Generating Structural Rich Random Abstract Data Type Values

Agustín Mista
Chalmers University of Technology
Gothenburg, Sweden
mista@chalmers.se

Alejandro Russo
Chalmers University of Technology
Gothenburg, Sweden
russo@chalmers.se

***Abstract*—Automatic generation of random values described by abstract data types (ADTs) is often a hard task. State-of-the-art random testing tools often can automatically synthesize random data generators based on ADTs definitions. In that manner, generated values comply with the structure described by ADTs—something that proves useful when testing software which expects complex input like web pages, images, or even programs. However, it sometimes becomes necessary to generate structural richer ADTs values in order to test deeper software layers. In this light, we propose to leverage static information found in the codebase as a manner to improve the generation process. Namely, our generators are capable to consider how programs branch on input data as well as how ADTs values are built via interfaces. We implement a tool, called DRAGEN*, responsible to synthezise generators for ADTs values while providing compile-time guarantees about their distributions. Our compile-time predictions allow DRAGEN* to provide an heuristic that tries to adjust the distribution of generators to what developers might want.**

Ale: Case study

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Random testing is a promising approach to finding bugs [1]–[3]. *QuickCheck* [4] is the dominant tool of this sort used by the Haskell community. It requires developers to specify (i) *testing properties* which describe programs behavior and (ii) *random data generator* based on the *types* of the expected inputs (e.g., an integer, an string, etc.). QuickCheck then generates random test cases and reports those violating testing properties.

QuickCheck comes equipped with random generators for built-in types, while it requires to manually write generators for user-defined ADTs. Recently, there have been a proliferation of tools to automatically derive QuickCheck generators for ADTs [5]–[9]. The main difference about these tools lies on the guarantees provided to ensure *the termination of the generation process* and the *distribution of random values*. Despite their differences, these tools guarantee that generated values are *well-typed*. In other words, generated values follow the structure described by the definition of the ADT.

Well-typed ADT values are specially useful when testing programs which expect highly structured inputs, e.g., compilers [10], [11]. Generating ADT values also proves fruitful when looking for vulnerabilities with fuzzers [8], [12]. Despite these success stories, ADT type-definitions do not often capture all the invariants expected from the data that they are intended to model. As a result, even if random values are well-typed, they might not be built with enough structure to penetrate into deep layers of software.

In this work, we propose a novel improvement in the generation process of ADT values by exploiting some static information found in the codebase. More specifically, to refine the structure of generated values, we propose a generation process that is capable to consider how programs branch based on ADTs values' patterns as well as how they get manipulated by abstract interfaces. Furthermore, we show how to predict the distribution of the *expected* numbers of ADT constructors, values fitting certain branching patterns, and calls to interfaces. For that, we extend some recent results on applying *branching processes* [13]—a simple stochastic model conceived to study population growth. We implement our ideas in a tool, called **DRAGEN***, that is capable to automatically synthesize QuickCheck generators for *rich ADT values*, where the distributions of random values can be adjusted at compile-time to what developers might want based on our predictions.

Ale: Add here later about test cases when we know what they are

We remark that, although this work focuses on Haskell algebraic data types, this technique is general enough to be applied to most programming languages.

II. Background

In this section, we briefly introduce the common approach for automatically deriving random data generators for ADTs in QuickCheck by using the information found at the type-level. To exemplify this, and for illustrative purposes, let us consider the following ADT definition to encode simple *Html* pages:

$$\begin{aligned}\textbf{data}\ \mathit{Html} = {} & \mathit{Text}\ \ \mathit{String} \\ \mid {} & \mathit{Single}\ \mathit{String} \\ \mid {} & \mathit{Tag}\ \ \mathit{String}\ \mathit{Html} \\ \mid {} & \mathit{Join}\ \ \mathit{Html}\ \ \mathit{Html}\end{aligned}$$

The type *Html* allows to build pages via four possible constructions: *Text*—which represents plain text values—, *Single* and *Tag*—which represent singular and paired Html tags, respectively—, and *Join*—which concats two Htmls pages

one after another. In Haskell, *Text*, *Single*, *Tag*, and *Join* are known as *data constructors* (or constructors for short) and are used to distiguish which variant of the ADT we are constructing. Each data constructor is defined as a product of zero or more types known as *fields*. For instance, *Text* has a field of type *String*, whereas *Join* two recusive fields of type *Html*. In general, we will say that a data constructor with no recursive fields is *terminal*, and *non-terminal* or *recursive* if it has at least one field of such nature. With this representation, the page `<html>hello<hr>bye</html>` can be encoded as:

```
Tag "html" (Join (Join
  (Text "hello") (Single "hr")) (Text "bye"))
```

