Re-using Generators of Complex Test Data

Simon Poulding
Software Engineering Research Lab (SERL Sweden),
Blekinge Institute of Technology, Sweden
Email: simon.poulding@bth.se

Robert Feldt
Software Engineering Research Lab (SERL Sweden),
Blekinge Institute of Technology, Sweden
Email: robert.feldt@bth.se

***Abstract*—The efficiency of random testing can be improved by sampling test inputs using a generating program that incorporates knowledge about the types of input most likely to detect faults in the software-under-test (SUT). But when the input of the SUT is a complex data type—such as a domain-specific string, array, record, tree, or graph—creating such a generator may be time-consuming and may require the tester to have substantial prior experience of the domain.**

In this paper we propose the re-use of generators created for one SUT on other SUTs that take the same complex data type as input. The re-use of a generator in this way would have little overhead, and we hypothesise that the re-used generator will typically be as least as efficient as the most straightforward form of random testing: sampling test inputs from the uniform distribution.

We investigate this proposal for two data types using five generators. We assess test efficiency against seven real-world SUTs, and in terms of both structural coverage and the detection of seeded faults. The results support the re-use of generators for complex data types, and suggest that if a library of generators is to be maintained for this purpose, it is possible to extend library generators to accommodate the specific testing requirements of newly-encountered SUTs.

I. Introduction

The strategy of random testing is to sample test inputs from the domain of the software-under-test (SUT) according to a chosen probability distribution. The key advantage of this strategy is that it is relatively cheap to derive a set of test inputs compared to, for example, white-box generation strategies that undertake a static or dynamic analysis of the SUT's source code: such an analysis can be costly especially when it is performed manually.

When the input domain of the SUT is a primitive numeric data type, such as an integer or floating point type, sampling random test inputs is particularly straightforward. A default strategy is to sample from a uniform distribution over the interval that defines the input domain, and many programming languages provide standard pseudo-random number generator functions that can be used for this purpose.

However many SUTs take more complex data types that are a composition of primitive types; examples include dates, strings, arrays, records, trees, and graphs. A valid instance of such a data type must typically satisfy a number of constraints. For example, the constituent day, month, and year in a valid date instance must satisfy not only constaints on their individual values (e.g. day must be an integer between 1 and 31), but also constraints involving one another (e.g. day must be no more than 30 when month is 11). Similarly, a valid instance of a proper binary tree must satisfy the constraint that each non-leaf node has 2 children.

When applying random testing to complex data types such as these, often one cannot simply sample each constituent primitive value independently: the constraints on a valid instance are, in general, unlikely to be satisfied. Instead, more sophisticated generation approaches are required that guarantee validity of the instance by construction. One common approach is to use a formal grammar to define the construction of an instance of the type as a series of production rules [1], [2]. An extension to this approach is to use a generating program written in a high-level language; examples include QuickCheck [3] and UDITA [4]. Generating programs can provide more flexibility than a formal grammar; for example, by storing state in one part of the program and retrieving it in another. It is use of programs for generating test data that we consider in this paper.

In order to generate different instances each time it is run, a generating program must incorporate some degree of choice as the execution path it takes, or the values it assigns to variables. By changing how these choices are made stochastically—for example, favouring one execution path over another—the probability distribution from which generated instances are sampled may be controlled. We will use the term *generator* to refer to the *combination* of the generating code and specific settings for its stochastic choices: a particular generator will therefore not only emit valid instances, but sample them according to a specific probability distribution.

In the absence of any guiding information, e.g. from the SUT's source code or specification, a sensible initial choice for a generator is one that emits an instance according to a uniform distribution. But the uniform distribution—whether over a primitive or complex data type—is unlikely to be the most efficient probability distribution, where efficiency is the size of the randomly-sampled test set required to detect faults in the SUT. Since each test case adds to the cost of testing—through executing the test case itself as well as the effort required in checking the test output for evidence of a fault—efficient distributions reduce the cost of testing.

For a specific SUT, it is often possible to derive a much more efficient probability distribution than the uniform distribution by using additional information from the SUT itself (in a similar manner to white-box testing) or using the expertise of the tester. In previous work we have demonstrated automated search-based approaches that optimise generators in this way [5], [6]. However, the need for prior expertise when constructing a generator, and the overhead of search-based optimisation, could negate the major advantage of random testing as a strategy: that of straightforward, low-cost test data generation.

We propose instead that it may be possible to re-use an efficient generator created for one SUT on other SUTs that take inputs of the same complex data type. The re-used generator is unlikely to be as efficient as a generator optimised specifically for the new SUT, but we hypothesise that it will be more efficient than the uniform distribution in many cases, while avoiding the cost of creating a generator specifically for the new SUT. If so, it would be cost-effective to maintain a library of generators for complex data types.

It is this proposal we investigate in this paper by empirically testing the underlying hypothesis: that there is sufficient commonality in the way in which the same complex data type is processed by different SUTs that a test data generator that is efficient for one SUT will often be efficient when testing other SUTs.

The paper is organised as follows. Section II introduces GödelTest, the framework used throughout the paper to define and optimise generators for complex data types. In Section III we expand on the proposal for re-using generators between SUTs and define four research questions that guide the empirical investigation of Section IV. Section V discusses the relationship of this paper to related work, and in Section VI we summarise our conclusions and outline future work.

II. The GödelTest Framework

The proposal for generator re-use made in the introduction is independent of the method used to implement the generator. However empirical work to assess this proposal must choose a specific implementation, and so in this paper we implement generators using GödelTest, a framework to faciliate writing and optimising generator programs that we have demonstrated in earlier work [5]. In this section we provide an overview of GödelTest both as background to the empirical work and as a concrete example of generators for complex data types.

A. Generating Program

A GödelTest generator has two components: a *generating program*, and a *choice model*. The generating program is code, written in a general-purpose language, that defines how to construct an instance of the data type. The choice model (described below) controls how stochastic choices are made in the generating program.

In this paper we use the GödelTest framework for Julia, a high-level programming language for technical computing that has performance comparable to native C code [7]. While it is most straightforward to pass test data generated by GödelTest to SUTs implemented in Julia itself (or in languages that Julia can directly interface with, such as C/C++), the test data can nevertheless be used to test SUTs implemented in other languages by saving the test data to a file, or passing it over a network connection.

