INTERVAL APPROACH TO TESTING SOFTWARE

Vladik Kreinovich[1], Thomas Swenson[1], Alex Elentukh[2]
[1]Computer Science Department, University of Texas at El Paso, El Paso, TX 79968
[2]Mail Stop C3-10, Codex, Motorola, Mansfield, MA 02048, USA

Abstract. For a sufficiently simple program it is possible to eliminate all the errors. Moreover, there exist methods (called "proving program correctness") that can guarantee that all the errors has indeed been removed and the resulting program is "flawless". However, the experience of many programmers shows that it is actually impossible to extract all the faults from system-type software, that is commonly involved in resource contention, or from the programs with a sophisticated user interface. If after some debugging such a program works fine, it simply means that we have not yet reached the point where it will start erring. So we are unable to debug such a program completely; moreover we are sure that sooner or later the program will err. Therefore it is necessary to estimate the time interval during which the remaining faults will not influence the program.

At present about three dozen statistical models (called *software reliability models*) are used to get such estimates. The most widely accepted models are basic (exponential) and the logarithmic Poisson models proposed by Musa, Iannino, and Okumoto. In many situations these models proved to be a good fit. However, these statistical models lack convincing theoretical explanation. They are semi-heuristic, and often look like curve-fitting.

In this paper, we consider an interval approach to program testing. We formulate the problem of choosing the best interval software reliability model as a mathematical optimization problem, and solve this problem. As a result, we explain Musa's experiments.

1. INTRODUCTION

It is possible to eliminate all the errors from a small simple program. For a sufficiently *simple* program it is possible to eliminate all the errors.

Moreover, there exist methods (called "proving program correctness") that can guarantee that all the errors has indeed been removed and the resulting program is "flawless". These methods were initiated by the pioneering works of McCarthy [McCarthy 1963a, b] and became widely applicable staring from the breakthrough paper [Hoare 1969] (for recent surveys on program correctness see [Bakker 1980], [Cousot 1990] and [Clarke Kurshan 1991]). Alas, these methods do not work for more complicated programs.

For more complicated programs it is difficult to decide when to stop testing. For *complicated* programs it is much more difficult to decide when to stop testing, since it is extremely difficult (and often practically impossible) to prove formally the correctness of such a program. A reasonable idea is to stop testing when several tests in a row do not discover any more bugs, but to rely on such a criterion would be sometimes misleading. It is often practically impossible to perform a fully exhaustive test, and therefore failures often occur during the customer's usage, long after the product has been tested and released. Moreover, the experience of many programmers shows that it is actually impossible to

extract all the faults from system-type software, that is commonly involved in resource contention, or from the programs with a sophisticated user interface. So if such a program works fine it simply means that we have not yet reached the point where it will start erring. For example, Robert L. Glass in his essay *"Some thoughts on software errors"* [Glass 1991, pp. 33-36] summarizes his *"over 35 years of experience in the industrial and academic worlds of software engineering"*, concludes that *"not all software errors are found EVER"* and so *"we cannot remove all errors from software"*. And he is not alone in this opinion: this viewpoint is widely spread in the literature (see, e.g., [Farr 1983]) and is also shared by the users who still find bugs in the old compilers, operating systems and other software.

The only thing that we can do is to estimate the time before a next failure. If we take seriously this wide-spread opinion that it is impossible to eliminate all the errors, then when do we stop testing? Whenever we stop there will be some bugs remaining, so we can never guarantee that the released product will always work fine. The only thing that we can try to guarantee is that it will work fine for some given period of time. So in order to decide when to stop testing we must fix some desired time interval and stop testing when the program is guaranteed to work fine during this time interval. In order to do this we must be able to estimate the time interval during which the remaining errors will not influence the program.

Existing methods: basic idea. Suppose that we are on some stage of a debugging process, and we need to know when to expect the next failure, i.e., the next occasion when the program will not function properly. If we keep documenting the debugging process, then we have the moments of time $t(1), t(2), ..., t(m)$, when the first, second, ..., m-th bug were discovered. Therefore we know what would have happened if we stopped debugging one bug earlier: the interval $t(m) - t(m-1)$ between that previous bug and the last bug is the time during which the program would have worked fine. Likewise we can easily compute the times during which the program would have worked fine if we stopped debugging two, three, etc bugs earlier: these times is correspondingly equal to

$$t(m-1) - t(m-2),\ t(m-2) - t(m-3), ...$$

So we have a sequence of time intervals during which the program would work fine if we stopped testing after 1-st, 2-nd, ..., $(m-1)$-st failures.

So to estimate the desired value $t(m+1) - t(m)$ of the time interval before the next failure, we must apply some extrapolation procedure to this sequence. In order to be able to extrapolate we need a model $f(n)$ describing how often the bugs appear. Such models are called *software reliability models.*

When a model $f(n, \vec{C})$ with parameters $\vec{C} = (C_1, ..., C_p)$ is chosen, we can predict the time $t(m+1) - t(m)$ before the next failure as follows: First, we find the values of the parameters $\vec{C}$ for which $f(1, \vec{C}) \approx t(1), f(2, \vec{C}) \approx t(2), ..., f(m, \vec{C}) \approx t(m)$. Then, we use $f(m+1, \vec{C}) - f(m, \vec{C})$ for this $\vec{C}$, as a prediction of the time $t(m+1) - t(m)$ during which the program will work fine. If this predicted interval exceeds the specified time T_0, then we can stop testing.

