Subdomain Testing of Units and Systems with State

Dick Hamlet[*]
Portland State University
Portland, OR, USA
hamlet@cs.pdx.edu

ABSTRACT

This paper extends basic software-testing theory to software components and adds explicit state to the theory. The resulting theory is simple enough to abstractly model the construction of systems from their parts ('units'). It provides an unconventional insight into the relationship between testing units and testing systems. Experiments exploring the theory support the following conclusions:

- Units should be independent, more like what are called "components" than subroutines or object-oriented classes.
- Units' persistent state should be local.
- Units should be extensively tested.

A new kind of system testing is proposed: Unit-test results are combined to approximate the system behavior. Testing the approximation is cheaper and easier than testing the actual system and more likely to expose system problems.

Category and Subject Descriptor: D.2.5 Software engineering, Testing and debugging
General Terms: Verification
Keywords: Testing theory, unit/system testing, persistent state

1. INTRODUCTION

Software testing is a practical activity usually conducted for the purpose of discovering failures so that they can be fixed before an application system is released. The practice is labor-intensive and often conducted in a haphazard way, yet it has a firmly established place in software development, commonly consuming a large fraction of a project's time and resources. It is frequently stated that testing desperately needs a theoretical foundation, which would provide insight and direction to practice. There is such a theory, which began with the work of Goodenough and Gerhard [4], a theory which should be just what practice needs, because it is simple, abstract, and revealing. Yet the theory has not been very influential, perhaps because its results are largely negative. Having captured the essence of testing, it shows that what we do in practice cannot be easily justified. Two examples of such results are of interest:

[*]Supported by NSF ITR grant CCR-0112654 and by an E.T.S. Walton grant from Science Foundation Ireland. Neither institution is in any way responsible for the statements made in this paper.

Subdomain testing. Howden [8] formulated a theory of subdomain-based testing that describes virtually all methods in use, and showed that ideal subdomain-based methods do not exist. Any algorithmic method must give potentially misleading results. Honest practical testers know this, although they probably never heard of Howden's theory.

Random testing. Butler and Finelli [3] looked at the practical limitations of the only alternative to subdomain testing, random testing, and concluded that it is impractical. To conduct a random test to establish high confidence that a program will not fail in N hours of operation requires more than N hours of testing. So for strong guarantees, e.g., in aircraft flight-control software, the required testing cannot be accomplished in any reasonable time. This result is not well known to practical testers and necessarily ignored (except perhaps in engineers' nightmares) in safety-critical applications.

In this paper, abstract testing theory is extended to investigate three fundamental questions:
(1) What form should the "units" of a software system take?
(2) How should state be handled in the units and in the system?
(3) How should units and systems be tested?
The theoretical answers can be worked out in complete detail for simple but revealing experiments that suggest how to test software.

Section 2 reviews the theoretical model and extends it to include state. Section 3 models the 'units' of software as executable programs, an idea that comes from the component-software world. Section 4 reports on two experiments with the role of state in testing and the relationship between unit- and system tests. Section 5 draws conclusions from these experiments, and proposes a new kind of system testing in which combinations of component test executions substitute for executing the actual system.

2. THE TESTING-THEORY MODEL

The model used by Goodenough and Gerhart, Howden, and almost all testing theoreticians, assigns functional semantics to programs. The theory is reviewed here to establish a consistent notation; the reader familiar with its ideas may want to read the new definitions in Section 2.4 and then skip to Section 3.

2.1 Black-box Functional Semantics

A program P is taken to have a meaning that is a function mapping an input domain D to an output range R. This idea goes back to Turing, and Harlan Mills [11] suggested a clever notation that is a variant of one used by Kleene [9]. The meaning of P is a function $\boxed{P} : D \to R$. Mills's notation is literally the 'black-box' meaning of P as a mapping from input to output. A specification for a program is similarly taken to be a (partial) input-output function[1] $F : D \rightarrow R$, and correctness of P wrt F means that $\boxed{P} = F$. A testset T is a subset of the input domain, $T \subseteq D$. For program P with specification F to fail on T means precisely that $\exists t \in T, \boxed{P}(t) \neq F(t)$.

Subdomain testing divides the input domain D into n subdomains $S_i, 1 \leq i \leq n$, $D = \cup_{i=1}^{n} S_i$. A testset T covers the subdomains if $\forall i, T \cap S_i \neq \varnothing$. The success of a testset is misleading if the program is not correct in consequence. (It was Howden's result [8] that algorithmically-defined coverage techniques are in general misleading.) Other program properties are easy to capture in the functional theory, by imagining that a program P computes other functions like $\boxed{P}$. For example, P's run time is a function $r : D \rightarrow \mathbb{R}$, where $\mathbb{R}$ is the non-negative real numbers. If desired, correctness can be defined to include such properties, for example, that a program never run too long: $\exists B \in \mathbb{R}, \forall t, r(t) \leq B$.

These definitions abstract away from all the hard problems of real programs, specifications, test oracles, test selection, etc.

2.2 Unit Testing

The intuition behind testing as soon as possible in the development cycle is that it should be easier to deal with small units of code, and that if the units work better because their problems have been found and fixed, the system will work better.

Conventionally, units are defined by the programming language used. For a C-like language, they are separately compiled subroutines; for an object-oriented language they might be classes incorporating several methods and private persistent storage. For separate testing of units an input-output convention must be adopted. The parameters of the routine/method, along with any values of global/private-state variables are the obvious test inputs. Returned values, along with any changed state variables, make up the output.

To actually conduct a unit test requires some scaffolding. A driver program must be written to surround the unit with an executable main program that takes external input, to convert that input to the appropriate parameter/state values, and pass them to the unit. The driver then fields the results and converts them back to external, observable outputs. Drivers can be tedious to write, but the process can be automated. Stubs pose a larger difficulty. A unit may very well require the services of other units, and if so it can only be tested in isolation by faking that support. No fakes are really satisfactory, but neither is the alternative of bottom-up integration that tests subsystems, not units.

In Section 3.1 it will be suggested that making each 'unit' an executable program, as is done in the component world, is a good way to study the unit/system relationship.

2.3 Testing in the Presence of Persistent State

Functional-semantics testing theory as presented in Section 2.1 models only programs that do not retain state from test to test. This property can be achieved in practice by requiring that prior to the running of each test case, some kind of 'reset' is done, so that cases are independent and repeatable. Leaving out state goes too far toward simplicity—the model cannot provide insight into state-related testing problems. But testing theorists have been reluctant to complicate their tidy model, and have instead tried to incorporate state implicitly, in two ways.

[1] If specification is defined to be a relation rather than a function, it captures the idea that more than one result may be correct and allows 'don't care' inputs where *any* result is correct. If F is functional, when it is undefined for input u, technically $\boxed{P}$ must also be undefined at u, that is, P must not terminate. These peculiarities of functional specifications must be balanced against the more cumbersome, less intuitive mathematical machinery of relations.

The first state circumlocution comes from software reliability-growth modeling, where a program is executed repeatedly using pseudo-random inputs to measure its failure rate. The pure-function model of such tests can include state variables by artificially adding them to the input collection. Thus when a test-case input is selected by assigning random values to the program inputs, a random value is also selected for each state variable. Unfortunately, this treatment is intuitively wrong because state-variable values cannot be independently chosen. In reality, state values are created in a decidedly systematic way by the program in execution.

The second attempt to incorporate state is more successful. A program with state is treated as a function mapping *sequences* of inputs to outputs. Single test cases are then input sequences that capture successive executions without 'reset' along the way. There is no need to explicitly describe state, since in terms of these input sequences the program semantics remains pure-functional. The 'reset' is done between sequences, which are still independent trials with a repeatable outcome. Although the second approach 'saves' the simple functional-semantics theory, it would be better if state were explicitly modeled. Otherwise state-related testing issues in the theory are no different than problems not involving state, and every practical tester knows that position to be false.

