CONSTRUCTION OF RELIABLE SOFTWARE IN RESOURCE CONSTRAINED ENVIRONMENTS

Mladen A. Vouk, Department of Computer Science, Box 8206

North Carolina State University, Raleigh, NC 27695, USA

Tel: 919-515-7886, Fax: 919-515-7896, vouk@csc.ncsu.edu

Anthony T. Rivers, Viatec Research, 514 Daniels St. #274 Raleigh, NC 27605

Tel: 919-349-4448, Fax 919-834-1071, arivers@viatecresearch.com

Abstract

Current software development practices are often "business-driven" and therefore tend to encourage testing approaches that reduce verification and validation activities to shorten schedules. In this paper, we focus on efficiency of testing and resulting software quality (reliability) in such an environment. We use a resource constrained testing model to analyze data from several projects. Our results indicate that in a practical resource constrained environment there is little room for improving testing during the actual process. Furthermore, there is likely to be considerable variability in the quality of testing, within a testing phase, that may stem from a variety of sources such as the human element, software development practices and the structure of the software itself. However, it is interesting to note that there are experimental results showing that testing efficiency can be improved over longer periods of time through process improvement, and that software quality does appear to be positively correlated with an increase in the average testing efficiency that stems from process improvements.

1. INTRODUCTION

The challenges of modern "market-driven" software development practices, such as the use of various types of incentives to influence software developers to reduce time to market and overall development cost, seem to favor a resource constrained development approach. In a resource constrained environment, there is a tendency to compress software specification and development processes (including verification, validation, and testing) using either mostly business model directives or some form of "shorthand" technical solution, rather than a combination of sound technical directives and a sensible business model. The business models that often guide modern "internet-based" software development efforts advocate, directly or indirectly, a lot of "corner-cutting" without explicit insistence on software process and risk management procedures, and associated process tracking tools, appropriate to these new and changing environments, e.g., Extreme Programming (XP), use of Web-based delivery tools (Potok and Vouk (1999a), Potok and Vouk (1999b), Beck (2000)). There are also software development cultures, such as XP, specifically aimed at resource constrained development environments (Auer and Miller (2001), Beck (2000), Beck (2001)). Yet, almost as a rule, explicit quantitative quality and process modeling and metrics are subdued, sometimes completely avoided, in such environments. Possible excuses could range from lack of time, to lack of skills, to intrusiveness, to social reasons, and so on. As a result, such software may deliver with an excessive number of problems that are hard to quantify and even harder to relate to process improvement activity in more than an ad hoc manner.

For example, consider testing in "web-year" (i.e., 3-month development cycle (Dougherty (1998)) environments. While having hard deadlines relatively often (say every 3 months) is an excellent idea since it may reduce the process variance, the decision to do so must mesh properly with both business and software engineering models being used or the results may not be satisfactory (Potok and Vouk (1997), Potok and Vouk (1999a), Potok and Vouk (1999b)). One frequent and undesirable effect of a business and software process mismatch is haste. Haste often manifests during both design and testing as a process analogous to a "sampling without replacement" of a finite (and sometimes very limited) number of **pre-determined** structures, functions, and environments. The principal motive is to verify required product functions to an **acceptable** level, but at the same time minimize the re-execution of already tested functions, objects, etc. This is different from "traditional" strategies that might advocate testing of product functions according to the relative frequency of their usage in the field, that is, according to their operational profile[1] (Musa (1999), Vouk (2000)). "Traditional" testing strategies tend to allow for much more re-execution of previously tested functions/operations, and the process is closer to a "sampling with (some) replacement" of a specified set of functions. In addition, "**traditional**" operational profile based testing is quite well behaved, and when executed correctly, allows dynamic quantitative evaluation of software reliability growth based on "classical" software reliability growth metrics and models. These metrics and models can then be used to guide the process (Musa, et al (1987)). When software development and testing is resource and schedule constrained, some traditional quality monitoring metrics and models may become unusable. For example, during non-operational profile based testing, failure intensity decay may be an impractical guiding and decision tool (Rivers (1998), Rivers and Vouk (1999)).

End-product reliability can be assessed in several ways. The aspect of reliability that we are concerned with is efficient reduction of as many defects as possible in the software being shipped. In the next section we describe and discuss the notion of resource constrained software development and testing. In Section 3, we present a software reliability model that formalizes resource constrained development and testing with respect to its outcome. Empirical results, from several sources, that illustrate different constrained software development issues are presented in Section 4.

2. CONSTRAINED DEVELOPMENT

Software development always goes through some basic groups of tasks related to specification, design, implementation, and testing. Of course, there are always additional activities, such as deployment and maintenance phase activities, as well as general management, and verification and validation activities that occur in parallel with the above activities. How much emphasis is put into each phase, and how the phases overlap, is a strong function of the software process adopted (or not) by the software manufacturer, and of how **resource constraints** may impact its modification and implementation (Pressman (2001), Beck (2000), Potok and Vouk (1997)). A typical resource constraint is development time. However, “time” (or exposure of the product to usage, in its many forms and shapes) is only one of the **resources** that may be in short supply when it comes to software development and testing. Other resources may include available staff, personnel qualification and skills, software and other tools employed to develop the product, business model driven funding and schedules (e.g., time-to-market), social issues, etc. All these resources can be **constrained** in various ways, and all do impact software development schedules, quality, functionality, etc., in various ways, e.g., (Boehm (1988), Potok and Vouk (1997)).

Specification, design, implementation, and testing activities should be applied in proportions commensurate with a) the complexity of the software, b) level of quality and safety one needs for the product, c) business model, and d) level of software and process maturity of the organization and its staff.

In a resource constrained environment, some, often all, elements are economized on. Frequently, this "economization" is analogous to a process of "sampling without replacement" of a finite (and sometimes very limited) number of **pre-determined** software system input states, data, artifacts/structures, functions, and environments (Rivers (1998), Rivers and Vouk (1998))."

For instance, think of the software as being a jar of marbles. Take a handful of marbles out of the jar and manipulate them for whatever purpose. For example, examine them for quality or relationship to other marbles. If there is a problem, fix it. Then put the marbles into another jar marked "completed." Then, take another sample out of the first jar, and repeat the process. Eventually, the jar marked "completed" will contain "better" marbles and the whole collection (or "heap") of marbles will be "better." However, any imperfections missed on the one-time inspection of each handful, still remain. That is sampling without replacement. Now start again with the same (unfixed) set of marbles in the first jar. Take a handful of marbles out of the first jar and examine them. Fix their quality or whatever else is necessary, perhaps even mark them as good, and then put them back into the same jar out of which they were taken. Then, on the subsequent sampling of that jar (even if you stir the marbles), there is a chance that some of the marbles you pull out are the same ones you have handled before. You now manipulate them again, fix any problems, and put all of them back into the same jar. You have sampled the marbles in the jar with replacement. Obviously, sampling with replacement implies a cost overhead. You are handling again marbles you have already examined, so it takes longer to "cover" all the marbles in the jar. If you are impatient, and you do not cover all the marbles, you may miss something. However, second handling of a marble may actually reveal additional flaws and may improve the overall quality of the marble "heap" you have. So, a feedback loop or sampling with replacement may have its advantages, but it does add to the overall effort.

To illustrate the impact of resource constrained software development it is interesting to explore the implications of three different software development approaches (Rivers (1998), Rivers and Vouk (1999)):

1) A methodology that has virtually no resource constraints (including schedule constraints), that uses operational profiles to decide on resource distribution, and that allows a lot of feedback and cross-checking during development - essentially a methodology based on development practices akin to sampling

with replacement.

2) A well executed methodology designed to operate in a particular resource constrained environment, based on a combination of operational profiling and effective “sampling without replacement” (Rivers (1998), Rivers and Vouk (1999)).

3) An Ad Hoc, perhaps poorly executed, methodology that tries to reduce resource usage (e.g., the number of test cases with respect to the operational profile) in a manner that, in the marble example, inspects only some of the marbles out the first jar, perhaps using very large handfuls that are given more cursory inspection.

Consider Figure 1. It illustrates fault elimination by the three methods in a hypothetical 100 state system that has 20 faulty states. The actual computations are discussed in section 3.3 in this chapter. Vertical axis indicates the fraction of the latent faults that are **not** “caught” by the selected fault-avoidance or fault-elimination approach, and the horizontal axis indicates expenditure of the resources. These resources are often constrained by a predetermined maximum. The resource usage is expressed as a fraction of the maximum allowed. The actual resource could be anything relevant to the project and impacting fault avoidance and elimination (e.g., time, personnel, a combination of the two, etc.). Details of the calculations suitable in the testing phase are discussed in Rivers (1998) and Rivers and Vouk (1998). The associated constrained testing model is based on the hypergeometric probability mass function and the concept of static and dynamic software structure and construct coverage over the exposure/usage period. This usage period refers to the exposure software receives as part of static and dynamic verification and validation, execution-based testing, and field usage.

