Software Quality and Productivity Improvement

R.G. Dromey
Software Quality Institute,
Griffith University, Nathan, QLD., 4111, AUSTRALIA

Abstract

For there to be order-of-magnitude improvements in the quality of software and the productivity of software development and evolution there needs to be a much more strategic approach to software engineering. Such an approach needs to place a balanced emphasis on both software products and software development processes. A set of factors that are likely to make the most significant contributions to quality and productivity improvement are identified and discussed.

Keyword Codes: D.2.0 D.2.4, D.3.0, K.1.0, F.3.1
Keywords: Software Engineering, Reengineering, Reuse, Software Metrics

1. INTRODUCTION

For a long time in software engineering the dominant focus has been upon all that is associated with software development. What has become clear over recent years is that if we want to produce quality software, cost-effectively then it will be necessary to focus on improving both the products of software development and the processes used to develop software. Both software and the processes used to develop software may be subjected to three things:

- development
- evaluation (evaluation = assessment, measurement, inspection, audit)
- improvement

As the following diagram illustrates, up until recently the focus in software engineering has been in the top left-hand comer rather than in the other five segments. The challenge that we face is to raise the maturity of our knowledge, tools and practices associated with these other five segments. As progress is made in these areas it will make increasingly important contributions to the quality of software and the productivity of software development.
2. PROBLEM DOMAIN

In broadening the focus of software engineering to concentrate on quality and productivity improvement [1] a number of key problems need to be addressed:

- We need to develop a much more mature understanding of what constitutes quality software. Progress on this problem through the construction of better quality models is a prerequisite for (a) building quality into software, (b) evaluating the quality of software and (c) improving the quality of existing software.

- Much currently produced software and large tracts of legacy code is of poor quality and must undergo rework restructuring and/or reengineering. More powerful reengineering methods and tools are needed. A major problem looming is the need to protect and exploit existing software investments in the face of massive changes in computer architecture expected over the next ten years.

- Software is difficult and costly to maintain and yet there is a high demand to modify and/or enhance it because organisations must keep adapting to changing environments. We must build on the contribution that object-oriented techniques have made to making software more resilient to change. Object-oriented techniques only partly solve the problem.

- The high cost of developing software c.f. other technology means that we must find ways to make reuse of existing software effective in order to improve productivity and quality. Environments must be developed that make all forms reuse easy to practice for computing professionals. Again object-oriented techniques only partly solve the problem.

- There are ever-increasing needs to develop highly complex systems for poorly understood problems. A priority is to find less vulnerable ways of responding to such problems through more effective means of identifying the real requirements for such systems. Developing a system is a learning process, rather an activity with clear preset goals.

- It is not well-established how to create, and maintain, effective and repeatable software development processes that produce quality software, economically. In responding to this we will need to come to terms with the role technology needs to play in modelling, managing and guiding processes executed by software engineers

- Estimation of cost and schedule for large software projects remains very difficult. We need to develop methods of estimation and risk assessment that model tangible reality rather than reflect seat-of-the-pants understanding.

3. FRAMEWORK FOR IMPROVEMENT

A rigorous framework for engineering quality software that responds to the problems we have identified must be sought [2]. To achieve this goal it will be necessary to strike a balance between the empirical and the formal and between a software process [3] and a software product [4] orientation. No single approach alone, is powerful enough to comprehensively address the problem of engineering quality software. An integrated set of methodologies are needed to specify, design, implement, measure, assess, manage, re-engineer and reuse industrial and commercial software and the processes used to develop such software. At the highest level there are three key issues that must be addressed if we are to make substantial gains in software engineering.
Product Quality
To improve software product quality an intensive effort, building on current empirical and scientific/mathematical methods' research, is needed. Useful models and standards for various aspects of software product quality must be defined. These must be supported by tools to assist building quality into software, measuring, assessing and assuring the quality of software and improving poor software by applying powerful restructuring and re-engineering methods. In our quest to improve product quality we should not underestimate the importance of the role that programming languages play. Languages must play a greater role in all aspects of software development rather than simply supporting the implementation.

Process Effectiveness
The effectiveness of the processes used to develop software has a very significant impact on both the quality of software produced and the productivity of the development, maintenance and evolution of software. Defined, repeatable and assured processes are needed to cost-effectively produce quality software - that is, a quality process is needed to produce quality products. To improve the productivity of software development and software utilisation several approaches must be pursued. One approach involves focussing on key aspects of process improvement. Another approach involves reducing the need to develop new software by effective reuse of existing software in the development of new systems. A third approach involves integrating specification, design and implementation into a much more seamless environment. Productivity is also heavily dependent on the effective and strategic deployment of human resources throughout projects.