What model $f(n, \vec{C})$ to use?

Existing models of software reliability: briefly. Nowadays many statistical models of software reliability help to obtain time estimates; see, e.g., a survey in [Musa et al 1987]. This book contains also the results of experimental comparison of different models. The authors of this experimental study used different criteria for their choice: computational simplicity, capability, predictive validity, etc. It turned out (see Section 13.5 of [Musa et al 1987]) that with respect to all these natural criteria two models are superior: so-called basic Poisson execution time model (proposed first in [Musa 1975]) and the logarithmic Poisson execution time model proposed first in [Musa Okumoto 1984]. In both models failures occur as a random process (namely, nonhomogeneous Poisson process). In the basic model failure intensity λ decreases linearly with the number of experienced failures μ. So in due time the failure intensity becomes equal to 0; this means that all bugs have been found and no more failures will occur. In the logarithmic model failure intensity decreases exponentially (as $\lambda_0 \exp(-\theta\mu)$). The expected total number of failures increases with execution time t as $c(1 - \exp(-kt))$ for the basic model and as $a \ln(1 + bt)$ for the logarithmic model (hence the name of this model), where a, b, c, k are constants.

We can "invert" this dependency and get the dependency of the average time $\bar{t}(n)$ of n-th failure on n: In the basic model it is $\bar{t}(n) = c \ln(N - n) + k$, where c, k are constants and N is a total number of errors in the initial program. In the logarithmic model it is $\bar{t}(n) = c(\exp(kn) - 1)$, where c, k are constants.

Comments.
1. *Why did we choose Musa's experiments and not any of the competing ones?* After the experimental comparison of various models, whose results are given in [Musa et al 1987], several others researchers undertook likewise comparisons. Some of them came out with different results, namely that in their experiments some other methods turned out to be better than both the basic and the logarithmic models. Actually on practically every conference several papers with competing experiments appear (see, e.g., [Proceedings 1991]). However, to the best of our knowledge, the amount of experimental data and varieties of data covered by [Musa et al 1987], is still greater than by any of the competing experiments, therefore we used Musa's results as the most general ones. These models have also been recommended in [Seifert 1990], [Seifert Stark 1992].

2. The above-described approach gives the reliability estimates that do not take into consideration the environment in which the analyzed software will be used. Estimates that take environment into consideration have been proposed in [Elentukh Elbert 1991], [Elentukh et al 1991], [Musa 1993].

The main problem with the existing statistical models is that their foundations are not very convincing: at most, they prove that these models form a reasonable first approximation. So, we need new (theoretically more plausible) foundations for program testing.

What we are going to do. There have already been cases when statistical models lacked sufficient theoretical foundations and were thus unreliable: this problem surfaced in the

early 60s in error estimation. To deal with the situations when probabilities are not known, interval analysis was invented (see, e.g., [Moore 1979]).

In this paper, we follow the same pattern: namely, we propose an *interval* software reliability model. We also formulate the problem of choosing the "best" interval software reliability model (for approximating the expected time of $n-$th failure) as an optimization problem, and solve this problem.

As a result, we explain why existing models work fine.

Comments.
1. The basic underlying ideas and preliminary results were published in [Kreinovich 1985, Kreinovich et al 1986, Finkelstein et al 1988, Kreinovich 1988, Finkelstein et al 1989, Kreinovich 1989, Kreinovich 1990, Kreinovich et al 1991, Kreinovich et al 1993].

2. We want to warn the readers that the main goal of this paper is to propose new ideas, and not the solution that would always work. In spite of the visible progress, we have not yet achieved a complete solution to the problem of testing software. Namely, in this paper, we consider an "optimistic case", when all the bugs are eventually discovered, and as a result, the program runs better and better, and fewer and fewer failures occur. In this case, the main problem is when to stop testing software. However, every programmer working in a real world knows that in many practical cases, the situation is not that rosy:

- In some cases, in spite of all the debugging efforts, failures continue to occur with a non-decreasing rate. In this case, the problem is not when to stop, but how to change debugging so that the program will start working better (and is this program worth working at at all).
- Cases like the above ones when everything goes wrong are not the most difficult ones: in these cases, we always have an option of throwing the program into the wastebasket and starting anew. An even more complicated situation occurs when the debugging process so to say "goes wild": for some time interval, there are no bugs at all, then suddenly, lots of bugs are discovered. Then again everything seems fine and the programmers think that they are done, but no, bugs start coming back. In this case, if we try to cover the rate with which the bugs are discovered by a reasonable interval function, we get nothing better than an interval $[0, \infty)$.

Our approach is far from being final. We have just criticized the existing statistical models for lacking convincing foundations. We believe (and we will try to convince the readers) that the foundations for our interval models are *more convincing*. However, we agree with the anonymous referees that our foundations are still *not absolutely convincing*. Again, our main goal is *not* to propose the final *results*, but to propose a new *approach*.

