Flexible Probabilistic Modeling for Search Based Test Data Generation

Downloaded from: https://research.chalmers.se, 2021-12-03 05:28 UTC

Citation for the original published paper (version of record):
Feldt, R., Yoo, S. (2020)
Flexible Probabilistic Modeling for Search Based Test Data Generation
Proceedings - 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops,

N.B. When citing this work, cite the original published paper.
Flexible Probabilistic Modeling for Search Based Test Data Generation

Robert Feldt  
Chalmers University  
Gothenburg, Sweden  
robert.feldt@chalmers.se

Shin Yoo  
School of Computing, KAIST  
Daejon, Republic of Korea  
shin.yoo@kaist.ac.kr

ABSTRACT
While Search-Based Software Testing (SBST) has improved significantly in the last decade we propose that more flexible, probabilistic models can be leveraged to improve it further. Rather than searching for an individual, or even sets of, test case(s) or datum(s) that fulfill specific needs the goal can be to learn a generative model tuned to output a useful family of values. Such generative models can naturally be decomposed into a structured generator and a probabilistic model that determines how to make non-deterministic choices during generation. While the former constrains the generation process to produce valid values the latter allows learning and tuning to specific goals. SBST techniques differ in their level of integration of the two but, regardless of how close it is, we argue that the flexibility and power of the probabilistic model will be a main determinant of success. In this short paper, we present how some existing SBST techniques can be viewed from this perspective and then propose additional techniques for flexible generative modelling the community should consider. In particular, Probabilistic Programming languages (PPLs) and Genetic Programming (GP) should be investigated since they allow for very flexible probabilistic modelling. Benefits could range from utilising the multiple program executions that SBST techniques typically require to allowing the encoding of high-level test strategies.

KEYWORDS
Probabilistic Programming, Software Testing

1 INTRODUCTION
Search Based Software Testing (SBST) has been successful for different types of testing such as structural testing [10, 19, 24] and non-functional property testing (e.g. temporal [26], energy [16]). Recently its maturity has also been shown by the success of the two but, regardless of how close it is, we argue that the flexibility and power of the probabilistic model will be a main determinant of success. In this short paper, we present how some existing SBST techniques can be viewed from this perspective and then propose additional techniques for flexible generative modelling the community should consider. In particular, Probabilistic Programming languages (PPLs) and Genetic Programming (GP) should be investigated since they allow for very flexible probabilistic modelling. Benefits could range from utilising the multiple program executions that SBST techniques typically require to allowing the encoding of high-level test strategies.

While the same techniques and tools can typically be used both to generate test input data as for generating whole test cases the latter is much less common and we will use the former as an example and focus throughout, for simplicity; what we propose is applicable in both settings though.

While the same techniques and tools can typically be used both to generate test input data as for generating whole test cases the latter is much less common and we will use the former as an example and focus throughout, for simplicity; what we propose is applicable in both settings though.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSEW’20, May 23–29, 2020, Seoul, Republic of Korea  
© 2020 Association for Computing Machinery.  
ACM ISBN 978-1-4503-7965-2/20/05 ... $15.00  
https://doi.org/10.1145/3387940.3392215

537
Figure 1: Overview of existing (white boxes), as well as our proposed (green), approaches to SBST seen as Generative Probabilistic Models. QuickCheck allows manual programming of data generators to specify input structures, but the sampling of inputs is random (or has to be hand-coded into the generator code making advanced probabilistic modeling hard). GödelTest separates generators and probabilistic models, allowing more diverse probabilistic distributions and models for sampling inputs. We propose a novel use of Probabilistic Programming Language as a means to express generative models. Additionally, we also propose the use of genetic programming for automated programming of both generative models and data generators.

a diverse set of test data. While the latter approach \cite{6} already parameterize a test data generator into a structural specification, that describes the valid, syntactic structure of data, and a (probabilistic) choice model that makes the non-deterministic choices within that structure during an actual generation process we argue that there is a continuum of more or less powerful such models that should be further explored.

In the following we describe an overview model of SBST approaches as seen as probabilistic generative models, map some existing techniques into it, and then propose new techniques that cover new and exciting niches of the model. In particular, we argue for the use of Probabilistic Programming Languages and Genetic Programming to be explored in future work since they can provide very powerful as well as flexible modeling, respectively.

2 SBST AS MODEL INFERENCE

This section discusses some of the limitations in existing test data generation techniques using a small, low-level, motivating example. It then presents a simple, overview model for SBST approaches seen as generative models and discusses some novel ideas of how to infer, or learn, such models based on Probabilistic Programming Languages and Genetic Programming.

