Test-based Specifications of Components and Systems

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

Abstract

Several program-analysis schemes now make unusual use of information derived from execution sampling. From finite test sets these techniques derive measures with wider meaning, which can then be exploited in novel ways. We call test information used beyond its actual limitations a test-based specification. *The idea is quite different from the usual software specification, which is an a priori statement of what a program should do. Rather, a test-based specification is an empirical approximation to what a program actually does do. The great virtue in analysis using test-based specifications is that (in contrast to the usual software specifications) test-based analysis is decidable and automatic.*

A test-based specification system for component-based software development (CBSD) has been implemented over the past five years, along with an extension of fundamental testing theory to precisely describe its properties. The CBSD tools provide an ideal context for experiments to study test-based specification, providing insights into subdomain testing, composition of test results, and especially the role that program persistent state plays in testing and analysis. This paper describes the CBSD theory and tools, lists insights gained, and suggests new ways to think about and practice testing using test-based specifications.

Keywords: Component/system testing theory, test-based specification, persistent state

1. Introduction

An engineering component, from a plumbing elbow to a CPU chip, is expected to come with a 'specification.' This specification, from a handbook or a data sheet, is a document to be used in designing systems that employ the component. Unlike a computer-program specification, a data sheet is not a requirements wish-list; it is a factual statement of how the component has been observed to behave, a promise of what can be expected from it. The promise has a statistical character because of chance variations in manufacture, but these too can be quantified on the datasheet. Computer-program specifications are quite different: (1) They are in principle exact descriptions, without any statistical variation; and (2) They might not describe actual behavior. (1) is a positive quality; (2) is profoundly negative. The connection between program specification and program behavior must be established by empirical testing[1]. It is striking that the programming paradigm is so different from the engineering one. Most engineers measure component properties and use the measurements in design. In software, we try a few cases against an abstract specification and perhaps correct some problems, but then make no further use of the unit test results. For programs, we use detailed specifications but know that they do not hold; in engineering, measurements known to hold are used even though they lack detail[2].

'Specifications' are the main subject of this paper, so it is important to distinguish the computer-program ideal sense of this term from the engineering data-sheet sense. We will call the latter *test-based specifications*, and will be very careful not to omit the adjective unless we mean an a priori description of what a program should so. When speaking informally, it is usual to refer to program 'behavior.' *Specified behavior* is what a program should do; *behavior* is the complete version of what it does do, and *tested behavior* is the subset of behavior that has been observed in a test. A test-based specification and tested behavior mean the same thing for the same test; sometimes we will say *approximate behavior* or *test-based approximation* to refer to a test-based specification.

The software entities considered here are components and system assemblies made from them. Test-based specifications are obtained for components, and we investigate what can be done with these at the system level. Section 2 presents the background of component-based software development (CBSD) with a testing theory and tools to enable experiments. Testing insights gained from these experiments are listed in Section 3. Finally, Section 4 explores the difference between the engineering view of test-based software specification and the conventional view of software specification and testing.

[1]Formal mathematical proof is in principle a way to establish that a perfect description holds. It is still controversial whether such proofs will ever be put to daily engineering use. This is a paper about testing, not proving.

[2]Edward Tufte [20], quotes J. W. Tukey as suggesting that "approximately right" is much better than "exactly wrong."

2. Components and Systems in CBSD

Following Szyperski [19], a 'software component,' is an executable program described only by its interface and black-box behavior using only local persistent state. A component is a program, but viewed only through its behavior. A software system is an assembly of components, also a program, so any system is itself a 'component'.

2.1. Subdomain Testing of Components

Subdomain testing is the natural way to examine program behavior. The input domain is divided into subdomains on each of which the behavior is intuitively 'the same.' A few test cases are tried in each subdomain. In practice, a meaningful sense of 'the same' is hard to define and harder to verify. Behavior on a subdomain can only be known by extensive testing there, the very thing subdomains were invented to avoid. Ultimately, it is failure behavior that must be 'the same' for subdomain testing to work. But for all algorithmic partitions into subdomains, subdomains may contain some success points and some failure points [12]. If failure points happen not to be tested a subdomain test will succeed and lull the tester into a false belief that all is well.

2.2. Extended Testing-theory Model

The model used by Goodenough and Gerhart [4], Howden [12], and almost all subsequent testing theoreticians, assigns functional semantics to programs. The theory is reviewed here to establish a consistent notation, then extended to include state.