Auxiliary (minor) problem: how to count the errors. There are several aspects of this problem.

i) First, do we count the bugs discovered before we initially executed a program on a computer? Some programmers prefer to scrutinize the code first and find lots of errors before touching a terminal. Other designers type the raw code in and start debugging

on the computer. Since the usual statistical procedures use only the bugs that were discovered during the computer run, these procedures will underestimate the total number of bugs for the designers who do some preliminary code analysis and thus end up with the biased predictions of the program's reliability.

This problem is not easy to deal with, because it is difficult to force a designer to write down all the errors he revealed in his mind, (before touching the computer). Moreover, even if we succeed in forcing him to do it, this additional activity will seriously slow him down and will certainly badly interfere with his ability of creative thinking.

So what we need is not an administrative solution to this problem, but some kind of a mathematical solution that will somehow take care of these possible additional revealed bugs without actually counting them (i.e., in some indirect way).

ii) Second, how do we count bugs: do we take one failure for one bug or count the number of lines that demanded a change? Or somehow else take into consideration the fact that, citing [Glass 1991] again, *"errors are not equal"*. If we count just failures, then we can count them automatically, but we lose some essential part of the information. This problem is discussed in detail in Section 9.1.4 of [Musa et al 1987]. However, we do not have the feeling that this problem has been completely and satisfactorily solved.

2. INTERVAL APPROACH

2.1. Motivations of the following definitions

Basic idea. We will not try to describe a random process $t(n)$. Instead, we will try to find an interval function $\mathbf{f}(n) = [f^-(n), f^+(n)]$ that will describe the actual debugging process, i.e., for which $t(n) \in \mathbf{f}(n)$ for all n.

So, we must somehow choose a family of interval functions $\mathbf{f}(n, \vec{C})$. As soon as a family is chosen, the answer to the question "when to stop testing?" is as follows:

- first, based on the known values $t(1), ..., t(m)$, we find the values of the parameters $\vec{C}$ for which $t(n) \in \mathbf{f}(n, \vec{C})$ for all n;
- for this $\vec{C}$, we produce $f^-(m+1, \vec{C}) - t(m)$ as the guaranteed interval during which the software will function properly. If this interval exceeds the desired value T_0, then we stop testing, else, we continue testing. In case we continue testing, we can estimate the necessary additional testing time as the time $f^+(N, \vec{C}) - t(m)$, where N is the first integer for which $f^-(N+1, \vec{C}) - f^+(N, \vec{C}) \geq T_0$.

Comment. In the following, bold-face type will denote intervals.

We must choose a family of interval functions. Before we start discussing what is the best choice (and what do we mean by "the best") let's first make the following remark (that will later on prove to be helpful). Suppose that an interval function $\mathbf{f}(n) = [f^-(n), f^+(n)]$ describes the results $t(n)$ of some actual debugging process performed by an actual programmer (i.e., $t(n) \in \mathbf{f}(n)$ for all n). As Robert Glass remarks in [Glass 1991], *"not all software error finders are equal"*. The differences range in magnitude up

to 30 : 1. So for another programmer, who is c times faster, $c\mathbf{f}(n)$ will be a good fit. For the third programmer, who started the actual debugging process later, a good fit will be $c\mathbf{f}(n) + a$, where c is his relative rate of debugging and a is the moment of time when he started debugging. In view of that it is reasonable to demand that if a function $\mathbf{f}(n)$ is a good fit (i.e., belongs to a family of functions $\mathbf{f}(n, \vec{C})$), then all its linear transformations $a + b\mathbf{f}(n)$ must also belong to such a family.

This family of functions must be at least 2-dimensional (i.e., we need to fix the values of at least two parameters to choose a function from this family). So, the smallest possible number of parameters is two, when the family consists of the functions $a + b\mathbf{f}_0(n)$ for different a, b and some fixed interval function $\mathbf{f}_0(n)$.

What is a criterion for choosing a family of functions? What does it mean to choose a *best* family of functions? (in answering this question we follow the general idea outlined in [Kreinovich 1990] and [Kreinovich et al 1993]). It means that we have some criterion that enables us to choose between the two families. This criterion can be computational simplicity, predictions ability as in [Musa et al 1987] or something else. The most frequent criteria are numeric ones, when to every family we assign some value expressing its performance, and choose a family for which this value is maximal. However, it is not necessary to restrict ourselves to such numeric criteria only. E.g., if we have several different families that have the same prediction ability P, we can choose between them the one that has the minimal computational complexity C. In this case the actual criterion that we use to compare two families is not numeric, but more complicated: a family F_1 is better than the family F_2 if and only if either $P(F_1) > P(F_2)$, or $P(F_1) = P(F_2)$ and $C(F_1) < C(F_2)$. A criterion can be even more complicated. The only thing that a criterion *must* do is to allow us for every pair of families to tell whether the first family is better with respect to this criterion (we'll denote it by $F_1 > F_2$), or the second is better ($F_1 < F_2$) or with respect to this criterion these families have the same quality (we'll denote it by $F_1 \sim F_2$). Of course, it is necessary to demand that these choices be consistent. For example, if $F_1 > F_2$ and $F_2 > F_3$ then $F_1 > F_3$.

