University of Bristol

DEPARTMENT OF COMPUTER SCIENCE

Component-Based Software Reliability Analysis

John May

May 2002 | CSTR-02-002

Component-Based Software Reliability Analysis

John May
Safety Systems Research Center
Dept. of Computer Science
University of Bristol, UK

j.may@bristol.ac.uk

ABSTRACT

New reliability models are derived to describe how test-based software reliability estimates depend on the component structure of code. The models can analyze dependent component failures. Models of this type are important for two main reasons.

Firstly, they provide a quality model for software development based on component reuse. For simple software architectures, it is shown that it is feasible to re-use evidence of a component's reliability from previous testing/usage in a different system. This technique has potential to provide extremely efficient software verification.

Secondly, the new models provide a meaning for 'reliable software design,' making it possible to identify software designs whose reliability can be demonstrated. Traditionally, the complexity of a computation is measured in terms of the number of elemental computations used. In contrast, a statistical complexity measure is proposed to describe complexity in terms of statistical software testability. A highly complex program requires intense testing in order to justify a claim that the program achieves a given level of reliability. A low complexity program requires less testing to achieve the same claim.

1. INTRODUCTION

Statistical Software Testing[1] (SST) is a branch of testing research into methods of obtaining one of the most practically useful of all software test adequacy measures, namely the reliability statement warranted by test results. SST models estimate operational reliability for programs whose reliability is stable [3]. In its simplest form, Black Box SST (BBSST), SST has been successfully applied to complex commercial systems [1]. BBSST is so called because, given a N valid BBSST test results, the code (length, complexity, or any aspect of its structure) has no effect on the reliability estimate produced by the BBSST model.

However, neither BBSST nor any other existing testing models are capable of adequately supporting modern component-based software development techniques. To support these techniques, testing models are needed to:-

1. Explain how the failure probability of software depends on its components;
2. Be able to exploit re-used software components of known reliability in estimating an overall software system reliability;
3. Provide a new definition of software complexity based on testability which can be used to discover how to design component-based software that is inherently testable i.e. demonstrably reliable.

In the second point above, re-use means more than the transplantation of code from an old system into a new one. It also entails the ability to use a component's history of successful previous testing or usage in the old system as evidence of its reliability in the new system, and thereby to contribute to the reliability estimate for that new system.

The three points above require statistical models to describe the failure patterns of both individual software components and compositions of those components. This requires an understanding of the interplay between deterministic software behavior and the randomized choice of inputs in statistical testing. Such models can be called structural, architectural or component-based models. The advantages of the models (e.g. 2 and 3 above) mean that they are highly desirable, potentially providing a new, highly efficient software verification method. However, they have not been widely applied. The main remaining reason for this is the problem of dependent component failures. Previous structural models have not succeeded in analyzing this phenomenon, and neither has it proved possible to argue that components fail independently in real-world software. This paper reports Component-based Statistical Software Testing (CBSST) models, which propose a solution to the dependency problem in simple cases and that appear suitable for generalization to more complex software architectures.

1.1 Background

This section presents the core concepts of BBSST that remain relevant to the discussion of CBSST in later sections. All SST models verify software design using empirical evidence from testing or, if the software has been observed appropriately, from operational use. In BBSST, when a software system executes correctly on N valid BBSST tests, it is possible to make an estimate of the system probability of failure on demand ***(pfd)*** [11]. This statistical test experiment requires that test selection is performed by sampling from a global set of possible tests to produce the 'statistical test set' that is presented to the software. The frequency distribution of the different kinds of tests in the statistical test set must be consistent with the operational profile faced by the software in practice [12]. Generation of the correct test frequency distribution clearly requires knowledge of the inputs the system will see over a representative life or mission time. However, with this knowledge, methods for the synthesis of the distribution are available. The essential techniques have been

[1] Sometimes called Statistical *System* Testing since it can be applied to design verification of hardware or software systems, and to systems consisting of both.

developed and demonstrated for a real large system application [1].

In addition, an argument must be made to justify the assumption made in the statistical model that the behavior of the system on each test is always independent of the previous tests that have been performed i.e. that the outcome of a particular test is not influenced by internal software states computed during previous tests. In BBSST this is traditionally justified by:-

- A demonstration that the software algorithms do not cause any state transfer across tests; or
- The system is repeatedly driven by its environment into the same start state for each test; or
- The system resets itself to the same start state for each test.

This clearly places constraints on test definition. Consider a boiler protection software (BPS) that monitors the state of a boiler and shuts down the boiler if its state becomes unsafe. Suppose the BPS calculates rates of change of certain of its input parameters, which are obtained by polled readings from sensors monitoring critical boiler parameters. Then a BBSST test can not be one cycle of inputs read from the sensors. However, it may be defined as a 'transient' of parameter trajectories over time, that departs from steady state operating conditions as the result of a boiler fault (e.g. a stuck valve), and finishes at the point where the BPS should shut down the boiler, according to the BPS specification.

The fundamental reason why CBSST models are needed to support component-based development is that the BBSST model cannot incorporate or reveal information about software components since components have no identity in the model – the model does not reference 'parts' of the software in any way. The most important characteristic of CBSST models is that they do reference the software 'parts.'

