Finding Test Data with Specific Properties via Metaheuristic Search

Robert Feldt
Dept. of Computer Science and Engineering,
Chalmers University of Technology, Sweden
Email: robert.feldt@chalmers.se

Simon Poulding
Dept. of Computer Science,
University of York, York, UK
Email: simon.poulding@york.ac.uk

***Abstract*—For software testing to be effective the test data should cover a large and diverse range of the possible input domain. Boltzmann samplers were recently introduced as a systematic method to randomly generate data with a range of sizes from combinatorial classes, and there are a number of automated testing frameworks that serve a similar purpose. However, size is only one of many possible properties that data generated for software testing should exhibit. For the testing of realistic software systems we also need to trade off between multiple different properties or search for specific instances of data that combine several properties. In this paper we propose a general search-based framework for finding test data with specific properties. In particular, we use a metaheuristic, differential evolution, to search for stochastic models for the data generator. Evaluation of the framework demonstrates that it is more general and flexible than existing solutions based on random sampling.**

I. Introduction

Effective testing of realistic software systems can require the use of highly-structured test inputs. Examples are tree-like index structures for testing information retrieval software, source code for testing interpreters and compilers, and concrete model instances for testing toolchains used in model-driven engineering.

These data structures are ‘well-formed’ in the sense that they satisfy often complex constraints regarding their construction. For trees, the constraints may be expressed in terms of the number of child nodes at non-leaf nodes: e.g. all nodes in a binary tree must have at most two child nodes. Valid source code, i.e. text that exercises compilation functionality rather than raising errors in the lexer or parser, must satisfy the syntactical constraints of the programming language. Concrete model instances must conform to a metamodel describing the relationships between objects and the attributes they contain.

Many classes of data structure have no restriction on the size of instances in the class. For some forms of testing, it may be appropriate to choose a small upper bound on the size of the structure, e.g. the number of nodes in a tree, and exhaustively generate all possible instances of the data structure up to this size bound. However, if testing requires larger structures—for example, to stress the software, or to evaluate how its performance scales—exhaustive generation is no longer feasible: the number of valid instances typically grows very quickly as the upper bound increases. The solution in this case is to sample instances at random from a probability distribution, and to choose the distribution so that the average size of the instances is sufficiently large. It is this random sampling approach to test data generation that we consider in this paper.

Depending on the purpose of the testing, data characteristics other than size will be relevant. For example, unit testing may benefit from the frequent generation of data structures that exercise particular parts of the code, while predicting reliability may require that the distribution of data emulates that experienced by the software in operation. Such distributions consider additional characteristics that may be intrinsic to the data structures themselves, such as the height of a tree or cyclometric complexity of the source code; or extrinsic, such as the structural coverage of the software.

We use the term *bias objective* to refer to the specific properties that the sampled data structure instances should exhibit. Large average size is an example of such a bias objective, but desirable biases may involve any other intrinsic or extrinsic characteristics of the data structures. The challenge is therefore to derive an efficient algorithm that randomly samples well-formed data structures from distributions that satisfy the bias objectives specified by the test engineer.

To meet this challenge, we propose (a) the use of programs that incorporate non-deterministic constructs in order to concisely express how to generate all well-formed instances of a particular data structure class; and, (b) the application of automated search to tune the non-determinism in order to create desirable biases.

This approach has two significant advantages. Firstly, non-deterministic programs can generate a much wider range of data structure classes than existing analytical approaches such as state-of-the-art algorithms based on Boltzmann samplers (see section II-A), and approaches that use formal grammars (see section II-B). Secondly, any bias objective expressible as a fitness metric may be targeted by the search, and multi-objective optimization can be used to explore the trade-off between competing biases.

In section II, we consider existing approaches to the generation of well-formed data structures, and discuss the potential benefits and difficulties using non-deterministic programs for this purpose. Our proposed search-based framework, called GödelTest, is described section III. In section V, we explore the capabilities of the framework on a problem defined by multiple bias objectives, and compare the results to those obtained from a Boltzmann sampler and from an existing automated testing framework. We conclude the paper and discuss research directions in section VI.

II. Background and Related Work

A. Data Generation by Boltzmann Samplers

Boltzmann samplers were proposed by Duchon et al. as efficient algorithms for the random sampling of well-formed combinatorial data structures [1]. Boltzmann samplers guarantee uniform sampling in the sense that all instances of the *same* size, n, from a given data structure class have the same probability of being sampled. (In the absence of any other bias objective, uniform random sampling can be a desirable characteristic for testing purposes since there is no unnecessary bias in the sampling.) This probability value differs according to the size n: it is proportional to x^n where $x > 0$ is a control parameter of the sampler. The distribution of sizes of the sampled objects may therefore be controlled by choosing a suitable value for x.

Boltzmann samplers are derived by first decomposing the generation of the data structure class using a small set of construction operators that combine structurally simpler data structures. At the bottom of this decomposition is an atomic structure (e.g. a tree node) that cannot be decomposed any further. The decomposition results in set of grammar rules that may include recursion.

To illustrate this decomposition, let us denote the class of (unlabeled) general trees—trees in which there is no restriction on the number of child nodes—as $\mathcal{G}$. The root of the tree is a single atomic tree node, denoted by the class $\mathcal{Z}$, combined with an ordered list of zero or more child sub-trees, each of which themselves is a general tree (independent members of the class $\mathcal{G}$). Therefore the class of general trees is decomposed into the recursive grammar rule:

$$\mathcal{G} = \mathcal{Z} \cdot \mathbf{sequence}(\mathcal{G}) \tag{1}$$

Here the *product* construction operator, denoted $\mathcal{A} \cdot \mathcal{B}$, combines one randomly selected instance from the subclass $\mathcal{A}$ with one randomly selected instance from the subclass $\mathcal{B}$. The *sequence* operator, denoted $\mathbf{sequence}(\mathcal{A})$, first samples a random integer $k \geq 0$, then samples k independent instances from the subclass $\mathcal{A}$, and combines them as an ordered list.

