Towards Formalization of Feasibility, Randomness, and Commonsense Implication: Kolmogorov Complexity, and the Necessity of Considering (Fuzzy) Degrees

Vladik Kreinovich[1], Luc Longpré[1], and Hung T. Nguyen[2]

[1]Department of Computer Science, University of Texas at El Paso
El Paso, TX 79968, USA, emails {vladik,longpre}@cs.utep.edu

[2]Department of Mathematical Sciences, New Mexico State University
Las Cruces, NM 88003, USA, email hunguyen@nmsu.edu

1 Introduction

In this paper, we show that several seemingly crisp notions, notions which are extremely difficult to formalize, can be easily formalized if we take into consideration that intuitively, these notions are not crisp ("yes-no" notions), but rather allow different degrees.

Understanding commonsense implication. In traditional first order logic, implication "A implies B" is interpreted as "B or not A". As a result, every true statement A implies every other true statement B, and every false statement A implies every other statement. These conclusions contradict to common sense understanding of "if-then", according to which logically correct statement of the type "if 2+2=4, then $E = mc^2$", or "if 2+2=5, then $E \neq mc^2$" make no big sense.

There has been many attempts by logicians to formalize the *commonsense* meaning of the implication, as opposed to the above formal definition; these attempts have not yet led to a universally acceptable and absolutely convincing definition.

In this paper, we will describe a new definition for implication.

Let us recall how we normally understand commonsense implication. When, in mathematics, we say that a statement B *follows* from the statement A (e.g., that B is a *corollary* of A), we mean that it is easier to prove B when we know A than to prove B "from scratch" (i.e., from axioms). This "easier" may mean that the resulting proof is shorter, or than it is easier to find (e.g., for a reasonable automatic theorem prover), etc.

In our formalization, we will mainly consider the case when "easier" means "shorter"; it is easy to modify the resulting definitions to accommodate other notions of easiness. For "easier" as "shorter", "A implies B" means that "there exists a proof of B from A that is shorter than any proof of B from the axioms". Depending on how shorter, we get different *degrees* of implication. Thus, an appropriate formalization of implication (a seemingly crisp notion) involves *degrees* similar to fuzzy logic.

We show that the resulting formalization is related to the notion of *Kolmogorov complexity*, the notion that was originally proposed to describe information and randomness in algorithmic terms.

We hope that this definition will be helpful for formalization of the commonsense reasoning, especially since it is related to a well-developed computer science formalism.

Understanding feasibility. Another seemingly crisp notion that is difficult to formalize is the notion of feasibility:

- some algorithms are feasible, while
- some other algorithms require so much computations time that they are practically non-feasible.

It is difficult to formalize this notion as a crisp one, because it seems like adding one step to a feasible algorithm keeps it feasible, but this would imply that any algorithm is feasible. We show that the introduction of the notion of "degree of feasibility" helps to design a consistent definition [2, 12, 13].

Understanding randomness. Similarly, the explicit formulation of degrees of randomness helps to consistently formalize this important notion as well.

2 Understanding commonsense implication

Formulation of the problem. In traditional first order logic, implication $A \rightarrow B$ (meaning "A implies B") is interpreted as $B \vee \neg A$. As a result, every true statement A implies every other true statement B, and every false statement A implies every other statement. These conclusions contradict to common sense understanding of "if–then", according to which logically correct statement of the type "if $2+2=4$, then $E=mc^2$", or "if $2+2=5$, then $E \neq mc^2$" make no big sense.

There has been many attempts by logicians to formalize the *commonsense* meaning of the implication, as opposed to the above formal definition (see, e.g., [3, 1]); these attempts have not yet led to a universally acceptable and absolutely convincing definition.

In this short paper, we will describe a new definition and show how it is related to the notion of *Kolmogorov complexity*, the notion that was originally proposed to describe information and randomness in algorithmic terms (for a detailed description of Kolmogorov complexity, see, e.g., [8]).

We hope that this definition will be helpful for formalization of the commonsense reasoning, especially since it is related to a well-developed computer science formalism.

