Theory of Software Reliability Based on Components

Dick Hamlet[1]
Portland State University
Portland, OR, USA
hamlet@cs.pdx.edu

Dave Mason[2] Denise Woit
Ryerson Polytechnic University
Toronto, Ontario, CANADA
{dmason,dwoit}@scs.ryerson.ca

Abstract

We present a foundational theory of software system reliability based on components. The theory describes how component developers can design and test their components to produce measurements that are later used by system designers to calculate composite system reliability — without implementation and test of the system being designed. The theory describes how to make component measurements that are independent of operational profiles, and how to incorporate the overall system-level operational profile into the system reliability calculations. In principle, the theory resolves the central problem of assessing a component, which is: a component developer cannot know how the component will be used and so cannot certify it for an arbitrary use; but if the component buyer must certify each component before using it, component-based development loses much of its appeal. This dilemma is resolved if the component developer does the certification and provides the results in such a way that the component buyer can factor in the usage information later, without repeating the certification. Our theory addresses the basic technical problems inherent in certifying components to be released for later use in an arbitrary system.

Most component research has been directed at functional specification of software components; our theory addresses the other, equally important, side of the coin: component quality.

Keywords: Software components, reliability composition, foundational theory, COTS, CBSE

1 Promise and Pitfalls of Components

Software components are the most promising idea extant for the efficient design of quality software systems. Most of the research in components is devoted to specification, design, reuse, and cataloging of the components themselves. The complementary issue — component quality — is also important, but has received less attention. There are no accepted standards for the quality of software components, largely because there is no theoretical foundation on which to base standards. Developers of safety-critical software, and the regulatory agencies responsible for the systems they design, currently use subjective assessments of software quality. It would be of great value to replace these with hard data.

In many engineering disciplines, the idea of aggregating standardized components to create a complex system has allowed the creation of better systems more easily. Components are described in a handbook, where each has a 'data sheet' entry. Its data sheet describes what a component does, and equally important, it gives constraints that allow the system designer to decide if the component is 'good enough' for the application. For example in analog electronic systems, these constraints concern the allowable voltages for the component.

The success of the component-construction paradigm in mechanical and electrical engineering has led to calls for its adoption in software design. (The current buzzword is 'COTS' for Commercial Off-The Shelf components.) But the system designer who today looks for technical data sheets on software components will be disappointed. Without the solid information of a data sheet, a software component may be no bargain. To buy off-the-shelf software of unknown quality is only to trade the difficult task of assessing your own work, for the more difficult task of assessing someone else's.

Software reliability theory [16] is a candidate for describing 'quality' on a component's data sheet, but it cannot be applied without addressing a central problem. Reliability of electrical components depends on their physical environment (e.g., temperature, voltage) and can be given without regard for the expected usage. However, software reliability depends critically on usage, in the form of an operational profile for inputs. A component developer must use some profile to test and certify a component, yet when that component is embedded in a system, the profile it sees will be different, invalidating the component test.

2 Foundational Theory

Our theory uses statistical reliability as the quality information to appear on a software-component data sheet. The other side of the components' coin — the technical description of what a component is supposed to do — is of equal

[1]Work supported by an EPSRC grant through City University, London.
[2]Work supported by NSERC grant.

or greater importance, but it is not part of the theory. However, the component developer must have an effective oracle to carry out statistical testing, and it is here assumed that a formal specification supplies this oracle [21].

The problem of software system reliability from component data has been studied for more than 20 years [15, 13, 14, 11, 12]. Perhaps because a straightforward analysis seems overwhelming, past approaches all model components and systems in some abstract, 'high-level' way, for example as a Markov chain. In contrast, the approach taken here is fundamental and direct: the failure behaviors of components and system are modeled in detail.

2.1 Basis of the theory

To be useful in system design, component data sheets must contain technical information sufficient for the system designer to make reliability calculations; and, it must be possible for the component developer to collect this information. Subjective and methodological assessments of the software development process are not data-sheet information, because they amount to little more than assertions that the developer is trying to do a good job. A current prestandards study group on component quality is using such subjective measures almost exclusively [1]. The reputation and process of the developer are not inconsequential factors in selecting a component; but, their role should be to back up hard technical data.

Two measures of software quality that qualify for a data sheet are (1) that the component has been proved correct, or (2) that it has been operationally (random) tested. Both of these require a formal specification of what the component is supposed to do (the part of the data sheet not considered here). A statement that the component has been proved correct is a guarantee that it will perform according to its specification. A statement that (say) its reliability is better than $1 - 10^{-4}$ per execution with an upper confidence bound of 99%, is a statistical assertion that there is no more than 1 chance in 100 that it will not perform according to the specification in 10,000 trials. Correctness proofs (1) can be thought of as the special case of a profile-independent reliability of 1.0, with 100% confidence. We will be largely concerned with the more practical case (2) of testing.