This form of decomposition is not specific to Boltzmann samplers. Instead, the contribution of Boltzmann samplers is a systematic process for constructing an algorithm based on this decomposition which samples data structures from a distribution with the uniformity property discussed above. Each construction operator is associated with a specific local probability distribution that is sampled during the generation process. For example, each sequence operator is associated with a geometric distribution that determines the size of sequence. The parameters of these local probability distributions are calculated from the control parameter x, using functions that are themselves constructed systematically according to the decomposition of the subclasses on which the construction operators operate.

Boltzmann samplers have two very desirable characteristics. Firstly, it is possible to calculate, with relative ease, the value of x that will give a specific mean size for the sampled instances. Secondly, the generation process scales well with target instance sizes. A practical application of a sampler may additionally apply sophisticated rejection mechanisms to filter out sampled instances that have sizes outside a small range around the chosen mean size. Even when combined with filtering in this way, Duchon et al. show that for many classes, the time complexity of Boltzmann samplers scales linearly with the mean size.

A testing application of Boltzmann sampler that leverages this scalability is described by Canou and Darrasse [2]. The objective is to generate very large (sizes of 10^5 nodes and greater) random instances of tree-like data types in Objective Caml; such instances are used in a statistical analysis of the performance of the software-under-test. The data type definition is decomposed into a grammar that uses construction operators discussed above, and from this a type-specific Boltzmann sampler is constructed systematically.

However, there are two limitations to the use of Boltzmann samplers in the context of software testing. Firstly, the data structure must be capable of decomposition using the small set of construction operators from which Boltzmann samplers may be derived, and this limits the applicability of the approach. For example, red-black trees have constraints on the tree structure that cannot be expressed in the Boltzmann sampler grammar. Secondly, the underlying mathematics of Boltzmann samplers considers biases only in terms of the property of size: there is no analytical approach to biasing the Boltzmann sampler for other properties. We consider this a significant barrier to the application of Boltzmann sampling to the testing of complex, realistic software.

B. Data Generation Using Formal Grammars

As for Boltzmann samplers, approaches that generate test data using formal grammars—a technique known as grammar-based testing—use a set of production rules to specify how the data structure may be constructed.

Grammar-based approaches may be used for both bounded-exhaustive and random generation of test data. In the latter case, the grammar is converted into a stochastic form by assigning weights to the production rules: when more than one rule can be applied during the generation of the data structure, one of the rules is chosen at random using probabilities proportional to the weights. Maurer [3] uses such a stochastic grammar to verify digital circuits in simulation; McKeeman [4] for testing compilers; and Sirer and Bershad [5] to test implementations of Java virtual machines.

The advantage of using formal grammars is that they are often able to represent a wider range of data structure classes than Boltzmann samples, particularly when the canonical grammar is extended in order to express particular features of the test data. For example, Maurer extends a context-free grammar in order to store and retrieve state information during the generation process, and with code fragments that are executed when production rules are applied.

The disadvantage of stochastic grammars is that, unlike Boltzmann samplers, there is no general analytical approach for determining how to set the production rule weights (the equivalent to the local probability distributions of Boltzmann samplers) in such a way that the distribution of sampled data structures meets the specified bias objectives.

However, techniques based on constraint solving and on search have been recently proposed for this purpose, and it is notable that they are able to meet objectives for properties other than size. Dreyfus et al. [6] propose a constraint-based technique for satisfying an intrinsic bias objective—coverage of features of the test data itself—and illustrate their approach on the objective of covering of all terminal symbols in the grammar as frequently as possible. Both Beyene and Andrews [7], and one of us (Poulding) [8], describe the use of metaheuristic search in optimising the weights of a stochastic grammar to meet an extrinsic bias objective: coverage of structural elements in the software-under-test.

C. Data Generation by Non-Deterministic Programs

Just as the extension of canonical grammars permits a wider range of data structure classes to be generated, non-deterministic programs widen the applicability yet further by dispensing with the remaining restrictions of a grammar-based representation. The programs are written in languages with non-deterministic constructs that are able to concisely express choices during the generation of data structures.

UDITA, although used for bounded exhaustive generation, is an example of a sophisticated non-deterministic programming framework for generating test data [9]. UDITA extends Java to permit non-deterministic choices of both primitive data types and objects that enable generators for data structures to be created. In [9], the authors demonstrate the application of UDITA to the generation of well-formed red-black trees.

QuickCheck is a testing tool that utilizes non-deterministic programs to generate random test data [10]. As for UDITA, the programs—written in the functional programming language Haskell—construct generators by combining generators for simpler data types and built-in random number generators.

The flexibility of non-deterministic programs further complicates the problem of deriving a distribution over the data structures that satisfies the chosen bias objectives. Indeed, neither QuickCheck nor UDITA provide a mechanism for optimising the local non-determinism in generators in order to meet global bias objectives: the engineer must explore such biases manually. In stochastic grammars, the problem can be abstracted away from the details of the grammar by considering only the production rule weights: the problem becomes one finding suitable values for a fixed number of weights. However, the non-deterministic constructs of these programs are more complex in nature than production rule weights and so the abstraction and solution of the problem is more challenging. It is this issue that we address in this paper.

III. GÖDELTEST - A SEARCH-BASED DATA GENERATION FRAMEWORK

Our framework, GödelTest, combines the flexibility of non-deterministic programs as a means to generate a wide range of data structures with metaheuristic search to optimize the biases in the generated data. In this section, we describe the features and capabilities of this framework.

We chose to implement the version of the framework used in the empirical work of this paper using Ruby since this language provides extensive meta-programming features that facilitate the implementation. (These implementation details are discussed in section IV.) For this reason, we illustrate our description of the GödelTest framework using data-generating programs written in Ruby. This choice of implementation language need *not* restrict the language in which the target of the generated data—the software-under-test—is written: the data may be exported by the framework to be imported by a suitable test harness.

A. Overview

The two core concepts in the framework are *generators* and *choice models*.