Technology Transfer
Technology transfer must also play a vital role in improving the quality of software and the productivity of software development. It is generally recognised in software engineering that there is a large gap between state-of-practice and state-of-the-art. This is reflected in such identified problems as low usage of superior software inspection methods that have been available since the 1970's and low uptake of technology to support software testing. Already there are best-practices being used by a select few to achieve remarkable gains in quality and productivity. The greatest challenge we face is to transfer proven technology to the vast majority who practice software engineering.

In the discussion which follows we will endeavour to respond to some of the problems that have been identified. Our focus will mainly be on product issues that have the potential to influence both quality and productivity as these are areas that have been somewhat neglected in the software quality debate thus far.

4. PRODUCT QUALITY IMPROVEMENT

There are a very diverse range of things that influence the quality of software. It is therefore very easy to expend effort exploring avenues that are likely to have little payoff. What we must do is identify which factors are likely to make the most significant contributions to improving software quality and focus our efforts accordingly. Here we suggest that there six key drivers that have the greatest potential to deliver the highest payoff in improving software product quality

- product quality model
- automated inspection
- language
- re-engineering
- measurement
- reuse
Software quality and productivity improvement

We will seek to explain why these factors are the most important and identify how they can contribute to software product quality and productivity improvement. We will also discuss what can be done to exploit these product quality drivers.

Much of the effort to date directed at improving software product quality has focused on the process, establishment of quality management systems and gaining accreditation for those systems. What has guided this focus has been the view that a quality process is needed to produce quality software. Such efforts however are unlikely to lead to substantive improvements in software product quality unless they are guided by comprehensive models of software product quality for each of the key deliverables of development: the requirements specification, the design, and the implementation. As the Roman philosopher Senaca wisely put it: when you know not what harbour you are sailing for any wind is the right one. What has happened thus far is that many organizations have adopted Quality Management without having a clear idea about what quality software is. As a result there are plenty of organizations that have costly certified Quality Management Systems but the bottom line is that their software has not shown any marked improvement.

A very high priority for improving software product quality is therefore to construct suitable models that systematically and comprehensively capture what we mean by quality software. Without such models we have no focus at which to direct our efforts to produce quality software through improvement and refinement of the processes that are used. You cannot build quality into software if you don't know what quality is no matter how good your intentions! Having a model for software product quality can also have a strong influence on each of the other key quality drivers.

Up until now there has been very little interest in specifically designing programming languages to facilitate building quality software. Existing languages leave essentially all of the responsibility for building quality into software to the programmers. They do little to provide an environment that supports and encourages the building of quality software. In fact, many languages actively encourage practices which are well known to seriously detract from the quality of software. Good models for software product quality can influence language designers to design better languages that support building quality into software. For example, programs that have structures that result in the execution of statements that use uninitialized variables should not be able to even compile! Embodying sound quality principles within programming languages and their compilers is by far the most efficacious way to realize product quality objectives. We will now make a claim which to many will seem controversial as a way for the future. However existing beliefs and folklore must be challenged if order-of-magnitude progress is to be made. We suggest that programming languages which embody quality principles have the potential to make the single most far-reaching impact on software product quality for the least investment. Good language tools are an indispensible prerequisite for producing quality software.

It is well recognized in the quality community [5] that in order to improve products and processes it is necessary to make meaningful measurements. In many situations it is clear what must be measured. With software however it is far from clear what needs to be measured. This situation has not stopped many organizations from feverishly trying to measure a whole range of things about software in the vain hope that once they have got the measurements they will somehow know how to use such measurements to substantially improve the quality of their software. Unfortunately most of these efforts have had very little useful impact. The reason for this is that they are not linked in any systematic way to a model of software product quality. The most desirable situation for the application of measurement for improvement is to be able to define an ideal for some property and then make measurements of a product to see if the ideal is realized. If the ideal is not realized then their is a basis and guidance for improvement. Existing measurement theory for software has not sought such an approach to measurement.
Although the goal should always be to build quality into software it is also necessary to ensure that it has actually been built-in, given human fallibility in the execution of complex tasks. For a long time it has therefore been recognized that inspections have an important role to play in producing quality software. Despite this, they are rarely applied effectively and when they are employed properly they are costly to implement given their labour-intensive nature. The whole process of inspection has been hampered by the lack of suitable models of software product quality. Once stable models for software product quality are available it then becomes possible to automate significant aspects of the inspection process and thereby improve the efficiency and effectiveness of inspections.