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 Mills et al. [17] suggested a graphic notation: The meaning of P is a function $\boxed{P} : D \rightarrow R$. 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 an input-output function[3] $F : D \rightarrow R$, and *correctness* of P wrt F means that $\boxed{P} = F$. A test set 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 test set T *covers* the subdomains if $\forall i, T \cap S_i \neq \varnothing$. The success of a test set is *misleading* if the program is not correct in consequence.

Other program properties are easy to capture in the functional theory, by imagining that a program P computes other functions as well as $\boxed{P}$. For example, P's run time is a function $T : D \rightarrow \mathbb{R}$. If desired, correctness can be defined to include non-functional properties, for example, that a program achieve a response-time bound Q: $\forall t \in D, T(t) \leq Q$.

A *test-based specification* for a program P is a finite function obtained from subdomain testing. Each subdomain is sampled and values of $\boxed{P}$ are averaged over the subdomain, defining a step function approximation to the complete behavior.

This functional-semantics testing theory models only programs that do not retain state from test to test. But many testing problems are intrinsically state-related, so we extend the theory[4] to explicitly include local state.

Along with the program input domain D and output range R, consider a new, distinct state set H. The behavior of program P is defined 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 input-state pairs $(d, h) \in D \times H$.

A *specification* is a (partial) function

$$F : D \times H \rightarrow H \times R.$$

Let P be in a special *initial state* $h_0 \in H$. 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 F 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 *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,

[3] 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 the discussion of 'don't care' inputs. However, the mathematical machinery of relations is less intuitive than functional notation, so in this paper we use functional specifications.

[4] A more detailed description of the extended theory was presented at ISSTA 2006 [7]. It can be thought of as a compromise between trace semantics [14] and explicit formal states (as in Z [18], for example). However, its primary motivation is that the theory be a natural extension of the pure-function testing theory begun by Goodenough and Gerhart, and Howden.

$$\begin{aligned}(h_{i+1}, r_{i+1}) &= ((\textcircled{P}(x_i, h_i), \boxed{P}(x_i, h_i)) \\ &= F(x_i, h_i') = (h_{i+1}', r_{i+1}'), \\ & \quad 0 \le i \le n-1.\end{aligned}$$

The definition requires P to terminate exactly where F is defined, so that the domains of F and $\boxed{P}$ match.

The orthogonal state space H can be divided into subdomains for testing as is the input space; non-functional properties can similarly be formalized as additional functions, but mapping $D \times H$ instead of D alone. The generalization of test-based specification is also straightforward: the two-dimensional grid of subdomains on $D \times H$ is sampled and the resulting finite function is a step-plateau. Care is required in the sampling method, however. To realize the definitions of $\boxed{P}$ and $\textcircled{P}$ requires test *sequences* applied to program P. There seems no sensible way to choose these sequences other than randomly [8].

In summary, the extension using $\boxed{P}$ and $\textcircled{P}$ to capture state behavior extends basic testing theory as Goodenough and Gerhart might have done.

2.3. Tools for CBSD Experimentation

A research-prototype implementation [5] of tools supporting CBSD has been developed over the past five years. In designing these tools, a conscious choice was made to restrict the form of components and systems so that the theory describing them is stripped down to essentials. This allows the tools to efficiently accomplish what would be impossible in a more general setting. To this end, components are permitted only a single floating-point input and output value, and may keep only a single float value for state. In theoretical terms, for any component program C, $\boxed{C} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and similarly for $\textcircled{C}$. (The rationale is that multiple values add more to the mathematical overhead than they return in insight. Numerical values allow straightforward random sampling.)

To analyze a component, a person provides its executable code and a collection of subdomains on which to test. A tool samples sequences of inputs to obtain the plateaus of a test-based specification. Figures 1 and 2 show tool output for an artificial component C designed to expose insights about test-based specification[5]. Fig. 1 summarizes a test measurement of $\boxed{C}$, and can be thought of as part of conventional testing of component C. It might very well be used by C's developer to study C and to compare its behavior with a formal specification.