The criterion must be final, i.e., it must pick the unique family as the best one. A natural demand is that this criterion must choose a *unique* optimal family (i.e., a family that is better with respect to this criterion than any other family). The reason for this demand is very simple. If a criterion does not choose any family at all, then it is of no use. If several different families are the best according to this criterion, then we still have a problem to choose among those best. Therefore we need some additional criterion for that choice, like in the above example: if several families turn out to have the same prediction ability, we can choose among them a family with minimal computational complexity. So what we actually do in this case is abandon that criterion for which there were several "best" families, and consider a new "composite" criterion instead: F_1 is better than F_2 according to this new criterion if either it was better according to the old criterion, or they had the same quality according to the old criterion and F_1 is better than F_2 according to the additional criterion.

In other words, if a criterion does not allow us to choose a unique best family it means

that this criterion is not final, we'll have to modify it until we come to a final criterion that will have that property.

This criterion must not depend on the (unknown) number of errors that we revealed before typing the program in. The next natural demand on the criterion is connected with the above-mentioned uncertainty in counting the number of errors. Suppose that we have two programmers who have the same abilities to debug (so they reveal the same errors in the same order), but the first programmer discovers the first 20 faults in his mind and only then starts testing the program on the computer, while the second programmer starts executing his program from the very beginning. Then what is the error number 3 for the first programmer, for the second programmer it will be the error number 23, because the second programmer also counts the 20 errors that the first programmer discovered before he started computer testing.

On the other hand, an error number 4 for the second programmer was discovered by the first programmer before he went to the computer and started computer testing. If the first programmer assigns number 1 to the first error that he discovered during computer testing, then it is natural to assign numbers $0, -1, -2, ..., -19$ to the errors that he discovered before typing the program into the computer. In particular, an error number 4 for the second programmer is error number -16 for the first one. This example show that although at first glance the expression $t(n)$ makes sense not only for positive n, but in all the cases when some errors were detected before the testing (and it is a very frequent case) it is quite reasonable to consider negative values of n as well.

Let's denote by n_0 the total number of errors that the first programmer discovered before he started testing on a computer. Then an error number n for a first programmer is error number $n + n_0$ for the second one. So if we denote the moment of time, when i-th programmer discovers his error number n, by $t_i(n)$, we can conclude that $t_1(n) = t_2(n+n_0)$.

It is natural to demand that the fact that a family is better (in some reasonable sense) that some other family should not depend on the number of bugs that were discovered before we started computer testing. In other words, if a family $\{f(n), g(n), ...\}$ is better than the family, that consists of the functions $h(n)$, $k(n)$, ..., and n_0 is an integer, then the family $\{f(n + n_0), g(n + n_0), ...\}$ is better than the family

$$\{h(n + n_0), k(n + n_0), ...\}.$$

It is also reasonable to demand that the criterion should not depend on how we define a bug. Another reasonable demand is associated with another uncertainty in counting bugs: depending on how detailed we are, we can either count failures or count the lines that demanded correction, etc. Crudely speaking, if we count lines instead of counting failures, then we count C bugs where we initially counted just one, where C is an average number of changed lines per failure. So what was $t(n)$ for one programmer turns out to be $t(Cn)$ for another. This transformation means that we have changed a *scale* for measuring bugs just like we change scales from kilograms to pounds: what was 1 kg is now ≈ 2.2 lb; likewise what was 1 bug in the old scale is C bugs in the new one. It may also

seem reasonable to demand that the fact that one of the families is better should be still true if we change the way we count bugs.

This consideration causes an additional problem: previously the number of bugs was always a positive integer, now we must consider "fractions" of bugs, because, e.g., a failure in average means that several lines should be changed, and therefore a line of changed code correspond in average to only part of a failure.

In this case what was 1 bug for one programmer is *part of the bug* for another one. So we come to the notion of a *fractional bug* (see, e.g., [Musa et al 1987]). From the mathematical viewpoint it means that in this case the argument n of the desired function $t(n)$ is not necessarily an integer, as before, but it can be an arbitrary real number.

Now we are ready to introduce formal definitions.

2.2. Definitions and the main results

Definition 1. By a *family of approximations* (or *family* for short) we mean a set of all functions of the type $a + b\mathbf{f}_0(n)$, where a is an arbitrary real number, b is an arbitrary positive number and $\mathbf{f}_0(n) = [f_0^-(n), f_0^+(n)]$ is a function from integers to intervals. Two families are considered *equal* if the corresponding sets coincide (i.e., if they consist of the same functions).

Denotation. Let's denote the set of all families by Φ.

Definition 2. [Kreinovich 1990, Kreinovich Kumar 1990, Kreinovich et al 1993]. A pair of relations $(<, \sim)$ is called *consistent* if it satisfies the following conditions: (1) if $a < b$ and $b < c$ then $a < c$; (2) $a \sim a$; (3) if $a \sim b$ then $b \sim a$; (4) if $a \sim b$ and $b \sim c$ then $a \sim c$; (5) if $a < b$ and $b \sim c$ then $a < c$; (6) if $a \sim b$ and $b < c$ then $a < c$; (7) if $a < b$ then it is not true that $b < a$, and it is not true that $a \sim b$.

