Some Directions for the Measurement of Objects

Mark Harman and Sebastian Danicic
Project *Project*
School of Computing
University of North London
Eden Grove, London, N7 8DB
tel: +44 71 753 5751
fax: +44 71 753 7009
e-mail: m.harman@unl.ac.uk

Keywords: Slicing, Measurement theory, Significance, Publicness, Cohesion

Abstract

This paper outlines some approaches to the measurement of classes based upon data flow analysis techniques applied to the data members of the class.

Metrics based upon the sets of data members defined and referenced by member functions are considered, together more finely grained metrics based upon slices constructed from member functions. Ott and Bieman's cohesion metrics are adapted for object oriented software and a measure of a data member's significance is proposed. Slicing is put forward as a basis for the measurement of a data member's publicness and significance and of the cohesion of a class.

It is also argued that such measurements are, by definition, approximations, as the values upon which they are calculated are inherently non–computable. This observation may have an impact upon the theory of scales of measurement.

1 Introduction

Following Chidamber and Kemerer's seminal work on object oriented metrics [12, 13], much work has been undertaken to define measurements for attributes of object oriented code, much of which has been concerned with design [1, 11, 31, 44, 55, 57], reusability [22, 39, 4, 6, 26] and traditional software code measurement concerns such as complexity [8, 27, 29, 54].

Robin Whitty at Southbank University Centre for Systems and Software Engineering provides a long, annotated (and electronically available [53]) list of source references concerned with Object Oriented Metrics. Mark Lorenz and Jeff Kidd [36] provide an extensive survey text on the subject.

In this paper it is argued that analyses associated with data flow properties of a program are highly attractive as a basis for the measurement of class attributes. In particular, program slicing is used as a basis for the measurement of the cohesion of a class and the level of significance and publicness of its data members.

Section 2 outlines some of the observations that can be made about the data members and member functions of a class based upon the defined and referenced variable sets of its member functions. Section 3 reviews program slicing, which is used in section 4 to measure cohesion, in section 5 to measure the data member significance and in section 6 to measure the level of publicness and, by extension, encapsulation enjoyed by a class. Section 7 raises some issues associated with the approximate nature of measurements based upon non–computable quantities such as program slices, arguing that this 'approximate nature' is not only unavoidable, but also prevalent in existing and widely used metrics.

```
class Time {
public:
     Time();
     void set(int, int, int);
     void print();
     void add(int, int, int);

private:
     int hr;
     int min;
     int sec;
};
```

Figure 1: A Simple Class Definition

2 Defined and Referenced Data Members

In this section all data members will be assumed to be *private*. In section 6, analysis techniques for describing properties of public data members will be considered.

In this paper, the concept of what constitutes a 'member function' (or 'method') will be regarded as a parameter to the metrics proposed. Churcher and Shepperd [14] have pointed out that the 'assumed understanding' of what constitutes a method suffers from similar vagaries to the concept of a 'line of code' in classical metrics literature. Churcher and Shepperd suggest various possibilities depending upon whether or not operators, inherited methods and macro–defined methods are considered to 'count'. Each of these choices is independent, and the resulting combination of choices leads to a considerable number of possibilities for the number of methods which are considered to be 'in the class'.

However, this choice does not affect the structure of the metrics proposed here, merely the values they calculate. Indeed, the metrics are also constructed with respect to a notion of 'amount of code', which is, for the sake of exposition, instantiated to 'Lines of Code'.

In order to describe properties of classes, some notation will be introduced to describe the affected and needed variables of the member functions of a class.

Definition 1 (Affected)
The *affected* variable set, $\mathcal{A}(s)$, of a statement s consists of those identifiers whose value changes for at least one of the initial states in which s may be executed.

Definition 2 (Needed)
The *needed* variable set, $\mathcal{N}(s)$, of a statement s consists of those identifiers whose value in some initial state affects the computation of s.

The affected and needed variables of a statement can be approximated by the sets of syntactically defined and referenced variables of the statement.

Definition 3 (Defined)
An identifier, i, is in the set of *defined*[1] variables, $\mathcal{D}(s)$, of a block of code s iff i occurs on the left–hand side of an assignment statement in s.

Definition 4 (Referenced)
An identifier i is in the set of *referenced* variables, $\mathcal{R}(s)$, of a statement s iff i is live [3] at the entry node of the Control Flow Graph (CFG) of s.

[1]This definition applies to side–effect free programs only. For programs with side effects, the defined variables of *expressions* must also be included in the set of defined variables.