2.1 Test Data Generators as Generative Models

Although generative models (GMs) have been around for very long, e.g. Jaakkola and Haussler \cite{15} argues for using traditional statistical model such as hidden Markov models as GMs, they have seen a resurgence of interest in recent years with the rise of Probabilistic Programming languages (PPLs) as well as Deep Neural Nets (DNNs). As defined by Foster \cite{9} a “generative model describes how a dataset is generated, in terms of a probabilistic model [and] by sampling from this model, we are able to generate new data.” We propose that this is a very natural way to view software test data generators and that SBST and many automated testing techniques can be seen in this light.

Our proposal is thus that automated testing techniques and SBST, in particular, should infer probabilistic generative models that will produce test input values that are adequate for the given criterion, instead of searching for a single (or small set of) adequate input value(s). As just one example, given a branch predicate $x > y + 42$, we would like to be able to produce a generative model that uses, say, two normal distributions, $x \sim N(84, 4)$ and $y \sim N(0, 3)$. Sampling $x$ and $y$ from these distributions will lead to covering the branch under test with high probability, each time with different values.

Many test data generation techniques do not make a difference between the structure of the data that is allowed and the probabilistic model that determines how to make specific choices. For example, in QuickCheck the tester or developer manually programs data generators; the structure that is allowed is described directly in Haskell\footnote{Nowadays QuickCheck style of tools are available in a plethora of languages, e.g. Hypothesis for Python \cite{1}} code while the probabilistic choices are mainly made by uniform random sampling or have to be explicitly specified right in the generating code itself \cite{5}. Even though the flexibility and
power of the probabilistic model is thus basically unlimited, since it can be expressed in the Turing complete language that is the host language of the tool, in practice data generation is basically (uniformly) random and the technique itself is described as a random testing technique.

While the probabilistic model in QuickCheck is thus embedded in and, thus, integrated with the description of the structure of the generated data, the GödelTest system proposed to clearly separate the two [6]. This is achieved by allowing the code that describes how to generate data, and thus its valid structure, to use a set of primitives to express the type of choices taken while generating a specific datum. This separation allows different, so called (in GödelTest parlance), choice models to be used and learned based on specific and even multi-objective goals [6]. These choice models are essentially probabilistic models that determines, via some form of sampling, which specific choice to make.

While Feldt and Pouling, when introducing GödelTest, focused on using local, univariate probability distributions for choice and then tuned the parameters of the distributions with a search algorithm they also considered more complex probabilistic (choice) models. For example, already in the initial study [6], they needed to use a "depth-dependent" probability distribution which was formed by multiplying the parameters of the probability distribution each time a choice was made. Thus the more choices of a specific type that had been made the more different the probability distribution could be. This was found to be useful when searching for tree data structures when the goal was to generate a tree of a certain size or depth. In a later study [21], they also proposed the use of Monte-Carlo Tree Search to optimize the choices made in the data generator; this is thus an example of a more flexible probabilistic model which can enable complex dependencies between multiple choices and aspects of the generation process.

### 2.2 Categorisation of Generative Models for SBST

Based on the discussion above there are thus at least two main aspects to be considered in a generative model: (a) to what degree is its probabilistic model separated from the description of the structure of the data, (b) how powerful and flexible are the probabilistic model itself and thus what level of control does it give. Separation of the model from the structure of the data allow more free experimentation with different probabilistic models, since the latter is not explicitly described in, and thus tied to, the former. And a more powerful and flexible probabilistic model (PM) allows more fine-grained control of the choices made during the generation process and can thus, at least in theory, come closer to the choices that lead to the data that we need during testing.

Figure 1 shows a conceptual overview of this model with flexibility/power of the PM on the X axis and the level of separation on the Y axis. We can see that the original QuickCheck based on random choices during data generation integrates the probabilistic model with the description of the data structure, while the simplest GödelTest variant uses local, univariate probability distributions. More flexible is to use the depth-varying probability distribution or the Monte Carlo Tree Search which allows for more complex dependencies between choices during data generation.

However, while the GödelTest technique essentially is a search-based approach to the learning, and thus inference in statistical parlance, of generative models we argue that SBST should exploit this spectrum even more broadly. In the following, we give three specific proposals of techniques that could provide benefits. For separated GMs, such as GödelTest, we suggest that the probabilistic model could be encoded in small programs which can be learnt via search, i.e. the probabilistic modeling is done via Genetic Programming. For integrated GMs we discuss how Probabilistic Programming Languages could automatically infer complex probabilistic models for test data generation. We also briefly discuss how Genetic Programming could be used to directly evolve also the code describing the structure of the data itself and not only its probabilistic model.