A generator is a non-deterministic program that describes how to construct a well-formed data structure. In this respect it is equivalent to the grammars used by Boltzmann samplers and grammar-based testing. The non-determinism in the generator code—necessary for it to generate different concrete instances of the data structure—arises from the use of a small set of constructs that either (a) influence control flow by identifying alternative execution paths that can be taken; or (b), influence state by setting specific variables with primitive data types to random values. Each instance of such a construct is called a *choice point*.

The choice model describes how to resolve the non-determinism when running the generator program. Each time a choice point is encountered at execution time, the choice model is queried in order to decide which execution path to take, or which value a variable should take. The interface between the generator and choice model is a sequence of numbers, which we term *Gödel numbers*[1], that identifies which of the options to choose at each choice point.

Therefore, by varying the choice model, we may influence the probability distribution over the data structures output by the generator with the objective of introducing biases that optimize the efficiency of the test process. In this paper, we describe the use of metaheuristic optimization ('search') algorithms for this purpose. The clean separation of generator and choice model facilitates this process: the algorithm need only search over the space of possible choice models and can ignore other irrelevant details of how the data structure is constructed. The space of choice models is defined by a *sampler factory* which specifies a set of *samplers*: elementary components from which a choice model may be constructed and then tuned.

This structure of GödelTest framework is summarized in Figure 1. In the sections below we describe the features of generators and choice models in more detail.

B. Generator and Choice Points

A generator consists of a set of related rules (each implemented as a Ruby method) that together describe how a valid data structure is generated. It has an entry point and can then,

[1] A Gödel numbering is an assignment of elements from a countable mathematical object to natural numbers to allow algorithmic manipulation of the mathematical object. Even though our simpler choice models typically do not operate on the sequence as a whole (even though they can in principle and might in the future), and the analogy is thus often not perfect, we utilize this terminology for both the numbers and the name of the framework.

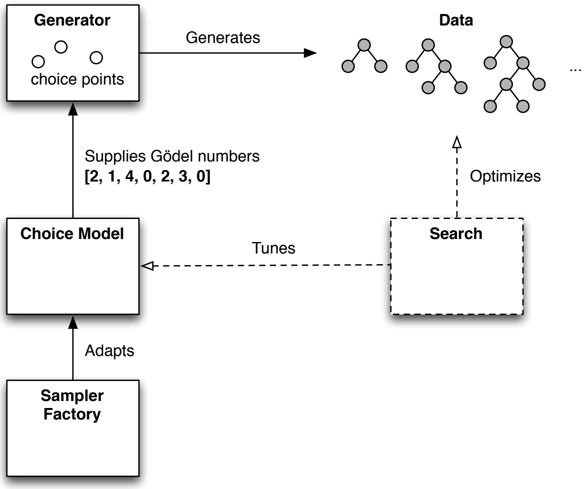

Fig. 1. Overview of the GödelTest framework, its core concepts and their connections

possibly recursively, call other methods in the same way as deterministic code.

Choice points enable non-determinism in the generator by identifying choices that can be taken as to control flow or data state in the generator. In the current implementation of GödelTest there are four types of choice points: one implicit and three explicit[2]. The four choice points are listed in Table I.

The `mult` and `plus` explicit choice points capture the two types of repetition common during the construction of data structures, and are the equivalent of similar constructs in regular expressions and other grammars. They take a name of a method, call that method a random number of times, and combine all the values returned by the method calls as an `Array`. The only distinction between them is that while `plus` always makes at least one call to the method, `mult` might make no calls at all and just return an empty array.

The `choose_int` choice point samples an integer at random from an interval defined by arguments to the choice point. It enables a more concise and controllable mechanism for specifying non-deterministic integer values than would be possible with the repetition constructs alone.

The implicit choice point *rule choice* corresponds to randomly selecting one of a set of alternative rules, much like the 'or' (`|`) construct in context-free grammars. A *rule choice* is created automatically whenever at least two methods in a generator have the same name. When the code for the generator is loaded each method is given a unique index among the methods with the same name. Thus a natural way to specify alternative constructions of a sub-structure of the data is to implement each one as a method in the generator.

Each time any of these four choice points is executed when running the generator, the choice point queries the choice model to determine its behavior. The Gödel number provided by the choice model determines the number of times a method is executed by `mult` or `plus`; the integer returned by a call to `choose_int`; and the index of the rule executed when a *rule choice* method is called.

TABLE I. CHOICE POINTS IN GÖDELTEST

Type	Returns	Description
`mult(:m)`	Array	Zero or more repeated calls to the method named *m*, returns an array of the values returned from each call
`plus(:m)`	Array	One or more repeated calls to the method named *m*, returns an array of the values returned from each call
`choose_int(min, max)`	Integer	Sample an integer between *min* and *max*, inclusive
rule choice	*Varies*	Select one of a set of possible methods with the same name (this choice point is implicit), calls it and then returns its return value

Figure 2 shows a simple example of a GödelTest generator which illustrates these constructs[3]. This generator can construct sequences of simple arithmetic expressions. The generator is written as a normal Ruby class that inherits `GodelTest::Generator`. The generator has six methods and uses three different types of choice points: `plus`, `choose_int` and two *rule choices* (one rule choice for the method named `expression` and one for `operation`). The method named `start` is always the entry point to a generator and will be called each time an object is to be generated. Here it is defined as a call to `plus` which will perform one or more calls to the method `expression` and return an array of the value returned from each such call. The `expression` method has two definitions and is therefore a *rule choice*: each time the method is called, the choice model is queried to determine which of the two implementations to execute. The second implementation of `expression` will simply call the `number` method which generates an integer and returns it as a string.

The example also shows one of the benefits of writing generators in a programming language rather than using grammars: the number of compound, parenthesized expressions, i.e. the first definition of the `expression` method, can be limited by a count kept in the instance variable named `@large_left`. Each time a compound expression is returned the state variable is decremented and when it reaches `0` this method will fall back to just generating a number. This would be difficult to achieve with a grammar-based approach.

