Analytical and Empirical Evaluation of Software Reuse Metrics *

Prem Devanbu, Sakke Karstu, Walcelio Melo and William Thomas

To appear in the Proc. of the 18th Int'l Conf. on Software Engineering, Berlin, Germany, 1996. IEEE Press. [Also available as a Technical Report, University of Maryland, Department of Computer Science, College Park, MD, 20770. July 1995. CS-TR-3505. (Cross referenced as UMIACS-95-82)]

Abstract

How much can be saved by using existing software components when developing new software systems? With the increasing adoption of reuse methods and technologies, this question becomes critical. However, directly tracking the actual cost savings due to reuse is difficult. A worthy goal would be to develop a method of measuring the savings *indirectly* by analyzing the code for reuse of components. The focus of this paper is to evaluate how well several published software reuse metrics measure the "time, money and quality" benefits of software reuse. We conduct this evaluation both *analytically* and *empirically*. On the *analytic* front, we introduce some properties that should arguably hold of any measure of "time, money and quality" benefit due to reuse. We assess several existing software reuse metrics using these properties. *Empirically*, we constructed a toolset (using GEN++) to gather data on all published reuse metrics from C++ code; then, using some productivity and quality data from "nearly replicated" student projects at the University of Maryland, we evaluate the relationship the known metrics and the process data. Our empirical study sheds some light the applicability of our different analytic properties, and has raised some practical issues to be addressed as we undertake broader study of reuse metrics in industrial projects.

1 Introduction

Software reuse is considered to be one of the most promising approaches for increasing productivity. By re-using existing software, in addition not having to re-implement it, one can avoid downstream costs of maintaining additional code, and (if the re-used artifact has been thoroughly tested) increase the overall quality of the software product. Several industrial and governmental initiatives are underway to increase the reuse of software, involving both adjustments to process, and the adoption of new technologies. As these efforts mature, it is very important to demonstrate to management and funding agencies that reuse makes good business sense; to this end, it is necessary to have methods to gather and furnish clear financial evidence of the benefits of reuse in real projects. Thus, we need to define good *metrics* that capture these benefits, and develop tools and processes to allow the effective use of these metrics.

We can think of *reuse benefit* of a project or system, as being the normalized (percentage) financial gain due to reuse. This is an example of an *external* process attribute (see [7]), concerned with an external input (money) into the software development process. Unfortunately, the direct measurement of the actual financial impact of reuse in a system can be difficult. The project as a whole may not have the machinery in place to gather financial data. There are also other difficulties associated with measuring the financial impact of reuse. There are different types of reuse—reuse of specifications, of design, and code. Specification and design processes often have informal products (such as natural

*Devanbu is with the Software & Systems Research Laboratory, AT&T Bell Laboratories, 600 Mountain Av., Murray Hill NJ 07974, USA. Karstu is with Michigan Technological University, Houghton, MI. Melo and Thomas are with the University of Maryland, Institute for Advanced Computer Studies and Computer Science Dept., College Park, MD 20742 USA. E-mails: {prem,karstu}@research.att.com, {melo,bthomas}@cs.umd.edu

language documents) which can be quite incommensurate. Even in reuse of code, there are different *modus operandi*, from the primitive "cut, edit, and paste", to the formal, controlled language based approaches provided in languages such as C++ and ML. In any case, to determine cost savings, one may have to ask individual developers to estimate the financial benefit of the code that they reused. This information may be unreliable and inconsistent.

Fortunately, one of the key approaches to reuse is the use of features such as functions and modules in modern programming languages. In this context, one can find evidence of (some kinds of) reuse directly in the code; thus, it may be possible to find an *indirect* measure of the benefits of software (code) reuse *directly in the code*. Measures derivable directly from the code are *internal* measures. Several such measures of software reuse have been proposed in the literature [8, 14, 3, 10, 13]. This paper is concerned with the evaluation of how well various *indirect, internal measures* of software reuse actually measure the relevant *external process attribute*: reuse benefit.

The rest of the paper is organized as follows. First, following the lead of Weyuker [17] in the field of complexity measures, we develop some *general properties* or axioms that (we argue) should apply to any measure of reuse benefit. Although (for reasons discussed above) it is difficult to develop a direct, external measure of reuse benefit, these axioms give us a yardstick to evaluate candidate *internal* measures. We then look at the internal measures of reuse reported in the literature and analytically examine their relationship to these properties. Finally, we describe an empirical evaluation of these metrics. We have constructed tools to gather the internal metrics, and methods to gather corresponding process data. We use statistical methods to assess the relationship of the various internal metrics with the corresponding process data. The results suggest some possible improvements to the published internal measures of software reuse. This paper is aimed at establishing a broad framework to assist in the study of reuse metrics, covering: a) the formulation of analytic properties, b) analytic evaluation of published metrics, c) construction of metrics gathering tools, and d) empirical evaluation which in turn shed some light on the analytic properties.

2 Indirect Measurement of Reuse Benefit

Fenton [7] categorizes software measures along two orthogonal axes. The first is the process/product axis: a metric may measure an attribute of software *product*, (*e.g.,* quality of code), or an attribute of software *process* (*e.g.,* cost of design review meetings). Another, orthogonal axis is the internal/external axis. A metric may measure an *internal* attribute (*e.g.,* the number of loops in a module), or an *external* attribute (*e.g.,* maintainability of a module). Our goal is to develop a reasonable way of measuring the actual financial impact of reusing software. By Fenton's categorization, this is an external process attribute. We would like to measure reuse benefit as a normalized measure of the degree of cost savings achieved by adopting software reuse. Thus, we define R_b, the reuse benefit of a system S, as follows:

$$R_b(S) = \frac{cost\ of\ developing\ S\ without\ reuse - cost\ of\ developing\ S\ with\ reuse}{cost\ of\ S\ without\ reuse} \quad (1)$$

It is important to note here that we are really concerned with the cost of development, which is quite different from the incremental benefit to revenue from the product. It may be possible that by doing reuse, we bring out the product to market earlier, and with greater functionality. This may well increase revenue. Our model ignores this: R_b is solely concerned with the effect on coding costs.

For reasons given in the introduction, it can be difficult to get a reasonable direct measure of R_b. In cases like this, *indirect* measures have been used. For example, the external process attribute of *maintainability* is often measured indirectly[1] by internal product measures of *complexity* such as cyclomatic complexity. Likewise, the internal product measure of software *size* (in units of NCSL) is considered be a reasonable indirect measure of the external process attribute of *development cost.*

[1]Indirect measures are also used often in the physical and social sciences. For example the attribute of *temperature* is measured indirectly by the *length* of a mercury column in a thermometer.

