Requirements on automatically generated random test cases

Thomas Arts
Quviq AB

Alex Gerdes
Quviq AB

Magnus Kronqvist
Ericsson AB

Abstract—**Developing, for example, a simple booking web service with modern tools can be a matter of a few weeks work. Testing such a system should not need to take more time than that. Automatically generating tests from specified properties of the system using the tool QuickCheck provides professional developers with the required test efficiency. But how good is the quality of these automatically generated tests? Do they cover the cases that one would have written in manual tests? The quality depends on the specified properties and data generators and so far there has not been an objective way to evaluate the quality of these QuickCheck generators. In this paper we present a method to assess the quality of QuickCheck test data generators by formulating requirements on them. Using this method we can give feedback to developers of such data generators in an early stage. The method supports developers in improving data generators, which may lead to an increase of the effectiveness in testing while maintaining the same efficiency.**

I. Introduction

THIS paper provides a solution to a problem originating from the use of property-based testing of a simple, but realistic web service developed and used by a telecommunication company. Property-based testing [1] is a technique with which one describes properties of a software system using QuickCheck. QuickCheck has many implementations, for example [2], [3]. The general methodology is that one writes properties of the software under test, from which QuickCheck automatically generates test cases to validate these specified properties.

It has been shown that property-based testing increases efficiency and effectiveness of software testing [4]. Prowess [5], a recent EU STREP project, addresses the challenge to reduce time spent on testing, whilst increasing software quality, in order to quickly launch new, or enhancements of existing, web services and internet applications. In this paper we do not evaluate property-based testing, but focus on one particular challenge in using this technique. The use of property-based testing requires the definition of data generators that control QuickCheck's random data generation. There are many ways to define such data generators, and it requires some skills and experience to define a data generator with good data distribution. We explore how we can help developers to measure the quality of their data generators.

A danger in using QuickCheck is that we no longer see the generated test data. In fact, we would not want to see it, because QuickCheck can generate many test cases. As a result, we may be tricked into a false sense of security by a large number of passing tests, but fail to notice that the distribution is badly skewed. Even if we observe it by using QuickCheck's possibility to collect statistics on the test data, we would need an expert to judge whether the provided data is good test data.

We address the manual interaction of judging test data by capturing the expert knowledge in formal requirements. These requirements are used to automatically assess the quality of the test data generators. In this way, the generators can be developed with limited involvement of experts.

A clear example of the problem of judging the quality of data generators came to our attention when testing a web service created by a telecommunication company. Telecommunication systems often use special purpose hardware, which is rather expensive to build. This raises a cost issue for testing such systems; unlike commodity PCs, one cannot simply put as many machines in a test lab as one would like to. Hardware becomes a resource and more efficient use of this resource lowers the total production cost. When sharing resources, one needs a booking system. In our case, the interoperability requirements were to fit the already in-house built continuous integration and other test and deployment tools. Based on these specific requirements and experience with purchasing this kind of heavily integrated software in the past, this booking system was decided to be build in-house by spending a few weeks of effort. This resulted in a simple web service used by several sites in the world described in more detail in Sect. III.

Unit level testing of this system was performed following the existing literature [6], [7] and revealed that it is hard to judge the quality of the generated test cases. One can be mislead in believing that the system is well tested, although the randomly generated test cases do not cover the interesting test cases. We identified the need for assessing the quality of the generated test cases. In Sect. V we describe how we can express requirements on QuickCheck generators that we use to assess the quality of the generated test data.

After successfully using our method in this proprietary first application, we have evaluated the method in a different context. We have used the method to asses the data generators that we use to test our second application: the open source scheduling web application Dudle[8]. In Sect. VI we describe the requirements with which we validated the data generators for testing Dudle.

Although this paper describes the application of our method for two particular applications, the techniques we describe for

Partially supported by the EU STREP project Prowess, grant 317820

formulating requirements on generated test cases are generally applicable to many applications of this kind.

II. QuickCheck

QuickCheck [2] is a tool that tests universally quantified *properties*, instead of single test cases. QuickCheck generates random test cases from each property, tests whether the property is true in that case, and reports test cases for which the property fails. QuickCheck also "shrinks" failing test cases automatically, by searching for similar, but smaller test cases that fail as well. The result of shrinking is a "minimal"[1] failing test case, which often makes the root cause of the problem easy to find.