The final line of code of the example will call the generator, here by specifying the initial value of the state variable. This will return the generated object and print it out as a string. One example of the output from running this Ruby program is:

```
["(6+3)", "((9+2)+8)", "(2-5)", "6"]
```

In this case the generator created 4 expressions, three of which were compound.

[2] As in any language design problem there is a trade-off between what can be efficiently expressed and how many base constructs are needed to enable that expression; we strive for practical trade-off rather than minimality, at least in this version.

[3] The example is somewhat contrived for illustration purposes.

```
class SeqOfExprGen < GodelTest::Generator
  def start
    plus(:expression)
  end

  def expression
    return number if @large_left < 1
    @large_left -= 1
    "(" + expression + operation + expression + ")"
  end
  def expression
    number
  end

  def operation
    "+"
  end
  def operation
    "-"
  end

  def number
    choose_int(0, 10).to_s
  end
end

puts SeqOfExprGen.generate_with_state\
  ({"@large_left" => 7}).to_s
```

Fig. 2. Example of a generator for sequences of simple, size-limited expressions

C. Choice Model and Samplers

A choice model controls the resolution of non-determinism when a generator is executed. In this paper, the choice models are stochastic and thus represent a probability distribution over the data structures that are output by the generator Such a choice model enables the random generation of test data with desirable biases as described in the introduction. However, the separation of generators and choice models in GödelTest permits a great deal of flexibility as to the nature of choice models: we have implemented choice models that permit bounded exhaustive testing, and envisage models in the form of adaptive learners.

Choice models are constructed from basic building blocks called samplers. In this paper we use a simple form of choice model whereby each choice point instance is assigned its own dedicated sampler in order to provide Gödel numbers. The Gödel numbers sampled for one choice point are thus independent of the numbers sampled for other choice points in the generator. We considered this simple form of choice model first, and found it effective for the empirical work presented here. However, it is a strength of our design that other choice models can implement more elaborate schemes. For example, we have experimented with choice models that describe dependencies between the Gödel numbers sampled for different choice points.

Samplers have parameters that determine which Gödel numbers are returned by the choice model. It is the values of these parameters that are tuned by the search algorithm in order to optimize the choice model. For the stochastic samplers used in this work, the parameters control the statistical distributions from which the Gödel numbers are sampled.

A sampler factory defines a coherent set of samplers that can be used to construct a choice model. This mechanism facilitates extensions to the GödelTest framework: as we investigate new types of samplers, we can provide them as additional sampler factories.

To provide concrete examples of samplers, we describe here two base samplers and one meta sampler that we utilize in the empirical work of section V.

The `GeometricSampler` samples Gödel numbers according to a geometric distribution for which the probability of the Gödel number k is:

$$\mathbb{P}(k) = \begin{cases} (1-p)p^k & \text{if } 0 \leq k \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where $0 \leq p \leq 1$ is a parameter to the sampler[4].

The `HistogramSampler` defines a categorical distribution across an interval of integers. The interval is by default $[0, 4]$, but a different interval can be specified on instantiation of the sampler. For this default interval, the sampler takes five weight parameters, p_0, p_1, p_2, p_3, p_4, and the probability of the Gödel number k is:

$$\mathbb{P}(k) = \begin{cases} p_k / \sum_{i=0}^{4} p_i & \text{if } 0 \leq k \leq 4 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where the denominator, $\sum_{i=0}^{4} p_i$, is used to normalize the weights so that they form a set of probabilities.

Each time the choice model is queried at a choice point, the generator provides information about the choice point (its type and unique identification number) as well as the execution context. This information is made available to the sampler and can be used to dynamically adapt the behavior of the choice model during execution.

The is particularly useful when a generator method calls itself: a common idiom in the generation of many data structures. The context information includes the recursion depth and so the choice model can change its output depending on the depth. In the sampler factory used in this paper, a meta sampler called `DecaySampler` may be applied to a base sampler for this purpose. If the base sampler takes n parameters, the `DecaySampler` takes a further set of n parameters that specify how the base sampler's behavior changes with recursion depth.

For example, the `DecaySampler` (with one parameter $0 \leq r \leq 1$) may be applied to `GeometricSampler` (with one parameter p). The combined sampler returns Gödel numbers according to a geometric distribution with the parameter $p \cdot r^d$ where d is the recursion depth. Assuming r is strictly less than one, the combined sampler is increasingly likely to return Gödel numbers closer to 0 as the recursion get deeper: Figure 3 illustrates this behavior. Since a Gödel number of 0 typically indicates no further recursion (for example, when the choice point is `mult`), this behavior avoids recursive generation of data structures of infinite size.

[4] Note that the definition of the sampler parameter (p) differs from that normally used for the geometric distribution (p') in that $p = 1 - p'$. This is for compatibility with the `DecaySampler` described below—it ensures that a the deepest levels of recursion the Gödel number 0 has the highest probability of being sampled—and for consistency with the definition of the geometric distribution used in Boltzmann samplers.

Similarly, for a combination of the DecaySampler (parameters r_i) and HistogramSampler (parameters p_i), at recursion depth d the i^{th} weight is $p_i \cdot {r_i}^d$.

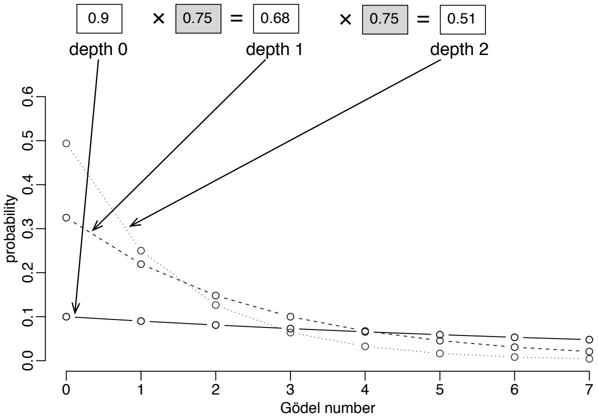

Fig. 3. The probability distributions used by a combination of a DecaySampler (parameter 0.75) and a GeometricSampler (parameter 0.9) at the first three recursion depths.

