A Vector-Based Approach to Software Size Measurement and Effort Estimation

T.E. Hastings, *Member*, *IEEE*, and A.S.M. Sajeev, *Member*, *IEEE*

Abstract—Software size is a fundamental product measure that can be used for assessment, prediction, and improvement purposes. However, existing software size measures, such as Function Points, do not address the underlying problem complexity of software systems adequately. This can result in disproportional measures of software size for different types of systems. We propose a Vector Size Measure (VSM) that incorporates both *functionality* and *problem complexity* in a balanced and orthogonal manner. VSM is used as the input to a Vector Prediction Model (VPM) which can be used to estimate development effort early in the software life cycle. We theoretically validate the approach against a formal framework. We also empirically validate the approach with a pilot study. The results indicate that the approach provides a mechanism to measure the size of software systems, classify software systems, and estimate development effort early in the software life cycle to within +/-20 percent across a range of application types.

Index Terms—Algebraic specification, effort estimation, functionality, gradient, magnitude, problem complexity, semantic properties, software metrics, software size, software life cycle, software specification, syntactic properties, validation, vector.

1 Introduction

There are increasingly strong business imperatives for software development organizations to build quality software for competitive global markets. In order to meet these high demands, organizations need to manage the software process [16]. To manage the software process, we need to qualify and quantify software *products*, *processes*, and *resources* [8].

Software size is a fundamental product measure that can be used for assessment, prediction, and improvement purposes [8], [9], [16], [17]. Unfortunately, software size has proved difficult to define and measure—particularly early in the Software Life Cycle (SLC). Traditional software size measures, such as Source Lines of Code, have been the subject of much criticism [1], [6], [7], [8]. As an alternative, Function Points [1] are gaining wide popularity for assessing *application size* [24]. Albrecht's approach [1] has matured to the point of (defacto) standardization [19].

The advantage of Function Point Analysis (FPA) is that it can be applied early in the SLC. However, one of the major criticisms of FPA is its inability to address *complexity* adequately [8], [9], [14], [21], [22], [26], [28], [29], [31]. This can result in a disproportional measurement of size. Therefore, it is difficult to assess the size of different types of systems and to estimate development effort in an objective and comparable manner.

- *T.E. Hastings is with the School of Computer Science and Software Engineering, Monash University, Caulfield East, VIC 3145 Australia. E-mail: hastings@csse.monash.edu.au.*
- *A.S.M. Sajeev is with the Department of Computer Science and Software Engineering, University of Newcastle, Newcastle, NSW 2308 Australia. E-mail: sajeev@cs.newcastle.edu.au.*

Manuscript received 24 June 1998; revised 29 Mar. 1999; accepted 6 Aug. 1999.
Recommended for acceptance by S.L. Pfleeger.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107066.

There have been several Function Point extensions proposed to overcome complexity and other issues [14], [22], [29], [31]. Of these, MKII Function Points [29] and Feature Points [22] are the most tested and accepted alternatives. The MKII approach [29] requires calibration that may prove difficult for specific application types when there is little or no history. The Feature Point technique [22] was popular but has lost favor within the Function Point community. 3D Function Points [31] and Full Function Points [26] have potential but have not been extensively tested.

This paper defines and validates a Vector Size Measure (VSM) and Vector Prediction Model (VPM). The purpose of VSM is to measure the size of software systems and to classify software systems. The purpose of VPM is to estimate development effort early in the SLC.

The paper is organized as follows: In Section 2, we examine software size measurement. In Section 3, we describe how syntactic and semantic properties of software requirements can be specified using algebraic specification. In Sections 4 and 5, we derive VSM and VPM using an algebraic specification language we call ASL. In Section 6, VSM is validated against the theoretical validation framework of Kitchenham et al. [23]. In Section 7, we empirically validate VSM and VPM against eight industrial projects varying from control systems to management information systems (MIS). We also compare our approach to Function Points [19] and MKII Function Points size measures; and to linear regression and COCOMO 2.0 [4] effort prediction models. Finally, we analyze and discuss the results.

2 Software Size Measurement

Before we can measure the size of software, we must first understand what we are measuring. Fenton [8] describes three classifications of software measurement: 1) *Processes* which are software related activities, such as development

and support, 2) *Products* which are deliverables, such as requirements, designs and code, and 3) *Resources* which are assets, such as people, equipment, and tools. According to Fenton [8], all software measures and predictive models fall within these categories.

Measurement is obtained by measuring the *attributes* of *entities* within these categories. Fenton [8] distinguishes between *internal attributes* and *external attributes*. Internal attributes are measured purely in terms of process, product, or resource. External attributes can only be measured in respect to their environment. From a software size measurement perspective, we are concerned with identifying, specifying, and measuring *internal product attributes*.

Fenton [8] suggests that software size, S, is a function, f_S, of *length*, *functionality*, and *problem complexity*, such that:

$$S ::= f_S(l, f, c), \tag{1}$$

where l represents the total number of entities in a system, f represents the number of *functions* a system provides, and c represents the underlying *problem complexity*.

There are many types of complexity, such as *problem complexity*, *solution complexity*, *algorithmic complexity*, *structural complexity*, and *cognitive complexity* [9]. In this paper, when we refer to *complexity*, we mean *problem complexity*. Fenton and Pfleeger [9] informally define problem complexity as the amount of resources required for an optimal solution to a problem. It is difficult to measure a solution early in the SLC. However, we are able to measure software size from software specifications.

3 Software Specification

An important aspect of software engineering is to have a clear and concise specification of "what" a system provides [18]. A software specification is an abstraction of an implemented (or to be implemented) software system [5], [17]. A software requirements specification is an early SLC product from which the size of the system may be measured [8], [9], [24]. For example, the functional size of a software system is commonly measured from software requirements specifications using Function Points [19], [24], [29]. The ability to measure the size of a system from software specifications early in the SLC provides project managers and developers the ability to assess, predict, and improve products, processes, and resources throughout the remainder of the SLC.

To accurately measure the size of a system from software specifications, we must first specify software in a concise, consistent, and complete manner. Software systems are specified using abstract concepts and notations, i.e., the unnecessary detail is omitted at the appropriate point in the SLC [5]. There are many forms of abstraction, such as *data modeling* [6], *algebraic specification* [12], *object modeling* [5], [30], *use cases* [20], [30], *formal specification* [27], and a host of other techniques including *natural language*. We must ensure that the fundamental *entities*, their *attributes*, and *relationships* are captured in a software specification. The challenge is to have an adequate level of abstraction to enable accurate software size measurement at the appropriate point in the SLC.

3.1 Algebraic Specification

We will now describe an algebraic specification approach which will form the basis of the vector size model presented in the next section. A combined or unified approach is required to take the data and the functional aspects of software systems into account. Ideally, we would like these basic elements to be defined in terms of themselves. This would provide a consistent and orthogonal mechanism. One such mechanism is *algebraic specification* [12].

Algebraic specification is based on *Abstract Data Types* (ADT) and provides a formal mathematical description of software without "over specification" [25]. It is important, particularly for software size measurement, that software is not over specified nor under specified—we require a concise, consistent, and complete representation at the right level of abstraction.

Meyer [25] describes ADT specification as "... a class of data structures not by an implementation, but by the list of *services* [*functions*] available on the data structures, and the formal [*semantic*] *properties* of these services." Meyer [25] describes the advantage of this approach as: "...it reintroduces the **functions** of software systems in a balanced way.... Thus, the loop is closed: we have all the elements for a harmonious decomposition technique, guided by the data, but assigning functions their proper place."