In an “ideal” situation, the fault-avoidance and fault-elimination processes are capable of avoiding or detecting all 20 faults, one or more at a time, and defect removal is instantaneous and perfect. An “ideal” approach based on sampling without replacement requires less resources to reach a desired level of “defects remaining” (vertical axis), than methods that sample with replacement, i.e., re-cover already tested states. However, traditional operational profile testing is quite well behaved, and when executed correctly, it may turn out test cases which are more efficient and more comprehensive than those constructed using some non-ideal coverage-based strategy. In that case, testing based on sampling without replacement will yield poorer results.

In Figure 1, **"Ad Hoc**" illustrates a failed attempt to cut the resources to about 20% of the resources that might be needed for a complete operational-profile based test of the product. In the ad hoc method, state coverage was inadequate and although the testing is completed within the required resource constraints, it only detects a small fraction of the latent faults (defects). In the field, this product will be a constant emitter of problems and its maintenance will probably cost many times the resources "saved" during the testing phases.

A way to develop "trimmed" test suites is to start with test cases based on the operational profile and trim the test suite in a manner that preserves coverage of important parameters and coverage measures. One such approach is discussed by Musa (1999). Another one is to use a pairwise test case generation strategy (Cohen, et al (1996), Lei and Tai (1998), Tai and Lei (2002)). Of course, there are many other approaches, and many of the associated issues are still research topics. The software reliability engineering (SRE) model and metrics discussed in the next section allow us to decide early in the process which of the curves the process is following. This then allows us to correct both the product and the process.

From the point of view of this paper of most interest are the SRE activities during system and field testing, and operation and maintenance phases. Standard activities of a **system** and **field testing** phases include: a) execution of system and field acceptance tests, b) checkout of the installation configurations, and c) validation of software functionality and quality. While in the **operation and maintenance** phases the essential SRE elements are a) continuous monitoring and evaluation of software field reliability, b) estimation of product support staffing needs, and c) software process improvement. This should be augmented by requiring software developers and maintainers to a) finalize and use operational profiles, b) actively track the development, testing and maintenance process with respect to quality, c) use reliability growth models to monitor and validate software reliability and availability, and d) use reliability-based test stopping criteria to control the testing process and product patching and release schedules.

Ideally, the reaction to SRE information would be quick, and the corrections, if any, would be applied already within the life-cycle phase in which the information is collected. However, in reality, introduction of an appropriate feedback loop into the software process, and the latency of the reaction, will depend on the accuracy of the feedback models, as well as on the software engineering capabilities of the organization. For instance, it is unlikely that organizations below the third maturity level on the Software

Engineering Institute (SEI) Capability Maturity Model (CMM)[2] scale (Paulk et al (1993b)) would have processes that could react to the feedback information in less than one software release cycle. Reliable latency of less than one phase, is probably not realistic for organizations below CMM level 4. This needs to be taken into account when considering the level and the economics of "corner-cutting." Since the resources are not unlimited, there is always some part of the software that may go untested, or may be verified to a lesser degree. The basic, and most difficult, trick is to decide what must be covered, and what can be left uncovered, or partially covered.

3. MODEL AND METRICS

In a practice, software verification, validation, and testing (VVT) is often driven by intuition. Generation of use cases and scenarios, and of VVT cases in general, may be less than a repeatable process. To introduce some measure of progress and repeatability, structure-based methodologies advocate generation of test cases that monotonously increase the cumulative coverage of a particular set of functional or code constructs the methodology supports. While **coverage-based** VVT in many ways implies that the "good" test cases are those that increase the overall coverage, what really matters is whether the process prevents or finds and eliminates faults, how efficient this process is, and how repeatable is the process of generating VVT activities (test cases) of the same or better quality (efficiency). This implies that the strategy should embody certain fault-detection guarantees. Unfortunately, for many coverage-based strategies these guarantees are weak and not well understood. In practice, designers, programmers and testers are more likely to assess their software using a set of VVT cases derived from functional specifications (as opposed to coverage derived cases). Then, they supplement these "black box" cases with cases that increase the overall coverage based on whatever construct they have tools for. In the case of testing, usually blocks, branches and some data-flow metrics such as p-uses[3] (Frankl and Weyuker (1988)) are employed. In the case of earlier development phases, coverage may be expressed by the cross-references maps and coverage of user-specified functions (requirements) (Pressman (2001)). In fact, one of the key questions is whether we can tell, based on the progress, how successful the process is on delivering with respect to some target quality we wish the software to have in the field (e.g., its reliability in the field), and how much more work we need to assure that goal. In the rest of this section we first discuss the process and the assumptions that we use in developing our model, then we present our model.

3.1. Coverage

Coverage $q(\mathrm{M}, \mathrm{S})$ is computed for a construct, e.g. functions, lines of code, branches, p-use, etc. through metric "M" and an assessment or testing strategy "S" by the following expression

$$q(M, S) = \frac{Number \ of \ executed \ constructs \ for \ M \ under \ S}{Total \ number \ of \ constructs \ for \ M \ under \ S} \tag{1}$$

Notice that the above expression does not account for infeasible (un-executable) constructs under a given testing strategy "S." Hence, in general, there may exist an upper limit on the value of $q(\mathrm{M}, \mathrm{S})$, $q_{\max} \leq 1$. Classical examples of flow metrics are lines of code, (executable) paths, and all definition-use tuples (Beizer (1990)). But, the logic applies equally to coverage of planned test cases, coverage of planned execution time, and the like. We define the later examples as plan-flow metrics, because the object is to cover a "plan."

Assumptions. The resource constrained model of VVT discussed here is based on the work reported in Vouk (1992), Rivers and Vouk (1995), Rivers (1998), Rivers and Vouk (1998) and Rivers and Vouk (1999). We assume that:

1) Resource constrained VVT is approximately equivalent to sampling without replacement. To increase (fulfill) coverage according to metric M, we generate new VVT cases/activities using strategy S so as to cover as many remaining uncovered constructs as possible. The 're-use' of constructs through new cases that exercise at least one new construct (i.e., increase M by at least 1) is effectively "ignored" by the metric M.

2) VVT strategy S generates a set of (VVT or test) cases $T = \{T_i\}$, where each case, T_i, exercises one or more software specification or design functions, or in some other way provides coverage of the product.

3) The order of application/execution of the cases is ignored (and is assumed random) unless otherwise dictated by the testing strategy S.

4) For each set T generated under S and monitored through M, there is a minimal coverage that results in a recorded change in M through execution of a (typical) case T_i from T. There may also be a

maximum feasible coverage that can be achieved, $1 \geq q_{\max} \geq q(M,S) \geq q_{\min} \geq 0$. In general, execution of a complete entry to exit path in a program (or its specifications) will require at least one VVT case, and usually will exercise more than one M construct.

5) Faults/defects are distributed randomly throughout software elements (a software element could be a specification, design or implementation document or code)**.**

6) Detected faults are repaired immediately and perfectly, i.e., when a defect is discovered, it is removed from the program before the next case is executed without introducing new faults.

7) The rate of fault detection with respect to coverage is proportional to the effective number (or density) of residual faults detectable by metric M under strategy S, and is also a function of already achieved coverage.

Assumptions 5 and 6 may seem unreasonable and overly restrictive in practice. However they are very common in construction of software reliability models, since they permit development of an idealized model and solution which can then be modified to either theoretically or empirically account for practical departures from these assumptions (Shooman (1983), Musa (1987), Tohma (1989), Lyu (1996), Karcich (1997)). Except in extreme cases, the practical impact of the assumptions 5 and 6 is relatively limited and can be managed successfully when properly recognized, as we do in our complete model.

3.2. Testing Process

The efficiency of the fault detection process can be degraded or enhanced for a number of reasons. For example, the mapping between defects and defect sensitive constructs is not necessarily one to one, so there may be more constructs that will detect a particular defect, or the defect concentration may be higher in some parts of M-space, i.e. space defined by metric M. An opposite effect takes place when, although a construct may be defect sensitive in principle, coverage of the construct may not always result in the detection of the defect. To account for this, and for any other factors that modify problem detection and repair properties of the constructs, we introduce the **Testing Efficiency** function g_i.

