CONTINUITY IN SOFTWARE SYSTEMS

Dick Hamlet*
Portland State University
hamlet@cs.pdx.edu

ABSTRACT

Most engineering artifacts behave in a continuous fashion, and this property is generally believed to underlie their dependability. In contrast, software systems do not have continuous behavior, which is taken to be an underlying cause of their undependability. The theory of software reliability has been questioned because technically the sampling on which it is based applies only to continuous functions.

This paper examines the role of continuity in engineering, particularly in testing and certifying artifacts, then considers the analogous software situations and the ways in which software is intrinsically unlike other engineered objects. Several definitions of software 'continuity' are proposed and related to ideas in software testing. It is shown how 'continuity' can be established in practice, and the consequences for testing and analysis of knowing that a program is 'continuous.'

Underlying any use of software 'continuity' is the continuity of its specification in the usual mathematical sense. However, many software applications are intrinsically discontinuous and one reason why software is so valuable is its natural ability to handle these applications, where it makes no sense to seek software 'continuity' or to blame poor dependability on its absence.

1. INTUITIVE CONTINUITY

The untrustworthy nature of software systems running on digital computers is often blamed on the lack of continuity inherent in these systems. The absence of continuity is an 'obvious' fact that is seldom precisely stated or considered. The intuitive meaning of continuity, as it relates to the behavior of systems, can be seen from a description of the role continuity plays in a simple mechanical tool.

1.1 The Shovel as a Continuous System

The usual garden shovel is constructed by inserting a wooden handle into a tubular socket in a metal blade. The resulting tool is very versatile, and can be used to dig out a rock or a tree stump. In this application, the blade of the shovel is forced under the rock, and the handle used to pry. The shovel acts as a lever with the fulcrum at the top of the blade where it joins the handle (bearing against the ground at the side of the hole), the blade being one arm of the lever and the handle the other. Because the handle is longer, a large upward force can be brought to bear on the rock as the handle is pushed down.

Anyone who has used a shovel in this way knows that it is possible to break the tool on a too-large rock. The handle breaks just where it goes into the blade socket, or the blade itself fractures at the base of the socket. The experienced rock digger learns to sense the limits of a shovel, which is possible because the behavior of the shovel is continuous. Intuitively, as more force is applied to the handle, the stress at the weakest point of the tool rises, and does so in a simple way: a bit more force, a bit more stress. The relationship is not necessarily linear (although often it is, in so called elastic behavior), but the stress changes smoothly with the force – in a word, the change is continuous. The shovel breaks when the stress exceeds what the materials can bear. The experienced shovel user learns to stay in the continuous region, and can thus depend on a shovel year after year.

Suppose that the behavior of a shovel were not continuous as described in the previous paragraph, the stress not smoothly related to the applied force. Then a digger might find that at a certain peculiar applied force value the handle suddenly snaps, even though in previous digging it was fine for forces both smaller and larger than this. Such a shovel would be hopeless to use, because one would never know when it was going to misbehave.

1.2 The Software Analogy

The analogy between software systems and shovels is obvious: if the software is analogous to the shovel, the data inputs to the system are like the applied force, and the system outputs are like the stress. The system breaks (fails) when an output is incorrect. The software system is not intuitively continuous, because no matter how much it has been used (tested) without failure, it can happen that an input causes it to fail, and that input can be arbitrarily close to other inputs that did not fail. To carry things a bit farther, it is possible to test a shovel by prying with a few different forces, and if it does not break on these tests, to know that it will not break at intermediate values. This is precisely what cannot be done for software.

There is also a deeper sense in which continuity applies to shovels but not to software. The parameters that define an individual shovel in a batch produced by some manufacturing process also enter into the behavior of the shovel in a continuous way. This means that it is possible to take a sample from the shovel production line, test these few shovels, and then statistically predict how the entire population

* Supported by NSF ITR grant CCR-0112654

of shovels is likely to behave. (For example, to assign a probability and confidence to the proposition that no shovel will break for any applied force in a given range of values.) This statistical sampling is called 'life testing,' and it is the original source for the theory of software reliability [13], in which the sampling is analogous to a random software test. The use of life-testing theory is technically invalid for software, because the samples are not drawn from a continuous distribution.