The original Haskell QuickCheck has inspired a number of different versions for a wide range of programming languages such as C++ [9], Java [10], or ML [11]. The work in this paper is based on the use of QuviQ QuickCheck. QuviQ QuickCheck[2] [3] is a commercial application that includes many advanced features, such as model-based testing using a state machine model [12]. State machine models are tested using a QuickCheck library, which invokes call-backs supplied by the user to generate and test random, well-formed sequences of API calls to the software under test.

An example of a QuickCheck property is shown below. This property is used to test a timeline datatype. A timeline is considered an ordered list of intervals and an interval is an ordered pair of dates. The property uses data generators for an interval (`interval()`) and for a timeline (`timeline()`). The software under test provides a function `add` that, given an interval and a timeline, should add this interval to the timeline, provided the interval does not overlap with an already existing interval. After successfully adding the interval, it should be a member of the newly created timeline.

```
prop_add() ->
  ?FORALL({I, T},
          {interval(), timeline()},
     begin
       case catch add(I, T) of
           {'EXIT', {overlap,_}} ->
             is_overlap(I, T);
           NewT ->
             member(I, NewT)
       end
     end).
```

The functions `is_overlap` and `member` are provided by the software under test as well.

At a unit testing level, one could express a number of such general properties for API functions. This would already be effective in finding a number of defects, but the problem is that it is hard to know when one has provided enough properties to cover the implementation. This problem has been addressed in literature [6] for datatypes. Based on this approach, the solution for the timeline example would be: create a model implementation, a generator for a timeline in which all possible constructors are used in the generation, and one property per operation. This solution, however, does not address the problem of generating data with a good distribution.

Let us consider how to write data generators for an interval and a timeline, which are used in the property above. A simple way to construct a timeline would be to generate a random number of intervals and put those in a timeline. If an interval consists of two completely random dates, the first less or equal to the second, then the size of the interval may be huge and the possibility to get overlapping dates in a timeline increases quickly. In this case, generating a timeline becomes problematic, since it must not contain overlapping intervals. We therefore should put the generation of intervals and dates under control and steer it in the right direction. For example, if we only randomly pick the first date and then add a random (small) number of days to this date to generate an interval, then we may end up with timelines that contain far more intervals. But the possibility of negative testing, i.e., testing that overlapping intervals are rejected, decreases.

Using QuickCheck, one needs to control the randomness in the generators. The problem we address in this paper is *not* to come up with different generators that are better or worse for certain kind of testing, the problem is to know whether the generated test cases provide a good coverage of the things you want to test.

III. Application 1: booking web service

The booking web service is a tool used internally within a testing organisation at a telecommunication company to manage and enable efficient sharing of hardware equipment. Specialised hardware in the telecommunication industry is usually associated with profound costs, and efficient sharing of those resources is fundamental to achieve a cost-effective environment. It is also becoming more commonplace to configure larger networks of nodes. Those are setups that take longer time to install and configure, and re-using them between teams saves a considerable amount of time for testers.

The web service serves two main use cases with different needs: manual use of the test equipment and automatic use of the test equipment. In the first use case, engineers want to search the labs for hardware that they need for their tests. When they find what they need, they reserve the hardware for some days or a couple of weeks. Engineers can return to the service for additional information or to extend their bookings. The second use case is a fully automatic use of the test equipment during the continuous integration process. Every time a new software package is delivered, a few different sets of suites are run, organised as short, medium, and long term, each suite defined to run on a specific network setup. Every time this activity starts, the web service will be queried for available hardware of the proposed topology. Any network that contains this topology is accepted, and that network will be configured to disconnect the unwanted hardware. All networks not containing the proposed topology will not be considered. If there are no networks available that satisfy the request, then

[1]In the sense that it cannot shrink to a failing test with the shrinking algorithm used.

[2]We use QuickCheck from here on to denote this version

continuous integration will pause for some time and retry again some time later.

The web service is built in Erlang [13], based on a Mnesia database, and a YAWS front-end [14]. At the core of the application it uses a timeline data structure to manage bookings. This data structure represents a calendar in which certain intervals are blocked (the days that equipment is booked).

Keeping an efficient regression suite is important. Implementing a small booking system is a relatively small task compared to implementing other telecommunication software. It took less than five weeks to implement the first version of the functionality. However, as the system evolved over time, as is common in industry, it had to be adapted to new and changing requirements many times over. The first implementation, for example, was used with a single lab only, having users all at the same location. The system today has evolved and users can now be found in a handful of locations around the world. The software has had to retain its integrity over several adaptations like this.

