On the Estimation of Reliability of a Software System Using Reliabilities of its Components

Saileshwar Krishnamurthy*
Aditya P. Mathur†
Software Engineering Research Center
1398 Department of Computer Sciences
Purdue University, W. Lafayette, IN 47907, USA

April 21, 1997

Abstract

We report an experiment to evaluate a method, known as *Component Based Reliability Estimation* (CBRE), for the estimation of reliability of a software system using reliabilities of its components. CBRE involves computing path reliability estimates based on the sequence of components executed for each test input. Path reliability estimates are averaged over all test runs to obtain an estimate of the system reliability. In the experiment reported, three components of a Unix utility were seeded with errors and the reliability of each component was measured. The faulty components were then introduced systematically into the utility, in various combinations, to produce several faulty versions of the utility. For each faulty version, test cases were drawn from an operational profile to measure the "component-based reliability". The "true reliability" of the faulty version was estimated using the frequency count approach. The goodness of CBRE was assessed in terms of the accuracy and efficiency of the estimates with respect to the "true reliability." Results from this experiment suggest that CBRE yields reasonably accurate results at an efficient rate. However, the accuracy and efficiency of CBRE is sensitive to the dependency among successive calls to a component.

Index terms:: Components and reliability, software testing, software reliability.

*Saileshwar Krishnamurthy's research was supported by an award from the Center for Advanced Studies, IBM Toronto Laboratories. email: krish@cs.purdue.edu.

†Aditya P. Mathur's research was supported in part by an award from the Center for Advanced Studies, IBM Toronto Laboratories and NSF award CCR-9102331. All correspondence regarding this report may be sent to Aditya P. Mathur. email: apm@cs.purdue.edu

Contents

List of Figures

List of Tables

1 Introduction

Reliability of a software system is its probability of failure free operation in a given environment. Several approaches have been proposed to measure reliability [10]. As argued by Horgan and Mathur [7, 8], these approaches are not entirely satisfactory for use in practical software development environments. The goal of research reported here is to experimentally investigate a method for estimating the reliability of a system given the reliabilities of its components and of their interfaces with other components. This method is termed "Component-Based Reliability Estimation" and is referred to as CBRE. Considering the variety of programs[1] and programming languages to which CBRE can be applied, the term "component" defies a general purpose formal definition. In this report a component is viewed as any collection of program functions. An example given later in Section 3 illustrates this definition. The interface between two components could be another component, a collection of global variables, a collection of parameters, a set of files, or any combination of these.

CBRE relies on using the sequence of components executed during system or subsystem testing. In this sense it is different from the approach reported by Laprie and Kanoun [9] which models a system using the Markov chains and computes system reliability using a knowledge of component interconnections, their failure rates, inter-component transition probabilities, and other statistical information. A comparison of our approach with the one proposed by Laprie and Kanoun has not been attempted in this report.

CBRE appears equally applicable to the estimation of the reliability of complete software systems and of their subsystems. For example, the experiment reported here used a Unix utility as a system. The same utility may also be considered as a subsystem of another system.

The two primary needs that led us to propose and experiment with CBRE are given in the following.

1. The need to understand how the system reliability depends on its component reliabilities and their interconnections.

2. The need to make reliability estimates early in the system development cycle.

The first of the above needs arises from a desire to know which system components need relatively thorough testing. Considering that resources are limited, development managers need reliable metrics to help apportion testing effort. CBRE allows one to conduct sensitivity analysis to meet this need. However, in the experiment reported in this report, no sensitivity analysis was performed. How one may go about performing such analysis is discussed in Section 6.

The second of the above needs arises from a desire to know early enough the quality of the components completed so far. Resorting to reliability estimation at the end of the development leaves little room for redoing, partly or fully, components that might be responsible for unacceptable system reliability.

[1] We use the words "program" and "system" interchangeably.

The remainder of this report is organized as follows. Section 2 explains the CBRE approach with an example. In Section 3 we describe the design of our experiment. The results appear in Section 4. In Section 5 we present a brief analysis of the results. A discussion on various aspects of CBRE appears in Section 6.

2 Component Based Reliability Estimation

CBRE is applied in three steps. In the first step one identifies the components and their interfaces in the system under consideration. In the second step one estimates the reliabilities of components and their interfaces. Dependencies amongst components are also identified in this step. In the third step the system reliability is computed. The first two steps need not be applied in that order. Identification of components, interfaces and their reliabilities can be done when the components become available. The experiment reported is concerned only with the third step. The first two steps are discussed by Garg [5]. However, we are not aware of any evaluation so far of the entire CBRE approach.

Below we illustrate the third step of CBRE which is the estimation of system reliability given component reliabilities. Interface reliabilities are assumed to be 1 in this experiment. We begin with the definition of an operational profile.