What "implies" means in mathematics? An idea. Let us recall how we normally understand commonsense implication.

- When, in *mathematics*, we say that a statement B *follows* from the statement A (e.g., that B is a *corollary* of A), we mean that it is easier to *prove* B when we know A than to prove B "from scratch" (i.e., from axioms).
- Similarly, in *commonsense* reasoning, by saying that A implies B we mean that it is easier to *argue* for B if we already assume A than to argue for B without this assumption. In other words, it is, in essence, the same notion, except for using *arguments* (informal proofs) instead of mathematically formal proofs.

This "easier" may mean that the resulting proof is shorter, or than it is easier to find (e.g., for a reasonable automatic theorem prover), etc.

In our formalization, we will mainly consider the case when "easier" means "shorter"; it is easy to modify the resulting definitions to accommodate other notions of easiness. For "easier" as "shorter", "A implies B" means that "*there exists a proof of B from A that is shorter than any proof of B from the axioms*".

Towards a formalization of the above idea. To avoid a logical-type" formulation (with a universal quantifier "any"), we can take into consideration that "is shorter than any proof" is equivalent to "is shorter than the shortest possible proof". Similarly, in this context, "there exists a proof of B from A that is shorter than ..." can be replaced with "the shortest possible proof of B from A is shorter than ...". After these two replacement, the commonsense meaning of "A implies B" can be reformulated as: "*the shortest possible proof of B from A is shorter than the shortest possible proof of B*". If we denote the length of the shortest possible proof of B by $S(B)$, and the length of the shortest possible proof of B, given A, as $S(B|A)$, then we can be reformulated commonsense implication as an inequality $S(B|A) < S(B)$.

Degree of implication.

- If $S(B|A) \ll S(B)$, then we are very confident that A implies B;
- if $S(B|A) \approx S(B)$, then we are not that confident about it.

Therefore, we can take the difference $S(B) - S(B|A)$ as the *degree* to which A implies B.

Formalization. To complete this description, we must describe what $S(B)$ and $S(B \,|\, A)$ mean. The value $S(B)$ was defined as the shortest proof of B, i.e., as the shortest length of a text p that is a proof and that proves B.

For each formal system, checking whether a given text p is a proof is easy: for each statement in this proof, if this statement is claimed to be an axiom, we simply check whether it indeed is the right axiom; if it is claimed that this statement is obtained from the previously proven statements according to one of the legitimate rules, then we just need to check that it indeed follows. Deciding whether a proof results in B is also simple: it is sufficient to look at the last statement of the proof. As a result, we have a very simple (usually, quadratic-time) algorithm (we will denote this algorithm by f) that takes a given text p and returns either a statement "this text is not a proof", or a statement B that is actually proved by this proof. In terms of this algorithm f, a given text p is a proof of B if and only if $f(p) = B$. So, we can reformulate the definition of $S(B)$ as

$$S(B) = \min\{l(p) : f(p) = B\}, \qquad (1)$$

where $l(p)$ stands for the length of the word p.

Similarly, it is easy to check whether a given text p is a proof of a statement B on the assumption that A is already known to be true (i.e., in effect, that we can use A as a new axiom). Let us denote by ϕ an algorithm that, given a pair consisting of a text p and a statement A, returns returns either a statement "this text is not a proof", or a statement B that is actually proved (given

A) by this proof. In terms of this algorithm f, p is a proof of B using A if and only if $\phi(p, A) = B$. So, we can reformulate the definition of $S(B|A)$ as

$$S(B|A) = \min\{l(p) : \phi(p, A) = B\}. \qquad (2)$$

For an empty statement A, $\phi(p, A)$ coincides with the algorithm $f(A)$ defined above.

Relation to complexity of discrete objects. The right-hand side of the formula (1) is known as *the complexity of object B with respect to the specifying method f* (see, e.g., [8], Chapter 2), and it is denoted by $C_f(B)$. According to [8]: "In computer science terminology, we would say that p is a program and f is a computer, so that $C_f(x)$ is the minimal length of a program for f (without additional input) to compute output x."