Figure 1 is an example of a GödelTest generating program that emits simple arithmetic expressions such as: `-4+(99/(2*5))`.

The `@generator` keyword introduces a generating program which consists of a number of methods in a `begin...end` block. Each method can make use of the Julia standard library. For example, this generator uses `*`, `string()`, and `join()`, that are the standard library functions for string concatenation, conversion of a value to a string, and concatenation of an array of strings, respectively. Although this particular generator uses fairly simple methods, the complexity of the method code is not restricted by GödelTest and so permits much greater flexibility and compactness than, for example, a formal production grammar. The methods typically call one another and may do so recursively; in this example the mutually recursive calls between the `expression` and `operand` methods permit the generation of expressions with any level of nesting of parenthesised expressions. The generator includes a `start` method that is called to initiate the generation of an instance of the data type.

```
@generator ExprGen begin
  start = expression()
  expression = operand() * operator() * operand()
  operand = "(" * expression() * ")"
  operand = begin
    number = join(plus(:digit))
    if choose(Bool)
      number = "-" * number
    end
    number
  end
  digit = string(choose(Int,0,9))
  operator = "+"
  operator = "-"
  operator = "/"
  operator = "*"
end
```

Fig. 1. Example of a GödelTest generator for simple arithmetic expressions

In addition, the generating program will normally include one or more GödelTest-specific constructs that identify locations, called *choice points*, where choices may be made stochastically when the program is executed. There are three classes of choice point:

1) Value Choice Points: The `choose(Type,...)` constructs returns a random value of built-in Julia data types such as Bool, Int, and Float64. Additional parameters to `choose` may be used to restrict the sampled values to a numeric range. Examples of this class are the construct `choose(Int,0,9)` (in the `digit` method) that returns an integer between 0 and 9; and `choose(Bool)` (in the second `operand` method) that returns either `true` or `false`.

2) Sequence Choice Points: The construct `reps(:method,min,max)` builds a random-length sequence of values created by repeated calls to the specified method; the sequence is returned as a Julia array. The parameters specify the minimum and maximum length of the sequence, the latter of which may be infinite. The constructs `mult(:method)` and `plus(:method)` are syntatic sugar for `reps(:method,0,Inf)` and `reps(:method,1,Inf)`, and thereby specify a sequence of length 0 or more, and of length 1 or more respectively. An example of this class is the construct `plus(:digit)` (in the second `operand` method) that returns a sequence of digits by repeated calls to the `digit` method.

3) Rule Choice Points: When more than one method has the same name, an implicit 'rule' choice point occurs. Each time the method name is called elsewhere in the generator, one of the methods (or 'rules') is chosen at random to be executed. In the example generator there are four methods named `operator`; when `operator()` is called in the `expression` method, one of the four methods is chosen at random to be executed, and a new random choice is made each time the method is called.

B. Choice Model

GödelTest uses a choice model to specify which choice is made each time a choice point is encountered when a generating program is run, and thus which of the many possible instances of the data type is emitted by the generator.

By design, the choice model is abstracted away from the generating program itself. Each choice that can be made at a choice point is uniquely identified by a number—an integer when there is a countable number of choices, or a real number for the `choose(Float64)` construct—that is called a Gödel number. Thus the choice model need only supply a sequence of Gödel numbers in order to define the stochastic choices the generating program should make when it is executed. This relationship between choice model and generating program is illustrated in Figure 2.

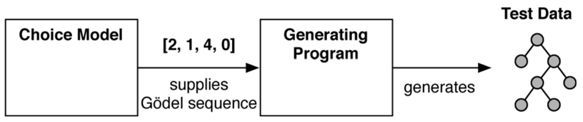


Fig. 2. The components of the GödelTest framework.

The abstraction of the choice model from the generating program has a number of advantages. For the same generating program, it is easy to apply different choice models since the interface between the two is always a Gödel sequence. Similarly, the same type of choice model may be applied to different generating programs. In our previous work we have shown that this separation facilitates the application of both metaheuristic optimisation [5] and Monte-Carlo tree search [6] algorithms to the generation of effective test data: by applying these algorithms to the choice model rather than generating program directly, the details of the generation process that are not relevant to the algorithm are avoided.

C. Sampler Choice Model

In this paper we use a *sampler choice model*. This is a simple stochastic choice model that defines a local probability distribution for each choice point in the generating program, and samples from that distribution each time a Gödel number is required for the choice point. Since the choice points control the specific instance of the data type emitted by the generating program, the set of local distributions in the sampler choice model define the global probability distribution over all the instances of the data type from which this generator samples. GödelTest can utilize much more sophisticated models, but the sampler choice model is a natural default that is sufficiently flexible for the empirical investigation in this paper.

The local probability distribution assigned to each choice point by the sampler choice model are listed in Table I. The general distribution family assigned to each choice point is fixed, but some distributions have one or more parameters that control the exact form of the local probability distribution. For example, the Geometric distribution assigned to sequence choice points has a parameter, p, that favours short sequence when close to 1, and long sequences when close to 0. It is through these parameters that the local distributions, and thus the global probability distribution of the generator as a whole, are modified. When initially defined, the sampler choice model assigns sensible default values to each parameter. For example, the parameter to the Bernoulli distribution that is assigned to `choose(Bool)` choice points is by default set to 0.5 so that `true` and `false` have the same chances of being sampled.

TABLE I. LOCAL PROBABILITY DISTRIBUTIONS ASSIGNED BY THE SAMPLER CHOICE MODEL.

Choice Point	Local Distribution	Parameter(s)	Default
`choose(Bool)`	Bernoulli	p	$p = 0.5$
`choose(Int)`	Discrete Uniform	—	—
`choose(Float64)`	Continuous Uniform	—	—
`reps`, `mult`, `plus`	Geometric	p	$p = 0.5$
rule choice (n methods)	Categorical	$w_1, \ldots, w_n$	$w_i = 1/n$

D. Generator Optimisation

The global probability distribution from which the generator samples instances is entirely determined by the choice model, which—for a sampler choice model—is determined by the parameters of the local probability distributions in the model. Thus to optimise the global probability distribution of the generator, the optimisation algorithm can be applied to these distribution parameters.