An ADT is composed of a set of functions, F, and a set of rules, R, such that:

$$ADT ::= F \bullet R. \tag{2}$$

An ADT has two major parts: 1) *Functions*—a set of *function signatures* that define the *syntactic properties* (interface) of an ADT and 2) *Rules*—a set of *assertions* and *axioms* that specify the *semantic properties* (meaning) of an ADT. Assertions may be specified as *preconditions* or *postconditions*. Preconditions ensure that partial functions are not invoked under conditions that fall outside the domain of a function. Post conditions specify the results (range) of a function. Axioms specify the semantic properties of an ADT in terms of functions and *free-variables* formed into predicates that must hold.

The specification of a system (subsystem or component), *System*, is the set of all ADTs, $\{ADT_1, ..., ADT_n\}$, that are required to define that system (subsystem or component), such that:

$$System ::= \{ADT_1, ..., ADT_n\}. \tag{3}$$

3.2 ASL

We will now briefly describe an Algebraic Specification Language called ASL [15]. ASL is a simple specification language that is based on ADTs. ASL does not have any reserved words and has few symbols. This makes ASL easy to measure; yet, ASL is a richly featured specification language. To illustrate the basic concepts of ASL, we will use a simple example of a bank account. Listing 1 begins with the name of the ADT, *Account*. Comments are enclosed in braces.

```
Account ::=
{Syntactic Section}
open      (Money):          Account;
balance   (Account):        Money;
deposit   (Money, Account): Account;
withdraw  (Money, Account): Account;
{Semantic Section}
open (initialAmount): account' |
   balance (account') =
      initialAmount; {post condition}

deposit (amount, account): account' |
   balance (account') =
      balance (account) add
         amount;   {post condition}

withdraw (amount, account): account' |
  (balance (account) sub amount)  gt
      Zero = true {precondition}
         and
  (balance (account') =
     balance (account) sub amount);  {post}
```

Listing 1: Account Specification

The syntactic section specifies the functions of an ADT. All functions are *public* by default. ASL provides the ability to declare *private* functions similar to UML [30]. Each function is specified by its signature. A signature consists of the function's *identifier* (name), *domain* (input parameters), and *range* (return parameters).

The semantic section specifies the semantic properties of an ADT in a declarative manner [12], [27]. In the Account example, we have specified the semantics as *assertions*. In many cases this is sufficient. However, *axioms* are sometimes required to fully define the meaning of an ADT. An assertion is specified by a *declaration* and a list of *predicates*. An axiom is specified as a list of predicates. Arguments "take-on" the types specified by the relative function signature. The scope of an argument is contained within a specific rule. Note that range parameters are also declared as actual arguments—a concept that is a little different to most programming languages. A function declaration is followed by the "such that" symbol, "|," as commonly used in predicate calculus, which connects a *declaration* with *predicates*.

ASL adopts a functional approach to maintain a mathematical underpinning [11], [27]. We read the deposit assertion as: *deposit* an *amount* of money into an *account* and return the updated *account'*. The *account* before invocation is not the same as the *account'* after the deposit function has been invoked. The resultant *account'* has a new balance. By convention, we use a *prime* to differentiate between domain and range parameters. However, not all functions are transformers, e.g., the *balance* function is a simple inquiry that does not affect the *account* value.

In the above specification, a number of ADTs are referenced, such as Boolean, Money, and Account. Each of these types is specified by an ADT. There are two distinct categories: 1) *BaseTypes*, such as Boolean, Integer, and Set, which are defined as part of an ASL Library, 2) *Application Types*, such as Account, which are specific to the application domain. From an application measurement perspective, we are interested in identifying, specifying, and measuring the attributes of the application types.

4 Vector Size Measure

We will now use ASL as a mechanism to derive a Vector Size Measure (VSM). We will first look at the software size attributes and the relationships between attributes. We will then look at a vector representation, scale types, and units of measure. Finally, a simple example is presented.

4.1 Attributes

Measurement captures information about attributes of entities and the relationships between attributes and entities [8], [9]. The software size measure we propose has two principle attributes: *functionality* and *problem complexity*. Functionality represents the services provided by a software system to its clients. Problem complexity represents the underlying semantics (meaning) of a software system. Intuitively, both users and developers of software systems refer to the functionality a system provides and the underlying complexity of the problems a system solves. Shortly, we will see how these intuitive attributes are represented with numbers.

With respect to (1), Fenton [8] stated: "there appear to be three such [orthogonal and fundamental] attributes of software: *length*, *functionality*, and *complexity* of the underlying problem which the software is solving." In our measure, length is a derived attribute. We show this formally below.

4.2 Representation

From (3), a software system is specified by a set of ADTs. An ADT is composed of its properties. Properties are composed of operators and operands, i.e., the atomic units. We have adapted Halstead's [13] concept of *operators* and *operands* as OPs, where: 1) *Operators* map to function references and rule connectors ("|" and "=") and 2) *Operands* map to generic parameters, parameters, and free variables. Halstead [13] used his approach to measure code in the solution domain. We have mapped his principle to a higher level abstraction of software specification in the problem domain. Refer to Appendix 1.3 for a formal definition of OP.

4.2.1 Functionality

The *functionality* of an ADT is defined as the distinct number of syntactic properties measured by the sum of OPs in the syntactic section of an ADT. The following rules apply:

1. A syntactic property is composed of one or more operators (the function identifier plus parameters that are functions) and zero or more operands (parameters).
2. The addition of one atomic unit, that defines a syntactic property, increases the functionality of an ADT by exactly one OP.

3. The removal of one atomic unit, that defines a syntactic property, reduces the functionality of an ADT by exactly one OP.
4. An ADT with no syntactic properties has zero functionality.
5. An ADT with exactly one void function (that does not take any parameter nor return any value) has unit functionality.

Thus, the functionality of an abstract data type, A, measured in OPs is:

$$f_{\mathrm{A}} = \Sigma \mathrm{OP}_{\mathrm{F}} \qquad [\mathrm{OPs}]. \tag{4}$$

For example, the Account ADT in Listing 1 has four function signatures where *open* and *balance* have three OPs each and *deposit* and *withdraw* have four OPs each. Therefore, the functionality of the Account ADT is:

$$f_{\mathrm{Account}} = 3 + 3 + 4 + 4 = 14 \qquad [\mathrm{OPs}].$$

4.2.2 Problem Complexity

The *problem complexity* of an ADT is defined as the distinct number of nonredundant semantic properties measured as the sum of OPs in the semantic section of the ADT. Semantic properties are specified by rules, i.e., *assertions* and *axioms*. The following rules apply:

1. An assertion is expressed in terms of a *declaration*, a *connector*, and a *predicate list*. An axiom is specified as a predicate list. A predicate may be a simple expression or a compound expression. Expressions are recursively defined in terms of operators and operands.
2. The addition of one atomic unit that defines a semantic property increases the problem complexity of an ADT by exactly one OP.
3. The removal of one atomic unit that defines a semantic property reduces the problem complexity of an ADT by exactly one OP.
4. An ADT with no semantic properties has zero problem complexity.
5. An ADT with exactly one atom, e.g., *true* (), has unit problem complexity.