Following this approach, we are concerned with the development of an indirect internal measurement of R_b, the reuse benefit of a system S, from the product, by searching the source code of S for instances of language-based reuse such as subroutine calls.

With such an indirect measure, there is a risk that we are not really measuring what we seek to measure; we would therefore like to validate our indirect measure in some way. One approach to validating indirect measures is to perform empirical studies, whereby one gathers statistical data about both the indirect and direct measures of the attribute in question, and tries to show that there are some correlations between the direct and indirect measures, and perhaps construct a regression model. A parallel (or perhaps preceding) approach, proposed by Weyuker [17] and others is to enumerate some formal properties that should hold of any measure (direct or indirect) of the attribute in question. Then, given a candidate measure, one can evaluate whether these properties apply to it. Weyuker used this approach to evaluate several internal measures of complexity. Of course, we are using this approach differently than Weyuker: she "axiomatized[2]" properties of a complexity internal measure, and evaluated several internal complexity measures against these properties. We are seeking to "axiomatize" an external measure—reuse benefit—and use these "axioms" to evaluate and develop indirect internal measures of reuse benefit. In addition, measuring reuse benefit is quite different from measuring complexity; thus many of her axioms aren't relevant in our context. However, her Property 4 (implementation dependence) is critically important in measuring reuse, and in fact, we reformulate and strengthen Property 4 in several ways applicable specially to measures of reuse benefit.

We begin with some notation, and present some "axioms", moving from the simple to the more complex.

2.1 Notation

Some definitions of the terminology that will be used in this paper:

$S_i =$ A software system or a subsystem whichever is appropriate, subscript i is to distinguish the systems from one another.

$c_j =$ A software component (module, class, function, subsystem). With a superscript e (e.g.,) "c^e" refers to an $e$xternal component, which existed independently of the system in which it is being used.

$Cu(S_1, c_1) =$ The number of times component c_1 is used in a system S_1

$Cost(X) =$ The cost of developing system or component X. it may often be hard to determine the actual cost; we use size as an indirect measure of cost.

$Function(S) =$ The "meaning" of the system S, from the customer's point of view. Two systems S_1 and S_2 are equivalent for a customer if

$$Function(S_1) = Function(S_2) \tag{2}$$

We also use (2) to denote equivalence of components.

Before we present our "axioms" of reuse benefit, it is important to emphasize that our goal here is precisely *not* to claim that our properties are the final and complete word on reuse benefit measures; we simply offer them as a candidate set for further additions/modifications.

[2] We use the quotation marks here because these are not necessarily axioms in the formal mathematical sense, but rather a list of properties that would appear to most people to hold of the measures in question.

2.2 Minimal and Maximal R_b

To begin with, we'd like to postulate what the maximum and minimum possible values of reuse benefit are. First, consider the system which uses no external components, and uses each internal component at most once. Such a system does not derive any cost savings from reuse, and should have a reuse benefit of zero. It is certainly possible (if silly) to construct such a system $\hat{S}$ which gives us the minimal possible value of R_b, when:

$$i.e., Cu(\hat{S}, {c^e}_j) \;=\; 0 \;\&\; Cu(\hat{S}, c_k) \;\leq\; 1$$

for all internal components c_k and all external components c^e_j. In this case,

$$R_b(\hat{S}) \;=\; 0$$

This is a little optimistic: it is also possible that there is actually a negative R_b. We might have a case where component provides only a very trivial functionality, and/or is very difficult to locate and understand, and/or involves a great deal of set-up or "glue" code to use. For the purposes of this paper, we assume that we only have "rational" re-use, and that there is actually a net positive benefit to every re-used component, perhaps after some number of re-uses (this topic is also dealt with later in § 2.4).

Now, for the maximal value (or upper bound) we consider a system that is built in its entirety by reusing external components. Such a system would still need some "glue" to tie all the external components together; writing the "glue" would involve some (possibly very small) additional cost. So the maximal value of reuse benefit would be strictly less that 1[3]. Thus, we have, for any system S:

Property 1

$$0 \leq R_b(S) \;<\; 1$$

2.3 Implementation Dependence

Weyuker's Property 4 [17] asserts that there are systems with the same function, but different complexity measures (based on the implementation style). This *implementation dependence* is a crucial aspect that we demand of any good measure of reuse benefit. Clearly, it is possible to produce the same functionality with and without reuse. Our measure *must* be able to distinguish between one that enjoys a great deal benefit of from reuse and one that doesn't. Thus, we insist that:

Property 2

$$\exists\; S_1,\; S_2 \; such\ that\ Function(S_1) \;=\; Function(S_2) \;\&\; R_b(S_1) \;\neq\; R_b(S_2)$$

Property 2 simply states that R_b's are different for different implementations; we need to make a stronger requirement for a reuse benefit measure. We want to be able to compare different implementations, and see which one is better or worse with respect to reuse. For example, given a system S with a nonzero reuse benefit, we should be able to find a way to syntactically perturb S, eliminate some reuse, and create a system $\tilde{S}$ that is functionally identical, but has less reuse.

Property 3

$$Given\ any\ S,\ s.t.\ R_b(S) \;>\; 0,\ there\ \;\exists\; \tilde{S}\ s.t\ Function(S) \;=\; Function(\tilde{S}) \;\&\; R_b(S) \;>\; R_b(\tilde{S})$$

[3] If it were 1, that would mean that we are simply using an entire existing system.

Property (3) is fundamentally important. It says that by changing the implementation, you can increase (or reduce) reuse while maintaining functionality. Using this property, we can successively consider different techniques implementers can use to increase reuse in a system, and demand that each of these show a corresponding increase in any good measure of reuse benefit. However, in the ensuing discussion, we always perturb an existing system by *eliminating* some reuse, while leaving the functionality untouched. This simplifies the analysis of the desired impact on the reuse benefit. The rest of this section considers different kinds of reuse implementation techniques in turn and develops a specialization of (3) for each technique.