A. Type-driven generation of random values

In order to generate random value of types involving ADTs, most approaches require users to provide a random data generator for each one of them. This is a cumbersome and error prone task and usually *follows closely the ADT structure*. For instance, consider the following definition of a *QuickCheck* random generator for the type *Html*:

$$
\begin{aligned}
&gen_{Html} = sized\ (\lambda size \rightarrow \\
&\quad \textbf{if}\ size \equiv 0\ \textbf{then}\ frequency \\
&\qquad\qquad [(2,\ Text \quad \langle\$\rangle\ gen_{String}) \\
&\qquad\qquad ,(1, Single\ \langle\$\rangle\ gen_{String})] \\
&\quad \textbf{else}\ frequency \\
&\qquad [(2,\ Text \quad \langle\$\rangle\ gen_{String}) \\
&\qquad ,(1, Single\ \langle\$\rangle\ gen_{String}) \\
&\qquad ,(4, Tag \quad \langle\$\rangle\ gen_{String}\ \langle*\rangle\ smaller\ gen_{Html}) \\
&\qquad ,(3, Join \quad \langle\$\rangle\ smaller\ gen_{Html}\ \langle*\rangle\ smaller\ gen_{Html})])
\end{aligned}
$$

We use the Haskell syntax $[\,]$ and $(,)$ for denoting lists and pairs of elements, respecitvely, e.g., $[(1,2),(3,4)]$ is a list of pairs of numbers. The random generator gen_{Html} is defined using *QuickCheck*'s function *sized* to paremetrize the generation process up to an external natural number known as the *generation size*—capture in the code with variable *size*. This parameter is chosen by the user, and it is used to limit the maximum amount of recursive calls that this random generator can perform and thus ensuring the termination of the generation process. When called with a positive generation size, this generator can pick to generate among any *Html* data constructor of with an explicitly given *generation frequency* that can be chosen by the user—in this example, 2, 1, 4 and 3 for *Text*, *Single*, *Tag*, and *Join*, respectively. When it picks to generate a *Text* or a *Single* data constructor, it also generates a random *String* value using the standard QuickCheck generator gen_{String}.[1] On the other hand, when it picks to generate a *Join* constructor, it also generates two independent random subexpression recursively, thus decreasing the generation size by a unit on each recursive invocation ($smaller\ gen_{Html}$). The case of random generation of *Tag* constructors follows analogously. This random process keeps calling itself recursively until the generation size reaches zero, where the generator is constrained to pick only among terminal data constructors, being *Text* and *Single* the only possible choices in our particular case.

The previous definition is rather mechanical, except perhaps for the chosen generation frequencies. **DRAGEN** [9] is a tool conceived to mitigate the problem of finding the appropriated generation frequencies. It uses the theory of branching processes [13] to modelize and predict analytically the expected number of generated data constructors. This prediction mechanism is used to feedback a simulation-based optimization process that adjusts the generation frequency of each data constructor in order to obtain a particular distribution of values that can be specified by the user—thus providing a flexible testing environment while still being mostly automated.

As many other tools for automatic derivation of generators (e.g., [5]–[7], [12]), **DRAGEN** synthesizes random generators similar to the one shown before, where the generation process is limited to pick *a single data constructor at the time and then recursively generate each required subexpression independently*. In practice, this procedure is often too generic to generate random data with enough structural complexity required for testing certain applications.

III. Sources of structural information

In this section, we describe the motivation for considering two additional sources of structural information which lead us to obtain better random data generators. We proceed to exemplify the need to consider such sources with examples.

A. Branching on input data

To exemplify the first source of structure information, we consider that we want to use randomly generated *Html* values to test the function $simplify :: Html \rightarrow Html$. In Haskell, the notation $f :: T$ means that program f has type T. In our example, function *simplify* takes an *Html* as an input and produces an *Html* value as an output—thus its type $Html \rightarrow Html$. Intuitively, the purpose of this function is to assemble sequences of *Text* constructors into a single big one. More specifically, the code of *simplify* is as follows.

