When Can We Test Less?[1]

Tim Menzies, Justin Di Stefano, Kareem Ammar
Lane Department of Computer Science
West Virginia University, USA
tim@menzies.us, justin@lostportal.net,
kammar@csee.wvu.edu

Kenneth McGill, Pat Callis
NASA IV&V Facility
Goddard Space Flight Center
kenneth.g.mcgill@ivv.nasa.gov
Patrick.E.Callis@ivv.nasa.gov

Robert (Mike) Chapman
Galaxy Global Corporation
Robert.M.Chapman@ivv.nasa.gov

John Davis
DN American
jfdavis@mindspring.com

Abstract

When it is impractical to rigorously assess all parts of complex systems, test engineers use defect detectors to focus their limited resources. In this article, we define some properties of an ideal defect detector and assess different methods of generating one. In the case study presented here, traditional methods of generating such detectors (e.g. reusing detectors from the literature, linear regression, model trees) were found to be inferior to those found via a PACE analysis.

1 Introduction

It may be too expensive to rigourously evaluate all parts of large software systems. When working against fixed deadlines, test engineers often use defect detectors to find areas of code that deserve more attention. For example, Figure 1 shows the *battlemap* generated by the McCabes QA tool. The darker modules in this display are predicted to be more error prone, using a method described later in this paper (see Figure 7). The intent of the battlemap is to tell test engineers what modules to "attack" first.

The danger with defect detectors is that they are imperfect, and if they have weaknesses then errors in what they test can slip by unnoticed. If the defect detector is flawed, then test engineers will focus on the wrong parts of the system; hence, they may miss important software errors.

We present here a new method for *assessing* and *generating* detectors. The new generation method is an extension to receiver operator characteristics (ROC) curves which we call PACE curves (PACE is short for Probability of detection and false alarm, Accuracy, Cost, Effort). The new *generation* method is called *ROCKY*. Unlike other generation methods, *ROCKY* takes into account PACE-type issues when finding detectors. We will show here that *ROCKY* can out-perform traditional detectors such as those used in the battlemap.

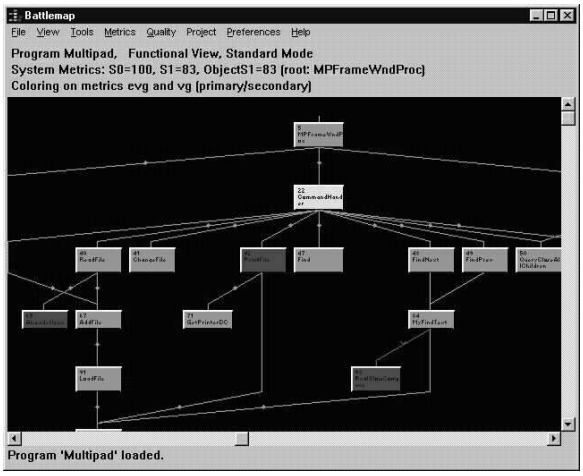


Figure 1. A battlemap from `http://www.mccabe.com/mccabe_qa.php`**.**

Our conclusions will not be some simplistic statement of the form (e.g.) "$v(g) \geq 10$ (cyclomatic complexity) is a good predictor for software defects". We strive to replace such simplistic statements with dialogue such as:

- What are the safety implications of an incorrect detector?
- Given limited resources, how much $effort$ can we al-

[1] Submission: 9th International Software Metrics Symposium, Sydney, Australia, September 3-5, 2003 `http://metric.cse.unsw.edu.au/Metrics2003/`. WP: 03/metrics03/whatmatters.tex.

locate to this V&V task?
- How can we alert our customers if the allocated $effort$ is inappropriate to this V&V task?
- The $costs$ of collecting the data required to run detector X is $\$Y$. Is detector X worth that expense?
- What detectors best manage the win/loss ratios associated with finding/missing an error in our application?

Based on case studies from two NASA systems, we will offer two findings. First, the "best" detector for an application depends on local factors. In order to find the "best" detector for a particular development, it is therefore *vital* that organizations maintain an *active data repository* on past and present software projects. This repository should include software product details *and* associated defect logs.

Second, some methods of generating detectors are too restrictive; i.e. they block a full discussion about V&V options. Ideally, we'd like a *range* of detector options so that test engineers can make arguments like (e.g.) "we can't work too well with the current level of funding but a 10% increase gets you a great increase in the chances of us funding an error". Such an argument is impossible unless the test engineer has a range of detectors around the current funding point. Hence, the best detector generators produce a wide *range* of options. This paper compares our own detector generator (*ROCKY*) with a set of other detector generation methods such as:

- a McCabe battlemap;
- simple linear regression;
- a Delphi method;
- regression tree learners [2] or
- state-of-the-art model tree learners [14].

It will be our claim that *ROCKY* is better than these other methods since its detectors have wider *range*.

We begin this paper with a discussion on why we have elected to study defect detectors. Next, we will define ROC curves then extend them to include *cost* and *effort* information. The resulting *PACE curves* will then be used to assess *ROCKY* against the other detector generators.

Before beginning, we pause for two caveats:

- In §5, we list the 11 observations used to make our conclusion. These conclusions are based on case studies with two NASA applications. All our conclusions must therefore be treated as tentative until we can check for those 11 observations in a larger number of applications.
- While the case studies of this paper use static code metrics, it would be a mistake to assume that PACE curves are *only* useful for McCabe metrics. On the contrary, we study PACE curves since they are a method of assessing detectors based on any form of metric. For example, in the near future we are looking at conducting a PACE analysis to comparatively evaluate static code metrics such as McCabe versus entropy-based metrics or metrics generated from a run-time pointer analysis of a "C" program.

2 Why Study Defect Detectors?

We study defect detectors for two reasons. First, we work for the NASA Independent Verification and Validation (IV&V) Facility which has a duty to assess and improve software practices within NASA. Second, in our experience, defect detectors are widely used within NASA and elsewhere.

The IV&V Facility in Fairmont, West Virginia is responsible for verifying that software developed or acquired to support NASA missions complies with the stated requirements. Additionally, the Facility validates that the software is suitable for its intended use. In short, the Facility ensures that the software is being developed properly, and that the right software is being developed or acquired.