First, we can expect that a reuse benefit measure will be sensitive the number of times a component is reused. Thus, suppose we have a system S where a component c is reused n times (for $n \geq 2$, in case it is an internal component: it must be used at least twice to be considered reused). We denote this system by S_c^n. Now suppose we create a mutation of this system, with functionality identical to it: S_c^{n-1}, by eliminating one reuse of the component c, and re-implementing the functionality by "open-coding" c; we also assume that the usage of the other components is unaffected. We can now demand the following axiom of a candidate R_b measure[4]:

Property 4

$$R_b(S_c^n) > R_b(S_c^{n-1})$$

Reuse benefit measures should also be sensitive to the cost of the component being reused. Reusing a more expensive component is more beneficial than reusing a cheaper component. Consider a system S, which reuses two components C and c each at least once; also assume that $Cost(C) > Cost(c)$. Now consider two perturbations of S, S_C^- and S_c^-. S_C^- (respectively, S_c^-) is created from S by *eliminating* one reuse of C (respectively, c) and re-implementing its functionality. Now we can say:

Property 5

$$If\ Cost(C) > Cost(c)\ then\ R_b(S_c^-) > R_b(S_C^-)$$

It should also be the case that reusing external components is better than reusing internal components (as a first approximation; there are complicating factors that we list later, in § 2.4). Thus consider a system which uses an external (pre-existing) component c^e for a certain functionality (irrespective of how often it is reused). We denote this by S_{c^e}. Now consider a perturbation of S, which replaces c by a custom-implemented, (for this system) equivalent component c. Call this new system S_c, which we will assume has the same functionality. In this case, we demand that:

Property 6

$$R_b(S_{c^e}) > R_b(S_c)$$

Consider another system $S_{c^{e,n}}$, where the external component c^e is used n times. Now we eliminate the n^{th} reuse of an external component c^e, and replace it with a use of a different, identical external component $\acute{c}^e$, thereby yielding system $S_{c^{e,n-1},\acute{c}^e}$ This often happens in large systems: a careless developer, unaware of a previously incorporated external component that performs a certain function, incorporates a distinct, but functionally identical one again from an external repository or library. The incorporation of this new code involves needless additional work to identify, procure, and validate the component; therefore, the added extra component should not increase the benefit from reuse:

Property 7

$$R_b(S_{c^{e,n}}) \geq R_b(S_{c^{e,n-1},\acute{c}^e})$$

[4] This axiom doesn't account for initial difficulties (during the first several reuses) involved in learning about an external component (or implementing it in a re-usable fashion, if it is an internal component). We address this later in § 2.4.

Finally, we have an axiom that relates to "cut & paste" reuse. For this, consider a system S with three variants that are functionally identical: S_ϕ S_{c^m} and S_{c^v}. S_ϕ is implemented by simply adding custom-crafted code to S. S_{c^m} is implemented by obtaining a component c^m from somewhere (internal or external) modifying it "slightly"[5] and linking it into S. S_{c^v} is created in a similar manner to S_{c^m}, except that an additional verbatim use c^v has been included to implement it. In this case, we should expect that verbatim reuse is better than "cut & paste" reuse which is better than no reuse at all:

Property 8

$$R_b(S_{c^v}) > R_b(S_{c^m}) > R_b(S_\phi)$$

Since "slightly modified" is hard to define, Property 8 can be particularly difficult to measure a repeatable way; perhaps for this reason, most published measures ignore this property, with the exception of [13]. However, as discussed below, our empirical study suggests that this in an important property.

Most existing measures of reuse benefit turn out to be not strictly consistent with one or more of the properties listed above; in fact, as we shall see below, there are some inherent difficulties in any approach to measuring reuse.

2.4 Difficulties in Measuring Reuse

There are some critical factors that complicate the measurement of reuse from code. Some of them are due to theoretical (computability) considerations, and some are due to human performance factors in reuse. Both kinds of difficulties complicate the application of the "axioms" (discussed in the previous section) to candidate indirect measures of reuse benefit, and make it difficult to implement tools to gather these measures.

It is certainly possible to write the same program in ways that can artificially inflate internal measures of reuse. Thus, one can artificially insert function invocations into the different branches of a conditional (this would inflate a reuse measure conformat with Property 4); the invocation could perhaps just as well been "hoisted above" the conditional. Likewise, one could re-use a component that was needlessly large (or more expensive) to artificially inflate a measure that was conformant with Property 5. In most cases, it is infeasible to detect occurrences of spurious inflation.

There are so some practical difficulties with Property (4), which takes the position that reuse benefit increases monotonically with the number of reuses. In the case of external components, there is the overhead of first finding a suitable component and learning how to use it (or with internal components, the cost of building it). This is an initial cost that would be amortized over a number of uses of the component. There may often be an initial negative cost to re-use, which is later amortized over many uses. Moreover, the parameters of this "learning curve" is likely to vary with the functionality of component, the complexity of the component, the talents of the re-user, and the type of system in which it is being reused. This is difficult to quantify

These two problems: detecting "computationally gratuitous" reuse and modeling the "learning curve" seem endemic to any tool that measures re-use, and do not appear to admit of any immediate solutions. Now, we survey several related research efforts that deal with the measurement of reuse and discuss their relationship to the properties laid out in this section.

3 Analytic Evaluation of Reuse Metrics

There are many models and metrics [2, 4, 10, 9, 3, 14] in the literature that try to evaluate the degree of reuse in a software system. Most of these measures are concerned with estimating the actual

[5] The precise definition "slight" modification will vary with the circumstances.

financial benefits due to reuse. Bieman suggests a range of measures of various reuse occurrences in object oriented software. Our theoretical framework, as well the empirical study, is concerned more with measures that yield a single number that could potentially estimate the savings due to reuse. In this section we will compare some of these models to our proposed set of properties of reuse benefit measures.

3.1 Producer/Consumer models of Software Reuse

Several researchers [4, 10, 9, 3, 14] seek to evaluate the benefits of reuse in a corporation. They use different models, but essentially, they all comprise a producer-consumer framework. Reusable artifacts are created by the *producer* (*e.g.,* a domain engineering group which produces reusable software) and re-used by several *consumers*. The producer groups have to undertake extra cost burdens to create high-quality reusable assets. Consumers save by avoiding re-implementation costs. The return on the asset producer's investment is proportional to use by consumers. Business-case oriented models of reuse metrics seek to measure the overall benefit to the corporation of re-use practices: thus they include measurements of code size, relative cost of producing re-usable software, number of reuses etc, into a unified model that can combine all these numbers into a figure for overall cost benefit of reuse. Gaffney *et al* have investigated different models for computing the financial benefits of reuse [10, 9]. Poulin *et al* [14] have developed and institutionalized a comprehensive reuse program that incorporates a producer/consumer financial model of reuse benefits. Bollinger and Pfleeger [3] propose financial and accounting practices to motivate multi-project reuse, based on the producer/consumer model.

A key component of all these efforts is a model for the amount of savings during the coding phase, directly attributable to reuse. However, the methods used for computing coding-phase savings in [3, 14, 10, 9] do not necessarily conform to the properties presented in § 2.3. For example Poulin [14] gives reuse benefit credit only for external components, and for each reused component just once, regardless of the number of times it is called. His argument is that the cost of implementing the component is saved only once; after that each additional use should not get additional credit. Programmers should be expected to use components that are in the system as a matter of course, and should not get credit for that. Since larger components are given more credit, their treatment of *external component* is consistent with the Property 5. However, the "credit for one use only" assumption is not consistent with our Property (4). For his computation of the cost savings due to re-use, he uses a *product reuse level* number, which is a normalized ratio of the number of lines of reused source instructions (RSI) to the total number of lines. To estimate the actual cost savings (Reuse Cost Avoidance, or RCA) he multiplies the RSI number by the a per-line cost savings. Chen *et al* [4] use a very similar computation, but have constructed a repeatable, tool-based measurement apparatus[6].