2. FORMAL DEFINITIONS OF SOFTWARE 'CONTINUITY'

The definition of a continuous real function is the most intuitively appealing, and describes the behavior of many physical systems like the shovel.

DEFINITION: A real function f is *continuous* at x_0 iff: Given any $\epsilon > 0,\ \exists \delta > 0$ such that

$$\forall x\ (|x - x_0| < \delta \implies |f(x) - f(x_0)| < \epsilon). \qquad (1)$$

Or,

$$\lim_{x \to x_0} f(x) \to f(x_0).$$

The $\epsilon - \delta$ form of the definition is better suited to the purpose of defining notions of continuity for software.

Notions of left- and right- and piecewise-continuous functions are similarly defined.

2.1 Continuity on Discrete Sets

As might be expected, there has been previous work on 'discrete continuity.' The idea is a starting point for so-called digital topology, which emerged from image processing. Rosenfeld's seminal paper [12] considered the case of a function from discrete pixels to the same, an image transformation. In the one-dimensional case the function domain and range sets are integer intervals, which Rosenfeld takes to be finite. Intuitively, the functions to be considered approximate real-valued functions by taking as value at an input pixel, the nearest output pixel to the real-function's value. Points are *neighbors* in this space iff their Euclidean separation is no more than 1; that is, iff the points are the same or they are successive integers. Rosenfeld's definition is:

DEFINITION: An integer function f defined on a finite interval of the integers is *discretely continuous* iff:

Given any $\epsilon \geq 1,\ \exists \delta \geq 1$ such that

$$\forall x\ (|x - x_0| \leq \delta \implies |f(x) - f(x_0)| \leq \epsilon). \qquad (2)$$

The changes in this definition, in comparison with definition (1), are appropriate to the discrete space and its Euclidean distance.

Rosenfeld proves that discretely continuous functions are characterized by carrying neighbors into neighbors. He also establishes elementary properties of the definition, among which are:

1. If f is discretely continuous the continuity is uniform. (That is, for all points x_0 the same δ works for the same ϵ in (2).)
2. A discretely continuous function has the intermediate-value property. (That is, if $f(x) < m < f(y), \exists z$ such that $f(z) = m$.)
3. The composition of discretely continuous functions is discretely continuous.
4. Most arithmetic combinations of discretely continuous functions (e.g., the sum) are not necessarily discretely continuous.

Property 1) is stronger than for the definition on the reals; property 4) is weaker; the others are the same. The proofs of these properties are straightforward. Intuitively, the discretely continuous functions fail to be closed under arithmetic operations because ϵ values cannot be adjusted arbitrarily as they are in the proof for the real case.

Further properties of discretely continuous functions were established by Rosenfeld and more recently by Boxer [4]. For example, the only continuous 1-1 functions are combinations of translations and reflections.

2.2 Floating-point Continuity

As approximations to mathematical real values, floating-point quantities in digital systems cover a finite range defined by the exponent-field size, and have limited precision defined by the mantissa-field size. This is exactly what is needed to capture physical measurements. The floating-point subset of the reals is characterized by the existence of a minimum fractional spacing γ which is the granularity of the mantissa, $\gamma = 2^{-M}$ for a binary field of M bits. Software can vary γ by using a multiple-word mantissa, even beyond that supported by the hardware operations if necessary.

There are two ways to look at the approximate nature of floating-point values. The program specification may explicitly mention the approximations; or, the specification may be expressed in continuous terms, with the understanding that its statements about values apply only approximately. The former is a more cumbersome but more precise version of the latter. A specification statement like:

> The program will compute f to within 0.05%,

really means something like the following[1]:

> For all real x such that $f(x) \neq 0$, program inputs will approximate x with error at most h_x, and for all input values z such that $|x - z| < h_x$ the program output v_z at z will satisfy $|(f(x) - v_x)/f(x)| < .0005$.

The intuitive content of the requirement is that when any value x arises in the operation of the system, the program will operate on an input approximation to x limited by the hardware, and the output will be similarly limited, but the calculated value will nevertheless be close enough to $f(x)$.