In most engineering disciplines reuse is widely practised. Unfortunately in software engineering this has not been the norm. Given the current labour-intensive nature of software development, the fact that reuse is not significantly exploited makes software development a very costly activity. Object-oriented design and implementation is being hailed by many as the solution to our reuse problems. We have no argument with the idea that O-O can make an important contribution to reuse. However O-O is far from a complete solution to our reuse needs. Key challenges that we face in the reuse area are: how to meaningfully identify and retrieve reusable functionality, how to exploit reuse at much higher levels than O-O allows and how to reclaim reusable software from the vast tracts of legacy code in existence.

In our quest for software product quality improvement we cannot afford to ignore existing software. In any one year the amount of new software created is small compared with the amount of existing software that has been built up over many years at huge cost. The major problem we face in regard to existing software is that much of it is of poor quality which makes it troublesome to maintain and change. What is more in today's competitive environment there is always ongoing pressure for organizations to change. And, invariably, this flows through to generating a need for organizations to change their existing software. There is therefore substantial demand to develop powerful means to reengineer poor quality software into restructured software that is much less costly to change. We should also look to apply powerful reengineering tools to newly developed software, as it is developed, to ensure that it does not end up as software that will be costly to maintain in the future.

In what follows we will make a number of suggestions for tackling the key factors that have the greatest potential for contributing to an order-of-magnitude improvement in software product quality and software development and maintenance productivity.

4.1 Model for Software Product Quality

Considerable effort has gone into setting up accredited Quality Management Systems and a plethora of process standards. These efforts must be balanced by an equally substantial focus on software product quality if significant improvements in quality and productivity are to be realized. To date there have been very few attempts to tackle the problem of software product quality systematically and comprehensively. Efforts in this whole area have largely stalled because of:

- the perceived scale of the problem
- the diversity of quality defects in software
- the difficulty of factoring high level quality attributes down to the implementation level
These barriers can be overcome by adopting the strategy of always proceeding from the tangible and measurable to high level less tangible quality attributes. Both empirical and formal means are needed in this undertaking.

4.1.1 Empirical Model

There is plenty of empirical advice scattered throughout the literature about what constitutes good software. However none of this advice amounts to a systematic model for software product quality. We suggest an appropriate starting point for developing such a model is to make the observation that programs are constructed using structural forms (e.g., variables, expressions, assignments, loops, etc). A basic characteristic of forms is that they possess properties. The form-property model [6] can facilitate (i) building quality into software, (ii) detecting quality defects in software (iii) and the systematic classification of quality defects. A focus on quality defects is also able to make a positive, if indirect, contribution to building quality into software by telling us what not to do. In considering software the two principal categories of structural forms are those that describe processes (that is, the executable statements and statement components of the language) and those that describe data (the variables, records etc). To illustrate the process of applying the form-property model we will use several simple examples.

Examples:

Consider the fundamental structural forms: variables and expressions. To be free of quality defects they should possess a number of defined properties. Here we will simply list the quality-carrying properties of variables and expressions:

<table>
<thead>
<tr>
<th>STRUCTURAL FORM</th>
<th>QUALITY-CARRYING PROPERTIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARIABLE</td>
<td>Self Descriptive, Assigned, Consistent, Utilized, Encapsulated, Documented, Precise</td>
</tr>
<tr>
<td>EXPRESSION</td>
<td>Computation, Consistent, Effective, Encapsulated, Adjustable</td>
</tr>
</tbody>
</table>

To build-quality-into variables and expressions when we implement programs we should therefore ensure that all of the above properties are satisfied. In a similar way we can associate a set of quality-carrying properties with each of the other structural forms used in programs for a particular programming language. The result of this exercise is something tangible that programmers and others can use to produce quality software.

In the case of expressions, for example, violation of any of these properties represents a quality defect that destroys the expression's integrity. There are also other problems that can arise with expressions. For example, an expression may contain an unassigned variable. However, because of the precedence rule used in the classification discipline that is employed this defect will be classified as a variable integrity defect rather than as an expression integrity defect even though it impacts the computation of the expression.
This model for software product quality enables us to construct a powerful quality defect classification discipline. The discipline fits our normal intuition regarding defects. It focusses on the source of the problem rather than on the consequences of the defect. In the previous example there is no defect directly with the expression. Rather the problem resides elsewhere - the variable should have been assigned before being used in the expression. A crucial requirement in developing such a framework for categorizing quality defects is that the process is repeatable. That is, two people confronted with the same defect should arrive at the same classification for that defect. The present proposal attempts to achieve repeatability through the use of bottom-up structural precedence. (e.g. in the example above since variable may be contained within an expression, the variable classification for the defect assumes precedence).

Examples
Defects that violate the quality-carrying properties of expressions are given below:

- uncomputable - divide by zero
- inconsistent - contains side-effects
- ineffective - contains computational redundancy
- unencapsulated - uses variable from outside its scope
- unadjustable - contains numbers

All of the above types of defects detract from the quality of expressions. Therefore to build quality into expressions when we implement programs we should avoid these types of defects.