However, the purpose of our tools is not conventional code analysis or verification, but rather the measurement of

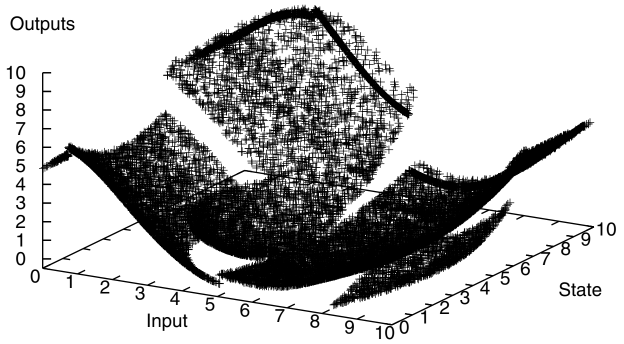

Figure 1. Functional behavior of C. Each data point is an execution from a random sequence of inputs. There were 188 sequences of length between 1 and 188 and a total of 17798 test points.

test-based specifications like Fig. 2 for C. The errors reported have nothing to do with specified behavior desired

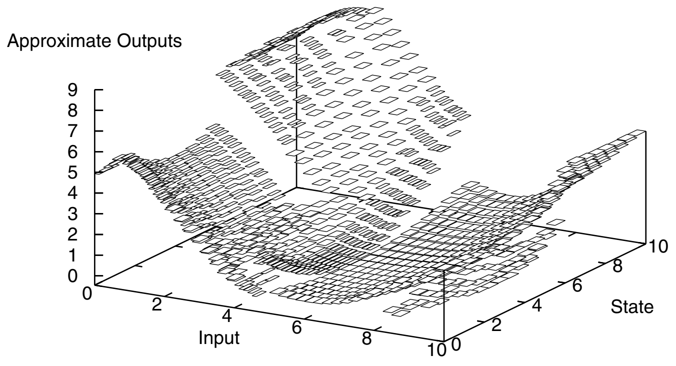

Figure 2. Output test-based specification of component C. Each rectangular plateau approximates the behavior in an input$\times$state subdomain. The tools report r-m-s error in each subdomain; here the weighted average for all subdomains is 11%.

for C; they measure the difference between actual behavior (Fig. 1) and test-based approximation (Fig. 2). In verification, a component developer has to convince herself that C is behaving according to specification; except insofar as Fig. 1 helps to visualize the output behavior, our tools are not involved. Rather, once she is satisfied with the behavior, the tools will measure Fig. 2 and report how close this approximation comes to the actual behavior.

Similarly, Fig. 3 shows a test-based specification of $\textcircled{C}$.

Once test-based specifications are obtained for a group of components, they are used to predict the behavior of system designs. In the simplest theory only the 'structured' constructs of series, conditionals, and loops are allowed. (These are a sufficient set of connectors [1], although far

[5]The tools also treat the non-functional property of run time, not used in this paper.

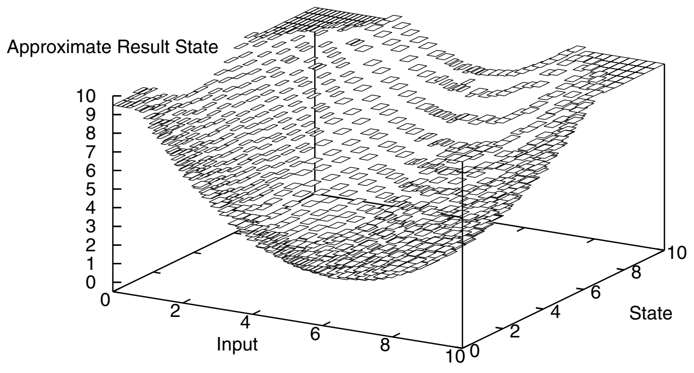

Figure 3. State test-based specification of component C. Against the actual execution (not shown) the r-m-s error has a weighted average of 6.2%.

from the many possibilities offered by practical component frameworks.) To study a system, a person defines its structure and the components to be used. Details of the synthesis prediction algorithms are presented in [9]. They work by looking up approximate output values by subdomain in measured tables for each component, and doing the bookkeeping to see approximately what the system will do on those subdomains. The heart of the implementation of the prediction algorithms is a CAD tool that synthesizes a test-based specification for the system. No actual system assembly or execution is needed, nor is any component information used beyond test-based specifications. The calculations are much faster than system testing.