1.2 Previous work

Much of the work on measurement of test effectiveness, and software quality assessment based on testing, has emphasised the importance of code structure [14]. However, it has proved difficult to link code structure and statistical testing models. One line of research has studied the effectiveness with which SST achieves code-based test adequacy measures [13]. Another approach, that we will call Structural Statistical Software Testing (SSST), seeks to evaluate the influence of software structure reliability on reliability estimates. SSST models reference parts of the software, but the 'parts' are not necessarily components in the usual sense of the word – they might be execution paths for instance. A review of some of these models is given in [2]. The proposed models in this paper fit into this latter category. They are the latest in a particular line of research to determine how different code structures produce different reliability estimates given a number of failure free statistical tests. Some of our early work in SSST is reported in [8] [9] [7].

In our recent work on SSST, software is represented as a set of components [10] [4] [5] [6]. We call these models CBSST models. A CBSST model is a type of S^3T model that focuses on the way in which the reliability estimate for a whole program is influenced by reliability estimates for its components, where a component is defined in a traditional way. A component is a physical 'package' that, without internal modification, can be physically plugged in to a system to provide some fixed functionality. One difficulty with our previous CBSST models is that it is not clear what code should be chosen to represent a component, yet different software decompositions produce different failure probability estimates. A further problem is common with all SSST models - independent failure of components is assumed. Both problems are addressed by the CBSST models proposed in this paper.

In our previous CBSST models, both series and conditional component execution structures are studied. The CBSST models in this paper take a different approach, and show that the type of data flow between interacting components is important in ways that have not been identified before. By data flow we simply mean any exchange of data between components, such as occurs by for example, parameter passing in method calls, or global variables. Data flow determines the software decomposition required by the models, and different statistical modeling is required for different patterns of data flow**.** The research studies some simple data flow mechanisms, and will need generalization to cover other forms of communication between components in later work.

2. MODELLING

In CBSST, software components whose stochastic failure pattern under testing is described by the BBSST model are used as fundamental elements. It is assumed that failure of a component that satisfies the test independent conditions in section 1.1 is described by the BBSST model. A fundamental modelling concept in this paper is that it is possible to decompose software into (compose software from) a set of components that, under specially designed statistical system testing, each behave independently of their own test history on a given test. For the same system tests, history dependent *system* behaviour can occur because a components behaviour may be dependent on the test histories of other components.

The BBSST model is also called the Single Urn Model (SUM), and the random pattern of failures it describes will be referred to as a SUM process. Components failing according to that pattern will be called SUM process components, sometimes abbreviated to 'SUM components.' Whether or not a component failure pattern is a SUM process depends on a triple (C, S, T): respectively the component code, its specification, and the statistical tests it receives.

A statistical test set T for a component C with specification S, here means a sequence of tests generated by picking tests sequentially with replacement, using a random choice mechanism from a set ψ that covers all valid executions of C based on S, and according to a probability distribution (the operational profile) over ψ. A statistical test set T does not necessarily result in a BBSST failure pattern during testing. That is, relative to a valid statistical test set and a component specification, not all components are SUM components. The reason is that the determinism of the software computation can prevent the random *test choice* mechanism from causing random *failures*. Section 2.2 provides a concrete example of how this can occur.

Simply testing all of a system's components individually is not a full test of a system. The interactions between components also need to be tested. Interactions between components are caused by the *connectivity* between components: the program code causing data transfer and flow of control between components. Traditionally, tests of components are called unit tests, and tests of connectivity are called integration tests. We make a simplifying assumption that failure of any component or connectivity fails the software.

In this paper connectivity is restricted to simple code mechanisms for sharing state (i.e. data) between components, such as parameter passing or use of global variables[2]. Conditional execution is not studied; all components execute on each test. In a system *decomposition* we distinguish components and their connectivity. Each software *element* (component, connectivity) in this picture receives testing during system testing, and in addition components may be tested separately. Given a specification for the system and each of its components, it is clear that a system can fail on a demand (contradict its specification) due to its connectivity when all of its components are meeting their specifications perfectly. The source of such failure is incorrect connectivity – it is 'wired wrong.' We define a *SUM interaction* to be any interaction between two SUM components that causes the two components considered as one entity to fail as a SUM process.

A hierarchical view of decomposition can be taken, since components can be defined at different levels of abstraction. However, this paper only studies one level of decomposition.

In this paper, given a statistical test set T and specification S, any decomposition D is acceptable (in the sense that it permits analysis) where the failure behaviour of each component under T is a SUM process. Whilst the models in this paper do not analyse the code at the instruction level, it is natural to think of components as collections of instructions. The question is where to draw the component boundaries. Consider code comprising m instructions. If m is anything other than a very small number, there are a huge number of possible partitions of the code instructions, most of which contain a vast number of components (i.e. partition subsets) chosen from the set of subsets of the set of single instructions. An plausible constraint is that instructions in a single component are next to each other in the code text and this is what happens in normal software development, where component boundaries are drawn according to the developer's intuitive allocation of function to components. This results in a natural description of a component's interf ace with its environment. These constraints reduce the combinatorial explosion, but do not produce a unique decomposition. In the remainder of section 2, some results are presented to indicate how CBSST suggests additional criteria for system decomposition i.e. taking the bottom-up view, CBSST suggests new constraints on how to connect components to build software.