As the sole entity with the responsibility for IV&V of all NASA mission software, the IV&V Facility is in a unique position to create and maintain a master repository of software metrics. Under this charter, the IV&V Facility reviews requirements, code, and test results from NASA's most critical projects; hence, many of the required metrics are collected as a matter of course. No other organization has insight into such a broad range of NASA projects. This affords the IV&V Facility an unequalled opportunity to research not only the early life cycle indicators of software quality, but other topics as well. Many large corporations have similar software metrics repositories; however, it is not always in their best interest to release data or results to the public. In the case of the IV&V Facility, the objective is to improve NASA's mission software regardless of the source. Once NASA projects agree to distribution, then sanitized data[1] would be made available to NASA, industry, and academia to support software development and research by other organizations. This is consistent with the IV&V Facilities research vision of "See more, learn more, tell more."

More specifically, defect detectors are worth studying since they are widely used in the software industry. For example, certain government organizations mandate the use of an independent consultancy team to verify and validate software. In the case of NASA, these IV&V teams work under strict budgetary constraints which can limit how much of a system can be tested rigorously. Criticality Assessment and Risk Analysis (CARA), a process developed by Titan Systems Corporation for NASA, is a mechanism for prioritizing modules so an informed decision can be made about what to cull [9]. In the case where too much code passes the

[1] Sanitized data has project-specific identifiers removed.

CARA cull, contractors often use static code metrics (e.g. a battlemap-like analysis) for a second-level cull.

NASA is not the only place where defect detectors are applied. Widely-used verification and validation (V&V) textbooks (e.g. [15]) advise a battlemap-like analysis to (e.g.) decide which modules are worthy of manual inspections. We know of several large government software contractors that won't review software modules *unless* tools like McCabes QA predict that they are fault prone. Hence, defect detectors have a major economic impact on a project when they may force programmers to rewrite code.

Critics of defect detectors might argue that the detectors based on simplistic syntactic measures might miss important semantic issues. Advocates of deep semantic techniques such as formal methods might argue that a global analysis of a program to find livelocks is more important than (e.g.) a battlemap-analysis, especially for mission-critical software. In reply, we note that automatic formal methods can be very expensive. These costs include the hiring of scarce PhD-level consultants with the required mathematical background; and the remodelling of the software into a mathematical format. Further, these automatic methods may take an exponential amount of time to execute, despite decades of optimization research [12]. Hence, even proponents of automatic formal methods use defect detectors to find software sections that are both small enough to be practical for automatic formal methods, yet critical enough to justify their formal modelling cost.

Debates on the merits of detect detectors notwithstanding, the issue of how to best focus limited resources is a daily discussion at the NASA IV&V facility. In response to this need, we are augmenting source code browsers with defect detectors. In the "Traffic Light Browser" of Figure 2, code modules are colored red, green, or yellow. Red modules are those with known faults. Yellow modules are those predicted to be fault prone by the defect detectors. Green modules are neither faulty nor predicted to be faulty. This browser is based on open-source tools (JAVA) and can be adapted to new languages faster and cheaper than (e.g.) the McCabes toolkit. However, no matter how promising something like the Traffic Light Browser appears to be, it is only as good as its defect detectors. This is the old problem of garbage in, garbage out: if the detectors are bad then the screens will be misleading. What is required is some methodology for the V&V of defect detectors.

The rest of this article defines such a methodology by augmenting ROC curves with $cost$ and $effort$ knowledge. These augmented curves will be generated using code metrics and defect data taken from KC2 and JM1, the two NASA C++ applications shown in Figure 3.

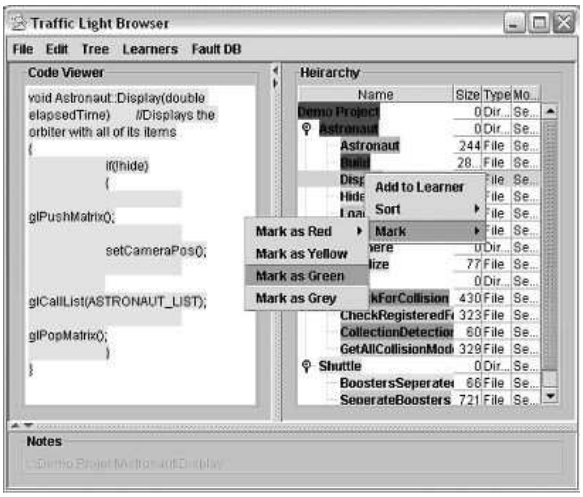


Figure 2. A "traffic light browser" for source code. Module source code is shown on the left and a package hierarchy is shown on the right. Packages are colored according to how strongly we believe they contain faults. Analysts can sort the package list by colors to focus on just the most fault prone modules.

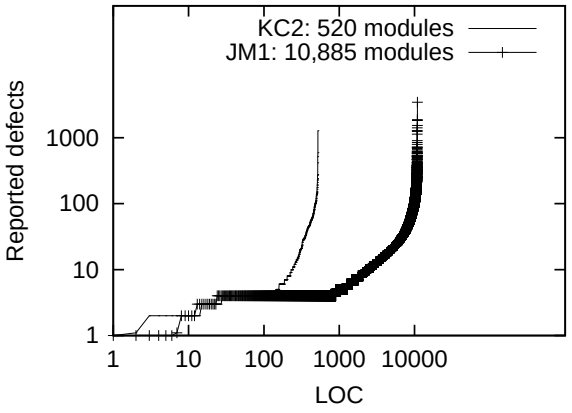


Figure 3. Defect and LOC data from two C++ NASA systems. Note that JM1 is much larger than KC2.

		signal present?	
		no	yes
signal detected?	no	A= true negative	B
	yes	C	D= true positive

Figure 4. A ROC sheet.

3 ROC Curves

Formally, a defect *detector* hunts for a *signal* that a software module is defect prone. Signal detection theory [6] offers *receiver operator characteristic* (ROC) curves that are an analysis method for assessing different detectors. ROC curves are widely used in various fields including assessing different clinical computing systems [1] and assessing different machine learning methods [13]. The central intuition of ROC curves is that different detectors can be assessed via how often they correctly or incorrectly respond to the presence or absence of a signal.

To draw a ROC curve, the ROC sheet of Figure 4 must be first completed. If a detector registers a signal, sometimes the signal is actually present (cell D) and sometimes it is absent (cell C). Alternatively, the detector may be silent when the signal is absent (cell A) or present (cell B).