Similarly, the right-hand side of the formula (2) is known as *the complexity of B conditional to A* ([8], Section 2.1); in computer science terms, it means the minimal length of a program p that, given A, computes B; it is usually denoted by $C_\phi(B|A)$.

In these terms, "A implies B" is equivalent to $C_\phi(B|A) < C_f(B)$, i.e., to $C_f(B) - C_\phi(B|A) > 0$, and the difference $C_f(B) - C_\phi(B|A) > 0$ can be viewed as a degree to which A implies B.

Relation to Kolmogorov complexity. The most well known case of this definition is when we take as f a *universal computer* (in some reasonable sense of this word), i.e., a computer with "the best possible" computing properties. For an arbitrary f, the complexity $C_f(x)$ characterizes not only the complexity of x itself, but it also describes the ability of f to capture it: e.g., x can be a simple word, but a computer f may be so limited that it cannot compute x by a short program. For a *universal* computer f, the complexity $C_f(x)$ is *not* bounded by any drawbacks of the computer and therefore, characterizes the *complexity* of the word x. This notion was introduced independently by Kolmogorov, Solomonoff, and Chaitin; it is usually called the *Kolmogorov complexity* $C(x)$ of the word x.

Similarly, in case ϕ is a universal computer, the value $C_\phi(x|y)$ characterizes the complexity of x with respect to the object y, and it is called a *conditional Kolmogorov complexity* of x with respect to y.

Information. For universal computers, the difference $C_f(x) - C_\phi(x|y)$ that describes how using y makes computing x easier, is called the *algorithmic information about x contained in y*, and it is denoted by $I(y : x)$. This notion is in good agreement with the information defined in traditional mathematical statistics and engineering information theory.

Using this definition as an analogy, we can call the difference $C_f(B) - C_\phi(B|A)$ the *information* about B contained in A, and denote it by $I_\phi(A : B)$. In terms of this newly defined "information" I_ϕ, we can define "A implies B" as $I_\phi(A : B) > 0$, and $I_\phi(A : B)$ is the degree to which A implies B.

Information is asymmetric since we take resource boundedness into consideration. In our definitions, we used fast (quadratic-time) algorithms f and ϕ, and, since we restrict ourselves with resource-bounded analogues of Kolmogorov complexity, we end up with the *non-symmetric* information ([10]; [9]; [8], Chapter 7).

This allows us to avoid the problem which could occur if we used the original Kolmogorov complexity, in which asymptotically, the information is symmetric $I(A : B) \sim I(B : A)$, and therefore, we would have gotten a counterintuitive conclusion that A implies B iff B implies A.

3 Understanding feasibility

Introduction. It is well known that not all algorithms are realistic (see, e.g., [7], Section 7.1). If an algorithm requires, say, 2^{2^n} computational steps for an input of length n, then for realistic n (e.g., $n = 100$) this number of steps will exceed the lifetime of the Universe (according to modern cosmology). So if we are interested in separating purely theoretical algorithms from the ones that can be actually run on the computers (existing or future ones), we must somehow formalize the notion of feasibility.

The most widely used formalization of this notion is that feasible algorithms are exactly the ones that are time-polynomial, i.e., the ones for which the running time is limited by some polynomial $P(n)$ of the input length n (see, e.g., [7], Section 7.4; [11], Ch. 23). There exist formal systems of reasonable axioms that justify this choice (see, e.g., [14]).

However, the majority of the researchers agree that this is not the precise description of a feasible algorithm, because some time-polynomial algorithms are evidently not feasible. For example, an algorithm that takes $10^{10^{10}} n$ time to compute is time-polynomial (even linear-time) but it can hardly be called feasible: even for $n = 1$ it requires the computation time that is exponentially bigger than the lifetime of the Universe.

So the *problem* is: *how to get a better formalization of the notion of a feasible algorithm?*

L. Zadeh remarked that this problem may be caused by the fact that we are trying to describe feasibility as a *crisp notion*, according to which an algorithm is either feasible or not. In reality, it is natural to distinguish be-

tween different *degrees* of feasibility: e.g., a linear-time algorithm is clearly more feasible than a quadratic-time one. So, instead of saying that an algorithm is feasible, we would like to say that an algorithm is feasible to a certain extent. In other words, feasibility is a *fuzzy* notion.