2.1 Non-branching, SUM interaction

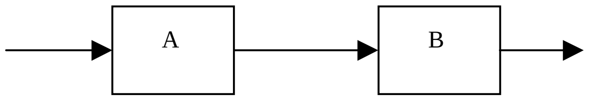

Figure 1 A two-component system

Consider a linearly executed list of instructions without jumps i.e. with no conditional, iterative or recursive execution. Assume for the moment that this list can be partitioned into two SUM components in series, interacting with data flow shown by the arrows in Figure 1. In diagrams such as Figure 1, a box represents a SUM component and an unbroken arrow represents data flow that occurs within the duration of a single test, communicating data that has not been influenced by previous tests. This connectivity is described as non-branching since A and B both execute on all tests.

We will initially assume that the connectivity and resulting data flow is correct. In this case, it can be argued that the system in Figure 1 behaves as a SUM process. The reason is that this configuration ensures an unchanging size of failure domain in the system test space over time (i.e. for every test in any test sequence). The test selection mechanism is the only influence on whether a given test succeeds or fails, and that mechanism is random. *A sufficient characteristic for a SUM process is that the failure domain in the test space remains constant over time* (imagine a Venn diagram on the set of tests). The consequence is that the overall system behaves as a single SUM process.

It is important to identify the conditions under which the above kind of component composition applies since it leads to simpler modelling, and more efficient reliability estimation – fewer tests are needed (section 4). Therefore decompositions which are minimal in the number of SUM components are of interest.

In practice, the connectivity cannot be assumed correct. It must be treated as a third 'element' which itself can fail. Connectivity failures (system failures not due to either component contradicting its specification) in Figure 1 also have a static failure domain. The connectivity involves only simple data delivery, and so its correctness on any test can not be dependent on previous test history. Hence the whole system, including connectivity, is SUM.

Figure 1 shows a very simple example. SUM components and interactions can be complex. Internally, components are not restricted to non-branching code, and interactions need not have uni-directional data flow. This is discussed further in section 3.1.

2.2 Non-branching, non-SUM interaction

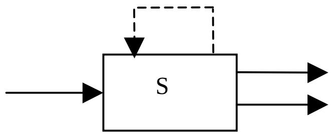

Figure 2 A system transferring state across tests

A system S in which data is transferred across tests is shown in Figure 2. The dotted arrow indicates transfer of data across tests. S is arbitrarily shown with one data input and two data outputs. Such a configuration will generally, although not necessarily, result in a non-SUM process. This section looks at an example that produces this system behaviour by composing two SUM components as in Figure 3.

The dotted arrow shows the same data flow as the dotted arrow in Figure 2 and indicates that on a single test, A performs a computation that is dependent on B's state in previous tests. The computation of A is therefore 'conditioned' by previous tests. A simple example of code that is described by Figs 2 and 3 is given below. For example, `t` could be an integer, x a global variable, m a local variable, and the f_i simple arithmetic functions.

[2] If components are 'glued' together using arbitrary programming, this additional 'glue code' must be separated into additional components and connectivity.

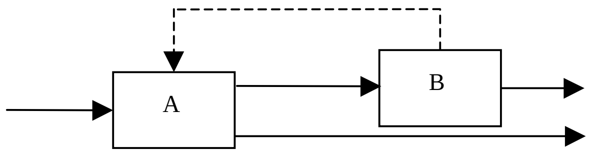

Figure 3 Non-SUM connectivity

```
{component A:}
input(t)
m = read_memory(addr)
x = f1(t)
output (f2(x,m))
{component B:}
write_memory(addr, f3(x))
output(f4(x))
```

The persistence of state between tests in Figure 3 means the resulting system cannot be assumed to behave as a SUM process despite A and B behaving individually as SUM components. An example can be constructed to illustrate this. We will initially assume that the connectivity is correct. Suppose that when B fails on any test in its failure subset in the test space, it correctly generates specific data d (generated by $f_3(x)$) to send to A (which A receives in the following test), and that when B succeeds the data it sends to A is never d. Further assume that d is an acceptable input for A according to A's specification. Now suppose that A fails on receiving d from B, irrespective of the value of t. Under these circumstances the system fails in a particular pattern. If the results of system tests were spread across the page from left to right with time, and a dot represented a test failure whilst a space represented a test success (this will be called a fail line), the pattern would have the following form.

```
..     ..  ..       ..   ..         ..    etc.
```

This failure pattern is not a SUM process: a failure of B is always followed immediately by failure of A on the subsequent test. The example shows how easily interactions between components can produce a non-SUM process. In contrast to the system of Figure 1, we cannot employ the SUM to analyse system reliability directly. In general, it might sometimes be possible to redesign the system tests to achieve SUM testing. However, the aim of the CBSST models in this paper is to allow re-use of component tests as evidence of component reliability, and we know of no way to model this in the case where the tests vary in different applications.