Note that $\mathcal{A}(s) \subseteq \mathcal{D}(s)$ and $\mathcal{N}(s) \subseteq \mathcal{R}(s)$. The set of defined variables of s is thus an approximation to the the set of variables whose values are affected by some computation of s (the affected variables of s). The set of referenced variables of s is an approximation to the set of variables whose initial values affect the computation of s in some state (the needed variables of s). The issues for measurement theory implied by such approximations are discussed in more detail in section 7, until which time it will be assumed that these approximations are 'perfect', that is $\mathcal{A} = \mathcal{D}$ and $\mathcal{N} = \mathcal{R}$.

Suppose that from the class in figure 1 the defined and referenced variables of the member functions `set` and `Time` are calculated, and it is found that $\mathcal{D}(\texttt{set}) = \{\texttt{hr}, \texttt{min}, \texttt{sec}\}$, $\mathcal{R}(\texttt{set}) = \{\}$, $\mathcal{D}(\texttt{Time}) = \{\texttt{hr}, \texttt{min}, \texttt{sec}\}$ and $\mathcal{R}(\texttt{Time}) = \{\}$.

This information suggests that both `set` and `Time` are 'constructors'. In general, any member function which has an empty referenced variable set and a non–empty defined variable set which contains at least one data member can be regarded as a constructor. Such a member function will alter the value of one or more data members of a class and the alteration will be *independent* of the value of any of the data members of the class. Unfortunately the term 'constructor' already has a specific meaning in object–oriented nomenclature, and so the less pleasing term 'definer' will be used for member functions which define a non–empty set of data members, but which do not reference any data members in a class.

Similarly, a function which only references data members, but does not defined the value of any data member can be viewed as a selector[2].

A function which defines and references data members, performs some transformation on objects in the class. Among these transformations, a distinction can be made between those which *redefine* the values of data members in terms of their previous values and those which 'copy' information from one part of the object's data space to another. The former variety will have a non–empty intersection between their defined and referenced data members, whereas, for the later, defined and referenced sets will be disjoint.

The preceding informal observations allow for the introduction of some straight forward definitions which allow data members to be partitioned into five disjoint categories:

Definition 5 (Class Components)
Let C be a class. The set of data members in C will be denoted $\mathbf{D}_C$, the set of member functions in C will be denoted $\mathbf{F}_C$.

Definition 6 (Definer)
Let C be a class. A function $f \in \mathbf{F}_C$ is a *definer* iff $\mathcal{D}(f) \cap \mathbf{D}_C \neq \emptyset \wedge \mathcal{R}(f) \cap \mathbf{D}_C = \emptyset$.

Definition 7 (Selector)
Let C be a class. A function $f \in \mathbf{F}_C$ is a *selector* iff $\mathcal{R}(f) \cap \mathbf{D}_C \neq \emptyset \wedge \mathcal{D}(f) \cap \mathbf{D}_C = \emptyset$.

Definition 8 (Mover)
Let C be a class. A function $f \in \mathbf{F}_C$ is a *mover* iff $\mathcal{D}(f) \cap \mathcal{R}(f) \cap \mathbf{D}_C = \emptyset \wedge \mathcal{D}(f) \cap \mathbf{D}_C \neq \emptyset \wedge \mathcal{R}(f) \cap \mathbf{D}_C \neq \emptyset$.

Definition 9 (updater)
Let C be a class. A function $f \in \mathbf{F}_C$ is an *updater* iff $\mathcal{D}(f) \cap \mathcal{R}(f) \cap \mathbf{D}_C \neq \emptyset$.

Definition 10 (anomaly)
Let C be a class. A function $f \in \mathbf{F}_C$ is an *anomaly* iff $(\mathcal{D}(f) \cup \mathcal{R}(f)) \cap \mathbf{D}_C = \emptyset$.

Definition 10 highlights a strange situation; a member function which has no defined or referenced data members. Perhaps such a function should *not* be a member function. Using this simple DEF–REF analysis, two other anomalous situations can be detected:

- A data member which is not referenced by any member function. Such a data member can only be stored in an object, but can never affect any computation.
- A data member which is not defined by any member function. Such a data member can only be referenced, and is therefore only feasible if initialised with a constant in the class declaration.

[2]Such a function may not return a value, and so is not a *selector* in the strict sense of an Abstract Data Type. However, the function will not alter the value of any data member when invoked, and its behaviour *will* be dependent upon the values of the data members at invocation time, so it seems reasonable to describe it as a selector in a slightly less rigorous sense.

1 read(n); 2 s:=0; 3 p:=1; 4 while n>0 do 5 begin 6 s:=s+n; 7 p:=p*n; 8 n:=n-1 9 end	1 read(n); 2 s:=0; 3 4 while n>0 do 5 begin 6 s:=s+n; 7 8 n:=n-1 9 end
Original Program	Slice w.r.t. $(\{\mathtt{s}\}, 9)$

Figure 2: A Program and One of its Slices

3 Program Slicing

A number of more finely–grained metrics can be defined for classes using program slicing [52].