An important additional feature of the empirical model of software product quality that we have described is that it makes direct links between product characteristics and high-level quality attributes

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>QUALITY ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized Variable</td>
<td>Reliability</td>
</tr>
</tbody>
</table>

Property of Variable: Assigned
Quality Attribute: Reliability
Variable Defect: Unassigned

In developing this model the intent is to identify and associate a set of properties with each structural form that may be used in a program. The accompanying constructive theorem may then be used to make an assertion about the quality of a given piece of software.

Constructive Theorem
If each of the properties associated with a particular structural form is satisfied when that structural form is used in a program, it will contribute no quality defect to the software.

In other words a program is quality-defect-free when none of the quality-carrying properties of any of its structural forms are violated. On the other hand, there is the complementary assertion that if any of the properties associated with a structural form are violated in a program then each violation will contribute a quality defect to the software.

This model for software product quality is put forward as a framework that may be criticized, refined and if necessary corrected. No claim is made in the expanded treatment of this model [6] that we have listed a complete set of quality-carrying properties for each structural form. For example, at an earlier informal presentation of this model someone in the audience pointed out that an important quality-carrying property of numeric variables is that they have the appropriate precision. This is a perfectly reasonable observation and precision has therefore been added to the quality-carrying property-list for variables. What is important is that such an
addition to the model does not destroy it but instead constitutes a refinement. For something as complex and difficult to define as software quality it is important to have a model that is easy to refine both in terms of identifying additional quality-carrying properties and in attaching such properties to particular structural forms. Over time such a model has the potential to evolve into a powerful definition. Another advantage of the form-property framework is that it is easy to adapt to any kind of programming language. It is a systematic, generic yet simple model for software product quality that guides us down the path from the tangible towards quality.

The fully developed model for software product quality [6] is quite extensive. Here we have only attempted to summarize some of the features of the model. The model is important in that it allows us to achieve our two primary goals. It gives us direct advice on building quality into software and at the same time it may be used to assist in the systematic, repeatable classification of quality defects in software. These are critical steps towards scientific understanding and engineering applicability.

4.1.2. Formal Model for Software Product Quality

While the empirical model we have described is very useful in practice there are still many deficiencies in software that it cannot easily capture. We suggest that this approach needs to be complemented with a formal model for software product quality. Such a model can assist us in three important ways:

- it admits constructive verification and detection of a wider class of defects
- it provides a basis for improving and re-engineering software
- it points to a powerful measurement discipline for software quality

The feature of formal methods is that they allow us to focus on the semantics of programs. A powerful theoretical tool that can be used to help capture the semantics of program structures is the strongest postcondition [7,8]. In developing a framework for assessing, improving and measuring software quality we may exploit the application of strongest postconditions. Their important advantage is that they allow the derivation of specifications from program components. Such specifications formally and accurately characterize program semantics. A conceptual definition of a strongest postcondition is given below.

**Strongest Postcondition**

Given any Hoare triple \( \{P\} S \{Q\} \), the strongest postcondition \( Q \) is the final state such that for any variable set, if the initial value set satisfies \( P \), then the final values must satisfy \( Q \). The strongest condition at the execution point after execution of \( S \) under the precondition \( P \) is denoted by \( \text{sp}(P,S) \).

Since the strongest postcondition describes the strongest condition at each execution point, we can use it to identify a number of significant quality defects. For example, an unreachable branch is one where the precondition, i.e. the strongest postcondition of the previous statements, implies the negation of the corresponding branch guard.

**Unreachable Branch**

*Given any multi-branch statement if \( C_1 \rightarrow S_1 \) \( C_2 \rightarrow S_2 \) \( \ldots \) \( C_n \rightarrow S_n \)fi, under precondition \( P \), where \( n \geq 2 \), a branch \( C_i \rightarrow S_i \) is called unreachable if \( P \Rightarrow \neg C_i \).*

This definition formally captures an ideal for branch statements. It tells us that all branches of an if-statement should be reachable. It also defines the logical (semantic) property that is exhibited by a branch when it violates the ideal by being unreachable.
Example
Under the precondition \( x = a \land y = x \), only the second branch in
\[
\text{if } x + y > 2a \rightarrow z := x + y \quad \text{if } x + y \leq 2a \rightarrow \]
\( z := (x + y)/2 \) is reachable.

The results of strongest postcondition calculations may also be exploited to identify redundant assignment statements.

Redundant Assignment
An assignment \( S \) is redundant under a given precondition \( P \) if \( \text{sp}(P \bullet S) = P \).