Thus, the problem complexity of an abstract data type, A, measured in OPs is:

$$c_A = \sum \mathrm{OP}_{\mathrm{C}} \qquad [\mathrm{OPs}]. \tag{5}$$

For example, the Account ADT in Listing 1 has three assertions where *open* has eight OPs, *deposit* has 12 OPs, and *withdraw* has 21 OPs. The problem complexity of the Account ADT is:

$$c_{\mathrm{Account}} = 8 + 12 + 21 = 41 \qquad [\mathrm{OPs}].$$

4.2.3 Length

The *length* of an ADT is defined as the sum of atomic units specified in the syntactic and semantic sections of an ADT. The following rules apply:

1. The addition of one atomic unit increases the length of an ADT by exactly one OP.

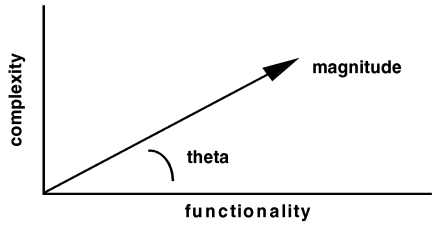

Fig. 1. Software size as a vector.

2. The removal of an atomic unit reduces the length of an ADT by exactly one OP.
3. An ADT with no properties, hence no atoms (OP), has a length of zero.
4. An ADT with exactly one atom has unit length.

Thus, the length of an abstract data type, A, measured in OPs is:

$$\mathrm{l_A} = \Sigma \mathrm{OP_A} = \Sigma \mathrm{OP_F} + \Sigma \mathrm{OP_C} = \mathrm{f_A} + \mathrm{c_A} \qquad [\mathrm{OPs}] \tag{6}$$

Therefore, length can be derived from functionality and problem complexity. For example, the length of the Account ADT is:

$$l_{\mathrm{Account}} = 14 + 41 = 55 \qquad [\mathrm{OPs}].$$

4.3 Size Formulae

The *size* of an abstract data type, S_{A}, is defined by the tuple, $(f_{\mathrm{A}}, c_{\mathrm{A}})$, such that:

$$S_{\mathrm{A}} = (f_{\mathrm{A}}, c_{\mathrm{A}}) \qquad [\mathrm{OPs}]. \tag{7}$$

The size of a software system (subsystem or component), S_{s}, is defined as the sum of the size of all ADTs, S_{N}, that specify the system (subsystem or component), such that:

$$S_{\mathrm{s}} = \Sigma S_{\mathrm{N}} \qquad [\mathrm{OPs}]. \tag{8}$$

An example of system size is provided in Section 4.7.

4.4 Vector Representation

Given that we have two fundamental software size attributes, i.e., *functionality* and *problem complexity*, we can represent software size as a two-dimensional *vector* which has both *magnitude* and *direction*. This representation allows us to understand and transform software size measurements using well-defined mathematical functions. Fig. 1 illustrates the concept.

Using plain vector algebra [2], magnitude, m, is defined as:

$$m = \sqrt{(f^2 + c^2)} \qquad [\mathrm{OPs}]. \tag{9}$$

For example, the magnitude of the Account ADT in Listing 1 is:

$$m_{Account} = \sqrt{(14^2 + 41^2)} = 43.32 \quad [\mathrm{OPs}].$$

Direction is defined as $theta = tan^{-1}(c/f)$. However, we are more interested in the ratio between *problem complexity* and *functionality*, i.e., the gradient, g, to indicate the relative dimensions of a software system, where:

$$g = c/f \qquad (f > 0). \tag{10}$$

TABLE 1
Scale Types

Scale Type	Examples	Allowable Transformations
Nominal	Labeling and classifying entities.	$g()$ is an unordered set
Ordinal	Preference, hardness, air quality, Function Points.	$g()$ is strictly increasing
Interval	Time (calendar), temperature (Fahrenheit, Centigrade).	$g(x) = ax + b \quad a > 0$
Ratio	Length (physical), lines of code.	$g(x) = ax \quad a > 0$
Absolute	Counting entities.	$g(x) = x$

For example, the gradient of the Account ADT in Listing 1 is:

$$g_{Account} = 41/14 = 2.93$$

The *magnitude* is a measure that takes *problem complexity* and *functionality* into account in a balanced and orthogonal manner. The *gradient* is a ratio of *problem complexity* and *functionality* which tells us about the relative dimensions of systems, i.e., the characteristics of software systems. The condition $f > 0$ is implicit because problem complexity implies that functionality must exist, i.e., intuitively, a software system must provide some service (internal or external) otherwise, it is a meaningless system and, therefore, its gradient is undefined $(0/0)$.

4.5 Scale Types

Scale types determine what interpretations we can meaningfully make and the kind of transformations we can perform [8], [9], [32]. Table 1 defines well-known scale types [9].

The nominal scale is the lowest order scale and the absolute scale is the highest order scale. To allow flexibility, software measures should aim for a high order scale such as *ratio*. This will enable us to perform more meaningful transformations. Even simple transformations, such as percentage, require a ratio scale [8], [32]. This will be of particular benefit when deriving higher order measures and prediction models.

The first impression is that our software size measure is based on counting and, therefore, is an *absolute scale* measure. However, we can represent software size as a pair of *scalar* attributes and as a *vector*. This involves transformations between *scalar* and *vector* representations. Vector and scalar representations have a range of permissible transformations that may prove to be useful. Functionality and problem complexity are *discrete ratio* scale measures as they increase or decrease by exactly one. Magnitude is calculated using a vector transformation from functionality and problem complexity and is, therefore, a discrete ratio scale measure that is mapped to *real numbers.* Gradient is an absolute scale measure as it is in the form $g(x) = x$ with no units, i.e., it cannot be logically transformed into another form.

4.6 Units of Measure

A fundamental part of the measurement theory is to define the units of measurement (scales) for ratio scale measures [8]. OPs are a fine grain measure. Properties are a potential coarse grain measure. Our initial experiments [15] clearly showed that the number of semantic properties is not an accurate reflection of problem complexity. For example, a simple rule, such as the Account *open* function, may have one predicate composed of a few operators and operands. However, a more complex rule, such as the Account *withdraw* function, may have two or more predicates composed of many operators and operands. This is a similar problem to what constitutes a line of code [7], [13], [17]. Using atomic building blocks, of operators and operands (OPs), we resolve this problem.

4.7 A Simple Example

Table 2 illustrates the results of measuring an embedded operating system; this is one of the projects we have included in our pilot study in Section 7. The system was specified by converting the original requirements specification to ASL. Note that the total magnitude and the total gradient must be calculated as a vector.

5 Vector Prediction Model

Prediction is a form of calculation [8], [9]. Direct or indirect measures are used as input to a prediction model that is used to estimate the future value of an entity's attribute [8], [9]. The purpose of VPM is to predict the effort required to produce a solution. VPM may be applied early in the SLC by measuring software specifications written in ASL.

There are many effort prediction techniques that could be adopted, including *expert opinion*, *analogy*, *decomposition*, and *models* [9]. VPM is based on a *model*. We will now discuss some common effort prediction models, including: *productivity*, *linear regression*, and *cost* models. We will then present the Vector Prediction Model.

TABLE 2
Example System Size

ADT ID	Attributes		Vector Rep.	
	f [OP]	*c* [OP]	*m* [OP]	*g*
1	8	15	17.00	1.88
2	47	62	77.80	1.32
3	23	72	75.58	3.13
4	14	27	30.41	1.93
5	37	81	89.05	2.19
6	119	812	820.67	6.82
Total	**248**	**1069**	**1097.39**	**4.31**

5.1 Productivity-Based Prediction Models

Simple effort prediction models are based on productivity assessment of past projects [9], [24], [29]. Productivity, P, can be calculated as:

$$P = S/E, \tag{11}$$

where S is the measured size of the software system and E is the measured total actual effort used to produce a solution.