The reader of a data sheet might doubt that the developer has actually established its precise technical claims. But this is a question that can be answered scientifically, and if the developer has lied and there is a damaging failure as a result, there are legal remedies.

This theory is based on two ideas:

Profile mappings. Operational profiles must be taken into account when measuring component parameters for a data sheet. Since the component developer cannot know how the component will be used, and hence what profile it will face as part of a system, the data-sheet must take profile as a parameter. That is, the data sheet specifies *mappings* from a profile to reliability parameters [18].

Component subdomains. A component has a natural partition of its input space into functional subdomains, and the practical description of its operational profile is as a vector of weights over these subdomains. This form for the operational profile allows the component developer to test a component without knowing the profile that it may later face [6].

2.2 Component and System Development Process

In outline, the process of making and using components is idealized as follows:

making components

- The component developer defines a set of natural subdomains for an implemented component.
- The component developer measures properties of the component for each subdomain.
- The component developer publishes a data sheet listing its subdomains, and giving reliability information as mappings of profiles.

using components

- The system designer decides on a system structure utilizing components.
- Using the data sheets for prospective components and a trial profile of the system, the system designer calculates the system reliability.
- If the desired system reliability cannot be achieved, it will be necessary to find better components, or to change the system structure, and repeat the reliability computation.

This idealized model is a compromise that acknowledges some of the realities of practical application, but simplifies the real-world situation so that it can be analyzed and the theory tested. No fundamental theory for computing system reliability from components currently exists, probably because the details of practical application are so complex and varied. A successful theoretical foundation must use a relatively simple model, but it must also draw on existing testing and reliability theory.

3 Problems and Proposed Solutions

Any theory of software component reliability must address two important issues: the operational profile used to test the component vs. the profile it encounters within the

system where it is used; and, the design process for the system's main program that invokes its components.

3.1 Operational Profiles

When the quality information on a component's data sheet is statistical, it must be obtained by random testing. The fundamental problem of assessing component quality statistically is that any standard profile from which test inputs are drawn will *not* match the profile that the component will experience when placed in a system. Furthermore, the profile seen by a component depends not only on the system input profile, but also on the component's position in the system and on the actions of the other components there.

Section 4 proposes a solution to the profile problem as two data-sheet profile mappings (one for reliability and the other for profile transformation) defined using a subdomain decomposition of the component input domain. A system designer can use these maps to calculate the system reliability before implementation.

3.2 System Design

To make calculations of system properties from component properties, the system designer must not only have the component data sheets, but needs to try various system designs that satisfy the system requirements. In trying designs, there are two technical problems to which Section 5 provides solutions:

'Glue logic.' Any system will contain explicit code that invokes its components, perhaps first adjusting parameter values and later combining their results as needed, and perhaps performing significant computations unrelated to any component. These pieces of 'glue logic' have to be analyzed along with the components. Glue logic is handled by converting system code to fragments that act like independent components [23].

System control flow. Component invocation patterns come from the top-level control structures in the system design. These patterns are handled by providing a reliability algebra for the constructions of sequence, conditional, and looping.

4 Operational Profiles

Component data sheets cannot directly include 'reliability' of the component, because reliability is an input-profile-dependent quantity, and the component developer cannot know what profile will occur in a system being designed. Furthermore, each system component distorts the profile presented to it as input, and hence influences the profile seen by other components. We propose that the component developer supply information in the form of profile mappings, but this requires that the form of profiles be standardized.

4.1 Profiles Defined on Subdomains

A profile P is a probability density $P : D \rightarrow [0, 1]$, where D is a discrete input domain. In practice, it is very difficult to obtain precise profile data from software users. The best that can be done is to describe a profile as a histogram of probabilities over quite broad classes of inputs [19]. We exploit this idea as a standard form for profiles.

Let a component C have input domain D, partitioned into n disjoint subdomains,

$$D = S_1 \cup S_2 \cup ... \cup S_n .$$

Let each subdomain have its own profile P_i, and its own probability of failure f_i:

$$f_i = \sum_{x \in (D_f \cap S_i)} P_i(x) ,$$

where D_f is the subset of D on which C fails and P_i is the profile within subdomain S_i. The overall profile P for the component may be expressed as a normalized vector of probabilities that each subdomain will occur in use:

$$P =< h_1, h_2, ..., h_n > .$$

In the practical case, the profile is literally user-defined: a person estimates the likelihood that various inputs will arise. Such a person is hard-pressed enough to make estimates of subdomain weightings h_i, and can usually say nothing about distributions P_i. Thus within the subdomains it is usual to take a uniform distribution $P_i = 1/|S_i|$. Hence:

$$f_i = \frac{|D_f \cap S_i|}{|S_i|}.$$

In the sequel, a profile will always be a normalized vector of weightings applied to a set of subdomains.