As an example, consider again the generating program of Figure 1. The choice points that have distribution parameters are those associated with `plus` (1 parameter); `choose(Bool)` (1 parameter); the rule choice implied by the 2 `operand` methods (2 parameters); and the rule choice implied by 4 `operator` methods (4 parameters). The global probability distribution of this generator is thus determined by a vector of 8 real-valued parameters, and this vector is the target of the metaheuristic search that optimises the distribution.

III. Generator Libraries and Re-use

Our key hypothesis is that an optimised generator of complex data re-used between SUTs is more efficient (in terms of test size) than uniform random testing. We propose two sets of research questions to investigate this hypothesis: the first set determine the efficiency of a generator optimised to a single SUT compared to uniform random testing; the second assess the relative efficiency of an optimised generator when it is re-used between SUTs rather than optimised to a single SUT.

A. Optimised Generators Compared to Uniform Random

A generator may be optimised to a SUT in two ways. The first method is encapsulate domain knowledge from the SUT's specification or implementation, or from the tester's own experience, in the generating program itself. For example, the generator may include code to emit a small subset of instances of the data type that the tester believes will be particularly effective at detecting faults; if this code is guarded by a `choose(Bool)` construct in the generating program, then the default sampler choice model will cause the generator to emit these instances 50% of the time. This motivates the research question:

RQ1—How much more efficient are generators for complex data types that incorporate domain knowledge than random testing using the uniform distribution?

The second method is to optimise the generator's choice model using metaheuristic search as described in Section II. This motivates the question:

RQ2—To what extent can the choice model be optimised to the specific SUT in order to improve the generator's efficiency?

The primary objective of these two research questions is to quantify the efficiency that can be achieved by optimising a generator *to a single SUT*, rather than specifically to show that optimised generators *can* improve testing efficiency compared to uniform random testing. Instead, by measuring the improvement of generator optimised for a single SUT compated to uniform random testing, we can use this improvement as a baseline against which to assess the efficiency of a generator re-used *between* SUTs.

B. A Library of Generators Re-used Between SUTs

If our hypothesis of re-using generators holds true, it would be possible to maintain a library of generators for commonly-used complex data types. We propose that a convenient method of representing a generator in such a library is as an *ensemble* of sub-generators, where each sub-generator has individually proven effective for one or more SUTs.

In GödelTest such an ensemble can be represented using a generator of the form shown in Figure 3: the two individual generators are passed to this ensemble generatorsas the parameters `g1` and `g2`, and the rule choice point implied by the two `start` methods samples one of the sub-generators each time the ensemble is executed.

```
@generator Ensemble(g1, g2) begin
  start = g1()
  start = g2()
end
```

Fig. 3. An ensemble generator taking two sub-generators

The choice model for the ensemble is formed by combining the choice models of each sub-generator and adding a local probability distribution for the rule choice point in the top-level generator. When maintaining a library of generators for complex data types, this ensemble choice model may be optimised for testing efficiency over a representative portfolio of SUTs. This motivates the third research question:

RQ3—Can an ensemble generator, optimised for efficiency against a portfolio of existing SUTs, also be efficient for the testing of new SUTs that were not part of the portfolio?

Of course, an ensemble generator will not be the most effective generator for *all* SUTs taking the complex data type as an input: it will always be possible to construct a SUT for which the ensemble is less effective than the uniform distribution. If it is found that the ensemble generator in the library is inefficient for a particular SUT or subclass of SUTs, then it would always be possible to extend the ensemble using a new sub-generator created specifically for this subclass. This gives rise to the fourth research question:

RQ4—What is the effect, in terms of efficiency, of extending an ensemble with additional SUT-specific sub-generators?

IV. Empirical Investigation

In this section, we describe three experiments that investigate the research questions defined above.

A. Data Types and SUTs

The experiments investigate two complex data types: Georgian dates and regular expressions (regexes). Testing efficiency is evaluated using a total of seven real-world SUTs: four taking the date type and three taking the regex type. The relevant characteristics of the SUTs are summarized in Table II, and for brevity we will refer to the SUTs using the abbrevations defined in this table.

The last column in the table gives the number of conditions to be covered for each SUT since, as discussed below, condition coverage will be used as part of the measurement of generator efficiency. Conditions in both predicates and assignments are counted, and the total includes conditions in any 'helper' functions called by the SUT and are part of the same package or class. The number of conditions also provides an indication of the complexity of each SUT.

We apply generators that emit dates as the triple `(year,month,day)` and convert this to a SUT-specific representation where necessary. All four SUTs taking date input contain little or no code to check the validity of the date input—they implicitly assume its validity has been checked beforehand—therefore for these experiments, we restrict the input domain to valid dates; moreover we arbitarily restrict the domain to dates with a year between 1600 and 2099 to create a bounded domain.

We apply generators that emit regexes as ASCII character strings. All three SUTs parse the regex (REP additionally performs a match-and-replace using the regex) and so contain code to check the input's validity. Therefore, in contrast to the date type, we do *not* restrict the input domain to valid regexes. The input domain is thus any ASCII string of length zero or more, and since the length is unbound, the domain has infinite cardinality. Both BBRP and GTRP use standard POSIX regex syntax, while REP uses non-standard metacharacters; we choose to create generators that emit POSIX regexes and perform a simple translation of metacharacters before the test inputs are supplied to REP.

To improve generalisability of the results, the data types and SUTs have been chosen to exhibit some diversity in their important characteristics. The date type has a fixed structure and its domain has finite cardinality, while the regex type can vary in size and has infinite cardinality. The SUTs are implemented in three different languages, are from five different libraries, and demonstrate a range of complexities.

B. Measuring Generator Efficiency

We define the efficiency of a generator as the average number of test inputs that must be randomly sampled from it

TABLE II. RELEVANT CHARACTERISTICS OF THE SUTS USED IN THE EMPIRICAL INVESTIGATION.