Equation (11) can be rearranged to predict development effort, such that:

$$E' = S/P, \tag{12}$$

where E' is the estimated effort required to produce a solution and P is an assessment of productivity that is based on similar past projects.

The MKII Function Point estimation process is based on (12), where Software Size is measured in MKII Function Points and Productivity is measured as MKII Function Points/Hour [29]. Typical productivity figures for MIS projects are quoted as 0.06 MKII Function Points/Hour [29]. Effort estimation techniques using Albrecht's approach [1], [19] are also based on productivity assessment [24]. This approach assumes a stable and predictable environment [29], i.e., the products, process, and resources are similar. Difficulty arises when we wish to estimate the effort required to produce different products with different processes and resources. For example, if we used the productivity figures from past MIS projects to estimate the effort to produce an embedded real-time system, then we may be grossly underestimating the development effort.

5.2 Linear Regression

Linear regression models are commonly used to determine the correlation between Effort and Size for a range of projects [9], [17]. Linear regression is a "best fit" algorithm based on *least squares* [9], [17]. If a significant strong correlation is determined, then a linear regression model may be used for prediction purposes [17], such that:

$$E' = \beta_0 + \beta_1 S, \tag{13}$$

where E' is the estimated effort, S is the measured software size, and β_0 and β_1 are linear regression coefficients which are determined empirically from past projects. Test criteria for correlation, significance, and error are discussed in the Empirical Validation section of this paper.

5.3 Cost Models

Cost models, such as COCOMO [3], [4], are based on empirical data that reflect the contributing factors [9]. Many models are based on one primary factor and a number of secondary factors called *cost drivers*. For example, the primary factor may be the size of the system measured in Function Points. The primary factor may not correlate strongly to development effort. Cost models attempt to overcome this by identifying secondary factors (cost drivers) that cause the variations between predicted effort and actual effort [9]. For example, application type, team size, team experience, and project duration may be considered secondary factors that influence actual effort. These factors are assigned weights and applied to the effort equation, such that:

$$E' = aS^b\,(F), \tag{14}$$

where E' is the estimated effort, S is the size of the software, and (F) is the *effort adjustment factor* which is calculated as the product of all cost drivers identified for a specific model. The a and b coefficients are determined empirically.

COCOMO 2.0 is based on software size and up to 17 cost drivers with associated weights [4]. Factor analysis of many cost drivers becomes very difficult [9]. Fenton and Pfleeger [9] assert that cost drivers should be based on empirical data and not just expert opinion, i.e., subjectivity should be replaced by objective measurement. Furthermore, cost drivers should be independent of each other [9].

5.4 The Vector Approach

VPM is based on a cost model. The primary and secondary inputs to VPM are the VSM magnitude and gradient measures, respectively. There is no guess work to VPM, such as rating application characteristics. VPM uses a multivariate regression model to determine the relationships between effort, magnitude, and gradient.

5.4.1 Contributing Factors

VPM is based on the premise that a defined and predictable software development process is in place [16]. A defined and predictable development process implies that appropriate resources are defined and deployed. Given that a defined and predictable development process is in place, we propose that the most influential factors in predicting development effort are software size and the type of application. VSM provides us these basic inputs in the form of:

1. *Magnitude*—which is the primary input. The magnitude is a measure of software size that takes functionality and problem complexity into account.
2. *Gradient*—which is a cost driver. The gradient is a measure of the relative dimensions of a system that may be used to classify systems. For example, real-time systems are considered computationally complex, whereas MIS systems are considered not as complex [22], [26], [28], [29], [31]. In Section 8.1, we show that the gradient differentiated between these types of systems.

5.4.2 Formulae

From (14) and our previous discussion on the contributing factors, the estimated effort, E, is calculated as:

$$E' = am^b g^z, \tag{15}$$

where m is the measured magnitude of a software specification and g is its measured gradient; a, b, and z are coefficients. In order to determine the relationship between the estimated effort, magnitude, and gradient, we transform (15) into a multivariate regression equation as follows:

By taking logs on both sides of (15), we get:

$$ln\ E' = ln\ a + b\ ln\ m + z\ ln\ g.$$

From this, we can get the multivariate regression formula:

$$E'' = \beta_o + \beta_1 M + \beta_2 G, \tag{16}$$

where E'' is $ln\ E'$, M is $ln\ m$ and G is $ln\ g$. The coefficients β_0, β_1, and β_2 are determined empirically. The results of regression analysis are given in Section 7.

6 Formal Validation

The purpose of theoretical validation is to confirm that proposed measures do not violate any necessary properties of measurement theory and defining models. The Kitchenham et al. [23] framework is used to theoretically validate VSM. They propose four theoretical criteria that must be satisfied for direct measures (in this case software size):

1. ***For an attribute to be measurable, it must allow different entities to be distinguished from one another***—An ASL specification is composed of a set of abstract data type (ADT) entities. An ADT is defined by syntactic (functions) and semantic (rules) properties. ADTs are distinguishable by definition, i.e., by identification and their properties.
2. ***A valid measure must obey the Representation Condition***—Fenton [8] summarizes the representation condition as: "To measure the attribute we need to have corresponding relations in some number system; then measurement is the assignment of numbers to the entities in such a way that these relations are preserved." VSM has two principle attributes: *functionality* and *problem complexity*. Functionality and problem complexity are intuitive to the users and developers of software systems. We have defined functionality as the count of the number of OPs in the syntactic section of an ADT. The syntactic section specifies each function of an ADT in terms of a function's identifier, domain, and range. Intuitively, as the number of functions increase, the functionality increases. Similarly, if a function has more parameters, it can be considered as providing more functionality than one with less parameters since the former is dealing with more values. As an example, a graphic function `drawLine (endX, endY, lineType)` is providing more functionality than a function `drawLine (endX, endY)` since the former provides the functionality of setting the line type as well. This relation holds in the corresponding number system where we count the number of OPs. Similarly, as the number of entities in the rules specified in the semantic section of an ADT increase, intuitively, the more complex the rule becomes. By counting the OPs in the semantic section as a measure of the underlying problem complexity, we again retain the relation.
3. ***Each unit of an attribute contributing to a valid measure is equivalent***—The atomic unit of our measurement is an OP. An OP could be an operator or an operand, i.e., a function identifier or a data element. There are two issues to be considered here: 1) Is one operator equivalent to another operator (similarly, is one operand equivalent to another operand) and 2) is one operator equivalent to one operand? We do not need to make a distinction between one operator and another. If an operator is more complicated to implement, then it will be reflected in the specification of that operator (in an ADT) with a corresponding size measure for the operator. Since the size of the software is the sum of the size measures of all ADTs, operator size will be reflected in the size measure without considering one operator as different from another. Regarding question 2, we have argued in Section 3, the relation between function and data. An attribute can be represented either as a function or as a data item and we take the view of authors like Meyer [25] that which representation it takes should not matter for the users of the system, i.e., they are interchangeable. For example, from a declarative perspective, `a:X` is equivalent to `a():X`.
4. ***Different entities can have the same attribute value (within limits of measurement error)***—There is no restriction. Different ADTs can have the same size and different properties can have the same size.

7 Empirical Validation

Empirical validation confirms measures are useful and collaborates that measured values are consistent with predicted values [9], [23]. Therefore, *actual* data is required which can then be compared to *estimates*.

7.1 Scope

The vector approach is empirically validated using a pilot study consisting of eight industry projects and resulting products from different organizations, application domains, and application types. Each project was chosen for the pilot study on the following basis:

1. The project was successful, i.e., the resulting software product met defined requirements, was delivered on time and was within budget.
2. The project employed a defined software development process.
3. Software requirements were documented.
4. Actual development effort was recorded.