This definition once again expresses the ideal for inclusion of an assignment statement in a sequence of statements. An assignment statement should only be included in a sequence if the strongest postcondition that holds after execution of the statement is not equivalent to the precondition that holds before the assignment executes. In other words every state-changing statement must change the state of a computation - otherwise it is redundant.

Example
Given \( x := 1; y := x + 3 \), if \( x + y > 2a \rightarrow y := 4 \quad \text{if } x + y \leq 2a \rightarrow z := (x + y)/2 \) under the precondition \( \text{true} \), strongest postcondition calculations enable the redundant assignment \( y := 4 \) to be found.

Elsewhere [9] a comprehensive treatment is provided which shows how methods based on the use of strong postcondition calculations may be used to construct a powerful formal model for defining, measuring and improving software product quality.

4.2 Automated Inspection - product evaluation

The empirical model of software product quality that we have sketched above is well-suited to directly linking defects associated with product characteristics to the quality attributes of the ISO-9126 Software Product Evaluation Standard [11]. For example, we have already seen how a defect in which a variable is unassigned in an expression directly impacts the reliability quality attribute.

A tool called PASS-C (Program Analysis and Style System[12]) which is based on the above empirical form-property model has been developed as a rule-based static analysis system. This system, which contains a large number of rules, is able to analyze the text of C programs and report the presence of a wide range of quality defects. The system is like a compiler in that it pinpoints the line number where each defect occurs and it indicates the nature of the defect. The system classifies the various defects both as product structural form integrity violations and as quality attribute violations such as maintainability, efficiency or reliability problems etc. It also provides a set of summary statistics about the quality of the program being analyzed and its rating against an industry average product quality profile. A typical high-level product quality summary is as follows:
Systems like PASS-C have the potential to make an important and productive interim contribution to assuring the quality of software as it is developed. They allow a significant model of software product quality to be implemented in a very efficient way. Such tools should be on every programmer’s workbench aiding him/her to efficiently monitor the quality of software as it is developed and thereby providing the means to build in quality during development. Of course, in the future as quality support is integrated into programming languages the need for such “quality compilers” should largely disappear.

### 4.3 Language Contribution

We must not underestimate the importance of language in producing quality software cost-effectively [10]. An often quoted example is the difficulty of performing arithmetic computations using the Roman number system. In contrast our present system makes such computations straightforward and mechanical. What we are suggesting is that most existing programming languages do not facilitate building quality software. By implication we also
suggest that with better languages the enormous task of building quality software will become significantly easier. This is a strong claim but one which we believe is defensible.

In making a case for this position we will start by presenting a view held by many linguists (and others): the language that we use to express solutions to problems strongly influences and limits our thinking. If this is the case, and if it also applies to programming languages then it has significant implications for the shape/architecture of such languages. We claim that the dominant programming languages that are currently in use are most suited for implementing intensive processing activities/computations - that is, algorithms and functions. Unfortunately most of the real world that we want to model consists systems and subsystems of various forms that interact with one another in a variety of different ways. Within such systems/subsystems there is often plenty of computation and large numbers of objects of varying complexity. However computations across systems rarely dominates.

Intensive computations tend to be confined within functions/programs. On the other hand only the results of intensive computations tend to be exchanged between programs or subsystems. Existing languages, because of their computational focus, have neglected to provide adequate support for effectively modelling large systems made up of loosely-coupled computationally-intensive programs and subsystems. This forces us to violate the fundamental principle of correspondence which demands that the structure of the solution should mimick the structure of the problem. This state of affairs has two important consequences:

- it distorts our thinking about problems that need to be solved using software, and
- it distorts the architecture of software systems which in turn detracts from their quality and their architectural transparency with respect to the problems that they solve.

If we are able to overcome these design flaws in existing languages then we will need languages that support not only programming but also design and reuse at a much higher level (i.e. at the system/subsystem level). Another consequence of such a change in language architecture would be that it would obviate the need for separate CASE tools. What we must do is come to the realization that programming languages should support design as well as implementation of computations because design is intimately connected with modelling the interaction between subsystems and programs. Integrating design and implementation is a worthy step forward that has the potential to make a very substantial impact on the quality of software and the productivity of software development.

People no longer want to build just programs. There is an increasing need to assemble powerful systems and subsystems - however most programming languages do not support this. Instead they are structurally too flat. This causes us to distort our thinking when trying to solve large problems. Our languages should make it easy to design not only algorithms and programs but also rich systems structures made up of many interacting programs and subsystems. Object-oriented programming languages go only part of the way by modelling objects. In order to control complexity and achieve natural interaction between subsystems and extensive reuse we need to limit the interaction between objects. The object-oriented approach only deals with low level reuse. Much greater progress will occur when we start to reuse whole systems and subsystems rather than just objects.