4.2 Estimating the Failure Probabilities

In practice, the failure probability f_i in subdomain S_i can be estimated by random testing with a uniform profile. The usual case is that no failures are observed, and this supplies an upper bound on f_i. If N tests are conducted without failure, there is good confidence that f_i is roughly below $1/N$ [5]. Thus this basic parameter can be measured by a component developer.

4.3 Data-sheet Mappings

The two mappings on a component data sheet give the system designer the ability to calculate the reliability of the component wherever it is placed in a system, and to calculate the way in which its input profile there is distorted to an output profile. Thus these mappings are defined in terms of a profile $P =< h_1, h_2, ..., h_n >$. The system designer uses them to propagate profiles through the system structures being tried out.

Reliability Mapping. The reliability mapping carries a profile vector to a real value $R \in [0,1]$, the probability that the component will not fail on an input drawn according to this profile. To give this mapping on the data sheet, the component developer estimates the failure rates f_i within each subdomain using uniform random testing. Then

$$R = \sum_{i=1}^{n} h_i(1 - f_i). \qquad (1)$$

Given this mapping (that is, given the f_i measured by the component developer) and a profile (the h_i for the system to be designed), the system designer can use equation (1) to calculate R, the component reliability under that profile.

Profile-transformation Mapping. The profile-transformation mapping carries an input profile vector to an output profile, taking as a parameter the output subdomains. The mapping must be able to handle an *arbitrary* set of output subdomains $U_1, U_2, ..., U_m$, unrelated to the subdomains on the component data sheet, since such a set of subdomains will describe some following component in a system design. Let the weightings to be calculated for an output profile be

$$Q =< k_1, k_2, ..., k_m >$$

on subdomains U_i. Each k_j is the sum of the contributions from each input subdomain,

$$k_j = \sum_{i=1}^{n} h_i \frac{|\{z \in S_i | c(z) \in U_j\}|}{|S_i|}, \qquad (2)$$

where c is the function computed by the component.

The system designer, given an input profile for a component C (the h_i passed on by the component before C in the system design), C itself (to calculate c by executing it), the S_i from C's data sheet, and the desired subdomain breakdown for output (the U_i, which come from the data sheet of the component that is to follow C in the design), can use equation (2) to transform a profile through C to the next component. Each S_i can be randomly sampled, and the sample points mapped by c into the U_j. The fraction of points from S_i falling in each of the U_j, weighed by h_i, is the contribution to k_j from that S_i.

The construction using equation (2) is at the heart of our theory. In the sequel it will be called the *basic composition construction*.

4.4 System Design — an Example

Anticipating the discussion of system control structure in Section 5, here is a hypothetical example of the calculations using equations (1) and (2) that a system designer can make during design using components. Consider two functional software components A and B in sequence. A's input profile is presumed available from the previous component; A invokes B, passing its output as B's input. For this part of the analysis, the system designer needs to compute the reliability of the sequence $A; B$.

In order to keep the example simple, suppose that A and B both take a single integer parameter limited in magnitude to $2^{16} - 1$, and that A computes the function $f(x) = \sqrt{|x - 13|}$. Suppose that A's data sheet lists three subdomains:

$$A_1 = \{n | n < 0\}, A_2 = \{0\}, A_3 = \{n | n > 0\},$$

with failure-rate bounds of (say) .01, 0, and .001 respectively. The failure rate measurements would be obtained by the component developer from exhaustive testing on the small subdomains, and by uniform random testing that exposed no failures on the large subdomains. Suppose that the input profile to A is $< .3, .1, .6 >$. Then the reliability of A alone from equation (1) is:

$$R_A = .3(1 - .01) + .1(1 - 0) + .6(1 - .001) = .996$$

Suppose that B's data sheet has four subdomains:

$$B_1 = \{n | n \leq 0\}, B_2 = \{n | 1 \leq n \leq 10\},$$

$$B_3 = \{n | 11 \leq n \leq 100\}, B_4 = \{n | n > 100\},$$

with failure rates of .1, 0, 0, and .02 respectively.

