Kolmogorov Complexity Justifies Software Engineering Heuristics

Ann Q. Gates, Vladik Kreinovich, and Luc Longpré

Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968, USA
email {agates,vladik,longpre}@cs.utep.edu

Abstract

Many software testing techniques are heuristic, so the "clean bill of health" produced by such a technique does not guarantee that the program is actually correct. In this paper, we show that several heuristic techniques for software testing that have been developed in software engineering can be rigorously justified. In this justification, we use Kolmogorov complexity to formalize the terms "simple" and "random" that these techniques use. The successful formalization of simple heuristics is a good indication that Kolmogorov complexity may be useful in formalizing more complicated heuristics as well.

Formulation of the problem. It is desirable to have programs that are 100% justified. Such programs exist, but they are extremely rare. Most programs do not use only mathematically justified methods of solving equations etc., they also use heuristic and semi-heuristic methods and ideas, i.e., methods that are not 100% justified. For such not-100%-justified programs, we must use *testing* to check whether a program is correct.

The half-a-century experience of software testing has led to several important techniques and recommendations for choosing such inputs (see, e.g., [1, 3, 5]).

However, many existing recommendations are only *heuristics*, i.e., methodologies that are justified by the experience and intuition rather than by a precise mathematical justification. Without a justification, undertaken on the mathematical strictness level, we cannot be sure that the tested program is correct.

An additional problem with these heuristics is that many of these recommendations use *imprecise* terms, i.e., words that are more or less understandable, but that are not precisely defined.

In this paper, we will show how such recommendations can be formalized and mathematically justified.

Let us start by describing the two heuristics that we will formalize.

Recommendations using the word "simple". A typical software engineering recommendation is *to try the program on simple data*: e.g., if a variable takes values from 1 to n, we must check it for 1, for n, maybe for a midpoint. It is also known that for *simple* programs and specifications, it is sufficient to check a few simple cases, while for more complicated programs and specifications, more complicated data must be also used for testing.

The word "simple" is more or less clear, and in different specific examples, it is explicitly and formally defined, but this word is not formally described in the general descriptions of this recommendation.

Recommendations using the word "random". Another term which is efficiently used in software engineering (and which is not always formally defined) is "random".

Namely, the recommendation is that *if we test the program on several "random" sets of data, we will thus be sure that the program works well on "almost all" cases* (in some unspecified sense).

The word "random" brings to mind methods of mathematical statistics; these methods indeed help in formalizing *some* of these recommendations. However, there are some additional features of these recommendations that traditional statistics cannot capture. For example (like in the above case), the more complicated the program, the more tests we need to achieve the same level of confirmation. In contrast, the degree of confidence guaranteed by methods of traditional statistics does not depend on whether the hypotheses are "simple" or not.

The main objective of the present paper. In the present paper, we will show that the known formalizations of the terms "simple" (as having a small Kolmogorov complexity $C(x)$) and "random" (as having Kolmogorov complexity close to the length $l(x)$) make these recommendations mathematically justified.

These results were first announced in [2].

Software testing: brief informal description. Starting with some input x, we must produce an output y. We know *specifications*, i.e., the description of the properties that this y must satisfy. Often, specifications consist of an equation that we are trying to solve (plus, maybe, some additional conditions).

In order to be able to check the correctness of the program, we must be able to check, for any given x and y, whether y satisfies these given specifications. So, we need to have a specification-checking *program* program that checks whether a given y satisfies the specifications for a given x.

For example, if x and y represent real numbers, and the equation that we are trying to solve is $x^2 = y$, then the specification-checking program $s(x,y)$ consists of simply computing x^2 and comparing the result with y.

Definition 1. *By a software testing situation, we mean a pair of programs (p,s), where p transforms binary sequences into binary sequences, and s transforms pairs of binary sequences into "true" or "false". The program p will be called a tested program; the program s will be called a specification-checking program.*

Definition 2. *Let (p,s) be a software testing situation.*

- *We say that the program p satisfies the specifications for an input x if $s(x,p(x)) =$ "true".*
- *We say that the program p satisfies the specifications if it satisfies the specification for all inputs x.*

Denotation. *For a given program (word) p, by $l(p)$, we will denote its length (i.e., the number of bits in the binary description of p).*

Definition 3. *Let $c > 0$ be an integer, and let (p,s) be a software testing situation. We say that, with respect to this situation, an input x is c-simple if $C(x) \le c + l(p) + l(s)$.*

Comment. In other word, an input is simple if its complexity does not exceed the complexity (length) of the tested program + the complexity (length) of the specification checking.

Theorem 1. *There exists a number $c > 0$ with the following property: For every software testing situation (p,s),* if *the program p satisfies specifications for* all c-simple *inputs,* then *it satisfies specifications for* all *possible inputs.*

Comment 1. Theorem 1 says that to check whether a given program p is correct, it is sufficient to check this program only on inputs that are not too complicated (i.e., whose Kolmogorov complexity does not exceed $c + l(p) + l(s)$). The more complicated the program p and/or the specification t, the higher the bound $c + l(p) + l(s)$, and therefore, the more examples we need to test. This Theorem thus explains the above simple software testing heuristic.