The language SAFE which seeks to respond to the quality-support deficiencies with existing languages is discussed elsewhere [13].

4.4 Reengineering

In our quest for product quality improvement we must not ignore the very large body of software already in existence. In any one year the amount of new software written is very small compared with the huge amount of software already in existence which keeps most large
Software quality and productivity improvement

organizations operational. The problem with this large resource of legacy code is that much of it
is of questionable quality and difficult to maintain. However so much money has already been
invested in this software that it cannot be easily thrown out and replaced by new and better
software. In many instances organizations are stuck with having to maintain and modify large
tracts of poor quality software. The problem with this is that it is a very costly and labour­
intensive activity.

Strongest postcondition calculations provide a powerful framework for formally re­
engineering software to produce improved program structures that satisfy the original
specification [14]. In this section we will show how formal methods may be used to remove
redundant statements and simplify the bodies of loops.

4.4.1 Removing Redundant Statements.

Strongest postcondition calculations coupled with simplifying logical equivalence
transformations provide a powerful means for removing redundancy from complex structures.

Example:
Consider the following highly redundant Abstract Program AP suggested by Manna [15].

\[
\begin{align*}
\text{If } p(y) \rightarrow & \text{} x:=y; \text{ do } \neg p(x) \rightarrow x:=f(x) \text{ od} \\
\text{ [] } & \neg p(y) \rightarrow \\
& x:=a; \text{ if } p(x) \rightarrow \text{ do } \neg p(x) \rightarrow x:=f(x) \text{ od} \\
& x:=f(x); \text{ if } p(x) \rightarrow \\
& \text{ do } \neg p(x) \rightarrow x:=f(x) \text{ od} \\
& f l \\
& f l
\end{align*}
\]

The strongest postcondition calculation for this Abstract Program under the initial precondition true
yields[ ]:

\[
\begin{align*}
\text{sp(true, AP)} \\
= \text{sp(true } \land p(y), x:=y; \text{ do } \ldots \text{ od) } \lor \text{sp(true } \land \neg p(y), x:=a; \text{ if } \ldots \text{ fl) } \\
= (p(y) \land x=y) \lor (\neg p(y) \land p(a) \land x=a) \lor (\neg p(y) \land \neg p(a) \land x=f(a))
\end{align*}
\]

The calculated strongest postcondition characterizes the behaviour of the above program and the
formally transformed equivalent program below from which all logical redundancy has been removed:

\[
\begin{align*}
\text{If } p(y) \rightarrow & \text{} x:=y \\
\text{ [] } & \neg p(y) \rightarrow \text{ if } p(a) \rightarrow \text{} x:=a [\neg p(a) \rightarrow x:=f(a) \text{ fl} \\
\text{ fl}
\end{align*}
\]

4.4.2 Restructuring Complex Loops

Loops are frequently poorly structured. We may use a similar approach to that described in the
previous section to identify the reachable branch set for loop structures. The formal process for
re-engineering branched loop structures is called loop rationalization. It is described in detail
elsewhere[14]. Here we will only briefly summarize the process. We start with a branched
loop body and assume that Inv is the invariant of the loop (when the loop invariant Inv is not
available the weakest invariant true is used). We then proceed to calculate the strongest
postcondition for each branch, i.e.
\[ \text{do } G \rightarrow \]
\[
\text{if } C_1 \rightarrow S_1 \quad \{ \text{sp}(G \land \text{Inv} \land C_1, S_1) \}
\]
\[
\text{if } C_2 \rightarrow S_2 \quad \{ \text{sp}(G \land \text{Inv} \land C_2, S_2) \}
\]
\[
\text{...} \quad \{ \text{sp}(G \land \text{Inv} \land C_n, S_n) \}
\]
\[
\text{od,}
\]
\[
\text{where}
\]
\[
\forall i \forall j (i,j) \in [1, n] \land j \\
(C_i \Rightarrow C_j)
\]
\[
\text{and}
\]
\[
C_1 \lor C_2 \lor \ldots \lor C_n = \text{true} \quad \text{for } i,j \in [1, n].
\]

These \( n \) postconditions \( \text{sp}(G \land \text{Inv} \land C_i, S_j) \) determine the successor set of each branch, that is, \( \beta_i = \{1, 2, \ldots, n\} - \alpha_j \), where \( \forall j (j \in a_j \Rightarrow (\text{sp}(G \land \text{Inv} \land C_i, S_j) \Rightarrow \neg (G \land C_j)) \). We may assign each branch \( C_i \rightarrow S_i \) to a separate node of a directed graph, then connect all nodes that \( \beta_i \) indicates, to form the \textit{branch successor graph} [14].