All projects available to the authors at the time of the pilot study that met the above criteria were included in the pilot study.

7.2 Objectives

The objectives of the empirical validation are to show that:

1. VSM is a reliable measure of software size and can be used to classify software applications.
2. VPM is a reliable predictor of development effort across a range of application types.

7.3 Method

The following empirical validation method was used:

1. Software Requirements formed the baseline for the validation. For each product, the requirements were reengineered into a "common form requirements model" that is based on the Object-Oriented Software Engineering (OOSE) process [20] using the Unified Modeling Language (UML) [30]. A requirements model consists of a) *Use Case Model*—which defines functionality in terms of the interactions between Actors and External Systems with a system, b) *Domain Model*—which defines the structure of a system in terms of key entities and their relationships, and optionally c) *State Transition Diagram* (STD)—which defines dynamic behavior.
2. Product attributes, including *application domain*, *application type*, and *delivery platform*, were gathered from product documentation.
3. Process and resource attributes, including *effort*, *duration*, *team size*, and *implementation languages*, were gathered from project documentation. Effort was normalized into a common project life cycle to ensure consistency across projects. The normalized project life cycle includes all activities performed in: Analysis, Design, Construction, and Test. That is, all effort that was "billed" to a project was counted.
4. Algebraic specifications using ASL were developed from the "common form requirements model." Each Use Case model was mapped to a high-level ADT that specifies the functions provided by a system. Each entity class in the Domain Model was mapped to an ADT. The events, guards, and actions of the STD were mapped to the high-level ADT.
5. A measurement tool was developed and used to automatically measure the VSM attributes from ASL specifications. The tool provides measurements in units of Properties and OPs.
6. A Function Point Count of the "common form requirements model" was performed for each product. The counting technique was based on the mapping from the OOSE requirements model to the Function Point model defined by Fetcke et al. [10]. International Function Point User Group counting practices were observed [19].
7. A MKII Function Point Count [29] of the "common form requirements model" was performed for each product.
8. Linear regression models were used to analyze the correlation, significance, and error of a) Function Points vs. Effort, b) MKII Function Points vs. Effort, and c) magnitude vs. Effort.
9. VPM constants were calculated using multivariate regression.
10. COCOMO 2.0 cost drivers [4] were rated using product and project documentation and confirmed with project team members.
11. Development effort was estimated using: a) FP Best Fit: based on Function Points [19] vs. Effort linear regression model, b) MKII Best Fit: based on MKII Function Points vs. Effort linear regression model, c) COCOMO 2.0: based on the COCOMO 2.0 cost model [4] with Function Points [19] as the primary input, and d) VPM as defined in (15).
12. The results were tested against defined test criteria.

7.4 Test Criteria

To test if an objective has been met, we need to define test criteria. Humphrey [17] asserts that when using a relationship for planning purposes, we require high correlation, high significance, and a low error.

7.4.1 Correlation

Statistical correlation determines the degree to which two variables are related. If two variables are highly correlated and have a cause-effect relationship, then one variable's value may be used to predict the other. Humphrey [17] asserts the following correlation relationships for planning purposes (where R is the correlation coefficient):

1. $0.9 \leq R^2$ is considered a predictive relationship and can be used with high confidence.
2. $0.7 \leq R^2 < 0.9$ is considered a strong relationship and can be used with confidence.
3. $0.5 \leq R^2 < 0.7$ is considered an adequate relationship and should be used with caution.
4. $R^2 < 0.5$ is not reliable for planning purposes.

Our study is based on a post-hoc statistical analysis which can not confirm a causal relationship between variables. However, it is an appropriate method for a preliminary study, particularly, since we are comparing our approach with other standard prediction methods not simply looking for a significant regression line.

7.4.2 Significance

If we determine a strong correlation, we must determine if it is of practical significance [17]. It is possible to have a high correlation that is not significant. Significance is determined by the probability of finding a correlation by chance. The significance level for the pilot study was set at $p < 0.05$, i.e., a probability of less than 5 percent of finding a correlation by chance.

7.4.3 Error Margins

It is important to understand the error margins that are acceptable and not acceptable. Given that the context is to predict development effort early in the SLC, the following error margin, e, may be used for planning purposes:

1. $e \leq \pm$ 20 percent is considered predictive and can be used with confidence early in the life cycle.
2. $\pm$ 20 percent $< e \leq \pm 50$ percent is deemed acceptable but should be used with caution.
3. $e > \pm 50$ percent is not reliable for planning purposes.

The error margin is calculated as: $e = (estimated - actual)/actual$.

7.5 Constraints

The pilot study was constrained to include projects that met defined criteria. Ideally, *processes* and *resources* would be standardized across all projects. That is, the way software

TABLE 3
Product Information

Project ID	Organisation ID	Application Domain	Application Type	Delivery Platform
1	A	Communications	Control System	Embedded System
2	A	Communications	Control System	Embedded System
3	A	System	Operating System	Embedded System
4	B	System	System Management	UNIX
5	B	Banking	Management Information	UNIX, X Windows
6	B	Banking	Management Information	MS Windows
7	C	Accounting	Management Information	MS Windows
8	C	Accounting	Management Information	UNIX

TABLE 4
Product, Process, and Resource Information

ID	Team Size	Effort (Hours)	Duration (Months)	Implementation Languages
1	3	5,040	12	C, Assembler
2	2	2,240	9	Assembler
3	4	4,480	8	C, Assembler
4	5	7,875	12	C++, RDBMS
5	3	2,363	6	C++, RDBMS
6	3	3,938	10	C++, RDBMS
7	2.5	2,100	6	Smalltalk
8	2	1,120	4	RDBMS, 4GL

TABLE 5
Software Size Attributes

ID	FP [FP]	MKII FP [MKII FP]	f [OP]	c [OP]	m [OP]	g
1	110	130	119	744	753	6.25
2	86	94	65	348	354	5.35
3	160	137	248	1069	1097	4.31
4	911	969	562	1987	2065	3.54
5	241	200	213	704	736	3.31
6	550	460	381	1083	1148	2.84
7	216	156	353	857	927	2.43
8	289	258	224	514	561	2.29

engineering is performed and the people doing the work would be uniform. Clearly, this is unrealistic if not impossible and probably not useful for a practical measure.

7.6 Results

Tables 3 and 4 show the product, process, and resource details. The Function Point Count [19], MKII Function Point Count [29], and VSM attributes are presented in Table 5. Table 6 defines the coefficients for each estimation model. Table 7 defines the COCOMO 2.0 effort adjustment factor (F) for each project. Figs. 2, 3, and 4 illustrate the linear regression (best fit) for Function Points vs. Effort, MKII Function Points vs. Effort, and magnitude vs. Effort respectively. Fig. 5 illustrates the relationship between Actuals vs. COCOMO 2.0 estimates. Fig. 6 illustrates the relationship between Actuals vs. VPM Estimates. Table 8 provides the results of the FP Best Fit, MKII Best Fit, COCOMO 2.0, and VPM estimates. Table 9 compares the results with the test criteria.

All prediction models exhibited correlations between actuals and estimates that are statistically significant ($p < 0.05$). The quoted R-squared values are the square of the correlation between actuals and estimates.

7.7 Observations

The results of the empirical validation indicate:

1. The correlation of Function Points to Effort is not reliable ($R^2 = 0.4392$, see Fig. 2).
2. The correlation of MKII Function Points to Effort is adequate and should be used with caution ($R^2 = 0.5302$, see Fig. 3).
3. The correlation of VSM's magnitude to Effort is strong ($R^2 = 0.7292$, see Fig. 4).

