THEORY OF SYSTEM RELIABILITY BASED ON COMPONENTS[1]

Dick Hamlet
Portland State University
Portland, OR, USA
+1 503 725 3216
hamlet@cs.pdx.edu
Dave Mason
Denise Woit
Ryerson Polytechnic University
Toronto, Ontario, CANADA
+1 416 979 5000
{dmason,dwoit}@scs.ryerson.ca

ABSTRACT

Most software-component research has been directed at functional specification of software components. The other, equally important, side of the coin is component quality. We present a foundational theory of reliability based on components. The theory describes in principle how component developers can make measurements that are later used by system designers to calculate — without implementation and test — system reliability. The theory is a ”microscopic” one that describes in detail how component properties are reflected in systems designed using those components.

1 PITFALLS OF SOFTWARE 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, use mostly subjective assessments of software quality. It would be of great value to replace these with hard data.

In electrical and mechanical engineering, 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 mechanical components, these constraints concern, for example, the life expectancy of the component. Software is embedded in systems with mechanical and electrical components, systems designed using component techniques from these other branches of engineering. The system designer using an embedded software system would like the software components to have data sheets.

Without the solid information of a data sheet, software components 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 [10] is a candidate for describing “quality” on a component’s data sheet, but it cannot be applied without addressing a central problem. Whereas reliability of mechanical components depends on their physical environment, and can be given without regard for the expected usage so long as that usage remains within a

[1]A short abstract describing some of this work appeared in ISSRE ‘99, Boca Raton, FL, Nov 1999; it was also used in a position paper for Workshop-12 in OOPSLA-99, Denver, CO, Nov 1999.

tolerance range, software reliability depends critically on usage, in the form of the so-called "operational profile." 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. The profile seen in the system depends on the details of system design, and can be very "spikey," so no standardized test profile can be justified.

Section 3 proposes a solution to this profile problem.

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 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 [15].

The problem of system reliability from component data has been studied for more than 20 years [9, 7, 8, 5, 6]. Perhaps because a straightforward analysis seems overwhelming, these 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 analyzed in full detail. Microscopic theories like this one are the most satisfying in explanatory power, and the easiest to experimentally investigate, but they may be difficult to apply in practice without supporting tools.

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 at a reasonable cost. Subjective assessments derived from the software development process do not qualify as data-sheet information, because they amount to no more than assertions that the developer is trying to do a good job. For example, stating that the developer uses a careful inspection process is no help in calculating statistical reliability. A current pre-standards study group on component quality is almost exclusively using such subjective measures [1]. The reputation of the developer is not an inconsequential factor in selecting a component; but, its role should be to convince the designer that hard technical data can be believed.

A statement that (say) for a certain user profile a component's 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 using that profile. The reader of a data sheet might doubt that the developer has actually established such a precise technical claim. But this is a question that can be answered scientifically, and if the developer has lied 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 information must take profile as a parameter. That is, the data sheet specifies *mappings* from profile to reliability parameters [12].

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 developer to test a component within these subdomains without knowing the profile, whose arbitrary weights may be applied later [3].

2.2 Component and System Development Process

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

- 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 mappings of profiles expressed as weightings over them.
- The system designer decides on a system structure utilizing components.
- Using the data sheets for prospective components the system designer calculates the system reliability for a given user profile of the system.
- If the necessary system reliability is not achieved, better components must be selected or the system structure changed.

3 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 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.

We propose 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 predict the system reliability before implementation.

3.1 Profiles Defined on Subdomains

A profile P is a probability density $P : D \to [0,1]$, where D is a discrete input domain. In practice, it is very difficult to obtain 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 [13]. 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 there is no justification for other than 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.

3.2 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 random testing. Then

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

Given this mapping (that is, given the f_i) and a profile (the h_i), the system designer can calculate R, the component reliability under that profile.

Profile-transformation Mapping. The profile-transformation mapping carries an input profile vector to an output profile. It is important that the latter be expressed as a weighting vector over an *arbitrary* set of 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 of any 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,

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

where c is the function computed by the component.

The system designer, given an input profile for a component (the h_i), the component itself (to calculate c), the S_i from the component's data sheet (which can be randomly sampled to estimate membership in the numerator of equation (2)), and the desired subdomain breakdown for output (the U_i for a following component), can use equation (2) to transform a profile through a component, that is, to map P into Q. This construction is at the heart of our theory. In the sequel it will be called the *basic composition construction*.

3.3 System Design – an Example

Here is an example illustrating the calculations a system designer can make during design using components, using the basic composition construction.