Even though these small scale tools are not business critical, the consequences for an organisation can still be severe when they malfunction. In the case of the booking web service, it would have only modest effects on the engineers since their sole use of it is to find and book new equipment. The already booked nodes will not be affected, and thus tests can be carried on as normal. However, the global usage puts stress on the availability, and small errors at any time will most likely create annoyances or delays for someone, somewhere.

The effects are more severe for the automatic testing since it is continuously dependent on up to date information of hardware availability. It was decided that it should not occupy any resources when not running because the total number of hardware that would idle between runs would have an unacceptable impact on the lab size.

The web service shows a small and limited, but practical example of a software entity ubiquitously found in industry. These kind of systems are not business critical in themselves, but their cumulative effect on the business process as a whole is. Testing them sufficiently to ensure their quality will therefore be important to keep the bigger machinery running smoothly.

IV. Testing datatypes

The booking web application consists of a number of components, of which the implementation of the timeline datatype is central. This datatype is used to store, compare and remove time intervals for bookings. Since testing datatypes with QuickCheck is well documented [6], [7], we just need to follow the methodology described: create a generator using the constructors of the data type, create a model implementation of the datatype, and write one property for each operation on the datatype.

The timeline datatype represents a calendar in which certain intervals are blocked (the days that equipment is booked). A timeline can be constructed and manipulated using the following functions:

`new`	create an empty timeline,
`add`	add an interval to a timeline,
`delete`	delete a specific interval from a timeline,
`after_`	remove all intervals before a certain date from the timeline; used to prune old bookings,
`tail`	remove the first interval of a timeline.

According to [6] we should use all these operations in the timeline generator.

In addition to these operations, we also define functions to compare timelines, extract elements from a timeline, such as an particular interval, and to check whether two intervals overlap:

`equals`	check whether two timelines are equal,
`empty`	check whether a timeline is empty, i.e., does not contain any interval,
`member`	check whether a particular interval is already in the timeline,
`overlap`	check whether a given interval overlaps with any of the intervals in the timeline,
`valid`	check whether the intervals in a timeline are chronologically ordered and do not overlap,
`get_overlap`	return the first interval in the timeline that overlaps with a given interval,
`head`	return the first interval in a timeline,
`nth`	return the n^{th} interval in a timeline,
`overlap`	check whether two given intervals overlap.

Following the method mentioned earlier, we now need to create a generator for timeline data structures, and a model of a timeline that can be constructed with corresponding operations. We can then define a QuickCheck property for each operation, which applies the operation to a timeline and the corresponding operation to a model of that timeline, and validates if the resulting timeline conforms to the resulting model. For example, we define the following property for the `add` operation:

```
prop_add() ->
  ?FORALL(
    {I, SymT}, {interval(), timeline()},
    begin
      T = eval(SymT),
      case catch add(I, T) of
        {'EXIT', {overlap, _}} ->
          is_overlap(I, model(T));
        NewT ->
          equals(model(NewT),
                 model_add(I, model(T)))
      end
    end).
```

A random interval `I` and a timeline `SymT` are generated by the generators `interval()` and `timeline()` respectively. The timeline generator generates a symbolic timeline, that is, a value generated by this generator is a list of symbolic calls to constructor operations. Symbolic values allow us to inspect how an actual value is constructed. Whenever we need the actual value, we evaluate the symbolic value using the `eval` function. We also want to perform negative tests and check if a proper error message is produced. When an exception is raised, we validate if it is raised for the right for the right reason, in this case if the generated interval overlaps with the generated timeline. If no exception is raised, the model of the newly obtained timeline should be equal to the model of the generated timeline to which we add the interval via the corresponding model operation.

The model and properties are easy to come up with following the aforementioned method, but the tricky parts are the generators for intervals and timelines. Decimal numbers [6] and ordered sets [7] can be generated from a simple recursive generator or grammar description, since there is little dependency between values generated in different recursive calls. However, in our case we have an invariant on the generated timeline, namely that intervals should be non-overlapping. This makes the data generation severely more difficult. The first contribution of this paper is *a method to evaluate the data distribution* for datatype generators that need to meet some requirements. The second contribution is *a timeline generator that performs well* with this evaluation.