### 3 FUTURE SBST TECHNIQUES FOR GENERATIVE MODELING

#### 3.1 Probabilistic Programming

Probabilistic Programming is a programming paradigm that aims to both describe and automatically train probabilistic models using dedicated Probabilistic Programming Languages (PPLs) [11]. Many contemporary PPLs, such as Pyro [3] and Anglican [23], provide the combined package of the expressiveness of a full programming language with state-of-the-art model inference techniques. We will consider these languages as the model inference engine. Since the final probabilistic model is created automatically and based directly on the program, expressed in the PPL, this is an example of an integrated technique that can allow for diverse and very powerful probabilistic modeling.

As one example, existing search based test data generation techniques can be elegantly abstracted as Bayesian model-based reasoning which is used by several PPLs. Let \( X \) denote an input value for the Program Under Test (PUT), and \( Y \) denote an arbitrary fitness value. Searching for a specific value that satisfies a fitness criterion is in fact searching for a specific point in the joint distribution of input and fitness values. More generally, searching for an input can be considered as conditioning of a model on an observable output, i.e., the characterisation of a conditional distribution between \( X \) and \( Y \), \( P(X|Y) \), based on the instrumented PUT as a model. Bayes rule 1 can be used to approximate the conditional distribution: \( P(X) \) is the prior distribution of input \( X \), while \( P(X, Y) = P(Y|X)p(X) \) represents the observations of concrete executions based on the random choices made for \( X \).

\[
P(X|Y) = \frac{P(Y|X)p(X)}{P(Y)} = \frac{P(X, Y)}{\int P(X,Y)p(X) \, dX}
\] (1)

One of the major benefits of PPLs is that, being Turing complete, PPLs can be used to construct very sophisticated generative models, to which the aforementioned conditioning is applied. A widely used example is the problem of breaking Captcha images. One machine learning approach to this is to sample many Captcha images and then train a neural network to decode a string value out of a Captcha image [4]. Using PPLs, we can instead write a short program that generates Captcha images from strings, and condition this model on the image we want to break [25]. From this perspective, test data generation is an ideal application for PPLs, as the models we want to condition already exist (the source code and/or executable of the
PUTs), and PPLs can directly interact with them. We also note the opportunistic benefits for SBST to use state-of-the-art probabilistic inference techniques that are being rapidly incorporated into PPLs.

3.2 Genetic Programming for Probabilistic Modeling

While several different probabilistic modeling techniques have been used as choice models in GödelTest they each have their specific limitations and thus limited power and flexibility. The local, univariate probability distributions cannot model dependencies between different choices during generation, the depth-varying models only considered the individual depth per probability distribution not the number of choices made etc.

A potentially very flexible technique would be to evolve small genetic programs that give information about the overall generation process and the choices made so far calculates the next choice to be made during generation. The strength and flexibility of this model could be substantial, e.g. evolving programs that generate choices for bounded exhaustive testing or encode complex test strategies, but the search might also be very hard. Future work should investigate the trade-off between flexibility and feasibility, how to select between individual programs per each choice in a generator or having a single model/program for every choice etc.

3.3 Genetic Programming of Generators

One potential weakness of QuickCheck- and GödelTest-like techniques is that the cost of writing generators can quickly surpass the cost of manual testing. While there are techniques that can exploit existing data format specifications and grammars to generate the code needed for data generators [12] they depend on such specifications being available. This might not be so for new or custom-developed software.

In such scenarios, one of the most flexible as well as automated techniques for generative modeling would be to use automated programming techniques to search for the whole generator itself using e.g. Genetic Programming [17]. We show this as an integrated generative model in Figure 1 since it seems simpler to evolve both the structure-determining data generator code and the probabilistic model at the same time. However, hybrid approaches are possible and future work should explore their respective pros and cons.

While Genetic Programming [17] seems a natural choice here, as well as for the probabilistic models in Section 3.2 above, other forms of automated programming should be investigated, e.g. Inductive Logic Programming [20].

4 CONCLUSION

We propose an extension to SBST that aims to perform generative model inference instead of solution search. This will allow us to search for strategies rather than single instances of solutions as well as to sample many test inputs likely to have desirable attributes. As specific and promising means of this extension, we propose three technical components: the use of Probabilistic Programming Languages, the use of Genetic Programming, or other forms of automated programming, to automate the probabilistic modelling, and to automatically and directly create data generator code.

REFERENCES