Comment 2. For reader's convenience, all the proofs are placed in the special Appendix.

Comment. In formalizing heuristics that use the word "simple", we considered programs that can potentially process inputs of arbitrary length (e.g., sorting programs are like that). For such heuristics, our conclusion was that the program is correct for *all* inputs.

For heuristics that use the word "random", we want to be able to conclude that the program is correct for "almost all" inputs, i.e., to be more precise, for a fraction of the inputs that exceeds a given number $1 - \varepsilon$. To be able to talk about fractions, we must restrict ourselves, e.g., to the case when we only allow inputs of fixed length L.

Definition 4. *Let $C > 0$ be an integer. A word x is called C-random if $C(x) \ge l(x) - C$. We say that $x_1, \ldots, x_k$ is a C-random sequence of k inputs of length L if $l(x_1) = \ldots = l(x_k) = L$ and $C(\vec{x}) \ge l(\vec{x}) - C$, where $\vec{x} = x_1 \ldots x_k$ is a concatenation of the words $x_1, \ldots, x_k$.*

Definition 5. *Let L and C be integers, let $x_1, \ldots, x_k$ be a C-random sequence of k inputs of length L, and let (p,s) be a software testing situation. We say that a program p satisfies specifications for this sequence if $s(x_i, p(x_i))$ is true for all $i = 1, \ldots, k$.*

Definition 6. *Let $\alpha \in (0,1)$ be a real number, let $P(x)$ be a property of binary sequences, and let L be a positive integer. We say that the property $P(x)$ is true for α-almost all sequences of length L if $N_P(L)/N(L) \ge \alpha$, where $N(L)$ is the total number of sequences of length L, and $N_P(L)$ is the total number of sequences of length L that satisfy the property $P(x)$.*

Theorem 2. *There exists a number c with the following property: For every two integers L and C, and for every software testing situations (p,s),* if *the program p satisfies specifications s for* some *C-random sequence of k inputs of length L,* then *the program p satisfies specifications for* α-almost all *inputs x of length L, where $\alpha = 2^{-a(k)}$ and*

$$a(k) = \frac{c + l(p) + l(s) + \log_2(k) + C}{k}.$$

Comment. When $k \to \infty$, we have $a(k) \to 0$, and hence, $\alpha \to 1$. So, the more random inputs we check, the larger the fraction of values x about which we can *conclude* that the program is correct.

This fraction depends on the complexity of the program p and of the specification s: the simpler the program and the specification, the larger the fraction. Thus, for a more complicated program, we must undertake more tests to achieve the same value α (i.e., the same degree of confidence about the correctness of the tested program).

Thus, we have justified the above heuristics that use the word "random".

A word of warning. The main objective of this paper is *justification* of several existing simple software engineering testing heuristics. This formalization makes us confident that these heuristics will work, but it does not help us to implement them: the words "simple" and "random" used in these simple heuristics are formalized, but they are formalized in terms of the notion of Kolmogorov complexity, and Kolmogorov complexity is, in general, *not* algorithmically computable [4].

Acknowledgments. This work was partially supported by NSF Grants No. EEC-9322370, CCR–9211174, and DUE-9750858, by NASA Grants No. NAG 9-757 and NCCW-0089, and by the Future Aerospace Science and Technology Program (FAST) Center for Structural Integrity of Aerospace Systems, effort sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-95-1-0518.

The authors are greatly thankful to Daniel E. Cooke for valuable discussions.

References

[1] R. E. Fairley, *Software Engineering Concepts*, McGraw-Hill Co., N.Y., 1985.

[2] A. Q. Gates, V. Kreinovich, and L. Longpré, "Towards Theoretical Foundations of Software Engineering Heuristics: The Use of Kolmogorov Complexity", *Complexity Conference Abstracts 1996*, June 1996, Abstract No. 96–15, p. 17.

[3] P. C. Jorgensen, *Software testing: a craftsman's approach*, CRC Press, Boca Raton, FL, 1995.

[4] M. Li and P. Vitányi, *An introduction to Kolmogorov complexity and its applications*, Springer-Verlag, N.Y., 1997.

[5] I. Sommerville, *Software Engineering*, Addison-Wesley, Reading, MA, 1996.

Appendix: Proofs

Proof of Theorem 1. To prove Theorem 1, let us fix some ordering $<$ on the set of all possible binary words: e.g., let us order the words by their length, and for a fixed length, lexicographically.

Let us now describe the first x (in this order) for which $\neg s(x, p(x))$; we will denote this first x by f. Computing this f (if it exists at all) can be easily done by a simple `while` loop; inside the loop, we have two calls: for p and for s. Therefore, the length of the resulting program is equal to $c + l(p) + l(s)$, where c is the length of the necessary additional structure (while loop itself, going to the next word in the above-defined ordering, etc.; this additional structure does not depend on s or p).