A's profile-transformation mapping can be used with the subdomains from B's data sheet to calculate the profile B will see. (This is the basic composition construction.) A carries 0 to $\sqrt{13}$, so the second of A's subdomains maps entirely to the second of B's subdomains. Sampling uniformly with 1000 values in each of the two other A subdomains, the fraction of A outputs falling in B's subdomains are:

Subdomain	from A_1	from A_2	from A_3
B_1	0	0	0
B_2	.003	1.0	.002
B_3	.147	0	.162
B_4	.850	0	.836

Putting these numbers in equation (2):

$$\begin{aligned} k_1 &= .3(0) + .1(0) + .6(0) = 0 \\ k_2 &= .3(.003) + .1(1.0) + .6(.002) = .102 \\ k_3 &= .3(.147) + .1(0) + .6(.162) = .141 \\ k_4 &= .3(.850) + .1(0) + .6(.836) = .757 \end{aligned}$$

So the profile B sees from A is $< 0, .102, .141, .757 >$, and B's reliability is:

$$\begin{aligned} R_B &= 0(1-.1) + .102(1-0) \\ &\quad +.141(1-0) + .757(1-.02) = .986 \end{aligned}$$

The system reliability of the sequence is finally calculated as $R_A R_B = .996(.986) = .982$.

4.5 'Spikes' in Intermediate Profiles

When the basic composition computation is carried out even for simple examples like the one in Section 4.4, the applicability of the theory depends critically on the functional behavior of the first component in a composition. If that first function spreads its outputs relatively evenly across an input subdomain of the second, following component, then the tests done by the developer of the latter are valid, because they used a uniform profile on that subdomain. However, if the first component in the composition produces an output profile with a 'spike' in any following subdomain, then the developer's testing of the second component is called into question. This difficulty occurs in random system testing when a uniform profile is used because user data is not available, as discussed in reference [7].

As an extreme example, if a component computing the constant function with value K is first in a composition, then the subdomain S_K of the following component in which K falls has a spike at K. The basic composition computation assigns 100% of the input profile to subdomain S_K as it should, but the computation obscures the fact that uniformly sampling S_K and predicting its failure rate may be wildly inaccurate. In fact, if the second component is correct for input K then the composed failure rate should be 0; or, it it fails on input K, the composed failure rate should be 1. But the uniform sampling of the component developer is very unlikely to have tried input K at all. In a slightly more realistic example, a component that identifies leap years, placed in a business accounting system, may seldom be required to go outside the years (say) 1990-2010, and in any given year, will most frequently be asked about that year itself. If this component is tested by its developer with a uniform profile over a range of (say) 1,000 years, the effective sampling rate during test is 50-1000 times lower in the actual usage range than what the component will experience in place.

Although uniform sampling seems the wrong thing to do in testing a component that may receive a profile spike, it is also the only possibility. The chance that it will fail at exactly the points under the spike cannot be properly investigated when those points are unknown, and the uniform test profile is the best available.

4.6 Correct (Proved) Components

In extreme situations, a required system reliability may be obtainable only by using some components that are perfectly reliable — they have been proved correct. Such a component has a data sheet that specifies no subdomains or profile-transformation mapping, and a reliability function of constant 1.0. So long as only correct components are used in a system design, there is no need for calculation — the system reliability is 1.0. However, when correct components are used in conjunction with others whose data sheets include subdomains and the mappings given in Section 4.3, the correct components transform profiles, and the system designer must find these transformations to calculate how subsequent components will behave. Equation (2) can be used for correct components as follows:

> Let C (computing function c) and W be two components with statistical data sheets. Suppose that C is followed by a sequence of correct components, which without loss of generality we take to be one correct component V, computing function v. V is in turn followed by W. That is, the component sequence is $C; V; W$. Then the basic composition construction can be applied directly from C to W by replacing $c(z)$ in equation (2) with $v(c(z))$. That is, the correct component can be treated as if it simply extends the functionality of C.

5 System Design

The calculations of a system designer, made to determine if a component-based design is good enough to satisfy requirements, are driven by the control structure selected for the system's 'main' program.

5.1 An Algebra of System Design

From the data sheets of possible components and the structure of the system design, the system reliability can be computed. This section gives the rules for combining component reliabilities in sequences, within conditional constructs, and for indefinite loops.

Sequences of Components

At the end of Section 4.3, a basic composition construction for two components was presented in detail. The profile from the first component was transformed using equation (2) to a profile for the second component, and from these profiles the two component reliabilities were computed using equation (1). This construction can be extended to an arbitrary composition of components $C_1; C_2; \ldots; C_n$ by successively transforming an input profile to C_1 into profiles for the successive intermediate subdomains from the data sheets for $C_2, \ldots, C_n$.

Conditional

Consider a conditional composition of components:

`if` b `then` C_T `else` C_F `fi`.

The test b for the conditional partitions the domain D into $D_T = \{x \in D | b(x)\}$ and $D_F = \{x \in D | \neg b(x)\}$.