This function represents the “visibility” or “cross-section” of the remaining defects to testing step i. Sometimes we will find more and sometimes less than the expected number of defects, sometimes a defect will remain un-repaired or only partially repaired. g_i attempts to account for that in a dynamic fashion by providing a semi-empirical factor describing this variability. Similar correction factors appear in different forms in many coverage-based models. For instance, Tohma, et al (1989) used the term *sensitivity factor, ease of test,* and *skill* in developing their Hypergeometric Distribution model. Hou, et al (1994) used the term *learning factor,* while Piwowarski, et al (1993) used the term *error detection rate constant* to account for the problem detection variability in their coverage-based model. Malaya, et al (1992) analyzed the Fault Exposure Ratio (Musa, et al (1987)) to account for the fluctuation of per-fault detectability while developing their exponential and logarithmic coverage based models.

Let the total number of defects in the program P be N. Let the total number of constructs that need to be covered for that program be K. Let execution of T_i increase the number of covered constructs by $h_i > 0$. Let u_i be the number of constructs that still remain uncovered before T_i is executed. Then,

$$u_i = K - \sum_{j=1}^{i-1} h_j \tag{2}$$

Let D_i be the subset of d_i faults remaining in the code **before** T_i is executed. Let execution of T_i detect (and result in the removal of) x_i defects. Let there be at least one construct, in u_i, the coverage of which guarantees detection of each of the remaining d_i defects. That is, all items in D_i are detectable by un-executed test cases generated under S.

The hypergeometric probability mass function (Walpole and Myers (1978)) describes the process of sampling without replacement, the process of interest in a resource constrained environment. Under our assumptions, the probability that test case T_i detects x_i defects is at least:

$$p(x_i | u_i, b_i, h_i) = \frac{\binom{b_i}{x_i} \cdot \binom{u_i - b_i}{h_i - x_i}}{\binom{u_i}{h_i}} \tag{3}$$

where $b_i = g_i \cdot d_i$. Note that g_i is usually a positive real number, i.e., it is not limited to the range of all positive integers as equation (3) requires. Hence when using (3) one has to round off computed b_i, to the nearest integer subject to system-level consistency constraints. In practice, there is likely to be considerable variability of g_i within a testing phase. g_i's variability stems from a variety of sources, such as, the structure of the software, the software development practices employed, and of course, the human element.

In equation (3), the numerator is the number of ways h_i constructs with x_i defectives can be selected. The first term in the numerator is the number of ways in which x_i defect sensitive constructs can be selected out of b_i that can detect defects (and are still uncovered). The second term in the numerator is the number of ways in which $h_i - x_i$ non-defect sensitive constructs can be selected out of $u_i - b_i$ that are not defect sensitive (and are still uncovered). The denominator is the number of ways in which h_i constructs can be selected out of u_i that are still uncovered. The expected value $\overline{x_i}$ of x_i is

$$\overline{x}_i = (g_i \cdot d_i)\left(\frac{h_i}{u_i}\right) \tag{4}$$

(Walpole and Myers (1978)). Let the total amount of cumulative coverage up to and including $T_i - 1$ be

$$q_{i-1} = \sum_{j=1}^{i-1} h_j \tag{5}$$

and let the total number of cumulative defects detected up to and including $T_i - 1$ be

$$E_{i-1} = \sum_{j=1}^{i-1} x_j \tag{6}$$

Since $d_i = (N - E_{i-1})$, equation (4) becomes

$$\overline{x}_i = \overline{E}_i - \overline{E}_{i-1} = \Delta\overline{E}_i = g_i\left(N - \overline{E}_{i-1}\right)\left(\frac{q_i - q_{i-1}}{K - q_{i-1}}\right) = g_i\left(N - \overline{E}_{i-1}\right)\left(\frac{\Delta q_i}{K - q_{i-1}}\right) \tag{7}$$

where $\overline{E_i}$ denotes the expected value function. In the small step limit, this difference equation can be transformed into a differential equation by assuming that $\Delta\overline{E_i} \rightarrow d\overline{E_i}$, and $\Delta q_i \rightarrow dq_i$, i.e.

$$d\overline{E}_i = g_i\left(N - \overline{E}_i\right)\left(\frac{dq_i}{K - q_i}\right) \tag{8}$$

Integration yields the following model:

$$E_i = N - (N - E_{\min})e^{-\int_{q_{\min}}^{q_i} \frac{g_i}{K - \varsigma} d\varsigma} \tag{9}$$

where $q_{\min}$ represents the minimum meaningful coverage achieved by the strategy. For example, one test case may, on the average, cover 50-60% of statements (Jacoby and Masuzawa (1992)). Function g_i may have strong influence on the shape of the failure intensity function of the model. This function encapsulates the information about the change in the ability of the test/debug process to detect faults as coverage increases.

Because the probability of “trapping” an error may increase with coverage, stopping before full coverage is achieved makes sense only if the residual fault count has been reduced below a target value. This threshold coverage value will tend to vary even between functionally equivalent implementations. The model shows that metric saturation can occur, but that additional testing may still detect errors. This is in good agreement with the experimental observations, as well as conclusions of other researchers working in the field, (e.g., Chen, et al (1995)). For some metrics, such as the fraction of planned test cases, this effect allows introduction of the concept of “hidden” constructs, re-evaluation of the “plan” and estimation of the additional “time” to target intensity in a manner similar to “classical” time-based models (e.g., Musa, et al (1987)).

Now, let $g_i = a$, where “a” is a constant. Let it represent the average value of g_i over the testing period. Then the cumulative failures model of (9) becomes

$$E_i = N - (N - E_{\min})\left(\frac{K - q_i}{K - q_{\min}}\right)^a \tag{10}$$

To derive a failure intensity equation λ_i based on $g_i = a$, we take the derivative of equation (10) with respect to q_i, which yields

$$\lambda_i = \left(\frac{a}{K - q_{\min}}\right)(N - E_{\min})\left(\frac{K - q_i}{K - q_{\min}}\right)^{(a-1)} \tag{11}$$

3.3. Sampling With and Without Replacement

Let us further explore the implications of a well-executed testing methodology based on “sampling without replacement” (see Figure 1). We did that by developing two solutions for relationship (7), using a reasoning similar to that described in Tohma, et al, (1989) and Rivers (1998). These solutions compute the number of defects shipped, i.e., the number of defects that leave the phase in which we performed the testing.

The first case is a solution for the expected number of defects that would be shipped after test-step i based on an “ideal” non-operational profile based testing strategy is (Rivers, 1998):

$$l_{i0} = N - \overline{E}_i = N - N\left(1 - \prod_{j=1}^{i}\left(1 - \frac{g_i \cdot h_i}{u_i}\right)\right) \tag{12}$$

In the second case, to emulate an “ideal” strategy based on sampling with replacement, we start with a variant of equation (4) with K in place of u_i and find the solution (Rivers (1998)):

$$l_{i1} = N - \overline{E}_i = N - N\left(1 - \prod_{j=1}^{i}\left(1 - \frac{g_i \cdot h_i}{K}\right)\right) \tag{13}$$

In an “ideal” situation: a) the test suite has in it the test cases that are capable of detecting all faults (if all faults are **M-detectable**), and b) defect removal is instantaneous and perfect. In “best” cases, g_i is consistently 1 or larger. Plots of (12) and (13) are shown in Figure 1. Equation 12 is denoted as “guided constrained testing” in the figure, and equation (13) is denoted as “traditional testing” with g_i of 1 (constant during the whole testing phase) and with N=20 and K=100 in both cases. The horizontal axis was normalized with respect to 500 test cases. We see, without surprise, that “ideal” methods based on sampling without replacement require less test-steps to reach a desired level of “defects remaining” than methods that re-use test-steps (or cases) or re-cover already tested constructs.

However, when there are deviations from the “ideal” situation, as is usually the case in practice, the curves could be further apart, closer together, or even reversed. When testing efficiency is less than 1, or the test suite does not contain test-steps (cases) capable of uncovering all software faults, a number of defects may remain uncovered by the end of the testing phase.

The impact of different (but “constant” average) g_i values on sampling without replacement (non-operational testing) was explored using a variant of (10) and (12) (Rivers (1998)):

$$\text{ShippedDefects} = N - \overline{E}_i = N - N\left(1 - (K - q_i)^a\right) \tag{14}$$

If g_i is constantly less than 1, some faults remain undetected. In general, the (natural) desire to constrain the number of test cases and “sample without replacement” is driven by business decisions and resource constraints. As mentioned earlier, the difficult trick is to design a testing strategy (and coverage metric) that results in a value of g_i that is larger than 1 for most of the testing phase.