DEFINITION: A program computes a *floating-point continuous* function F if F approximates a continuous function to the fractional accuracy given in the program specification.

The definition leaves open the possibility that a program might compute a floating-point continuous function F, but F might not be the function it was specified to compute. The definition does make the specification the arbiter of the required accuracy.

Floating-point continuity is a natural extension of discrete continuity in that by interpreting the mantissa as an integer and taking the

[1]The excluded case $f(x) = 0$ must be appropriately included, and the restriction to a real interval would also be present in any actual specification, but including these technicalities would obscure the point about approximating continuity.

required accuracy to be one mantissa unit, the two notions are intuitively the same, but discrete continuity has no concept that corresponds to the specification's ability to require an arbitrary accuracy.

The floating-point-continuous functions are closed under composition, but not under arithmetic operations like addition, for the same reasons that establish these properties for discrete continuity.

In the remainder of this paper, statements such as "the computed value is $f(0)$" are to be taken to mean that the floating-point approximation holds for an accuracy given in the specification. It can of course happen that the digital nature of the computation is the very reason why the program fails to obtain a sufficiently accurate result, always a thorny problem for the numerical analyst. But this special case is no different in principle from one in which any incorrect result is obtained.

2.3 Testing for Trustworthiness

The rationale for trusting a tested mechanical system at some usage point x is that its (tested) good behavior at points nearby x, along with the continuity at x of functions describing its properties, guarantee that it will behave properly at x. A bit more than this is required, because assurance is needed that the behavior cannot vary too wildly. It can be as dangerous to have a narrow 'spike' in a continuous function as to have a discontinuity. Suppose that a mechanical system has a function Z that describes a system property. *Z satisfies a Lipschitz condition* at x_0 iff there is a parameter $B_0 > 0$ such that:

$$\exists b > 0,\ \forall x\ (|x - x_0| < b \implies |Z(x) - Z(x_0)| \leq B_0).$$

Intuitively, the function cannot change value arbitrarily in some neighborhood of each point for which it satisfies a Lipschitz condition. A real function that is continuous at x satisfies a Lipschitz condition at x; but the converse is not true, since the limited variation requires nothing about the behavior inside the bound.

In testing a mechanical system whose property function Z satisfies a Lipschitz condition, one proceeds as follows:

> Let the failure limit of Z be L_Z, which for simplicity take as a maximum magnitude. Test the system using points $\{x_1, x_2, ..., x_k\}$ whose Lipschitz constants are respectively $\{B_1, B_2, ..., B_k\}$ and whose neighborhoods overlap to cover the usage range. At each test point t, $Z(t)$ must be within the failure limits reduced by $B = \max_{1 \leq i \leq k} B_i$, that is:
>
> $$\forall i, 1 \leq i \leq k\ (|Z(x_i)| < L_Z - B).$$

There is some justification in calling B the *safety factor*[2] for Z. The testing regimen then guarantees that the system cannot fail, for it has been tried at points near enough to each other, and the worst variation between points has been allowed for by the safety factor.

2.4 Failure Continuity

The software analog of the mechanical behavior of the previous section would be something like this:

DEFINITION: Let a program p have specification S. p is *failure continuous* at x_0 iff $\exists b > 0$ such that:

$$p(x_0) \neq S(x_0) \implies \forall t, |x_0 - t| < b\ (p(t) \neq S(t)).$$

That is, if the program fails to meet its specification at some input, then it also fails in a neighborhood. The failure statement implies that if such a program is correct at a point, it must also be correct in a neighborhood of that point.

Intuitively, failure continuity means that it is possible to test a program at particular points, and its success there guarantees that it will also succeed at nearby points. A certification of the program can be accomplished as for the mechanical system, by testing at points whose neighborhoods cover the domain. If all of them succeed, there can be no failures.

In his seminal 1976 paper [8], Howden defined a property of subdomain testing methods that he called 'reliable'[3]. In a 'reliable' testing method, one successful test in each subdomain proves that the program is correct. Failure continuity is the same as Howden's idea, except that he thought of the program as computing an arbitrary function and the testing method as defining the 'reliable' subdomains. The present viewpoint ascribes the property to the program itself. The difficulty with both ideas in practice is that in Howden's formulation, there is no way to determine if subdomains are 'reliable;' for failure continuity, there is no way to find the failure-continuous neighborhoods.