Definition 1 *Let I denote the input domain of a program such that $I = \bigcup_{1 \leq i \leq n} SD_i$, where SD_i is a subdomain of $\mathcal{I}$. An "operational profile" is a mapping from the set of subdomains $\{SD_1, SD_2 \ldots SD_n\}$ to probabilities; the sum of these n probabilities is 1.*

In the definitions below, it is assumed that the system under consideration is tested on inputs generated in accordance with a given operational profile. However, as discussed in Section 6, CBRE can also be applied when the operational profile is not available.

Definition 2 *The "true reliability" of a program P, which could be a component or a system, is the probability of successful operation of P on any input selected from its input domain.*

In adapting this method to software systems, we consider the components executed for each test case. A sequence of components along a path traversed for test case t can be considered as a series system that is executed for a subset of test cases of which t is one element. The CBRE approach is based on computing a "reliability estimate" for each path. If the reliability of individual components is assumed to be independent of each other, known as the *independence assumption*, then the path reliability is the product of component reliabilities given that the interface reliabilities are assumed 1.

An interesting case occurs when at least one component, say component C, along a path is invoked from inside a loop several times during one execution. If the independence assumption holds, the path reliability will tend to 0 with the increase in the number of times C is executed. If most paths executed have components within loops and that these loops are traversed a sufficiently large number of times, individual path reliabilities are likely to be low resulting in

low system reliability estimates despite the chance that the system reliability is higher than estimated. This leads to the problem of modeling inter- and intra-component dependencies. This problem has been addressed in the experiment described here.

Definition 3 *The component trace of a program P for a given test case t is the sequence of components executed when P is executed against t. Such a trace is denoted by $M(P,t)$.*

Definition 4 *The "component-based reliability" estimate of a program P with respect to a test set T, is given by:*

$$R_c = \frac{\sum_{\forall t \in T} R_c^t}{|T|}$$

where the reliability of the path in P traversed when P is executed on test-case $t \in T$ is given by:

$$R_c^t = \prod_{\forall m \in M(P,t)} R_m$$

Definition 5 *Let R_{tf} be the true reliability of P measured over the test set T. The* relative error ϵ *in estimates of R_c is defined as:*

$$\epsilon = \frac{|R_c - R_{tf}|}{R_{tf}}$$

Definition 6 *Let N_c be the number of executions of P used to arrive at R_c. The δ-efficiency of the estimate is denoted by η_δ, where $\epsilon \leq \delta$. The δ-efficiency is computed as:*

$$\eta_\delta = 1 - \frac{N_c}{N_{tf}}$$

Efficiency as defined above can be negative when $N_c < N_{tf}$ which implies that CBRE is not an appropriate method to use for the system under consideration for the given δ. We use η_δ as a measure of how fast R_c will converge as compared to the convergence of R_{tf}. It is important to note that CBRE does not require that R_{tf} be computed for a given software system. R_{tf} was computed in the experiments reported here to evaluate the goodness of CBRE.

2.1 Example

Consider a hypothetical system with component-call graph as in Figure 1. The component reliabilities for this system are listed in Table 1. The reliabilities vary from 0.6 to 0.9. Data in Table 2 shows four test cases and reliability estimates for the system. The first test case results in the component trace 1,2,4. The reliability of this path is $R_1 \times R_2 \times R_4 = 0.9 \times 0.6 \times 0.9 = 0.49$. The third test case resulted in the component trace 1,3,2,3,2,3,4. The reliability for this path is $R_1 \times {R_2}^2 \times {R_3}^3 \times R_4 = 0.9 \times 0.6^2 \times 0.8^3 \times 0.9 = 0.15$. The system reliability estimate of 0.42 is obtained by averaging over the path reliabilities.

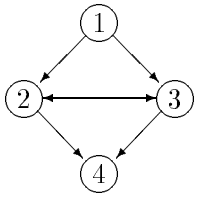

Figure 1: Example function call graph of a system with 4 components

Table 1: Sample reliabilities for the components in the example.

Component	Reliability
1	0.9
2	0.6
3	0.8
4	0.9

2.2 Component dependence

Given two components C_1 and C_2 of a software system we say that C_2 depends on C_1 if the probability of failure, i.e. the reliability of C_2 is effected in any way by the execution of C_1. If C_1 and C_2 are different components then this dependence is known as inter-component dependence, else it is known as intra-component dependence. Inter-component dependence can arise, for example, due to an error in the design of the interface between C_1 and C_2. Thus, when C_2 is tested in an environment in which C_1 is not present, it might not fail at all. However, when tested in the presence of C_1 it might fail due to an error in interfacing the two components. Intra-component dependence can arise, for example, when a component C is invoked more than once in a loop by another component C' and successive calls to C make use of the state of C'.

Computing path reliability as the product of the reliabilities of the components executed on

Table 2: Sample test cases and reliability for the program in the example.