$$
\begin{aligned}
&simplify :: Html \rightarrow Html \\
&simplify\ (Join\ (Text\ t_1)\ (Text\ t_2)) \\
&\quad = Text\ (concat\ t_1\ t_2) \\
&simplify\ (Join\ (Join\ (Text\ t_1)\ x)\ y) \\
&\quad = simplify\ (Join\ (Text\ t_1)\ (simplify\ (Join\ x\ y))) \\
&simplify\ (Join\ x\ y) = Join\ (simplify\ x)\ (simplify\ y) \\
&simplify\ (Tag\ t\ x) = Tag\ t\ (simplify\ x) \\
&simplify\ x = x
\end{aligned}
$$

The body of *simplify* is described using *pattern matching* over possible kind of *Html* values. (Function *concat* just concatenates two strings.) Pattern matching allows to define functions idiomatically by given different function clauses for each input pattern we are interested in. In other words, pattern matching is a mechanism that functions have to branch on input

[1] The operators $\langle\$\rangle$ and $\langle*\rangle$ are used in Haskell to combine values obtained from calling random generators and they are not particularly relevant for the point being made in this work.

arguments. In the code above, we can see that *simplify* patterns match against sequences of *Text* constructors combined by a *Join* constructor—see first and second clauses. Generally speaking, patterns can be defined to match specific constructors, literal values or variable subexpressions (like x in the last clause of *simplify*). Patterns can also be nested in order to match very specific values.

Ideally, we would like to put approximately the same amount of effort into testing each clause of the function *simplify*. However, each data constructor is generated independently by those generators automatically derived by just considering ADT definitions. Observe that the probability of generating a value satisfying a nested pattern (like $Join\ (Text\ t_1)\ (Text\ t_2)$) decreases multiplicatively with the number of constructors we simultaneously pattern against to. As an evidence of that, in our tests, we found that the first two clauses of *simplify* get exercised only approximately between 1.5% and 6% of the time when using the state-of-the-art tools for automatically deriving QuickCheck generators MegaDeTH [12] and **DRAGEN** [9]. Most of the generated values were exercising the simplest clauses of our function, i.e, $simplify\ (Join\ x\ y)$, $simplify\ (Tag\ t\ x)$, and $simplify\ x$.

Altough the previous example might seem rather simple, branching against specific patterns of the input data is not an uncommon task. In that light, and in order to obtain interesting test cases, it is desirable to conceive generators able to produce random values capable to exercise patterns with certain frequency—Section IV shows how to do so.

B. Abstract interfaces

A common choice when implementing ADTs is to transfer the responsibility of preserving structural invariants to the interfaces that manipulate values of such types. To illustrate this point, let us consider three new primitives responsible to handle *Html* data.

$$hr :: Html$$
$$hr = Single\ \texttt{"hr"}$$
$$div :: Html \to Html$$
$$div\ inner = Tag\ \texttt{"div"}\ inner$$
$$bold :: Html \to Html$$
$$bold\ inner = Tag\ \texttt{"b"}\ inner$$

These functions encode additional information about the structure of *Html* values in the form of specific Html tags. Primitive *hr* represents the tag `<hr>` used to separate content in an HTML pages. Function *div* and *bold* place an *Html* value within the tags `div` and `b` in order to introduce divisions and activate bold fonts, respectively. For instance, the page `<html><b>hello</b><hr>bye</html>` can be encoded as:

$$Tag\ \texttt{"html"}\ (Join\ (Join\ (bold\ (Text\ \texttt{"hello"}))\ hr)\ (Text\ \texttt{"bye"}))$$

Observe that, instead of including a new data constructor for each possible HTML tag in the *Html* definition (recall Section II), we defined a minimal general representation with a set of high-level primitives to build valid *Html* tags. This programming pattern is often found in a variety of Haskell libraries, being *html* **CITE**, *svg-builder* **CITE**some examples of this. As a consequence of this practice, generators derived by only looking into ADT definitions often fail to synthesize useful random values, e.g., random HTML pages with valid tags. After all, most of the *valid structures* of values has been encoded into the primitives of the ADT abstract interface. When considering the generator described in Section II, the chances of generating a *Tag* value representing a commonly used HTML tag such as *div* or *b* are extremely low.

So far, we have introduced two scenarios where derivation approaches based only on ADT definitions are unable to capture all the available structural information from the user codebase. Fortunately, this information can be automatically exploited and used to generate interesting and more structured random values. The next section introduces a model capable to encode structural information presented in this section into our automatically derived random generators in a modular and flexible way.