A. Interval generator

We want to test the timeline functions on random input and need data generators for the arguments of those functions. Many functions take an interval as argument. We represent an interval as a pair of two triples: year, month and day. A naive approach would be to construct an interval using two (ordered) random generated dates. Using QuickCheck one would generate such a triple with the `choose` generator and use the `?SUCHTHAT` macro to filter dates that the Erlang `calendar` module rejects as valid date.

```
ymd() ->
 {choose(2012, 2013),
  choose(1, 12),
  choose(1, 31)}.

date() ->
  ?SUCHTHAT(Date, ymd(),
            calendar:valid_date(Date)).
```

A tuple of two such dates, however, does not provide good test data. We want the test data to typically be a few days, preferably around week, and containing month and year transitions. For example, 2012-12-28 to 2013-1-1 would make for a nice test case. We should create a generator that chooses such intervals with reasonable likelihood. As noted before, an interval generator that picks the date purely randomly would create intervals that are very large. Limiting the year to be either 2012 or 2013 reduces the number of extremely large intervals, but at the same time, choosing more than 4 non-overlapping random intervals in that domain is unlikely to happen with the uniform distribution of `choose`. We therefore steer the generation to make it more likely to select intervals that we are interested in by adding a few days to the date and discard dates that therewith become invalid.

```
interval() ->
  ?LET(D1, date(),
       ?LET(D2, larger_date(D1),
            {D1, D2})).
```

```
larger_date({Y, M, D}) ->
  ?SUCHTHAT(
    frequency(
      [{9, ?LET(Days, nat(),
                shift({Y, M, D}, Days))},
       {1, Date, date()}])
    Date > {Y, M, D}).
```

After picking the first random date, the second date is constructed by adding an arbitrary number of days to the date. Alternatively, in 10 percent of the cases we also allow a completely random date as second alternative, provided it is larger than the first date.

This is one attempt to get a good distribution of intervals in a timeline. The question is, how good? And are there any obvious cases that we do not test with such a distribution or cases that are unlikely to be generated in a run of hundred tests?

V. Testing Generator Requirements

We would like to be able to assess the quality of a test data, in order to convince ourselves that a generator is good enough. To assess the quality of a test data we propose to define *requirements* on values produced by such a generator. A requirement for a generator is a property that should hold for a certain percentage of the generated tests. So, we can specify that a minimum (or maximum) number of generated test values should adhere to a given property. We have extended QuickCheck with the possibility to define such requirements on generators in a convenient way. For example, a requirement on a generator for natural numbers between 1 and 10, may be that it should generate a 1 within say 12 tests. We can express such a requirement as follows:

```
req_has_one() ->
  Gen = eqc_gen:choose(1, 10),
  ?REQ_EXISTS(1, Gen, 12).
```

The `req_has_one` function returns a QuickCheck property that we can test, just as any other ‘normal’ property, with the `quickcheck` function:

```
1> eqc:quickcheck(req_has_one()).
OK, passed
true
```

Not surprisingly the generator meets this requirement. In case a generator meets a requirement, QuickCheck prints an acknowledgement and returns the value `true`.

A slightly larger example is the following requirement:

```
req_half_is_larger_than_five() ->
  Gen = eqc_gen:choose(1, 10),
  ?REQ_MIN(X, Gen, X >= 5, 50.0, 100).
```

This requirement demands from the generator that at least 50% of the generated values are equal or larger than 5. Running `quickcheck` on this requirement results in the following output:

```
2> eqc:quickcheck(req_half_is_larger_than_five()).
Failed! Only 46 percent meets the condition.
```

```
[6,4,1,9,2,8,9,1,6,2,7,2,10,9,3,9,8,5,6,1,2,
 ...
 2,4,3,2,5,3,10,2,8,2,5,5,9,4,1,9,2,10,8,5,8]
false
```

QuickCheck reports that the generator `choose(1, 10)` did not meet the `req_half_is_larger_than_five()` requirement. It shows the generated test data, which can be regarded as a counterexample, and the percentage of the data that did meet the requirement. Since the `choose` generator has a linear distribution, it is possible that we generate 50 numbers that are smaller than 5. The counterexample allows us to inspect the generated data. Using this information we can improve the generator, or, if we are satisfied with the data distribution, we could weaken the requirement.