A conditional system structure can be analyzed with the basic composition construction. Let B be a component that is invoked just prior to the conditional, and let the conditional invoke a following component E. That is, the sequence is B; `if` b `then` C_T `else` C_F `fi`; E.

The system designer can calculate the reliability of the conditional as follows: Apply the basic composition construction to B; C_T, using the subdomains U_i for C_T, but in calculating the set in the numerator of equation (2), count a point z only if $b(c(z))$ is true. That is, if C_T has subdomains $T_1, T_2, ..., T_r$, in equation (2) take $m = r$ and $U_i = D_T \cap T_i, 1 \leq i \leq m$. Similarly, treat B; C_F with C_F's subdomains adjusted to include only the part of each where b is false, by intersecting the subdomains on C_F's data sheet with D_F. These two calculations determine the output profiles from B that are the input profiles for C_T and C_F, which allows the reliabilities of each (R_{C_T} and R_{C_F}) to be calculated from their data-sheet mappings. The reliability of the entire conditional construct is therefore the weighted average of these:

$$w_T R_{C_T} + w_F R_{C_F},$$

where the weights w_T and w_F are the fractions of outputs sent to D_T and D_F respectively, which can be determined from the input profile to B. (Another way to look at $< w_T, w_F >$ is as the profile into which the basic composition construction transforms B's profile, on the two subdomains D_T and D_F of the conditional.)

The conditional also influences the profile seen by the following component E. The basic composition construction can be used to map the input profiles obtained above for C_T and C_F to two profiles for E, by examining the sequences C_T; E and C_F; E, each using for output the data-sheet subdomains for E. The profile seen by E is then the weighted average (using w_T and w_F above) of these two profiles.

A conditional without an `else` is the special case in which C_F is a component computing the identity function with reliability 1.

Conditional glue logic may be more complex than the simplest form given above, by the inclusion of glue fragments in the two branches before and/or after C_T and/or C_F. However, such fragments can be treated as components in sequence and composed with C_T or C_F using the rules already given.

Indefinite Loop

Consider an indefinite loop: `while` b `do` C `od`.

The loop test b partitions D into $D_T \cup D_F$ as for the conditional.

A system designer can analyze a loop construction in a way similar to that described above for a conditional. The component before the loop supplies the initial profile for the loop-body component C (adjusted by the test b). Then C transforms this profile as input to itself (again adjusted by b). At each iteration of the analysis, the accumulated weighting that exits the loop increases, as C sends a fraction of its outputs into D_F. The designer repeats the calculation until the residual probability that control will remain in the loop is sufficiently small that the reliability of the loop body does not have much impact on the reliability of the loop construct as a whole. Each profile seen by C along the way can be used to calculate a reliability R_C for one iteration; the loop's reliability is the product of these. The profile seen by a component that follows the loop is the weighted average of all the profiles calculated for C along the way.

That is, the system designer considers a loop unrolling:

`if` b `then` C `fi`; `if` b `then` C `fi`; ...; `while` b `do` C `od`

and then replaces the part not unrolled with a conditional that fails:

`if` b `then` C `fi`; `if` b `then` C `fi`; ...; `if` b `then fail fi`

Suppose k copies of the conditional were unrolled. The basic composition construction applied to the sequence assigns a profile to the final component in which the weight assigned to D_T is negligible. Then the reliability calculation involves only a product of the k values of R_C under the successive profiles of the sequence. The zero reliability of `fail` has minimal impact, since its weight is near zero. As long as the loop terminates, this is a conservative estimate — it assumes that the residual executions of the loop *will* fail. A separate proof of termination is assumed.

In practice, k may be too large; or, if too few points are chosen in investigating the profile transformations, it may appear as an artifact of poor sampling that D_T is empty when it is not. Loops are the problematic constructions in program analysis, since it is always difficult to determine their composite effect. This analysis is no exception, but the situation is less grim than usual. This theory is only concerned with the way in which profiles are transformed by the loop, and this is far less demanding than to ask about the exact functional behavior.

5.2 Subroutine Calls

Although subroutines are the primary decomposition mechanism of programming languages, they are cannot be used directly to factor a system's reliability. The reason was identified by Parnas [20]:

> When X calls Y in a conventional language, X depends on Y to return properly and to produce correct results. If Y fails, X usually fails. Thus the reliability of Y cannot be separated from that of X. Parnas calls this usual call relationship USES(X,Y).