Definition 3. Assume a set A is given. Its elements will be called *alternatives*. By an *optimality criterion* we mean a consistent pair $(<, \sim)$ of relations on the set A of all alternatives. If $a > b$ we say that a is *better* than b; if $a \sim b$ we say that the alternatives a and b are *equivalent* with respect to this criterion. We say that an alternative a is *optimal* (or *best*) with respect to a criterion $(<, \sim)$ if for every other alternative b either $a > b$ or $a \sim b$.

We say that a criterion is *final* if there exists an optimal alternative, and this optimal alternative is unique.

Comment. We'll consider optimality criteria on the set Φ of all families.

Definition 4. By a *m–shift* of a function $\mathbf{f}(n)$ we mean a function $\mathbf{g}(n) = \mathbf{f}(n + m)$. By a *$m$–shift* of a family F we mean the family consisting of m–shifts of all functions from F.

Denotation. m–shift of a family F will be denoted by $S_m(F)$.

Definition 5. We say that an optimality criterion on Φ is *shift-invariant* if for every two families F and G and for every integer m the following two conditions are true:

i) if F is better than G in the sense of this criterion (i.e., $F > G$), then $S_m(F) > S_m(G)$;

ii) if F is equivalent to G in the sense of this criterion (i.e., $F \sim G$), then $S_m(F) \sim S_m(G)$.

Comment. As we have already remarked, the demands that the optimality criterion is final and shift invariant are quite reasonable. At first glance they may seem rather trivial and therefore weak. However, these demands are strong enough, as the following Theorem shows:

THEOREM 1. *If a family F is optimal in the sense of some optimality criterion that is final and shift-invariant, then it either contains functions $a + \mathbf{b}\exp(kn)$ (that corresponds to the logarithmic model), or functions of the type $\mathbf{a} + bn$.*

(The proofs are given in Section 4).

Comment. In addition to a logarithmic model we get an additional case, when $t(n) = a+bn$. This case corresponds to a disastrous situation when the bugs are found again and again with a non-decreasing rate. Such situations happen sometimes; the usual reaction is to throw away this software as non-repairable and write everything anew.

PROPOSITION 1. *If an optimality criterion is final and shift-invariant then the optimal family F_{opt} is also shift-invariant, i.e., $S_m(F_{opt}) = F_{opt}$.*

Comment. This Proposition shows that if we use an optimal approximation family then it does not matter how you count bugs: if you start counting them from m-th bug you still get the same approximation family; therefore if you use this family for extrapolation, you get the same extrapolation results! This irrelevance to the choice of the starting point for bugs explains why it is so difficult to choose such a point; on the other hand, what this result says is that there is no need to worry about that: no matter how we count, the reliability estimates will still be the same.

Comment. We have used shift-invariance. Let us now express the demand of scale-invariance in mathematical terms. Before we write down the definitions we have to make two remarks related to the problem of how to count bugs.

First, since scale invariance corresponds to the question "what is a bug": from one point of view this particular error is a bug, from another it is a part of a bug, the number of bugs n is not necessarily an integer, it can be a real number.

The second remark corresponds to the fact that different people can start counting bugs in different moments of time: a bolder programmer can start debugging at a point which a more cautious person will still consider a part of the design process, when a product is not yet ready for real testing and debugging. In the shift-invariant case we were lucky enough to have Proposition 1, that allowed us to disregard this difference. But in scale-invariant case it may be not true (and it really turns out to be not true), so we better take this possible difference into consideration in our definitions.

With these remarks in mind we come to the following definitions.

Definition $1'$. By a *family of approximations* (or *family* for short) we mean a set of all functions of the type $a + b\mathbf{f}_0(n)$, where a is an arbitrary real number, b is an arbitrary positive number and $\mathbf{f}_0(n) = [f_0^-(n), f_0^+(n)]$ is an function from real numbers to intervals for which both $f_0^-(n)$ and $f_0^+(n)$ are monotonic. Two families are considered *equal* if the corresponding sets coincide (i.e., if we consist of the same functions).

Definition $4'$. Assume some value k is fixed. By a *k-shift* (or simply a *shift* for short) of a function $\mathbf{f}(n)$ we mean a function $\mathbf{f}(n+k)$. By a *k-shift* of a family F we mean a family, that consists of k-shifts of all the functions from F.

Comment. This k corresponds to the difference between the two starting points for counting bugs: some fixed starting point that we assumed when we deduced the specific type of functions from the family, and a starting point that was really used in the debugging documentation to which we want to apply these functions.

Denotation. The set of all the families (in the sense of definition $1'$) will be denoted by Φ'.

Definition 6. Suppose $C > 0$. By a *C–rescaling* of a function $\mathbf{f}(n)$ we mean a function $\mathbf{g}(n) = \mathbf{f}(Cn)$. By a *$C$–rescaling* of a family F we mean a family consisting of C–rescalings of all functions from F.

Denotation. C–rescaling of a family F will be denoted by $R_C(F)$.

Definition 7. We say that an optimality criterion on Φ' is *scale-invariant* if for every two families F and G and for every real number $C > 0$ the following two conditions are true:

i) if F is better than G in the sense of this criterion (i.e., $F > G$), then $R_C(F) > R_C(G)$;

ii) if F is equivalent to G in the sense of this criterion (i.e., $F \sim G$), then $R_C(F) \sim R_C(G)$.