Although the definition of failure continuity appears to be exactly the property we are seeking for dependable software (just as Howden's 'reliable' seemed just what is needed in testing), the definition goes too far by capturing an essentially undecidable idea. An example will make clear the difficulty. Suppose that a specification requires the computation of $\sin(x)$ with 1% accuracy over the interval [0, 2π), and that a program trying to meet this specification computes the constant zero function. Both specification and program are continuous and the program is within 1% of specification on three intervals, roughly [0, .01], [3.13, 3.15], and [6.27, 2π), but the program is not failure continuous near the internal interval end points. The difficulty with the definition, and perhaps with all of computer programming, is that a specification is arbitrary, and no effective property can capture whether or not a program meets it.

2.5 Intuitive Program Continuity

In the sequel we will speak of the 'continuity' of the function computed by a program (or even of a 'continuous program') without specifying the particular technical definition. This intuitive way of speaking recognizes that the intuitive content of any definition in the $\epsilon - \delta$ form is that 'jumps' in functional value must be limited to those within the digital granularity. In Rosenfeld's definition, the granularity is 1; in floating-point continuity it is the accuracy required by the specification.

It is common practice in program analysis to treat variables of a program like mathematical variables, and to write formulas using them as if they described program behavior. This false viewpoint is dangerous – the values are limited and of limited precision, so assuming otherwise can lead to obvious mistakes. But at the same time, treating program variables as mathematical gains a great simplicity and access to powerful methods of analysis. What underlies

[2] In engineering practice, safety factors are usually multiplicative rather than additive, but using a real 'factor' in the definition of limited variation raises irrelevant difficulties when Z can take zero values. Petroski's excellent book [10] gives case studies in which safety factors saved (or did not save) flawed designs. Addis [1] even makes a case for safety factors saving designs whose theoretical calculations are scientifically indefensible.

[3] The inverted commas are needed because Howden's idea has nothing to do with the usual statistical reliability.

the successful use of this abstraction is the approximation ideas discussed in Sections 2.1 and 2.2, justified by the following theorem:

THEOREM. If a program computes (in the sense of symbolic execution) a mathematically continuous function when its variables are taken to be real-valued, then: (1) The program is discretely continuous over a suitable interval, and (2) There is a specification accuracy for which the program is floating-point continuous.

Symbolic execution yields a formula for the function f that the program would compute if it used real values; if f is continuous, then the program is technically continuous in the senses of Sections 2.1 and 2.2. The essential idea in proving the theorem is that the difference between values the program actually computes and values of f can be controlled by restricting the interval or widening the specified accuracy. A given program may be incapable of computing a particular specified function to an arbitrary accuracy, for all that it is floating point continuous.

The converse of the theorem is false, since a program-computed function may contain small jumps that destroy mathematical continuity but are within the limits of the technical definitions. For example, a program using a partial sum of a convergent series may compute an adequate approximation to the continuous function defined by the series, yet its symbolic-execution formula can be for an almost-everywhere discontinuous function.

3. DISCUSSION

It is a persistent mistake in program testing and analysis to confuse properties of the specification with those of a program being analyzed, and thereby compromise the analysis. A classic case is a specification that requires the computation of a polynomial function. Knowing that for polynomial degree m such functions are determined by m distinct points, a tester may believe that m test points are sufficient to prove correctness. The fallacy underlying this belief is the implicit assumption that the program is somehow constrained to follow the specification. But the program may not in fact compute a polynomial of degree m, in which case the test has no significance.

Testing and proving analysis methods might be profitably combined to exploit properties of the specification. If a property of program P can be established by proof methods (e.g., that P *does* compute an m-degree polynomial), then the correctness argument can be completed by testing (e.g., that the polynomial is the one specified, using m test points.) Matthew Geller made a start on such a method [6], but was unable to go beyond piecewise linear functions and their computation by elementary program constructs.

In the present context, we wish to establish that a particular program *does* compute a 'continuous' function, and then explore what analysis opportunities this opens up.