TABLE 6
Estimation Model Coefficients

Coefficient	FP Best Fit	MKII Best Fit	Coefficient	COCOMO 2.0	VPM
β_0	1991.7	2026.6	a	10.0	0.1324
β_1	5.1588	5.3841	b	1.0	1.0244
			z		0.8960

4. Using Function Points with a linear regression model results in a high estimated error range of -49 percent to $+211$ percent (see Tables 8 and 9).
5. Using MKII Function Point with a linear regression model results in a high estimated error range of -46 percent to $+205$ percent (see Tables 8 and 9).
6. Using COCOMO 2.0 [4] with Function Points [19] results in an error range of -62 percent to $+34$ percent (see Tables 8 and 9).
7. Using the VPM approach, we can predict development effort to within -17 percent to $+13$ percent. This represents a significant improvement for a range of projects (see Tables 8 and 9). A significance of 1.7E-5 indicates that results are meaningful and not produced by chance.

8 Discussion

We now look at the benefits of the vector approach, including how VSM can be used to classify systems and its limitations.

8.1 Product Classification

We can use VSM attributes to classify software systems. For example, Project 1 produced an embedded control system and Project 8 produced an accounting system. These systems are at opposite ends of the spectrum where the gradient ranges from 6.25 to 2.29. Fig. 7 illustrates how we can compare these systems. The vector representation gives us visual and mathematical clues to what type of system is to be built. Understanding this early in the SLC helps us decide what processes and resources should be adopted.

The ability to classify software systems forms an important part of VPM where magnitude is the primary driver and the gradient is the single cost driver. The ability to objectively classify a system removes subjectivity. It is interesting to note that the correlation between magnitude and gradient is very low ($R^2 = 0.0073$) which means they are independent—an important criteria for stable cost models [9]. It is also interesting to note that the relationship between functionality and problem complexity is strong ($R^2 = 0.7979$). This is expected because functionality implies problem complexity and vice versa.

TABLE 7
COCOMO 2.0 Effort Adjustment Factors

ID	(*F*)
1	1.72
2	1.78
3	2.64
4	1.07
5	1.1
6	0.96
7	1.0
8	0.5

8.2 VPM Approximation

Equation (15) is a general form for VPM where:

$$Effort' = am^b\, g^z.$$

The pilot study was used to determine the a, b, and z coefficients. Fenton and Pfleeger [9] assert that the b coefficient for cost models is typically 1. Our pilot study confirms this with $b = 1.024$ and $z = 0.896$. A VPM approximation can then be defined as:

$$E' = amg \tag{17}$$

Using regression analysis with a Y intercept value of 0, $a = 1.3793$. Table 10 compares the results of VPM and the VPM approximation. The results indicate that an approximation provides predictive results and meets the defined test criteria.

8.3 Benefits

The vector approach has a number of benefits:

1. VSM includes both functionality and problem complexity that is represented as a vector composed of magnitude and gradient.

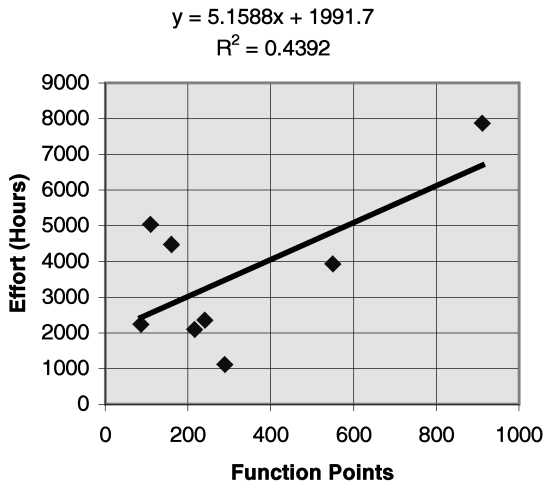

Fig. 2. Function points vs. effort linear regression.

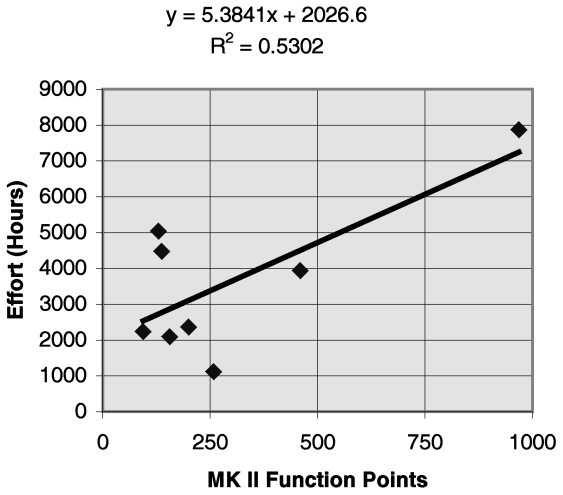

Fig. 3. MKII function points vs. effort linear regression.

2. The magnitude represents the size of a software system that incorporates functionality and problem complexity.
3. The gradient is a relative index that may be used to classify software systems.
4. VSM and VPM can be applied early in the SLC using software specifications written in ASL.
5. The correlation of magnitude vs. Effort is strong and significantly better than Function Points vs. Effort and MKII Function Points vs. Effort.
6. VPM is based on a cost model approach that uses magnitude as the primary input and gradient as a cost driver. Subjective assessment is removed and the number of cost drivers is reduced significantly when compared to cost models, such as COCOMO [3], [4].
7. The pilot study indicates that the approach can be used to estimate development effort early in the SLC to within ± 20 percent across a range of applications. This is a significant improvement over the alternative approaches presented.

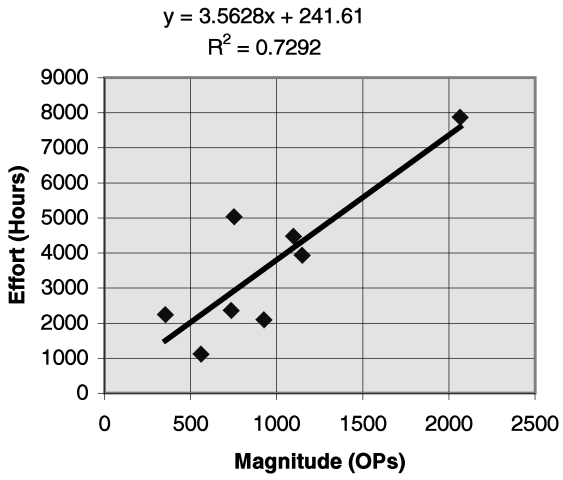

Fig. 4. Magnitude vs. effort linear regression.

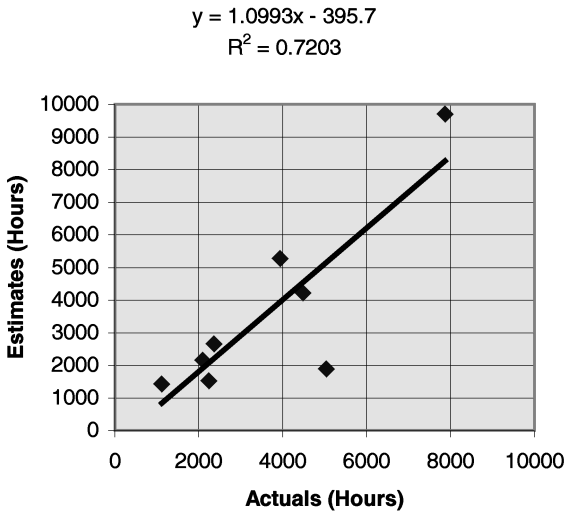

Fig. 5. Actuals vs. COCOMO 2.0 estimates.