We offer the following macros to construct requirements on QuickCheck test data generators:

`?REQ_EXISTS(X, Gen, N)`,
: check if a generator `Gen` will at least generate a value equal to `X` within `N` number of tests,

`?REQ_EXISTS_FOR(X, Gen, P, N)`,
: check if a value for which predicate `P` (that takes a value as argument and returns a Boolean value) holds, is generated within `N` tests,

`?REQ_BETWEEN(X, Gen, P, Min, Max, N)`,
: check if the percentage of the values for which `P` holds lies between `Min| and \erlang|Max`,

`?REQ_MIN(X, Gen, P, Min, N)`,
: same as `?REQ_BETWEEN` but only with a lower bound,

`?REQ_MAX(X, Gen, P, Max, N)`,
: same as `?REQ_BETWEEN` but only with an upper bound,

`?REQ(X, Gen, P, C, N)`,
: the above requirement macros are expressed in terms of this is general macro, which generalises the condition check (which takes an percentage as argument and returns a Boolean value).

The last argument, which specifies the number of tests, of all macros can be left out. If the number of test is not specified we use the default of a hundred tests. We can check individual requirements with the `quickcheck` function. In addition, we provide a function, named `req_module`, which checks all requirements defined in a module. The name of a requirement needs to be prefixed with `req_`.

QuickCheck already offers the possibility to *measure* the probabilities of different kinds of test data. This can be done by instrumenting a QuickCheck property to collect statistics during testing. For example, we might instrument a property as follows, to measure how often a one is generated by the `choose(1,10)` generator:

```
prop_has_one() ->
  ?FORALL(N, eqc_gen:choose(1,10),
          collect(N == 1, N < 11)).
```

The effect of the line `collect(N == 1, ...)` is to collect the value of `N` in each test, and after testing is complete, to display the distribution of the values collected. In this case, testing the instrumented property yields:

```
3> eqc:quickcheck(prop_has_one()).
.................................................
OK, passed 100 tests

89% false
11% true
true
```

The collected statistics show that `N` was 1 in 11% of the generated tests. This is already valuable information. However, we cannot use the collected data to give a judgement, nor can we let the property succeed or fail based on these statistics. As a consequence, an expert must (re)examine the result in order to check if a generator meets its requirements. Using the requirement macros defined above, we can. Note that the requirement functionality is not meant to replace the statistics collection functionality. Both are useful in their own right.

Interval generator: Let us now return to our running example. Using the above macros we can introduce some requirements on the interval generator, which we introduced in the previous section. For example, we can state that an arbitrarily generated list of intervals should consist of non-overlapping intervals in 75% of the cases:

```
req_non_overlap() ->
  ?REQ_MIN(Is, eqc_gen:list(interval()),
           non_overlapping_pair(Is),
           75.0).
```

The `non_overlapping_pair` function checks whether or not there is an overlap between one of the elements of `Is` with any of the other elements. When we check the requirement for the generator of intervals with two arbitrary dates (first smaller than the second), we get a requirement success rate of around 30%, thus in 70% of the generated lists of intervals, the lists contains overlapping intervals. Moreover, when we only generate lists containing five intervals, we seem to be unable to create any of these without an overlapping interval. However, for the smarter generator for intervals described above, we come close to a success rate of 80%.

A. Timeline generator

The problem of bad data distribution gets even more obvious if we follow the generator construction explained in literature, where we build a data structure by using the constructors defined by the datatype.

```
timeline() ->
  ?SIZED(Size, well_defined(timeline(Size))).

timeline(0) ->
  {call, ?API, new, []};
timeline(N) ->
  ?LAZY(oneof(
    [timeline(0),
     {call, ?API, add,    [interval(),
                           timeline(N-1)]},
     {call, ?API, tail,   [timeline(N-1)]},
     {call, ?API, delete, [interval(),
                           timeline(N-1)]}
    ])).
```

This generator creates an arbitrary timeline by recursively adding and deleting intervals from a previously defined timeline. We do this symbolically, which means that we build a data structure containing the calls to the API instead of calling the API directly. But this timeline generator does a very poor job. By deleting an arbitrary interval from the timeline it is most often the case that this interval is not present in this timeline. The software under test will in such case raise an exception. Exceptions are handled in the function `well_defined`, which takes a generator as input, and recomputes it if exceptions are raised under evaluation. When sampling this data generator, about 70% of all values created is the empty timeline, followed by timelines with one or at most two intervals in it. That does not make for good test data.