3.1 Demonstrating Program 'Continuity'

The source of discontinuity in imperative programs (i.e., those written in a language like C) is conditional statements. In the simplest case, a basic block of code computes a polynomial function of the variables used in the block. When two such blocks are placed in the `THEN` and `ELSE` arms of a conditional statement, the conditional expression partitions the domain into two subdomains. When two nearby points lie one in each subdomain, the values computed at each need not be close to each other, since these values come from the different blocks. A trivial example is the C fragment:

```
if (x > 2.71) z = 0;
else z = 1.414;
```

The computation in each block is a constant function, but the whole program is discontinuous at `x` = 2.71.

To make a long story short, for imperative programs the input space can be partitioned into path subdomains, on each of which the computation is polynomial. That the path subdomains are somehow fundamental to the structure of the program has long been believed in both practical and theoretical testing work. The most stringent certification tests (for example, those required by the FAA) are based on paths; path testing has been studied from the beginning of the discipline, by Howden [8] and by Richardson and Clarke [11]. Dave Mason is currently working on a component-based testing and proof theory based on path domains [9].

Thus any imperative program is piecewise continuous, the potential discontinuities being confined to the boundaries of its path domains. A conventional boundary-coverage test, using all boundary points and nearest points in the path subdomains, would detect all discontinuities. In a typical numerical program, the specification function is not discontinuous at these boundary points, so boundary tests should be good at uncovering failures. Testing for continuity across a boundary has the great practical advantage that it does not require an oracle.

Unfortunately, there are a potential infinity of path subdomains boundary points for any imperative program that contains looping or recursive constructions. Hence it is not possible in practice to mechanically test for continuity. Nor is symbolic execution a decidable technique – again because of the existence of loops and recursion. However, proving continuity analytically is not as difficult as symbolic execution itself, since the point is not to solve for the function computed, but only to show that it does not have discontinuities of a given size. This might be accomplished by replacing loops by bounded unrolling (and similarly limiting the depth of recursion), then showing that the neglected tails cannot cause too large a jump in values.

Another approach is to use a different program structure. If a program is a (functional) composition of components then establishing continuity of the latter is sufficient, because the notions of program continuity are closed under composition. This suggests that functional programming languages will be easier to analyze for continuity, and that component-based programs should confine iteration and recursion to the components.

3.2 Analyzing 'Continuous' Programs

If a program's specification is continuous, and it can be established that the program is continuous, there remains the possibility that the program is still incorrect because it is computing the wrong continuous function.

In principle, the decomposition into path subdomains can be used to show correctness, by a combination of testing and proving methods. On each path subdomain the program computes a polynomial, and this can be compared with the specification function restricted to that subdomain. If the restricted specification is also polynomial, testing solves the equivalence problem. It is not necessary to obtain the program's polynomial explicitly, only to bound its degree to determine the number of test points required. However, it is unlikely that the specification is explicitly in polynomial form. The more likely situation is that the program's polynomial only approximates

the specified function, an approximation hoped to be sufficiently accurate over the restricted subdomain.

There is a solution in principle to the problem of deciding whether a polynomial adequately approximates a specification function, but it may prove impractical. Any polynomial satisfies a uniform Lipschitz condition whose constant can be estimated from a bound on the degree and test points. (That is, the constant can be determined, given the program.) If the specification function also satisfies a Lipschitz condition, then a test spacing Δ exists such that correct (that is, sufficiently accurate) behavior on test points with this spacing guarantees correctness between them. The impracticality arises because Δ may be so small that the number of tests required is prohibitive.

Operational (random) testing can be legitimately used on a continuous program. If there is a Lipschitz condition, there is an alternative to the operational distribution. Test points drawn from a uniform distribution have a probability of falling inside Lipschitz neighborhoods so that no calculated program value is very far away from one of these test results. This can be used to define a probability that the program is correct. This testing scheme is a kind of "random structural testing" in that the uniform distribution seeks to explore the program structure as defined by its continuity and Lipschitz properties. There is no intrinsic reason why 'random structural testing' for a high probability of correctness must be subject to the impracticality limitations [5] of operational testing using the operational distribution.