In the above example, the subset of system tests that fail is not constant, it varies over time. It does so because A's fail region in the system test space varies over time depending on the data A receives from B, that is *it depends on computation in a previous test*. The fail region in the system's test space caused by B remains constant. B's computation on a system test does not depend on test history.

The example given above shows only one possible pattern of failure behaviour. In general, the computation can produce an endless variety of failure patterns. Analysis of specific patterns is not possible since it relies on knowledge of the actual failure behaviour, which is not known. Therefore we seek a bounding analysis. Our key assertion is that the example interaction described above for Figure 3 is special - it provides a bounding case. If we estimate the system failure probability on demand (pfd) assuming this form of interaction, then if the real interaction is anything different, the estimate will be conservative i.e. an overestimate. The argument for conservatism is developed below. It is based on the observation that the pattern in the above fail line is the one with the largest probability of long fail-free test sequences, when the system failure probability is as large as it could possibly be given the two component pfds, θ_A and θ_B.

The example of adjacent test failures, described above for Figure 3, shows a particular extreme form of dependence between failure of A and B. It is extreme in the following senses: the dependence on the same test is total negative dependence (exclusivity), whilst the dependence between different tests shows the closest clustering possible. This can be pictured using separate fail lines for A and B.

```
A    .   . .     .   .     .     . .
B   .   . .     .   .     .     . .
```

In general, estimation of system failure probability using component fail probability estimates must make assumptions regarding the dependence of failures between any two components. These can be separated into the two above kinds of dependence. Firstly, dependence regarding the coincidence of A and B failures on the same test, and secondly dependence regarding the clustering of A and B failures on different tests. The two fail lines below show two SUM processes behaving independently: in this case coincident failure and clustering both occur randomly.

```
A   .   . .     .   .     .     . .
B .   . .     .      .  .     .
```

Notice that if the above failure pattern is assumed for the purposes of modelling, the probability of observing a long fail-free sequence of tests is lower than in the case of the earlier example of adjacent component failures. Therefore, given an observation of a fail-free sequence, the assumption of the independence failure pattern leads, statistically, to lower estimates for the component pfds. In this paper every fundamental component produces a SUM process. However, simply because two processes are individually SUM processes does not mean they are independent. The example of adjacent failures described above for Figure 3 illustrates this. A different example can easily be constructed to illustrate the same point, but based on coincident failures in which all fails of one process are coincident with fails of the other process (a special case being a proportion of 100%).

```
A   .   . .     .   .     .     . .
B   .     .         .             .
```

Now consider the SUM interaction of Figure 1. The first kind of dependence is possible – it is quite possible that more (or less) joint failures of A and B occur than would occur if the two SUM processes were independent. However, the second type of dependence, clustering on different tests, is prevented by the randomness of the demand selection procedure working over the system failure domain in the test space. Note that since the test selection mechanism is unchanging, the only phenomenon that could cause clustering is a varying system failure domain over time. In Figure 1 this phenomenon can not occur, but Figure 3 constructs a computational explanation for such a variation, and allowed an 'extreme' concrete example of the phenomenon.

Given SUM processes with on-demand fail probabilities θ_A and θ_B, over the range of all possible same-test dependencies, exclusivity produces the largest value of θ_S, the system probability of failure on demand. Similarly given θ_A and θ_B, and given any state of same-test dependence, extreme clustering (adjacent failures of A and B) produces the largest probability of zero system fails in N tests. In the usual statistical fashion, if we turn this statement around, extreme clustering is the assumption that produces the estimators that most favour large θ_A and θ_B when we observe no failures. Therefore if we observe no failures during testing and use both of these assumptions simultaneously, we effectively estimate the largest θ_A and θ_B possible, and then sum them to obtain the system fail probability. That is, from the point of view of system fail probability, we are making the worst case assumptions for the two types of dependence, both in isolation and in conjunction. Given the observed evidence, the system fail probability estimator that most favors large fail probabilities will be obtained.

Following this approach, the estimator for θ_S is given in (1) below.

(1) $$f_2(\Lambda \mid 0, N) = \begin{cases} (N+2)[(1-\frac{\Lambda}{2})^{N+1} - (1-\Lambda)^{N+1}]; & \Lambda \in [0,1] \\ (N+2)(1-\frac{\Lambda}{2})^{N+1}; & \Lambda \in (1,2] \end{cases}$$

Proof: See Appendix A.

Λ is not to be thought of as the system on-demand fail probability θ_S but simply a sum of two fail probabilities, $\theta_A + \theta_B$, which is a quantity bigger than θ_S. Λ ranges between 0 and 2. We can use (1) to specify λ and δ such that $P(\Lambda>\lambda) = \delta$, in which case the $P(\theta_S > \lambda) < \delta$, giving us the upper bound λ on fail probability that is the important value in decision making e.g. to achieve certification of safety-critical system.