Abbreviation	SUT Name	Provenance	Language	Number of Conditions
DOY	day_of_year	Julia Dates package	Julia	10
DOWIM	days_of_week_in_month	Julia Dates package	Julia	22
EOMD	end_of_month_day	Boost 1.56 Date_Time library	C++	14
WN	week_number	Boost 1.56 Date_Time library	C++	20
BBRP	basic_regex_parser	Boost 1.56 Regex library	C++	544
REP	replace	Software-artifact Infrastructure Repository	C	168
GTRP	regex_parse	`GödelTest.jl` package	Julia	118

in order to detect a given set of faults in the SUT. We might, therefore, measure generator efficiency in these experiments by seeding faults in the SUTs and assessing the number of test inputs that must be randomly sampled for the generator to detect all the seeded faults. However seeding faults in an objective and equivalent way across all the SUTs would be difficult owing to their differing provenances and the variety of languages used to implement them.

Instead we use condition coverage as a proxy for the fault-detecting ability of a generator. While structural coverage is not necessarily a good indicator of fault-detecting ability, the advantage is that instrumenting the SUTs for condition coverage is a more objective process than seeding faults. Condition coverage is chosen because much of important functionality of the SUTs operating on the date type is encoded in conditions in both predicates and assignment statements, and would not be fully exercised by, for example, branch coverage.

The Software-artifact Infrastructure Repository provides a set of variants with seeded faults for REP. For this SUT, we therefore measure and report testing efficiency using *both* condition coverage *and* seeded faults. This provides a check on our premise here that condition coverage is a suitable proxy for fault-detecting ability.

C. Optimising Generators

In experiments that consider generators optimised to a SUT or portfolio of SUTs, we apply metaheuristic search to parameters of the choice model as described in Section II-D. For this purpose, we use Differential Evolution [9] as implemented in the `BlackBoxOptim.jl` Julia package by one of the authors; our earlier work [5] provides more details of this approach. The optimisation process is limited to 50 evaluations of the fitness metric. For the SUTs used in this work, this number of evaluations limits the resources used for optimisation to a few minutes of processing time on a typical CPU core, and was found to be sufficient during preliminary experiments.

To assess the fitness of a candidate generator during the optimisation process, a test set of size M is created by sampling M test inputs from the generator. We use $M = 50$ for date generators, and a larger value, $M = 100$, for the regex generators since the SUTs taking this data type generally require more test cases to achieve good coverage. A version of the SUT instrumented for condition coverage is then executed with each test input from the set in turn. After each input, the proportion of the conditions in the SUT that have been exercised by test inputs so far, i.e. up to and including the most recent test input, is calculated. This gives a prediction of coverage that would be achieved by test sets of all sizes between 1 and M, and may be visualised by a graph of the form shown in Figure 4.

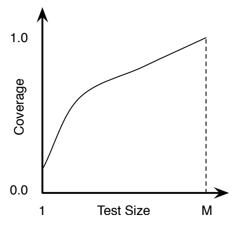


Fig. 4. An example of a coverage plotted against test size.

A more efficient generator will achieve a high coverage and/or will achieve its highest coverage at small test sizes. These desirable qualities may be quantified using the following metric:

$$f_{\text{eff}} = \sum_{i=1}^{M} c_i \qquad (1)$$

where c_i is the coverage as a proportion between 0 and 1, measured for a test set consisting of the first i test inputs, and M the maximum test set size. This metric measures the area under the curve in the graph of coverage plotted against test size: the region bounded by the axes, curve, and the dotted line in Figure 4.

Since the generators are stochastic, the fitness metric is calculated by taking the mean value of f_{eff} over 20 different tests sets sampled from the candidate generator. When the generator is optimised for more than one SUT, f_{eff} is calculated for each SUT, and the mean value over the SUTs is used to calculate the fitness. A more efficient generator will have a larger area under the curve, and therefore a larger value of f_{eff}, and so during optimisation the objective is to maximise this fitness.

D. Data Type Generators

In the first two experiments we consider two generators for dates: `UniformDate` and `LeapYearDate`; and four for regexes: `Uniform8Regex`, `Uniform16Regex`, `Uniform32Regex` and `PosixRegex`. The generators are listed in appendices A and B repectively.

The `UniformDate` generator is uniform in terms of months and years across the input domain—each combination of year and month is sampled with the same frequency—and near-uniform in terms of the full combination of year, month, and day. We argue that achieving precise uniformity to the level of day would require a more complex generator that would not be realistic for an tester to create for purpose of exploratory uniform random testing.

The `LeapYearDate` generator emits dates in leap years, and the generator code identifies three classes of these dates that domain knowledge might suggest are effective when testing for faults: years that are a multiple of 400, the month of February, and the date of 29 February. Each of these cases is guarded by a `choose(Bool)` choice point and this has two desirable consequences. Firstly, when the default choice model is used for this generator the choice point returns `true` with probability 0.5, and so these classes will be more frequently sampled than they would be from a uniform distribution. Secondly, when the choice model is optimised, these choice points have a parameter controlling the local Bernoulli distribution (see Section II-C), and this provides a mechanism for the optimisation process to change how frequently these classes are sampled.

We defined the domain of the regex data type to be the set of ASCII strings. Since there is no bound on the length of the string, precise uniform sampling from this domain is impractical: as length increases, the more strings there are of that length, and so uniform sampling would return extremely long strings that would be unrealistic for testing. We consider instead three generators that uniformly sample ASCII strings of length 8, 16, and 32 respectively; it is the generator for the last of these, `Uniform32Regex`, that is listed in appendix B.

The `PosixRegex` generator emits regexes that comply with the extended POSIX syntax for regular expressions. Using knowledge of this domain, the generator was written so as to explicitly construct many of the most common features of this syntax. In a similar manner to the `LeapYearDate` generator, the choice points may be used to control how often each of these features appear in regexes sampled from the domain.

E. Experiment 1

This experiment addresses research questions RQ1 and RQ2 by evaluating the efficiency of a uniform generator, an ensemble generator using a default choice model, and the same ensemble generator using a choice model optimised to the SUT.