Figure 4 lets us define the *accuracy*, or "Acc", of a detector as the number of true negatives and true positives seen over all events:

$$\text{accuracy} = Acc = \frac{A+D}{A+B+C+D}$$

If the detector registers a signal, there are two cases of interest. In one case, the detector has correctly recognized the signal. This *probability of detection*, or "PD", is the ratio of detected signals, true positives, to all signals:

$$\text{probability detection} = PD = \frac{D}{B+D}$$

In the other case, the *probability of a false alarm*, or "PF", is the ratio of detections when no signal was present to all non-signals:

$$\text{probability false alarm} = PF = \frac{C}{A+C}$$

In the ideal case, a detector has a high probability of detecting a genuine fault (PD) and a very low probability of false alarm (PF). This ideal case is very rare. Typically, engineers must trade-off between PF and PD. For example, a typical ROC curve is shown in Figure 5. Note that a high PD (indicated by the box along the left) is only achievable when PF (the box at the top of the graph) is also high.

The advantage of ROC curves is that they allow for cost-benefit trade-offs between different detectors. Defect detectors that fall into the *cost-adverse region* shown in Figure 5 have low probabilities of false alarms. Such defect detectors are best when the V&V budget is limited and the extra effort associated with chasing false alarms is unacceptable.

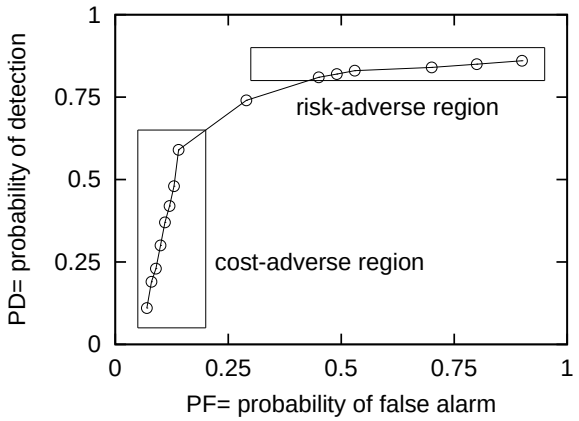


Figure 5. Regions of a typical ROC curve.

On the other hand, detectors that fall into the *risk-adverse region* of Figure 5 have high probabilities of detecting a signal. However, due to the usual relationship between PD and PF, this high probability comes at the cost of a high false alarm rate. Hence, detectors that fall into this region are best for safety-critical system where the cost of chasing false alarms is out-weighted by the cost of system failure.

3.1 Beyond Standard ROC Curves

It is important to fully consider the resource implications of a detector. V&V in general and IV&V in particular is usually a resource-bound activity. The disadvantage of standard ROC curves is that they are insensitive to certain resource implications. Hence, we introduce the twin concepts of $cost$ and $effort$:

- $Cost$ represents the resources required *before* a detector can be executed;
- $Effort$ represents the resources required *after* a detector reports a signal.

We saw above that selecting detectors may be a process of trading off PD vs PF. We shall see below that similar trade-offs are required between $cost$ and $effort$.

3.2 Effort

To better understand $effort$, recall that test engineers use detectors to focus their limited resources; e.g. if a module triggers a detector, then it will require a time-consuming

		module found in defect tracking log?	
		no	yes
signal detected?	no; i.e. $v(g) < 10$	A = 395 $LOC_A = 6816$	B = 67 $LOC_B = 3182$
	yes i.e. $v(g) \geq 10$	C = 19 $LOC_C = 1816$	D = 39 $LOC_D = 7443$

$$Acc = \quad accuracy = \quad 83\%$$
$$PF = \quad Prob.falseAlarm = \quad 5\%$$
$$PD = \quad Prop.detected = \quad 37\%$$
$$E = \quad effort = \quad 48\%$$

Figure 6. A ROC sheet assessing the detector $v(g) \geq 10$. Each cell {A,B,C,D} shows the number of modules, and the lines of code associated with those modules, that fall into each cell of this ROC sheet.

manual software inspection. Hence, when modules are selected by a detector, this adds some $effort$ to the V&V task. Based on a cost-model developed by one contractor here at NASA's IV&V facility, our analysis will assume that $effort$ is linearly proportional to lines of code[2]. Under that assumption, the $effort$ for a detector is what percentage of the lines of code in a system are selected by a detector.

In Figure 6, a defect tracking tool has been used as the oracle that reports if a defect was ever reported for a module. That figure shows the lines of code (LOC) of the modules from a NASA system that fall into each cell of ROC sheet using the detector "$v(g) \geq 10$". This detector is the standard McCabes detector for fault prone modules (and is explained further in Figure 7). The $effort$ associated with this detector is the amount of code that it selects for further V&V:

$$effort = E = \frac{LOC_C + LOC_D}{LOC_A + LOC_B + LOC_C + LOC_D}$$

3.3 $Cost$

To better understand $cost$, recall that a detector needs information before it can execute. A battlemap-style analysis requires the McCabes QA tool and site licenses for this tool may be expensive. Collecting information for other detectors may be much cheaper. For example, detectors based on simple lines of code counts can be implemented using very simple parsers than can be coded in a few minutes. We discuss below the various $costs$ of a spectrum of detectors.

[2]This assumption is very easily changed. The tool we use to generate PACE curves inputs a comma separated file where each row represents information from one module. The first column of this file stores the $effort$ value associated with evaluating this module. To repeat our analysis with a different effort model, we need only change this first column.