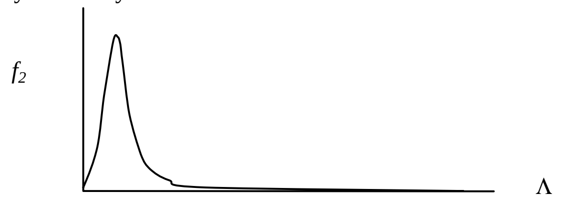

Figure 4 The form of *$f_2(\Lambda|0,N)$*

The formula in (1) has the form shown in Figure 4. The pointedness and left-skew of the distribution in Figure 4 increases with the number of tests N. The basic idea here is that if we test two SUM components in their role within the system, we will increase our confidence in the system. The type of interaction between the components influences the rate at which our confidence increases: on the same number of tests, (1) produces larger estimates of system fail probabilities, but applies to a wider range of interactions than the standard SUM analysis in section 2.1 (see section 4).

2.2.1 Analysis of connectivity

This section discusses the case where the connectivity is not assumed to be correct. It gives examples where f_2 in (1) remains the appropriate model for Figure 3.

Figure 3 covers many different specific mechanisms of computation. To take one example, if we take an object-oriented view and consider memory 'belonging' within the components, one possible configuration is that A is delivered data from B in a test, which A then maintains in its memory until it uses the data in the following test. The maintenance of memory in A does not make A a non-SUM component, the computation in A does not depend on computation by A in a previous test. It does depend on computation by B in a previous test and so has the potential to cause failure dependence with B. In this view, the connectivity is extremely simple, it simply transfers data on each test and involves no further computation, and in particular has no memory.

Denote testing of a SUM interaction between elements E1 and E2 by E1-E2, and testing of a non-SUM interaction as E1!E2. We first consider the idea of an interaction between a component B and connectivity Conn. The interaction between B and Conn in this case is a SUM interaction. This is because on a single test, the combined failure behaviour of B and Conn is fully determined by the system test inputs; there is no dependence on computation prior to the test. Therefore a minimal configuration is A! [B-Conn] i.e. the overall system can be analysed as a two-component non-SUM interaction, using the f_2 estimator in (1).

An alternative configuration is one in which memory is considered to lie outside components i.e. it resides in the connectivity. This view is suitable for the code in section 2.2. In this case, the connectivity can be considered in two parts, one each side of the memory. The system can be analysed as [A-ConnA]![ConnB-B]. Thus the model to analyse the system in this configuration is the same as before, as given by (1).

2.2.2 A 3-component system

A similar analysis for 3 SUM components in series, failing dependently in this way (exclusive, adjacent test failures) produces the formula in (2). Like f_2, it shows a linear combination of 'harmonics' of f_1. Since bounding the fail probability with values above 1 is never going to be of interest, it is unnecessary to calculate the formula over the range [1,3]. Over the range [0,3], f_3 is a probability density function (pdf), but the formula in (2) is not a pdf, due to its restricted range.

(2) $$f_3(\Lambda \mid 0, N) = \frac{(N+3)}{2}\left(3(1-\frac{\Lambda}{3})^{N+2} - 4(1-\frac{\Lambda}{2})^{N+2} + (1-\Lambda)^{N+2}\right) \quad \Lambda \in [0,1]$$

As N increases, (2) tends to $\frac{3(N+3)}{2}(1-\frac{\Lambda}{3})^{N+2}$. However, for large but practically achievable levels of N the probability of interest $\int_0^{\lambda} f_3(\nu)d\nu$ depends on the other two terms when λ and δ are small.

3. SYSTEMS ANALYSIS

CBSST analysis contains BBSST analysis as a special case when a system is behaving as a SUM process. As stated in section 2, in this paper, CBSST analysis of non-SUM system behaviour is performed by considering interactions between SUM-components. That is, whilst the system as a whole is not behaving as a SUM process during testing, its components at some level of decomposition are behaving as SUM-processes on the same tests. The cases studied in this paper are simple, and further work is needed to generalise the results, but the general idea is to identify a system decomposition that distinguishes the SUM and non-

SUM interactions between the SUM components and apply the analysis correspondingly.

The example in section 2.2 illustrates the idea. By separating out a certain data flow and two components, we effectively create three elements that could fail, all requiring testing (or some other verification) to assure their reliability. The bounding calculations in section 2.2 give on-demand failure probability estimates for use in cases where the SUM does not apply to the software system. That is, the new analysis relaxes the requirement that the software tests form a random sample with respect to overall software failures.

3.1 What systems/components are SUM?

Systems or components that avoid memory at some level of abstraction, such as combinatoric electronic circuits, naturally produce SUM processes. A component containing explicit memory can also be SUM if the memory effects can be encapsulated within tests by choosing tests appropriately i.e. if tests can be found such that state does not 'escape' from one test to the next. Such a component may have extremely complex memory usage and state evolution within tests.

Arbitrarily complex software components containing memory, simple conditional branching, iteration, recursion etc. can be connected to build a SUM system if under some system testing: 1. they are individually SUM, and 2. they are connected together using connectivity that only causes SUM interactions. SUM interactions can be recognized by simple checks on the code - if the connectivity between components delivers data that is derived wholly from data input in the current test, the interaction is SUM. In the case where data is delivered that is not derived solely from the current test, the interaction is SUM if that data is the same on every test, otherwise the interaction must be assumed to be non-SUM.