Test case	Component traces	Reliability
1	1,2,4	0.49
2	1,3,4	0.65
3	1,3,2,3,2,3,4	0.15
4	1,2,3,4	0.39
System reliability		0.42

Table 3: Sample test cases for the example program.

Test case	Component traces	Reliability		
		DOI = 1	DOI = 2	DOI = 3
1	1,2,4	0.49	0.49	0.49
2	1,3,4	0.65	0.65	0.65
3	1,3,2,3,2,3,4	0.39	0.19	0.15
4	1,2,3,4	0.39	0.39	0.39
System reliability		0.48	0.43	0.42

this path, relies on the assumption that these components are independent of each other. This may not be true in software systems since components are often tightly coupled. Specifically, when a component is executed in a loop, it may be the case that multiple occurrences of this component are not independent.

There are several ways of modeling both inter- and intra component dependence. In the experiment reported we assumed that components were independent, i.e. there is no inter-component dependency. Intra-component dependency was modeled by "collapsing" multiple occurrences of a component on an execution path into a single call. The assumption here is that all occurrences of a component on an execution path are *dependent*. An alternative approach could "collapse" multiple occurrences of a component into k occurrences, where $k > 0$. If any path contains l copies of a component, and $l < k$, then no further "collapsing" need be done. k is referred to as the degree of independence and abbreviated as DOI. Complete dependence is modeled by setting DOI to 1 and complete independence by setting DOI=∞.

The effect of DOI on reliability estimates is illustrated using the example in Section 2. Table 3 lists four sample test cases for the example program. The table also lists the component-based reliability for degree of independence (DOI) values 1, 2 and 3. The third test case leads to a component trace of $\{1,3,2,3,2,3,4\}$. The reliability for this path assuming different values of DOI are given below.

- (with DOI = 1) $R_1 \times R_2 \times R_3 \times R_4 = 0.9 \times 0.6 \times 0.8 \times 0.9 = 0.39$
- (with DOI = 2) $R_1 \times {R_2}^2 \times {R_3}^2 \times R_4 = 0.9 \times 0.6^2 \times 0.8^2 \times 0.9 = 0.19$
- (with DOI = 3) $R_1 \times {R_2}^2 \times {R_3}^3 \times R_4 = 0.9 \times 0.6^3 \times 0.8^3 \times 0.9 = 0.15$

Larger values of DOI lead to smaller estimates of path, and hence system, reliabilities. As the example here is hypothetical and we do not know what is the "true reliability" of the program, we cannot determine which value of DOI is the most appropriate for this example. We discuss this issue again in the following section with respect to the program used in the our experiment.

Dependence modeling in our experiment is exploratory in nature. We decided to explore how well a given dependence model will work. As discussed in Section 6, perhaps a more scientific way would be to use the system architecture and source code to determine the nature of dependence. Also, we decided to consider distinct components on a path to be independent of each other simply because inter-component dependence did not appear to be the prime cause of poor accuracy in our experiment.

3 Design of experiment

The experiment reported here was aimed at investigating the accuracy and efficiency of CBRE. More precisely, the objectives of the experiment are:

1. To study the accuracy of R_c (component-based reliability) estimates of a system with respect to its R_t (true reliability) observed over a test set T. Definition 5 formalizes the notion of accuracy.

2. To study the efficiency of R_c estimates of a system with respect to its R_{tf} estimates over a test set T. Definition 6 formalizes the notion of efficiency.

3.1 Terminology

1. S: Program under test; `grep` is used in the experiment.

2. OP: Operational profile for S - constrains the generation of random inputs sampled from the input domain.

3. N: Number of components in S

4. N_m: Number of components seeded with errors.

5. N_f: Number of faulty versions of S.

6. N_{fm}: Number of faulty components in a faulty version of S.

7. M_j: j^{th} faulty component in the system; components are arbitrarily assigned distinct numbers.

8. $|M_j|$: Number of executable lines of code in M_j

9. R_j: Reliability of M_j.

10. FD: Fault density, or the number of errors seeded per executable line of code.

11. E_i: Set of errors seeded in component M_i.

12. E: Set of errors seeded in the program S, $E = \bigcup_{1 \leq i \leq n} E_i$.

13. α: Confidence level required for convergence of reliability. We uses a 99% confidence level and hence α is 0.99.

14. δ_r: Half-width of the confidence interval; we use 0.05.

15. $z_{\alpha/2}$: $Prob(|Z < Z_{\alpha/2}|) \leq 1-\alpha/2$, where Z is a random variable representing the Standard Normal Distribution.

16. *Output*: The output of a program S when executed with an input t is denoted by $S(t)$.

3.2 Assumptions

- N_m components of the program under test are seeded with errors such that

$$\forall i, 1 \leq i \leq N_m, \frac{|E_i|}{|M_i|} = FD$$

.

- The reliability of components not seeded with errors is assumed to be 1.

- The reliability of all component interfaces is assumed to be 1.