Conceptually, slicing is attractively simple and easy to define – all those statements which *cannot* affect the values of variables of interest are deleted to form the slice. Definition 11 below, provides an informal definition of a slice. Figure 2 shows a program fragment and one of its slices.

Slicing is attractive as a basis for measurement because it captures the 'tracing behaviour' observed in psychological studies of programmers [51, 11]. This would appear to make slice–based metrics particularly sensitive to the conceptual effort required during comprehension [17, 25] and maintenance activities [10, 21].

Definition 11 (Backward Slice)
A slice of a program p, at a line number n, with respect to a set of variables K, is a program p', constructed by deleting certain commands from p. The commands which may be deleted are those which can have no effect upon the values residing in any of the variables in K, when execution reaches line n. The pair (K, n) is known as the 'slicing criterion'.

Slicing finds many applications to software engineering problems such as algorithmic debugging [45, 30], re–engineering [33, 47], testing [23], reuse [5] and maintenance [21, 38]. All of these applications are based upon the way that a slice is a simplified version of the original which maintains a projection of its semantics.

Much work has been carried out over the past few years to improve and extend slicing algorithms [2, 16, 18, 49, 28]. The technology has advanced to the point where a slicing tool, `unravel`, developed for the National Institute of Standards and Technology, is capable of producing slices of almost any[3] ANSI C program [37]. Tip [50] and Binkley and Gallagher [9] provide detailed surveys of the paradigms, applications and algorithms for program slicing.

For the calculation of object oriented metrics both forward and backward slices are useful. Definition 11 defines what has come to be known as a 'backward slice', after the introduction of forward slicing by Horwitz et al [28]. A forward slice [28, 50] constructed for a slicing criterion (V, i) contains all the statements whose behaviour depends (transitively) upon the value of at least one variable in V at i.

In this paper, backward slices will always be constructed for a single variable at the 'end' of the original program fragment to be sliced, similarly, forward slices will be constructed at the 'start' of the fragment to be sliced. Thus backward slices contain statements upon which a code fragment depends for the computation of variables in the set V, whilst forward slices contain those statements affected by the initial value of a chosen variable. The notation $S^B_{(P,v)}$ will be used to denote the backward slice of P with respect to $(\{v\}, n)$, where n is the 'end' of P. The notation $S^F_{(P,v)}$ will be used to denote the forward slice of P with respect to $(\{v\}, n)$, where n is the 'start' of P.

The next three sections using forward and backward slices as a basis for many properties of classes, objects and data members.

[3] `goto` statements are not handled by `unravel`, but these could easily be incorporated.

4 Cohesion

Bieman, Ott et al [7, 43, 42, 41, 40, 34, 35, 48, 24], suggest that the overlap of a section of code's slices represents the code's cohesion. Lakhotia [32] suggests a similar approach, in which the seven levels of cohesion introduced by Constantine and Yourdon [15] are defined in terms of data flow relationships in the program.

Using the approach of Bieman et al, the cohesiveness of a code fragment is measured with respect to a set of variables V. The metric describes the 'amount' of code in the overlap of slices constructed for variables in V, relative to the size of the fragment as a whole. Ott calls this $Tightness$ [43], and introduces a number of similar measurements, all of which are based upon the slices constructed for variables in the set V. Ott's tightness metric is defined below[4]:

Definition 12 (Tightness)

$$Tightness(P, V) = \frac{\#(\bigcap_{v \in V} S^B_{(P,v)})}{\#(P)}$$

For Ott, the set of variables of concern, V, is the set of variables which are

- involved in an output statement
- global
- reference parameters

For the purpose of measuring class cohesion, the set V will be the set of data members defined by a function. The tightness measure (and Ott's related measurements) will thus provide a measure of the cohesion of a member function with respect to the data members it defines. The cohesion of a class could then be obtained by taking the mean[5] value of Ott's measure of tightness for each member function with respect to the data members its defines.

This idea is highly tentative as there is an assumption that the cohesion of a class is an answer to the question "how interwoven are the computations performed on the data members?". However, perhaps, it is not sensible to expect the computation on data members to be cohesive in this way - it would be to demand high coupling between the member functions of the class. However, the idea behind work of Bieman et al does extend quite naturally to the measurement of other properties of data members and objects, which are introduced in the following two sections.

5 Data Member Significance

In tasks related to maintenance and reuse of classes, it may be useful to rank the data members of the class according to some measurement of their overall significance in the class. Such a measurement could be used to assess the likely impact of a change in the way that data is structured, and perhaps guide the choice of where, in the data space of the class, test effort should be expended.