The extension to testing theory in Section 2.4 will include and relate these prior attempts to capture state.

2.4 Formalizing Program State

In addition to the program input domain D and output range R, we introduce an explicit state set H, and give the behavior of program P in two parts, each depending on state as well as input. Retaining the box notation for the 'functional' part of P's behavior,

$\boxed{P} : D \times H \rightarrow R$.

A similar state notation is needed, and since the state maps onto itself, a circle seems appropriate: $\textcircled{P} : D \times H \rightarrow H$. Thus both the program output and a final value for the state depend on an input-state pair $(d, h) \in D \times H$.

Private state, local to a program P, also has a peculiar abstract aspect in specifications. The *concrete state* H itself is directly manipulated by the code function $\textcircled{P}$. The *abstract state* J is an entity that similarly enters specifications. The reason for making a distinction between H and J is that J may be a high-level, intuitive state not available in the programming language. It is then necessary to *represent* values of J by some combination of program entities in H. The connection between the two is established by an *abstraction map* $A : H \rightarrow J$. This process of representation and abstraction is the basis for information hiding, a design technique of the first importance. However, it does no violence to testing theory to identify J with H, and we do that here.

In principle, specifications need not concern themselves with software state at all. To describe what is required of a program does not necessarily require a description of persistent storage it will maintain. However, it often seems impossible to give a formal description of required actions that depend on previous history without explicitly capturing that history.

A *specification* is a (partial) function $F : D \times H \rightarrow H \times R$.

The simplest definition of a program meeting its specification is: A program P is *state-blind correct* wrt specification F iff:

$\forall x \in D, \forall h \in H, (\textcircled{P}(x, h), \boxed{P}(x, h)) = F(x, h)$.

Practical testing explores state-blind correctness by arranging that the program under test be forced into particular states, then various inputs tried there. Yet state-blind correctness is intuitively incorrect, because it accords state the same status as input: it captures the false idea that state can be independently sampled.

To do better, consider sequences of inputs and the sequences of state values that result. It is usual for a program P with state to have a testable *reset* condition that defines the need for P to initialize. (In practice the reset condition is often a missing file that the program creates; to reset, the file is removed by some external agent.) Starting from reset, the behavior of P is repeatable: the same sequence of inputs will produce the same results.

Let P be in a special *initial state* $h_0 \in H$ signifying reset, and consider a sequence of inputs $t = (x_0, x_1, ..., x_n)$. The corresponding states reached by P are:

$h_i = \textcircled{P}(x_{i-1}, h_{i-1}),\ 1 \leq i \leq n.$

Successive functional values of the program are:

$\boxed{P}(x_0, h_0),\ \boxed{P}(x_1, h_1),\ ...,\ \boxed{P}(x_n, h_n),$

that is, the i^{th} output $r_i = \boxed{P}(x_{i-1}, h_{i-1})$. Similarly, the specification prescribes a sequence of states h'_i and outputs r'_i:

$F(x_{i-1}, h'_{i-1}) = (h'_i, r'_i),\ 1 \leq i \leq n,$

starting with $h'_0 = h_0$.

P is *sequence correct* wrt F iff for every sequence of inputs $(x_0, x_1, ..., x_n)$ and the corresponding h_i and h'_i as above,

$(h_{i+1}, r_{i+1}) = ((\textcircled{P}(x_i, h_i), \boxed{P}(x_i, h_i))$
$= F(x_i, h'_i) = (h'_{i+1}, r'_{i+1}),\ 0 \leq i \leq n-1.$ The definition requires P to terminate exactly where F is defined so that the domains match.

The difference between state-blind and sequence correctness is in the states that appear in the definitions. In state-blind correctness, the proof obligation ranges over the whole set H; in sequence correctness only some states in H need be considered, namely those that are specified to occur, and actually do occur, in the order(s) they occur. The latter is a subtle point: if, for example, state $h \in H$ only appears in one sequence, then the proof of sequence correctness can use any properties that have arisen in that sequence; that sequence is the only one that need be tested to cover state h.

State-blind correctness $\implies$ sequence correctness, but not the reverse.

Sequence correctness is essential for correctness proofs, because an intuitively correct program usually fails to be state-blind correct. However, it might seem that for testing one should avoid the more complicated definition and just explore the whole state set. This intuition, although it is often followed in practical testing, is wrong for three reasons: First, testing over all of H increases the number of tests required, which is already overwhelming. Second, successful testing on states that do not actually occur gives a false confidence in a program's reliability. Third, when a test fails on a state that does not actually occur, time is wasted resolving the spurious problem.

The state space H can be divided into subdomains for testing as was the input space in Section 2.1. An arbitrary division of H may contain subdomains that cannot or should not occur, and part of the testing problem is to investigate state questions such as: "Can the program reach states forbidden by the specification?". Following the presentation of experiments in Section 4, Section 5.3 will further discuss testing in the presence of state.

2.5 Deficiencies of this Testing Theory

The notation of Sections 2.1 and 2.4 restricts the input domain and the state domain each to a single variable. In the sequel these will be further restricted to real-number representations. Many of the difficulties that arise in testing are present for one real variable; adding more variables adds to the mathematical overhead without a corresponding gain in insight.

In the sequel the input domain and output range sets D and R will be made to coincide, because it will be of interest to connect programs by passing the output of one as the input to another. In practice there are many variables and types at interfaces between programs, which must be checked for agreement. In a fundamental theory it is better to arrange that the interfaces necessarily match.

The functional computation model excludes any discussion of concurrency or non-determinism, and it is not useful for describing event-driven computation that happens bit-by-bit instead of in tidy input-output pairs.

Decisions about what the theory will describe are a sort of 'clearing of the intellectual underbrush.' We mean to investigate systems and their parts and to focus attention on testing. The theory of this section allows us to do that.

3. SOFTWARE COMPONENTS

The use of standardized parts characterizes successful design in mechanical and electrical engineering; software engineers have long felt the need of a corresponding idea for software.

3.1 Components as Test 'Units'

Szyperski has framed a general definition of 'software component,' taking it to be executable, described only by its interface and black-box behavior, using only local persistent state [15]. The testing theory of Section 2.4 immediately applies to Szyperski components: A component C is a program with black-box behavior $\boxed{C}$ and local-only state behavior $\textcircled{C}$.

The question to be investigated here is the fundamental relationship between testing a complete system and testing its parts. Treating both parts and system as executable programs gives us the cleanest possible relationship between the two.

3.2 Subdomain Testing of Components

When a developer sets out to test any piece of software, it is a daunting task. The input domain is huge and there is time to sample only an insignificant fraction of it. The specification may be complex and usually is not precise. So the tester does not know which inputs to try, and isn't sure what the results should be. At the unit level, the testing problem seems easier because the specification is limited and so is the code volume; this is the rationale behind unit testing. On the other hand, there can be no usage information at the unit level to guide test selection—the whole input domain has to be examined.

The natural response to this problem is to divide the input domain of a unit into subdomains on each of which the behavior is intuitively 'the same,' and to try only a few test cases in each subdomain. In practice, 'the same' is hard to capture. It is common to use so-called functional subdomains based on the specification. Each such subdomain comprises a collection of inputs for which the required behavior has intuitive similarities, e.g., "should clear the screen." But 'should' is not 'does,' which leads to a second kind of subdomain decomposition based on what the program really does, e.g., "clears the screen." Evidently, the program is correct in the intersection of such subdomains, e.g., "should clear the screen and does do so." The program fails outside the intersection, e.g., by clearing the screen when it should not, or by failing to clear it when it should. Unfortunately, a tester cannot know anything about these subdomain intersections except by testing. Neither a specification-based subdomain nor a program-based one can be depended upon to have only inputs that are really 'the same': both may contain some success points and some failure points. If failure points happen not to be selected, a subdomain test will succeed and be misleading according to the definition in Section 2.1.