8.4 Limitations

The limitations of any measurement technique or model should be understood before it is applied to "real life" situations. The vector approach has a number of limitations:

1. Formal requirements must be specified in ASL. This will require additional work and expertise. The approach taken in the pilot study was to use the OOSE approach [20] to model software requirements and then transform these requirements into ASL specifications. This was quite straight forward and did not require significant effort—approximately three person months to cover all eight products.
2. As a project progresses, the size of the system may grow (or shrink)—particularly in the early phases of the life cycle when product and project scope are not fully realized. Growth is commonly referred to as "functionality creep" [16], [24]. Functionality creep may be due to not capturing all of the requirements and/or new requirements being added. Since the vector approach take semantics into account, we could also expect

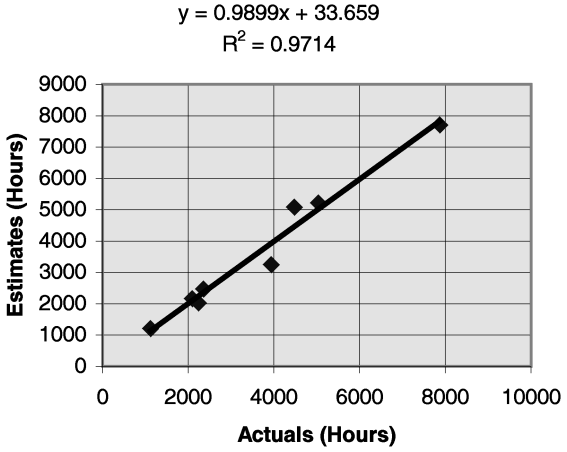

Fig. 6. Actuals vs. VPM estimates.

TABLE 8
Actuals vs. Estimates (Hours)

ID	Actual Effort	FP Best Fit		MKII Best Fit		COCOMO 2.0		VPM	
		Estimate	%Error	Estimate	%Error	Estimate	%Error	Estimate	%Error
1	5040	2559	-49%	2727	-46%	1892	-62%	5220	4%
2	2240	2435	9%	2533	13%	1531	-32%	2034	-9%
3	4480	2817	-37%	2764	-38%	4226	-6%	5080	13%
4	7875	6691	-15%	7244	-8%	9707	23%	7705	-2%
5	2363	3235	37%	3103	31%	2659	13%	2470	5%
6	3938	4829	23%	4503	14%	5280	34%	3252	-17%
7	2100	3106	48%	2867	37%	2160	3%	2161	3%
8	1120	3483	211%	3416	205%	1431	28%	1208	8%

TABLE 9
Results Compared to Test Criteria

Test Criteria	FP Best Fit	MKII Best Fit	COCOMO 2.0	VPM
Correlation R^2	0.4392	0.5302	0.7203	0.9714
Error Margin e	-49% to +211	-46% to +205	-62% to +34%	-17% to +13%
Standard Error	1756	1608	1667	288

"problem complexity creep," e.g., the more we find out about a problem, the more (or less) elaborate its semantics specification could be. Naturally, this will depend on the nature of the problem and the rigor of the development process. However, software size, estimated effort, and actual effort can and should be tracked over the SLC. Software size stability could be used as a project maturity indicator, i.e., as software size stabilizes the more confidence we have.

3. Different people may model and specify systems differently, which may result in different measures of software size and estimated development effort. This has been a concern with other early life cycle measures such as Function Points [8], [9]. The International Function Point Users Group (IFPUG) address counting consistency by publishing a comprehensive counting practices manual [19]. The MKII approach also has well-defined rules [29]. By using a defined process, including review mechanisms, specification, and measurement, inconsistency can be minimized. However, in any measurement, one must acknowledge a level of measurement error.

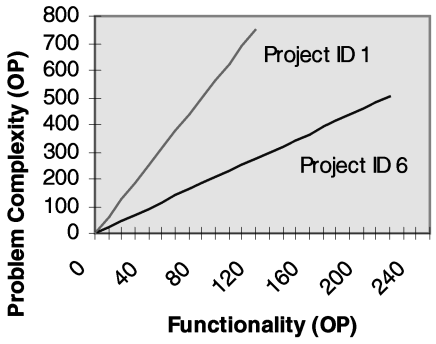

Fig. 7. Comparing software systems using VSM.

4. Naturally, there are projects and products that do not fit within the method and constraints specified for the empirical validation of this paper. Software measurement and prediction of development effort must be underpinned by predictable and stable software processes. The assumption is that a well-defined software development process is in place [16], [17]. Projects and resulting products that do not fall within this constraint are probably difficult to measure and estimate regardless of the technique used.

5. Certain classes of problems may not be specified adequately using ASL. This may result in a disproportional size measurement and an inaccurate effort prediction. ASL specifies the syntactic and semantic properties of software systems. The semantics of a problem, i.e., the meaning, is specified using assertions and axioms. While ASL does not limit the use of recursion, definition of types, and functions, there may be some complex algorithms that could be difficult to specify. As with any modeling technique, including UML [30], there may be classes of software systems when its use is inappropriate. Our approach cannot cover all systems; however, we claim it covers a larger range of systems than other techniques and is based on a mathematical foundation.

TABLE 10
VPM Approximation

Test Criteria	VPM	VPM Approx.
Correlation R^2	0.9714	0.9571
Error Margin e	-17/+13%	-20%/+14%
Standard Error	288	450

9 Summary

The importance of engineering principles, including specification, measurement, and validation, have played a pivotal role in developing the concepts and models presented in this paper. The proposed vector representation provides a balanced view where software size is measured in terms of *magnitude* and *gradient*. The magnitude provides a measure that takes into account *functionality* and *problem complexity* in a balanced and orthogonal manner. The gradient is a ratio of problem complexity and functionality that measures the relative dimensions of systems. This provides a more complete view of software size when compared to approaches that generally consider functionality or length in isolation. One of the distinguishing features of our approach is the ability to compare systems objectively. Theoretical validation confirms that the approach does not violate any necessary properties of measurement theory. Empirical validation confirms that the approach can be used early in the SLC for the prediction of effort required to provide solutions across a range of application types and application domains.

9.1 Future Direction

For the concepts presented in this paper to be of practical use, we need methods and techniques that are useable by practicing software engineers. With the advent and acceptance of the Unified Modeling Language (UML) [30], we can practically apply VSM and VPM principles. For example, requirements (specified in terms of a Use Case Model, Domain Model, and State Transition Model) used in the case studies were modeled using UML [30] within the Jacobson et al. process framework [20]. The main practical problem is ASL which was specifically designed for deriving VSM. A practical replacement is required. In 1997, the Object Management Group (OMG) released the UML Object Constraint Language (OCL). OCL has the potential of replacing ASL in a manner that is integrated directly with an accepted modeling language with CASE tool support. Potentially, this makes measurement and prediction fully integrated with accepted modeling methods thus providing little or no overhead to the software process. When we reach this level of measurement integration with software process, we will be in a strong position to measure software size and predict development effort effectively.