A rather crude measure of how 'significant' a data member is in a class can be obtained by simply counting the number of functions for which the data member is a defined variable. This could be termed the *backward significance* of the data member, because the more backward significance a data member has, the more impact the member functions of the class will have upon any part of a program which depends upon the data member.

Similarly, an alternative measure of how significant a data member is in a class can be obtained from counting the number of functions which reference it. This could be termed the *forward significance* of the data member. Objects will be more sensitive to changes in the values of data members with a high measure of forward significance than those with lower values.

[4] where # denotes set cardinality.

[5] But here there is an important issue – the scale upon which the measurement of cohesion of individual functions takes place. It is perfectly justified to define a measure of class cohesion as the mean measure of the individual readings of Ott's Tightness measure for each member function. It is *not* reasonable, however, to say that the cohesiveness of a class is obtained from 'the mean cohesiveness of its member functions'. Cohesion is (according to Ott and Bieman [7]) measured on an ordinal scale. Thus it is meaningless to speak of 'mean cohesion'.

The overall significance of a data member could be defined to be the cardinality of the union of the defining and referencing member functions of the data member. However, a more finely grained metric could be constructed in terms of the *slices* of member functions with respect to the data member. This could be thought of as simply a measurement of the significance of a global variable d, among a set of functions F, giving rise to the definition of backward significance and forward significance in definitions 13 and 14 below:

Definition 13 (backward significance)
The *backward significance* of a data member d with respect to a set of member functions F is

$$\frac{\sum_{f \in F} \#(S^B_{(f,d)})}{\sum_{f \in F} \#(f)}$$

Definition 14 (forward significance)
The *forward significance* of a data member d with respect to a set of member functions F is

$$\frac{\sum_{f \in F} \#(S^F_{(f,d)})}{\sum_{f \in F} \#(f)}$$

6 Public Data Members

There is a prevalent philosophy which views public data members as a rather bad thing. This view is motivated by the desire for *encapsulation*. One consequence of encapsulation is that an object can be understood in isolation, independent of the context in which it is used. Clearly the presence of public data members detracts from the level of encapsulation enjoyed by a class, since public data members may be referenced and defined by code other than that contained in the member functions of the class. It would be natural therefore, to count the number of public data members (or perhaps the ratio of public to private data members) as an obvious measurement of the level of encapsulation enjoyed by an object.

Once again, using slicing, a more finely grained measurement may be calculated from an object *with respect to the context in which it is used*. Using backward slicing the amount of code outside the class which defines an instance of a public data member (the backward publicness of the data member instance) may be calculated. Similarly the amount of code outside the class which depends upon a data member instance (its forward publicness) can be measured using forward slicing.

Here, a distinction is required between an instance of a class and the class itself. Using the ratio of public data members to data members in general, gives a measurement of publicness which is *insensitive* to the *use* of the class in some context (i.e. in some *program*). The philosophy adopted in this paper, is that the level of publicness of a class depends upon the way in which instances of the class are used in some context.

Based upon the measurement of the publicness of an instance of a class, it will be possible to suggest a measurement of the overall publicness of a class, but this will still be dependent upon the *context* in which the class is placed.

Definition 15 (Complement of a class)
Let p be a program containing a class c. The complement of c with respect to p, written $\mathcal{C}^c_p$ is the set of all functions in p which are not member functions of c.

Definition 16 (Instance Publicness)
Let o be an instance of a class c in a program p and let d be a data member of c.

The *backward instance publicness* of d in o wrt p is given by:

$$\frac{\sum_{f \in \mathcal{C}^p_c} \#(S^B_{(f,o.d)})}{\sum_{f \in \mathcal{C}^p_c} \#(f)}$$

The *forward instance publicness* of d in o wrt p is given by:

$$\frac{\sum_{f \in \mathcal{C}_c^p} \#(S^F_{(f,o.d)})}{\sum_{f \in \mathcal{C}_c^p} \#(f)}$$

The *instance publicness* of d in o wrt p is given by:

$$\frac{\sum_{f \in \mathcal{C}_c^p} \#(S^B_{(f,o.d)} \cup S^F_{(f,o.d)})}{\sum_{f \in \mathcal{C}_c^p} \#(f)}$$

The instance publicness of a data member d in an instance o of a class c, measures the ratio of the amount[6] of code (relative to the size of the whole program), which either depends upon (forward instance publicness) or affects (backward instance publicness) d in a particular instance of c. Instance publicness of a data member d in an instance o can thus be thought of as a measure of the

"amount of code outside the class which involves the d component of o".

Backward and forward instance publicness are calculations based upon backward and forward slices respectively. Observe that instance publicness of a data member (in general) is *not* regarded as the *sum* of the backward and forward instance publicness measures, since some statement in a function f may be a member of both the backward and of the forward slice of f (this is a decision in the construction of the metric which is certainly open to question).