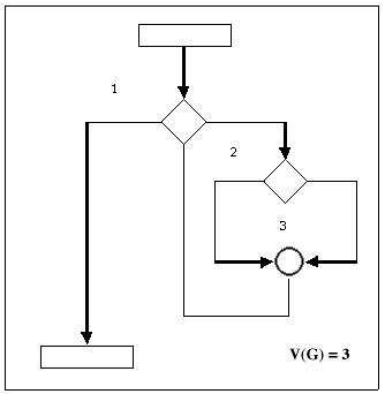
The McCabe metrics are a collection of four software metrics: essential complexity, cyclomatic complexity, design complexity and LOC, Lines of Code [8].
Essential Complexity, or $ev(G)$, is the extent to which a flowgraph can be "reduced" by decomposing all the subflowgraphs of G that are *D-structured primes*. Such *D-structured primes* are also sometimes referred to as "proper one-entry one-exit subflowgraphs" (for a more thorough discussion of D-primes, see [4]). $ev(G)$ is calculated by:

$$ev(G) = v(G) - m$$

where m is the number of subflowgraphs of G that are D-structured primes. [4]
Cyclomatic Complexity, or $v(G)$, measures the number of *linearly independent paths*. A set of paths is said to be *linearly independent* if no path in the set is a linear combination of any other paths in the set through a program's *flowgraph*. A flowgraph is a directed graph where each node corresponds to a program statement, and each arc indicates the flow of control from one statement to another. $v(G)$ is calculated by:

$$v(G) = e - n + 2$$

where G is a program's flowgraph, e is the number of arcs in the flowgraph, and n is the number of nodes in the flowgraph [4]. For example, the following simple flowgraph has a cyclomatic complexity of 3, since the graph has 6 arcs and 5 nodes ($v(G) = 6 - 5 + 2 = 3$).



The standard McCabes rules, are used to identify fault-prone modules in the battlemap analysis of Figure 1. If $v(g) \geq 10$, the battlemap shows that module in a dark color.
Design Complexity, or $iv(G)$, is the cyclomatic complexity of a module's reduced flowgraph. The flowgraph, G, of a module is reduced to eliminate any complexity which does not influence the interrelationship between design modules. This complexity measurement reflects the modules calling patterns to its immediate subordinate modules [8].

Figure 7. The McCabes static metrics

In order to make $cost$ trade-offs, we need more details on the cost of various detectors. The detectors used in this study have the $costs$ shown below. These costs are listed in *increasing* order of magnitude; i.e.

$$cost_1 < cost_2 < cost_3 < cost_4 < cost_5 < cost_6$$

$Cost_1$- *using just basic lines of code*: Consider a lines of code (LOC) measure for a software module that counts all lines between module start and the last closing bracket. This simplistic LOC counter would be trivial to implement

Halstead argued that modules that are hard to read are more likely to be fault prone [5]. Halstead estimates reading complexity by counting the number of concepts in a module. Primitive concepts are:

$$\begin{aligned} \mu_1 &= \text{number of unique operators} \\ \mu_2 &= \text{number of unique operands} \\ N_1 &= \text{total occurrences of operators} \\ N_2 &= \text{total occurrences of operands} \end{aligned}$$

For example, the expression `return max(w+x,x+y)` has $N_1 = 4$ operators $(return, max, +, +)$, $N_2 = 4$ operands (w, x, x, y), $\mu_1 = 3$ unique operators $(return, max, +)$, and $\mu_2 = 3$ unique operands (w, x, y).

Using these measurements, Halstead defined the *length* of a program P as: $N = N_1 + N_2$ and the vocabulary of P to be $\mu = \mu_1 + \mu_2$.

From these basic measure, Halstead derives other metrics shown in Figure 9.

Figure 8. The basic Halstead metrics.

for any number of programming languages.

$Cost_2$- *using just basic Halstead*: Another measure of code complexity advocated by Halstead [5] is the number of operators and operands in a module (for more on the Halstead metrics, see Figure 8). A simple tokenizer, such as available with the standard JAVA distribution[3], would suffice for finding all the operators and operands. Given a hash table of known keywords in a particular language, and functions defined in the current program, it would be fast to compute the operators and operands.

$Cost_3$- *using just derived Halstead*: Given operators and operand frequencies, Halstead computes a range of metrics including T, the Halstead estimate for how long it would take a human to read a module. The calculation of these derived metrics is shown in Figure 9.

$Cost_4$- *using McCabe*: The McCabes metrics may be quite costly to collect since expensive compiler tools are required to compute the flowgraphs. One advantage of buying McCabes is that the above attributes are all collected automatically by this tool.

$Cost_5$- *using Delhi attributes*: "Delphi" is a fancy name for "ask an expert". The $cost$ of the delphi method may be very high. Firstly, experts may base their assessment on the output of expensive commercial toolkits. Secondly, it is hard to find such experienced test engineers, then convince them to reveal their trade secrets.

$Cost_6$- *using combinations of the above*: All our examples to date assumed *singleton* detectors; i.e. rules that only test one attribute. *Group* detectors use sets of attributes, combined in some manner. The $cost$ of such groups is the cost of collecting the information needed by members of this group. Note that these $costs$ don't just sum since, sometimes, spending money of one metric means that other metrics are free to collect. For example, if a group uses two basic Halstead measures and one McCabe measure, then the cost of that group is only the $cost$ of buying one McCabe license (since that one McCabe's license can collect all the McCcabe measures and the Halsteads as well).

[3] `StringTokenizer`, see `java.sun.com/products/jdk/1.2/docs/api/java/util/StringTokenizer.html`

Apart from the basic metrics shown in Figure 8, Halstead also derives further metrics. These other metrics use the following values:

$$\begin{aligned} \mu_1^* &= \text{potential operator count} \\ \mu_2^* &= \text{potential operand count} \end{aligned}$$

μ_1^* and μ_2^* as the *minimum* possible number of operators and operands for a module. This minimum number would occur in a (potentially fictional) language in which the required operation already existed, possibly as a subroutine, function, or procedure. In such a case, $\mu_1^* = 2$, since at least two operators must appear for any function; one for the name of the function, and one to serve as an assignment or grouping symbol. μ_2^* represents the number of parameters, without repetition, which would need to be passed to the function or procedure.

According to Halstead, the *volume* of P, akin to the number of mental comparisons needed to write a program of length N, is:

$$V = N * log_2 \mu$$

V^* is the volume of the minimal size implementation of P.

$$V^* = (2 + \mu_2{}^*) log_2 (2 + \mu_2{}^*)$$

The *program level* of a program P with volume V is:

$$L = V^*/V$$

The inverse of level is *difficulty*:

$$D = 1/L$$

According to Halstead's theory, we can calculate an estimate $\hat{L}$ of L as:

$$\hat{L} = 1/D = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$$

The intelligence content of a program, I, is:

$$I = \hat{L} * V$$

The effort required to generate P is given by:

$$E = \frac{V}{\hat{L}} = \frac{\mu_1 N_2 N log_2 \mu}{2\mu_2}$$

where the unit of measurement E is elementary mental discriminations needed to understand P. The required programming time T for a program of effort E is:

$$T = E/18 seconds$$

Figure 9. The derived Halstead metrics.