\textbf{Example:}

Given the following poorly formed loop structure taken from a commercial program:

```c
while ((Option = getopt(argc, argv, Voptions)) != ERROR) {
    if ((char) Option == Optionq) {
        QuitFlag = TRUE;
        if (LevelFlag) {
            Error = TRUE;
            break;
        }
        if (SubsystemFlag) {
            Error = TRUE;
            break;
        }
    }
    else {
        if ((char) Option == Option) {
            LevelFlag = TRUE;
            if (QuitFlag) {
                Error = TRUE;
                break;
            }
            LevelStr = optarg;
        }
    }
    else {
        if ((char) Option == Options) {
            SubsystemFlag = TRUE;
            Subsystem = optarg;
            if (QuitFlag) {
                Error = TRUE;
                break;
            }
        }
    }
    else {
        if ((char) Option == "?") {
            Error = TRUE;
            break;
        }
    }
}
```

Stage I: Guard Resolution

As part of the loop reengineering process we must first resolve the guards associated with each branch. This amounts to calculating all their strongest preconditions - that is we find the condition that applies directly before execution of the assignments in the branch. The result of this process is shown below:

while ((Option = getopt(argp, argv, Vailoptions)) != ERROR) {
    if (Optionq) { LevelFlag = TRUE; Error = TRUE; break; }
    if (Optionq) { LevelFlag = TRUE; Error = TRUE; break; }
    if (Optionq) { LevelFlag = TRUE; Error = TRUE; break; }
    if (Optionq) { LevelFlag = TRUE; Error = TRUE; break; }
    fi
}

Branches that do not cause termination are shown in bold

Stage II: Calculation of Strongest Postconditions for Each Branch

The strongest postconditions that apply after each branch is executed must then be calculated. The calculation for the third branch is shown below.

Precondition for Execution of Third Branch

((Option = getopt(...)) == Option) \wedge \neg LevelFlag \wedge \neg SubsysFlag

Third Branch

{ QuitFlag = TRUE; }

Strongest Postcondition After Execution of Third Branch

((Option = getopt(...)) == Option) \wedge \neg LevelFlag \wedge \neg SubsysFlag \wedge QuitFlag

Successors Accessible After Execution of Third Branch

Comparing the strongest postcondition result with each of the branch guards it is clear that the branches 1, 2, 5, and 7 will not be accessible directly after the third branch has executed. The successors of the third branch are therefore:

{3, 4, 6, 8}

In a similar way we can perform strongest postcondition calculations to determine the successor sets for each of the other branches. We may then use this information to build the branch successor graph for the loop structure. The branch successor graph for our example above is shown in the leftmost part of the figure below:
Stage III: Loop Rationalization

Once the BSG has been constructed we have the information we need to transform the loop into a normalized form. Loop rationalization involves removing all the terminating branches from the loop body (i.e. those shown in stage I that are not in bold). These terminating branches correspond to leaf nodes in the BSG. The result is shown below.

```c
    do ((Option = getopt(...)) => Optionq)∧LevelFlag∧SubsystemFlag → QuitFlag = TRUE
      [] ((Option = getopt(...)) => Optionl) ∧QuitFlag → LevelFlag = TRUE; LevelStr = optarg
      [] ((Option = getopt(...)) => Options) ∧QuitFlag → SubsystemFlag = TRUE; Subsystem = optarg
    od;
    if (Option == Optionq)∧LevelFlag → QuitFlag = TRUE; Error = TRUE
      [] (Option == Optionq)∧LevelFlag∧SubsystemFlag → QuitFlag = TRUE; Error = TRUE
      [] (Option == Optionl)∧QuitFlag → LevelFlag = TRUE; Error = TRUE
      [] (Option == Options)∧QuitFlag → SubsystemFlag = TRUE; Subsystem = optarg; Error = TRUE
    [] (Option == "") → Error = TRUE
```

We will explain later details of the accompanying metric calculations involved for this example. If a loop structure has a branch-successor graph (BSG) that is made up of only strongly connected components then it contains no latent inefficiencies and redundancies. On the other hand, if a BSG is not strongly connected it can be improved by processes of loop rationalization and loop normalization. Loop rationalization is a re-engineering technique based on strongest postconditions calculations that leads to the removal of all terminating branches from the loop body. In our example above these transformations have led to the construction of a loop with a considerably simpler body. Its BSG (see rightmost graph above), consists only of strongly connected components.

4.5 Measurement

There is a large body of knowledge [16] relating to patterns of composition, structuring, modularization and methods of consistency, proof and verification which provide important, but not metric indications of quality. The one thing that links most, if not all these indicators of quality, is semantics. If semantics can be used as the basis for software measurement then it has the potential to provide more direct and meaningful measures of software quality. Strongest postconditions allow us to tap the semantics of program structures. Their important advantage is that they may be used to derive specifications directly from program components. Such specifications formally and accurately characterize program semantics.