A subroutine-based system can be restructured so that its subroutines act as independent components to which our theory applies. Parnas called the relationship of independence INV(X,Y): X passes control to Y, but is indifferent to what Y does. When X is a component (or main program) calling component Y, and USES(X,Y), to transform the code so that INV(X,Y), separate X into fragment X1 before the call, and X2 after the call. X1 ends by invoking Y, passing it proper parameters, but also passing X2. Instead of returning, Y invokes X2, passing any result from Y.

Components themselves can be analyzed into fragments in the same way that system control structures were treated, but for simplicity in this explanation components are restricted to subsystems that make no outside calls. In this way the component developer's job remains measurement of properties of self-contained units, while the system designer's job involves calculation based on system structure.

5.3 Discussion of Glue-logic Design

As indicated in the analysis of the conditional construction above, glue logic can best be treated by reducing system structure to the simplest form possible. Any blocks of code in the highest level control structure of the main program should be treated as components themselves, to be split off, analyzed, and then composed with other components used. Subroutine calls should be restructured so that the INV relationship holds. This leaves a minimal control structure for analysis using the algebra of Section 5.1. This basic structure of the main program is not itself a component, in that while it does have a profile-transforming aspect, it is not given a reliability aspect, but treated as correct. The insight this provides is obvious in practice: the system structure should be as simple as possible, so that the system designer can trust it, and can be sure that the only source of design flaws occurs in the components, where the reliability theory is brought to bear.

6 Component Independence

The question of component independence does not arise in the basic composition construction or the algebra given in Section 5.1, except in a subtle form to be discussed below. Each component has its reliability mapping, independent of any other component, and the effect of one component on another is captured entirely by the profile transformations among them.

6.1 Sequential Case: Conservative Calculations

In a composition $A;\ B$, if A may be incorrect (that is, when only statistical testing information is given on its data sheet), a false profile for B may be produced by the basic composition construction, and thus A would influence the reliability calculated for B. Although we have done some preliminary work on this problem and found ways to estimate B's profile so that the reliability computations err only in the direction of safety, further investigation is needed.

6.2 Redundant Case: Design Diversity?

From a quality standpoint, all uses of sequential control structure compromise reliability. When two components are combined, the combination is subject to the failures of both. The sequence can magnify the failures of the second component to an arbitrary degree, since the profile it inherits may emphasize the subdomains where it is least reliable.

To buy back reliability lost in sequential design, or simply to obtain system reliability better than that of its components, requires the use of parallel, redundant system structures. The simplest scheme is 'Multi-Version Programming' (MVP) in which a computation is done independently three or more times and the majority result taken. If redundant computations are done by two components whose failure rates are f_A and f_B, then if their failure behavior is independent, the system failure rate is the product $f_A f_B$ (equivalently, the reliability is $1-f_A f_B = R_A+R_B-R_A R_B$). By using enough independent components that must agree, arbitrarily good reliability can be obtained in principle. Given the impracticality of testing software to safety-critical levels [4], the use of parallel components is the only way that (for example) commercial aircraft flight control programs can meet their safety requirements.

Unfortunately, it has been experimentally observed [10] that MVP does not realize the expected decrease in the failure rate, because in practice the redundant routines have coincident failures — they are *not* independent. 'Design diversity' is the name given to an attempt to minimize coincident failures by using disparate designs in the redundant routines.

The control structures of redundant computation are not fundamental; rather, they use sequences and conditionals. If the analysis of Section 5.1 is carried through for the control structure of MVP, the calculated reliability will not be better than that of one redundant component. Redundant computation requires special analysis that takes into account the attempt to replicate the same function in distinct components. Imagine that two algorithms are being executed with their results to be compared, each made up of components. In our model, diversity can be quantitatively described: The two computations are intuitively different if the subdomains and profiles at each interface differ.

7 Discussion — Theoretical Issues

In creating a data sheet, the component developer must identify a set of appropriate input subdomains for the component as the first step in measuring the profile mappings.

There appears to be little theoretical guidance in choosing subdomains for a given component. The developer faces an old testing problem: the best subdomains should be those on which the component 'acts the same' over the subdo-

main, so that samples of its behavior there are not misleading. It is the common wisdom (which unfortunately lacks any very strong theoretical or empirical support) that structurally defined subdomains best capture this 'sameness.' Structural subdomains are familiar to the software tester as the basis for coverage techniques; for example, branch coverage is defined by subdomains each of which corresponds to execution of one branch condition.

The other factor that a component designer must consider is potential profiles that will reach the component when it is placed in a system. The choice of a uniform testing profile is potentially dangerous because within a system, one component may send another a profile that contains 'spikes,' as described in Section 4.5. The developer can respond to this problem only by being careful to consider as many different behaviors as possible, assigning each its own subdomain. For this the 'functional' subdomains obtained from the component's specification are appropriate. The developer is not assigning profile weightings, and so can arbitrarily decompose functional domains based on any subjective criteria of 'sameness.'