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 A's data sheet lists three subdomains:

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

with failure rates of .01, 0, and .001 respectively. Suppose B's data sheet has four subdomains:

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

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

with failure rates of .1, 0, 0, and .02 respectively. (The data sheet measurements are obtained by the component developers using exhaustive testing on the small subdomains, and by uniform sampling that exposed no failure on the large subdomains. Finally, suppose that the input profile to A is the weighting $< .3, .1, .6 >$.

The system developer calculates as follows:
The reliability of A alone is:

$$R_A = .3(1 - .01) + .1(1 - 1) + .6(1 - .001) = .896$$

from equation (1). A's profile-transformation mapping can be used with the subdomains from B's data sheet as the U_i in equation (2), 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

(For simple numerical functions like A's, all of the fractions in equation (2) can be found analytically, but in general sampling and executing the component will be required.) 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 $R_A \times R_B = .896(.986) = .883$.

4 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:

"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 [16]. Section 5.1 indicates the necessary transformation.

System control flow. Components are invoked in patterns that come from the top-level control structures in the system design, the "main program" in a conventional programming language. These patterns are handled by providing an algebra of the constructions of sequence, conditional, and looping. Section 3.3 illustrates the sequence case. Conditionals can be handled exactly using the basic composition construction separately for the two branches. Loops are more difficult; they require the system designer to iterate the conditional construction until the profile being propagated back into the loop has sufficiently small weight.

Our theory suggests that glue logic should be kept as simple as possible, with as much complexity as possible pushed off onto the components. This shifts the burden of testing and analysis to the component developer, who stands to gain by creating a quality component. Even the simplest control structure will require a good deal of calculation to analyze, so tools to help the system designer will be essential.

5 COMPONENT INDEPENDENCE

The problem of component independence takes two quite different forms.

First, components' data sheets are developed in isolation, and cannot be expected to describe dependent behavior. So the system calculations require that this independence be preserved. The usual subroutine-calling mechanism incorporates the reliability of the called routine with that of the caller. When X calls Y in a conventional language, X depends on Y to return properly and to produce correct results. If Y fails, X fails. Thus there is no sense in which Y has a reliability independent of X. Transforming profiles through components can also introduce a subtle dependence. These dependencies can be eliminated or allowed for in conservative approximations.

The second question of independence arises only in systems that use redundant components to achieve a higher reliability than that of the components themselves. The question of coincident failure [4] then becomes crucial. This very difficult problem is not yet addressed by our theory.

5.1 USES vs. INV

Parnas [14] has characterized the relationship of one routine calling another, by a pair of alternative predicates:

USES(C, D) if and only if C calls D and C is considered incorrect if D does not function properly.

INV(C, D) if and only if C passes control to D but is indifferent to what D does.

If USES(X,Y), then the reliability of component X will incorporate the reliability of Y and their reliabilities are intertwined. The relationship needed for independent components X and Y is INV(X,Y).

When X is a component (or main program) calling component Y, their usual relationship is USES(X,Y). To transform to INV(X,Y) requires that the calling program be separated into fragment X1 before the call, and X2 after the call. X1 ends by invoking Y, passing it proper parameters, but also passing X2. When Y is ready to return, it instead invokes X2. Details of this fragmentation differ depending on the control structure of X. In effect, the calling of components treats the system design as if it were itself made up of artificial components invoking each other and the "real" components.

Components themselves can be analyzed into fragments, 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 estimation of properties of self-contained units, while the system designer's job is calculation based on system structure. The proper level at which to make the component/system distinction is an open question.

5.2 Sequential Case: Conservative Calculations

Even when components invoke, rather than use, each other, a subtle dependence may be created by the profile that the invoking component provides to the invoked component. If A invokes B and A is incorrect, a false profile for B will be calculated by the basic composition construction, and thus B and A are not independent. 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.

5.3 Redundant Case: Design Diversity?

System designers use the set of “structured” control constructions to keep intellectual control of a design. From a quality standpoint, however, all uses of sequential control compromise reliability. When two components are combined, the combination is subject to the failures of both. The sequence can in fact magnify the failures of the second component to an arbitrary degree, since the profile it inherits may emphasize the points where it fails. (The opposite can also happen, but still the combination has the failures of the first component.)

To buy back reliability lost in sequential design, or simply to gain system reliability better than that of its components, requires the use of parallel, redundant system structures. A “voting” structure repeats a computation three or more times and compares the results, the majority being taken as the answer. (This scheme is sometimes called “Multi-Version Programming” or MVP.) 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, an arbitrarily good reliability can be obtained in principle. Given the impracticality of testing software to safety-critical levels [2], 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 [4] that MVP does not realize the expected product decrease in the failure rate, because in practice the redundant routines have coincident failures — they are *not* independent. Furthermore, it is impractical to determine by testing if the problem exists.