Given a project where all the programmers can be always expected to be aware of and likely to use all the re-usable components, Poulin's argument for giving credit only once, to just the linecount of the external components, seems applicable. But in many large, long-lived software systems, with frequent personnel turnover, programmers may be unaware of reusable components, whether internal or external. Conversations with developers have revealed cases where the same function had been re-implemented dozens of times in a very large project. Such practices complicate the calculation of the reuse benefit. As a specific example, consider a 1,000,000 line system S with 400,000 lines of RSI[7] (including a 2000-line component c_1) Now, assume that subsequently, a programmer (unaware of the existence of c_1) creates S_1, with some new functionality, by retrieving and using a component c_2 (with functionality identical to c_1, but implemented differently) of the same length (2000 lines) from an external repository. Now suppose a more careful programmer, creates S_2 from S_1, by adding another reuse of the component c_1. By Property (7), S_1 should be assigned a higher reuse benefit. However, using the RSI count, S_2 would be assigned a higher normalized reuse benefit. Even if the existing component was hard to find

[6] They are also concerned with the measurement of effectiveness of different component libraries; which libraries are used more often ? Our focus is on the *consumer* of reusable components

[7] Poulin's method assigns this system a *normalized reuse level of* 0.4.

(because of poor retrieval support), it is unclear whether the needless introduction of a new external component predicates a greater benefit from reuse.

This kind of needless re-use, by "re-discovering" external components, might inflate the RSI count and thus complicate the return on investment computations. This would appear to present difficulties for both of [10] and [14]. Intuitively, the problem seems to arise from the exclusive focus on the reused code (RSI) rather than the *manner in which it is reused in the rest of the code.* Thus simply by inflating RSI, without re-using it effectively, one can get an inflated relative benefit number. On the other hand, consider a system that is implemented without any external components at all, but which incorporates a highly modularized and parametrized architecture which allows a high degree of reuse of internal (custom crafted) components. Such a system would have an RSI of zero, but might well realize high levels of reuse benefit. Our empirical data (See § 4) includes some student projects that illustrate this possibility.

Some of the other measures discussed in this section, notably the measures of Frakes and Terry, and the R_{sf} measure, don't focus solely on the RSI, but give credit for each reuse of a component. However, these measures are still susceptible to the intractable problems noted in § 2.4: Poulin also gives examples of spuriously inflated reuse benefit resulting from such measures. Thus both methods are subject to anomalies, albeit in different contexts.

Finally, the RSI measure, (like all measures discussed in this section with the exception of RR (Section § 3.4) does not give any credit for non-verbatim reuse, *i.e,* the reuse of components that have been adapted somewhat; RSI is thus not consistent with Property 8 .

3.2 Reuse Level models of Frakes and Terry

Unlike the work described in the previous chapter, which is concerned exclusively with *how much* code is being reused, Frakes and Terry [8] focus on *how* code is being reused. Their *reuse level* and *frequency* measures are concerned with how frequently components are being used. They distinguish between internal and external reuse; total reuse is the sum of these two.

In calculation of their reuse level and reuse frequency, Frakes and Terry use *threshold levels* to determine when a component is considered being reused. This offers a pragmatic approach to dealing with the learning curve issue discussed above in § 2.4 A threshold is a value that determines when a module will be reused. If a threshold is 2 then an item that has been used more than two times is considered to be reused. Different threshold values (respectively, ETL and ITL) can be used for external reuse and internal reuse. Given these numbers, the number of internal and external components (resp., IU and EU) which are used more than the threshold can be counted; the total number of components is given by T. Frakes and Terry also count the *frequency* of reuse: the number of references to internal and external items (which are reused more than the threshold) are counted by IUF and EUF, and the total number of references is denoted by TF. Given these numbers, the overall reuse level (RL) and reuse frequency (RF) measures are computed thus:

Internal RL = IU / T; External RL = EU / T; Total RL = (IU + EU) / T
Internal RF = IUF / TF; External RF = EUF / TF ; Total RF = (IUF + EUF) / TF

The RL & RF measures are two different measures of reuse level, which could both be used as indirect measures of reuse benefit. For this purpose, these measures differ from the RSI measure used by [14]; here, there is actually a focus on *how* the reusable components are used, rather than just the total line count of reused code. In addition Frakes and Terry give credit for *both* internal and external components. However, RL and RF are different. After a given threshold value, RL is not sensitive to the number of uses of a particular component; therefore, it does not strictly conform to Property (4). RF, on the other hand, is usage sensitive.

However, these measures are insensitive to the cost of the modules being reused; thus, they don't incorporate Property (5). However, [8] does describe a simple method to weight these measures based on computation of certain ratios of the average sizes of reused modules. While this "size weighting" method accounts for the size to some extent, it is not sensitive to the level of reuse of modules of various sizes. According to Property(5), it is better to reuse larger modules (if size is taken as a good proxy for cost).

Finally, RL and RF only count verbatim reuse; if a slightly modified version of an existing component is used again, it would be treated as a use of a new component; depending on the level at which the threshold is set, this may not be recognized as being re-used. Thus, RL and RF may not always conform to Property 8.

3.3 Size and Frequency metric - R_{sf}

In this section, we describe another normalized indirect measure of reuse benefit, R_{sf}, first described in [6]. This measure tries to account for both *how much* code is being reused, as well as in *what manner* it is being reused (sf stands for size and frequency) It uses a notion of *expanded code size* $Size_{sf}$, which indicates how much code would have to be written to implement the system, had there not been any reuse. The actual code size is denoted by $Size_{act}$. We model our measures in general thus:

$$R_{sf}(S) = \frac{Size_{sf} - Size_{act}}{Size_{sf}} \tag{3}$$

The form of this equation is almost identical with the form of the equation (1) on page 2. In fact, equation (3) follows directly from equation (1) using a simple two step argument. First, we take the size of a system to be a good indicator of the effort taken to implement (and thus the cost of) the system. Second, we take the expanded size $Size_{sf}$ of the system as a proxy for the cost of the system without reuse, and $Size_{act}$ be a proxy of the actual cost of implementing the system. $Size_{act}$ is simply the number of statements in the newly written functions of implemented system (not counting reused pre-existing code from external repositories). This is a fixed number, computed in the usual way. It should be immediately clear (since $Size_{act}$ is a positive non zero number) that if $Size_{sf} \geq Size_{act}$, the indirect measure defined above conforms strictly to Property (1) on page 4.