IV. Capturing ADTs Structure

In this section, we show how to augment the automatic process of deriving random data generators in **DRAGEN** with the structural information expressed by patterns matchings and abstract interfaces. The key idea of this work is to represent the different sources in an homogeneous way. Intuitively, the derived generators take the specification of the different structural sources and generate composite structured values on each random choice that it performs.

Figure 1 shows the workflow of our approach for the *Html* ADT. Based on the codebase, the user of our **DRAGEN*** indicates: (i) the ADT definition to consider (noted as $Html_{ADT}$), (ii) its patterns of interest (noted $Html_{Patterns}$), and (iii) the primitives from abstract interfaces worth to involve in the generation process (noted as $Html_{Interface}$). The tool then *automatically derived generators* for each source of structural information. These generators produce random *partial ADT values* in a way that it is easier to combine them in order to create structural richer ones. For instance, the generator obtained from $Html_{ADT}$ only generate constructors of the ADT but leaving incomplete the generation at the recursive fields, e.g., it generates values of the form $(Text\ \texttt{"xA2sx"})$, $(Single\ \texttt{"xj32da"})$, $(Tag\ \texttt{"divx234jx"}\ \bullet)$ and $(Join\ \bullet\ \bullet)$, where $\bullet$ is a placeholder denoting a "yet-to-complete" value. Similarly, the generator obtained from $Html_{Patterns}$ generates values satisfying the expected patterns where recursive fields are also left uncompleted, e.g., it generates values of the form $(Join\ (Text\ \bullet)\ (Text\ \bullet))$ and $(Join\ (Join\ (Text\ \bullet)\ \bullet)\ \bullet)$. Finally, the generator derived from $Html_{Interface}$ generates calls to the interface's primitives, where each argument of type *Html* is left uncompleted, e.g., $(div\ \bullet)$ and $(bold\ \bullet)$.

Observe that *partial ADT values* can be combined easily and the result is still a well-formed value of type *Html*. For instance, if we want to combine the following random

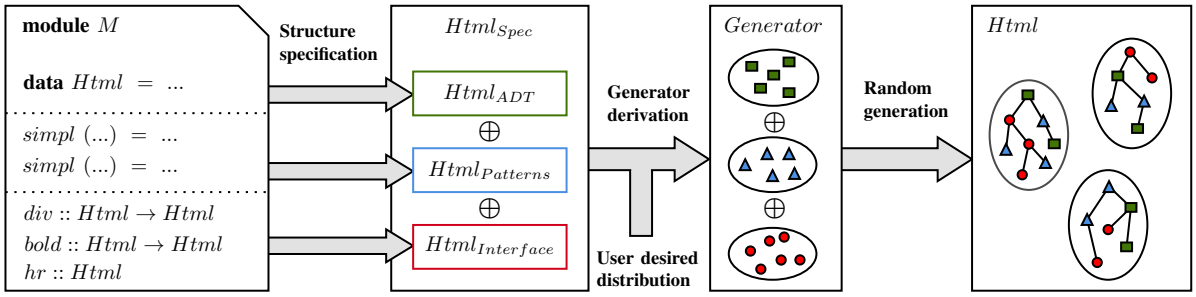

Fig. 1: Deriving a generator for the ADT *Html* with the structural information found in module M.

generated string `"bdfxwe"`, ADT value (*Text* `"xx34s"`), pattern (*Join* (*Join* (*Text* •) •), and interface call (*div* •), we can obtain the following well-formed *Html* test case:

Join (*Join* (*Text* `"bdfxwe"`) (*div* (*Text* `"xx34s"`))

Finally, our tool put all these three generators together into one that combines partial ADT values into fully formed ones. Importantly, the user can specify the desire distribution of the expected number of constructors, patterns, and interface calls that the generator will produce. All in all, our approach offers the following advantages over usual derivation of random generators based only on ADT definitions:

► **Composability:** our tool can combine different partial ADT values arising from different structural information sources depending on what property or sub-system becomes necessary to verify using randomly generated values.

► **Extensibility:** the developer can specify new sources of structural information and combine them with the existing ones simply by adding them to the existing specification of the target ADT.

► **Predictability:** the tool is capable to synthesize generators which distribution is adjusted to what developers might desire, e.g., an uniform distribution of patterns, a distribution where some constructors are generated twice as much as others, etc. We will explain the prediction of distributions in the next section.

We want to remark that, for space reasons, we were only able to introduce the specification of a rather simple target ADT like *Html*. In practice, this reasoning can be extended to mutually recursive and parametric ADT definitions as well.