As a simple example, consider a system using C with three (stateless) components C_c, C_s, and C_e in the structure:

$$\texttt{if } C_c \texttt{ then } C_s; C \texttt{ else } C_e \texttt{ fi}. \tag{1}$$

C_c is *false* on [0,1) and on [9,10); C_s has a saw-tooth output; C_e has output that first rises linearly, then falls linearly. Fig. 4 shows $\boxed{C_s}$ and $\boxed{C_e}$ and their measured test-based specifications using 20 subdomains. The component behaviors are chosen to be comprehensible, yet complex enough to challenge the tools. Figure 5 shows results of the CAD synthesis for the output of system (1). The 400 plateaus in the figure are calculated from component test-based specifications in Figs. 2 and 4.

To validate the CAD tools, the actual system code is assembled from the components' code and executed with the result shown in Fig. 6. Fig. 5 has the same general appearance as Fig. 2, but a quite different meaning. It is not a measurement, but a theoretical prediction of approximate system behavior, calculated from test-based specifications of the components. The errors reported are deviations of the prediction from actual system behavior. Even with the simple behaviors of C, C_c, C_s, and C_e, the system behavior is surprisingly complex. In Fig. 5 the only easily seen feature is the state-independent output on [0,1) and [9,10) which results from the conditional selecting C_e.

Similarly, the CAD tool calculates system result-state

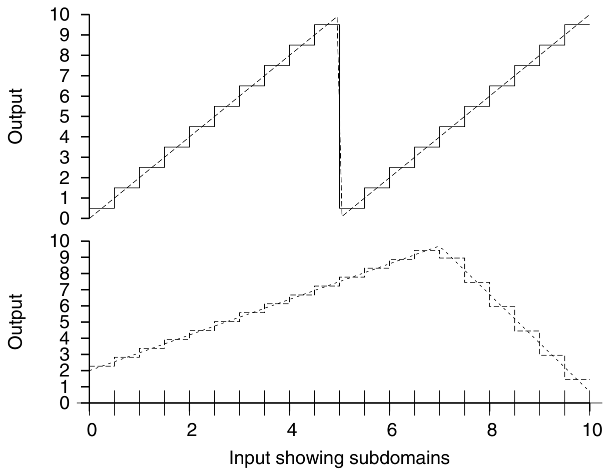

Figure 4. Behaviors of C_s (upper, r-m-s error 7.0%) and C_e (lower, 5.6% error).

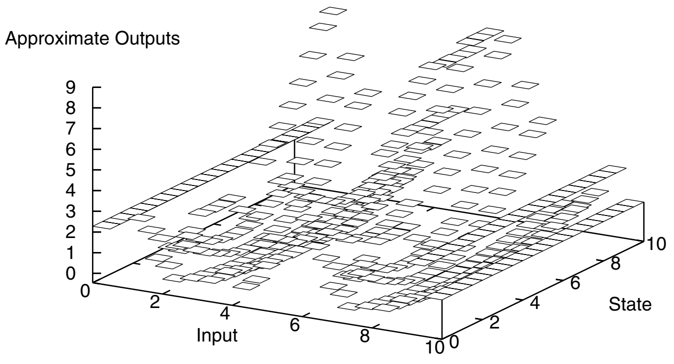

Figure 5. Predicted functional behavior of system (1) made from four components.

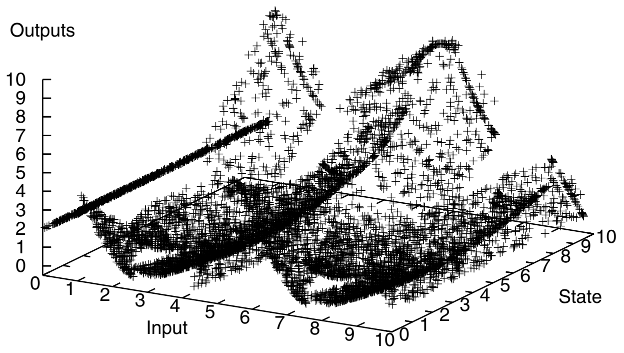

Figure 6. Actual measured output behavior of system (1), from sampling assembled code for it with 120 sequences containing 7217 test points. The weighted average r-m-s error in the calculation is 13%.

behavior, shown in Fig. 7 with the actual execution values superimposed.