The definition of R_{sf} makes use of the function call graph of a program:

Definition 1 *A callgraph $CG(S)$ for a system S is a connected, directed graph rooted at the main procedure, and described by a pair $\langle N_S, E_S \rangle$ where the nodes N_S represent the functions in the system, and the edges E_S represent the function invocations. For each node n in N_S, the in-degree (the number of calls to n) of n is denoted by $calls(n)$, and the code size of n by $size(n)$. $EXT(S)$ is the set of nodes in N_S that represent functions from external libraries, and $INT(S)$ is the rest.*

$$Size_{act}(S) = \sum_{for\ all\ nodes\ n\ in\ INT(S)} size(n) \tag{4}$$

$$Size_{sf}(S) = \sum_{for\ all\ nodes\ n\ in\ N_S} size(n) * calls(n) \tag{5}$$

$$R_{sf}(S) = \frac{Size_{sf}(S) - Size_{act}(S)}{Size_{sf}(S)} \tag{6}$$

With this definition, it's easy to see that R_{sf} satisfies Property (1). In the case where there is no external component use, and each internal component is used only once, we get $Size_{sf} = Size_{act}$; in all other cases, $Size_{sf} > Sizeact$, as desired.

The $Size_{sf}$ measure is sensitive both to the size of the function being reused, and the number of times it is being used. It is easy to see that it conforms to Properties (4) and (5) provided we assume that size is a good proxy for cost. We remind the reader here that Properties (2 & 3) are weaker preliminaries to Property (4).

Now consider Property (6). Suppose we have an external function component c^e in S, of size $size(c^e)$ which is used i times ($i > 1$). Now suppose we create $\tilde{S}$ by removing one use of c^e, and re-implementing c^e as a component c^{int} (internal to S); we also make the reasonable assumption that the size of c^e is much larger than the *difference* between $size(c^e)$ and $size(c^{int})$, (*i.e.,*):

$$size(c^e) \;>> \mid size(c^{int}) - size(c^e) \mid \tag{7}$$

Under this assumption, we can easily show (the details are omitted here for clarity, and may be found in [6]) that

$$R_{sf}(S) > R_{sf}(\tilde{S})$$

as specified by Property (6).

Now we turn to property (7). Assume that we have a system S with an external function c_1^e, invoked i times ($i > 1$). Now we create a mutation $\hat{S}$, where one use of c_1^e is replaced by a functionally identical new external function c_2^e. In the case where $size(c_1^e) \geq size(c_2^e)$ we can show a result consistent with Property (7):

$$R_{sf}(S) \geq R_{sf}(\hat{S})$$

Thus, unlike the purely size-sensitive metrics described in § 3.1, R_{sf} doesn't get fooled by the inclusion of a functionally identical component of the same or smaller size. Unfortunately, if the new component is larger, this measure is also fooled, and reports a gain in reuse ! In general, however, as noted in § 2.4 such phenomena as needlessly large components are likely to pose difficulties of any practical tool that derives an indirect reuse benefit measure from the code.

Finally, R_{sf} only counts verbatim reuse. Use of a slightly modified component is not given any reuse credit; it can be easily shown that R_{sf} does not conform to Property 8. We now describe a measure that actually accounts for non-verbatim reuse.

3.4 Reuse Ratio

The reuse ratio has been used for many in the NASA Software Engineering Laboratory [12]. Recently this metric has been further investigated on object-oriented systems developed in C++ and Ada [13, 16]. It is the only measure examined here that addresses Property 8. This measure is defined for a system S, with components C_i, $i \ldots n$. For each component C_i, we use a $Size(C_i)$, as before. But we now also have a *change ratio* $Change_i$ (where $0 \leq Change_i \leq 1$) which measures what portion of the component has been hand-crafted (added, modified or deleted) for inclusion into S. Thus, for a component C_i drawn from a library and used verbatim, $Change_i$ would be zero, and for a component for which exactly 50% of the code has been rewritten $Change_i$ would be 0.5. In practice, it is difficult to account precisely for the degree of custom coding in a reused component. In [13, 16] this problem has been handled by asking the reuser if 25% or more of a component had been changed; then, the value of $Change_i$ is thresholded as follows (IR is a binary value standing for *is reused*)

$$IR(i) \;=\; 1 \; if \; Change_i < 0.25 \;, \; 0 \; otherwise$$

Using these, Melo *et al* define RR, the reuse ratio measure, thus:

$$RR(S) = \frac{\sum_{C_i \in S} IR(i) * Size(C_i)}{\sum_{C_i \in S} Size(C_i)} \tag{8}$$

The computation shown in equation 8 is very similar to that used by Poulin *et al* in the product reuse level number. Indeed, if the $IR(i)$'s were all set to zero, except for the components which were reused verbatim , the computation is identical. Thus, the analytical evaluation here is identical to the discussion in § 3.1, except for one vital difference: RR is the only measure discussed in this paper that actually conforms to Property 8. Of course, it conforms only for components which are modified 25%

Property	R_{sf}	RL	RF	RSI	RR
P1	X	X	X	X	X
P2	X	X	X	X	X
P3	X	X	X	X	X
P4	X	*	X	-	-
P5	X	-	-	*	*
P6	X	*	*	X	X
P7	*	X	X	-	-
P8	-	-	-	-	*

Table 1: Summary of Reuse Measure Conformance to Reuse Benefit Properties

or less. This deficiency stems from the difficulty of identifying the "degree of cutting and pasting" in modified components. However, we are experimenting some new algorithms due to Baker [1] which might lead to repeatable, analytic approaches to quantifying the level of modification.

3.5 Discussion

Table 1 provides a summary of the examined reuse measures in terms of their conformance to the properties listed in section 2. An "X" indicates that the measure conforms to the property, a "-" indicates that is does not conform, and a "*" indicates that it partially conforms to the property.

While all of the examined reuse measures satisfy properties 1,2, and 3, none of the measures conform to all properties. The two measures that do not consider internal reuse, (RSI and RR), do not satisfy the property associated with internal reuse, the sensitivity to multiple reuses (Property 4). These also do not satisfy Property 7. In addition, they are only partially conform to Property 5, since the size of reused *internal* components is ignored. RL and RF combine internal and external reuse; if ERL and ERL were used, they would conform to Property 6. However, they do not strictly account for the size of the reused components (Property 5). Moving to Property 7: RL and RF are only affected by the frequency of reuse of components, and are thus not fooled by the needless introduction of new external components, as are RSI and RR. R_{sf}, can be fooled in some cases, as discussed in Section 3.3, page 10. R_{sf} satisfies all properties except for Property 8, which accounts for the benefit from modifying an existing component. This property is not fully satisfied by any of the measures, and only partially satisfied by RR.