IV. Implementation of GödelTest

The implementation of GödelTest is written in the Ruby programming language [11]. Its core currently contains 945 non-blank, non-comment lines of code with another 146 lines implementing standard generators and 369 lines implementing different choice model and sampler classes. This excludes the search and optimization framework for which the generic evolutionary and statistical optimization classes of the Ruby gem feldtruby is used. Currently we use a Differential Evolution (DE) [12] optimizer for tuning model parameters but we see no reason simpler optimizers could not work well; it will depend on the specific model class and how well suited its models are to the generator being optimized.

There are three non-trivial techniques used in the current implementation that rely on Ruby and its meta-programming abilities: method renaming, code instrumentation, and simple static analysis. All three are performed dynamically as the code for a generator is loaded. Since all Ruby code is executed as it is loaded, we are able to detect the addition of methods to a subclass of the Generator class and to rename and instrument methods so that we can control their later execution. The method renaming allows multiple method definitions to have the same name without later definitions overwriting earlier ones which is the default Ruby behavior. It also allows us to instrument the generator code and insert a dispatch method for each *rule choice*. This dispatch method will first call the choice model and use the returned Gödel number to decide which specific method implementation to invoke.

The simple static analysis of the code in each method body of a generator is used to detect and extract information about the explicit choice points. The implementation also ensures that each choice point of a generator, whether explicit or implicit (rule choice), has a unique identification number. After all methods of a generator has been loaded we thus have instrumented the code to insert dispatch methods, renamed all methods of an implicit choice point and created information about all choice points, their context and constraints. This information is later used by model factories to adapt a choice model to a generator. It is also used by the samplers in the choice model to provide appropriate Gödel numbers during the generation process.

Even though Ruby's dynamic nature and flexibility makes it easier to perform these operations they are not essential; the main benefit of using Ruby and existing extension libraries is that these fairly advanced techniques can be implemented concisely. However, corresponding effects could be achieved in any other, modern programming language. For a language like C, that does not provide any meta-programming facilities, a combination of preprocessing and parsing to rename methods and extract information would be needed. For Java there are several bytecode manipulation and instrumentation libraries and solutions available that could be of help. As an alternative approach the current Ruby-based implemenation could be used to generate data or test code for these other languages. This can be accomplished easily by generating strings representing data or test code for other languages, even though performance and level of integration might suffer.

V. Empirical Evaluation

A. Objectives

In this section we describe a simple set of experiments that illustrate the capabilities of GödelTest. We compare the ability of GödelTest to generate data structures according to chosen bias objectives with that of Boltzmann samplers (an analytical technique) and QuickCheck (a non-deterministic program technique)[5].

Boltzmann samplers are the most restrictive of the three techniques in terms of the types of data structures that can be generated: the structure must be expressible in the combinatorial grammar used by Boltzmann samplers. We therefore choose unlabeled general trees—trees in which there is no restriction on the number of child nodes at each branch node—as the data structure: in section II-A we described how this data structure may be expressed in the Boltzmann sampler grammar. Although this data structure is simple and does not exercise the flexibility of either QuickCheck or GödelTest, it nevertheless illustrates the capabilities of all three techniques.

We target biases in two properties *simultaneously*: mean tree size (i.e. number of nodes) and mean tree height (i.e. maximum number of nodes from the root to a leaf). The concept of size is built into both Boltzmann samplers and QuickCheck, but there is no straightforward way to accommodate other properties such as tree height in either of these techniques. However, GödelTest can target any property—such as both tree size and height—that is expressible as a fitness metric for the search algorithm, and we use these experiments to illustrate this capability. In the following experiments we consider two specific targets: (mean size = 100, *and* mean height = 6); and (mean size = 100, *and* mean height = 36).

[5] We use QuickCheck rather than UDITA—the other major non-deterministic program for test data generation discussed in the related work of section II-C—because QuickCheck samples data at random as does the version of GödelTest used in this paper, while UDITA is used for bounded exhaustive testing. Generating very large data structures in an (bounded or not) exhaustive testing paradigm is infeasible.

B. Configuration – Boltzmann Sampler

To configure the Boltzmann sampler of general trees, we applied the analytical approach of [1] to calculate the parameter of a geometric probability distribution that is associated with the **sequence** operator in equation (1). There is no mechanism to accommodate tree height in this analysis, and so we considered only the mean size of 100 that is common to both targets. For this mean size, the geometric distribution must use a parameter of approximately 0.502512. (Space considerations preclude us from giving further details of the calculation here.)

C. Configuration – QuickCheck

To configure QuickCheck, we implemented a general tree data type, `Tree`, and a generator for this data type as an instance of the typeclass `Arbitrary`. The Haskell source code is shown in Figure 4. The `listOf′` function is used to derive a random length for the list of child trees at each branch node while specifying the size of those child trees in a manner consistent with the target tree size. When generating trees we use a modified version of the QuickCheck `sample′` function (not shown) that specifies a tree size of 100.

The data structure size is a concept built in to the standard QuickCheck implementation and therefore we have used it to meet our first bias objective of mean tree size 100. To accommodate the second bias objective of tree height *simultaneously* would require significant (and therefore costly) custom coding by the test engineer to adapt the generation code: such coding would be non-trivial even for a Haskell or QuickCheck expert. We therefore consider it unrealistic to implement it for this comparison experiment: if we had done, similar custom code would need to be added to the GödelTest generator to ensure a fair comparison.

```
data Tree  = Node Int [Tree] deriving (Eq, Show, Ord)

instance Arbitrary Tree where
  arbitrary = sized tree'
  where tree' n = liftM2 Node arbitrary (
    resize (n-1) (listOf' arbitrary))

listOf' gen = sized $ \n ->
  do k <- choose (0,n)
    if k == 0
      then vectorOf 0 gen
      else vectorOf k (resize ((n+k-1) `div` k) gen)
```

Fig. 4. QuickCheck generator and declaration for unlabeled general trees

D. Configuration – GödelTest

1) Generator: The generator for general trees in GödelTest is show in Figure 5.

```
class GeneralTreeGen < GodelTest::Generator
  def start
    Tree.new(mult(:start))
  end
end
```

Fig. 5. GödelTest generator for unlabeled general trees

2) Choice Models: We consider two different choice models for this GödelTest generator. The generator has only one choice point—the recursive `mult` construct—and so each choice model consists of a single sampler.