3.3 Experimental setup

The GNU version of `grep` was used in the experiment. The `grep` utility consists of about 12,000 lines of C code. The experiment was performed on a Sun Sparc 5 running under the Solaris 2.5.1 operating system. Components for seeding faults were chosen based on their complexities measured in the number of basic blocks and their frequency of occurrence measured using the Unix utility `prof`. The standard regression test suite of 123 test cases that is supplied with the GNU version of `grep` was used for profiling. The eight components of `grep` that we examined are listed in Table 4. The four relatively low complexity components are unsuitable for seeding faults at the density levels required. Three components out of the remaining four were chosen and seeded with faults as described in Table 5. Faulty versions of `grep` were constructed in three ways as described in Table 6.

3.4 Methodology

1. Construct an operational profile for `grep`. We used the operational profile used earlier by Garg[5].

2. Measure the reliability of each component M_i in S.

   - For each component M_i in S, generate a faulty program S_i by seeding S with errors in E_i.
   - Instrument S_i such that during each run whenever component M_i is invoked, its Component ID (CID) is written to a file.

Table 4: Complexities and frequency of occurrence of modules in `grep`.

Module	Complexity (No. of blocks)	Frequency (No. of calls)
`_obstack_begin`	12	86
`alloca`	16	13
`regex_compile`	991	123
`_getopt_internal`	129	123
`reset`	14	650
`kwsmusts`	26	117
`lex`	267	856
`dfaanalyze`	110	117

Table 5: List of components in `grep` seeded with faults. Fault density=0.01.

Component	Number of Lines	Number of Faults
`lex`	312	4
`dfaanalyze`	215	3
`regex_compile`	954	10

Table 6: Faulty versions of `grep`.

Faulty components	Total Lines	Total Faults	Fault Density $\frac{Faults}{Lines}$
`lex`	13029	4	0.0003
`lex`, `dfaanalyze`	13047	7	0.0005
`dfaanalyze`, `regex_compile`	13063	13	0.001

- Use the Algorithm below to generate reliability estimate for each component M_i as follows:

  $R_i \leftarrow$ `Component-Reliability` (S_i, S, OP, CID)

Algorithm
Purpose To measure the reliability of a component.
Inputs S', S, OP, CID.
Output Reliability of the component having component identifier CID.
Method

(a) Set `current_reliability` to 0.

(b) Set `total_executions` and `failures_observed` to 0.

(c) Sample a test case t randomly from the input domain of S constrained by OP.

(d) Execute S' and S on t.

(e)
- If the execution resulted in CID being written by S', then increment `total_executions` by 1, else repeat from Step 2c.
- If $S(t) \neq S_i(t)$, increment `failures_observed` by 1.

(f)
- Compute `current_reliability` as $(1 - \frac{\texttt{failures\_observed}}{\texttt{total\_executions}})$
- Build an *alpha* confidence interval centered at `current_reliability`. The variance is $\sigma = p \times (1 - p)$, where p is `current_reliability`, and the number of data points n is `total_executions`
- Compute the half-width of the confidence interval as: $z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}}$.
- If the half-width is less than δ_r, then reliability has not yet converged, repeat from Step 2c.

(g) Return `current_reliability`

End Method

3. Measure the reliabilities of each of the N_f faulty versions of S.

- For each faulty version construct a program S' such that each execution of S' results in a " component trace" consisting of the number of times each faulty component is executed.
- Use the Algorithm below to generate a set of "component traces", denoted by MT, and estimate R_t as:

  (R_t,MT) $\leftarrow$ `True-Reliability`(S',S,OP,N_{fm})
- The CBRE estimate R_c of S' is computed, using Definition 4, from the component traces obtained in the previous step.

Algorithm

Purpose To measure the reliability of S' and generate "collapsed component traces" for each execution.

Inputs S', S, OP, N_{fm}.

Outputs Reliability of S' and component traces.

Method

(a) Set `current_reliability` to 0.

(b) Set `total_executions` and `failures_observed` to 0.

(c) Set $Modrel \leftarrow \emptyset$.

(d) Randomly select a test case t from the input domain of S constrained by OP.

(e) Execute S and S' on t.

(f)
- Collapse the generated component trace so that no component is repeated.
- Append the collapsed component trace to $Modrel$.
- Increment `total_executions` by 1.
- If $S(t) \neq S'(t)$, increment `failures_observed` by 1.

(g)
- Compute `current_reliability` as $(1 - \frac{\texttt{failures\_observed}}{\texttt{total\_executions}})$
- Build an *alpha* confidence interval centered at `current_reliability`. The variance is $\sigma = p \times (1 - p)$, where p is `current_reliability`, and the number of data points n is `total_executions`
- Compute the half-width of the confidence interval as: $z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}}$
- If the half-width is less than δ_r, then reliability has not yet converged, repeat from Step 3d.

(h) Return (`current_reliability`, $Modrel$).