4. CASE STUDIES

4.1. Issues

We use empirical data on testing efficiency to illustrate some important issues related to resource constrained software development and testing. One issue is that of more **efficient use of resources**. That is, whether it is possible to constrain resources and still improve (or just maintain adequate) verification and validation processes and end-product quality as one progresses through software development phases.

The overall idea is that the designers and testers, provided they can **learn** from their immediate experience in a design or testing phase, can either shorten the life-cycle by making that phase more efficient, or they can improve the quality of the software more than they would without the “learning” component. This implies some form of verification and validation feedback loop in the software process. Of course, all software processes recommend such a loop, but how immediate and effective the feedback is depends, to a large extent, on the capabilities and maturity of the organization (Paulk, et al (1993a), Paulk, et al (1993b)). In fact, as mentioned earlier, reliable forward-correcting within-phase feedback requires CMM level 4 and level 5 organizational capabilities. There are not many organizations like that around (Paulk, et al (1993a), Paulk, et al (1993b)). Most software development organizations operate at level 1 (ad hoc), level 2 (repeatable process), and perhaps level 3 (metrics, phase to phase improvements, etc.). However, all three lowest levels are re-active, and therefore the feedback may not operate at fine enough task/phase granularity to provide cost-effective verification and validation “learning” within a particular software release.

The issue is especially interesting in the context of Extreme Programming (Auer and Miller (2001), Beck (2000), Beck (2001)). XP is a light-weight software process that often appeals to (small?) organizational units looking at rapid prototypes and short development cycles. Because of that, it is likely to be used under quite considerable resource constraints. XP advocates several levels of verification and validation feedback loops during the development. In fact, a good problem determination method integral to XP (Beck (2000), Beck (2001)) is Pair Programming[4]. Unfortunately, one of the characteristics of XP philosophy is that it tends to avoid “burdening” the process and participants with “unnecessary” documentation, measurements, tracking, etc. This appears to include most traditional SRE metrics and methods. Hence, to date (early 2002), almost all evidence of success or failure of XP is, from the scientific point of view, anecdotal. Yet, for XP to be successful it has to cut a lot of corners with respect to the traditional software processes. That means that fault-avoidance and fault-elimination need to be even more sophisticated than usual. Pair Programming, a standard component of XP, is an attempt to do that. However, because of its start-up cost (two programmers vs. traditional one programmer) it does imply a considerable element of fault-avoidance and -elimination super-efficiency. In the next sub-section, we will see how traditional methods appear to fare in that domain.

Some other notable issues include situations where the absence of a decreasing trend in **failure intensity** during resource constrained system testing is NOT an indicator that the software would be a problem in the field. This is something that is contrary to the concept of reliability growth based on the "classical" reliability models. There is also the issue of situations where the **business model** of an organization overshadows the software engineering practices and appears to be the sole (and sometimes poor) guidance mechanism of software development (Potok and Vouk (1999a), Potok and Vouk (1999b), Potok and Vouk (1997)).

4.2. Data

We use three sources of empirical information to illustrate our discussion. Two data sets are from industrial development projects (with CMM levels in the range 1.5 to 2.5), and one is from a set of very formalized software development experiments conducted by four universities (CMM level of the experiment was in the range 1.5 to 3). While formal application and use of software analysis, design, programming, and testing methods based on the object oriented[5] (Pressman (2001)) paradigm were not employed in any of the projects, it is not clear that the quality of the first generation object-oriented software (with minimal re-use) is any different from that of first generation software developed using "traditional" structured methods (Potok and Vouk (1997)). Hence, we believe that the issues discussed here still apply today in the context of many practical software development shops since the issues are related more to the ability of human programmers to grasp and deal with complex problems, than to the actual technology used to produce the code.

The first dataset (DataSet1) are test-and-debug data for a PL/I database application program described in Ohba (1984), Appendix F, Table 4. The size of the software was about 1.4 million lines of code, and the failure data were collected using calendar and CPU execution time. It is used, along with DataSet2, to discuss the issue of **resource conservation, or enhancement,** through on-the-fly "**learning**" that may occur during testing of software.

The second dataset (DataSet2) comes from the NASA LaRC experiment in which twenty implementations of a sensor management system for a redundant strapped down inertial measurement unit (RSDIMU) were developed to the same specification (Kelly, et al (1988)). The data discussed in this paper are from the "certification" phases. Independent certification testing consisted of a functional certification test suite and a random certification test suite. The cumulative execution coverage of the code by the test cases was determined through post-experimental code instrumentation and execution tracing using BGG (Vouk and Coyle (1989)). After the functional and random certification tests, the programs were subjected to an independent operational evaluation test which found additional defects in the code. Detailed descriptions of the detected faults and the employed testing strategies are given in Vouk, et al (1990), Eckhardt, et al (1991), and Paradkar, et al (1997).

The third dataset (DataSet3) comes from real software testing and evaluation efforts conducted during late unit and early integration testing of four consecutive releases of a very large (millions of lines of code) commercial telecommunications product. Where necessary, the plots shown here are normalized to protect proprietary information.

4.3. Within-Phase Testing Efficiency

We obtain an empirical estimate, $\hat{g}_i(q_i)$ (Rivers (1998), (Rivers and Vouk (1998)), of the testing efficiency function, $g_i(q_i)$, at instant i using

$$\hat{g}_i(q_i) = \frac{\Delta E_i (K - q_{i-1})}{(N - E_{i-1}) \cdot \Delta q_i} \tag{15}$$

where ΔE_i is the difference between the observed number of cumulative errors at steps i and i-1, q_i is the observed cumulative construct coverage up to step i, and Δq_i is the difference between the number of cumulative constructs covered at steps i and i-1. It is interesting to note that equation (15), in fact, subsumes the *learning factor* proposed by Hou, et al (1994), and derived from the sensitivity factor defined by Tohma, et al (1989), i.e.,

$$\frac{w_i}{N} = \frac{\Delta E_i}{N - E_{i-1}} \tag{16}$$

where w_i is the *sensitivity factor*. It also subsumes the metric-based failure intensity, $\left(\frac{\Delta E_i}{\Delta q_i}\right)$. We define estimated Average Testing Efficiency as

$$A = \sum_{i=1}^{n} \frac{\hat{g}_i(q_i)}{n} \tag{17}$$

4.3.1 "Learning." (DataSet1)

As mentioned earlier, researchers have reported the presence of so called "learning" during some testing phases of software development. That is to say, "learning" that improved the overall efficiency of the testing on the fly (e.g., Hou, et al (1994)). If really present, this effect may become a significant factor in software process self-improvement, and may become even more important if a product VVT team is considered as a collaborating community. Such a community needs an efficient collaborative knowledge-sharing method built into the process. Pair programming may be such a method in the scope of two developers, a paradigm that extends that concept to the whole project VVT community, and may solve a lot of problems. We explore the single-programmer issue further.

Figure 2 illustrates the difference between "learning factor" and "testing efficiency" functions using the same data as Hou, et al (1994) and Ohba (1984). We assume that their testing schedule of 19 weeks was planned, and that N=358. Therefore, on the horizontal axis we plot the fraction of the schedule that is completed. In Figure 2, we show the "learning factor" based on equation (16) on the left, the corresponding "testing efficiency" computed using equation (15) on the right. The thing to note is that the original learning factor calculations do not keep track of remaining space to be covered, and therefore tend to underestimate the probability of finding a defect as one approaches full coverage. When coverage is taken into account, *we see that the average efficiency of the testing appears to be approximately the same over the whole testing period, i.e., it would appear that no "learning" really took place.* We believe that this uniformity in the efficiency is the result of a schedule (milestone) or test suite size constraint. It is very likely that the effort used a test suite and schedule that allowed very little room for change, and therefore there was really not much room to improve on the test cases. Of course, the large variability in the testing efficiency over the testing period probably means that a number of defects were not detected. Comparison of the failure intensity and the cumulative failure fits obtained using our model and Ohba's model, shows that our coverage model fits the data as well as the exponential or the S-shaped models (Rivers (1998)).