In the first choice model, we use `GeometricSampler` as the base sampler. This is motivated by the use of a geometric distribution at the equivalent point in the Boltzmann sampler implementation. Since the choice point is recursive, we combine the `GeometricSampler` with a `DecaySampler` as described in section III-C in order to avoid infinite trees. We refer to this choice model in the following as Decay Geometric (DG). It has two parameters that can be tuned using metaheuristic: the single parameter to `GeometricSampler` and the decay rate parameter to `DecaySampler`. We apply the DG choice model to our first target: (mean size = 100, mean height = 6).

In the second choice model, we use a `HistogramSampler` (over the Gödel number interval [0,4]) as the base sampler. This sampler enables a wider range of probability distributions to be represented, but requires more parameters than the `GeometricSampler` for this purpose. Again we combined the base sampler with the `DecaySampler` to give the model more expressivity. We refer to this choice model as Decay Histogram[0,4] (DH04). It has a total of ten parameters that can be tuned: five for each of the Gödel number choices of the `HistogramSampler`, and a corresponding five decay rates for the `DecaySampler`. We apply the DH04 choice model to our second target: (mean size = 100, mean height = 36).

3) Metaheuristic Search with Differential Evolution: To tune the parameter of the choice models, we use a steady-state implementation of the Differential Evolution (DE) algorithm [12]. This choice was one of convenience more than by design; we see no reason why other search-based optimizers may not be effective on this problem. DE is known as a generally potent search-based optimizer for continuous variables; a recent survey of the DE state-of-the-art even calls it 'one of the most powerful stochastic real-parameter optimization algorithms in current use' [13]. It has been successfully used in complex, multi-objective optimization problems in many different domains.

DE is an example of an evolutionary algorithm taking its basic inspiration from Darwinian evolution as seen in nature. As such it evolves a population of candidate solutions by selection and breeding. Solutions that performs better than others, according to a fitness criteria, have a higher chance of being used as parents in the recombination that breeds new child solutions. These new solutions, in turn, will replace poorly performing solutions in the population. Over time, this evolutionary process leads to a set of solutions that gradually improve according to the fitness criteria. Evolutionary algorithms have proven to be very versatile optimization algorithms with multiple applications also in software testing [14], [15]. Compared to more traditional optimization methods they have fewer requirements of the fitness function and can thus be applied in more situations without problem-specific adaptations.

In contrast to other evolutionary algorithms, DE considers the difference among parent solutions when creating new candidate (child) solutions [13]. This has the important effect that the search will self-adapt its step size. As the population focuses on the best performing regions of the space of solutions being searched there will be smaller differences between solutions and the step size used when creating new solutions will decrease. This leads to a more fine-grained and focused optimization process over time. The DE implementation we use is the one implemented in the Ruby gem `feldtruby` (version 0.4.8). It is a steady-state version of DE, i.e. in each step of the algorithm only a single, new candidate solution is created. This is in contrast to most DE algorithms, which are typically generational, i.e. updates each solution in the population in each step. Even though there is some evidence that steady-state algorithms can improve the search process [16] there is no evidence that it is generally advisable. In our experience it can lead to simpler and shorter implementations and performs at least as good as generational evolutionary algorithms.

For the parameter settings of the DE algorithm we have used the defaults set in `feldtruby` which in turn are based on the recommendations from a recent survey of the DE literature [13]. The key parameter settings are shown in Table II below. We refer the reader to [13] for details and discuss here briefly only those parameters specific to this experimentation.

We use the traditional DE mutation and crossover strategy of DE/rand/1/bin. This is a random sampling of a total of 4 parents, using one difference vector and binomial crossover. Two of the sampled parents are used to calculate a difference vector (by an element-wise subtraction of their solution vectors) which is then added to one of the other sampled parents to create the 'donor' vector. A random, binomial crossover between the donor and the final, sampled parent then creates the new candidate solution. Even though there are studies were other DE strategies can give better performance DE/rand/1/bin has shown consistently good results over wide sets of problems.

The selection of parents in DE is typically performed by random, uniform sampling from the individuals in the population. In our prior experience of DE, the use of 'trivial geography', as proposed by Spector et al. [18], consistently improves performance on a large set of diverse test functions. We have thus used this type of sampling in the experimental work presented here and have restricted the radius, from which all parents for a DE mutation step are sampled, to eight consecutive individuals of the population array. Thus, the sampling of four parents is done by first randomly sampling a group of 8 consecutive solutions in the population and then sampling 4 parents from that group of 8.

For fitness we have used the range-independent fitness aggregation scheme of Bentley and Wakefield called Sum-of-Weighted-Global-Ratios (SWGR) [17]. We use this to get a single fitness value to optimize from three sub-objectives that we employ. The first two sub-objectives are to minimize the root mean square (RMS) value of the distance from the target objective value to the corresponding value for the generated trees. The third sub-objective is to minimize the failure rate, i.e. the number of generation attempts for a model that results in deeply nested calls that exceeds the call limit (1000 in our experiments). The fitness function thus puts pressure on the search process to favor choice models that do not lead to failed generation attempts and where the generated tree has a size and height close to the target.

E. Method

The two GödelTest choice models were each tuned in five separate optimization runs using the DE algorithm described above. For the DH04 choice model, a maximum of 20,000 fitness evaluations were performed, using a sample size of 10 to estimate mean tree size and height during evaluations. We refer to the tuned model as DH04[20k,10] where the values in brackets specify the number of evaluations and the sample size. For the DG choice model, a maximum of 5,000 fitness evaluations were performed, using a sample size of 25. We refer to the tuned model as DG[5k,25].