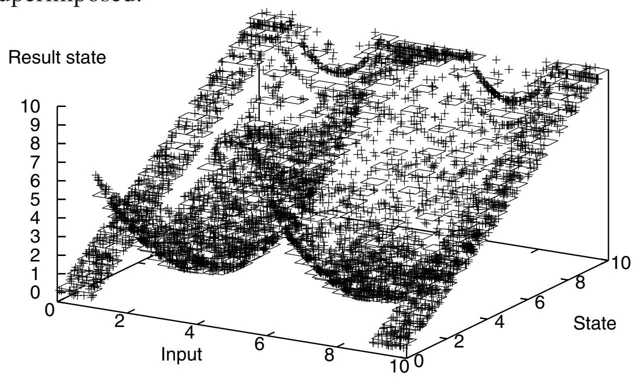

Figure 7. Calculated and measured state behavior of this system. The weighted average r-m-s error in the calculation is 7.8%.

The properties of calculated predictions and their use in system testing will be further considered in Section 4.

3. Testing Insights Gained

Experimenting with CBSD tools to measure test-based specifications and calculating system properties from them has led to new understanding of software testing. The use of simple artificial components is helpful in exposing underlying principles, because the absence of arcane details present in real testing makes it possible to formulate generalizations.

3.1. Complicated System Behavior

Fig. 7 displays the well known fact that behavior of systems built from very simple components can be quite complicated. The complication quickly escapes any attempt at visualization, particularly when system state is a cross product that cannot be graphed in 2-D projection. Black-box system testing is thereby called into question, because very dense sampling will be needed to investigate features of behavior like Fig. 6.

3.2. Subdomain Testing

Although subdomain testing is the primary practical method, it suffers from a 'stopping-rule problem.' The tester never knows how good the subdomains are, nor when to stop refining them and end testing. Our CBSD tools provide useful feedback in the error reports for a component. When an individual subdomain has a large r-m-s error, the subdomain is ill-chosen and needs to be split or its boundaries shifted to better capture the behavior of the component being measured. Graphs like Figs. 1 and 2, and Fig 4, pinpoint subdomain deficiencies. When the r-m-s errors are small, it means that the test-based specification is accurate.

Unfortunately, the real indication of how good subdomains are doesn't come until a system is assembled. It can happen that some component subdomain should have been further refined because in the system it gets heavy usage. The subdomain division stands in for an operational profile that weights system subdomains, and since at component-test time the system profile is unknown, the best a component tester can do is to minimize the r-m-s error of the test-based specification. Then no matter how a later-applied profile emphasizes some part of the input space, it will not have been neglected in component testing.

Our CBSD tools support an ideal engineering procedure for component testing. Given a testing time budget, the tester begins with an arbitrary set of subdomains, and using feedback from the CBSD tools, refines and adjusts them to get the best test-based specification possible within budget. In this process is easy to forget that as subdomains are refined and the test-based specification gets closer to a component's actual behavior, this tested behavior should be continually checked against the formal specification to see that the component is correct. The tester has two tasks to perform together and the CBSD tools provide feedback on only one of them. If there were an effective oracle, its feedback would elevate correctness checking to the same kind of engineering activity. The test-based specification would interact with an oracle by quantifying the extent of testing: It means little that an oracle has been satisfied unless the behavior has been captured accurately.

Comparing Fig. 1 with Fig. 2 and Fig. 6 with Fig. 5 shows how helpful the subdomain grouping can be in visualizing behavior. The data on what actually happens is harder to visualize than its approximation.

Our experiments [10, 6, 7] have shown that r-m-s errors in system prediction are roughly linear in the measurement errors in component test-based specifications and in the number of components that form the system. Discontinuities in components' behavior must be carefully investigated, but as test-based specifications become more accurate so do system predictions. However, the simplest experiments in composing components (such as for the system in Fig. 6) show that subdomain testing of units as practiced today is woefully inadequate. It is common practice to exercise a stateless unit with a dozen haphazardly chosen subdomains, when for system predictions accurate to (say) 5%, hundreds must be selected and refined with care. Adequately capturing state-dependent behavior requires tens of thousands of good subdomains.