THEOREM 2. *If a family F is optimal in the sense of some optimality criterion that is final and scale-invariant, then its k-shift consists either of the functions of the type $\mathbf{a} + b\ln(k-n)$ (that correspond to the basic model), or of the functions of the type $a+\mathbf{b}(n+k)^c$.*

Comment. Models with $t \sim n^\alpha$ (*power models*) have really been proposed and experimentally confirmed (see, e.g., Chapter 11 of [Musa et al 1987]).

PROPOSITION 2. *If an optimality criterion is final and scale-invariant then the optimal family F_{opt} is also scale-invariant, i.e., $R_C(F_{opt}) = F_{opt}$.*

Comment. This Proposition shows that if you use an optimal approximation family then it does not matter what scale you choose for counting bugs: you still get the same approximation family; therefore if you use this family for extrapolation, you get the same extrapolation results.

3. BASIC CONCLUSIONS

Theorems 1 and 2 solve the following problems:

- *We have solved the problem of what interval software reliability model to choose: whatever reasonable criterion we use, we shall get either the basic or the logarithmic model.*

- *We have thus explained why the basic model and the logarithmic model are experimentally the best.*

These theorems also address the auxiliary problem: that it is difficult to choose one of the possible reasonable ways to count bugs. Namely, the corresponding Propositions explain that the optimal family does not change if we use a different way to count bugs, and therefore the extrapolation results do not change. So the answer to this problem is as follows:

- *One can count bugs in any reasonable way, the approximation function and hence the extrapolation results will not depend on that choice.*

4. PROOFS

Let's first prove Proposition 1.

Proof of Proposition 1 (the proof is actually the same as in [Kreinovich 1990, Kreinovich et al 1993]). Since the optimality criterion is final, there exists a unique family F_{opt} that is optimal with respect to this criterion, i.e., for every other F either $F_{opt} > F$ or $F_{opt} \sim F$. If $F_{opt} \sim F$ for some $F \neq F_{opt}$, then from the definition of the optimality criterion we can easily deduce that F is also optimal, which contradicts to the assumption that this criterion is final, and hence there is only one optimal family. So for every F either $F_{opt} > F$ or $F_{opt} = F$.

In particular, this is true for $F = S_m(F_{opt})$. If $F_{opt} > S_m(F_{opt})$, then from shift invariance of the optimality criterion we conclude that $S_{-m}(F_{opt}) > S_{-m}(S_m(F_{opt})) = F_{opt}$, i.e., $S_{-m}(F_{opt})$ is better than F_{opt}, which contradicts to the fact that F_{opt} is optimal. Therefore F_{opt} cannot be better than $S_m(F_{opt})$, hence $F_{opt} = S_m(F_{opt})$. Q.E.D.

Proof of Theorem 1. Since the criterion is final, there exists an optimal family $F_{opt} = \{a + b\mathbf{f}_0(n)\}$ for some function $\mathbf{f}_0(n)$. This function $\mathbf{f}_0(n)$ belongs to the family F_{opt} (for $a = 0$ and $b = 1$). Due to Proposition 1 this family is shift-invariant, i.e., $F_{opt} = S_m(F_{opt})$. In particular, for $m = 1$ we conclude that $F_{opt} = S_1(F_{opt})$. Therefore the function $S_1(\mathbf{f}_0(n)) = \mathbf{f}_0(n+1)$, that belongs to $S_1(F_{opt})$, must also belong to F_{opt}. But F_{opt} consists of all functions of the type $a + b\mathbf{f}_0(n)$, therefore there exists a and b, for which the functions $\mathbf{f}_0(n+1)$ and $a + b\mathbf{f}_0(n)$ coincide (i.e., their values are equal for all n). So we have a recurrent formula for $\mathbf{f}_0$:

$$\mathbf{f}_0(n+1) = a + b\mathbf{f}_0(n)$$

for all n. Since $b > 0$, this means that $f_0^-(n+1) = a + bf_0^-(n)$ and $f_0^+(n+1) = a + bf_0^+(n)$ for all n.

If $b = 1$, then $f_0^-(n)$ is an arithmetic progression and therefore $f_0^-(n) = C^- + an$ for some constant C^-. Similarly, $f_0^+(n) = C^+ + an$ for some C^+. As a result, $\mathbf{f}_0(n) = \mathbf{C} + an$. The corresponding family $\{a + b\mathbf{f}_0(n)\}$, therefore, consist of the functions of the type $\mathbf{C}_1 + C_2 n$.

If $b \neq 1$, then we can reduce the equations for $f_0^{\pm}(n)$ to the equations for the geometric progression by taking $g^{\pm}(n) = f_0^{\pm}(n) + C^{\pm}$ for some appropriate constant $C^{\pm}$: namely,

$$g^{\pm}(n+1) = f_0^{\pm}(n+1) + C^{\pm} = a + bf_0^{\pm}(n) + C^{\pm} =$$
$$a + b(g^{\pm}(n) - C^{\pm}) + C^{\pm} = bg^{\pm}(n) + (a + C^{\pm} - bC^{\pm}).$$

If we choose such $C^{\pm}$ that $a + C^{\pm} - bC^{\pm} = 0$, i.e., $C^+ = C^- = C = a/(b-1)$, then the equation for $g^{\pm}(n)$ turns into the equation $g^{\pm}(n+1) = bg^{\pm}(n)$ that describes a geometric progression. Therefore, $g^{\pm}(n) = g^{\pm}(0)b^n$, and $f_0^{\pm}(n) = g^{\pm}(0)b^n - C$. Hence, $\mathbf{f}_0(n) = \mathbf{g}(0)b^n - C$. Therefore, the family $\{a + b\mathbf{f}_0(n)\}$ consists of the functions of type $\mathbf{C}_1 b^n + a$. To conclude the proof, it is sufficient to use the definition of the logarithm: $b = \exp(\ln b)$ and hence $b^n = \exp(kn)$, where $k = \ln b$. Q.E.D.