End method

4 Results

Using the component traces collected as described in the previous section, reliability estimates were obtained for the selected components and for the faulty versions of `grep`. Below we present the results of the experiments described in Section 3 in tabular and graphical form.

Reliabilities of faulty components are listed in Table 7. Reliabilities of `lex`, `dfaanalyze`, and `regex_compile` are, respectively, 0.649, 0.334 and 0.343. The CBRE estimates with DOI=1, their accuracy and efficiency are shown in Table 8. For faulty version 1 of `grep`the true reliability R_t is 0.688 and the component-based reliability R_c is 0.676. For faulty version 2 of `grep`R_t is 0.385 and R_c is 0.513. For faulty version 3 of `grep` R_t is 0.392 and R_c is 0.393.

Figure 2 plots the true and component-based reliabilities over executions for faulty version 1 of `grep`, with faulty component `lex`. This relationship is plotted in Figure 3 for faulty version 2 (with faulty components `lex` and `dfaanalyze`) and in Figure 4 for faulty version 3 which contains the faulty components `dfaanalyze` and `regex_compile`.

Table 7: Reliabilities of selected components in `grep`.

Component	Reliability
`lex`	0.649
`dfaanalyze`	0.334
`regex_compile`	0.343

Table 8: Dependent component based reliability estimates.

Faulty Version	Faulty Components	Reliability		Error(ϵ)	Efficiency	
		R_t	R_c		$\delta = 0.05$	$\delta = 0.01$
1	`lex`	0.688	0.676	1.78 %	0.97	0.93
2	`lex`, `dfaanalyze`	0.385	0.513	33.25 %	0.99	0.98
3	`dfaanalyze`,`regex_compile`	0.392	0.393	0.25 %	0.97	0.97

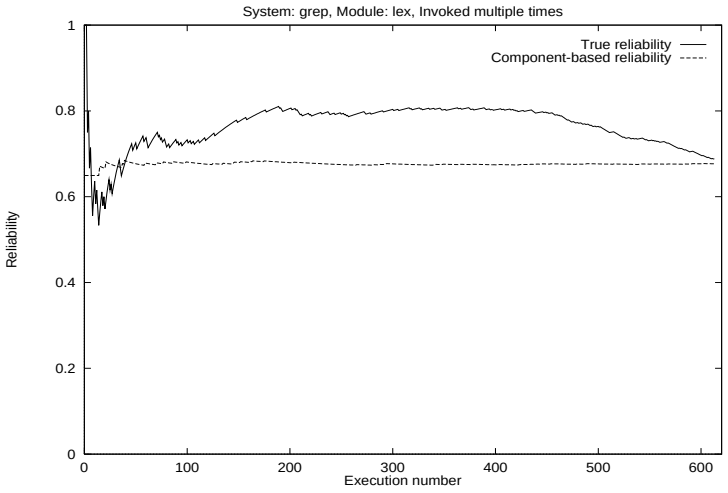

Figure 2: True reliability and component-based reliability for `grep` with faulty component `lex`.

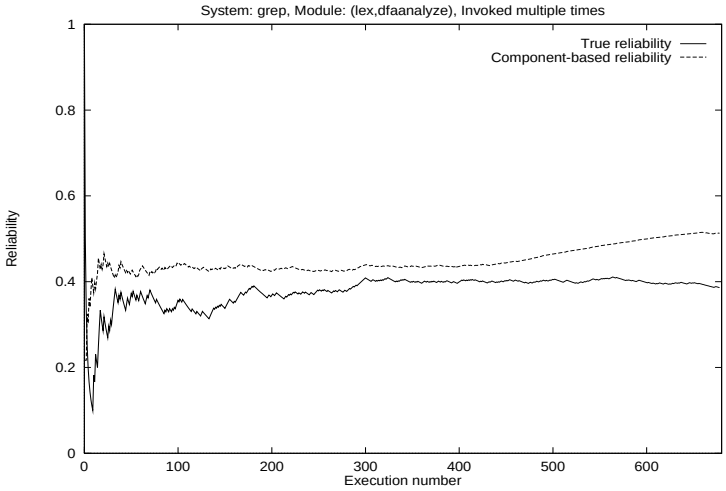

Figure 3: True reliability and component-based reliability for `grep` with faulty components: `lex` and `dfaanalyze`.