Appendix A
ASL Definition

A.1 Syntactic Conventions

```
<text> Terminal category
"symbol" Symbol
::= Defined by
[...]          Optional group
{...}          Repeating group—zero or many
|              Alternation
{...}+  Repeating group—one or many
```

A.2 Grammar

```
<System> ::= {<ADT>}+
<ADT> ::= <Identification>
  <Syntactic-Section><Semantic-Section>
<Identification> ::=
  <Type-ID>[<Generics>] "::="
<Generics> ::= "[" <Type-ID>{","<Type-ID>} "]"


<Syntactic-Section> ::= {<Signature> ";"}+
<Signature> ::= <Prefix>|<Infix>
<Prefix>::=[<Private>]<Function-ID>
  [<Domain>][":"<Range>]
<Infix> ::= [<Private>]["_"]<Function-ID>
  ["_"][<Domain>] ":"[<Range>]
<Private> ::= "-"
<Domain> ::= "("[<Parameters>] ")"
<Range> ::= ["("][<Parameters>][")"]
<Parameters> ::= <Parameter>{","<Parameter>}
<Parameter> ::= <Type>|<Functor>
<Type> ::= <Type-ID>[<Generics>]
<Functor> ::= "$"


<Semantic-Section> ::= {<Rule> ";"}
<Rule> ::= <Axiom>|<Assertion>
<Axiom> ::=  <Predicate-List>
<Assertion> ::= <Declaration>
  <Such><Predicate-List>
<Predicate-List> ::= <Predicate>
  {<Qualifier><Predicate>}
<Predicate> ::= <Expression>
  [<Equivalent><Expression>]
<Declaration> ::= <Reference>[":"<Results>]
<Reference> ::= <Prefix-Reference>|
  <Infix-Reference>
<Prefix-Reference> ::= <Qualifier><Arguments>
<Infix-Reference> ::= [<Argument>]
  <Qualifier>[<Argument>]
<Qualifier> ::= [<Type-ID> "."]<Function-ID>
<Results> ::= <Arguments>|<Argument>
<Expression> ::= <Reference>|<Free-Variable>
<Arguments> ::= "("<Argument>
  {","<Argument>} ")"
<Argument> ::= <Reference>|<Free-Variable>
<Such> ::= "|"
<Equivalent> ::= "="
<Comment> ::= "{"<Text> "}"
```

A.3 OP Definition

```
<OP> ::= <Operator>|<Operand>
<Operator> ::= <Function-ID>|<Such>|
  <Equivalent>|<Functor>
<Operand> ::= <Type-ID>|<Parameter>|
  <Free-Variable>
```

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees who provided invaluable feedback that led to significant improvement to the paper—especially to the Vector Prediction Model and the empirical analysis presented in this paper.

REFERENCES

[1] A.J. Albrecht, "Measuring Application Development Productivity," *Proc. IBM Applications Development Symp., SHARE-Guide,* pp. 83-92, 1979.
[2] F. Ayres, Jr., *Theory and Problems of Differential and Integral Calculus,* second ed. New York, N.Y.: McGraw-Hill, 1972.
[3] B.W. Boehem, *Software Engineering Economics.* Englewood Cliffs, N.J.: Prentice Hall, 1981.
[4] B.W. Boehem, B. Clar, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0.0," *Ann. Software Eng.,* vol. 1, no. 1, pp. 1-30, 1995.
[5] G. Booch, *Object Oriented Analysis and Design with Applications.* Redwood City, Calif.: Benjamin/Cummings, 1991.
[6] T. DeMarco, *Controlling Software Projects: Management, Measurement and Estimation.* New York, N.Y.: Yourdon Press, 1982.
[7] L.O. Ejiogu, *Software Engineering with Formal Metrics.* Boston, Mass.: QED Technical Publishing Group, 1991.
[8] N.E. Fenton, *Software Metrics: A Rigorous Approach.* London: Chapman and Hall, 1991.
[9] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach,* second ed. London: PWS, 1997.
[10] T. Fetcke, A. Abran, and T. Nguyen, "Mapping the OO-Jacobson Approach into Function Point Analysis," *Proc. TOOLs USA 97,* 1997.
[11] A.J. Field and P.G. Harrison, *Functional Programming.* Wokingham, UK: Addison-Wesley, 1988.
[12] J.V. Guttag and J.J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica 10,* pp. 27-52, 1978.
[13] M.H. Halstead, *Elements of Software Science.* New York, N.Y.: Elsevier, 1977.
[14] T.E. Hastings, "Adapting Function Points to Contemporary Software Systems: A Review of Proposals," *Proc. Second Australian Conf. Software Metrics (ACOSM '95),* pp. 103-114, 1995.
[15] T.E. Hastings and A.S.M. Sajeev, "A Vector Based Software Size Measure," *Proc. Australian Software Eng. Conf. (ASWEC '97),* pp. 7-15, 1997.
[16] W.S. Humphrey, *Managing the Software Process.* Reading, Mass.: Addison-Wesley, 1989.
[17] W.S. Humphrey, *A Discipline for Software Engineering.* Reading, Mass.: Addison-Wesley, 1995.
[18] *IEEE Guide to Software Requirements Specifications.* New York, N.Y.: IEEE Press, 1984.
[19] *Counting Practices Manual, Release 4.0.* Westerville, Ohio: International Function Point Users Group, 1994.
[20] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach.* Reading, Mass.: Addison-Wesley, 1992.
[21] R. Jeffery and G. Low, "Function Points and Their Use," *Australian Computer J.,* vol. 29, no. 4, pp. 148-156, 1997.
[22] C. Jones, *Applied Software Measurement.* New York, N.Y.: McGraw-Hill, 1991.
[23] B. Kitchenham, S.L. Pfleeger, and N. Fenton, "Towards a Framework for Software Measurement Validation," *IEEE Trans. Software Eng.,* vol. 21, no. 12, pp. 929-944, 1995.
[24] D.H. Longstreet, "How Are Function Points Useful?" *Am. Programmer,* pp. 25-32, Dec. 1995.
[25] B. Meyer, *Object-Oriented Software Construction.* Hemel, Hempstead: Prentice Hall, 1988.
[26] *Full Function Points: Counting Practices Manual, Technical Report 1997-04.* Montreal, Canada: Software Eng. Management Research Laboratory and Software Eng. Laboratory in Applied Metrics, Univ. of Quebec Montreal, 1997.
[27] J.M. Spivey, *The Z Notation: A Reference Manual.* Hemel, Hempstead: Prentice Hall, 1989.
[28] C.R. Symons, "Function Point Analysis: Difficulties and Improvements," *IEEE Trans. Software Eng.,* vol. 14, no. 1, pp. 2-11, 1988.
[29] C.R. Symons, *Software Sizing and Estimating: Mk II FPA.* Chichester, England: John Wiley, 1991.
[30] *The Unified Modeling Language Definition, Version 1.1.* Rational Software Corporation, 1998, http://www.rational.com.
[31] S.A. Whitmire, "3D Function Points: Scientific and Real-Time Extensions to Function Points," *Proc. Pacific Northwest Software Quality Conf.,* 1992.
[32] H. Zuse, *A Framework for Software Measurement.* Germany, Berlin: DeGruyther, 1997.

T.E. Hastings is a PhD candidate in the School of Computer Science and Software Engineering of Monash University, Australia. He earned a masters degree in computing from Monash University. Mr. Hastings has published several articles on software measurement and is a practicing software metrics consultant. Mr. Hastings is a member of the IEEE, the Australian Software Metrics Association (ASMA), and the Australian Computer Society (ACS).

A.S.M. Sajeev received the PhD degree in computer science from Monash University. He is an associate professor in the Department of Computer Science and Software Engineering of the University of Newcastle, Australia. He has more than 35 articles in refereed journals and conferences in the areas of software engineering and parallel and distributed computing. Professor Sajeev is a member of the IEEE and the ACM and a fellow of the Institution of Engineers, Australia.