3.3. Persistent Local State

Practical testers know that inadequate probing of system state is often to blame for failures missed in pre-release testing. When a system fails unexpectedly, the reason is often a latent state error. So it is not surprising that subdomain testing of components with state is difficult. Indeed, this is obvious from the dimensional change in theory (Section 2.2): The component test space is two-dimensional when there is state. If 100 subdomains were needed to obtain an accurate test-based specification of a stateless component, 10000 will be needed to handle 100 $\times$ 100 subdomains in two dimensions. If stateless-component testing uses far too few subdomains in practice, things are much worse when a component has state. In practice, state subdomains may not even be identified and systematic coverage of all input$\times$state combinations is seldom attempted. Our experiments have provided a number of insights about what makes state difficult to test. But at the outset the obvious lesson in design for testability is: confine state to as few components as possible.

Fig. 2 vs. Fig. 3 makes a less obvious point: Explicit study of output state $\textcircled{C}$ is less helpful in understanding behavior than working with $\boxed{C}$. For example, in Fig. 2 there are two discontinuities in the output surface — one along input 5 and the other along state 5. This behavior was inserted to study the effect of discontinuities, using straightforward conditional statements in C's code. A person doing debugging might insert code in this way to trace a problem or to correct one. However, to introduce such changes in Fig. 3 is not so easy. Adding straightforward conditional statements creates bizarre and unexpected changes in $\textcircled{C}$. About all that can be easily accomplished is to test for a state value and adjust the result state (e.g., 'clip' it to 9.5 as was done at the rear corners of Fig. 3).

People are used to state remaining hidden and they better understand its role as the second part of the domain of $\boxed{C} : D \times H \rightarrow R$. The reason state is hard to understand lies in the crucial difference between state and input as parameters determining program behavior. Input is an independent variable a tester controls, and output is its dependent variable. But state is *not* an independent variable. It can't be sampled directly because its values are not arbitrary— they are determined by the program and are self-dependent. State behavior is intuitively less 'functional.' As a simple example, a component P with an identity $\boxed{P}$ is created by assigning the input to the output. But a similar assignment of input state to output state results in a constant $\textcircled{P}$: each state is the same as the previous one and hence no possibility but the initial state.

When components are combined, their states combine as a cross product. The system state for a series of C_1 (states H_1) and C_2 (states H_2) is pairs from $H_1 \times H_2$. If in a conditional `if` C_c `then` C_t `else` C_f `fi` all three components have state, the system state is $H_c \times H_t \times H_f$. When further system combinations occur the dimension continues to increase, so for example, two conditionals in series might have a six-fold state. Since the number of test subdomains rises as the product, even three components each needing 100 state subdomains will produce a million system state subdomains. Again, the obvious advice for testability is to reduce the number of system components with state.

Current practice in testing systems with state uses 'state-coverage' algorithms that are often erroneous [7]. First, the states of a specification are used instead of states that actually arise in the implementation. Drastic errors in behavior can remain hidden from imaginary states that should have been implemented but weren't, and code-coverage metrics don't provide any check on failure to cover real states. Second, states are sampled explicitly, not implicitly using input sequences as in Section 2.2. It is incorrect to randomly select state values, since they are under complete program-, not tester-, control. Worse, a state selected for test may never actually be entered by the program. Externally setting such an infeasible state creates a phony execution that is an artifact of the test. Successful testing on infeasible states gives a false confidence in a program's reliability. On the other hand, when a test fails on an infeasible state, time is wasted on a spurious problem. It is hard to escape the conclusion that state-coverage testing as currently practiced is a procedure performed without basis. Engineers need well defined procedures, but when there is no necessary connection between actions taken (testing) and goals (to understand behavior, to find failures or increase confidence in their absence) a prescribed procedure is no more than make-work.

4. Using Test-based Specifications

Test-based specifications are a necessary part of CBSD, because they make honest engineering artifacts of software components. Section 2.3 has described how system predictions can be made from test-based specifications and implemented by CAD tools.

4.1. Performance of CAD Tools

In the CBSD paradigm where component test-based specifications are measured then combined algorithmically, the huge collection of cross-product system states is not sampled (except for validation such as in Fig. 6). The prediction algorithms trade storage for execution time. In the simplest case, let there be two components, each with N input subdomains and S state subdomains. Then the actual system storage is $S + S$, while the tables that hold the test-based specification take space NS^2. To sample the $(NS)^2$ system subdomains for average execution time R takes $R(NS)^2$. The prediction algorithm requires table-lookup and a number of copying operations. If the copying time and table-lookup overhead time is m per operation, the prediction time[6] is $m(N \log N)NS^2$. Cancelling common

[6]The reader must take these estimates on faith because of the stringent page limitations for this paper. The stateless case is presented in detail in

factors, the prediction retains a factor $m \log N$, while the system execution retains factor R. Roughly, the prediction factor $m \log N$ is on the order of 1 μs for nanosecond instructions and N under 1000, while the system execution factor is arbitrary.