Instance publicness may be useful in comparing objects with one another, making inferences concerning different patterns of use. Alternatively, it may be used, and perhaps with greater justification, to measure the contexts into which an object is placed. It would also be useful to make more general inferences about the publicness of data members irrespective of the instances of these data members. A natural step, seems to be to take the mean reading for each instance of a class in a program p for a data member d.

Thus a measure of 'class publicness' is arrived at:

Definition 17 (Class Publicness)
Let I be the set of instances of a class c in a program p. Let d be a data member of c.

The *backward class publicness* of d is the mean backward instance publicness of d for the set of instances I, where backward instance publicness is calculated according to definition 16.

The *forward class publicness* of d is the mean forward instance publicness of d for the set of instances I, where forward instance publicness is calculated according to definition 16.

The *class publicness* of d is the mean instance publicness of d for the set of instances I, where instance publicness is calculated according to definition 16.

Finally, for gross measurement of a class publicness,in a context, a measurement of overall publicness can be arrived at, by considering the publicness of each of the data members of the class:

Definition 18 (Publicness)
The publicness of a class c with respect to a context p is the mean publicness of the data members of c, where class publicness is measured according to definition 17.

Observe that in this metric for publicness, both public *and* private data members are given equal weight. Naturally, private data members will score zero according to definition 16 of instance publicness in *any* context. However, such private data members should be included in the calculation of the overall publicness of the class. To exclude them, would be to unduly 'penalise' a class definition in which data members were routinely described as private.

[6]How this 'amount' is measured is a separate issue. Here the amount of code in a sequence of statements is taken to be the number of lines of code in the sequence. Other possibilities clearly exist, leading to a concept of higher order metrics [24].

Scale Type	Transformation	Proposed Metric
Nominal	any 1-1 function	definer, selector, updater, mover
Ordinal	any monotonic increasing function	Cohesiveness
Interval	$f(x) = mx + c$	
Ratio	$f(x) = mx$	Significance, Publicness
Absolute	$f(x) = x$	Number of Affected, Needed Variables

Figure 3: Measurement Scales

Also, observe that, because the measurement of publicness is not based merely upon a count of the number of public data members in a class, such a measurement is sensitive to the context in which a class is used. The measurement of 'how public' a data member d is (the publicness of d), and by extension the level of encapsulation enjoyed by a class, is thus viewed as a function of the class *and* the context in which it is used.

7 Measurement Theory

This section discusses some issues raised by the approximate nature of the measurements described so far. This may have a bearing upon the view of the scales of measurement used to quantify the attributes considered in this paper, and perhaps, those measured by existing code complexity metrics.

7.1 Scales of Measurement

Figure 3 summarises the scales [46, 19] upon which the metrics proposed in this paper should be measured.

Scales of measurement [20, 56] delimit the inferences which may be drawn from the calculation of a metric value. Such a value will always be a number (usually a real number) and so it will be tempting to make inferences about two subjects for which metrics have been calculated. These inferences may, unwittingly, inherit known properties of the number system in which the measurements are made. For example it is always possible to find the mean value of two real numbers. Such inferences need to be treated with caution.

For many attributes the real number scale is simply *too prescriptive*. For example, if two people are measured for intelligence using an IQ test, two numbers will be obtained. It would be meaningful[7] to say

> "if one person obtains a higher score than the other then this person is 'more intelligent' than the person with the lower score."

However, should one person achieve a score twice as high as the other, it would not be meaningful to say that this person was 'twice as intelligent' as the other; the set onto which individuals are mapped by the 'intelligence metric' is not as rich as the real number scale.

In constructing inferences based upon measurement in a real number system it is essential to imbue the number system with a richness appropriate to the inferences which can reasonably be made concerning the attribute being measured. One way in which this can be achieved is to define *admissible transformations* for a metric. A transformation $\mathcal{T}$ on a metric function f is admissible if all and only those inferences which can be drawn from f can also be drawn from $\mathcal{T} \circ f$. For example, given two metrics for measuring intelligence, f and f', it would be expected that f would order individuals in the same way as would f'. In this case an admissible transformation, $\mathcal{T}$, must be a monotonic increasing function. That is $f' = \mathcal{T} \circ f$ and $x \leq y \Rightarrow \mathcal{T}(x) \leq \mathcal{T}(y)$.

[7] Although, notwithstanding measurement theoretic concerns, the authors believe IQ tests to be dubious.