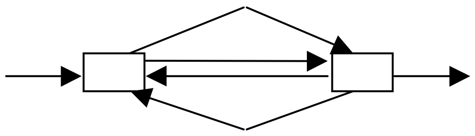

Figure 5 Multiple data flows

For example, the components in Figure 1 need not be simple code blocks executing in sequence (the 'function' view of components used in this paper). A and B could be objects, with multiple bi-directional data flows occurring in a single test due to method calling - see Figure 5. The composition result of section 2.1 still applies.

3.2 Analysis of systems built by connecting together Commercial Off-The-Shelf (COTS) components

3.2.1 Case 1: System testing of SUM interaction

Suppose two COTS components are both SUM under previous testing of the functions required of the components in a new application. Suppose in the new application they are connected together using connectivity that causes SUM interactions. Then the system will be SUM, and can be analysed using BBSST.

3.2.2 Case 2: System testing of non-SUM interaction

In section 2.2, A and B could be COTS components and despite the simplicity of the system, the BBSST cannot be used to estimate its on-demand fail probability. However, it is possible to estimate the system fail probabilities using the CBSST model in (2) to analyze system test results.

3.2.3 Case 3: Re-use of component reliabilities using a collaboration of CBSST and formal methods

Suppose two SUM COTS components A and B are connected in a configuration with non-branching execution, and with connectivity causing only SUM interactions. It would be possible to formally prove that the connectivity achieves the system requirements given the components implement their specifications. System testing could then be applied, and the system analyzed using the BBSST model, since the components compose as in section 2.1. However, the formal proof is pointless in this case since the whole system including connectivity composes to produce a SUM process, and can be analyzed with exactly the same statistical test efficiency (using BBSST) without performing the formal proof.

Now consider the same case but where there is previous individual execution history for each of the two COTS components. 'Previous execution history' of a component could be the results of testing performed by the manufacturer, or observed operational experience of the component in other systems. If N of A's previous tests are consistent with A's operational profile in the new system, and if N of B's previous tests are likewise consistent with B's operational profile in the new system, then the system pfd can be estimated conservatively using model f_2 with N tests. The two respective component test sets are obtained by normalizing the frequencies of tests performed in the old test environment to the operational profiles of the components in the new application, using a technique explained elsewhere [11]. The historical tests place constraints on the likely failure patterns of the individual components in the new system. Then f_2 obtains a constraint on the system failure pattern by analysing the worst possible way in which the component patterns could combine (worst in the sense that a higher system fail probability could produce the combined pattern).

The result is 'reuse of component reliabilities,' and generalises to any number of components. Furthermore, it also applies to non-SUM interactions i.e. any non-branching connectivity can be used to connect the components together. Therefore, systems developed in this way from pre-built components that have been previously statistically tested, can have a reliability associated with them based purely on the previous execution history of that system's components. The technique requires the formal proof, a log of the previous component tests/operation and knowledge of the components operational profiles in the new system. In fact, no system would be deployed without system testing, and this may be the only practical way to obtain the components' new operational profiles (through observation, using code instrumentation).

In the case where the interaction between A and B is a SUM interaction, the above analysis is grossly conservative. There is potential for improving the accuracy of the bounding analysis, since extreme clustering on different tests (as assumed by f_2) is impossible.

Various permutations of this approach are conceivable. The formal proof of connectivity simplifies the discussion, but models for reuse of component reliabilities omitting this proof will be studied in future work. In this case, the component interaction (i.e. the system as a whole) will require its own new testing since, unlike the components, it is unlikely any previous execution history will be available. Similarly, a full proof based approach would be possible in which proved components are connected together and the resulting connectivity tested. The latter is a risky approach for recently developed components. Components may be large and require large correctness proofs. Moreover, except by experience of actually attempting re-use of a component, it is seldom clear whether a component will be suited to re-use. Until a component has been shown to be highly re-usable, the effort involved in proving it might be regarded as unjustifiable. In addition, COTS source code is often withheld, rendering formal proof impossible.

3.2.4 Case 4: Non-SUM components

A different and more difficult scenario is where it proves impractical to find demands for which the COTS components are SUM. CBSST can still provide a solution, although it requires extra effort on the part of component manufacturers. In addition to selling components, the manufacturers would have to sell a model of the code structure. It would be necessary for them to provide a bespoke 'SUM-boxes' CBSST failure model of a component, relative to the demand space faced by the component. The model would be in the form of SUM-box diagrams with associated test sets. That is, a decomposition of the component to SUM sub-components, and details of the various demand classes placed on the component together with the type of interactions that occur for each demand class.

Providing this level of information would allow analysis of the COTS components in their new environment (the new system in which they are placed). It also stops far short of disclosing actual code, a feature that the manufacturers could use to protect their intellectual property.

4. SOFTWARE TESTABILITY

An important result in the field of statistical testing is the number of fail-free tests required to demonstrate a system failure probability of less than 10^{-3} at confidence 99%. The number is well known for BBSST:

- A system behaving as a SUM process requires approx. 4600 tests (using model f_1).

This can be compared against the two following new results:

- A system built from two SUM components with non-SUM interaction requires approx. 9200 tests (this follows from model f_2).
- A system built from three SUM components with non-SUM interaction requires approx. 18300 tests (model f_3).