The loop system construction is a special case in which the prediction can do even better. Actual loops can require an arbitrary execution time to test, and a non-terminating loop has to waste a lot of time before a tester decides to abort it. The prediction implemented in our CBSD tools requires a time independent of execution (depending only on the number of subdomains, roughly as above), and has the wonderful property that it decidably predicts non-termination [9].

We have been comparing the time to make a complete system prediction to that required to completely subdomain-test the system. For a single point, the execution-time comparison is also in the calculation's favor. To execute the actual system requires a sum of component execution times, which may include an arbitrary repetition factor for a loop. To 'execute' the prediction requires only looking up an input subdomain (from N possibilities) and k state subdomains (from S possibilities each). The total look-up time is roughly $m(N \log N + k(S \log S))$. Taking $k = 3$ in line with the advice to restrict components with state, and $N = S = 128$, the prediction time is about 4 ms, vs. an arbitrary execution time.

4.2. Conventional Specifications

There is a two-fold role for the usual kind of specification (that is, a description of what some program is supposed to do) in the CBSD process. First, when a component developer tests code, the results are checked against the specification before the component is released and before its test-based approximation is recorded. Second, system results (as predicted by CAD tools) are checked against a system specification. The two checks are complementary, and the quality of the first affects the latter. Should there be a system failure in the prediction but component tests did not fail, the system structure is likely to be at fault, and tracing a failed system test at component granularity should be profitable. Because the prediction is an approximation, it may be a good idea to repeat failed tests using the actual system code [7] to check that they are really failures.

But more important is the significance of a system prediction that agrees with the system specification. System structures are far less complex than is the aggregate of system code. There are therefore fewer ways for the structure itself to produce misleading coincidental success. The system is more likely to be that testers' dream: if it fails, it fails almost everywhere. This is the sense in which system tests are spot checks rather than part of an elaborate test plan.

reference [9].

In contrast, the conventional model of system development is to expend some resources on unit testing, then enter full-blown system test against the system specification. Leaving aside the fact that executing a system is slower than CAD calculation, the usual system test differs in principle from a check of CAD prediction. If the system should fail and be modified, the system test is usually restarted from scratch[7]. Since the quality of unit testing is not usually quantified in any way, there can be no separation of possible failures between the component and system levels. Hence when a system test fails, the problem may be anywhere in the code and the component structure is no help. When a system test is successful, it is no more than an isolated point in a huge sample space.

4.3. Composing Test-based Specifications

Although not yet adapted to CBSD, several schemes have been proposed for obtaining something akin to our test-based specifications. The best known is Daikon [3], which uses test samples to find pre- and post-conditions the program satisfies for those tests. These constitute a test-based specification in Floyd-Hoare logical form. Similarly, Henkel [11] finds test-based specifications in the form of algebraic axioms. Meinke [16] gives a procedure for locating a test that fails for a given program; so long as his procedure has not found this test point, it induces a series of increasingly accurate test-based specifications, and can be used to define a natural set of 'functional subdomains' based on a given formal specification. Finally, several papers [13, 2, 15] describe variations on what are being called bounded exhaustive test (BET) methods. BET methods look for program failures, but along the way they generate test sets that could be used with any scheme to get a test-based specification.

One of the best ways to understand a complex transformation M is to formalize it and study its 'decomposition theory.' The essential idea is to investigate homomorphic properties of M. M applies to an entity X that can be broken into subentities $x_1, x_2, ...$, for which there is a natural combination operation $\oplus$:

$$X = x_1 \oplus x_2 \oplus ...$$

The study of M seeks another operator $\otimes$ that combines M-values in a homomorphic way:

$$M(X) = M(x_1 \oplus x_2 \oplus ...) = M(x_1) \otimes M(x_2) \otimes ...$$

In our case, the entities are programs, M is the operation of taking a test-based specification. Combination $\oplus$ is component assembly, and $\otimes$ is the prediction algorithm for the system of assembled components. To check the homomorphism, suppose a system were directly subdomain tested to

[7]Regression testing might be profitably studied using a component model, but at present its theory is too cumbersome for practical use.