Testing theory succinctly describes this unfortunate situation, but offers only negative results such as: "Don't trust subdomain testing." Some studies do a little better, for example: "Subdomains formed by arbitrarily splitting existing ones are probably not an improvement" [5]. The theory stays in contact with the real world by capturing and examining subdomain testing abstractly, but leaves the difficult job of finding good subdomains and investigating them to see if they are misleading, etc., to practice.

When the software under test is a component, there is some justice in telling its developer: "You are responsible for testing this component. Do the best you can with the difficult process of subdomain testing." Components are made and sold partly on the basis of their quality. Working hard to convince oneself (and one's customers!) that the component does meet its specification, that its tests are not misleading, is in the developer's interest.

3.3 Subdomain-approximation Adequacy

The quality of a subdomain-based unit test of a component C can be measured by the degree to which it approximates the actual behavior of C. If $\boxed{C}$ and $\textcircled{C}$ were functionally 'the same' across each subdomain, then C would be accurately described by step functions, constant on each subdomain. For numerical domains, the accuracy of such a subdomain-defined approximation can be quantified. The best constant values to choose are the averages across each subdomain, for which the approximation error is the root-mean-square deviation from actual values. A good unit test (i.e., good subdomains) has small approximation errors. The subdomains and averages measured in them define approximations, and test quality is measured by the extent to which these agree with the actual C.

In conventional unit testing, subdomains are chosen haphazardly. Perhaps some failures are found by tests covering them, but no assessment of subdomain quality is made. What is proposed here is more stringent: to measure how good an approximation to actual behavior the subdomains define. In the next section we study the effect of getting the subdomains right or wrong.

4. TESTING EXPERIMENTS

This section reports on experiments testing component and system implementations. When we began to study component-based software, we used 'toy' programs as components but soon abandoned them. The experiments revealed nothing because the toy programs' behavior was too simple. Full-blown 'real' programs might have shown more, but their interesting behaviors are hard to control and they are beyond the capabilities of simple tools. We settled on artificial implementations as the best compromise. These components are Perl programs whose functional behavior is easy to adjust. For example, to give a component discontinuous behavior, an IF statement selects between two output expressions; to move the point of discontinuity requires only changing the conditional expression. We found that artificial components could be adjusted to display cases with analogs in real software, but without introducing too much complexity.

4.1 Prototype Tools

To study components and their testing, a set of prototype tools was implemented. These tools allow an experimenter to test any executable code unit having one floating-point parameter and one floating-point state variable and returning a floating-point value. The tools work in two modes:

Component Analysis: The experimenter supplies a set of subdomains for an implementation C. The tools find an approximation to the behavior of C by sampling each subdomain and computing there the average output value and the root-mean-square error between the actual execution values and the average. This is the support needed for unit testing. When the experimenter is satisfied with the subdomains and errors, the component test analysis is complete.

System Synthesis: In addition to a collection of component implementations and their approximations from component analysis, the experimenter supplies a control-structure system design for a system. This design connects the components using Boehm-Jacopini structured constructs of sequence, conditional, and iteration [2]. The output of one component is taken as input to the next. The tools can execute the system using the actual component implementations, but can also 'execute' the subdomain-based approximations by table-lookup[2]. Thus the system predictions (from the approximated components) can be compared with the actual behavior.

The rationale for carrying a unit-test approximation of component behavior to the system level is that this is an ultimate assessment of the unit-test quality. If the unit-test information is accurate enough to make good predictions about an arbitrary system, it is certainly adequate. Experiments reveal the relationship between unit- and system testing.

4.2 Stateless Experiments

Because state introduces its own difficulties and insights, we begin with analysis of a simple stateless component C_0. Its domain and range are arbitrarily chosen to be $D = R = [0, 100)$. $\boxed{C_0}$ is

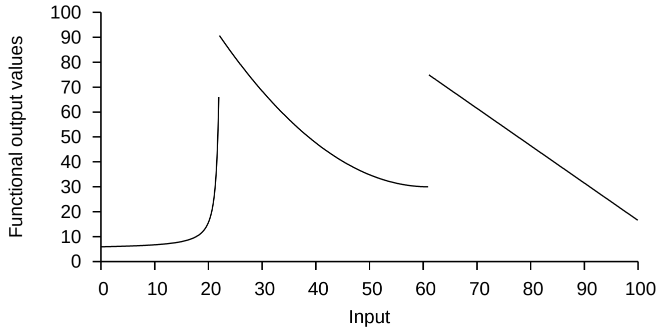

Figure 1: Measured functional behavior of component C_0.

discontinuous and rapidly varying as shown in Figure 1.

Figure 2 shows a portion of Fig. 1 for inputs in $[18, 28]$ and two subdomain-based approximations to $\boxed{C_0}$. One approximation is obtained by dividing the domain into 320 uniform-size subdomains; the other uses 140 hand-adjusted subdomains. In Fig. 2 the approximation curves have been shifted vertically for clarity—the vertical scale applies only to the middle curve. Each step in the approximations is technically a discontinuity, but vertical lines are shown to better group the data. The hand-adjusted subdomains were created by starting with 80 uniform-size subdomains, noting the r-m-s error in each, and subdividing those where the error was 2% or larger. A component tester might use such a procedure to get good subdomains by trial.

Table 1 summarizes the two approximations of Fig. 2. The column headed ">2%" counts subdomains in which the r-m-s error

[2]A theory has been developed [7] that allows system properties to be calculated directly from the component approximations, but the table-lookup 'execution' is equivalent (if less efficient).

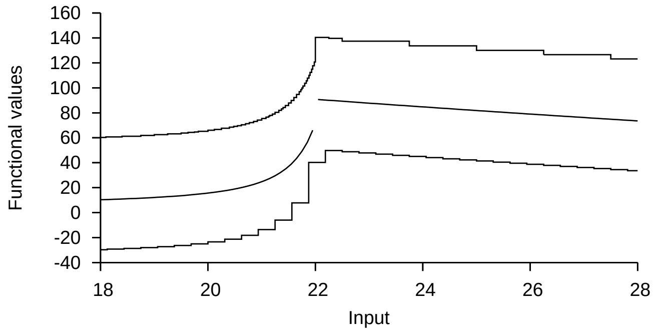

Figure 2: Measured behavior of C_0 (middle curve) approximated using uniform-sized subdomains (curve displaced down) and using hand-adjusted subdomains (displaced up).

	Subdomain	Functional r-m-s Error (%)		
Approximation	count	Ave	Max	>2%
uniform	320	0.6	30.4	11
hand-adjusted	140	1.2	1.95	0

Table 1: Comparison between uniform and hand-adjusted subdomain approximations to the behavior of C_0.

exceeds 2%. Although on average both subdomain divisions are good approximations, the hand-adjusted one has fewer bad subdomains despite using fewer than half as many subdomains.

4.2.1 System Synthesis from Components

The prototype tools can synthesize system properties for an arbitrary control structure built up from sequences, conditionals, and loops. The simplest and most revealing structure is sequence. When two components are composed so that the first invokes the second, it makes the greatest demand on unit testing of both components. The primary reason is that the input-domain distribution of values seen by the second component when in the system is distorted by the functional behavior of the first component. An adequate unit test of the second component should take this into account, but it cannot, since components are unit tested in isolation, without knowledge of the system that will later be formed. The only defense against getting it wrong at the system level is a good choice of subdomains at the component-test level.