Proposition 2 is proved just like Proposition 1, the only difference is that we consider $R_{C^{-1}}$ instead of S_{-m}.

Proof of Theorem 2. Like in the Proof of Theorem 1, from the scale-invariance of the optimal family $F_{opt} = \{a + b\mathbf{f}_0(n)\}$, we conclude that for every $\lambda > 0$ there exist a and b, depending on λ, such that $\mathbf{f}_0(\lambda n) = a(\lambda) + b(\lambda)\mathbf{f}_0(n)$ for all $n > 0$. In other words, $f_0^-(\lambda n) = a(\lambda) + b(\lambda)f_0^-(n)$ and $f_0^+(\lambda n) = a(\lambda) + b(\lambda)f_0^+(n)$. We want to reduce these functional equations to the equations whose solutions are already known. In order to do that let's introduce a new variable $x = \ln n$ and express $f_0^{\pm}$ in terms of this new variable. Since $n = \exp(x)$, this new expression is $F^{\pm}(x) = f_0^{\pm}(\exp(x))$. When we multiply n by a constant, we thus add a constant ($= \ln\lambda$) to its logarithm x. So in terms of $F^{\pm}(x)$ the above equations takes the form $F^{\pm}(x+C) = A(C) + B(C)F^{\pm}(x)$ for some functions A and B. Since both functions $f_0^{\pm}$ are assumed to be monotonic, the functions $F^{\pm}$ are monotonic as well. According to [Aczel 1966, Section 3.1], the most general monotonic solution of this functional equation is $F^{\pm}(x) = ax + b$ or $F^{\pm}(x) = a\exp(cx) + b$. Substituting $x = \ln n$, we get $f_0^{\pm}(n) = F^{\pm}(\ln n) = a^{\pm} + b^{\pm}\ln n$ and $f_0^{\pm}(n) = a^{\pm}n^{c^{\pm}} + b^{\pm}$.

For logarithmic $f_0^{\pm}$, we have $f_0^-(\lambda n) = a^- + b^-\ln(\lambda n) = (a^- + b^-\ln n) + b^-\ln\lambda$. Therefore, in this case, $b(\lambda) = 1$, and $a(\lambda) = b^-\ln\lambda$. Likewise, $a(\lambda) = b^+\ln\lambda$, so $b^- = b^+ = b$, and the function $f_0(n)$ takes the form $\mathbf{a} + b\ln n$.

For exponential $f_0^{\pm}(n)$, we have $f^{\pm}(\lambda n) = a^{\pm}(\lambda n)^{c^{\pm}}) + b^{\pm} = (a^{\pm}n^{c^{\pm}})\lambda^{c^{\pm}} + b^{\pm} = (f_0^-(n) - b^{\pm})\lambda^{c^{\pm}} + b^{\pm} = f_0^-(n)\lambda^{c^{\pm}} + b^{\pm}(1 - \lambda^{c^{\pm}})$. So, here, $b(\lambda) = \lambda^{c^{\pm}}$, and $a(\lambda) = b^{\pm}(1 - \lambda^{c^{\pm}})$. Since the values of $a(\lambda)$ and $b(\lambda)$ are the same for $f_0^-(n)$ and $f_0^+(n)$, we conclude that $c^- = c^+$, and $b^+ = b^-$. Therefore, $f_0(n) = \mathbf{C}_1 + C_2 n^c$.

Now, it is sufficient to write down all possible k-shifts of these families, and we get the desired result. Q.E.D.

Acknowledgments. The authors are greatly thankful to Michele Abrusci (University of Bari, Italy), Dines Bjoerner (Denmark Technical University), Daniel Cooke (University of Texas, El Paso, TX), Andrei M. Finkelstein (Institute for Applied Astronomy, St. Petersburg, Russia), Robert L. Glass (Computing Trends, State College, PA), Vitaly Kozlenko (St. Petersburg, Russia), Robert Kurshan (AT&T Bell Labs, Murray Hill), Vladimir Lifschitz (University of Texas, Austin, TX), John McCarthy (Stanford University) and Larry

Shepp (AT&T Bell Labs, Murray Hill) for valuable discussions. One of the authors (V.K.) was supported by NASA Research Grant No. 9-482, NSF grant No. CDA-9015006 and a Grant No. PF90–018 from the General Services Administration (GSA), administered by the Materials Research Institute and the Institute for Manufacturing and Materials Management (Texas Centers for Business and Enterprise Development), and partially supported by McCarthy foundations during his stay at Stanford. The authors are thankful to anonymous referees for valuable comments.