create a test-based specification G, as if the system itself were a component (this is $G = M(X)$). What is the relationship between G and the predicted system behavior Q calculated by our CAD tools from the system's component test-based specifications (that is, $M(x_1) \otimes M(x_2) \otimes ... = Q$)? So long as the deviation of Q from actual system behavior is small, the deviation of G should also be small (in the theory they differ only as the mean of squares differs from the square of the mean), and Q and G should be even more similar. Fig. 8 displays G and Q for the example system of Section 2.3. The average difference over the 400

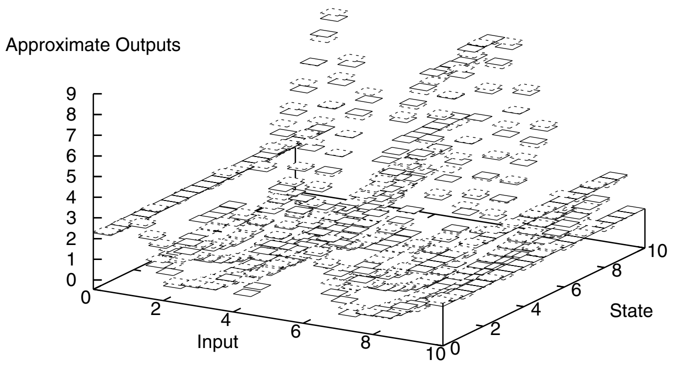

Figure 8. Comparison between measured system test-based specification (G, dashed) and CAD prediction from component test-based specifications (Q, solid).

subdomains is 7.5%. The prediction error in Q is 13% (Fig. 6), and the measurement error in G is 24%. Thus the homomorphic property holds approximately even for relatively poor approximations.

The same question can be formulated for other test-based specification schemes, using the same operator $\oplus$ to build systems from components. For example, how would composing component Daikon test-based specifications according to Floyd-Hoare propositional-logic rules (a good candidate for $\otimes$) compare with a system Daikon test-based specification? How is the BET set for a system made from components related to the BET sets for those components? (That is, what is $\otimes$ for the various BET schemes?) Studying component decomposition of these theories would be a way to understand them better.

References

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

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings ISSTA '02*, pages 123–133, Rome, 2002.

[3] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Soft. Eng.*, pages 99–123, Feb. 2001.

[4] J. B. Goodenough and S. 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. `www.cs.pdx.edu/~hamlet /components.html`.

[6] D. Hamlet. Tools and experiments for a testing-based investigation of component composition. Submitted to ACM TOSEM, October, 2006. Copy at: `http://www.cs.pdx.edu/~hamlet/TOSEM.pdf`.

[7] D. Hamlet. Subdomain testing of units and systems with state. In *Proceedings ISSTA 2006*, pages 85–96, Portland, ME, July 2006.

[8] D. Hamlet. When only random testing will do. In *Proceedings First International Workshop on Random Testing*, Portland, ME, July 2006.

[9] D. Hamlet. Software component composition: subdomain-based testing-theory foundation. *J. Software Testing, Verification and Reliability*, June 2007. (In press.).

[10] D. Hamlet, M. Andric, and Z. Tu. Experiments with composing component properties. In Wallnau [21].

[11] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *Proceedings ECOOP '03*, Darmstad, 2003.

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

[13] D. Jackson. Alloy: a lightweight object modeling notation. *ACM Transactions on Soft. Eng. Methodology*, pages 256–290, Apr. 2002.

[14] R. Janicki and E. Sekerinski. Foundations of the trace assertion method of module interface specification. *IEEE Trans. on Soft. Eng.*, 27:577–598, 2001.

[15] D. Marinov and S. Khurshid. Testera: a novel framework for automated testing of java programs. In *Proceedings 16th IEEE Int. Conf. on Automated Software Engineering*, pages 22–34, San Diego, 2001.

[16] K. Meinke. Automated black-box testing of functional correctness using function approximation. In *Proceedings ISSTA '04*, pages 143–153, Boston, 2004.

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

[18] J. M. Spivy. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

[19] C. Szyperski. *Component Software*. Addison-Wesley, 2nd edition, 2002.

[20] E. Tufte. *Beautiful Evidence*. Graphics Press, 2006.

[21] K. Wallnau. `http://www.sei.cmu.edu/pacc` (links to CBSE proceedings).