The ensemble generator for dates is formed from the `UniformDate` and `LeapYearDate` generators. Since the `LeapYearDate` generator emits only dates in leap years, the `UniformDate` generator can be used by the ensemble to sample dates in the rest of the input domain.

As discussed in Section IV-A, all three SUTs that take regex inputs check the validity of the regular expressions. Therefore the ensemble generator is formed from both the `Uniform32Regex` generator—in order to sample some strings that are not valid regular expressions—and the `PosixRegex` generator to sample valid regexes that will exercise particular features of the POSIX syntax.

Thus for each data type, the efficiency of the following three generators are evaluated individually on each SUT:

- the uniform generator;
- the ensemble generator using the default parameters to the sampler choice model; and,
- the ensemble generator using a sample choice model optimised to the SUT according to the process described in Section IV-C.

We present the results using graphs of condition coverage against test set size of the type shown in Figure 4. To account for stochasticity in the generation process, the coverage value plotted is the mean over the 50 test sets sampled from the generator. To account for stochasticity in the optimisation process applied to the third generator, these 50 test sets are the accummulation of 10 test sets sampled from each of the generators derived by 5 separate runs of the optimisation process.

The results for the four SUTs taking date inputs are shown in Figure 5, and for the three SUTs taking regex inputs in Figure 6.

As discussed in Section IV-B, we additionally report coverage for REP using the 32 variants of the SUT containing seeded faults that are provided by the Software-artifact Infrastucture Repository. In this case the coverage is the proportion of variants that return outputs that differ from the original version of the SUT.

The difference between the ensemble using the default sampler choice model (blue dashed line) and the uniform generator (red full line) is the quantity considered in RQ1. For all SUTs apart from WN, the graphs show that the ensemble is more efficient than the uniform generator: higher coverage is achieved more quickly[1].

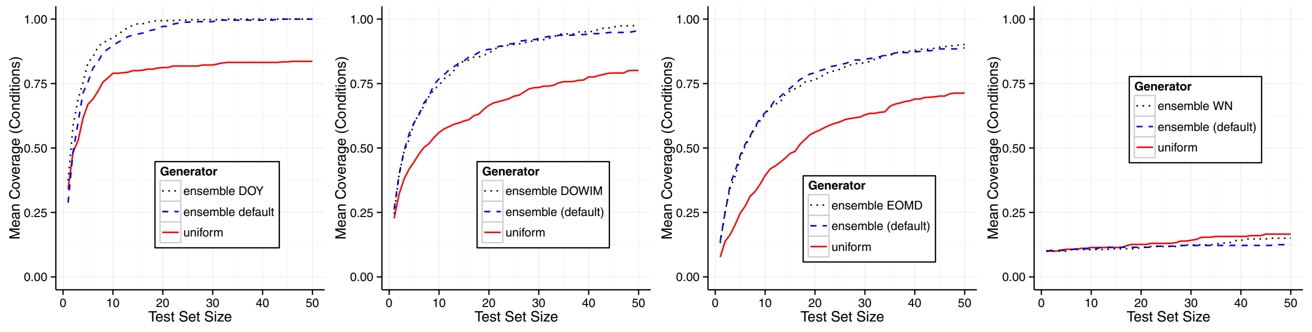
For clarity, we omit error bars from the graphs, but can assess whether the observed differences are statistically significant using the sets of f_{eff} values calculated for the 50 test sets sampled from the uniform generator, and for the 50 test sets for the default ensemble generator. When the Wilcoxon rank-sum test is applied to this data, the differences are significant between the uniform generator—for regexes, specifically the best performing uniform generator, `Uniform32Regex`—and the default ensemble generator for all SUTs apart from WN: all the p-values are (much) less than 1%.

We note that the larger the length of the uniform strings generated for the regex SUTs, the better the coverage. This is to be expected because a longer string can contain a greater number of regex features; indeed, it is possible that a very long string could achieve near total coverage. However it would much more difficult for the tester to manually check the output of the SUT for correctness when the test inputs are long. We argue on this basis that `Uniform32Regex` is a realistic uniform generator against which to compare the ensemble. Moreover, in this and subsequent experiments, the average length of regexes generated by the ensemble generator is always less than 32 characters, and so the improved efficiency of the ensemble generator compared to the uniform generator is not because it produces longer regexes.

In these graphs, the difference between the ensemble and the uniform generator can appear quite small. However, this belies the practical significance of the difference. For example, the gap between the ensemble and the uniform generator is relatively small for BBRP[2] in Figure 6a. However, the graph

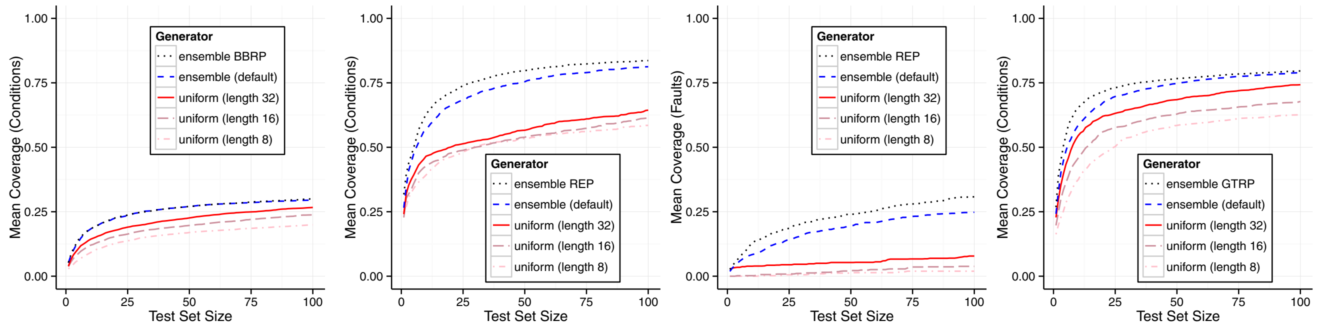
[1]Note that in graph of Figure 6a, the results for the ensemble (default) and ensemble (BBRP) are almost identical and therefore the lines overlay one another.

[2]We believe that the coverage of BBRP is lower than that of GTRP and REP because it implements more of the POSIX extended syntax, not all of which is emitted by the `PosixRegex` generator.