The simplest experiment that can be performed is to put C_0 in sequence with itself to form C_0; C_0. Other experiments with a variety of system structures [6] have shown that a well chosen C_0 using the simplest system structure reveals the same interesting features as does a more complex system. Figure 3 shows the behavior of this system (sampled 500 times), which is surprisingly complex. One of the component discontinuities (near 22) has apparently disappeared and four new discontinuities have appeared. Figure 4 displays the functional predictions from the unit tests using uniform- and hand-adjusted subdomain approximations, expanded for the dotted-boxed region of Fig. 3. The superiority of the approximation using hand-adjusted subdomains is evident—the uniform-subdomain prediction and the system testing of Fig. 3 both miss a spike around before input 22 (the component discontinuity there did not disappear, and there are five, not four, new discontinuities). The 140 test points from the hand-adjusted subdomains do the best job of system testing, and it is as good with those points to use the approximated component code as the actual system.

This investigation of component unit-test quality reflected at the system level suggests a new paradigm for system testing. The component subdomains have been adjusted by their developers to minimize (using whatever test resources were available) the error in approximating component behavior. The synthesis experiment shows that these subdomains are good ones for system test and to use them with the component approximations is very cheap—only one test point per subdomain is required[3]. This novel way of system testing will be further discussed in Section 5.2.

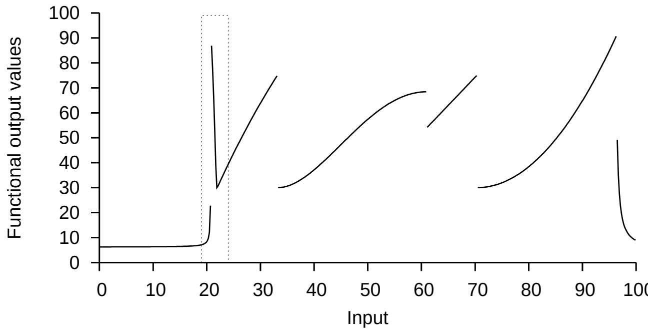

Figure 3: Measured behavior of C_0; C_0.

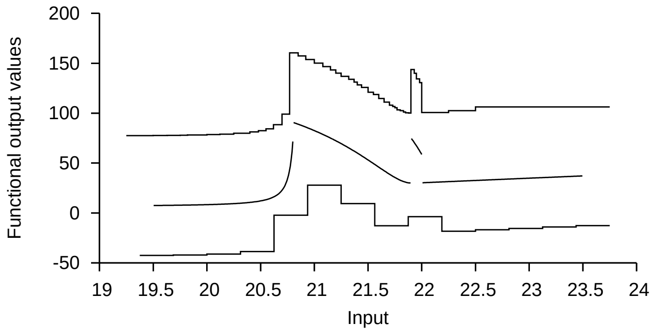

Figure 4: Functional behavior of C_0; C_0 (middle curve) approximated using uniform subdomains (shifted down) and hand-adjusted subdomains (shifted up) as in Fig. 2.

4.3 Components with State

When a component C has local persistent state, unit-level testing can still probe its behavior. By selecting a test point (x, h) in the (input $\times$ state) space $D \times H$, and executing component C, the tester obtains $\boxed{C}(x, h)$ and $\textcircled{C}(x, h)$, that is, the output functional and state values for C on input (x, h). From the results of many such test pairs two surfaces can be plotted showing how C behaves. Component subdomain analysis uses two subdomain collections, one subdividing the input space and the other subdividing the state space. These collections can be systematically sampled and values averaged across each subdomain to yield a surface of step plateaus that approximates the actual functional surface, and another that approximates the state surface. Proceeding in this way is the testing analog of state-blind correctness (Section 2.4), because it explores the state space as if it were an independent dimension, without regard for how C really sets its state. For convenience, we also call such tests 'state-blind.'

Alternately, the tester might reset C, supply an input sequence, and observe the resulting points on the behavior surfaces. Many such sequences allow the two surfaces to be plotted. Each input/state pair that arises in a sequence falls in some rectangular

[3]By calculating system properties from the approximations as described in [7], one would not have to execute even these few system tests. For example, the upper curve in Fig. 4 can be predicted from the upper curve in Fig. 2 without execution.

subdomain; keeping track of pairs by the subdomain produces a collection over which an average is taken to get the approximation. This procedure is the testing analog of sequence correctness, which we call 'sequence testing'.

The difference between state-blind- and sequence testing is clearly shown if the latter plotted surfaces have gaps, above subdomains in which no sequence-test pairs fall.

To illustrate, we modify the artificial component C_0 from Section 4.2, to create component C_1 that uses state in some common ways[4]. C_1 uses states $H = [-1, 1]$ to remember two things. The sign of state $h \in H$ signals one of two 'modes.' In the 'positive mode' $\boxed{C_1}$ is similar to $\boxed{C_0}$. In the 'negative mode' C_1 has a simpler functional behavior, constant in the input. The magnitude of the state value $|h|$ scales the output, but only to factors between 0.5 and 1.0 and only in the positive mode. The state itself is changed by inputs in the range $[0, 20]$. Any input $x \in [10, 20)$ sets the scale factor to $20/x$; any $x \in [0, 10)$ toggles the mode.

Component C_1 mimics in a simple way the state usage of a command-line text editor. An editor has the two modes 'input', in which almost every input character is accepted and stored, or 'command' mode, where input characters cause editing actions. A special input toggles the editor mode. The two different output functions of C_1 for positive and negative state values model this behavior. Along with a binary mode value, an editor also uses state values to do its work; for example, an editor has string-storage state to remember prior search and substitution strings in its edit mode. The use of state as a scaling factor in C_1 (but only in one mode) models a simple version of this.

First we apply state-blind testing to C_1, using a state set $H = [-1.2, 1.2)$ and 5880 test points drawn systematically from the cross product space $[0, 100) \times H$. Figure 5 shows the functional output surface for C_1. At the front of the graph the mode (state)

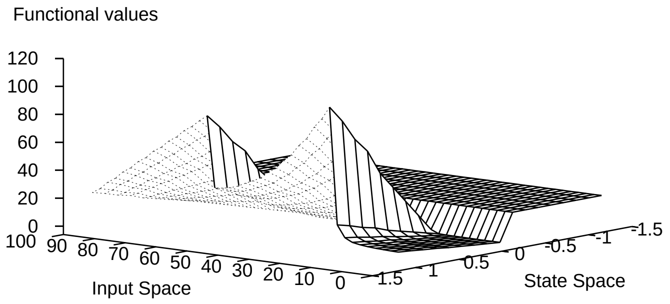

Figure 5: Measured state-blind functional behavior of C_1.

is positive, where the C_0-like behavior similar to Fig. 1 appears, scaled down as the state goes toward 0. At the rear of the graph is the constant, not-scaled, negative-mode behavior[5].

Figure 6 shows state behavior for C_1. For inputs above 20 C_1 makes no state changes, that is, the state function is identity. Figure 6 shows this as a sloping plane viewed from the underside. In the input interval $[0, 10)$, C_1 reverses the state sign, that is, the function is an identity plane with slope -1, at the right of Fig. 6. C_1's state behavior in $[10, 20)$ where the scale factor is adjusted is harder to visualize and describe. Perhaps what Figure 6 best illustrates is that people are not good at visualizing state functionally.

[4]Even with a simple artificial component, it is difficult to intuitively grasp the state behavior, which must be complex enough to pose interesting difficulties in component analysis (this Section) and in system construction (Section 4.3.1).

[5]The angled plane joining surfaces at state 0 is an artifact of the plotting.