4.3.2 Impact of Program Structure (DataSet2)

To investigate the problem further, we turn to a suite of functionally equivalent programs developed in a multi-university fault-tolerant software experiment for NASA. Figure 3 illustrates the testing efficiency that was measured for the certification test suite used in the NASA-LaRC experiment. The certification test suite was administered after programmers finished unit testing their programs. During this phase, the programmers were allowed to fix only problems explicitly found by the certification suite. In this illustration we use p-use coverage data for three of the 20 functionally equivalent programs, and we set N to the number of faults eventually detected in these programs through extensive tests that followed the "certification" phase (Vouk, et al (1990), Eckhardt, et al (1991)). In the three graphs in Figure 3, we plot the "instantaneous," or "per test-step," testing efficiency and its arithmetic average over the covered metric space. The first coverage point in all three graphs refers to the first test case which detected a failure, the last coverage point in all three graphs refers to the last test case in the suite. Note that in all three cases the last efficiency is zero because no new faults were detected by the last set of test cases. *We see that the same test suite exhibits considerable diversity and variance in both its ability to cover the code, and its ability to detect faults.* Similar results were obtained when statement- and branch-coverage were used (Rivers (1998)).

4.3.3 Discussion

From Figure 3 we see that average testing efficiency ranges from about 1 for P9, to about 1.5 for P13 to about 5 for P3. Since the programs are functionally equivalent, and we used **exactly** the same test suite (developed on the basis of the same specifications) and data grouping in all 20 cases, the opportunity to "learn" dynamically during testing was non-existent. *The root cause of variability is the algorithmic and implementational diversity.* That is, the observed testing efficiency range was purely due to the test suite generation approach (black-box specification-based, augmented with extreme-value and random cases, but fixed for all program tests) and the diversity in the implemented solutions stemming from the variability in the programmer micro-design decisions and skills. Yet, when we applied the “learning” models to the same data, “learning” was “reported” (Rivers (1998), Rivers and Vouk (1998)). This is a strong indicator that

a) a constrained testing strategy may perform quite differently in different situations,
b) one should not assume that there is much time to improve things on-the-fly when resources are scarce, and
c) constrained specification-based test generation needs to be supplemented with a more rigorous (and semi-automatic or automatic) test case selection approach such as predicate-based testing (Paradkar, et al (1997)) or pairwise testing (Cohen, et al (1996), Lei and Tai (1998), Tai and Lei (2002)) in an attempt to increase efficiency of the test suite as much as possible.

CMM levels for the unit development in the above projects were in the range between 1 and 2, perhaps higher in some cases. Since light-weight resource constrained processes of XP-type (Cockburn and Williams (2000), Cockburn and Williams (2001)), by their nature, are unlikely to rise much beyond level 2, it is quite possible that the principal limiting factor for XP-type development is, just like in the case-studies presented here, the human component. That is to say, individual analysts, designers, (pair-) programmers, testers, etc. An interesting question to ask is “Would a process constraining and synchronizing activity like XP embedded pair-programming have enough power to sufficiently reduce the variance in the pair-combination of individual programmer capabilities to provide a reliable way of predicting and achieving an acceptable cost to benefit ratio for large scale use of this paradigm?” Currently, there is no evidence that this is consistently the case unless the process is at least above level 3 on the CMM scale.

4.4. Phase-to-Phase Testing Efficiency

A somewhat less ambitious goal is to improve the resource constrained processes from one phase to another. This is definitely achievable, as the following case study shows. Again, we use the testing efficiency metric to assess the issue. Figure 4 illustrates the “testing efficiency” results for releases 1 and 4 of the resource constrained non-operational testing phase of four consecutive releases of a large commercial software system (DataSet3). Similar results were obtained for Releases 2 and 3 (Rivers (1998)). All releases of the software had their testing cycle constrained in both the time and test size domains. We obtain estimates for the parameter N by fitting the “constant testing efficiency” failure intensity model of equation (11) to grouped failure intensity data of each release. We used the same data grouping and parameter estimation approaches in all four cases. To obtain our parameter estimates, we used conditional least squares estimation. Logarithmic relative error was used as a weighting function. This provided a bound on the estimate corresponding to the last observed cumulative number of failures that was not lower than this observed value. The “sum of errors squared” (SSE) relationship we used is

$$SSE = \sum_{i=1}^{n} \left(\begin{array}{l} \left(\lambda_{actual_i} - \left(\dfrac{a}{K - q_{\min}} \right) (N - E_{\min}) \left(\dfrac{K - q_i}{K - q_{\min}} \right)^{(a-1)} \right)^2 \\ + \left(\ln(E_{end}) - \ln\left(N - (N - E_{\min}) \left(\dfrac{k - q_{end}}{k - q_{\min}} \right)^{a} \right) \right)^2 \end{array} \right) \tag{18}$$

where E_{end} is the number of cumulative failures we observed at juncture q_{end} within the testing phase. Note that the experimental “testing efficiency” varies considerably across the consecutive releases of this same software system despite the fact that the testing was done by the same organization. The running average is a slowly varying function of the relative number of planned test cases. The final average values of testing efficiency for each of the four releases is in Figure 5. It shows that the average testing efficiency increases from about 1 to about 1.9 between Release 1 through Release 4. We conjecture that the absolute efficiency of the test suites has also increased. Based on the above information, we feel that the average constant testing efficiency model is probably an appropriate model for this software system and test environment, at least for the releases examined in this study.

We fit the “constant efficiency” model to empirical failure intensity and cumulative failure data for all releases. We also attempted to describe the data using a number of other software reliability models. The “constant testing efficiency” model was the only one that gave reasonable parameter estimates for all four releases (Rivers (1998)). Relative error tests show that the “constant efficiency” model fits the data quite well for all releases. Most of the time, the relative error is of the order of 10% or less.

4.5. Testing Efficiency and Field Quality

In this section we discuss the relationship we observed for our “DataSet3” between the testing efficiency and the product field quality. Our metric of field quality is “cumulative field failures” divided by “cumulative in-service time.” In-service time is the sum of “clock times” due to all the licensed systems, in one site or multiple sites, that are simultaneously running a particular version of the software. To illustrate, when calculating in-service time, 1000 months of in-service time may be associated with 500 months (clock time) of two systems, or 1 month (clock time) of 1000 systems, or 1000 months (clock time) of 1 system (Jones and Vouk (1996)). Let C_j be the in-service time accumulated in Period_j. Then cumulative inservice time is *c*

$$c = \sum_{i=StartDay}^{EndDay} (C_j) \tag{19}$$

For example, if version X of the software is running continuously in a total of 3 sites on day one, then in-service time for day one is 3 days. If on day two software version X is now running in a total of 7 sites, then in-service time for day two is 7 days and the cumulative in-service time for the two days, measured on day two, is 10 days. Similar sums can be calculated when in-service time is measured in weeks or months. Let F_j be the number of field failures observed in Period_j. Then the number of cumulative field failures for the same period is *f*

$$f = \sum_{i=StartCount}^{EndCount} (F_j) \tag{20}$$

Our field quality metric (or field problem proneness) is *z*

$$z = \frac{f}{c} \tag{21}$$

In the case of this study, the lower the Problem Proneness, the "better" the field quality.

Figure 5 shows the relationship between the average testing efficiency and field quality. It appears that, as the average testing efficiency increases, the field quality gets better. However, since we do not know the absolute efficiency of the test suites used, we can only conjecture based on the fact that all test suites were developed by the same organization and for the same product. It is also interesting to note that the relationship between average testing efficiency and the quality is not linear. From Figure 5, we see that dramatic gains were made in average testing efficiency from Release 3 to Release 4. We also see that after Release 1, field problem proneness is reduced. This may be a promising sign that the organization's testing plan is becoming more efficient, and that the company quality initiatives are producing positive results.

5. SUMMARY

We have presented an approach for evaluation of testing efficiency during resource constrained non-operational testing of software that uses “coverage” of logical constructs, or fixed schedules and test suites, as the exposure metric. We have used the model to analyze several situations where testing was conducted under constraints. Our data indicate that, in practice, testing efficiency tends to vary quite considerably even between functionally equivalent programs. However, field quality does appear to be positively correlated with the increases in the average testing efficiency. In addition to computing testing (or VTT) efficiency, there are many other metrics and ways one can use to see if one is on the right track when developing software under resource constraints. A very simple approach is to plot the difference between detected and resolved problems. If the gap between the two is too large, or is increasing, it is a definite sign that resources are constrained and the project may be in trouble.