These results suggest that there is room for improvement of these measures. Since there is significant variation in the set of properties satisfied by each reuse measure, we would expect similar variation in the amount and type of benefit that they predict. We re-emphasize here that this is an *a-priori* property formulation. When a large, diverse set of reuse metrics data (with associated process data) becomes available, the validity of these different assumptions are be evaluated. As we shall see, our initial empirical study using student data indicates that some of these properties appear to be quite critical; it also indicates that there are some practical difficulties to be overcome while using some of the metrics listed in table 1.

4 Experimental validation

In order to experimentally validate the metrics discussed in the previous sections, we examined the degree to which these metrics show an impact on software productivity and quality. To do so, we used the data gathered in study performed at the University of Maryland [13]. Section 4.1 provides further details about this study, and section 4.2 describes the product and process measures that were collected in the study. Section 4.3 provides a summary of the metrics collected for each of the programs in the study. In section 4.4 we present and interpret results obtained from the statistical analysis performed on the data.

4.1 Description of the study

This study was run for four months (from September to December, 1994). The population under study was a graduate level class offered by the Department of Computer Science at the University of Maryland. All students had some experience with C or C++ programming and relational databases.

The students were randomly grouped into teams. Each team developed a medium-size management information system that supports the rental/return process of a hypothetical video rental business and maintains customer and video databases.

The development process was performed according to a sequential software engineering life-cycle model derived from the Waterfall model. This model includes the following phases: Analysis, Design, Implementation, Testing, and Repair. At the end of each phase, a document was delivered: Requirement specification, design document, code, error report, and finally, modified code, respectively. Requirement specification and design documents were checked in order to verify if they matched the system requirements. Errors found in these two first phases were reported to the students. This guaranteed that the implementation began with a correct OO analysis/design. The testing phase was accomplished by an independent group composed of experienced software professionals. This group tested all systems according to similar test plans and using functional testing techniques. During the repair phase, the students were asked to correct their system based on the errors found by the independent test group.

OMT, an OO Analysis/Design method, was used during the analysis and design phases [15]. The C++ programming language, the GNU software development environment, and OSF/MOTIF were used during the implementation. Sun Sparcstations were used as the implementation platform. Therefore, the development environment and technology we used are consistent with what is currently used in industry and academia.

The following libraries were provided to the students:

- MotifApp. This public domain library provides a set of C++ classes on top of OSF/MOTIF for manipulation of windows, dialogs, menus, etc. [18]. The MotifApp library provides a way to use the OSF/Motif library in an OO programming/design style.
- GNU library. This library is a public domain library provided in the GNU C++ programming environment. Its contains functions for manipulation of string, files, lists, etc.
- C++ database library. This library provides the implementation in C++ of multi-indexed B-Trees.

A hundred small programs exemplifying how to use OSF/Motif widgets were also provided. Finally, the code sources and the complete documentation of the libraries were made available. It is important to note that the students were not mandated to use the libraries and, depending on the particular design they adopted, different reuse choices were expected. We also provided a domain specific application library in order to make our experiment more representative of the "real world". This library implemented the graphical user interface for insertion/removal of customers and was implemented in such a way that the main resources of the OSF/Motif and MotifApp libraries were used.

4.2 Data collected

Both product and process data were gathered as a part of this study. We describe here only the product and process data that are relevant to help us validate the suite of reuse metrics presented in this paper. For further details about how these data were gathered and validated see [13].

4.2.1 Product data

We have built the software tool infrastructure to gather data about 4 different reuse measures: our R_{sf} metrics, the RSI metric used by Poulin and others, and the RL and RF metrics of Frakes and Terry.

Our tools have 3 elements. First, we have a static analyzer, built with the `GEN++` [5] analyzer generator, which analyses C++ programs and generates call graph and function size information. This information is generated into flat files. These are then processed by a relational database system (Daytona [11]) which supports such features as transitive closure (which is needed to identify a connected call graph), and aggregate queries (which are needed to compute the different summary metrics).

Unfortunately, we did not have a software tool to calculate reuse ratio. We used a form, the component origination form [13], to capture whether a component has been developed from scratch or has been developed from a reused component. In the latter case, we asked the developers to tell us if more or less than 25 percent of a component had been changed. In the former case, the component was labeled: *Extensively modified* and in the latter case: *slightly modified*. If the component was inserted into the system without any modification it was labeled: *verbatim reuse*. Only *verbatim reuse* and *slightly modified* have been used to calculate reuse ratio [13].

4.2.2 Effort

Here we are interested in estimating the effort breakdown for development phases, and for error correction. Again, we used forms filled out by the developers to track person-hours expended across development activities. These activities include:

- Analysis. The number of hours spent understanding the concepts embedded in the system before any actual design work. This activity includes requirements definition and requirements analysis. It also includes the analysis of any changes made to requirements or specifications, regardless of where in the life cycle they occur.

- Design. The number of hours spent performing design activities, such as high-level partitioning of the problem, drawing design diagrams, specifying components, writing class definitions, defining object interactions, etc. The time spent reviewing design material, such as walk-throughs and studying the current system design, was also taken into account.

- Implementation. The number of hours spent writing code and testing individual system components

- Rework. This includes the number of hours spent on isolating errors, as well as correcting them.

4.2.3 Number of Defects

Here we analyze the number of defects found for each system/component. We will use the term defect as a generic term, to refer to either an error or a fault. Errors and faults are two pertinent ways to count defects, thus they were both considered in this study. Errors are defects in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools. Faults are concrete manifestations of errors within the software. One error may cause several faults and various errors may cause identical faults. In our study, an error is assumed to be represented by a single error report form; a fault is represented by a physical change to a component.

4.3 Overview of the projects

Table 2 provides descriptive measures of the projects included in the study, showing the project ID, project size (source lines of code (SLOC)), total lifecycle productivity (SLOC/Hour), fault density (Faults/KSLOC), and error density (Errors/KSLOC).

Number	SLOC	Prod.	Fault Dens.	Error Dens.
1	5105	18.23	8.23	6.46
2	11687	32.02	3.76	3.59
3	10390	34.30	3.95	3.17
4	8173	51.40	8.20	3.18
5	8216	31.12	3.41	3.04
6	9736	69.54	1.64	1.54
7	5255	19.91	14.27	8.37

Table 2: Size, Productivity, Fault Density, and Error Density in the Examined Projects

Number	R_{sf}	RL	RF	RSI	RR
1	0.45	0.52	0.79	0.00	0.02
2	0.86	0.37	0.78	0.08	0.26
3	0.45	0.28	0.64	0.00	0.15
4	0.93	0.52	0.92	0.00	0.40
5	0.74	0.38	0.76	0.11	0.38
6	0.83	0.38	0.76	0.11	0.43
7	0.51	0.45	0.81	0.00	0.00

Table 3: Experimental Results: Reuse Measures

Table 3 shows for each project the reuse measures discussed in the previous sections: reuse benefit, reuse level, reuse frequency, Pct. RSI, and reuse ratio. As one can see, RSI shows very little variation across the projects: most of the projects have RSI equal to zero. Given that, we will not analyze the impact of RSI on productivity and quality, since the poor distribution in our sample can easily bias the stastistical analysis.