The scales of measurement proposed by Fenton [19] are reproduced in figure 3, indicating the scale upon which the authors believe the metrics proposed in this paper should be measured. The table in figure 3 is organised with richer metrics further down the table. A metric m is richer than a metric m' if m allows more of the properties of the real number scale than m' to be exploited in constructing inferences concerning the metrics calculated. Richer scales of measurement admit smaller sets of transformations to be applied to the values calculated upon these scales. In a nominal scale the metric values of two subjects are only significant inasmuch as two subjects with the same value have the same 'amount' of the attribute and two with differing metric values have different 'amount' of the attribute. This is the least rich of the measurement scales, it allows the set of subjects to be partitioned into equivalence classes and no more. The richest measurement scale, the absolute scale, allows no transformation to be performed upon metrics, because the metric value captures exactly, as a number, the amount of the attribute possessed by the subject.

7.2 Necessarily Approximate Scales

Over and above the important issue of the appropriate scale of measurement which should be used to calculate metrics of software attributes, the calculations upon which many metrics are based are not computable. For example, all the measurements introduced in this paper been based upon:

- affected variables,
- needed Variables and
- Slices

Even an *absolute* measure like 'number of affected variables' is an *approximate* measurement, if the view is taken that an 'affected variable' is one whose value may change during some execution of the program.

The statement

```
x = x ;
```

has `x` as both a defined and a referenced variable. However, the statement neither affects, nor depends upon the value of `x` during any execution of the program.

An alternative to accepting the approximate nature of metrics based upon these calculations would be to define the metrics in terms of the values actually calculated in defined and referenced variable sets and in terms of a particular slicing algorithm.

This can make the definition of such measurements rather cumbersome, and so perhaps it is better to 'pretend' that these measurements are precise, but to be careful to define which method of calculating defined and referenced variables is to be used and which slicing algorithm is to be adopted. Perhaps the metrics considered here are really forms of prediction system. For example, defined variables are undoubtedly a good predictor of affected variables since $\mathcal{A}(s) \subseteq \mathcal{D}(s)$ and, for 'many program fragments', $\mathcal{A}(s) = \mathcal{D}(s)$.

This issue is not peculiar to measurements of object oriented software neither does it only arise because of the reliance upon program slicing. The authors firmly believe that many measurements of software which are interesting and revealing will be inherently non–computable, but that this should be seen as inevitable. Measurements based solely on syntactic properties like the number of lines in the program reveal something about the way a coding problem has been tackled, but reveal nothing about the *problem itself.*

In order for a measurement to reveal deeper information about software, it must be sensitive to *semantic* issues and not merely to syntactic ones. The fact that such semantic issues are, by their very nature, non–computable, should not act as a deterrent, as the measurements taken should only be used in an approximate fashion; as guiding principles rather than as dogmatic rules with rigid thresholds.

This issue of the computability or otherwise of a metric applies to many existing software metrics. For example, consider the oft–quoted question:

"What does McCabe Cyclomatic Complexity Measure?".

One of two answers are usually offered in response to this question:

- "The number of cycle–free paths through which the program may pass",
- "The number of predicates in the program plus one".

Often these two answers are considered to be identical. They are not. The McCabe measure precisely measures the number of predicates in a program plus one. It is an approximate answer to the question "how many cycle–free paths are there through which the program may pass?". Some of the paths which, though syntactically possible (because, for example they form the consequent or alternate branch of a conditional statement) may not be semantically possible (because, for example, they form the consequent branch of a conditional whose predicate is a contradiction). Such paths are often referred to as 'infeasible paths'. That is, paths in the control flow graph of a program which cannot be traversed during any *execution* of the program.

Infeasible paths cannot be detected because the question of whether or not a path is feasible is, in general, undecidable. This may seem to be a pedantic point, but it does have an impact, for example where McCabe measurements are used to predict the likelihood of a fault occurring in software modules with widely differing numbers of infeasible paths. This might occur in a team where one programmer is in the habit of leaving infeasible code in the program together with a statement which prints something like:

> "this statement should not be executed"

.

Such code may be used to good effect by a programmer who uses assertions in programs as part of a testing strategy.

8 Conclusion and Future Work

This paper has outlined an approach to object oriented software measurement based upon the flow of data and control among class data members and functions. In particular it is argued that slices constructed from member functions with respect to the data members they define and reference form a basis from which many useful measurements can be calculated. This idea is illustrated by showing how slicing may be used as the basis for the calculation of finely grained measurements of the publicness of a data member, its significance in the class, and the overall level of cohesiveness which a class exhibits.

It is argued that slicing is a good stating point for software metrics because a slice captures semantic information about a program. Metrics calculated from program slices may therefore reveal interesting semantic insights into the way a program addresses a problem, rather than revealing properties of the type and number of syntactic constructs used in the text of the program.