Hence the best candidate subdomains for component testing are an intersection of natural structural- and functional-testing subdomains, as suggested by Richardson and Clarke [22]. However, the selection of subdomains in general is an open research problem.

7.1 Profile-independent Components

Although it is generally accepted that correctness proofs are not a practical method of software verification, for the serious component developer this position may need rethinking. The restricted size of components, and the existence of good specifications, may make proof an attractive alternative to testing. The great virtue of a correctness proof in our theory is that the reliability mapping is profile independent, and where a combination of components includes only those proved correct, it is unnecessary to carry along and transform the profile. In this property, two special cases are as good as proof:

Exhaustive testing. A component with only a finite input domain can be exhaustively tested so that its reliability mapping is 1; Knight [9] has suggested that this kind of design is possible more often than one might think.

Self-checking. Manuel Blum [3] and Paul Ammann [2] have independently suggested that some programs can be made to perform random redundancy checks at run time, which are sufficient to estimate a component reliability independent of profile. Although the reliability of such a component is not 1, it is profile independent.

7.2 Relaxing Assumptions

The primary limiting assumption of the theory presented here is that components are specified to compute pure functions. A functional-programming paradigm explicitly avoids state, and so this theory applies to functional programs without modification. It is an open question, which will only be addressed by experience in building and certifying component-based systems, whether it is wisest to keep state out of components and create it only at system level, or the reverse, to confine state to components and have little at system level. (The object-oriented development paradigm argues for the latter.) However state enters, it is not provided for in the theory presented here.

The proponents of random testing have long argued that system state can be handled within a pure-function theory by including extra 'state variables' in the input space. In principle this idea founders on an input profile for the state variables. Since state-variable values are created by the system as it runs, their profile is not, like that of the real input variables, under user control. The profile-transformation methods of Section 5 might be used to calculate a conservative state-variable profile; this would be most straightforward for system state created outside of components.

If structural subdomains are to be used for component testing, the assumption that the data-sheet subdomains are disjoint must be relaxed. As suggested in reference [8], it is always possible to form a true partition by intersection or union of overlapping subdomains, but there has been no investigation of the somewhat peculiar subdomains that result.

7.3 Size of Components

Component size can qualitatively alter the conception of what component-based software development means. It is quite different to imagine using a mathematical subroutine from a programming-language library, or Microsoft windows NT, as the typical 'component' being considered. Size does not enter the proposed theory, except through the implicit assumption that the component developer has tested a component to prepare its data sheet. It is silly to imagine that software of the size and complex functionality of an operating system (or even a compiler) can be tested as a whole, that its proper subdomains can be defined, and random tests conducted to make this theory applicable. If this theory does indeed describe the circumstances under which a reliable system can be built from reliable components, then it follows that such systems must be built from smaller components, ones for which direct measurements are feasible. Before an operating system could be considered as a component, it would have to be analyzed into its own components.

Component size does enter the theory through the assumption that components do not invoke other components.

So long as the measurements required on components can be made, this assumption is not restrictive. If a developer chooses to create a complex component, it can be analyzed in two ways: Either the internal structure can be ignored and the data sheet prepared from measurements for the whole; or, it can be treated as a (sub)system, and its data sheet mappings obtained from those of restricted components using the techniques of Section 5, in a recursive fashion.

8 Future Work

A plausible theory is the first step on the road to understanding system quality based on component quality. However, any theory must be validated.

8.1 Experimental Validation

Most experimental software research is very difficult to perform and consequently of doubtful value. Software development is a complex process, and different applications are developed in significantly different ways. So the investigator seeking to validate a practical technique faces an extensive, hard to control environment, and has difficulty characterizing 'typical' projects for study. The paradigm of theoretical hypothesis suggesting pointed experimental work that suggests changes in theory — the physical-science model that has been so successful — does not work for software theories.

But we are not proposing a software development process, or even a technique for practical use. Rather, we are trying to discover and understand the fundamental ways in which component reliabilities combine and influence each other in a system. It may be that the understanding gained will lead to practical development procedures, perhaps even brute-force application of the basic composition construction to design reliable systems, but that is not the subject for initial validation.