(a) DOY (b) DOWIM (c) EOMD (d) WN

Fig. 5. Results of Experiment 1, date type: coverage against test set size. (The legend 'ensemble X' indicates that the generator has been optimised for SUT X.)



(a) BBRP (b) REP (c) REP (seeded faults) (d) GTRP

Fig. 6. Results of Experiment 1, regex type: coverage against test set size. (The legend 'ensemble X' indicates that the generator has been optimised for SUT X.)

demonstrates to achieve a coverage of 25% would require a test set of 77 inputs using the uniform generator (red full line), but a much smaller test set size of 31 using the ensemble generator (blue dashed line).

We speculate that the poor efficiency of the ensemble for WN is because the domain knowledge it incorporates is not appropriate for its functionality. This SUT calculates the week number of a given date, and although the calculation does depend on whether the year is a leap year or not, this check is only made under very rare circumstances: when the date is the sixth day of the last week of the year. For this reason, there is little to gain in terms of test efficiency by generating leap year dates more frequently. We revisit this issue in Experiment 3 below.

The difference between the optimised ensemble (black dotted line) and the ensemble using the default sampler choice model (blue dashed line) is the quantity considered in RQ2. The effect differs between the two data types. For the SUTs taking a date type, there is little, if any, improvement in the efficiency compared to the ensemble using a default choice model; none of the differences are statistically significant at the 1% level. We speculate that the default choice model is already, by chance, near-optimal for this ensemble in the context of these SUTs. Of the SUTs taking a regex type, the improvement in efficiency is statistically significant at the 1% level for REP and GTRP, while there is no statistically significant difference for BBRP (indeed, the lines overlay one another on the graph).

For REP, the ordering of generator efficiencies are consistent between the two different measures of coverage: condition coverage and the detection of seeded faults. This gives some confidence that the differences in efficiency observed for the other SUTs assessed using only condition coverage are indicative of practical differences in the ability to detect faults. We note that the proportion of faults detected is much lower than the proportion of conditions covered, and that the efficiencies of uniform generators for REP are particularly poor by this measure compared to the ensemble.

We re-iterate that the objective of this paper is to demonstrate the re-use of generators of complex test data types, rather than to argue as to the potential efficacy of test data generation using stochastic generating programs or grammars: we have investigated the latter in previous work [6], [8], [11]. Nevertheless we comment briefly on the relatively poor coverage of the regex ensemble on all SUTs, and BBRP in particular. We believe this is because the regex ensemble generator covers only core regex syntax and that a more comprehensive generator of both the regex and the replacement string would be required to achieve near complete coverage.

F. Experiment 2

This experiment addresses RQ3. We consider the case of a library of generators (as proposed in Section III) which contains one generator for each complex data type. The objective is to assess how efficient generators in the library would be for new, previously unseen, SUTs. For the purpose of this experiment, we assume that generators provided by the library for the date and regex types are simply the ensembles used in Experiment 1.

For each SUT in turn, the library generator is optimised using *only* the other SUTs taking the same data type. For example, for REP, the regex ensemble is optimised using a portfolio consisting of BBRP and GTRP. The corresponds to the scenario in which the library generator was derived and optimised using BBRP and GTRP, and that REP is the new, previously unseen, SUT which we would like to test using the library ensemble. We therefore measure the efficiency of the library generator (i.e. the ensemble optimised using BBRP and GTRP) for testing REP. (For the reasons discussed above, we exclude WN for this experiment and revisit it in Experiment 3.)

The results for the SUTs taking date inputs are shown in Figure 7, and for the SUTs taking regex inputs in Figure 8. The results from Experiment 1 for the uniform generator (red full line) and ensemble optimised specifically for the SUT (black dotted line) are also plotted on the graphs for comparison.

In all cases, the library ensemble (green dot-dashed line) is more efficient than the uniform generator, and these differences are all statistically significant at the 1% level. In general the library ensemble (i.e. the generator optimised against the other SUTs) is less efficient than the ensemble optimised specifically for the SUT, as might be expected, although for DOWIM and EOMD the two are stastically indistinguishable.

These results provide evidence in support of RQ3: generators that have proven effective for existing SUTs can form a library of generators suitable for previously-unseen SUTs that use the sample complex data type: a library generator can be more efficient than the uniform distribution for random testing. This support the key premise of this paper: that generators for complex data types can be shared between SUTs, and are likely to be a more efficient 'first guess' than the uniform distribution.

G. *Experiment 3*

This experiment addresses RQ4 by revisiting WN, the SUT for which the ensemble generator was no better than the uniform distribution in Experiment 1. We envision a scenario in which the tester extends the ensemble with an additional generator, `FirstLastWeekDate` (listed in appendix A) in order to more frequently sample dates in the first and last weeks of the year: a simple analysis of the SUT's code or its specification suggests that special handling is performed by the SUT for these dates. We check whether this extended ensemble has improved efficiency for WN by optimising against all four SUTs, and then analysing its efficiency for WN.

However, a concern is that if this extended ensemble were to become the new generator for dates supplied by the library, to what extent would its efficency for other SUTs be reduced? To assess this, we repeat the procedure of Experiment 2 using the extended ensemble as the library generator, and additionally include WN in the portfolio of SUTs against which the extended library generator is optimised.

The results for the SUTs taking date inputs are shown in Figure 9. The results from Experiment 1 for the uniform generator (red full line) and optimised ensemble (black dotted line), and from Experiment 2 for the optimised library ensemble (green dot-dashed line), are included for comparison.

Figure 9a shows that the extended ensemble (cyan dashed line) has much better efficiency than the unextended ensemble. The other three graphs show that as of result of extending the library generator with `FirstLastWeekDate`, the efficiency is reduced (although the difference is not statistically significant at the 1% level for DOY). We therefore conclude that refining generators in the library as new SUTs are encountered may be beneficial for the new SUTs, but can lead to a reduction in efficiency for other SUTs. Nevertheless, the extended ensemble in this experiment remains significantly more efficient than the uniform distribution for random testing for all SUTs.