Much work is required to collect empirical test data to assess the validity and usefulness of these metrics. If such work suggests that these measurements are useful in the comprehension and maintenance of object oriented software it will be straight forward to define other measurements based upon similar principles.

The approach advocated here, in common with some existing approaches to software measurement, relies upon calculations which are *approximations*. This is an important consideration when considering the inferences which can be drawn from the metrics calculated.

This paper does not present definitive results, rather is it put forward as a basis for discussion, and, depending upon the outcome of this discussion, future work.

References

[1] Abbott, D. H., Korson, T. D., and McGragor, J. D. A proposed design complexity measure for object–oriented development. Tech. Rep. TR 94–105, Clemson University, Apr. 1994. Electronically available as `ftp://ftp.cs.clemson.edu/techreports/94-105.ps.Z`.

[2] Agrawal, H., DeMillo, R. A., and Spafford, E. H. Dynamic slicing in the presence of unconstrained pointers. In 4^{th} *ACM Symposium on Testing, Analysis, and Verification (TAV4)* (1991), pp. 60–73. Appears as Purdue University Technical Report SERC-TR-93-P.

[3] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, techniques and tools.* Addison Wesley, 1986.

[4] Banker, R. D., Kauffman, R. J., and Zweig, D. Repository evaluation of software reuse. *IEEE Transactions on Software Engineering 19*, 4 (1993), 379–389.

[5] Beck, J., and Eichmann, D. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)* (Los Alamitos, California, USA, 1993), IEEE Computer Society Press, pp. 509–518.

[6] Bieman, J. Deriving measures of software reuse in object oriented systems. In *Formal Aspects of Software Measurement* (London, 1992), Springer Verlag, pp. 63–83.

[7] Bieman, J. M., and Ott, L. M. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20*, 8 (Aug. 1994), 644–657.

[8] Binder, R. V. Design for testability in object–oriented systems. *Communications of the ACM 37*, 9 (1994), 87–101.

[9] Binkley, D. W., and Gallagher, K. B. Program slicing. In *Advances in Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1996, pp. 1–50.

[10] Canfora, G., Cimitile, A., De Lucia, A., and Lucca, G. A. D. Software salvaging based on conditions. In *International Conference on Software Maintenance* (Los Alamitos, California, USA, Sept. 1994), IEEE Computer Society Press, pp. 424–433.

[11] Cant, S. N., Henderson–Sellers, B., and Jeffery, D. R. Application of cognitive complexity metrics to object–oriented programs. *Journal of Object–Oriented Programming 7*, 4 (July/August 1994), 52–63.

[12] Chidamber, S. R., and Kemerer, C. F. Towards a metrics suite for object oriented design. In *OOPSLA'91, Sigplan notices* (1991), pp. 197–211.

[13] Chidamber, S. R., and Kemerer, C. F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering 20*, 6 (June 1994), 476–493.

[14] Churcher, N. I., and Shepperd, M. J. Comments on 'A metrics suite for object oriented design'. *IEEE Transactions on Software Engineering 21*, 3 (Mar. 1995), 263–265.

[15] Constantine, L. L., and Yourdon, E. *Structured Design*. Prentice Hall, 1979.

[16] Danicic, S., Harman, M., and Sivagurunathan, Y. A parallel algorithm for static program slicing. *Information Processing Letters 56*, 6 (Dec. 1995), 307–313.

[17] De Lucia, A., and Munro, M. Program comprehension in a reuse reengineering environment. In 1^{st} *Durham workshop on program comprehension* (Durham University, UK, July 1995), M. Munro, Ed.

[18] Ernst, M. D. Practical fine–grained static slicing of optimised code. Tech. Rep. MSR-TR-94-14, Microsoft research, Redmond, WA, July 1994.

[19] Fenton, N. E. *Software Metrics: A Rigorous Approach*. Chapman and Hall, 1990.

[20] Fenton, N. E. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering 20*, 3 (1994), 199–206.

[21] Gallagher, K. B., and Lyle, J. R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (Aug. 1991), 751–761.

[22] Georgiadou, E., Milankovic–Atkinson, M., and Sadler, C. RETRO– Reusability, engineering, testing, restructuring and objects. In 4^{th} *Software Quality Conference* (Dundee, 1995), pp. 157–167.

[23] Harman, M., and Danicic, S. Using program slicing to simplify testing. *Software Testing, Verification and Reliability 5*, 3 (Sept. 1995), 143–162.

[24] Harman, M., Danicic, S., Sivagurunathan, B., Jones, B., and Sivagurunathan, Y. Cohesion metrics. In 8^{th} *International Quality Week* (San Francisco, May 1995), pp. Paper 3–T–2, pp 1–14.