The physical-science model of experiments that direct basic theory applies here. Any particular piece of software is made up of components, which can be identified ex post facto, and data sheets can be derived from testing of these components. Using an arbitrary system operational profile, the theory can be used to calculate system reliability. Then the system reliability can be measured using conventional random testing with that profile. Comparing the measured value with that calculated from component values provides an overall check of the theory. A single system can be the source of many experiments, by varying the system input profile, and by varying the granularity of the units chosen to be its components. When there are discrepancies between theory and experiment, such an experiment contains intermediate information to help improve the theory. Instrumentation of the system under test will yield the profiles induced at each component, for comparison with those calculated from the theory. So-called 'toy' programs, usually uninformative about software issues, are the best subjects for revealing experiments. Beginning programming textbooks are a good source of programs, and simple UNIX command implementations are another. A preliminary experiment reported in [17] used the UNIX 'grep' program. Since grep is a subroutine-based program, it had to be broken into artificial components as described in Section 5.2. This experiment was the source of improved understanding of 'glue logic,' and yielded promising results.

8.2 Application to Practice

If this theory leads to a practical paradigm for component-based software engineering, it will be one in which component developers prepare data sheets, and system designers use them to select 'good enough' components and to evaluate system designs. The experimental investigations suggested in Section 8.1 require the investigators to themselves use this paradigm. In principle, the work can be done using existing tools and hand calculation, much as was done in the example of Section 4.4. We plan to develop research prototypes of tools to ease the burden of hand calculation.

Random testing by the component developer is a well understood process that does not require new tools. It is the system designer who needs practical help applying this theory. Even a simple system structure requires many applications of the basic composition construction of Section 4.3. It may be necessary for the system designer to try a number of system input profiles, and a number of trial designs, thus repeating these many constructions.

A supporting tool would take as input a system control structure, an assumed operational profile for the system in histogram form, a collection of components' data sheets, and the executable components themselves. It would then apply the basic composition construction to map the system input profile through to each component in turn, calculate its reliability in place, and combine these component reliabilities into a system reliability. The existence of a tool will not only make the experimental work much easier, but will allow measurements to be made on how efficient the brute-force algorithms are, and whether they scale to systems of practical size.

References

[1] A. F. Ackerman. http//: www.izdsw.org/projects/-CodeGrades/index.html.

[2] P. Ammann and J. C. Knight. Data diversity: an approach to software fault tolerance. *IEEE Trans. on Computers*, pages 418–425, 1998.

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

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

[5] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.

[6] D. Hamlet. Software component dependability, a subdomain-based theory. Technical Report RSTR-96-999-01, Reliable Software Technologies, Sterling, VA, Sept. 1996.

[7] D. Hamlet. On subdomains: testing, profiles, and components. In *Proceedings ISSTA '00*, pages 71–76, Portland, OR, 2000.

[8] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16:1402–1411, 1990.

[9] J. Knight, A. Cass, A. Fernandez, and K. Wika. Testing a safety-critical application. In *Proceedings ISSTA '94*, page 199, Seattle, WA, 1994.

[10] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, 1986.

[11] L. Krishnamurthy and A. Mathur. The estimation of system reliability using reliabilities of its components and their interfaces. In *Proceedings 8th Intl. Symposium on Software Reliability Engineering*, Albuquerque, NM, USA, Nov. 1997.

[12] S. Kubal, J. May, and G. Hughes. Building a system failure rate estimator by identifying component failure rates. In *Proceedings Fifth International Symposium on Software Reliability Engineering*, pages 32–41, Boca Raton, Florida, USA, Nov. 1999. IEEE.

[13] J.-C. Laprie. Dependability evaluation of software systems in operation. *IEEE Transactions on Software Engineering*, 10(6):701–714, Nov. 1984.

[14] J.-C. Laprie and K. Kanoun. *Software Reliability and System Reliability*, pages 27–70. In Lyu [16], 1996.

[15] B. Littlewood. Software reliability model for modular program structure. *IEEE Transactions on Reliability*, 28(3):241–246, Aug. 1979.

[16] M. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, 1996.

[17] D. Mason and D. Woit. Software system reliability from component reliability. In *Proc. of 1998 Workshop on Software Reliability Engineering (SRE'98)*, Ottawa, Ontario, July 1998.

[18] D. Mason and D. Woit. Input domain analysis for software reliability measurement. In *The Fifth International Conference on Computer Science and Informatics*, Atlantic City, USA, Feb. 2000.

[19] J. Musa, G. Fuoco, N. Irving, and D. Kropfl. *The Operational Profile*, pages 167–216. In Lyu [16], 1996.

[20] D. Parnas. On a "Buzzword": Hierarchical structure. In *Proc. IFIP Congress*, pages 336–339. North-Holland Publishing Co., 1974.

[21] D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In *Proceedings ISSTA '94*, pages 58–65, Seattle, WA, 1994.

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

[23] D. Woit and D. Mason. Software component independence. In *Proc. 3rd IEEE High-Assurance Systems Engineering Symposium (HASE'98)*, Washington, DC, Nov. 1998.