4 Generating Detectors

The claim of this article is that when different detectors are assessed by a PACE analysis (Probability of detection and false alarm, Accuracy, Cost, Effort), then the *ROCKY* method out-performs other, more standard, methods such as *traditional*, *delphi*, *linear regression*, and *model trees*. Those standard methods often combine many attributes to

produce a group detector. Before contrasting PACE with these standard methods, we must define them. For an explanation of the attributes used in the following detectors, see Figures 7, 8 and 9.

The *traditional* method of generating detectors is to use the thresholds used by the McCabes IQ battlemap. of $v(g) \geq 10$ and $iv(g) \geq 4$. We will study here three such detectors:

$$v(g) \geq 10 \quad (1)$$
$$iv(g) \geq 4 \quad (2)$$
$$iv(g) \geq 4 \vee v(g) \geq 10 \quad (3)$$

As explained above, the *Delphi* method is a fancy name for "ask an expert". The NASA IV&V facility has many such expert test engineers and so we can access several *Delphi* detectors.

$$v(g) > 20 \quad (4)$$
$$ev(g) > 8 \quad (5)$$
$$v(g) > 20 \vee ev(g) > 8 \quad (6)$$
$$ev(g) > 7 \quad (7)$$
$$LOC > 118 \quad (8)$$
$$LOC > 118 \vee ev(g) > 7 \quad (9)$$

Detectors 4, 5 and 6 come from an analysis by Chapman and Solomon [3]. Detectors 7, 8 and 9 come from another Menzies et.al. [11]. Another, more elaborate *Delphi* detector is:

$$x + y + z \geq 30 \quad (10)$$

where

$$\begin{aligned} x &= if\ v(g) > 10\ then\ v(g)\ else\ 0 \\ y &= if\ ev(g) > 4\ then\ ev(g) * 4\ else\ 0 \\ z &= if\ LOC > 200\ then\ 50\ else\ 0 \end{aligned}$$

Linear regression is an alternate method to $delphi$ that does not require expensive human expertise. Linear regression is a standard statistical method that fits a straight line to a set of points. The line offers a set of predicted values p_i. If the points are scattered, then a single regression line can't pass through each point. The distance from these predicted values to the actual values a_i is a measure of the error associated with that line. Linear regression packages search for lines that minimize that error. This error may be expressed in terms of the *correlation* between the actual and predicted values. This correlation ranges from 1 (for perfectly correlated) to -1 (for perfectly negatively correlated) and is calculated as follows:

$$\begin{aligned} p_i &= \text{predicted value} \\ a_i &= \text{actual value} \\ n &= \text{number of observations} \\ \overline{x} &= \text{mean of } n \text{ observations} \\ S_{PA} &= \frac{\sum_i (p_i - \overline{p})(a_i - \overline{a})}{n-1} \\ S_p &= \frac{\sum_i (p_i - \overline{p})^2}{n-1} \\ S_a &= \frac{\sum_i (a_i - \overline{a})^2}{n-1} \\ correlation &= \frac{S_{PA}}{\sqrt{S_p S_a}} \end{aligned}$$

A numeric prediction p_i for defects generated by linear regression becomes a defect detector by testing for $p_i \geq 1$. Two such detectors learnt by linear regression from the KC2 dataset are shown below. Detector 11 was learnt using only the LOC measure and detector 12 was learn using just the basic Halstead measures[4].

$$(0.0164 + 0.0114 LOC) \geq 1 \quad (11)$$
$$0.128 + 0.0147 I + 0.011 \mu_1 - 0.0243 \mu_2 - 0.0111 N_1 + 0.0282 N_2 \geq 1 \quad (12)$$

A drawback with linear regression is that the *same* line is fitted through all points. That is, linear regression assumed that all the data comes from a single simple linear distribution. Where this is not true, it may be better to divide the space into different regions and then make a different decision about each region. Two techniques for doing so are *regression trees* and *model trees*.

Regression tree learners such as CART [2] generate decision trees whose leaves are numeric values. Numeric values are calculated from these trees by starting at the root, and following the branches that are relevant to the current situation. Regression tree learning was applied to the KC2 data set, but the resulting correlations were far worse than either those found by linear regression or the model tree learning technique described below, and so regression tree learning will not be discussed further in this article.

A significant improvement of the regression tree technique is the M5-Prime model tree learner [14, 16]. A model tree is a decision tree with different linear regression equations at each leaf. Model trees are used like linear regression to generate defect detectors: if the prediction is p_i,

[4]Detectors learnt from derived Halstead or McCabes are not shown here since, using those attributes, model tree learning found predictors with higher correlation.

then the detector is triggered when $p_i \geq 1$. Using this method, the following detectors were generated from KC2 using nearly all available attributes (basic and derived Halstead McCabes, but not LOC):

$$\left(\begin{array}{lll} if & N_2 \leq 49.5 & \\ then & \left\{\begin{array}{lll} if & I <= 24.7 & \\ then & 0.0375 & \\ else & 0.284 & \end{array}\right. & \\ else & \left\{\begin{array}{lll} if & N_2 \leq 142 & \\ then & 1.06 & \\ else & \left\{\begin{array}{l} 0.663- \\ 0.164 * ev(g)+ \\ 0.0128 * T \end{array}\right. & \end{array}\right. & \end{array}\right) \geq 1 \quad (13)$$

A simpler model tree detector was learnt by M5-Prime from KC2 using just the McCabes metrics:

$$(0.0376 + 0.125 * v(g) - 0.0852 * ev(g)) \geq 1 \quad (14)$$

Model tree learning is a state-of-the-art machine learning technique. A much simpler technique, is our *ROCKY* detector learner that exhaustively explores all singleton rules of the form

$$attribute \geq threshold$$

Here, *attribute* is every numeric attribute present in a dataset and *thresehold* is found as follows. Every numeric attribute is assumed to come from a gaussian distribution. $Threshold$s are then selected corresponding to equal areas under that distribution. For example, in one of the datasets we examine, $v(g)$ had a mean of $\mu = 4.9$ and a standard deviation of $\sigma = 11$. If this Gaussian is converted to a unit Gaussian (by subtracting the mean and dividing by the standard deviation), then standard Z-tables could be used to calculate a $v(g)\ threshold$ value of 7.65 could be found as follows:

$$\begin{array}{rcl} area & = & 0.6 \text{ (just for example)} \\ Z^{-1}(area) & = & \frac{v(g)-\mu}{\sigma} \\ Z^{-1}(area) & \approx & 0.25 \\ \therefore v(g).threshold(area) & \approx & 7.65 \end{array}$$