In this section, we formalize this idea, and show that it leads to the solution of the above-mentioned problem:

- Our formalization will explain why feasible algorithms are time-polynomial.
- And, at the same time, in our formalization, it will be not true that all time-polynomial algorithms are feasible.

Motivations of the following definitions. When we say that an algorithm is feasible, we mean that for every input of reasonable length, the running time of this algorithm is also reasonable. Therefore, to decide whether an algorithm $\mathcal{U}$ is feasible or not (and to what extent this algorithm is feasible), we must know how fast the function $t_{\mathcal{U}}(n)$ (that describes the largest running time of $\mathcal{U}$ on all inputs of length n) grows. Let us say that a function $f(n)$ from integers to integers is *feasible* if it represents the running time of a feasible algorithm. In these terms, the above statement can be reformulated as follows: $F(f)$ is equivalent to

$$\forall n(R(n) \to R(f(n)), \tag{3}$$

where $F(f)$ means "f is feasible", and $R(n)$ means "n is a reasonable (i.e., sufficiently small) length" (or "n time units is a reasonable time").

In view of this equivalence, to describe what is feasible, we must describe what is reasonable. There are several natural properties of the notion "reasonable":

- First, if n is reasonable (i.e., sufficiently small), and m is even smaller, then m is also a reasonable length:

$$\forall n, m((R(n) \,\&\, (m < n)) \to R(m)). \tag{4}$$

- Second, if we have two feasible algorithms, and we run the first one, and then immediately after that, run the second one, then the resulting composite algorithm is also feasible. For this composite algorithm, the running time is equal to the sum of the running times of the first two. So, we can conclude that if n and m are reasonable times, then their sum $m + n$ is also a reasonable time:

$$\forall m \forall n((R(m) \,\&\, R(n)) \to R(m + n)). \tag{5}$$

  In particular, for $m = n$, we conclude that

$$\forall n(R(n) \to R(2n)). \tag{6}$$

- Third, if we have an algorithm that calls a certain subroutine a reasonable number of times, and this subroutine is feasible itself, then the resulting composite algorithm is also feasible. If we denote the number of calls by m, and the running time of the subroutine by n, then the running time of the composite algorithm (that consists of m calls of n time units each) is $m \cdot n$. So, the above property means that if m and n are both reasonable, then their product is also reasonable:

$$\forall m \forall n((R(m) \,\&\, R(n)) \to R(m \cdot n)). \tag{7}$$

  In particular, for $m = n$, we conclude that

$$\forall n(R(n) \to R(n^2)). \tag{8}$$

How can we describe these statements in fuzzy terms? We want to describe the properties of a predicate $R(n)$. Since we are going to use fuzzy logic, it is natural to describe $R(n)$ as a *fuzzy predicate*, i.e., as a function μ_R that transforms an integer n into the number from the interval [0,1] ($\mu_R(n)$ is our degree of belief that n is a reasonable length and/or time). To have a fuzzy descriptions of conditions (4) through (8), we need to choose fuzzy analogues of $\&$, $\to$, and $\forall n$. Since our goal is to show the potential possibility of to describe feasibility in fuzzy terms, we will choose the simplest possible fuzzy analogues of these logical operations. For $\&$, the simplest operations described in the original paper of Zadeh [16] were minimum and algebraic product. Minimum does not work here (see, e.g., [12, 13]), so, we will use the algebraic product.