REFERENCES

J. Aczel. *Lectures on functional equations and their applications.* Academic Press, N.Y.–London, 1966.

Jaco de Bakker. *Mathematical theory of program correctness.* Englewood Cliffs, NJ, 1980.

E. M. Clarke, Robert P. Kurshan (eds.). *Computer-aided verification'90. Proceedings of a DIMACS Workshop*, American Mathematical Society, Providence, RI, 1991.

Patrick Cousot. *Methods and logics for proving programs.* In: Jan van Leeuwen (ed.) *Handbook of Theoretical Computer Science*, Vol. B, Elsevier, Amsterdam, 1990, pp. 843–982.

Alex Elentukh, Michael Elbert. *Software reliability management for the data communication project.* In: Proceedings of the 9-th Annual Software Reliability Symposium, May 1991, Colorado.

Alex Elentukh, Ray Wang, Geoff Moss, Gene Carrubba. *Customer perceived software reliability assessment for network management software.* In: Proceedings 8-th International Conference on Testing Computer Software, June 1991, Washington D.C.

William H. Farr. *A survey of software reliability modelling and estimation*, NSWC Technical Report TR 82–171, September 1983.

Andrei M. Finkelstein, Vladik Kreinovich. *Formal models of nonformalizable reasoning*, In: Abstracts of the Heyting-90 Conference on Mathematical Logic, Sofia, Bulgaria, 1988, p. 23.

Andrei M. Finkelstein, Vladik Kreinovich. *Formal models of nonformalizable reasoning: applications to computer science*, Universita degli Studi di Bari, Dipartimento Di Scienze Filosofische, Rapporto Scientifico No. 7, Gennario 1989.

Robert L. Glass. *Software conflict. Essays on the art and science of software engineering.* Yourdon Press, Englewood Cliffs, NJ, 1991.

C. A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM, 1969, Vol. 12, pp. 576–580.

Vladik Kreinovich. In: Arkady Khaskin (ed.), *Software reliability*, USSR National Institute for Electromeasuring Devices Technical Report, 1985 (in Russian).

Vladik Kreinovich. *Group-theoretic approach to intractable problems.* In: COLOG-88, Proceedings of the International Conference in Computer Logic, Tallinn, Estonia, 1988, pp. 31–42.

Vladik Kreinovich. *Optimization in case of uncertain optimality criteria*, Center for New Informational Technology, Leningrad, Technical Report, 1989 (in Russian).

Vladik Kreinovich. *Group-theoretic approach to intractable problems.* In: Lecture Notes in Computer Science, Vol. 417, Springer-Verlag, Berlin-Heidelberg-N.Y., 1990, pp. 112–121.

Vladik Kreinovich, Vitaly Kozlenko. *Optimization in case of uncertain optimality criteria.* In: Proceedings of 4 National Conference on Applications of Mathematical Logic, Tallinn, USSR, 1986, pp. 126–128 (in Russian).

Vladik Kreinovich, Sundeep Kumar. *Optimal choice of $\&$– and $\vee$-operations for expert values.* In: Proceedings of the 3rd University of New Brunswick Artificial Intelligence Workshop, Fredericton, N.B., Canada, 1990, pp. 169–178.

Vladik Kreinovich, Chris Quintana, Olac Fuentes, *Genetic algorithms: what fitness scaling is optimal?* Cybernetics and Systems, 1993, Vol. 24, pp. 9–26.

Vladik Kreinovich, Thomas Swenson, Alex Elentukh, *When to stop testing software? A new approach that can lead to a final answer*, University of Texas at El Paso, Computer Science Department, Technical Report UTEP-CS-91-17, 1991.

John McCarthy. *Towards a mathematical theory of computation.* In: Proceedings IFIP Congress 62, North Holland, Amsterdam, 1963, pp. 21–28.

John McCarthy. *A basis for the mathematical theory of computation.* In: P. Braffort, D. Hirschenberg (eds.) *Computer programming and formal systems*, North-Holland, Amsterdam, 1963, pp. 33–70.

R. E. Moore, *Methods and applications of interval analysis*, SIAM, Philadelphia, 1979.

John D. Musa. *A theory of software reliability and its applications.* IEEE Transactions on Software Engineering, Vol. SE– 1, 1975, No. 3, pp. 312–327.

John D. Musa, *Operational profiles in software reliability engineering*, IEEE Software, 1993, Vol. 10, No. 2, pp. 14–32.

John D. Musa, Anthony Iannino, Kazuhira Okumoto. *Software reliability: measurement, prediction, application.* Mc-Graw Hill Co., 1987.

John D. Musa, Kazuhira Okumoto. *A logarithmic Poisson execution time model for software reliability measurement.* In: Proceedings Seventh International Conference on Software Engineering, Orlando, 1984, pp. 230–238.

Proceedings of the 9-th Annual Software Reliability Symposium, May 1991, Colorado.

David M. Seifert, *Minutes of AIAA SBOS Committee on Standards Software Reliability Working Group Meeting 5*, Space-Based Observation Systems, Nashua, NH, August 1990, p. 4.

David M. Seifert, George E. Stark, *Software reliability handbook: achieving reliable software*, IEEE Computer, December 1992, pp. 64–66.