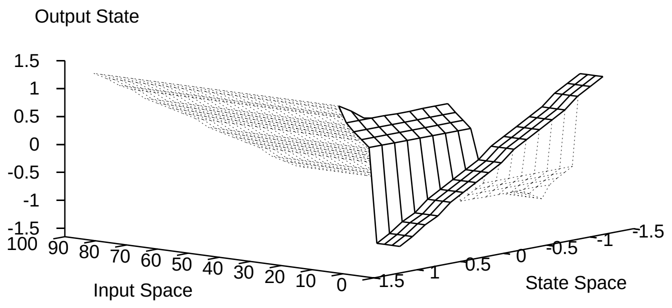

Figure 6: Measured state-blind state behavior of C_1.

Figs. 5 and 6 do not capture how C_1 really behaves. Its states do *not* stray outside $[-1, 1]$, and therefore scale factors greater than 1.0 are never applied as they appear to be in Fig. 5. The restriction to avoid scale factors in the $[0, 0.5)$ range *is* actually observed, so a band of states in the middle of each figure *does not* occur. It is not that component C_1 would not behave as the figures show if it were placed in the states shown. It does in fact behave that way: the figures are execution measurements. What is wrong is the state-blind testing samples: some of the states should not be sampled because they are impossible.

Sequence sampling gives an accurate assessment of C_1. Figure 7 shows the output using 130 random input sequences starting

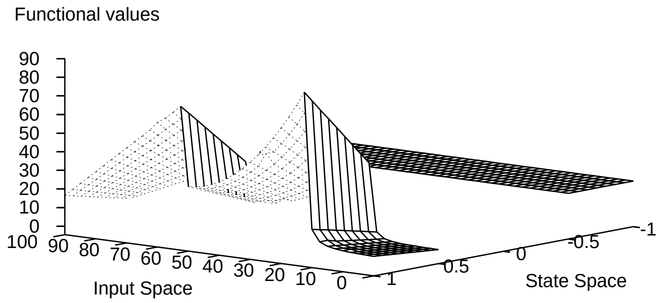

Figure 7: Measured sequence functional behavior of C_1.

from reset. It can be seen that the state is confined to $[-1, -0.5) \cup (0.5, 1]$, and scale factors are limited correspondingly. About 34% of the test points in Fig. 5 are wasted on states that never arise.

If the actual state behavior of a component is not understood at unit-test time, it leads to an explosion of problems at system-test time. For example, if Fig. 5 is taken to be C_1's output and C_1 is placed in a system, the spurious scale factors above 1.0 may lead to apparent system failures that are even more time-consuming to track down than real failures.

To approximate the behavior of C_1, 392 subdomains were created in $[0, 100) \times H$, and the random test sequences tracked by subdomain. No points fell in 132 of these subdomains. Figure 8 shows the approximation, which is a surface of plateaus above the subdomains that arose. The weighted average r-m-s error in Fig. 8 relative to Fig. 7 is 2.2%.

4.3.1 Systems from Components with State

When components with persistent state are combined into a system, the resulting system behavior is more difficult to display and understand than in the stateless case of Section 4.2. For state-

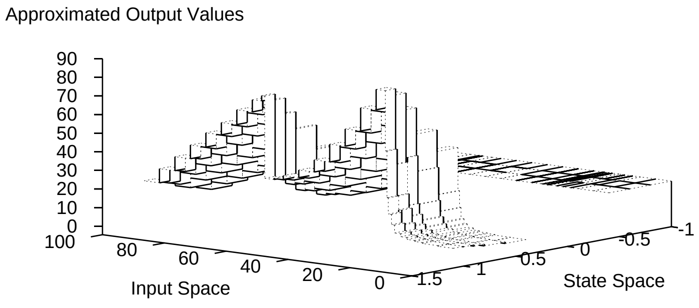

Figure 8: Approximation to functional behavior of C_1.

less components, systems take exactly the same form as their constituent components: stateless systems have a single input, a single output, and they are described approximately by step functions on a collection of subdomains (derived from the subdomains of their components). In contrast, a system built from components with private persistent state(s) differs in form from a single component. The system state is a cross product of the component states, a different entity for different numbers of components. This complex system state is a challenge to grasp intuitively and to present graphically. To display system behavior for n components would require $(n+2)$-dimensional graphs: $(n+1)$ independent dimensions for states and the input, and one dependent dimension for the result.

It is not obvious how state-blind testing of a system with state should be carried out. In contrast, sequence testing of a system is straightforward. System reset means resetting all the components. An input sequence is supplied to the first component in the system. The connections between components route inputs to the other components, creating sequences for them. (Not every component receives the same length sequence: conditionals split sequences between their alternatives and loops expand one input into a subsequence.) As it receives inputs, each component creates and maintains its own state. Thus sequence testing of a system with state is just the same as sequence testing one component. We will use only testing with randomly generated input sequences in the remainder of this paper.

If each component's code within a system is replaced by a step-function approximation, then executing these (by table-lookup) approximates the system behavior. Comparing sequence tests of the actual behavior with the same tests on the approximation system is the ultimate assessment of quality for the subdomain-based tests of the components.

Unfortunately, even though systems and approximations of systems can be easily sequence tested, the problem of visualizing the results remains. To see exactly what is happening requires higher-dimensional Euclidean spaces than humans can perceive. The best that one can do is to define some measure of the complex, composite state of a system to use in displays. Integer-valued states can be coded to a single value with a bijective pairing function [14], so that each different cross-product system state has a unique value. Pairing functions necessarily introduce an undesirable nonlinear distortion, because two values must be paired to a value something like their product. For real-valued states, no bijective pairing is possible. In one special case a good intuitive display is possible: if only a small number of state values arise, they can be labeled with the actual system cross-product state value.

We construct an artificial example in which state plays a revealing role, using the 'editor-like' component C_1 described above with a 'front-end-like' component C_2 that controls C_1. C_2 uses its state only to keep track of a few discrete modes. It 'shadows' the two modes of C_1 so that it knows what C_1 should do if invoked. It has a mode to invoke C_1 (but C_1 is not permitted to change state), another which explicitly forces C_1 to toggle its state (but only a limited number of times), and another which models a user dialog not involving C_1. Because C_2 is used as a conditional test, it needs two (stateless) filter programs C_3 and C_4 to adjust values passed on. Figure 9 is a flowchart of the system control structure.

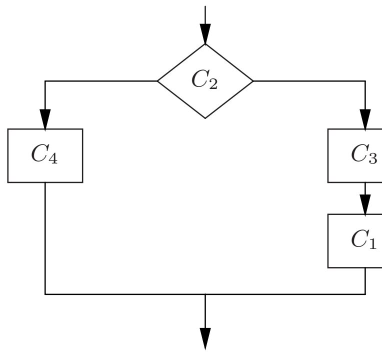

Figure 9: A simple system built from components.

For completeness the detailed state- and functional behavior of C_2 measured by sequence testing are shown in Figs. 10 and 11. (Note that only eight states actually arise.) Since the modes are

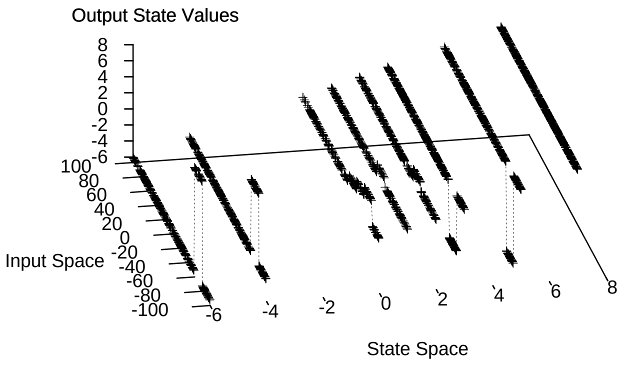

Figure 10: Measured state behavior of C_2.

discrete, no surface is fitted to the behavior, but the measured points are quite dense, and modes are traced by dotted lines[6]. Consider one of C_2's eight modes, state +3 in Fig. 10 (third from the right). In this mode C_2 mostly believes that C_1 will be in its positive mode. C_2 stays in state +3 except when it receives a negative input in $[-80, -60)$, then going into state -4. In Fig. 11 again look at state +3. For positive inputs C_2's output is 1, that is, it takes the right branch in Fig. 9. For negative inputs it takes the left branch in Fig. 9 (output 0), except for input in $[-80, -60)$ which again goes right (output 1). The actions in $[-80, -60)$ are the case in which C_2 is instructed to force a mode change in C_1, which occasions the described control flow and state change.