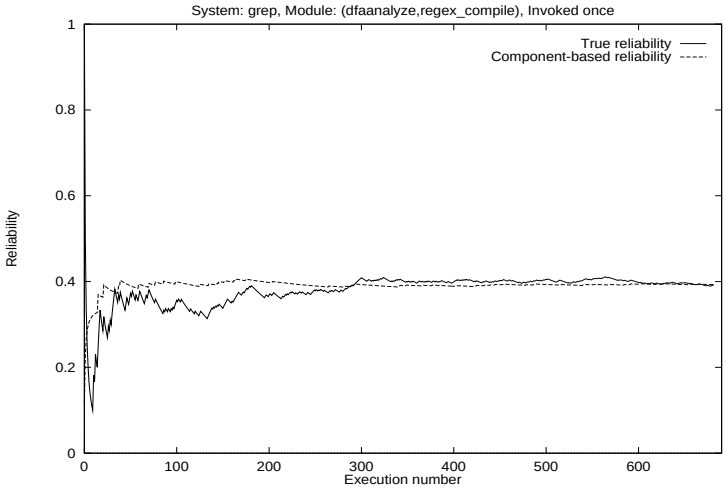

Figure 4: True reliability and component-based reliability for `grep` with faulty components: `dfaanalyze`,`regex_compile`

5 Analysis

We now examine the results presented above in terms of their accuracy and efficiency. We also analyze the effect of modeling intra-component dependency.

5.1 Accuracy

From Table 8 we see that the relative error in the component-based reliability (R_c) estimate of the true reliability (R_t) is 1.78% for version 1 of `grep`. For version 3 the relative error is 0.25%. For both these versions the R_c estimate appears to be a good predictor of the true reliability. However, for version 2 of `grep` the relative error is 33.25% and the R_c estimate is not a good predictor of true reliability. This can also be observed from the graphs in Figures 2, 3 and 4.

5.2 Efficiency

From Table 8 we see that the 0.05-efficiency and 0.01-efficiency is 0.97 and 0.93 for version 1 of `grep`. For version 3 the 0.05-efficiency is 0.97 and the 0.01-efficiency is 0.97. For both these cases, the efficiency is high and the R_c estimate computes the true reliability very fast. For version 2 of `grep` the 0.05-efficiency is 0.99 and the 0.01-efficiency is 0.98. Although the efficiency measures for version 2 are high, the poor accuracy of its R_c estimate means that its R_c values do not converge. Figure 3 illustrates that a high δ-efficiency might not lead to relatively high accuracy. Here the δ-efficiency is computed before the R_c estimate converged and therefore a relatively high value of η_δ is obtained. However, the error in the R_c estimates increased over subsequent executions leading to poor accuracy.

5.3 Dependence

Among the three faulty versions of `grep`, only version 3 results in R_c estimates being the same for DOI = 1 and DOI=∞. This is explained by the fact that in version 3 the faulty components do not occur more than once on any execution path. As discussed in Section 5.1, the accuracy of the R_c estimate, when DOI=1, is very low for version 2 of `grep`. For version 1, the faulty component occurs more than once on an execution path and the accuracy is relatively higher than in version 2.

Figures 5 and 6 show the ratio between R_c and R_{tf} for different degrees of independence. The DOI is best when this ratio is close to 1. From Figure 5 we see that for version 1, the best DOI is 1. However for version 2 (in Figure 6) the best DOI is between 1 and 2. This is why the error in R_c estimate for version 2 for DOI = 1 is high. Figure 7 shows R_t and R_c estimates over multiple executions with DOI = 2. Unlike the graph in Figure 3, the R_c and R_t values converge close to each other in Figure 7. However, the accuracy of the reliability estimate is not as good as it is for version 1 and version 3 of `grep`. This is perhaps because the ideal value of DOI for this version lies between 1 and 2.

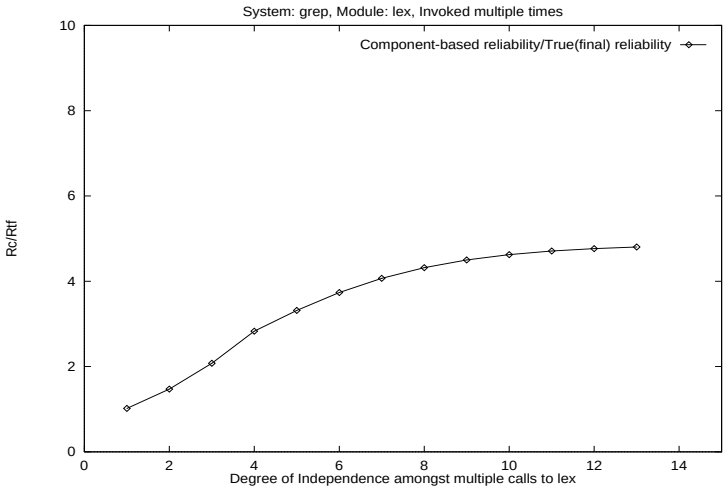

Figure 5: `lex`: Component-based reliability/true reliability vs degree of independence.