The CBSST bounding calculations are conservative. Use of the bounding calculation exacts a penalty in terms of requiring a greater number of tests to achieve the same level of estimated reliability. Therefore, system designs that can be tested with SUM behaviour, or '*close to* SUM' behaviour (meaning that only a few non-SUM interactions are present), are of interest since they have a higher test efficiency - their reliability can be demonstrated with fewer tests.

5. CONCLUSIONS

The study of behavior in complex man-made systems is an unusual application of statistics. The deterministic nature of the software interferes with the requirement of the statistics to study random phenomena. However, if it is to become widely used, statistical testing must accommodate the new component-based approach to software development, and this requires the juxtaposition of determinism and randomness to be modeled. This paper develops CBSST modeling, achieving a more sophisticated understanding of this relationship by identifying the code mechanisms that determine the statistical properties of software failure behavior.

CBSST has attractive advantages for software quality analysis. The re-use of test results is perhaps the most commercially appealing. There are also benefits that have not been discussed in this paper, such as the ability to associate Bayesian priors with components, as discussed elsewhere [4]. New conclusions from this paper are listed below.

- New failure probability estimation models have been derived. These are capable of modeling dependent component failure, a major obstacle that has prevented the derivation of convincing component-based software reliability models to date. The new models include BBSST as a special case.
- A commonly quoted stumbling block for testing as a V&V method is the vast number of states a program can enter. However, the power of randomness as a search mechanism is that the size of a population is not necessarily an impediment to statistical analysis. In the new statistical testing models it is not large state spaces that determine testability, it is the number of 'difficult' (non-SUM) interactions that the program exhibits under a statistical test regime.[3]
- A promising approach to software V&V combining formal methods and statistical testing has been suggested. It exploits the strengths of both techniques. This test/proof combination approach is consistent with the current trend to use formal methods for selective proof objectives, rather than full proof of code – thus long proofs at the detail of low level code are avoided. The approach is particularly suitable for software constructed from commercial-off-the-shelf (COTS) software components, avoiding the difficulties associated with a full proof-of-code approach.
- The models show how component test results can be re-used, and examples have been given of how this re-use can provide failure probability estimation for software systems built from COTS components.
- The models suggest commercial practices in which component manufacturers sell quality models alongside the components themselves.
- The new models suggest statistical testability as a new system complexity metric. Low complexity systems are defined as those that are decomposable to a small number of components (i.e. that are 'close to SUM behaviour') for the purposes of the statistical test models. The CBSST models

[3] There is an implicit assumption that the program under test cannot be exhaustively tested.

state that such low complexity software requires fewer tests to justify a given level of reliability. Software that is highly testable in this sense, using tests of a reasonable length, is therefore desirable. This may have important implications for the design of reliable software. Just as some software design solutions have desirable low *computational* complexity compared to others implementing the same requirements, so some designs will have higher testability. It would be interesting to see how these two properties relate – can low computational complexity and high testability coexist in software code?

- Research in CBSST is at a very early stage of development, and its practicality is currently speculative. This paper has attempted to identify open research questions that will prove important for the successful future application of the technique.

6. ACKNOWLEDGEMENTS

This work presented in this paper comprises aspects of studies (DDT project) performed as part of the UK Nuclear Safety Research programme, funded and controlled by the Industry Management Committee.

7. REFERENCES

[1] Hughes G, May JHR, Lunn AD "Reliability estimation from appropriate testing of plant protection software", IEE Software Engineering Journal, v10 n6, November 1995

[2] Goseva-Popstojanova K, & Trivedi K.S. "Architecture-based approach to reliability assessment of software systems" Performance Evaluation, v45 n2-3, July 2001

[3] Kanoun K, Laprie JC, Thevenod-Fosse P "Software reliability: state-of-the-art and perspectives" LAAS Report 95205, May 1995. http://www.laas.fr/laasve/

[4] Kuball S., May J.H.R. & Hughes G. "Structural software reliability estimation," Lecture Notes in Computer Science v1698 'Computer Safety, Reliability and Security' Felici, Kanoun, Pasquini (Eds) pp336-349 ISBN 3540664882 Springer 1999

[5] Kuball S., May J.H.R. & Hughes G. "Building a System Failure Rate Estimator by Identifying Component Failure Rates" Procs. Of the 10th Int. Symposium on Software Reliability Engineering (ISSRE'99, Boca Raton, Florida, Nov 1-4, 1999) pp32-41 IEEE Computer Society 1999

[6] Kuball S., May J.H.R. & Hughes G. "Software Reliability Assessment for Branching Structures: A Hierarchical Approach," 2nd Int. Conference on Mathematical Methods in Reliability, vol 2, Bordeaux, July 4-7 2000

[7] Lyu MR "Handbook of Software Reliability Engineering," McGraw-Hill 1996

[8] May JHR, Lunn AD "A Model of Code Sharing for Estimating Software Failure on Demand Probabilities" IEEE Trans. on Software Engineering SE-21 (9) 1995

[9] May JHR and Lunn AD "New Statistics for Demand-Based Software Testing" Information Processing Letters 53, 1995