[25] Harman, M., Danicic, S., and Sivagurunathan, Y. Program comprehension assisted by slicing and transformation. In 1^{st} *UK workshop on program comprehension* (Durham University, UK, July 1995), M. Munro, Ed.

[26] Henderson-Sellers, B. The economics of reusing library classes. *Journal of Object–Oriented Programming 6*, 4 (1993), 43–50.

[27] Henderson-Sellers, B., and Tegarden, D. Clarification concerning modularization and McCabe's cyclomatic complexity. *Communications of the ACM 37*, 4 (1994), 92–94.

[28] Horwitz, S., Reps, T., and Binkley, D. W. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988), pp. 25–46. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[29] Kalakota, R. The role of complexity in object–oriented systems development. In *Proceedings of the* 26^{th} *Hawaii International Conference on System Sciences* (Los Alamitos, California, USA, Jan. 1993), IEEE Press, pp. 759–768.

[30] Kamkar, M. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

[31] Korson, T., and McGregor, J. Technical criteria for the specification and evaluation of object–oriented libraries. *Software Engineering Journal 7*, 2 (1992), 85–94.

[32] Lakhotia, A. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)* (1993), pp. 34–44.

[33] Liu, L., and Ellis, R. An approach to eliminating COMMON blocks and deriving ADTs from Fortran programs. Technical report, University of Westminster, UK, Feb. 1993.

[34] Longworth, H. D. Slice–based program metrics. Master's thesis, Michigan Technological University, 1985.

[35] Longworth, H. D., Ott, L. M., and Smith, M. R. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of the Computer Software and Applications Conference (COMPSAC'86)* (1986), pp. 383–389.

[36] Lorenz, M., and Kidd, J. *Object–oriented software metrics*. Prentice Hall Object–Oriented Series, 1994.

[37] Lyle, J. R., Wallace, D. R., Graham, J. R., Gallagher, K. B., Poole, J. P., and Binkley, D. W. Unravel: A CASE tool to assist evaluation of high integrity software, Volume 1: Requirements and design. Tech. Rep. NISTIR 5691, US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899., 1995.

[38] Lyle, J. R., and Weiser, M. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications* (Los Alamitos, California, USA, 1987), IEEE Computer Society Press, pp. 877–882.

[39] Milankovic–Atkinson, M., and Georgiadou, E. Metrics for reuse of object–oriented software. In 4^{th} *Software Quality Management Conference* (Cambridge, Apr. 1996).

[40] Ott, L. M. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10^{th} *Annual Software Reliability Symposium* (1992), pp. 16–23.

[41] Ott, L. M., and Bieman, J. M. Effects of software changes on module cohesion. In *IEEE Conference on Software Maintenance* (Nov. 1992), pp. 345–353.

[42] Ott, L. M., and Thuss, J. J. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM Conference on Software Engineering* (May 1989), pp. 198–204.

[43] Ott, L. M., and Thuss, J. J. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium* (Los Alamitos, California, USA, May 1993), IEEE Computer Society Press, pp. 71–81.

[44] Rising, L. S., and Calliss, F. W. An information–hiding metric. *Journal of Systems and Software 26* (1994), 211–220.

[45] Shahmehri, N. *Generalized algorithmic debugging*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1991. Available as Linköping Studies in Science and Technology, Dissertations, Number 260.

[46] Shepperd, M. J. *Foundations of software measurement*. Prentice Hall, 1995.

[47] Simpson, D., Valentine, S. H., Mitchell, R., Liu, L., and Ellis, R. Recoup – Maintaining Fortran. *ACM Fortran forum 12*, 3 (Sept. 1993), 26–32.

[48] Thuss, J. J. An investigation into slice–based cohesion metrics. Master's thesis, Michigan Technological University, 1988.

[49] Tip, F. *Generation of Program Analysis Tools*. PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[50] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (Sept. 1995), 121–189.

[51] Weiser, M. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[52] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

[53] Whitty, R. Object oriented metrics: People and publications. available on the web, URL: `http://www.sbu.ac.uk/~csse/publications/OOMetrics.html`.

[54] Wilde, N., and Huitt, R. Maintenance support for object–oriented programs. *IEEE Transactions on Software Engineering 18*, 12 (1992), 1038–1044.

[55] Yamazaki, S., Ito, M., and Yasahara, R. Object–oriented design of telecommunications software. *IEEE Software* (Jan. 1993), 81–87.

[56] Zuse, H. Support of validation of software measures by measurement theory. In *15th International Conference of Software Engineering* (Baltimore, MD, May 1993).

[57] Zweben, S. H., Edwards, S. E., Weide, B. W., and Hollingsworth, J. E. The effects of layering and encapsulation on software development cost and quality. *IEEE Transactions on Software Engineering 21*, 3 (1995), 200–208.