Software Quality Is Inversely Proportional to Potential Local Verification Effort*

John E. Hopkins
Murali Sitaraman

Department of Statistics and Computer Science
West Virginia University
P. O. Box 6330
Morgantown, WV 26506-6330
Tel: (304)-293-3607, fax: (304)-293-2272
Email: jhopkins@cs.wvu.edu, murali@cs.wvu.edu

Abstract

Quality and methods for measuring quality are important for all software, especially for those that are meant to be reused. Two factors constitute the quality of a software product: (1) Whether the product correctly meets its specification and (2) whether the product is "well engineered." Intuitively, a well-engineered component is one that is easy to comprehend, maintain, or modify.

Software engineering literature [Pressman 92, Sommerville 89] is replete with factors and metrics for determining how well a software product has been engineered. Unfortunately, it is not at all clear how even a smart software designer can use these factors to produce a well-engineered software product. Most published factors and metrics are good only to the extent that they can be used to argue "statistically" that one product is better than another.

In this paper, we propose a single factor for evaluating the goodness of the engineering of a software product - the *Potential Verfication Effort (PVE) involved in locally establishing the correctness of the product.* It is possible to *design* software *to minimize the PVE.* In addition, *reduced PVE directly increases most conventionally-used software engineering metrics.* PVE is not affected by whether a software product is correct or whether it is possible to establish correctness.

Keywords: Reuse, software engineering, metrics, local verifiability

Workshop Goals: Interact; advance theory and practices of software reuse

Working Groups: certification, design for reuse, metrics, and reuse education.

*This research is funded in part by NASA Grant 7629/229/0824 and NSF Grant CCR-9204461; it has also benefitted from ARPA Grant DAAL03-92-G-0412.

1 Background

Over the past several years, we have been investigating various aspects of software reuse including specification of abstract functionality and performance, formal verification and testing, language design, portability, distributed and real-time computing, and education. We have regularly presented papers and participated in reuse conferences and workshops, including the Annual Workshops on Software Reuse. Our research in software engineering issues at the West Virginia University is currently funded in part by ARPA Grant DAAL03-92-G-0412, NASA Grant 7629/229/0824, and NSF Grant CCR-9204461.

2 Position

- Software quality is inversely proportional to potential verification effort (PVE) for establishing local correctness.

Sub-positions

1. It is possible to design to minimize the PVE.
2. Lower PVE improves conventional software engineering metrics.
3. Metrics for accurately measuring PVE are likely to be different from conventional SE metrics.

3 Justification for the position

The quality of a software product can be expressed as an ordered pair:

1. The "goodness" of the results
2. The "goodness" of the product.

The first part is easy to understand and evaluate. A software product that generates specified results is *correct*. To determine whether a software product is correct, it can be formally verified or can be tested.

The second part, however, is not as easy to quantify. Software engineering literature [Pressman, Sommerville] is replete with factors and metrics for measuring this aspect of quality - determining how good a software product is or how well a software product has been engineered. Most published factors and metrics are good only to the extent that they can be used to argue "statistically" that one product is better than another. Unfortunately, it is not at all clear how even a smart software designer can use these factors to produce a well-engineered software product.

We take the position in this paper that the potential verification effort (PVE) for evaluating part 1 (i.e., correctness) locally is a useful factor for evaluating part 2. We show that designing to minimize PVE improves software quality and that minimized PVE directly implies improved ratings on conventional software engineering metrics.

```
function Max returns control
   parameters
      preserves x: integer
      preserves y: integer
   ensures "Max iff x > y"
```

Figure 1: Specification of Max

function Max(x,y) is begin if x > y then return(x) else return(y); end Max;

function Max(x,y) is begin if x > y then return(x) else return(x); end Max;

Figure 2: Implementations of Max

3.1 Independence of correctness and PVE

First, we define PVE. PVE is the maximum amount of effort needed to verify a product. The maximum amount of verification effort will occur if a product is correct with respect to its specification. In other words, PVE is not influenced by an incorrect implementation. An incorrect implementation will be treated as if it were correct, thus yielding a PVE rating reflecting the maximum effort necessary for verification.

For example, consider the specification in Figure 1 and the implementations in Figure 2. The specification is in RESOLVE [Sitaraman 93]. Implementation 1 is correct; implementation 2 is not. With all other things for both implementations being equal, the PVE rating for both procedures is the same. Implementation 1, of course, is of higher quality because it is correct. Thus, it is entirely possible that a well-engineered incorrect implementation may have a lower PVE than a correct, but poorly-engineered implementation.

3.2 Example: Lower PVE implies formally-specified, modular design

In this section, we discuss one example that demonstrates the utility of the PVE rating. We use 3C terms in this discussion, where a concept means a specification, content means an implementation, and the context is the local environment in which a concept or content is explained [Latour 90, Sitaraman 92, Tracz 90]. Given a concept and a context, consider the following four possible contents:

1. Monolithic (no components and no layering)
2. Layered based on 1C (content-only) components
3. Layered based on 2C (content-with-concept-only) components
4. Layered based on 3C components

Let us assume that all four contents are equal on the dimension of correctness. However, it is clear that content 1 is poorly engineered compared to content 2 (assuming that the chosen components and the layering used in content 2 are appropriate). Content 2 which uses a modular design without any specification is worse than content 3. “Local certifiability” [Weide 92] and hence “reusability” makes content 4 superior to content 3. We will return to this topic later.

PVE for monolithic content

Verification effort in this case is a function of both the number of statements and the kind of statements. Clearly an implementation involving several loops will require more effort to verify than the one that uses only if-then-else statements which is probably more difficult to verify than one that uses no control statements. PVE for two loops need not be the same either because a complex loop involving a more complex loop invariant is harder to verify than one that is less complex. In general, PVE is based on the semantics of the statements used in an implementation and the implementation. Gotos and uncontrolled use of pointers obviously increase the PVE.