ROCKY generates one detector

$$X \geq X.threshold(area) \quad (15)$$

for the range

$$\begin{array}{rl} attribute \in & \{LOC, v(g), ev(g), iv(g), N, V, \\ & L, D, , E, B, T, \mu_1, \mu_2, N_1, N_2\} \\ area \in & \{0.05, 0.1, 0.15, \ldots 0.9, 0.95\} \end{array}$$

That is, the detector 15 is really hundreds of detectors

ROCKY seems to be a naive way to generate detectors. This learner can only generate singleton detectors while the other methods described above can generate group rules. Also, *ROCKY*'s use of a Gaussian assumption may be inappropriate for certain data distributions (e.g. highly-skewed data). Nevertheless, *ROCKY* is our preferred method of generating detectors. We will see below that *ROCKY* generates a wider $range$ of detectors that any of the other methods described above.

5 PACE Analysis

The previous section described 14 specific detectors plus hundreds of *ROCKY*-based detectors. Those detectors are assessed below using PACE curves. This section must be read carefully. The conclusions described here are drawn only from two data sets. The external validity of the following conclusions depends on how often we see the following **observations** we describe in other datasets.

The PACE curves for KC2 and JM1 are shown in Figure 10. Recall that JM1 is a much larger data set than KC2 (10,885 modules versus 520 modules) and so there is more *noise* in the JM1 plots (e.g. the PF plot in KC2 seems smoother than the PF plot in JM1).

The top half of each plot shows the detector number. For example, the top row of each plot is labelled *Delphi[4,5,6,7,8,9,10]* which denotes that detectors 4 through 10 of §4 where used to generate these dots. By looking vertically down from the marks on the *Delphi[4,5,6,7,8,9,10]*, we can infer (e.g.) the $effortrange$ associated with these detectors. In doing so, we see that:

> **Observation 1** *All the* Delphi *detectors lie in a narrow $effortrange$; i.e. 30% to 50%.*

If the available budget for V&V fell outside this $effortrange$, then these *Delphi* detectors would be inappropriate.

The linear regression detectors (marked *LR[11,12]* fare even worse than the *Delphi* predictors. In these two data sets:

> **Observation 2** *Linear regression detectors only service a very narrow $rangeeffort$ (approximately 50% to 55%).*

Figure 10 also shows us that:

> **Observation 3** Model trees *and* traditional *detectors (marked* Model trees[13,14] *and* Traditional[1,2,3], *respectively) service a wider $range$ of $effort$ than linear regression but their coverage within that range is very poor.*

Hence, there may be V&V $efforts$ not covered by these detectors.

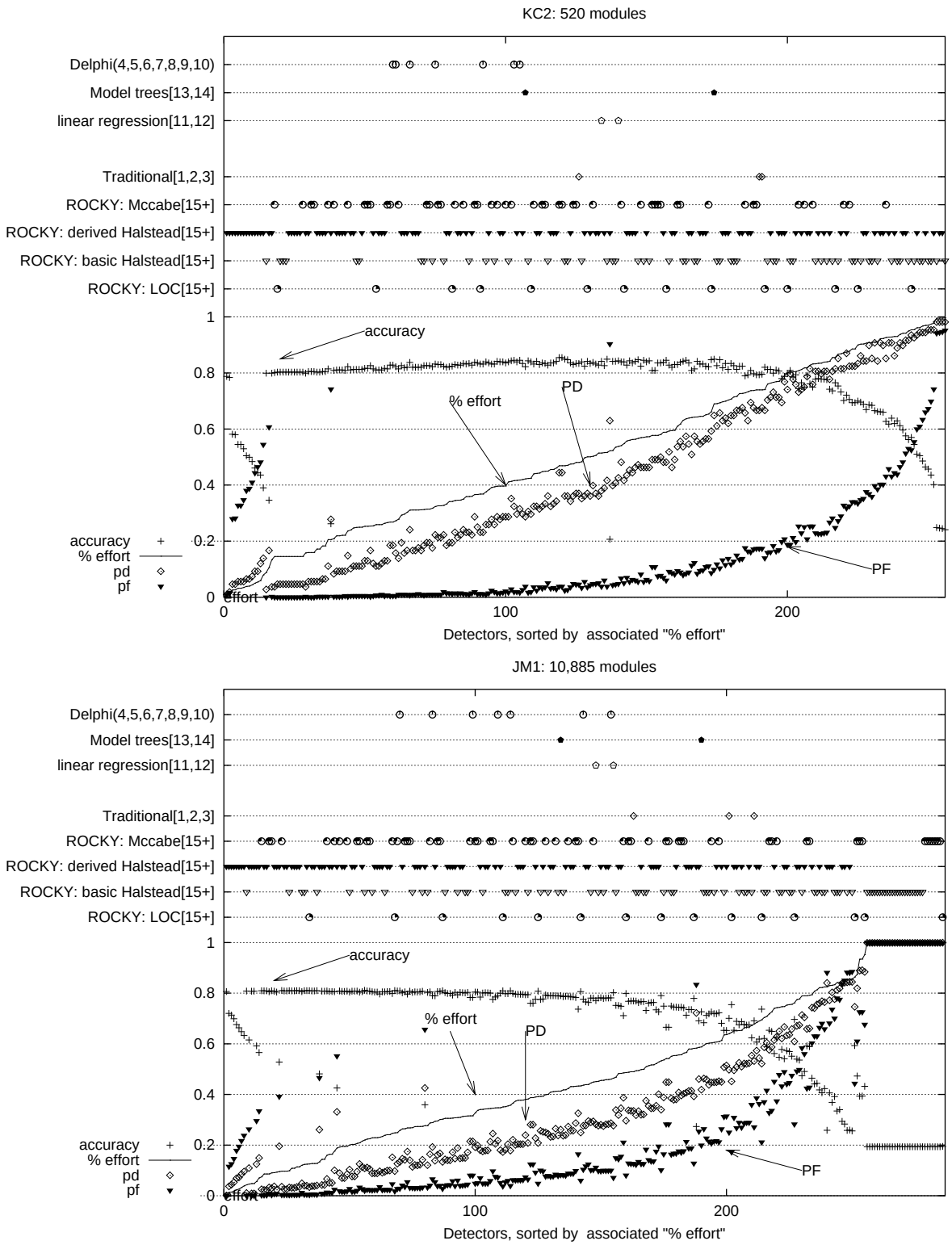


Figure 10. PACE curves from two NASA applications. Numbers in square brackets on the y-axis denote detector numbers from §4.