10,000 trees were then sampled from each of the four generation mechanisms: the Boltzmann sampler, QuickCheck, a tuned DH04[5k,25] GödelTest generator, and a tuned DG[20k,10] generator. The size and height of each sampled tree were measured in order to compare how accurately the target bias objectives were met.

F. Results and Discussion

Figure 6 show scatter plots of the sizes and heights of the 10,000 trees generated by each of the four generation mechanisms. The Boltzmann sampler and QuickCheck considered only the common tree size target of 100, and so both targets are indicated as crosses on these two scatter plots. The DH04[20k,10] targeted the simultaneous bias objective (mean size = 100, mean height = 36) indicated by a single cross; DG[5k,25] targeted the simultaneous bias objective (mean size = 100, mean height = 6). For each of DH04 and DG, we show a representative example resulting from one of the five optimization runs.

The two targets were chosen to illustrate that the Boltzmann sampler is unable to meet the tree height objectives. Even though the Boltzmann sampling model is theoretically justified and can, in principle, sample trees of specified mean size it does not seem to be a practical solution for more complex solutions: only a very small proportion of trees have sizes near the chosen mean value. QuickCheck shows a much narrower spread of tree sizes around the target of 100, but does not incorporate the tree height objective into its default generation mechanism. As a result, few trees are near the target of mean height 36.

The distribution of trees generated by the tuned GödelTest DH04 model is markedly different from that of the Boltzmann sampler as well as QuickCheck. Even though it is not sharply focused around the target, it samples a region in the space of trees which the Boltzmann sampler is unable to, and QuickCheck unlikely to, explore. The distribution achieved by the tuned DG model is again different from that of the Boltzmann sampler: while the distribution is close to the target for the tree height, the spread of tree sizes remains large.

Figure 7 illustrates the difference between the evaluated approaches in more detail. It plots the percentage of trees that are within a certain tolerance (as a percentage) of the target size and height. This is relevant because rejection filtering applied after the generation mechanism is the standard approach for

TABLE II. PARAMETER SETTINGS THE EXPERIMENTS

Parameter	Value	Description
Population size	100	Number of individuals in population
DE algorithm	rand/1/bin	Random parent selection, 1 difference vector (the difference between two of the parents) and binomial crossover
F	0.7	Scale factor (weight) of difference vector in DE mutation
CR	0.5	Crossover rate in DE binomial crossover
Number of parents	4	Number of parents sampled for each generation of a new candidate solution: one target parent (to be replaced), one base parent and two for calculating the difference vector
Sampling	Radius(8)	Parents are sampled from 8 consecutive individuals in the population (with wrap-around at the population 'edges')
Fitness aggregation	SWGR	Sum-of-Weighted-Global-Ratios, the range-independent fitness aggregation scheme for multi-objective optimization described in [17]
Fitness samples	10 or 25	Number of objects generated for each fitness calculation
Max calls	1000	Maximum number of method calls (in the generator) allowed when generating one object
Sub-objective 1	RMS to target size	Root mean square error between the size of a tree and the target size (used in both experiments)
Sub-objective 2	RMS to target height	Root mean square error between the height of a tree and the target height (used in the multi-objective experiment)
Sub-objective 3	Failure rate	Ratio of failed to total number of generation attempts

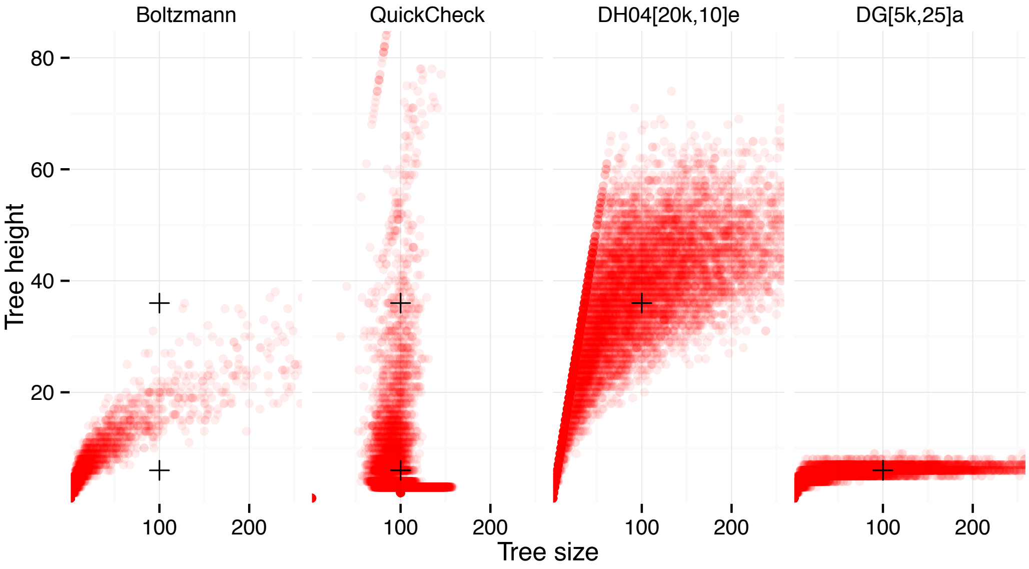

Fig. 6. Scatter plots of the size and heights of trees generated with three different generators: *left* Boltzmann, *mid left* QuickCheck, *mid right* Decay Histogram[0,4] with target (100, 36), and *right* Decay Geometric with target (100, 6)). The crosses in the left plots indicate the target sizes and heights that are the bias objectives in the right plots.

obtaining trees with specific characteristics both in the work on Boltzmann sampling and in the QuickCheck documentation: the higher the proportion of sampled trees that are within a chosen tolerance, the more efficient the rejection filtering approach will be.

For the GödelTest tuned DH04 model around 290 (2.9%) of the 10,000 generated trees have a size in the range 90-110 and a height in the range 33-39 (+/- 10%). This percentage is only 0.49% for the tested QuickCheck generator and 0.01% (a single tree) for the tested Boltzmann sampler. In the figure we also show the 95% confidence intervals calculated from the five separate optimization runs for the GödelTest generators; this shows that tuning results are quite tight on average even though individual search runs might produce sub-optimal results.