V. Related Work

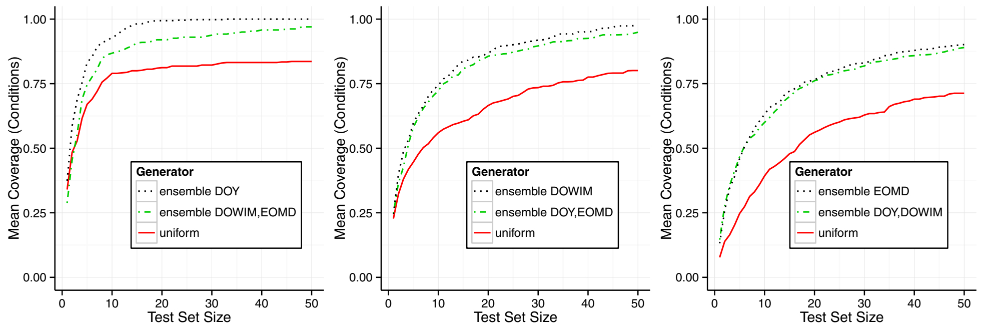
Automated techniques for generating structured test inputs typically use either grammars or non-deterministic generating programs. The algorithms of Beyene and Andrews [10], and of Poulding et al. [11] are examples of the grammar-based approach and similar to our approach [5], [6]: they all use optimisation to generate test cases with desirable properties.

However, grammar-based approaches can only express a subset of the generators expressible with non-deterministic generating programs of the type used by GödelTest. UDITA [4] is a Java-based framework that generates test sets by exhaustively enumerating all points of non-determinism in the generating program up to a chosen bound. QuickCheck [3] also promotes the use of a full programming language in defining random data generators and then use them to check that desirable properties are fulfilled. However, while GödelTest can automatically optimize the generators for desirable properties a tester must manually do this in Quickcheck and has no explicit support for how to do so.

There has been little previous work on reuseable libraries of test data generators even though re-use of testing artefacts has been a long-term goal. Von Mayrhauser et al. used domain analysis to create models that was claimed to support test reuse [12]. Their interactive and automated testing tool Sleuth allowed testers to reuse different elements of the test scripts, test commands as well as test cases themselves when testing within related domains. The tool was used in an industrial case study but no quantitative evidence of re-use or its benefits were reported. Similarly, Cai et al. [13] proposed that test reuse can be helped by a library based on Z specifications but provide no tangible evidence of actual re-use or test re-use benefits.

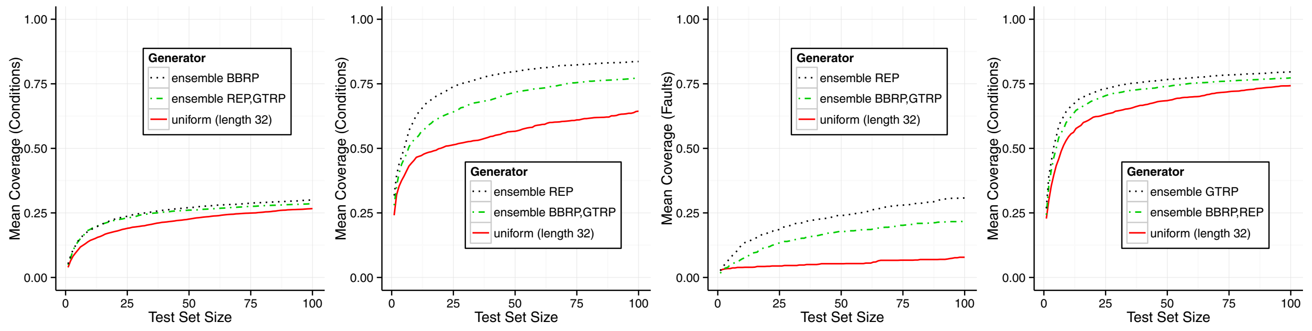
VI. Conclusions

We proposed that the generators of complex data types may be re-used between SUTs when sampling test inputs. The hypothesis underlying this proposal is that there exists sufficient commonality between SUTs in the way that the data type is processed for generators optimised against one SUT to demonstrate good efficiency when used to test another SUT. We performed an empirical investigation of this hypothesis for two data types and assessed generator efficiency using seven real-world SUTs. The results are consistent with the hypothesis: generators optimised against portfolios of SUTs generally demonstrate test efficiency for new SUTs that is significantly better than would be achieved using a uniform



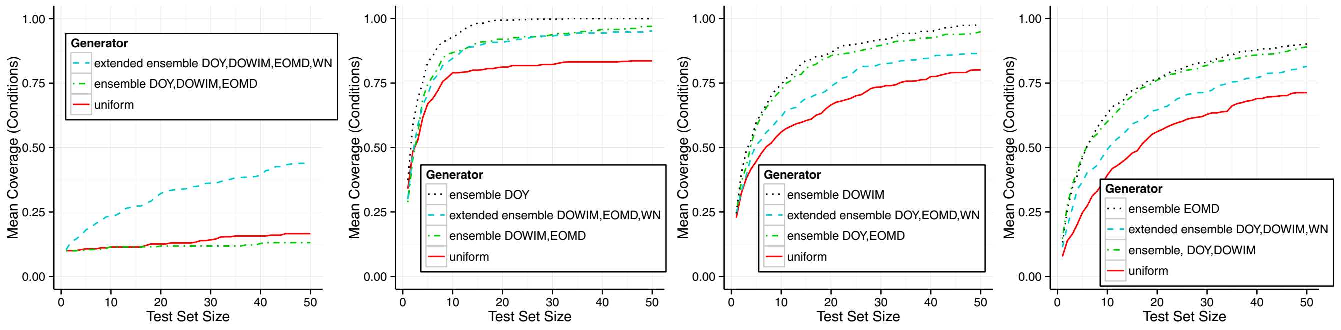
(a) DOY (b) DOWIM (c) EOMD

Fig. 7. Results of Experiment 2, date type: coverage against test set size. (The legend ‘ensemble X,Y’ indicates that the generator has been optimised for the porfolio of SUTs X and Y.)