We are now ready to prove Theorem 1 with this very value of c. Indeed, suppose that we have successfully tested a program p on all inputs x with $C(x) \le c + l(p) + l(s)$. Let us show, by reduction to a contradiction, that $s(x, p(x)) =$"true" for all x. Indeed, suppose that this is not true; this means that there exist words x for which $\neg s(x, p(x))$; therefore, there exists the first word f for which the specification s is not satisfied, i.e., for which $\neg s(f, p(f))$ and $s(x, p(x))$ for all $x < f$. We already know that this first word f can be generated by a program of length $c + l(p) + l(s)$. Since Kolmogorov complexity of a word is defined as the *smallest* length of a program that generates this word, we can conclude that the Kolmogorov complexity $C(x)$ of any word x cannot exceed the length of a program that generates x. In particular, for f, we conclude that $C(f) \le c + l(p) + l(s)$. But by assumption, we have successfully tested the program p on all inputs x of Kolmogorov complexity $\le c + l(p) + l(s)$; therefore, in particular, we have successfully tested the program p on f, thus concluding that $s(f, p(f))$. This conclusion contradicts to our choice of f as the first word for which s is *not* true.

This contradiction proves that our initial assumption – that p is not always correct – is false. Thus, p is always correct. The theorem is proven.

Proof of Theorem 2. We have assumed that our program p satisfied specification for a C-random sequence of k inputs $x_{,} \ldots, x_k$ of length L. Let us prove that in this case, the program p satisfies specifications for a large fraction of words x of length L.

Indeed, let us denote by $S = N_P(L)$ the total amount of words x of length $l(x) = L$ for which the program p satisfies the specifications, i.e., for which $s(x, p(x))$. Then, the desired fraction is equal to the ratio of S to the total number $N(L) = 2^L$ of words of length L: $F = S/2^L$. One can easily write a program that generates the first, the second, ..., the $n-$th, ..., the $S-$th x for which $s(x, p(x))$:

- this program starts with a counter set at 0;
- it generates all words x in the lexicographic order, and for each word x, checks whether $s(x, p(x))$ is "true";
  - if the property $s(x, p(x))$ is true, the program increases the counter by 1,
  - otherwise the program leaves the counter unchanged.
- When the counter reaches the value n, $n-$th number is generated.

This program uses p and s; therefore, the length of this program is $\le c_1 + l(p) + l(s)$ for some constant c_1.

Similarly, one can easily, for any given k, generate all sequences $x_1 \ldots x_k$ for which all components $x_1, \ldots, x_k$ satisfy the given property s. This program calls p and s, and uses k as one of the inputs; therefore, its length is $\le c_1 + l(p) + l(s) + l(k)$, where $l(k) = \lceil \log_2(k) \rceil$ is the length of the binary expansion of k.

If we fix the number n in this program, we get a program with no input that generates the sequence $x_1 \cdots x_k$. The length of this resulting program is equal to the length of the original program + the length of the actual number n that we are substituting instead of the variable n, i.e., this length is $\le c_1 + l(p) + l(s) + l(k) + l(n)$.

The total number of such sequences is S^k, so, $n \le S^k$, hence, $l(n) \le l(S^k)$; thence, the length of this program is $\le c + l(p) + l(s) + l(k) + l(S^k)$. Therefore, the Kolmogorov complexity of each such sequence is $\le c_1 + l(p) + l(s) + l(k) + l(S^k)$.

We have assumed that the program satisfied specifications for a C-random sequence of $\vec{x} = x_1 \ldots x_k$ of k inputs. This means that for $\vec{x}$, we have:

- on one hand $C(\vec{x}) \le c_1 + l(p) + l(s) + l(k) + l(S^k)$ (because the program satisfies the specifications), and,
- on the other hand, $C(\vec{x}) \ge l(\vec{x}) - C = kL - C$ (since $\vec{x}$ is random).

Therefore, we can conclude that

$$c_1 + l(p) + l(s) + l(k) + l(S^k) \ge k \cdot L - C. \tag{1}$$

It is well known that $l(S^k) = \lceil \log_2(S^k) \rceil$ and therefore, $l(S^k) \le \log_2(S^k) + 1 = k \cdot \log_2(S) + 1$. Similarly, $l(k) \le \log_2(k) + 1$. Therefore, from (1), we can conclude that

$$c_1 + l(p) + l(s) + \log_2(k) + 2 + k \cdot \log_2(S) \ge k \cdot L - C. \tag{2}$$

Moving terms linear in k into the right-hand side and all other terms into the left-hand side, we conclude that

$$k \cdot (\log_2(S) - L) \ge -(c_1 + l(p) + l(s) + \log_2(k) + 2) - C. \tag{3}$$

By definition of a fraction $F = S/2^L$, we conclude that $\log_2(F) = \log_2(S) - L$. If we substitute this expression into the left-hand side of (2) and divide both sides by k, we get

$$\log_2(F) \ge -\frac{c_1 + l(p) + l(s) + \log_2(k) + 2 + C}{k}. \tag{4}$$

From (4), we get the desired inequality for $c = c_1 + 2$. The theorem is proven.