4.4 Results

To provide some evidence of the usefulness of the measure of reuse benefit, we examined the relationship between reuse benefit and the quality factors of productivity, defect density, and rework effort. The coefficients of correlation between these quality measures and the measures of reuse benefit are shown in table 4. The following sections describe our observations on the relationship between these quality factors and the various reuse measures.

Measure	R_{sf}	RL	RF	RR
Productivity	0.66	-0.16	0.12	0.82
Fault Density	-0.39	0.62	0.47	-0.67
Error Density	-0.62	0.49	0.20	-0.79
Percent Rework	0.09	0.62	0.69	-0.24

Table 4: Experimental Results: Correlations with Product Quality Factors

Term	R_{sf}	RR
Intercept	2.07	2.94
std. err.	0.64	0.16
p-value	0.02	0.00
ln(R)	2.78	2.78
std. err.	1.21	0.69
p-value	0.07	0.01
R^2	0.51	0.77

Table 5: Comparison of Reuse Measures in Models of Productivity

4.4.1 Productivity

Productivity is typically calculated as size of the system divided by cost spent to develop it, for some measure of size and cost. Keeping the size of a system constant, increasing productivity will result in a reduction in cost. There are many ways to measure both of these quantities, so as a result, there are many different measures of productivity. We used the total number of hours spent across development phases (analysis, design, implementation, testing) and rework as our measure of cost. Size was calculated as the total source lines of code (SLOC).

Using this measure of productivity, we first examined the correlations between the various reuse measures and productivity. As shown in table 4, the reuse ratio measure clearly has the best correlation with this measure of productivity. The only other measure that has a significant correlation with productivity is R_{sf}, with a correlation of 0.66.

A model can be developed to quantify the impact of reuse benefit on productivity. Since both reuse benefit (R) and productivity (Π) are non-negative real valued variables, we can model their relationship as:

$$\Pi = a(1+R)^b,$$

for some coefficients a and b. When there is no reuse, productivity is a. As reuse benefit increases, productivity increases, with the maximum reuse benefit of 1 resulting in productivity of $a * 2^b$. Taking the natural logarithm of both sides of the equation and simplifying yields the following:

$$\ln(\Pi) = ln(a) + b\ln(1+R).$$

With this form of the model, we can use a standard least squares regression to estimate the coefficients a and b.

Table 5 shows models of this form developed using the two reuse measures best correlated with productivity. The table shows the calculated coefficients for the intercept ($\ln(a)$) and the explanatory variable R (b), as well as their standard error and level of significance. Using R_{sf}, the R^2 for the model is .51, indicating that R_{sf} explains half the variation in productivity. The model developed using RR is stronger, with an R^2 of 0.77. The intercept for this model is 2.94, so when $RR = 0$, $ln(\Pi) = 2.94$, and thus productivity without reuse is $e^{2.94}$, or 18.94 SLOC/Hour. As RR increases, productivity increases. For example, an increase in reuse ratio from 0.20 to 0.30 would result in an increase in productivity from 31.4 to 39.2 SLOC per hour. As there are no projects in this sample with RR greater than 50%, any conclusion about productivity for very high levels of RR would be purely speculative.

4.4.2 Product Quality

We examined the relationship of the reuse measures to the product quality measures of fault and error density. As with productivity, we used standard definitions of fault and error density, Faults per KSLOC and Errors per KSLOC, resp. The expected effect is that as reuse increases, these measures of fault

Term	Reuse Ratio
Intercept	2.28
std. err.	0.37
p-value	0.00
ln(R)	-3.32
std. err.	1.55
p-value	0.09
R^2	.49

Table 6: Comparison of Reuse Measures in Models of Fault Density

and error density will decrease. The coefficients of correlation of these defect density measures with the measures of reuse benefit are shown in table 4.

For fault density, the RR again has the best correlation (r=-0.67) followed by RL (r=0.62). However, RL had a correlation in the opposite direction, i.e., as RL increases, fault density increases. This is the opposite of the result for RR and R_{sf}, which shows the expected relationship that as reuse increases, fault density decreases. One reason that RL (and RF) are correlated in this direction is that RL is defined as a measure of the density of subprogram calls. Such measures have been identified as indicators of an increased error density. Another way of looking at this is that given a function f that is needed by the developer, if he can call an existing function g, there will be an increase of a single line of code in the total project SLOC. On the other hand, if the developer prefers to create a new function g' by copying the code from g, the change in project size will be an increase of the SLOC of g. The increase with the latter option will be greater than for the former, resulting in a smaller defect density for the case where code is copied, and a larger defect density when the function is called.

The reuse ratio had the strongest correlation with Error Density, showing the expected result, namely, that as reuse increased, error density decreased. R_{sf} also had a high negative correlation with Error Density (Pearson r = -0.62). Again RL and RF had a positive correlation with Error Density showing that as the frequency increases the quality did not increase. Based on these results it appears that property 4 (which says as frequency increases the benefit should also increase) may not be applicable to measure the reuse benefit in terms of software quality.

Using an approach similar to that described for productivity, models for defect density can be developed. Again, we used a logarithmic form of the model, and used a standard least squares regression to obtain estimates of the model coefficients. A comparison of the best models is shown in table 6, showing, for each model, the calculated coefficients for the intercept and explanatory variable, their associated standard errors and p-values, and the model R^2.

4.4.3 Rework Effort

We also looked at a measure of rework, the percentage of the effort that was spent in correcting errors, or rework hours divided by total hours. This measure quantifies the inefficiency in the development process due to development errors, and is independent of how the size of the system is computed.

As indicated in table 4, R_{sf} and RR did not correlate well with this measure.

RL and RF had correlations of similar strength, however, again the indicate a negative effect, as RL and RF increase, the percentage of rework also increases. This is in part due to the correlation with defect density discussed in the previous section.

5 Conclusion

This paper is concerned with an evaluation of *indirect* measurement of the benefit of software reuse. Five metrics proposed in the literature have been analytically and empirically assessed with regard to their capabilities to predict productivity and quality in object-oriented systems. To analytically evaluate the metrics, we have proposed a set of desirable properties of reuse benefit measures, and evaluated these metrics in terms of their compliance with these properties.