Modern business and software development trends tend to encourage testing approaches which economize on the number of test cases to shorten schedules. They also discourage major changes to test plans. Therefore, one would expect that, on the average, the testing efficiency during such a phase is roughly constant. However, learning (improvements) can take place in consecutive testing phases, or from one software release to another of the same product. Our data seem to support both of these theories. At this point, it is not clear which metrics and testing strategies actually optimize the returns. Two of many candidate test development strategies are the pairwise testing (Cohen, et al (1996) Lei and Tai (1998)) and predicate-based testing (Paradkar, et al (1997)).

ACKNOWLEDGMENTS

The authors wish to thank Marty Flournory, John Hudepohl, Dr. Wendell Jones, Dr. Garrison Kenney and Will Snipes for helpful discussions during the course of this work.

REFERENCES

Auer and R. Miller (2001). *XP Applied*, Addison-Wesley, Reading, Massachusetts.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Reading, Massachusetts.

Beck, K. (2001). “Aim, Fire,” *IEEE Software*, **18**:87-89.

Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold, New York, New York.

Boehm, B.W. (1981). *Software Engineering Economics*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

Boehm, B.W. (1988). “A Spiral Model of Software Development and Enhancement,” *IEEE Computer*, **21(5)**:61-72.

Boehm, B.W. (1989). “Software Risk Management,” *IEEE CS Press Tutorial.*

Chan, F., Dasiewicz, P., and Seviora, R. (1991). “Metrics for Evaluation of Software Reliability Growth Models,” *Proc. 2nd ISSRE*, Austin, Texas, **1**:163-167.

Charette, R.N. (1989). *Software Engineering Risk Analysis & Management*, McGraw-Hill.

Chauhan, D.P. (2001). “Software Process Improvement–A Successful Journey,” *Software Quality, ASQ*, **1**:1-4.

Chen, M.H., Mathur, Aditya P., and Rego, Vernon J. (1995). “Effect of Testing Techniques on Software Reliability Estimates Obtained Using A Time-Domain Model,” *IEEE Transactions on Reliability*, **44(1)**:97-103.

Cockburn A. and Williams, L. (2000). “The Costs and Benefits of Pair Programming,” *eXtreme Programming and Flexible Processes in Software Engineering -- XP2000*, Cagliari, Sardinia, Italy.

Cockburn A. and Williams, L. (2001). “The Costs and Benefits of Pair Programming,” *Extreme Programming Examined* Addison Wesley, Boston, MA, pp. 223-248.

Cohen, D.M., Dalal, S.R., Parelius, J. and Patton, G.C. (1996). “The Combinatorial Design Approach to Automatic Test Generation,” *IEEE Software*, **13(15).**

Dougherty, Dale (1998). “Closing Out the Web Year,” *Web Techniques*: (http://www.webtechniques.com/archives/1998/12/lastpage/).

Eckhardt, D.E., Caglayan, A.K., Kelly, J.P.J., Knight, J.C., Lee, L.D., McAllister, D.F., and Vouk, M.A., (1991). “An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability,” *IEEE Trans. Software Engineering*, **17(7)**:692-702.

Erdogmus H. and Williams, L., (2002). “Value-Based Analysis of Pair Programming,” *International Conference on Software Engineering (ICSE)*, Argentina.

Frankl, P. and Weyuker, E.J. (1988). “An Applicable Family of Data Flow Testing Strategies,” *IEEE Transactions on Software Engineering*, **11(10)**:1483-1498.

Hou, R.H., Kuo, Sy-Yen, Chang, Yi-Ping (1994). “Applying Various Learning Curves to the Hypergeometric Distribution Software Reliability Growth Model,” *5th ISSRE*, pp. 196-205.

IEEE (1989). *IEEE Software Engineering Standards*, Third Edition, IEEE, 1989.

Jacoby, R. and Masuzawa, K. (1992). "Test Coverage Dependent Software Reliability Estimation by the HGD Model," *Proceedings. 3rd ISSRE,* Research Triangle Park, North Carolina, pp. 193-204.

Jones, W. and Vouk, M.A. (1996). "Software Reliability Field Data Analysis," *Handbook of Software Reliability Engineering*, McGraw Hill.

Karcich, R. (1997). *Proceedings 8th International Symposium on Software Reliability Engineering: Case Studies*, Albuquerque, NM.

Kelly, J., Eckhardt, D., Caglayan, A., Knight, J., McAllister, D., Vouk, M. (1988). "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results", *Proc. FTCS* **18**, pp. 9-14

Lei, Y. and Tai, K.C. (1998). "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing," *Proceedings of the IEEE International Conference in High Assurance System Engineering*.

Lyu, M. (1996). *Handbook of Software Reliability Engineering*, McGraw Hill.

Malaiya, Y.K. and Karich, R. (1994). "The Relationship Between Test Coverage and Reliability," *5th ISSRE*, pp. 186-195.

Malaiya, Y.K., Von Mayrhauser, A. and Srimani, P. K. (1992). "The Nature of Fault Exposure Ratio," *3rd International Symposium on Software Reliability Engineering*, Research Triangle Park, NC, pp. 23-32.

Musa, J.D., Iannino, A., and Okumoto, K. (1987). *Software Reliability: Measurement Predictions, Application*, McGraw-Hill.

Musa, J. D. (1993). "Operational profiles in Software-Reliability Engineering," *IEEE Software*, **10(2)**:14-32.

Musa, J. D. (1999). *Software Reliability Engineering*, McGraw-Hill.

Ohba, M. (1984). "Software Reliability Analysis Models," *IBM J. of Res. and Development*, **28(4)**:428-443.

Paradkar A., Tai, K.C., and Vouk, M., (1997). "Specification-based Testing Using Cause-Effect Graphs," *Annals of Software Engineering*, **4**:133-158.

Paulk, M.C., Curtis, B., Chrissis, M. B., and Weber, C.V. (1993a). "Capability Maturity Model for Software," Version 1.1, Technical Report CMU/SEI-93-TR-024 ESC-TR-93-177, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Paulk, M.C., Curtis, B., Chrissis, M. B., and Weber, C.V. (1993b). "Capability Maturity Model," Version 1.1, *IEEE Software*, pp. 18-27.

Piwowarski, P., Ohba, M., Caruso, J. (1993). "Coverage measurement experience during function test," *ICSE 93*, pp. 287-300.

Potok, T. and Vouk, M.A. (1997). “The Effects of the Business Model on the Object-Oriented Software Development Productivity,” *IBM Systems Journal*, **36(1)**:140-161.

Potok, T. and Vouk, M.A. (1999a). “Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment,” *Software Practice and Experience*, **29(1)**:833-847.

Potok, T. and Vouk, M.A. (1999b). “A Model of Correlated Team Behavior in a Software Development Environment,” *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'99)*, Richardson, Texas, pp. 280-283.

Pressman, R.S. (2001). *Software Engineering: A Practitioner’s Approach* 5th Ed. McGraw-Hill Inc. New York.

Rivers, A.T. and Vouk, M.A. (1995). “An Experimental Evaluation of Hypergeometric Code Testing Coverage Model,” *Proc. Software Engineering Research Forum*, Boca Raton, FL, pp. 41-50.

Rivers, A.T. (1998). “Modeling Software Reliability During Non-Operational Testing,” *Ph.D. dissertation* North Carolina State University.

Rivers, A.T. and Vouk, M.A. (1998). “Resource constrained Non-Operational Testing Of Software,” *Proceedings ISSRE 98, 9th International Symposium on Software Reliability Engineering*, Paderborn Germany-**Received ‘Best Paper Award’**.

Rivers, A.T. and Vouk, M.A. (1999). “Process Decision for Resource Constrained Software Testing,” *Proceedings ISSRE 99, 10th International Symposium on Software Reliability Engineering*, Boca Raton, Fl.

Shooman, M.L. (1983). *Software Engineering*, McGraw-Hill, New York.

Tai, K.C. and Lei, Y. (2002). “A Test Generation Strategy for Pairwise Testing,” *IEEE Transactions on Software Engineering*, **28(1)**:109-111.

Tohma, Yoshihiro, Jacoby, Raymond, Murata, Yukihisa, Yamamoto, Moriki (1989). “Hypergeometric Distribution Model to Estimate the Number of Residual Software Faults,” *Proc. COMPSAC 89*, Orlando, Florida pp. 610-617.

Vouk, M.A. and Coyle, R.E. (1989). “BGG: A Testing Coverage Tool” *Proc. Seventh Annual Pacific Northwest Software Quality Conference*, Lawrence and Craig, Inc., Portland, OR, pp.212-233.

Vouk, M. A., Caglayan, A., Eckhardt, D.E., Kelly, J.P.J., Knight, J., McAllister, D.F., Walker, L. (1990). “Analysis of faults detected in a large-scale multi-version software development experiment,” *Proc. DASC '90*, pp. 378-385.