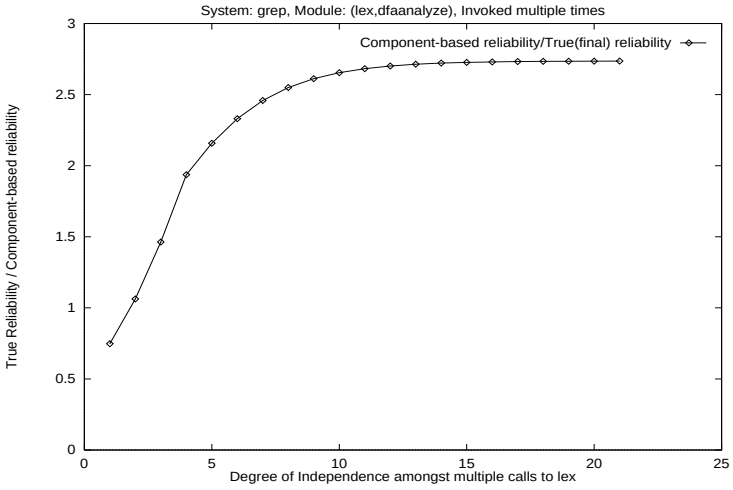

Figure 6: `lex`,`dfaanalyze`: Component-based reliability/true reliability vs degree of independence.

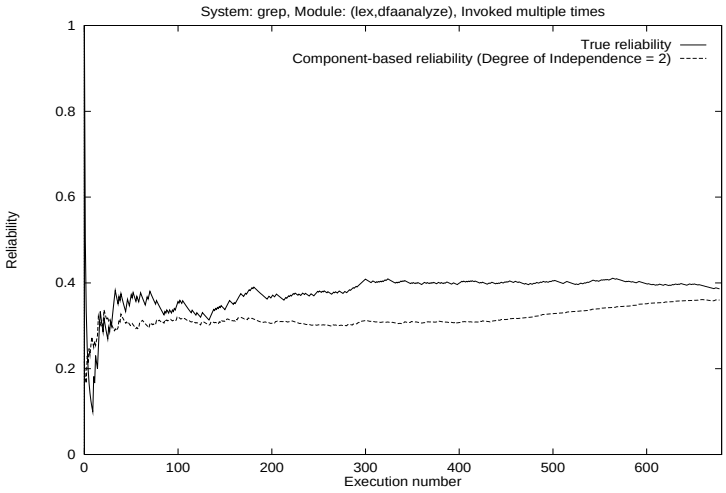

Figure 7: True reliability and component-based reliability (degree of independence = 2) for `grep` with faulty components: `lex`,`dfaanalyze`.

6 Discussion

Though the experiment reported here is insufficient to make any general conclusions about the effectiveness of CBRE, they do help us focus our attention on some key issues that need the attention of researchers. Below we discuss the issues of dependency, sensitivity, operational profile, estimation of the reliabilities of components and their interfaces, applicability of CBRE in a software development environment, and the scalability of CBRE.

Dependency modeling: From the results reported in Section 4, it is clear that the successful use of CBRE depends significantly on inter- and intra-component dependence. It seems that the intra-component reliability is likely to be the more important of the two types of dependencies in determining the accuracy of the system reliability estimate.

As explained earlier, by executing a component an arbitrarily large number of times, it is easy to bring the system reliability estimate much below its true reliability if intra-component dependency is ignored. At this point it is not clear how one should go about determining a suitable value of DOI. It is also not clear whether or not it is appropriate to model the dependency by "collapsing" multiple executions into fewer executions; certainly the results from this experiment show that this method appears to be an attractive when viewed with respect to the accuracy of the reliability estimates.

Sensitivity analysis: One advantage of CBRE is that it provides data that can be used to analyze the effect of changing component reliability on that of the system. Referring to Table 2 in Section 2, we can express the reliability R_c as follows:

$$R_c = \frac{R_1 R_2 R_3 + R_1 R_3 R_4 + R_1 R_2^2 R_3^3 R4 + R_1 R_2 R_3 R_4}{4} \quad (1)$$

Eq. 1 can be used to determine the variation in R_c with respect to that in the reliability of any component. This can be done by fixing the reliabilities of all but one component. Note

that the above formula depends on the tests executed. A different set of paths through the system might lead to different closed form formula for R_c and hence to different results from the sensitivity analysis.

Operational profiles: The traditional method of estimating system reliability relies heavily on the operational profile [10]. It also requires that tests be sampled randomly from the input domain of the system based on this operation profile. Both the determination of the operational profile and random sampling of test cases might be impractical in certain applications [7, 8]. CBRE allows one to use the existing test cases to execute the system and obtain reliability estimates. The existing test cases might be drawn from, or could well be, the system test suite or a regression test suite.

It is unlikely that in practice the existing test cases might have been generated randomly. However, using system or regression tests might lead to an R_c estimate different from the reliability perceived by the user of the system. This might happen when the tests used for estimating R_c are not representative of the user's operational profile. This leads us to propose that one should be able to associate a risk parameter with the R_c estimate. The risk parameter should be independent of the operational profile. One such risk parameter is the code coverage as discussed by Horgan and Mathur [7, 8]. However, more research is needed to help in the selection of and use of appropriate risk parameters to associate with R_c estimates.