None of the metrics satisfied all the properties, as all had strengths in some areas and weaknesses in others. RL and RF fall short in terms of the sensitivity to the cost of the reused object and the additional benefit from external reuse. RSI and RR do not cover the benefit of internal reuse. R_{sf} appears to provide a good balance, accounting for the benefit of both internal and external reuse. However, it does not account for reuse via modification, a weakness of all the measure except for RR.

To empirically evaluate the metrics, we have (1) constructed a set of tools to extract these metrics from C++ programs, (2) collected process data on the development of a set of small object-oriented systems, and then, based on the product and process data collected on these systems, (3) verified statistically the correlations between these metrics and the quality factors of productivity and defect density. Finally, for those metrics that correlated well with productivity and defect density, we also developed predictive models.

RR is well correlated with productivity, fault density and error density, but, not with the percentage of rework effort. R_{sf} has significant correlations with productivity and error density, but not with fault density of the percentage of rework effort. RL and RF appear correlated with fault and error density, and the percentage of rework effort, but interestingly, in the wrong direction. as RL and RF increase, we see an increase in fault density, rework density, and the percentage of rework effort.

A major difference between R_{sf} and RL/RF is that R_{sf} accounts for component size. This important difference may be the reason for the markedly different results found with these measures, with R_{sf} showing some correlation with the quality factors, and RL/RF showing either no correlation, or a significant correlation, but in the wrong direction.

Another interesting point raised with this work is the fact that the modified components also appear to have a significant effect in terms of increasing productivity and quality, and, thus, should be also considered in a comprehensive definition of a reuse metric. Nevertheless, this raises some questions. For instance, how can we accurately verify the extent to which component has been changed? What should the modification threshold be? In this work we assumed that only components changes less than 25 percent should be counted. This threshold may be domain dependent, i.e., different organizations should conduct empirical work in order to determine which threshold is most significant in their environment. In addition, tools must be built in order to determine automatically how much a component has been changed. This can, in fact, reduce the human error introduced in the analysis, thus increasing the accuracy and reliability of the results.

Finally, our empirical study has highlighted a practical difficulty in using RSI. In four out of the seven student projects used in our data, there was *no verbatim reuse* of components from external libraries. For this reason, RSI was zero in four out of seven data points. This precludes any useful analysis of the predictive power of the RSI data; however, our experience indicates that RSI may not provide helpful data in projects where a significant number of external components are used only after modifications. From our experience, it appears that that other metrics offer some ability to explain the variance in productivity and quality data even in such cases; this suggests that RSI may be taking too strict a view of what constitutes reuse.

The results indicate that different reuse metrics can be used as predictors of different quality attributes. For example, reuse ratio and size/frequency reuse metric each appeared to be well correlated with productivity and error density, but this size/frequency metric did not show any significant result with regard to faulty density. Further empirical validation is, thus, still necessary in order to evaluate these metrics in actual software organizations. Empirical work is (in general) hobbled by the difficulty in obtaining sufficient process data to allow for empirical validation of the metrics. This work provides a framework by which reuse metrics can be analytically and empirically evaluated prior to their use: the analytical properties, software tools and data collection programs developed in the framework of this study can be used in other studies, thus facilitating the replication of this work in academia and industry. As a continuation of this work, we intend to (1) perform case studies at the Software Engineering Laboratory (SEL) to assess the feasibility automated methods of determining the amount of modification in a component and to further identify what is an appropriate threshold of modification to still achieve a reuse benefit, (2) evaluate the set of metrics analyzed in this paper using the product and process data extracted from object-oriented systems under development at the SEL, (3) evaluate the capabilities of prediction of these metrics with regard to fault density, rework and maintainability, (4) continue the empirical analyses to better understand the importance of the proposed properties of reuse measures identified in this paper.

References

[1] B. Baker. A theory of parametrized pattern matching: algorithms and applications. *Journal of Comput. Sys. Sci.,* to appear, 1995.

[2] J. M. Bieman. Deriving measures of softwre reuse in object oriented systems. In T. Denvir, R. Herman, and R. W. Whittey, editors, *Formal Aspects of Measurement*, pages 79–82. Springer–Verlag, 1992.

[3] T. Bollinger and S Pfleeger. Economics of reuse: issues and alternatives. *Information and Software Technology*, 32(10):643–652, 1990.

[4] Y-F. Chen, B. Krishnamurthy, and K-P. Vo. An Objective Reuse Metric: Model and Methodology. In *Fifth European Software Engineering Conference*, 1995.

[5] P. Devanbu. GENOA a customizable, language and front–end independent code analyzer. In *Proc. of 14th Int'l Conf. on Software Engineering (ICSE)*, pages 307–317. IEEE Press, 1992.

[6] P. Devanbu and S. Karstu. Measuring the benefits of software reuse. Technical report, AT&T Bell Laboratories, 1994.

[7] N. E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman & Hall, 1991.

[8] W. Frakes and C. Terry. Reuse level metrics. Technical Report TR 94-03, Virginia Polytechnic Institute and State University, 1991.

[9] J. E. Gaffney and R. Cruickshank. A general economic model for software reuse. In *Proc. of the 14th Int'l Conf. on Software Engineering.* IEEE Press, 1992.

[10] J. E. Gaffney and T. A. Durek. Software reuse — key to enhanced productivity: some quantitative models. *Information and Software Technology*, 31(5):258–267, 1989.

[11] R. Greer. All about daytona. Technical report, AT&T Bell Laboratories, 1994.

[12] F. McGarry, R. Pajerski, G. Page, S. Waligora, V. Basili, and M. Zelkowitz. Software process improvement in the nasa software engineering laboratory. Technical Report CMU/SEI-95-TR-22, Carniege-Mellon Unv., S/W Eng. Institute, Dec. 1994.

[13] W. Melo, L.Briand, and V. Basili. Measuring the impact of reuse on quality and productivity in object-oriented systems. Technical Report CS-TR-3395, University of Maryland, Computer Science Department, 1995.

[14] J. Poulin, J. Caruso, and D. Hancock. The business case for software reuse. *IBM Systems Journal*, 32(4):567–594, 1993.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[16] W. Thomas, A. Delis, and V. Basili. An analysis of errors in a reuse-oriented development environment. Technical Report CS-TR-3424, Dept. of Computer Science, University of Maryland, College Park, MD, 20742, Feb. 1995.

[17] E. J. Weyuker. Evaluating software complexity metrics. *IEEE Transacations on Software Engineering*, 14(9):1357–1365, 1988.

[18] A. Young. *Object-Oriented Programming with C++ and OSF/MOTIF.* Prentice-Hall, 1992.