To support the semantics approach to measurement it is necessary to have an accompanying quantitative discipline. There are two primary choices: absolute versus relative measurement. To date most software metrics have focussed on absolute measurement. In the present proposal we will seek to exploit the principle of always measuring things with respect to an ideal. A primary motivation for measurement is to seek improvement. Measurement with respect to an ideal has the important property of being constructive in relation to quality improvement and re-engineering [17]. The key challenge is to find ways to meaningfully characterize the ideal. The exploitation of semantics for software measurement is directly compatible with, and supportive of, the application of this relative measurement principle. Based on the principles of formal semantics and relative measurement we can define a number of important measures of software quality.

In applying the proposed measurement methodology strongest postcondition calculations are used to evaluate software and return a number of values, called p-numbers. These numbers, which are ratios, each provide indications of quality relative to an ideal for the particular property that is being measured. If a chosen software component exhibits the ideal with respect to the property that is being measured then the corresponding ratio will have the value 1.
In what follows we will illustrate the flavour of the approach by considering a reachability ratio. We claim the reachability ratio provides a much more informative and useful metric than the McCabe Number because it can be used to guide the improvement of the software.

The $\rho_d$-number is used to measure the reachability of a branched software component. It defines the ratio of the number of reachable executable branches for a component relative to all possible branches that the syntax (not the semantics) indicates should be reachable for that component. The reachability of a multi-branch statement is defined as:

$$\rho_d = \frac{\text{number of reachable branches}}{\text{total number of branches}} = \frac{n - n_u}{n}$$

where the number of unreachable branches $n_u$ under a precondition are ascertained by applying the necessary implicative requirements for execution. Obviously, $1/n \leq \rho_d \leq 1$. Reachability calculations for a loop structure are given below.

**Example:**

We can calculate the internal reachability for the poorly structured loop in section 4.4. In this case the BSG has 8 nodes and ideally each node should be accessible from every other node. Counting up the edges and self-rings in the BSG we find that there are only 13 connections instead of $8 * 8 = 64$ and so the internal reachability for the loop structure is only:

$$\rho_d = \frac{13}{64}.$$  

This indicates that there is a lot of unnecessary structure in the loop. The validity of this measure is supported by the fact that, as we have shown in the previous section, five branches are able to be removed from the loop body in the re-engineered implementation.

Other formal measures that can be formulated along these lines relate to redundancy, reusability, cohesion and logical consistency in the use of variables. These measures are discussed in detail elsewhere [17].

### 4.6 Reuse

Earlier in our discussion of language contributions to quality and productivity improvement we considered how they should be designed to support the reuse of large and complex systems and subsystems. Here we wish to consider a more fundamental reuse problem which is inhibiting existing reuse efforts. Clearly until reuse technology can be made easy to use, versatile and powerful large numbers of software professionals are not going to show a serious interest in reuse. Object class libraries go part of the way to addressing this problem. However the fundamental problem that we still face in reuse is how to characterize the functionality of software components so they are easy to store and retrieve from large libraries. People have proposed keyword and facet-based systems [18]. However such systems by no means constitute a general, widely applicable solution to the problem because of the tenuous links between keywords and functionality.

The use of formal specifications offers some hope in that formal specifications in some instances at least are able to accurately capture functionality. There is however the problem that there are many different ways of formally characterizing the same functionality. Fortunately there is one possible way to overcome this obstacle - we can exploit the idea of constructing a canonical notation from formally expressed specifications. What a canonical notation does is allow us to map differently expressed but formally equivalent specifications to a single unique
canonical form. We would suggest that this approach offers some prospect for in the longer term building powerful reuse systems. Early results using such an approach are discussed elsewhere [2]. At the very least, such an approach has the potential to support the construction of safety critical software.

Conclusion

Order-of-magnitude improvements in software quality and productivity will require a concerted and systematic effort on a number of fronts. There is a need to combine the contributions of both empirical and formal approaches. A process orientation is certainly important but it will need to be guided by comprehensive and systematic models of software product quality. Better languages offer the greatest scope for building quality into software directly. Formal methods also offer considerable opportunities for re-engineering, reuse and meaningful measurement that can point directly to quality improvement and the development of powerful tools that will help us to produce quality software. In our efforts to produce quality software cost-effectively there is plenty of scope to be industrious - the challenge is to be effective.

REFERENCES

4. Technology for Evaluation and Certification of Software Product Quality, SCOPE, ESPRIT Project 2151
5. Deming, W.E., Out of the Crisis,