It is worth noting that the DG choice model has only two degrees of freedom and so is a very restricted model for the search to explore. When this model class (through its factory) was tuned to the other, (mean size = 100, mean height =

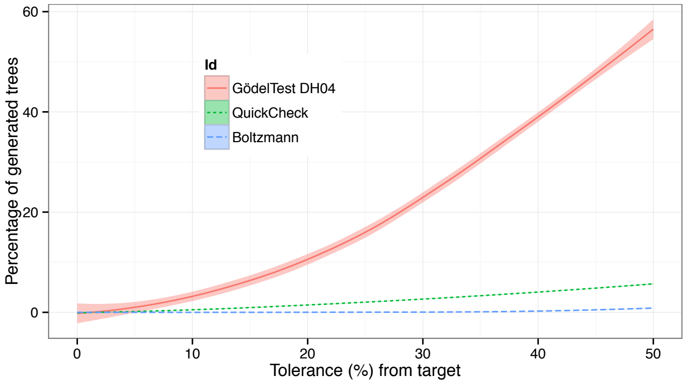

Fig. 7. Trees within a certain tolerance (%) from the target values (mean size = 100, mean height = 36) for GödelTest DH04 (red), Quickcheck (green) and Boltzmann sampling (blue)

36) target it had problems converging on good parameters. Understanding the properties of different model classes and how they can be (automatically) selected or searched for different targets and data generation problems is likely to be an important area of future research.

Importantly, the GödelTest generator for general trees is strikingly simple; essentially a single line of code (in addition to the minimum boilerplate). In comparison, the QuickCheck generator adapted to the targets in question is fairly complex and involves multiple, framework-specific constructs. For more complex generators and/or targets this advantage is likely to be even more substantial and will affect usability and adoption.

VI. Conclusion

This paper considers the problem of generating highly-structured test inputs in such a way that the distribution of test data has desirable bias objectives. These objectives may be chosen by the engineer for such purposes as evaluating scalability (a bias towards large structures), detecting faults efficiently, or predicting reliability by emulating the distribution of data that may experienced by the software in operation.

To address this problem, we have introduced the GödelTest framework. The framework enables a concise specification of the data structure's generation as a program extended with a small set of non-deterministic constructs: such a method of specification is much more flexible than either Boltzmann samplers or formal grammars. A choice model that describes the non-deterministic choice points in the program is abstracted automatically from the program. In this paper, we resolve the non-determinism using stochastic samplers that describe a local probability distribution at each point.

GödelTest enables the parameters of these local samplers to be tuned—here by metaheuristic search—in order to meet global bias objectives. To illustrate the capability of this approach, we considered the simultaneous satisfaction of two bias objectives for general tree instances. Both the Boltzmann sampler and QuickCheck generation techniques accommodate only one of these objectives, but GödelTest is able to bias the distribution in order to better meet *both* objectives.

The purpose of this paper has been to introduce the GödelTest framework and demonstrate its capabilities on a simple example that is nonetheless challenging for the analytical approach of Boltzmann samplers and the programmatic approach of QuickCheck. In future work we will explore the use of GödelTest in generating test data for more complex data structures (and thereby evaluate the framework's scalability), and formulations for the local samplers that provide a sufficient degree of freedom to enable bias objectives to be met while remaining amenable to search. As discussed in section III, GödelTest's abstraction of the choice model from the program will facilitate the use of the same framework for bounded exhaustive generation, and the dynamic adaptation of samplers using machine learning. It can also help improve early exploration in software development [19] where bias objectives can help ensure diversity of generated data.

Acknowledgments

This work is funded in part by EPSRC grant EP/J017515/1, DAASE: Dynamic Adaptive Automated Software Engineering, and in part by the SWELL research school and AVSATS grants from Vinnova.

References

[1] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer, "Boltzmann samplers for the random generation of combinatorial structures," *Combinatorics, Probability and Computing*, vol. 13, no. 4-5, pp. 577–625, 2004.

[2] B. Canou and A. Darrasse, "Fast and sound random generation for automated testing and benchmarking in objective caml," in *Proceedings of the 2009 ACM SIGPLAN workshop on ML*. ACM, 2009, pp. 61–70.

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

[4] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

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

[6] A. Dreyfus, P.-C. Heam, and O. Kouchnarenko, "Random grammar-based testing for covering all non-terminals," in *Proceedings of the 6th IEEE Conference of Software Testing, Verification and Validation Workshops (ICSTW)*, 2013, pp. 210–215.

[7] M. Beyene and J. Andrews, "Generating string test data for code coverage," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST 2012)*, 2012, pp. 270–279.

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

[9] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 225–234.

[10] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, 2000, pp. 268–279.

[11] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby*. Pragmatic Programmers, 2004.

[12] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[13] S. Das and P. N. Suganthan, "Differential evolution: A survey of the state-of-the-art," *Evolutionary Computation, IEEE Transactions on*, vol. 15, no. 1, pp. 4–31, 2011.

[14] M. Xiao, M. El-Attar, M. Reformat, and J. Miller, "Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques," *Empirical Software Engineering*, vol. 12, no. 2, pp. 183–239, 2007.

[15] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[16] R. Kumar and P. Rockett, "Improved sampling of the pareto-front in multiobjective genetic optimizations by steady-state evolution: a pareto converging genetic algorithm," *Evolutionary computation*, vol. 10, no. 3, pp. 283–314, 2002.

[17] P. J. Bentley and J. P. Wakefield, "Finding acceptable solutions in the pareto-optimal range using multiobjective genetic algorithms," in *Soft Computing in Engineering Design and Manufacturing*. Springer, 1998, pp. 231–240.

[18] L. Spector and J. Klein, "Trivial geography in genetic programming," in *Genetic programming theory and practice III*. Springer, 2006, pp. 109–123.

[19] R. Feldt, "Genetic programming as an explorative tool in early software development phases," in *Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering*, 1999, pp. 11–20.