In the case of model trees and linear regression, one reason for their narrow range might be that our analysis inappropriately constrained their effectiveness. In this study, detectors were generated only on the model trees and linear regression runs that generated the highest correlations. If all the model trees and regression results had been used, and not just the ones with highest correlation, then perhaps a wider range of detectors would have been found. However, we are not motivated to perform such a study since, except in the high $effort$ region, the *accuracy* curves are essentially flat. That is:

> **Observation 4** *Accuracy is not correlated to effort, probability of detection, or probability of false alarms.*

Machine learners are typically assessed in terms of average accuracy seen in 10-way cross validation trial[5]. Hence, if certain features change dramatically (such as PF and PD) while accuracy remains constant strongly means that accuracy is a useful predictor for those other features. As seen in Figure 10, focusing merely on accuracy can hide other important features of the learnt theory such as a low probability of detection. For a more detailed critique on the drawbacks of assessing learners merely on accuracy, see [13].

The detectors marked *[15+]* were generated from *ROCKY* using subsets of of attributes with different $costs$. Recall from §3.3 that the McCabes metrics were expensive, compared to simpler metrics such as LOC. The advantage of the McCabes metrics are clear from Figure 10: they cover a very wide $range$ of $effort$. The disadvantage of McCabes, at least in these two data sets, is that:

> **Observation 5** *Cheaper metrics cover the same range as McCabes.*

For example, the *derived Halstead* metrics seem to cover just as much, or more, as McCabes[6].

As we move to cheaper metrics such as *basic Halstead* and *LOC*, we see that:

> **Observation 6** *The $range$ for the cheapest detectors is very wide, but the coverage within those $range$s is patchy.*

This is especially true for the cheapest detectors generated from just LOC.

[5]Ten times, the learning is performed on a random selection of 90% of the data, then tested on the 10% not seen during training.

[6]In conversations with NASA IV&V test engineers, it has been argued that McCabes metrics predict for more that just defects. For example, as essential complexity $ev(g)$ grows, the number of pathways that need testing in a module increases. Modules with many pathways, it is claimed, are hard to understand and hard to maintain. At this time, we have no data to accept or refute this claim. However, if ever detectors for expensive maintainability were proposed, and historical records of defect effort could be accessed, then PACE curves could be used to assess these maintainability detectors.

Based on the above observations concerning $range$, we say that:

> **Observation 7** ROCKY*'s singleton detectors out-perform group detectors learnt from other, much more complex, methods.*

Assuming that $range$ is desirable (since it permits a discussion of a wider range of options), then the singleton detectors shown as *[15+]* out-perform the group detectors proposed by Delphi, linear regression, or model trees. This observation is the subject of active research at this site. We are experimenting with a new kind of learner. This learner incorporates issues of $effort$,$cost$, PD, and PF not the inner loop of all its learning. We speculate that such a learner might be able to generate good group detectors. However, at this time, we don't have any results that allow us comment on this speculation.

Some other interesting patterns can be seen in these PACE curves. For example:

> **Observation 8** *The probability of false alarms is usually low (less than 20%).*

This is finding suggests that resource-bound V&V can still be effective. Suppose that budgetary constraints mean that only (e.g.) 40% of the code can be read. In both KC2 and JM1, the detectors found at $effort = 0.4$ have a PF of less than 10%. An aggressive test engineer might summarize this position to her project manager as follows:

> *Ok, so you've only given me time to read 40% of the code. That means (in the case of KC1) that the probability of detecting an error is, at best, 35%. That's fine: if that's the available budget then so be it. BUT if I* do *detect something then you have to check it out, since my probability of false alarm is so low (less than 1 in 20).*

Another feature of these results is:

> **Observation 9** *The upper bound on the probability of detection is $effort$.*

That is, the more you analyze, the higher the chances of finding an error. This, in itself is no great breakthrough. However, what also seems to be the case is that:

> **Observation 10** *The probability of detection is (approximately) linearly correlated with $effort$.*

So the good news is that higher software quality need not be *exponentially* more expensive. The bad news is that:

> **Observation 11** *There is an upper-bound on the effectiveness of these metrics, and that upper bound is well below $PD = 1$.*

In KC2 and JM1, the probability of false alarm starts climbing rapidly after the probability of detection rises over 80% and 60% respectively. That is, there are upper limits on software quality improvements based on simple static source code metrics such as LOC, Halstead, or McCabes. For $PD \geq 0.6$ (approx), test engineers will have to explore other, more elaborate (and probably more expensive) defect detection methods. For a cost-benefit analysis of other methods for safety-critical systems, see [7, 10].

6 Business Decisions and PACE

PACE curves allow for the selection of detectors based on criteria generated from the current business scenario. To illustrate that selection process, we will consider two scenarios:

Scenario #1: A NASA IV&V team has been given fifteen days to analyze 100,000 LOC of non-critical software to be used in an unmanned routine space mission. That team uses a home-grown manual inspection process to assess software. That process takes, on average, 1 minute per eight LOC and can be conducted up to six hours per day.

Scenario #2: A NASA IV&V test engineer is responsible for evaluating a safety critical piece of software in a manned spacecraft which regularly launches in a blaze of publicity with all senior NASA management watching.

To apply PACE to these scenarios, we review the questions listed in the introduction.
What are the safety implications of an incorrect detector?

- Clearly, the safety implications of an incorrect detector are more dire in Scenario #2 than Scenario #1.

Given limited resources, how much $effort$ can we allocate to this V&V task?