Component and interface reliabilities: Past research in component based reliability estimation has resulted in methods for reliability estimation which assume that component reliabilities are available [2, 9]. The issue of how to determine component reliabilities has been mostly ignored. Of course, one might argue that the reliability of a component can be determined using the same method as the one proposed for determining the system reliability. However, using arguments from [7, 8] it is clear that this is not always possible. We propose that when it is not possible to use CBRE or any other available methods to obtain reasonable estimates of components, a coverage-based estimation technique proposed earlier [5, 11] may be used. However, this technique still needs extensive experimental evaluation.

The CBRE as described here makes use of interface reliabilities. In the experiment we did not consider faulty interfaces. In practice, one needs to estimate the reliability of each interface that is on the path executed by the test cases used in the estimation of R_c. When an interface is a program, any of the methods mentioned above can be used to estimate their reliabilities. However, when an interface consists of items such as global variables, parameters, and files it is not clear how to estimate its reliability. A method for integration testing proposed by Delamaro et. al [3]. seems promising for estimating interface reliabilities.

Applicability of CBRE in a software development environment: We believe that CBRE can be applied at all stages of the software development cycle once a part of the system design is available. When only the architecture, and not the code, is available, one can build a simulation model based on CBRE and conduct reliability studies such as ones designed to investigate the impact of component reliability on system reliability for a given architecture.

When the code is available, one can actually apply CBRE and ask questions such as "What is the reliability of a given subsystem?", "What is the system reliability?", and "How can the system reliability be improved?" A tool based on CBRE can help relate the system reliability to the reliability of its components and the architecture.

Scalability of CBRE: CBRE is intended for application to large software systems. The time to execute a system is one parameter that could become a bottleneck in the use of CBRE. If there are N test cases on which the system is to be executed, the total execution time is likely to be proportional to N. Considering the rate at which the CBRE estimates converge, as is evidenced by the efficiency data presented above, it seems that CBRE estimates will converge much faster than estimates based on random testing using traditional methods for estimating reliability [10]. Recall that using CBRE does not require random testing; a system may be executed on system or regression tests and reliability estimates obtained. Considering the difference in testing schemes implied by CBRE and methods that require random testing and the efficiency data presented herein, it appears that CBRE is likely to scale up to large systems more easily than techniques that require random testing.

Based on the above discussion it appears that CBRE is a promising method for estimating software system reliability. However, several issues, as outlined above, need to be resolved before it can be applied in practice.

Acknowledgements

The CBRE method reported in here was suggested by James Berger during the Model Development Workshop at Purdue held on June 17-18, 1996. Other attendees at the workshop were: Josè Maldonado, Aditya Mathur, Alberto Pasquini, Vernon Rego, and Nozer Singpurwalla.

References

[1] M. Chen, A. P. Mathur and V. J. Rego: "Effect of testing techniques on software reliability estimates obtained using a time-domain model," *IEEE Transactions On Reliability*, Vol. 44, No. 1, March 1995, pp. 97-103.

[2] R. C. Cheung, "A user oriented software reliability model," *IEEE Transactions on Software Engineering*, vol. SE-6, March 1980, pp118-125.

[3] M. Delamaro, J. Maldonado, and A. P. Mathur, "Integration testing using interface mutations," *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, White Plains, New York, pp 112-121, October 30-November 2, 1996.

[4] R. A. DeMillo and A. P. Mathur: "A grammar based fault classification scheme and its application to the classification of the errors of TEX," *Technical Report*, SERC-TR-165-P, Software Engineering Research Center, Purdue University, W. Lafayette, IN 47907, 1995.

[5] P. Garg, "On code coverage and software reliability," *M.S. Thesis*, Department of Computer Sciences, Purdue University, May 1995.

[6] J. R. Horgan and S. A. London, "ATAC – Automatic Test Analysis for C programs" *Internal Memorandum TM-TSV-017980, Bell Communications Research*, 1990.

[7] J. R. Horgan and A. P. Mathur, "Software testing and reliability" *Handbook of Software Reliability Engineering*, pp 531-566, McGraw-Hill, New York, 1996.

[8] J. R. Horgan and A. P. Mathur, "Perils of software reliability modeling," Technical Report, SERC-TR-160-P, 1995, Software Engineering Research Center, Purdue University, W. Lafayette, IN.

[9] J. C. Laprie and K. Kanoun, "Software reliability and system reliability," *Handbook of Software Reliability Engineering*, pp 27-70, McGraw-Hill, New York, 1996.

[10] J. D. Musa, A .Iannino, and K. Okumoto, "Software Reliability: Measurement, Prediction and Application" *McGraw-Hill, New York*, 1987.

[11] S. Krishnamurthy and A. P. Mathur, "On predicting the reliability of modules using code coverage," *CD-ROM in CASCON '96*, Toronto, 1996.