The next choice is to describe $\forall n$ and $\to$ in fuzzy terms. Namely, suppose that we have a formula F of the type $\forall n(A(n) \to B(n))$. Suppose also that for all n, we know the degrees of belief $\mu(A(n))$ and $\mu(B(n))$ in $A(n)$ and $B(n)$. What is the resulting degree of belief $\mu(F)$ in a formula F? The fact that F is true means that for every n, from $A(n)$, we can conclude $B(n)$. In other words, from $F \& A(n)$, we can conclude $B(n)$. Therefore, for every n, the degree of belief in $B(n)$ is at least as large as the degree of belief in $F \& A(n)$. Since we have chosen algebraic product as a fuzzy analogue of $\&$, we can conclude that $\mu(B(n)) \geq \mu(F) \cdot \mu(A(n))$. Therefore, it is natural to define $\mu(F)$ as the largest real number for which this inequality is true for all n. This inequality can be rewritten as $\mu(F) \leq \min(\mu(B(n))/\mu(A(n)), 1)$ ($\min(\ldots, 1)$ because $\mu(F) \leq 1$ even if $\mu(B(n)) > \mu(A(n))$), and therefore, this definition can be reformulated as $\inf[\min(\mu(B(n))/\mu(A(n)), 1)]$, where inf is taken over all non-negative integers n.

If we have a simpler universal formula F of the type $\forall n A(n)$, then this formula F simply implies $A(n)$ for all n. Therefore, we must have $\mu(A(n)) \geq \mu(F)$ for all n. Hence, as $\mu(F)$, we can take the largest real number that satisfies all these inequalities: namely, $\mu(F) = \inf(\mu(A(n))$.

Similar definitions can be given for the case when we have two universal quantifiers in a formula. Now, we are ready for formal definitions.

Definition 1. *Let $R(n)$ be a fuzzy predicate, i.e., a function $\mu_R(n)$ from the set N of all non-negative integers to the interval [0,1]. We will define degrees of belief in different statements as follows:*

- *For every integer n, by a degree of belief $\mu(R(n))$ in $R(n)$, we mean the number $\mu_R(n)$.*
- *For a crisp formula F, by a degree of belief $\mu(F)$ in F, we mean 1 if this formula is true, and 0 is this formula is false.*
- *If we already know the degrees of belief $\mu(A)$ and $\mu(B)$ in formulas A and B, then the degree of belief in $A\&B$ is defined as*

$$\mu(A\&B) = \mu(A) \cdot \mu(B).$$

- *If for every n, we already know the degree of belief $\mu(A(n))$ in a formula $A(n)$, then the degree of belief $\mu(F)$ in a formula F of the type $\forall n A(n)$ is defined as* $\inf\ \mu(A(n))$.
- *If for every n, we already know the degrees of belief $\mu(A(n))$ and $\mu(B(n))$ in formulas $A(n)$ and $B(n)$, then the degree of belief $\mu(F)$ in a formula F of the type $\forall n(A(n) \to B(n))$ is defined as* $\inf\,[\min(\mu(B(n))/\mu(A(n)), 1)]$.

Definition 2.

- *We say that a formula F is possibly true if $\mu(F) > 0$.*
- *We say that a fuzzy predicate R is a possible description of realism if for this predicate, formulas (2)–(6) are possibly true.*
- *We say that a function f from integers to integers is possibly feasible if for every possible description of realism, formula (3) is possibly true.*

Theorem 1. [12, 13] *For every function $f(n)$, the following two conditions are equivalent to each other:*

- *f is possibly feasible;*
- *there exists a polynomial P such that $f(n) \le P(n)$ for all n.*

This theorem means that every feasible function is time-polynomial. In other words, this result explains why feasible functions are currently associated with time-polynomial ones.

For this definition, it *is* true that every polynomially bound function is possible feasible. This makes sense, because in our definitions, we did not assume anything about our physical world. So, e.g., the function $10^{10^{10}}n$ that is not feasible in our world, is feasible in some hypothetic world of many dimensions. However, we would like to describe the fact that although every polynomial is possibly feasible in some world, but in every world, some polynomials are still not feasible. The statement that all polynomial-time algorithms are feasible is naturally formalized as $\forall A, k(F(An^k))$, where $F(f)$ is interpreted as (3). For this statement, the following is true:

Theorem 2. [12, 13] *It is not possibly true that every polynomial is feasible.*

Comment. It is therefore not possibly true that every time-polynomial algorithm is feasible; this is exactly what we wanted to prove.

4 Understanding randomness

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., [4, 6, 15]).

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 [5].

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 3. *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 4. *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 5. *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 3. *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 3 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 last section.