Table 2 summarizes the informal component specifications. Because in the system C_3 avoids the output range $(0, 20)$, it never allows C_1 to adjust its scale factor. C_1 is thus confined to two modes (1 and -1) that model 'input' and 'command' modes of an editor.

[6]In subsequent graphs the projection may be rotated to give the clearest view of the surface.

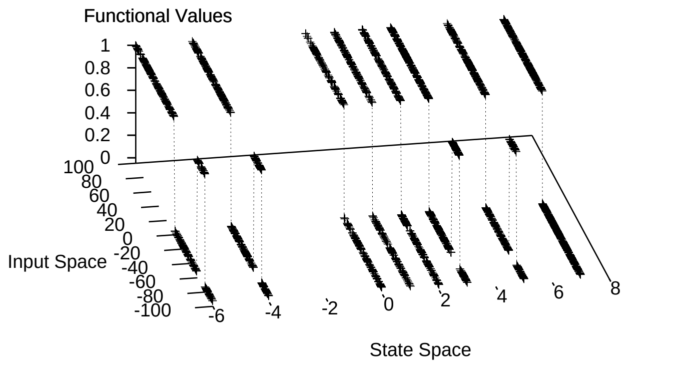

Figure 11: Measured functional behavior of C_2.

Component	State?	Type	Informal Specification
C_1	yes	imp	See Fig. 7
C_2	yes	cond	See Fig. 11
C_3	no	imp	Compresses positive inputs to the output range [20,100); any negative input gives output 0.
C_4	no	imp	Absolute-value function.

Table 2: Components used in the system of Fig. 9.

Figure 12 displays the system state behavior, each composite state labeled with a pair of state values for (C_1, C_2) in order. For

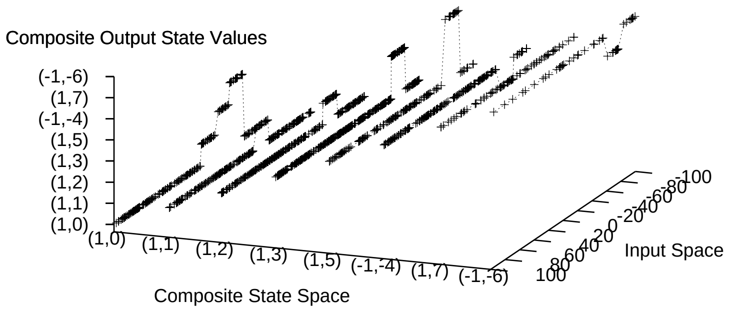

Figure 12: Measured state behavior of the system of Fig. 9.

example, in the mode at the left of Fig. 12 the composite state is (1,0): C_1 is in state 1 while C_2 is in state 0. Most system inputs leave the composite state unchanged. It can be seen that the state shadowing is working: in the composite states that arise the two component states always have the same sign. System state behavior almost mimics that of C_2, but one difference is of interest: In Fig. 10 the state-0 curve (third from the left) has a linear segment around input 0, while in Fig. 12 the corresponding curve (left-most) has three discrete steps in the segment. In the system, some feasible states of the component have become infeasible.

Figure 13 shows the system outputs that occur in each of the eight system modes. For negative system inputs, the output is the absolute value, reflecting the 'user dialog' through C_4. For positive inputs, the output is that of the 'editor' C_1, which is constant in negative modes (right-most and third from right in Fig. 13) and the shape of Fig. 1 in positive modes (the other six modes).

If this were a real system, Figs. 13 and 12 would be diligently studied by the system testers to see if the behaviors meet system specifications. For our purposes it is enough that the behavior is complex, because we now wish to try the approximation experi-

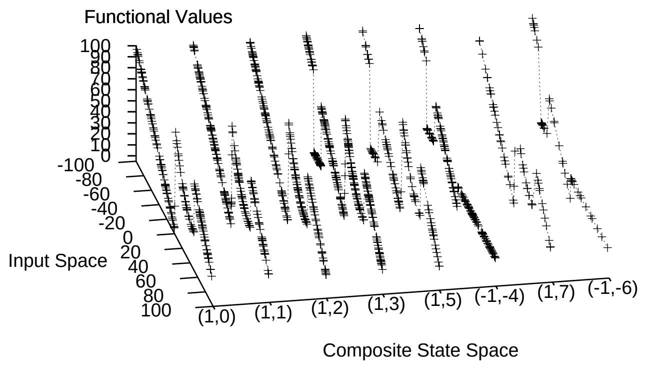

Figure 13: Measured functional behavior of the system of Fig. 9.

ments that in the stateless case showed that component measurements were sufficient to predict system behavior.

The four components C_2, C_3, C_1, and C_4 are described by subdomains (in the input space for the stateless ones and in the cross-product of input and state for the others). Approximation plots like Fig. 1 and Fig. 8 are obtained, and the subdomains refined until the r-m-s error is just over 2% for C1 and C3, about 13% for C4, and for C2 the approximation is perfect[7]. Then the approximation tables are 'executed' in the system combination of Fig. 9 and the results compared with the execution measurements from Fig. 13.

Again there is difficulty is displaying potentially hyper-dimensional data. The trick of explicitly labeling a few system states in graphs like Fig. 13 isn't as useful for subdomain-based approximations, because state subdomains are rectangles (in general, N-dimensional boxes for a system with N components that have state) that do not necessarily map into connected lines in the plotted state dimension. Nevertheless, Fig. 14 shows the output behavior of the approximate system formed from the approximated components.

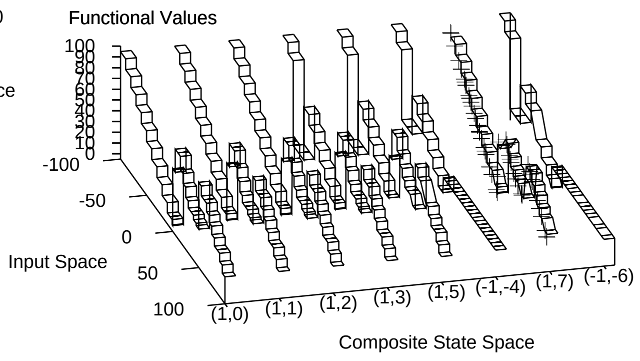

Figure 14: Approximation to Fig. 13 using approximating components.

The plateaus' sides in the state dimension are arbitrarily placed to span each mode. In the (1,7) mode (second from right in Fig. 14), the actual system data is plotted for comparison.

Displays like 14 quickly become incomprehensible in all but the simplest examples. Recalling that there is no difficulty in obtaining data about the actual and approximate behaviors, just in displaying them, suggests a different method of comparing the two that generalizes to any number of system components. Figure 15 displays one sequence of test data like those used to plot Fig. 13 and Fig. 14. In Fig. 15 functional values appear as vertical lines above

[7]The choice of subdomains is a compromise between accuracy and values that produce good graphical displays.