V. Case Studies

Intuitively, adding structure information to a random generator should provide better results when verifying a system using random testing. This section describes two case studies showing that considering structure information when deriving generators consistenly produces better testing results.

Instead of restricting our scope to Haskell, in this work we followed a broader evaluation approach taken previously to compare the performance of other state-of-the-art derivation techniques [8], [9]. Concretely, we evaluate how including additional structure information when generating a set of random test cases (often referred as a *corpus*) affects the code coverage obtained when using them to test a given target program. For this purpose, we targeted two external programs which expect highly structured inputs, namely *GNU CLISP*—the GNU Common Lisp compiler, and *HTML Tidy*—a well known HTML refactoring and correction utility. We remark that these applications are not written in Haskell. However, there exist Haskell libraries defining ADTs encoding their input structure, i.e., Lisp and HTML values respectively. These libraries are: *hs-zuramaru*, implementing an embedded Lisp intepreter for a small subset of this programming language, and *html*, defining a combinator library for constructing HTML values. These libraries also come with serialization functions to map Haskell values into corresponding test case files.

We started by compiling instrumented versions of the target programs in a way that they also return the execution path followed in the source code every time we run them with a given input test case. This let us distinguish the amount of different execution paths that a randomly generated corpus can trigger. On the other side, we used the ADTs defined on the chosen libraries to derive random generators using **DRAGEN** and **DRAGEN***, including structure information extracted from the library's codebase in the case of the latter. Then, we evaluated the code coverage triggered by independent, randomly generated corpora of different sizes varying from 100 to 1000 test cases each. In order to remove any external bias, we derived random generators optimized to follow a uniform distribution of constructors (and pattern matchings or function calls in the case **DRAGEN***), and carefully adjusted their generation sizes to match the average test case size in bytes. This way, any noticeable difference in the code coverage can be attributed to the present (or lack thereof) structure information when generating the test cases. Additionally, to achieve statistical significance we repeated each experiment 30 times with independently generated sets of random test cases.

Fig. 2 illustrates the mean number of different execution paths triggered for each permutation of corpus size and derivation tool, including error bars indicating the standard error of the mean on each case. We proceed to describe each case study in detail.

a) Generating Lisp code using pattern matching information: In this first case study we wanted to evaluate the observed code coverage differences when considering structure

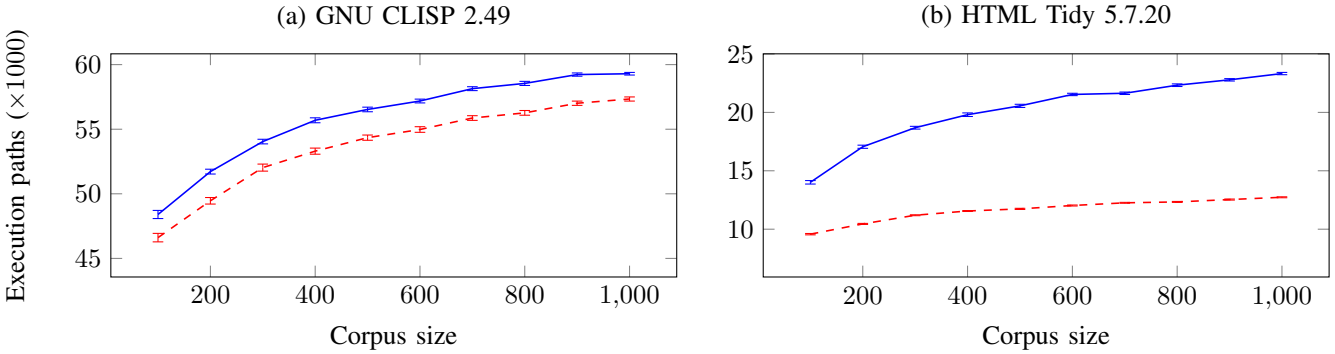

Fig. 2: Path coverage comparison between **DRAGEN** (- - -) and **DRAGEN*** (——).

information present on functions pattern matchings.

Our chosen library encodes Lisp S-expressions essentially as lists of symbols, represented as plain strings; and literal values like booleans or integers. In order to interpret Lisp programs, this unified representation of data and code requires this library to pattern match against common patterns like let-bindings, if-then-else expressions and arithmetic operators among others. In particular, each one of these patterns match a against special symbol of the Lisp syntax like `"let"`, `"if"` or `"+"`; and their corresponding sub-expressions.