- In $Scenario\ \#1$, there is time to inspect $8(LOCperminute) * 60(minutesperhour) * 6(hoursperday) * 15(days) = 43200LOC$ which implies a maximum $effort$ of $\frac{43200}{100000} \approx 43\%$ of the system. In $Scenario\ \#2$, no such limits are known.

The $costs$ of collecting the data required to run detector X is $\$Y$. Is detector X worth that expense?

- In Scenario #2, the cost of (e.g.) a McCabes license, while expensive, is negligible compared to the cost associated with loss of life or mission. However, in Scenario #1, there may be a case for looking at less expensive detectors since Figure 10 shows that such cheaper detectors can still be very effective.

How can we alert our customers if the allocated $effort$ is inappropriate to this V&V task? As mentioned above, **Scenario #1**'s $effort$ allocation implies that the maximum effort was 43%. In KC2, the PD, PF associated with this $effort$ is approximately 35% to 40%. If the safety requirements of that mission imply a detection probability higher than (e.g.) 50%, then clearly the resource allocation to Scenario #1 is inappropriate.

In **Scenario #2**, the failures of this system could be life threatening and, given the high visibility of this project, possible career limiting as well for the responsible test engineer. Hence, high PDs are required. High PDs incur the cost of high false alarm rates: in KC2, $PD = 1.0$ results in a $PF \approx 0.75$ and in JM1, $PF = PD$ at about the 0.8 level. Hence, if funding is not available to chase high false alarm rates or a PD greater than 0.75, then there exists no resource allocation to Scenario #2 that is demonstrably useful according to Figure 10. In this case, other defect detection methods are required (see [7, 10]).
What detectors best manage the win/loss ratios associated with finding/missing an error in our application? Depending on how the above questions are answered, the KC2 and JM1 projects may have moved away from defect detectors based on static code measures. If not, then the detectors properties closest the desired $effort$, PD, and PF could be read from Figure 10. The appropriate detector could then be read from the ascii report tables of *ROCKY*.

7 Conclusion

Defect detectors are used widely in software engineering to contain the cost of software code reviews. Traditional methods of generating detectors have been shown to have less $range$ than the detectors generated by *ROCKY*. Since a wide detector $range$ enables a wide ranging discussion of V&V options, we therefore recommend *ROCKY* as the preferred generator.

Our results showed that certain traditional detectors may be inferior for certain software projects. Hence, there is a pressing need for organizations to collect the data required to build their own detectors. Here at the NASA IV&V Facility, we are therefore building a centralized repository of code metrics and defect reports from a wide range of NASA projects. We anticipate that this repository will serve three important functions:

Audit A PACE analysis of the repository data could raise an alert if the detectors used for V&V were inappropriate.

Generate *ROCKY* could use the same data to generate detectors tuned to particular projects.

Reuse Early in the life-cycle, when there aren't enough defect reports for a PACE analysis, V&V personnel could

search the repository for detectors from prior projects that were similar to the current project. As defect data for the current project is collected, those reused detectors must be audited.

Our future direction is to see which, if any, of the 11 **observations** in §5 can be made in other software projects. To this end, we are working with the NASA organization to gain access to more code metrics and defect data sets. Our goal in 2003 is a PACE analysis on five to twenty different software projects. To this end, we would welcome data sets from non-NASA sources.

For more details on the IV&V metrics repository, contact Pat Callis (`Patrick.E.Callis@ivv.nasa.gov`) or Mike Chapman (`Robert.M.Chapman@ivv.nasa.gov`). For an AWK-script to generate PACE curves, contact Tim Menzies (`tim@menzies.us`).

Acknowledgements

The editorial assistance of Lisa Montgomery was both both timely and useful and we are most grateful. This research was conducted at West Virginia University under NASA contract NCC2-0979 and NCC5-685. The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

References

[1] K. P. Adlassnig and W. Scheithauer. Performance evaluation of medical expert systems using roc curves. *Computers and Biomedical Research*, 22(4):297–313, 1989.

[2] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984.

[3] M. Chapman and D. Solomon. The relationship of cyclomatic complexity, essential complexity and error rates, 2002. Proceedings of the NASA Software Assurance Symposium, Coolfont Resort and Conference Center in Berkley Springs, West Virginia. Available from `http://www.ivv.nasa.gov/business/research/osmasas/conclusion2002/Mike_C%hapman_The_Relationship_of_Cyclomatic_Complexity_Essential_Complexity_and_Erro%r_Rates.ppt`.

[4] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach (second edition)*. International Thompson Press, 1995.

[5] M. Halstead. *Elements of Software Science*. Elsevier, 1977.

[6] D. Heeger. Signal detection theory, 1998. Available from `http://white.stanford.edu/~heeger/sdt/sdt.html`.

[7] M. Lowry, M. Boyd, and D. Kulkarni. Towards a theory for integration of mathematical verification and empirical testing. In *Proceedings, ASE'98: Automated Software Engineering*, pages 322–331, 1998.

[8] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[9] D. McCaugherty. Criticality analysis and risk assessment (cara). presentation by Averstar Inc., February 1998. See also, `http://www.ivv.nasa.gov/about/tutorial/sld025.htm`.

[10] T. Menzies and B. Cukic. How many tests are enough? In S. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering, Volume II*, 2002. Available from `http://tim.menzies.com/pdf/00ntests.pdf`.

[11] T. Menzies, J. S. DiStefeno, M. Chapman, and K. Mcgill. Metrics that matter. In *27th NASA SEL workshop on Software Engineering*, 2002. Available from `http://tim.menzies.com/pdf/02metrics.pdf`.

[12] T. Menzies, J. Powell, and M. E. Houle. Fast formal analysis of requirements via 'topoi diagrams'. In *ICSE 2001*, 2001. Available from `http://tim.menzies.com/pdf/00fastre.pdf`.

[13] F. Provost, T. Fawcett, and R. Kohavi. The case against accuracy estimation for comparing induction algorithms. In *Proc. 15th International Conf. on Machine Learning*, pages 445–453. Morgan Kaufmann, San Francisco, CA, 1998. Available from `htpp://citeseer.nj.nec.com/provost98case.html`.

[14] J. R. Quinlan. Learning with Continuous Classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992. Available from `http://citeseer.nj.nec.com/quinlan92learning.html`.

[15] S. Rakitin. *Software Verification and Validation for Practitioners and Managers, Second Edition*. Artech House, 2001.

[16] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.