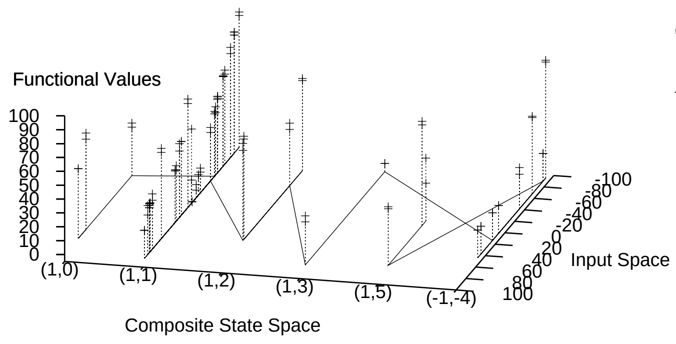

Figure 15: Behavior of the system of Fig. 9 for one input sequence. The two points plotted above each vertical line are the predicted and actual values.

a path in the base (input × state) plane. The path describes one sequence of length 50 used to test the system. The sequence of input points given to the actual and approximate systems is the same, but the state sequence that results might be different (which would result in different paths in the base plane), and the output may differ (different impulse heights). For this sequence, the actual and approximate states coincided, and the output values differed only slightly in most cases, as shown by two crossbars at the top of each impulse.

A plot like Fig. 15 quickly becomes so crowded that the approximation quality cannot be visualized, but the data can be analyzed in a way that generalizes to arbitrary systems: For the entire set of sequences tested and each test point in each sequence, compute the distance between actual and approximate points in the (output × state) space. For the test displayed in Figs. 13 and 14, the average error distance is 10.7%. Relatively poor subdomains were chosen for C_4 in this case to make the figures clearer; if the subdomains are divided until all component r-m-s errors are below 2%, the average error distance for the system is 2.5%. Thus small errors in approximating its components leads to small errors in the approximate system. In this example the approximation tracks the eight modes in the state perfectly. In a system with a more complex state, state errors in the approximation can lead to larger discrepancies.

5. DISCUSSION: TEST UNITS OR TEST SYSTEMS?

Testing theory as presented in Section 2.4 and component-based experiments in Section 4.3 expose the difficulty of testing any program. This difficulty can be summarized as one of finding and examining subdomains of the (input × state) space of the program, subdomains that characterize its behavior and do not mislead the tester into thinking all is well when it is not. Successful tests can mean that software is understood and under control; all too often they only mean that the test coverage is dreadful.

Only a few simple examples have been examined, but they suggest an unorthodox idea: creative, human-intensive testing should be done primarily at the unit level, but done far more extensively than in current practice. Then at the system level, testing can be a more routine checking activity. The system-level tests will be judged against the system specification, but the test points used will be those that were found to well approximate the unit behavior. Furthermore, the actual system need not be executed; it can be replaced by an approximate system of component approximations.

5.1 System- and Unit Testing

The twin facets of emphasizing unit- or system-testing are these:

Unit. At the unit level testing is knowledge-driven. The unit is small, it can be studied in detail, and the tester can be in good intellectual control.

The drawback at unit level is a complete ignorance of the way in which the unit will later be employed, particularly in component development, since future applications may vary. The technical form taken by this problem is that in application the unit will face a usage profile that depends strongly on both the system usage and the system structure, both unknown. So no unit-test profile can be adequate.

System. System-testing's great advantage is that the tested program and its usage are really the ones of interest, so there is no difficulty about testing the right things. Any information that is needed for an accurate test is in principle available.

The payment for operating in full reality is that system testing is seldom an activity in intellectual control. The specification for a complete system can be extremely complex and may have significant ambiguities or incompleteness. The code has been assembled from many sources into a whole whose complexity is overwhelming, beyond what any person can fully grasp.

A concise way to describe these two sides of the testing problem is to say that in unit test we know all about a program but aren't sure what's important; in system test we know what's important but it is hard to be sure of anything about the program. In deciding where to expend more effort, the clear engineering choice is: at unit level. Engineers need to know what to do and how to do it routinely [1, 16]. A principle is only useful to an engineer if it comes with a way to use it. There is of course another side to this engineering bias: prescribed activity has to be validated in principle, lest the engineer engage in ineffective make-work.

Subdomain testing itself is a solution to the lack of eventual usage information at the unit/component level. Technically, the problem is the disparity between the input profile used for unit testing, and the profile that the component will actually face when it is in place in a system. If these profiles are different, testing at the unit level can be expected to mean nothing at the system level. What a component sees in place depends on: (1) The system profile, and (2) The component's place in the system control structure and the behaviors of the other components. Any profile can be thought of as a weighting of subdomains. Indeed, the practical profiles long advocated for software reliability engineering by Musa [12] are precisely such weightings. The unit tester, using full information about the unit specification and its code, decides on a set of subdomains, based on making the unit behavior 'the same' on each subdomain. If this subdomain breakdown is successful, then the weights later placed don't matter—the unit behavior has been captured accurately for all possibilities.

This rosy view of the efficacy of unit testing may of course go wrong. If there is an error in 'sameness' of some unit-test subdomain, and the profile that component sees when in a system weights this subdomain heavily, then the system can fail because this component fails, having received an input that was not represented in its unit test.

5.2 A New System-testing Paradigm

We propose to use the results of unit-testing components as a way to structure system testing, reducing its cost both in terms of execution time and the creative human effort required.

When a system is assembled from components suppose that each component has been unit tested as were the examples in Sections 4.2 and 4.3, which brings into existence a collection of artificial components, each a subdomain decomposition of its input domain and a step-/plateau-function approximation defined on these subdomains. We suppose that the component developers have expended considerable effort to make these approximations as close to the actual components as they can, using tools like those described in Section 4.

System testing is then conducted not with the actual components, but with the approximations. The advantages are:

Speed. Each approximation component executes by table-lookup no matter how slow the actual component might be.

Coverage. It is unnecessary to test more than one point in any subdomain. The subdomains are by definition homogeneous and any point is like any other.

Test selection. Any test selection method can be used, but the random sequences of Section 4.3 are a choice requiring very little creative effort.

Test adequacy. The quality of a system test can be mechanically measured. On the assumption that the unit-test approximations are of good quality, the only dangerous situation occurs when some subdomain of some component is hit much more frequently than others as the system executes. Conventional instrumentation can measure the subdomain hit rates. (The tools used in Section 4 provide this feature.)

The sole disadvantage of using the approximated components is nevertheless a substantial one: The system-test results may be wrong. However, they are not likely to be misleading in that tests falsely appear to succeed. The approximate components' behaviors are distortions of their actual behaviors, which should make the approximate system fail to meet specification when the real-component system would succeed or fail in a different way. So the second part of the scheme proposed here is to execute the actual system on any test cases where the approximation system fails and to analyze only actual-system failures.

5.3 Component/System Persistent State

When programs have persistent state, as almost all useful programs do, it magnifies the difficulty of testing. One measure of this is that the number of relevant subdomains is the *product* of the numbers in the input- and state domains. This explosion in needed coverage is strong support for pushing testing to the unit/component level where there is some hope of keeping intellectual control. It is crucial in confining the creative part of testing—devising and exploring appropriate subdomains—to component level that state be also confined to components[8].

Imagine for a moment that a system were permitted global state, persistent values that could be used and set by all of its components. How could those components be unit tested? To partition state into appropriate subdomains for test requires knowledge of how a unit will use that state. For state confined to the unit this is difficult enough, as the experiments of Section 4.3 show. If state were global, accurate partitioning for unit testing would be impossible, because it would depend on all the other units that will eventually make up a system. The problematic nature of global state is really just another form of the testing-profile problem. Subdomains at the unit level must capture whatever a unit might do; if its state can be changed in arbitrary ways by unknown agents (the other parts of a future system) unit subdomains make no sense.