A final idea that deserves exploration for continuous programs is run-time self-testing as suggested by Ammann [2] and Blum [3]. Self-testing requires some means of knowing for a collection of randomly chosen variants of an input value, that agreement among the results is very unlikely if they are incorrect. A preliminary exploration of this idea [7] was not very successful, but the 'random structural testing' suggested in the previous paragraph might provide the means to do better.

3.3 Inherently Discontinuous Specifications

The attempt to bring a notion of continuity into software analysis is of course only useful if (an approximation to) continuous behavior is in fact what the software is required to have. This is the case in many safety-critical applications, since 'safety' is often defined as properly responding to real-world, physical phenomena, which are themselves often continuous. For example, flight-control software for aircraft and missiles includes many continuous aspects in computing position and momentum and in applying forces to alter them. Shutdown systems in nuclear power plants, and other similar process-control systems, are also required to behave continuously, and it is a current dilemma of regulation agencies that in these applications digital software systems are replacing older analog systems. The old systems behaved continuously and were certified by established procedures (as indicated in Section 2.3), but the regulators do not know what to do for the software systems. The techniques suggested in Sections 3.1 and 3.2 are appropriate to these continuous applications.

However, many software applications are not specified in any continuous way, and are inherently discontinuous. Indeed, the reason that software is so flexible and powerful, in comparison to mechanical or electrical hardware, is precisely that inherently continuous systems have a hard time acting in the discontinuous way that is required. Most string-input applications, for example compilers, are not continuous. In a compiler, if the source-program input string changes by a single character (which would be the 'granularity' of a string, analogous to the least change γ in a floating-point quantity), it often happens that a crucial output (`SYNTAX ERROR` vs. `COMPILE OK`) changes abruptly.

For discontinuous specifications, a theory of software continuity is no help in testing and analysis, except perhaps to provide clarity in thinking about the discontinuous program: if it behaves badly, this behavior cannot be blamed on the lack of continuity that is intrinsic to the application.

4. CONCLUSIONS AND FUTURE WORK

It has been suggested that a combination of proving and testing methods be based on software 'continuity,' a general property that shares qualities with correctness yet is easier to analyze. While we cannot hope in software to emulate fully the use that physical systems make of continuity in real-world phenomena, software analogs can be defined and appear worthy of investigation.

In principle, the 'continuity' of a program can be determined by testing on path boundaries. Software reliability theory should be valid for 'continuous' programs, and continuity properties suggest alternatives for random testing to establish probable correctness.

Just as it has seemed wise to separate safety features of systems from their other functional features, so continuity analysis suggests that computations that are intended to approximate continuous real functions should be separated from those that are inherently discontinuous. Components and subsystems should have one kind of function or the other, but not both.

5. REFERENCES

[1] William Addis. *Structural Engineering, the nature of theory and design*. Ellis Horwood, 1990.

[2] P. Ammann and J. C. Knight. Data diversity: an approach to software fault tolerance. *IEEE Trans. on Computers*, pages 418–425, 1988.

[3] M. Blum and S. Kannan. Designing programs that check their work. *JACM*, pages 269–291, January 1995.

[4] Laurence Boxer. Digitally continuous functions. *Pattern Recognition Letters*, pages 833–839, August 1994.

[5] R. W. Butler and G. B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.

[6] Matthew Geller. Test data as an aid in proving program correctness. *Comm. of the ACM*, 21:368–375, May 1978.

[7] Dick Hamlet. Predicting dependability by testing. In *Proceedings ISSTA '96*, pages 84–91, San Diego, CA, 1996.

[8] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. on Soft. Eng.*, 2:208–215, 1976.

[9] Dave Mason. University of Waterloo PhD thesis in progress.

[10] Henry Petroski. *To Engineer is Human: The Role of Failure in Successful Design*. St. Martin's Press, New York, 1985.

[11] D. J. Richardson and L. A. Clarke. Partition analysis: a method combining testing and verification. *IEEE Transactions on Software Engineering*, 11(12):1477–1490, December 1985.

[12] Azriel Rosenfeld. 'Continuous' functions on digital pictures. *Pattern Recognition Letters*, pages 177–184, July 1986.

[13] M. L. Shooman. *Software Engineering Design, Reliability, and Management*. McGraw-Hill, New York, NY, 1983.