Development and Applications of Tracking of Pellet Streams. 2014.
111. **Olov Rosén.** Parallel Stochastic Estimation on Multicore Platforms. 2015.
112. **Yajun Wei.** Ferromagnetic Resonance as a Probe of Magnetization Dynamics. A Study of FeCo Thin Films and Trilayers. 2015.
114. **Alexander Madsen.** Hunting the Charged Higgs Boson with Lepton Signatures in the ATLAS Experiment. 2015.
118. **Li Caldeira Balkeståhl.** Measurement of the Dalitz Plot Distribution for $\eta \rightarrow \pi^+ \pi^- \pi^0$ with KLOE. 2015.
121. **Andrey Andryjev.** Semantic Web Queries over Scientific Data. 2016.