Vouk, M.A. (1992). “Using Reliability Models During Testing with Non-Operational Profiles,” *Proc. 2nd Bellcore/Purdue Workshop Issues in Software Reliability Estimation*, pp. 103-111.

Vouk, M.A. (2000). “Software Reliability Engineering” - a tutorial presented at the *Annual Reliability and Maintainability Symposium* http://renoir.csc.ncsu.edu/Faculty/Vouk/vouk_se.html

Walpole, R.E. and Myers, R.H. (1978). *Probability and Statistics for Engineers and Scientists*, Macmillan Publishing Co., Inc.

Williams, L. A. and Kessler, R. R. (2000). "All I Ever Needed to Know About Pair Programming I Learned in Kindergarten," *Communications of the ACM*, **43**

EXERCISES

1. Construct/program and run simulations that will verify the results shown in Figure 1. Make sure you very clearly state all the assumptions, relationships (equations) you use, and so on. Plot and discuss the results, and write a report that compares your results with those shown in Figure 1.

2. Collect at least 5 to 10 key references related to practical and theoretical experiences and experiments with Extreme Programming (alternatively for the approaches called "Agile Methodologies and Practices). Write a 20 page paper discussing the selected methodology, its philosophy in the context of limited resources, and its quantitative performance in practice (quality it offers – say in faults or defects per line of code, cost effectiveness – e.g., person-hours expended vs. the time to complete the project and the product quality, field failure rates or intensities, etc.). Use the references given in this article to start your literature search, but do not limit yourself to this list – use the Web to get the latest information.

3. Collect data from a real project. For each project phase, obtain data such as the number of faults, fault descriptions, when each fault was detected – its exposure time, time between problem reports, failure intensity, number of test cases, inspection data, personnel resources expended, etc. Analyze, tabulate and plot the testing efficiency and the cost-effectiveness information for the project and discuss it. You will probably need to do some additional reading to understand the metrics better. Use the references given in this article to start your literature search.

4. Search the literature for two sources of data other than the ones mentioned in this paper and perform testing efficiency analysis on the data. One source of data is at (http://www.srewg.org/). Discuss your results with respect to the results shown in this article (examples of effects to compare are "learning," process improvement within and between phases, etc.)

5. Obtain the pairwise testing papers and read them. Obtain the pairwise toolset from (http://www.srewg.org/). Use the approach to generate a suite of test cases to evaluate a networking switch that uses the following parameters (parameter name is followed by possible parameter values, full/half refers to duplex settings, TCP/UDP are protocols, size is in bytes, speed is in megabits per second). Parameters: Port1 (half/full/off), Port2 (half/full/off), Port3 (half/full/off), each port (1,2,3) can have speed set to (10, 100, automatic), each port (1,2,3) can service the following protocols (TCP, UDP), each port (1,2,3) can service packets of size (64, 512, 1500). Constraints: At least one port MUST be always set to full. Dependencies: All parameters can interact/depend on each other.

Answer the following: How many test cases would you have to run to cover all combinations of the parameters? Discuss the difference, if any, between the test suite obtained using pairwise testing and the “all-combinations of parameters” approach. Are there any obvious items PairTest is missing? If so, what are they and why are they important?

6. Read about operational profiles (use the references given in this article to start your literature search). Construct the top three levels of the operational profile for a typical web-browser (limit yourself to about 10 major functions/parameters with not more than 3-4 values per parameter). Construct an operational profile-based test suite for this application. Construct a pairwise test suite for this application. Compare and discuss the two as well as any other results that stem from this study.

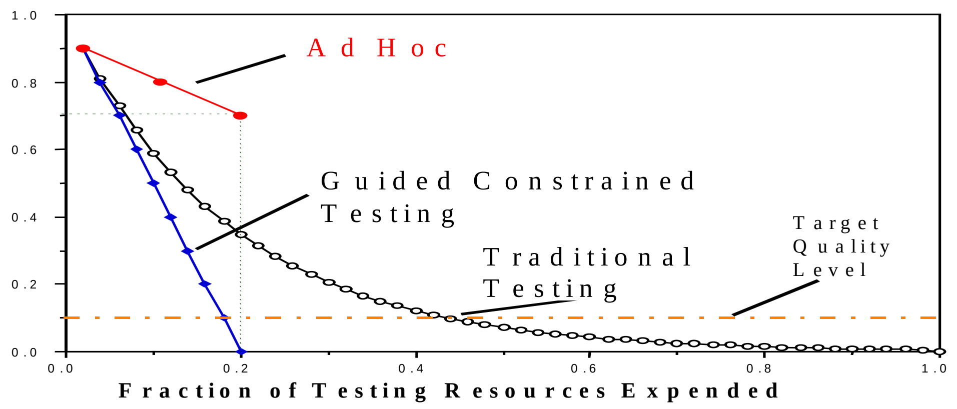


FIGURE 1 Fraction of Shipped Defects for “Ideal” Testing Based on Sampling With and Without Replacement.

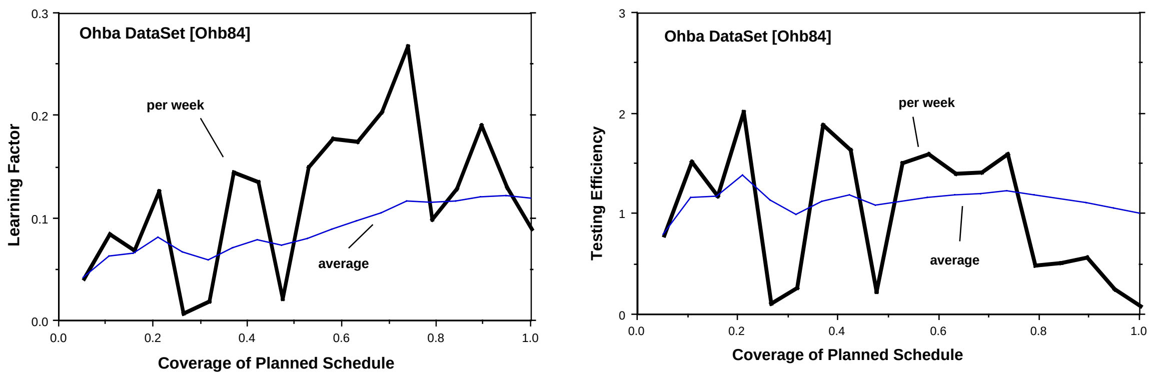
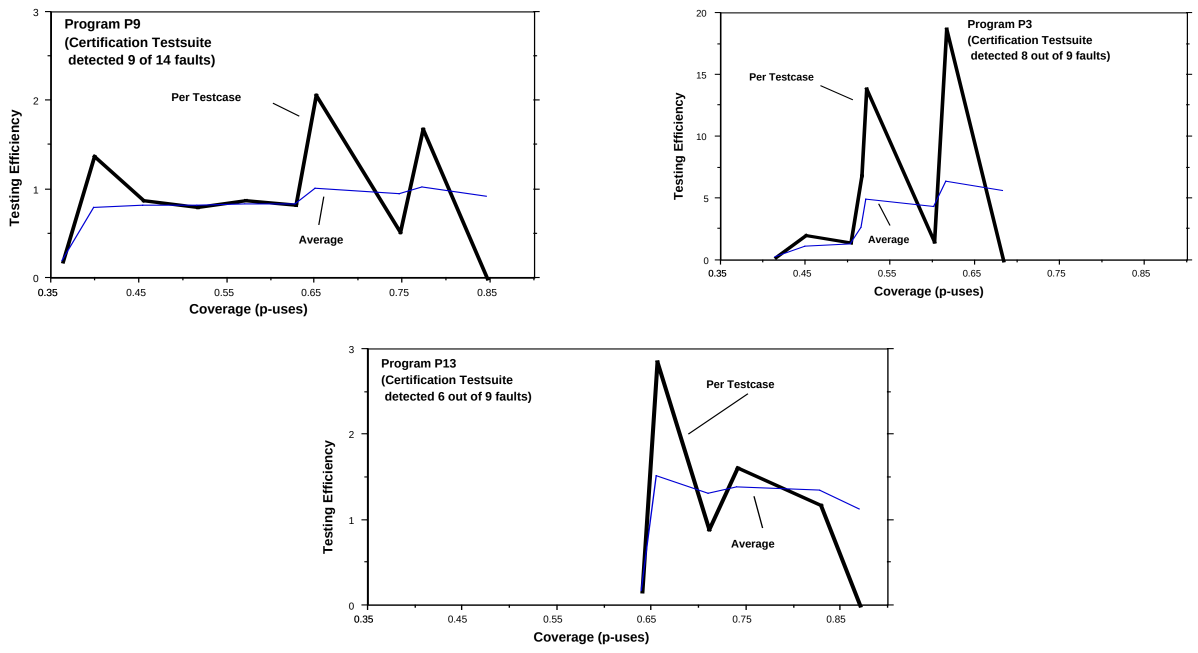
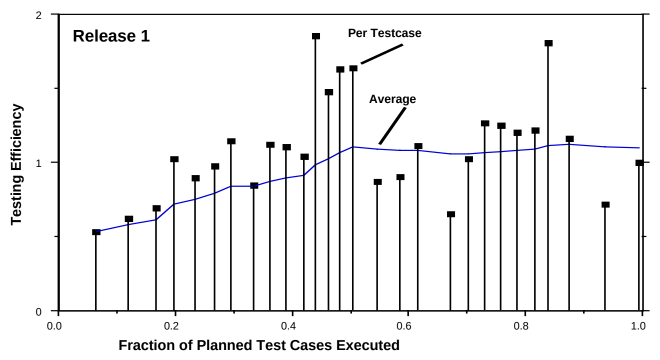