[8]It is also the conventional wisdom that persistent state should be local to the units that make up a system. 'Information hiding' is the name Parnas gave [13] to this design idea, which is seen in most versions of object-oriented design, and reflected in Szyperski's definition of a software component [15].

Thus a proper state for a system made from components is the cross product of the local states of these components. The state subdomains for system testing are then defined as cross products of the component-test subdomains. Just as some input subdomains are explored in unit testing that can never occur once a particular system is built, so some feasible component-state subdomains may never participate in a particular system's actual state. That is, the cross product of the feasible component-state subdomains is typically a strict superset of the feasible system states. (An example was described in the discussion following Fig. 12.) It is hopeless to try to explore by testing a complex system state in a state-blind way; the payment is to do more work at unit level than is strictly needed for any particular system that will follow later.

The theoretical study of exploring state by testing has barely begun. The testing examples in Section 4.3 and Section 4.3.1 are program-based: states investigated are those that *do* arise in executing C_1, C_2, and a system on input sequences. In practice, it is usual to look at specification-based states. Unfortunately, conventional specification-based testing practice is particularly deficient with regard to state coverage. Here's what is usually suggested:

> A collection of specification-based states is devised by hand. There are two ways to try putting the program into one of these specification states: (a) Invert the abstraction mapping (A in Section 2.4) to find by hand a representative implementation state and externally create this persistent value; (b) Devise a test sequence by hand that should according to the specification place the program in the specification state[9]. Program behavior is explored by a test collection in which each test point begins in what should be a specification state.

The deficiencies in such a scheme are clear. (1) The specification may have unreachable states, perhaps obscured by a state-machine formalism that does not capture data dependences. Testing an implementation is the way least likely to discover specification problems. (2) The program states and state transitions may fail to mimic the specification, either by design or because mistakes were made in implementation. Conditioning with an input sequence that *should* lead to some imagined state ((b) above) then takes the program to an unknown actual state. (3) The abstraction mapping A is usually many-to-one, so an implementation state selected to represent a specification state ((a) above) may not be one that is typical or even feasible. (4) The set of program states that can occur is not explored; they may include states with no specification equivalent, reached in unimagined ways.

It is not an unfair summary of current specification-based state exploration to say that it is unexamined. The tester continually confounds what the program should do with what it does do, and never really knows what states have been (or have not been) tried.

5.4 Quality of System and Component Tests

The experiments of Section 4 support the position that good system predictions can be made from the results of unit testing, as described in Section 5.2. We do not recommend interpreting these results to say that system testing is superfluous—that it only repeats a poor fraction of the unit-level test work. It is, however, instructive to imagine an independent subdomain-based test being conducted on a system, and to trace it into the unit-test results. Because things

[9]The test sequences can be mechanically generated if the specification uses a state-machine formalism. The machine itself defines 'specification state'.

are so complicated at the system level, the subdomains used are likely to be much larger and less related to behavior than ones at the unit level. Whatever system test points come from these diffuse subdomains will find their way into the various unit subdomains, but coverage of unit subdomains will be sparse. Thus the likelihood that anything new will be learned about the components from such a system test is low. The argument establishes that the imagined system test may be a poor one. What it leaves out is that the results will be judged by the system specification and only such judgments stand a chance of catching system-design flaws. Unit tests of components can do nothing whatever to expose mistakes in combining those components to realize a larger purpose.

We recommend the following interpretation of our experiments: System testing is essential, but it can be better carried out based on unit-test results as described in Section 5.2. An example appears in the experiments with the stateless component C_0 in the series system $C_0; C_0$ in Section 4.2. Figure 4 shows that there is some messy system behavior near input 22, behavior that arises only at the system level. Without conducting the system test, this behavior would never be seen; conducting the system test without using the unit results (as in Fig. 3) would also fail to expose it.

The success of the system testing scheme proposed in Section 5.2 depends on the accuracy of the system approximation using the approximate components obtained from unit-test analysis. If the proposed scheme is to be used in practice, there can be no validation that the unit approximations are good enough; they must be trusted. A few experiments are far less convincing than a proper theoretical error prediction, but the latter has not yet been attempted.

5.5 Related Work

Karl Meinke has proposed a very clever different testing-based program approximation for a different purpose [10]. He requires a first-order formal specification for program P and searches for a test point on which P fails, roughly as follows: Given a function f that is a piecewise polynomial approximation to $\boxed{P}$, a point x can be algorithmically found such that $f(x)$ is incorrect according to P's specification. Perhaps x causes P to also fail; if so, Meinke has succeeded. If not, he adds $(x, \boxed{P}(x))$ to f to form f', a more detailed approximation. The process is iterated until either a real failure is found or the piecewise approximation becomes so accurate that the tester believes there are no real failures.

Meinke's procedure produces a set of subdomains (the 'pieces' of the approximation) that can claim to objectively capture a 'functional' breakdown of P's domain, itself an important theoretical accomplishment. The work presented in this paper does not require a formal specification, but the payment is that our tester must construct subjective subdomains by hand. The experiments of Section 4 use piecewise approximation with constant steps. The tools we have implemented can also use linear pieces (not quadratic because linear is the highest degree at which the composition of two piecewise approximations is a same-degree approximation).

6. SUMMARY

The functional-model testing theory has been extended to make state explicit and subdomain testing has been examined in the light of the theory. The setting of software components combined into systems is a good one for study, since its 'units' are complete programs each with local persistent state. Prototype tools have been implemented to study components and their composition. Experiments show that it is possible to make good system-level predictions from unit-level tests, but only if the latter are much more extensive than in current practice.

The conclusion drawn from the theory and experiments is that component testing, if properly done, can support and simplify the system-testing process. The difficult parts of system testing are replaced by perhaps equally formidable difficulties at component level, but the difference is that understanding is far better at component level and one can hope to quantify the system software quality that will result, as it has not been possible to do with conventional system-level testing.

Acknowledgments

Zheng Tu, Milan Andric, Ben Buford, John Christmann, Alex Corrado, Paul Draghicescu, and Michael Plump worked on prototype tools. I thank Sharon Flynn for useful discussions of the state problem. One anonymous referee alerted me to Meinke's paper [10].

7. REFERENCES

[1] William Addis. *Structural Engineering: The Nature of Theory and Design*. Ellis Horwood, 1991.

[2] C. Boehm and G. Jacopini. Flow diagrams, turing machines, and languages with only two formation rules. *Comm. of the ACM*, 9:366–371, 1966.

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

[4] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, 1975.

[5] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Soft. Eng.*, 16:1402–1411, 1990.

[6] Dick Hamlet, Milan Andric, and Zheng Tu. Experiments with composing component properties. In Wallnau [17].

[7] Dick Hamlet, Dave Mason, and Denise Woit. Theory of software reliability based on components. In *Proceedings ICSE '01*, pages 361–370, Toronto, Canada, 2001.

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

[9] S. C. Kleene. *Introduction to Metamathematics*. Elsevier, 1980.

[10] Karl Meinke. Automated black-box testing of functional correctness using function approximation. In *Proceedings ISSTA '04*, Boston, 2004.

[11] H. Mills, V. Basili, J. Gannon, and D. Hamlet. *Principles of Computer Programming: A Mathematical Approach*. Allyn and Bacon, 1987.

[12] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, pages 14–32, 1993.

[13] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. of the ACM*, December 1972.

[14] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.

[15] Clemens Szyperski. *Component Software*. Addison-Wesley, 2nd edition, 2002.

[16] Walter G. Vincenti. *What Engineers Know and How They Know It*. Johns Hopkings University Press, 1993.

[17] Kurt Wallnau. `www.sei.cmu.edu/pacc` (links to CBSE proceedings).