The investigation of coincident failures and design of sound schemes for redundant computation is the most important unsolved problem in high-reliability computing. Our theory sheds a little light on the situation. Imagine that two algorithms executing in parallel with their results to be compared, are themselves made up of components. Consider a coincident failure of the two. They begin with a common input, and end with a common (failed) output. However, along the way the computations may differ, and it is the hope of advocates of “design diversity” that enough difference will make coincident failure less likely. In our model, diversity can be quantitatively described: The two computations are intuitively different if the subdomains invoked at each interface differ.

6 FUTURE WORK

A plausible theory is the first step on the road to achieving system quality based on component quality. However, any theory must be validated, and then put into practice.

6.1 Experimental Validation

A microscopic theory lends itself well to empirical validation, which can play also play a role in the theoretical development.

Most experimental software research is very difficult to perform. Software development is a complex process, and different applications are developed in significantly different ways. So the investigator seeking to validate a 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 spectacularly successful in physics, for example – does not work well for software theories.

However, for a microscopic theory like this one the physics paradigm does apply. Any particular piece of software is made up of components, which can be identified ex post facto, and data sheets derived for these components. Assuming any particular system operational profile, the theory can be used to calculate system reliability. Then the system reliability can be measured using conventional random testing with the assumed 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 validation experiments, by varying the system input profile, and by varying the granularity of the units chosen to be its components. Furthermore, if there are discrepancies, such an experiment contains intermediate information to help track down the source. Instrumentation of the system under test will yield the profiles induced at each component, for comparison with those calculated from the theory.

The expected result of such experiments would be disagreement between calculated and measured reliability, which would pinpoint failures to be corrected in 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 [11] used the UNIX ‘grep’ program. It was the source of improved understanding of “glue logic,” and yielded promising results.

Pointed, revealing experiments are part of the development of a sound theory. Validation of the theory that emerges must be performed on larger systems, with all the attendant difficulties of software experimentation. Large-scale validation will require good supporting tools.

6.2 Application to Practice

Our theory assumes that component developers will prepare data sheets, and that system designers will use them to select “good enough” components and to evaluate system designs. In principle, the necessary work can be done by engineers using existing tools and hand calculation. However, if there is to be substantial use of the theory in establishing an accepted method of component development and system design, the existence of specific tools will be important in convincing engineers to use the theory. We plan to develop research prototypes of the necessary tools, in parallel with experimentation described in section 6.1.

Tool Support for Component Developers

To create a data sheet for a software component, its developer needs to identify appropriate subdomains, and then to perform conventional random testing within them. Existing software-reliability tools (which do little more than organize test data and evaluate the formulas for predicted reliability) are adequate for this work. It will probably be useful to create a cosmetic interface directed at the component developer.

Tool Support for System Designers

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 3.2. 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. Although the calculations are straightforward, they can be extensive, because a brute-force implementation of the basic composition construction uses equation (2) with random coverage of subdomains.

A supporting tool would take as input a system control structure, an assumed operational profile for the system in the histogram form, a collection of components’ data sheets, and the executable components themselves. It would then use 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.

In using the basic composition construction, such a tool is doing almost all the work of executing a small random testset on the proposed system. Only the glue logic of the main control structure is not executed. It might therefore be worth investigating the possibility of completing the execution and using an oracle to verify that the results are correct. This addition would have two advantages: (1) It would solve the problem of loop calculations in a brute-force way; and (2) It would allow the system designer to verify that data-sheet information is roughly correct, by comparing differing components placed in the same system design.

Even the best theory is a long way from practice and an industry standard. But however much standards are needed, useful ones cannot be devised without underlying theory. The process begins with sound theoretical work, widely discussed in the software-engineering community.

References

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

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

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

[4] 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.

[5] L. Krishnamurthy and A Mathur. The estimation of system reliability using reliabilities of its components and their interfaces. In *Proceedings ISSRE 97*, Albuquerque, NM, November 1997.

[6] Silke Kubal, John May, and Gordon Hughes. Building a system failure rate estimator by identifying component failure rates. In *Proceedings ISSRE 99*, pages 32–41, Boca Raton, FL, November 1999.

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

[8] J.-C. Laprie and K. Kanoun. *Software Reliability and System Reliability*, pages 27–70. In Lyu [10], 1996.

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

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

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

[12] Dave Mason and Denise Woit. Input domain analysis for software reliability measurement. In *To appear, The Fifth International Conference on Computer Science and Informatics*, Atlantic City, USA, February 2000.

[13] John Musa, Gene Fuoco, Nancy Irving, and Diane Kropfl. *The Operational Profile*, pages 167–216. In Lyu [10], 1996.

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

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

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