In this case study we extracted the structure information from the pattern matchings present on this library and included it into the generation specification of our random Lisp values, which were generated by randomly picking from a total of 6 data constructors and 8 different pattern matchings. By doing this we obtained a consistent code coverage improvement of approximately 4% using **DRAGEN*** with respect to the one obtained with **DRAGEN** (see Figure 2 (a)).

AM: Shall we say anything justifing this little improvement?

b) Generating HTML pages using abstract interface information: For our second case study, we wanted to evaluate how including structural information coming from abstract interfaces when generating random HTML values might improve the testing performance.

The library we used for this purpose represents HTML values very much in the same way as we exemplify in Section II, i.e., defining a small set of general constructions representing plain text and tags—altough this library also supports HTML tag attributes as well. Then, this representation is extended with a large abstract interface consisting of combinators representing common HTML tags and tag attributes—equivalent to the combinators *div*, *bold* and *hr* illustrated before.

In this case study we included the structure information present on the abstract interface of this library into the generation specification of random HTML values, resulting in a generation process that randomly picked among 4 data constructors and 163 abstract combinators. With this large amount of additional structure information, we observed an increase of up to 83% in the code coverage obtained with **DRAGEN*** with respect to the one observed with **DRAGEN** (see Figure 2 (b)).

A manual inspection of the corpora generated with each tool revealed us that, in general terms, the test cases generated with **DRAGEN** rarely represent syntactically correct HTML values, consisting to a large extent of random strings within and between HTML tag delimeters (`"<"`, `">"` and `"/>"`). On the other hand, test cases generated with **DRAGEN*** encode much more interesting structural information, being mostly syntactically correct. We found that, in many cases, the test cases generated with **DRAGEN*** were parsed, analysed and reported as valid HTML values by the target application.

With these results we are confident that including the structural information present on the user codebase improves the testing performance. We consider that our approach is particularly useful when the data types encoding the shape of our data are vague or not sufficiently structured to be used to derive powerful random generators with the common type-driven derivation techniques.

VI. Related Work

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

VII. Final Remarks

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem.

REFERENCES

[1] J. Hughes, U. Norell, N. Smallbone, and T. Arts, "Find more bugs with QuickCheck!" in *Proc. of the International Workshop on Automation of Software Test, AST@ICSE*, 2016.

[2] J. Hughes, C. P. B, T. Arts, and U. Norell, "Mysteries of DropBox: Property-based testing of a distributed synchronization service," in *Proc. of the Int. Conference on Software Testing, Verification and Validation, ICST*, 2016.

[3] T. Arts, J. Hughes, U. Norell, and H. Svensson, "Testing AUTOSAR software with QuickCheck," in *In Proc. of IEEE International Conference on Software Testing, Verification and Validation, ICST Workshops*, 2015.

[4] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[5] N. Mitchell, "Deriving generic functions by example," in *Proc. of the 1st York Doctoral Syposium*. Tech. Report YCS-2007-421, Department of Computer Science, University of York, UK, 2007, pp. 55–62.

[6] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values," in *Proc. of the ACM SIGPLAN Symposium on Haskell*, 2008.

[7] J. Duregård, P. Jansson, and M. Wang, "Feat: Functional enumeration of algebraic types," in *Proc. of the ACM SIGPLAN Symposium on Haskell*, 2012.

[8] G. Grieco, M. Ceresa, A. Mista, and P. Buiras, "QuickFuzz testing for fun and profit," *Journal of Systems and Software*, vol. 134, no. Supp. C, 2017.

[9] A. Mista, A. Russo, and J. Hughes, "Branching processes for quickcheck generators," in *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, 2018, pp. 1–13.

[10] M. Pałka, K. Claessen, A. Russo, and J. Hughes, "Testing and optimising compiler by generating random lambda terms," in *The IEEE/ACM International Workshop on Automation of Software Test (AST 2011)*, 2011.

[11] J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson, and H. R. Nielson, "Effect-driven QuickChecking of compilers," *In Proceedings of the ACM on Programming Languages, Volume 1*, no. ICFP, 2017.

[12] G. Grieco, M. Ceresa, and P. Buiras, "QuickFuzz: An automatic random fuzzer for common file formats," in *Proc. of the International Symposium on Haskell*. ACM, 2016.

[13] H. W. Watson and F. Galton, "On the probability of the extinction of families," *The Journal of the Anthropological Institute of Great Britain and Ireland*, 1875.