Lets improve the `timeline` generator. However, before doing so, we want to formulate requirements on the timeline generator we are trying to construct. A requirement that 2% of the test cases should have a symbolic timeline that after evaluation contains more than 10 intervals would be specified as follows:

```
req_length() ->
  ?REQ_MIN(SymT, timeline(),
           length(eval(SymT)) > 10, 2.0).
```

A requirement that any set of generated values should have at least one timeline with an interval that spans over a year border is specified as follows:

```
req_year_span() ->
  ?REQ_EXISTS_FOR(SymT, timeline(),
    lists:any(fun({{Y1, _, _}, {Y2, _, _}}) ->
                Y1 < Y2
              end, eval(SymT))).
```

Similarly, a requirement that checks if an interval is present that spans over a month, is defined as follows:

```
req_month_span() ->
  ?REQ_EXISTS_FOR(SymT, timeline(),
    [1 || {{_, M1, _}, {_, M2, _}} <- eval(SymT),
          M1 < M2] =/= []).
```

We have specified additional requirements on the timeline generator, but we omit the definition.

The symbolic representation of calls helps us to define requirements on the construction of timelines. Since the data generator has a structure in which we save which calls we apply instead of the final result, we can express a requirement that 10% of the generates timelines should have been build with both a `delete` and a `tail` in its construction. With those requirements and the above generator for timelines, we get the following result:

```
4> eqc_requirements:req_module(booking_eqc).
Failed! After 1 tests.
Requirement req_length failed:
  only 0.00% meets the condition.

Failed! After 1 tests.
Requirement req_consecutive failed:
  only 0.00% meets the condition.

Failed! After 1 tests.
Requirement req_mix1 failed:
  only 0.00% meets the condition.

Failed! After 1 tests.
Requirement req_year_span failed:
  only 0.00% meets the condition.

OK, passed 1 tests

false
```

The failure rate is 100% for all but the requirement that we should have an interval over the month border. This means that none of the generated values fulfils any of the other requirements.

By selecting existing intervals from the earlier generated timeline and only taking the tail from a timeline that contains at least one interval we can do much better. The improved generator is defined as follows:

```
timeline() ->
  ?SIZED(Size, well_defined(timeline(Size))).

timeline(0) ->
  {call, ?API, new, []};
timeline(N) ->
  ?LAZY(
  ?LETSHRINK(
    [SymT],
    [well_defined(timeline(N-1))],
    begin
      T = eval(SymT),
      frequency(
        [{50,{call, ?API, add, [interval(),SymT]}},
         {1, {call, ?API, after_, [date(),SymT]}}]
        ++
        [{5, {call, ?API, tail, [SymT]}}
          || T =/= []]
        ++
        [{5, {call, ?API, delete,
          [elements(T), SymT]}} || T =/= []])
    end)).
```

This generator performs much better and passes all requirements with good margins.

Specifying requirements provides the developers the tools needed to ensure that data generators meet the expectations on test cases that they would use in manually written unit tests. In a similar way as deciding which unit tests one should write, we now decide which particular data distributions provide valuable test data. After that, we specify one property per operation and check the result against a model. In this way, we do get the complete testing as described in literature plus an additional quality assurance on the generated test data. In practice, this has helped us to motivate the designers of the generators to realise the short-comings of early versions of the generators and to improve them iteratively.

VI. Application 2: Dudle

Dudle is an open source web service, which can be used to schedule a meeting or poll people for an opinion. It is a relatively small web service with a simple and well defined interface. In case of a schedule, users can vote for one or more time slots, and in case of a poll, users can choose several options. Dudle has functionality for creating, deleting, editing a schedule or a poll. Participants can be invited via Dudle to take part in a schedule or a poll. And finally, the administrator of a schedule or poll can review the status in order to see which alternatives are preferred by the participants. Dudle is written in the programming language Ruby and can be deployed using a web server, such as Apache, via a common gateway interface (CGI).

We have tested the Dudle web service with QuickCheck, using the abstract state machine functionality. We maintain a model of the Dudle system while executing test commands, which are mapped to CGI-calls, and checking pre- and postconditions. We have developed a number of test data generators for this test, such as generators for a poll name, or a time slot. In this section we focus on a test data generator for a user name. We started out with a textbook case of a generator for random user names:

```
name() ->
  ?SUCHTHAT(
    Name,
    eqc_gen:non_empty(eqc_gen:list(eqc_gen:char())),
    not lists:member($\r, Name)).
```

We have constructed this generator in terms off standard QuickCheck generators. The `name()` generator produces a non-empty list of characters. We use the `?SUCHTHAT` macro to exclude user names containing carriage returns. To ensure that we pick equal names now and again we do not use this generator directly, but we use it to create a pool of names from which we choose.