[10] May JHR, Kuball S, Hughes G "Test Statistics for System Design Failure" International Journal of Reliability, Quality and Safety Engineering, v6 n3 1999 pp. 249-264

[11] Miller WM , Morell LJ, Noonan RE, Park SK, Nicol DM, Murrill BW and Voas JM "Estimating the probability of failure when testing reveals no failures" IEEE Trans. on Software Engineering v18 n1 1992

[12] Musa JD "Operational profiles in software reliability engineering," IEEE Software 10(2) 1993

[13] Thevenod-Fosse P, "On the efficiency of statistical testing with respect to software structural test criteria" IFIP Working Conference on Approving Software Products, Garmisch (Germany), 17-19 Sept 1990, pp.29-42. Also LAAS Report 90156, May 1990

[14] Zhu H, Hall PAV, May JHR "Software Unit Test Coverage and Adequacy," ACM Computing Surveys 1997

8. APPENDIX A

The proof below is for 2 components. The proof of the result in section 2.2.2, for 3 components, follows the same procedure. The manipulation of the limits of integration is more involved in the 3-component case. The N-component problem remains open.

For a given θ_1 and θ_2, the case of exclusive failures showing extreme clustering, (3) describes the probability of a failure-free sequence of N tests when N is very large and dwarfs the number of components (which in this case is 2). This can be seen from inspection of the fail-line in section 2.2, and noting that $f_1(0 \mid \theta_i, N) = (1-\theta_i)^N$ where 0 denotes zero failures.

$$P(0 \mid \theta_1, \theta_2, N) = MIN\{(1-\theta_1)^N, (1-\theta_2)^N\} \tag{3}$$

An expression for $f_2(\Lambda \mid 0, N)$ is shown in (4).

$$f_2(\Lambda \mid 0, N) = \int_{\Lambda=\theta_1+\theta_2} h(\theta_1, \theta_2 \mid 0, N)\, d\theta_1 d\theta_2 \tag{4}$$

which can be evaluated using (5).

$$h(\theta_1, \theta_2 \mid 0, N) = \frac{P(0 \mid \theta_1, \theta_2, N)\, g(\theta_1, \theta_2 \mid N)}{P(0 \mid N)} \tag{5}$$

where g is a prior for (θ_1,θ_2) and $P(0 \mid N) = \int_0^1\int_0^1 P(0 \mid \theta_1, \theta_2, N)\, g(\theta_1, \theta_2 \mid N)\, d\theta_1 d\theta_2$.

Using $g(\theta_1, \theta_2 \mid N) = 1$ to express no prior preference on the location of (θ_1,θ_2) within $[0,1]^2$, it follows that $h(\theta_1, \theta_2 \mid 0, N) = k.MIN\{(1-\theta_1)^N, (1-\theta_2)^N\}$ where $\frac{1}{k} = \int_0^1\int_0^1 MIN\{(1-\theta_1)^N, (1-\theta_2)^N\}\, d\theta_1 d\theta_2$. This gives (6).

$$h(\theta_1, \theta_2 \mid 0, N) = \frac{(N+1)(N+2)}{2} MIN\{(1-\theta_1)^N, (1-\theta_2)^N\} \tag{6}$$

Substituting $\theta_2 = \Lambda - \theta_1$, (4) becomes (7).

$$f_2(\Lambda \mid 0, N) = \begin{cases} \int_0^{\Lambda} h(\theta_1, \Lambda-\theta_1 \mid 0, N)\, d\theta_1 & ;\Lambda \in [0,1] \\ \int_{\Lambda-1}^{1} h(\theta_1, \Lambda-\theta_1 \mid 0, N)\, d\theta_1 & ;\Lambda \in (1,2] \end{cases} \tag{7}$$

Which can be rewritten as (8).

(8)
$$f_2(\Lambda\,|\,0,N)=\begin{cases}\dfrac{(N+1)(N+2)}{2}\left(\displaystyle\int_0^{\frac{\Lambda}{2}}(1-\Lambda+\theta_1)^N d\theta_1+\int_{\frac{\Lambda}{2}}^{\Lambda}(1-\theta_1)^N d\theta_1\right) & ;\Lambda\in[0,1]\\ \dfrac{(N+1)(N+2)}{2}\left(\displaystyle\int_{\Lambda-1}^{\frac{\Lambda}{2}}(1-\Lambda+\theta_1)^N d\theta_1+\int_{\frac{\Lambda}{2}}^{1}(1-\theta_1)^N d\theta_1\right) & ;\Lambda\in(1,2]\end{cases}$$

Which evaluates to (9).

(9)
$$f_2(\Lambda\,|\,0,N)=\begin{cases}(N+2)[(1-\dfrac{\Lambda}{2})^{N+1}-(1-\Lambda)^{N+1}]; & \Lambda\in[0,1]\\ (N+2)(1-\dfrac{\Lambda}{2})^{N+1}; & \Lambda\in(1,2]\end{cases}$$

This can be written as (10).

(10)
$$f_2(\Lambda\,|\,0,N)=f_1(\frac{\Lambda}{2}\,|\,0,N+1)-f_1(\Lambda\,|\,0,N+1) \qquad ;\Lambda\in[0,2]$$