FIGURE 2. Left: Learning Factor (Hou, et al (1994)), Computed Using Table 4 Data from Ohba (1984); Right: Testing Efficiency, Equation (1), Computed Using Data from Ohba (1984), Table 4.



FIGUER 3. Experimental Testing Efficiency Function g_i for Program (DataSet2): P3, P9 and P13



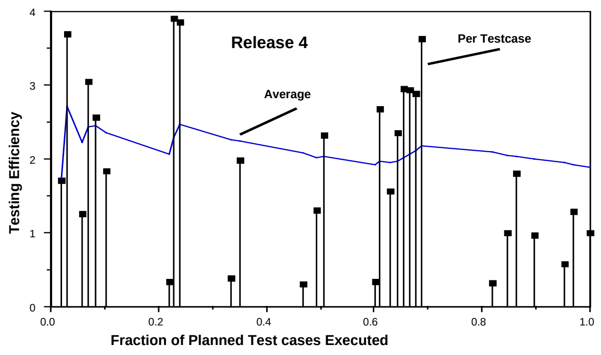
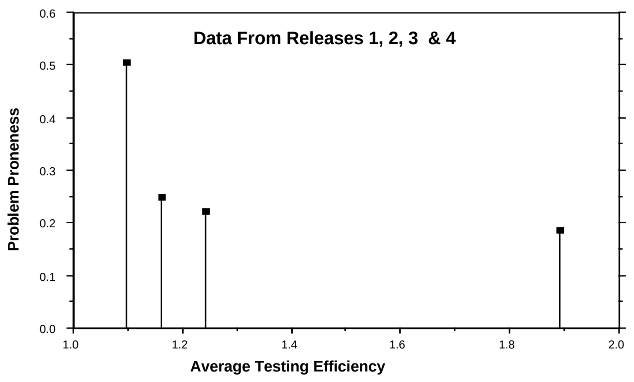
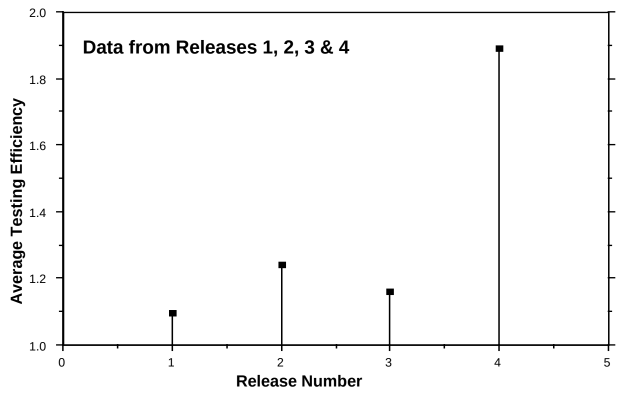


FIGURE 4. Experimental Testing Efficiency Function for Releases 1 and 4 (DataSet3).





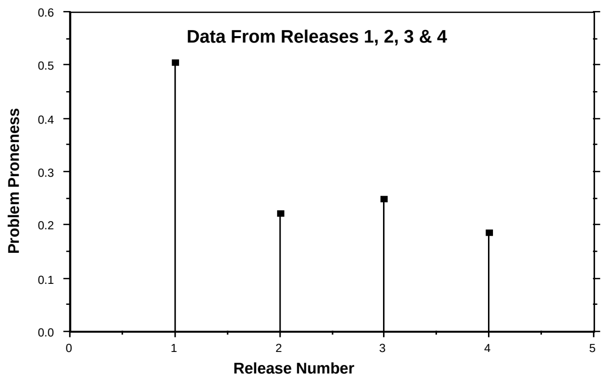


FIGURE 5. Average Testing Efficiency versus Problem Proneness, Release Number versus Average Testing Efficiency, and Release Number versus Problem Proneness for Releases 1-4 of (DataSet 3).

Footnotes:

1: Operational profile is a set of relative frequencies (or probabilities) of occurrence of disjoint software operations during its operational use. A detailed discussion of operational profile issues can be found in Musa, et al (1987), Musa (1993), and Musa (1999). A software-based system may have one or more operational profiles. Operational profiles are used to select verification, validation and test cases and direct development, testing and maintenance efforts towards the most frequently used or most risky components. Construction of an operational profile is preceded by definition of a customer profile, a user profile, a system mode profile, and a functional profile. The usual participants in this iterative process are system engineers, high-level designers, test planners, product planners, and marketing. The process starts during the requirements phase and continues until the system testing starts. Profiles are constructed by creating detailed hierarchical lists of customers, users, modes, functions and operations that the software needs to provide under each set of conditions. For each item it is necessary to estimate the probability of its occurrence (and possibly risk information) and thus provide a quantitative description of the profile. If usage is available as a rate (e.g., transactions per hour) it needs to be converted into probability. In discussing profiles, it is often helpful to use tables and graphs and annotate them with usage and criticality information.

2: "The Capability Maturity Model (CMM) for Software describes the principles and practices underlying software process maturity and is intended to help software organizations improve the maturity of their software processes in terms of an evolutionary path from an ad hoc reactive and chaotic process, to a mature, disciplined and proactive software process. The CMM is organized into five maturity levels: 1) Initial - The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics. 2) Repeatable - Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications. 3) - Defined. The software process for both management and engineering activities is documented, standardized, and integrated into a standard but still reactive software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software. 4) Managed - Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and proactively controlled. 5) Optimizing - Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies (http://www.sei.cmu.edu/cmm/cmm.sum.html)".

3: P-use (Predicate Data Use) is use of a variable in a predicate. A predicate is a logical expression which evaluates to TRUE or FALSE, normally to direct the execution path in the software program (code)" (Frankl and Weyuker (1988)).

4: Pair Programming (PP) is a style of programming in which two programmers work side by side at one computer, continuously collaborating on the same design, algorithm, code or test. It is important to note that pair programming includes all phases of the development process (design, debugging, testing, etc.) - not just coding. Experience shows that programmers can pair at any time during development, in particular when they are working on something that is complex. The more complex the task is, the greater the need for two brains. More recently, though, overwhelming anecdotal and qualitative evidence from industry indicates that programmers working in pairs perform substantially better than do two working alone. The rise in industrial support of the practice coincides with the rise in popularity of the Extreme Programming (XP) (Beck (2000), Auer and Miller (2001)) software development methodology, but ALSO with the rise in readily available collaborative computing systems and software (e.g., dual-head video cards). Pair programming is one of the twelve practices of the XP methodology (Williams and Kessler (2000)).

5: Object-oriented refers to a paradigm in which the problem domain is characterized as a set of objects that have specific attributes and behaviors. The objects are manipulated with a collection of functions (called methods, operations, or services) and communicate with one another through a messaging protocol. Objects are categorized into classes and subclasses. An object encapsulates both data and the processing that is applied to the data. This important characteristic enables classes of objects to be built and inherently leads to libraries of reusable classes and objects. Because reuse is a critically important attribute of modern software engineering, the object-oriented paradigm is attractive to many software development organizations. In addition, the software components derived using the object-oriented paradigm exhibit design characteristics (e.g., function independence, information hiding) that tend to be associated with high-quality software (Pressman (2001)).

List of Key Words