We had to improve the user name generator, such that it generates more realistic user names. Dudle was not always able to handle peculiar user names, for example names containing newlines and spaces. We do not blame Dudle for this, instead, we blame our slightly naive generator. The improved version of the user name generator is defined as follows (where Erlang's notation for a character is preceded by a dollar sign):

```
name2() ->
  Gen = frequency([{100, choose($a, $z)},
                   {25,  choose($A, $Z)},
                   {25,  choose($0, $9)},
                   {5, $ },
                   {1, $-}, {1, $_}]),
  ?LET(Name,
       eqc_gen:non_empty(eqc_gen:list(Gen)),
       string:strip(Name)).
```

This generator also produces a non-empty list of characters, but the characters are selected more carefully. Instead of choosing random characters we now choose alpha-numeric characters and occasionally a slightly unusual character, such as a space or a dash.

We have used the above generator in testing Dudle and are quite satisfied with it. But does it actually produce the user names that we expect? That is, does it meet our implicit requirements? Lets find out and make these requirements explicit, and specify them using the macros from Sect. V.

We had the following implicit requirements in mind when we defined the user name generator:

1) at least 10% of the generated user names should contain an unusual characters, such as a dash,
2) we should not generate names with more than four spaces,
3) a quarter of the generated user names should be longer than 8 characters,
4) we want to generate user names containing both upper and lower case characters.

These implicit requirements can be translated to formal requirement using the requirement macros as follows:

```
req_unusual() ->
  Intersect =
    fun(Xs, Ys) ->
      [X || X <- Xs, lists:member(X, Ys)]
    end,
  ?REQ_MIN(Name, name2(),
           length(Intersect(Name, "-_ ")) > 0,
           10.0).

req_spaces() ->
  Spaces =
    fun(Xs) ->
      lists:filter(fun(X) -> X == 32 end, Xs)
    end,
  ?REQ_EXISTS_FOR(Name, name2(),
                  length(Spaces(Name)) < 4).

req_name_length() ->
  ?REQ_MIN(Name, name2(),
           length(Name) > 8, 25.0).

req_upper_lower_case() ->
  IsUpper =
    fun(X) ->
      X >= $A andalso X =< $Z
    end,
  IsLower =
    fun(X) ->
      X >= $a andalso X =< $z
    end,
  HasUpperAndLower =
    fun(Xs) ->
      length([X || X <- Xs, IsUpper(X)]) > 0
      andalso
      length([X || X <- Xs, IsLower(X)]) > 0
    end,
  ?REQ_EXISTS_FOR(Name, name2(),
                  HasUpperAndLower(Name)).
```

We have defined these requirements in the Dudle test module named `dudle_eqc`. We use the `req_module` function to test all requirements defined in the Dudle test module, which generates the following output:

```
5> eqc_requirements:req_module(dudle_eqc).
OK, passed 1 tests

Failed! After 1 tests.
Requirement req_unusual failed:
  only 7.00% meets the condition.

OK, passed 1 tests

Failed! After 1 tests.
Requirement req_name_length failed:
  only 8.00% meets the condition.

false
```

These results show that the `name2` generator does not meet two of the four requirements, namely `req_unusual` and `req_name_length`. The latter suggests that the length of the generated user names are too short. This may explain why the requirement `req_unusual` fails as well, since the unusual characters have a low probability of being generated. We adapt the user name generator such that it generates longer names:

```
name3() ->
  Gen = frequency([{100, choose($a, $z)},
                   {25,  choose($A, $Z)},
                   {25,  choose($0, $9)},
                   {5, $ },
                   {1, $-}, {1, $_}]),
  ?LET(Name,
       eqc_gen:non_empty(eqc_gen:longlist(Gen)),
       string:strip(Name)).

longlist(Gen) ->
  ?SIZED(Size,
         resize(Size*2, list(resize(Size, Gen)))).
```

Most of the generator is left as is, but we have replaced the standard `list` generator with our own `longlist` generator. The `longlist` generator produces lists that are double the size of lists generated by the `list` generator. Lets check the requirements again:

```
6> eqc_requirements:req_module(dudle_eqc).
OK, passed 1 tests

OK, passed 1 tests

OK, passed 1 tests

OK, passed 1 tests
true
```

The `name3` generator meets all the requirements. This example shows that testing requirements supports the development of good test data generators. Not only does testing requirements have added value for validating large complex generators, but also for simple straightforward generators, such as the user name generator. It is all too easy to overlook something, such as generating list of the proper length.