(a) BBRP (b) REP (c) REP (seeded faults) (d) GTRP

Fig. 8. Results of Experiment 2, regex type: coverage against test set size. (The legend ‘ensemble X,Y’ indicates that the generator has been optimised for the porfolio of SUTs X and Y.)



(a) WN (b) DOY (c) DOWIM (d) EOMD

Fig. 9. Experiment 3, date type: coverage against test set size. (The legend ‘ensemble X,Y’ indicates that the generator has been optimised for the porfolio of SUTs X and Y.)

distribution, and suggest that it would be beneficial to maintain libraries of generators for common data types.

The final experiment hinted at an issue that we intend to address in future work: it is possible to refine a library generator when a SUT is encountered that requires a very different distribution of test inputs, but this is likely to reduce the efficiency of the generator in general. The challenge will be to identify when it is appropriate to establish a new generator specific to a subclass of SUTs rather than refine the existing generator.

Appendix A
Date Generators

```
@generator UniformDate begin
  start = begin
    y = choose(Int, 1600, 2099)
    m = choose(Int, 1, 12)
    if m in (1,3,5,7,8,10,12)
      d = choose(Int, 1, 31)
    elseif m in (4,6,9,11)
      d = choose(Int, 1, 30)
    elseif (y % 400 == 0)
      || ((y % 4 == 0) && (y % 100 != 0))
      d = choose(Int, 1, 29)
    else
      d = choose(Int, 1, 28)
    end
    y,m,d
  end
end

@generator LeapYearDate begin
  start = begin
    if choose(Bool)
      y = 1600 + 400choose(Int, 0, 1)
    else
      c = choose(Int,16,20)
      y = 100c + 4choose(Int,1,24)
    end
    if choose(Bool)
      m = 2
      if choose(Bool)
        d = 29
      else
        d = choose(Int,1,28)
      end
    else
      if choose(Bool)
        midx = choose(Int,1,7)
        m = [1,3,5,7,8,10,12][midx]
        d = choose(Int,1,31)
      else
        midx = choose(Int,1,4)
        m = [4,6,9,11][midx]
        d = choose(Int,1,30)
      end
    end
    y,m,d
  end
end

@generator FirstLastWeekDate begin
  start = begin
    y = choose(Int, 1600, 2099)
    if choose(Bool)
      m = 1
      d = choose(Int,1,7)
    else
      m = 12
      d = choose(Int,25,31)
    end
    y,m,d
  end
end
```

Appendix B
Regex Generators

```
@generator Uniform32Regex begin
  start = join(reps(:asciichar,32,32))
  asciichar = char(choose(Int,32,126))
end

@generator PosixGen begin
  start = (choose(Bool) ? "^" : "")
        * subexp() * (choose(Bool) ? "\$" : "")
  subexp = begin
    se = join(reps(:elementrep,0,4),
                choose(Bool) ? "|" : "")
    if choose(Bool)
      se = "(" * se * ")"
    end
    se
  end
  elementrep = element() * (choose(Bool) ? repeats() : "")
  element = literal()
  element = "."
  element = "[" * (choose(Bool) ? "^" : "")
            * join(reps(:bracketsubexp,1,4)) * "]"
  literal = begin
    lit = string(char(choose(Int,32,126)))
    if lit in ("[","]","*","+","^","\$","\\"
                ,"{","}","(",")",".","?","|")
      lit = "\\" * lit
    end
    lit
  end
  bracketsubexp = literal()
  bracketsubexp = literal() * "-" * literal()
  element = charclass()
  charclass = "[[:alpha:]]"
  charclass = "[[:digit:]]"
  charclass = "[[:upper:]]"
  charclass = "[[:lower:]]"
  charclass = "[[:space:]]"
  charclass = "\\w"
  charclass = "\\W"
  repeats = "*"
  repeats = "+"
  repeats = "?"
  repeats = "{" * string(choose(Int,0,16)) * "}"
  repeats = "{" * string(choose(Int,0,16)) * ",}"
  repeats = begin
    m = choose(Int,0,16)
    n = m + choose(Int,0,16)
    "{" * string(m) * "," * string(n) * "}"
  end
end
```

Acknowledgments

This work was funded by The Knowledge Foundation (KKS) through the project 20130085, Testing of Critical System Characteristics (TOCSYC).

References

[1] P. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Software*, vol. 7, no. 4, pp. 50–55, July 1990.

[2] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," *SIGPLAN Not.*, vol. 35, no. 1, pp. 1–13, 1999.

[3] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proc. 5th ACM SIGPLAN Int'l Conf. Functional Programming (ICFP)*, 2000, pp. 268–279.

[4] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *Proc. 32nd ACM/IEEE Int'l Conf. on Software Engineering (ICSE)*, 2010, pp. 225–234.

[5] R. Feldt and S. Poulding, "Finding test data with specific properties via metaheuristic search," in *Proc. of 24th IEEE Int'l Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 350–359.

[6] S. Poulding and R. Feldt, "Generating structured test data with specific properties using Nested Monte-Carlo Search," in *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, 2014, pp. 1279–1286.

[7] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[8] S. Poulding and H. Waeselynck, "Adding contextual guidance to the automated search for probabilistic test profiles," in *Proc. IEEE Int'l Conf on Software Testing, Verification and Validation (ICST)*, 2014, pp. 293–302.

[9] K. Price, R. M. Storn, and J. A. Lampinen, *Differential evolution: a practical approach to global optimization*. Springer, 2006.

[10] M. Beyene and J. Andrews, "Generating string test data for code coverage," in *Proc. IEEE Int'l Conf. Software Testing, Verification and Validation (ICST)*, 2012, pp. 270–279.

[11] S. Poulding, R. Alexander, J. A. Clark, and M. J. Hadley, "The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing," in *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, 2013, pp. 1477–1484.

[12] A. Von Mayrhauser, R. Mraz, J. Walls, and P. Ocken, "Domain based testing: increasing test case reuse," in *Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors (ICCD)*, 1994, pp. 484–491.

[13] L. Cai, W. Tong, G. Yang, and Z. Liu, "Reusable test models and application based on z specification," in *Proc. 2nd Int'l Conf. Pervasive Computing and Applications (ICPCA)*, 2007, pp. 562–567.