PVE is also dependent on the specification for which the content is written. Here, we're only comparing contents for the same specification and therefore, we can say that PVE = f(Content 1). Alternatively, PVE can also be expressed as the effort to "reverse engineer" content 1 into the form of Content 4 (re-1-4) plus the PVE for Content 4.

The effective PVE for monolithic content 1: min (PVE(Content 1), re-1-4 + PVE(Content 4)).

PVE for content layered on 1C-components

When the sub-contents used in a content are not specified, then verification effort is *not* far less than what is required in the monolithic case. Verification here involves inline code expansion. This requires calls to subprograms, procedures, etc. to be replaced with the actual content (with proper substitutions for the calling context) during the verification process. This means we must verify approximately the same code we did for the monolithic implementation. In some sense, if understanding the bigger content involves direct understanding of each of its constituents then modularization is of relatively little use.

The effective PVE for content 2: min (PVE(inline_expanded_Content 2), re-2-4 + PVE(Content 4)).

It seems likely that re-2-4 will be smaller than re-1-4 because reverse engineering smaller modules may be inherently easier.

PVE for content layered on 2C-components

In this case, we assume that the sub-contents have formally-specified concepts. The concept, however, contains only calling information (such as the pre-conditions for procedures) etc., but does not include module-level context. In this case, specification-based proof rules can be used for operation calls instead of the ones requiring inline code expansion; The sub-contents themselves need to verified locally and independently to meet their specifications. This may not be possible if the context does not contain sufficient information.

The effective PVE for content 3: min (PVE(Content 3), re-3-4 + PVE(Content 4)).

PVE for content layered on 3C-components

In this case, each sub-component can be *locally certified* to be correct [Weide 92]. Verification effort = PVE(Content 4) for this case is the lowest. This verification effort will be even lower if each sub-component is a reusable component and its verifiability effort is amortized over its many uses. (Without local certifiability, reusability is impractical.)

The effective PVE for content 4 = f(Content 4, rf) where rf is the reuse factor.

Though we have concentrated on the 3C model in this discussion, the Constraints module introduced in [Sitaraman 92] "contents layered using 4C components," can be useful in including performance issues in the PVE factor.

3.3 Discussion

We have demonstrated a strong connection between PVE and modular design of software in this section. We believe similar results can be established for most other issues that are essential for a "well-engineered" software product.

Metrics for PVE

How can the PVE be measured? Common metrics such as cyclomatic complexity and lines of code provide some indication of the PVE within a module. Module interaction can provide a measure of PVE across modules. However, for PVE to be an accurate predictor, *metrics based on formal assertions* such as pre- and post-conditions of procedures, loop invariants, and semantics of statements need to designed and developed.

Designing for lower PVE

It is possible to design software so that its PVE is minimized, and this is an important advantage of the PVE factor. For example, Ada components designed following the guidelines in [Hollingsworth 92] will require lower PVE than ones that are not. Alternatively, adherence to guidelines such as these may provide a useful PVE metric.

Finally, we emphasize again that PVE is independent of software correctness. PVE is a useful factor irrespective of whether verification is feasible. For a software engineer, it provides an objective that can be understood through formal training and can be followed. In the end, it is the thought (that verifiability is important) that counts!

4 References

[Biggerstaff 89] T. Biggerstaff and A. J. Perlis, *Software Reusability*, Volumes 1 and 2, Addison-Wesley, 1989.

[Hollingsworth 92] Hollingsworth, J., *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada*, Ph. D. Diss. The Ohio State Univ., Columbus, Ohio, 1992.

[Latour 90] Latour, L., Wheeler, T., and Frakes, W., "Descriptive and Predictive Aspects of the 3C Model: SETA1 Working Group Summary," *Third Annual Workshop: Methods and Tools for Reuse*, Syracuse, 1990.

[Pressman 92] Pressman, R. S., *Software Engineering: A Practioner's Approach*, McGraw-Hill, 1992.

[Sitaraman 92] Sitaraman, M., "A Unifrom Treatment of Reusability of Software Engineering Assets," *WISR'92 Proceedings*, Palo Alto, CA, October 1992.

[Sitaraman 93] Sitaraman, M., Welch, L.R., and Harms, D.E., "On Specification of Reusable Software Components," *International Journal of Software Engineering and Knowledge Engineering 3*, 2, World Scientific, 1993.

[Sommerville 89] Sommerville, I., *Software Engineering*, 3rd ed., Addison-Westley, 1989.

[Tracz 90] Tracz, W., "The Three Cons of Software Reuse," *Third Annual Workshop: Methods and Tools for Reuse*, Syracuse, 1990.

[Weide 92] Weide, B. W., and Hollingsworth, J., "Scalability of Reuse Technology to Large Systems Requires Local Certifiability," *WISR'92 Proceedings*, Palo Alto, CA, October 1992.

5 Biography

John Hopkins is a graduate student majoring in computer science at West Virginia University. He holds degrees in mathematics and computer science from West Virginia Institute of Technology. As part of the Software Reusability Group at West Virginia University, his research interests include formal specification, formal verification, language design and software quality.

Sitaraman is an assistant professor in computer science at the West Virginia University. He has a Ph.D. from The Ohio State University (1990). His research focuses on various aspects of software reuse and software engineering, in general. He and members of his group are currently working on specification of abstract functionality and performance, formal verification and testing, language design, portability, distributed and real-time computing, and education. Sitaraman has authored several technical papers on related topics in software engineering. He is a member of the ACM and IEEE Computer Society.