VII. CONCLUSIONS

From experience, strengthened by a scientific experiment [15], we know that it is difficult to write test cases that cover a good set of input data, both positive and negative data. Random generation of data makes testing immune to specific choices, but also introduces the possibility to generate data that does not cover border cases or specific inputs.

When QuickCheck data generators get more complicated to write and their distributions harder to grasp, one can get a false sense of trust by seeing many test cases pass. The actual generated data can be collected by built-in QuickCheck functions and printed as side-effect of testing. However, only presenting the data requires either domain experts to asses the statistics or forces engineers to subjectively judge whether the collected values are satisfactory.

In this article we contribute by showing how one can express and verify requirements on generators to convince oneself that the performed testing is sufficient. Interaction with domain experts is needed at the beginning of the test design, when the requirements on test data are stated. With data from testing two different web services, we have shown that with a naive approach to random data generation, we can easily produce test cases without the required quality. We have shown how we can make the quality requirements explicit and automatically verifiable. And finally, we have shown how to control the randomness so that the test cases we produce are of the required quality.

Mutation testing is a different way of judging the quality of a test suite. This is based upon introducing errors in the software under test and trying to find them by running the test suite. Mutation testing is a fundamentally different technique and requires code instrumentation with good mutants. It is further research how these techniques complement each other.

With the techniques presented in this paper, domain and test experts are able to write requirements to ensure that the tests they perform are of high quality. It allows for a high degree of automation by minimal intervention of domain experts and automatic feedback on the quality of the generated data.

REFERENCES

[1] J. Derrick, N. Walkinshaw, T. Arts, C. B. Earle, F. Cesarini, L.-Å. Fredlund, V. M. Gulías, J. Hughes, and S. J. Thompson, "Property-based testing - the protest project," in *FMCO*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, Eds., vol. 6286. Springer, 2009, pp. 250–271.

[2] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs." in *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2000, pp. 268–279.

[3] T. Arts, J. Hughes, J. Johansson, and U. T. Wiger, "Testing telecoms software with Quviq QuickCheck," in *Erlang Workshop*, M. Feeley and P. W. Trinder, Eds. ACM, 2006, pp. 2–10.

[4] A. Nilsson, L. M. Castro, S. Rivas, and T. Arts, "Assessing the effects of introducing a new software development process: a methodological description," *Int. J. on Software Tools for Technology Transfer*, pp. 1–16, 2013.

[5] "Property-based testing of web services," http://www.prowess-project.eu, 2012-2015.

[6] T. Arts, L. M. Castro, and J. Hughes, "Testing erlang data types with Quviq QuickCheck," in *Proceedings of the ACM SIGPLAN Workshop on Erlang*. ACM Press, 2008, pp. 1–8.

[7] T. Arts and L. M. Castro, "Model-based testing of data types with side effects," in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, ser. Erlang '11. New York, NY, USA: ACM, 2011, pp. 30–38.

[8] "Dudle," https://dudle.inf.tu-dresden.de, 2013.

[9] C. Soldani, "QuickCheck++," http://software.legiasoft.com/quickcheck/, 2010.

[10] T. Jung, "Java implementation of QuickCheck," http://quickcheck.dev.java.net/, 2010.

[11] C. League, "Qcheck/sml," http://contrapunctus.net/league/haques/qcheck/, 2010.

[12] J. Hughes, "Quickcheck testing for fun and profit," in *Practical Aspects of Declarative Languages*, ser. Lecture Notes in Computer Science, M. Hanus, Ed. Springer Berlin Heidelberg, 2007, vol. 4354, pp. 1–32.

[13] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[14] Z. Kessin, *Building Web Applications with Erlang: Working with REST and Web Sockets on Yaws*. O'Reilly, 2012.

[15] S. Eldh, H. Hansson, and S. Punnekkat, "Analysis of mistakes as a method to improve test case design," in *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '11, 2011, pp. 70–79.