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 6. *Let $C > 0$ be an integer. A word x is called C-random if $C(x) \geq l(x) - C$. 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 $C(\vec{x}) \geq l(\vec{x}) - C$, where $\vec{x} = x_1 \ldots x_k$ is a concatenation of the words $x_1, \ldots, x_k$.*

Definition 7. *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 8. *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 $N_P(L)/N(L) \geq \alpha$, 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 4. *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 section 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 [8].

5 Proofs

Proof of Theorem 3. To prove Theorem 3, 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 3 with this very value of c. Indeed, suppose that we have successfully tested a program p on all inputs x with $C(x) \leq c + l(p) + l(s)$. 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 $C(f) \leq c + l(p) + l(s)$. But by assumption, we have successfully tested the program p on all inputs x of Kolmogorov complexity $\leq c + l(p) + l(s)$; 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 4. We have assumed that our program p satisfied specification for a C-random sequence of k inputs $x_1, \ldots, x_k$ 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

$$c_1 + l(p) + l(s) + l(k) + l(S^k) \ge k \cdot L - C. \quad (9)$$

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 (9), we can conclude that

$$c_1 + l(p) + l(s) + \log_2(k) + 2 + k \cdot \log_2(S) \ge k \cdot L - C. \quad (10)$$

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

$$k \cdot (\log_2(S) - L) \ge -(c_1 + l(p) + l(s) + \log_2(k) + 2) - C. \quad (11)$$

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 (10) and divide both sides by k, we get

$$\log_2(F) \ge -\frac{c_1 + l(p) + l(s) + \log_2(k) + 2 + C}{k}. \quad (12)$$

From (12), we get the desired inequality for $c = c_1 + 2$. The theorem is proven.

Acknowledgments. This work was supported in part by NASA under cooperative agreement NCC5-209, by NSF grant No. DUE-9750858, by the United Space Alliance grant No. NAS 9-20000 (PWO C0C67713A6), 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.

References

[1] J. P. Cleave, *A study of logics*, Claredon Press, Oxford, 1991.

[2] D. E. Cooke, V. Kreinovich, and L. Longpré, "Which algorithms are feasible? MaxEnt approach", In: G. Erickson (ed.), *Maximum Entropy and Bayesian Methods*, Kluwer, Dordrecht, 1998 (to appear).

[3] *Ifs: conditionals, beliefs, decision, chance, and time*, D. Reidel Co., Dordrecht, Holland, 1981.

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

[5] 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.

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

[7] H. R. Lewis, C. H. Papadimitrou, *Elements of the Theory of Computations*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

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

[9] L. Longpré and S. Mocas, "Symmetry of information and one-way functions", *Inform. Proc. letters*, 1993, Vol. 46, No. 2, pp. 95–100.

[10] L. Longpreé and O. Watanabe, "On symmetry of information and polynomial time invertibility", In: *Proc. 3rd Int'l. Symp. Alg. Comput.*, Springer Lecture Notes in Computer Science, Vol. 650, Springer, N.Y., 1992, pp. 410–419.

[11] J. C. Martin. *Introduction to Languages and the Theory of Computation*, McGraw-Hill, N.Y., 1991.

[12] H. T. Nguyen and V. Kreinovich, "When is an algorithm feasible? Soft computing approach", *Proceedings of the Joint 4th IEEE Conference on Fuzzy Systems and 2nd IFES*, Yokohama, Japan, March 20–24, 1995, Vol. IV, pp. 2109–2112.

[13] H. T. Nguyen and V. Kreinovich, "Towards theoretical foundations of soft computing applications", *International Journal on Uncertainty, Fuzziness, and Knowledge-Based Systems*, 1995, Vol. 3, No. 3, pp. 341–373.

[14] V. Yu. Sazonov, "A logical approach to the problem 'P=NP?' ", *Lecture Notes in Computer Science*, Vol. 88, Springer-Verlag, N.Y., 1980, pp. 562–575.

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

[16] L. Zadeh, "Fuzzy sets", *Information and control*, 1965, Vol. 8, pp. 338–353.