Software metrics: fault content estimation and software process control

To overcome the prevailing problems of error management, missed deadlines and overspent budgets, commercial software developers will have to embrace the concept of 'measuring' projects. **Bo Lennselius, Claes Wohlin and Ctirad Vrana*** review the developing field of software metrics and present some preliminary findings

The paper shows how software metrics can be used to plan and control software projects. Software metrics will be essential if the software industry is to continue growing and developing complex systems. The only way to increase knowledge of the software development and maintenance processes and the final product is to measure them and use the measurements in models for estimating their future behaviour. The emphasis of this paper is on complexity metrics and reliability models, and especially on their use for fault content estimation and control of the development and maintenance processes. Empirical results and guidelines of how to use complexity metrics and reliability models are presented.

software development **complexity metrics** **reliability models**

Despite a great deal of effort, the software industry still suffers from problems such as error-prone products, projects missing their target dates and projects a long way over budget. It is therefore necessary to make the software development processes — requirement specification, design, fault correction during development, corrective maintenance, enhancement etc. — more effective throughout the software life cycle. The aim must be to reduce the total cost of developing software products.

To decrease the life-cycle cost it is necessary to consider the residual fault content of the product and its impact on reliability and cost. It is therefore necessary to estimate the fault content, and especially the number of faults that affect the operational behaviour of the product.

Department of Communication Systems, Lund Institute of Technology, Box 118, S-221 00 Lund, Sweden
*Telelogic AB, Baltzarsgatan 22, S-211 36 Malmö, Sweden
This project is supported by Telelogic AB and the Swedish Telecommunication Administration

The problems of the software industry originate from a lack of appropriate techniques and methods as well as poor management, both at the highest levels and at the project level, with regard to the complex systems to be developed. A lot of money and effort has been spent trying to solve these problems. Traditionally, technical issues such as better languages and tools have been emphasized; these are necessary for the improvement of software development and maintenance. It is also most important, however, to develop and adopt techniques to improve project management and understanding of the different software development processes; in this way we can

- improve planning and control of the project (e.g. obtain information about expected results)
- compare different methods objectively and try to improve them.

DeMarco[1] states 'You can't control, what you can't measure.' The authors totally agree with this. Metrics and measurement are essential to achieve meaningful control over the software process throughout the life cycle. Consequently, we need a set of different metrics giving information about the product and how the project is proceeding. The product is assumed to consist of the documentation produced throughout the life cycle, and not just the code. In this paper we use the definition of software metrics given by DeMarco[1]:

- 'A metric is a measurable indication of some quantitative aspect of a system.'

In this context the word 'system' refers to either the product developed or the process of development throughout the life cycle. This brings us to the following definitions used by Conte[2].

- 'Process metrics quantify attributes of the development process and of the development environment' (e.g. the cost of development).
- 'Product metrics are measures of the software product' (e.g. the number of pages of design documents).

Consequently, metrics can be related to product 'qualities', to the handling of the product throughout the software life cycle and indirectly to methods, tools and actors (i.e. organizations and humans that participate in the development and maintenance of the product).

In our work we emphasize the use of metrics to improve planning and control throughout the software life cycle. This has led to the concept of 'management by objectives' (Figure 1). This concept requires the following three conditions to be met.

- There must be measures on qualities.
- The relationships between these measures and other quantities (e.g. resources, planning and economy) must be known.
- To use the results for planning and control there must be some feedback from developers and users.

The first two of these conditions call for fully defined methods during specification, design, coding, testing etc.

Metrics are the theoretical bases for software engineering. The list of areas and occasions where different metrics are needed is very long. We have already mentioned planning and control of software projects. Some other typical examples are contracts or responsibilities, requirement specifications, guarantees, cost estimation, the choice of methods, the choice of basic techniques, quality assurance, tests etc. Metrics can also be used to evaluate projects and products, answering such questions as 'Why did our project cost three times more than predicted?' and 'Why does our product contain ten times more faults than predicted?'

Thus metrics have an important purpose in building up a 'corporate memory' which can be used, for example, in the planning and control of future projects. By building up a 'corporate memory' and by using models to explain the behaviour of the software development process we will increase our knowledge about the process, learn something about the inevitable consequences of our applied methods (see for example the section on the quality constraint model, below) and be able to improve the process. The use of software metrics to improve the software development process is further discussed by Basili[3].

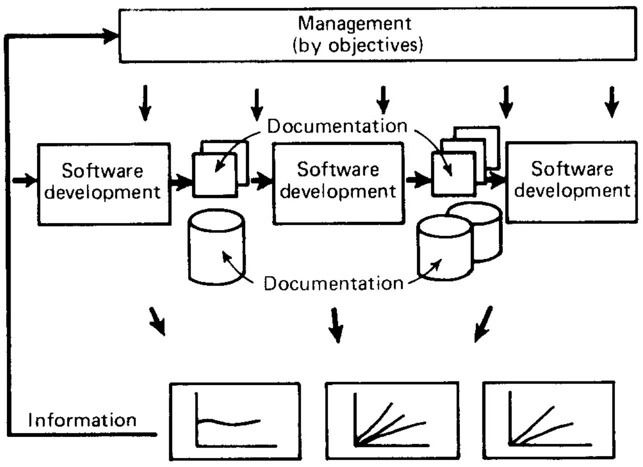


Figure 1. Management by objectives

Today, software metrics are prone to certain limitations. First, it should be made clear that the use of metrics can only be useful if one is comparing like with like. For example, the most familiar software measure is the number of lines of code; there is no general agreement about what constitutes a line of code, however.

Second, the same confusion abounds in the definitions of different qualities[4]. Everyone agrees, for example, that 'user friendliness' is a very important quality attribute of a software product, but we have no common definition of what user friendliness is. The conclusion must be that we need to have generally agreed definitions on metrics and qualities.

Third, we have to find a structure among the different qualities and see how this structure harmonizes with other models and ways of looking at things connected to the 'software product' concept. In the next section such a structure is presented.

Finally, we have to remember that the metrics and models cannot replace the decision-making process of the managers. As Boehm[5] states, 'The models are just there to help, not to make your management decisions for you.'

CRITERION STRUCTURE

In accordance with the system and life-cycle models for a system[6], a system is defined by its documentation (by definition the system does not exist if there is no adequate documentation). All handling of a system is defined as a transformation of these documents. From the abstract system (the source system) are derived (copied, constructed or manufactured) the operative systems.

The various economical and quality aspects can (and should) be structured in a similar manner. Even if the structure is not completely orthogonal it makes the derivation and study of metrics and their applications considerably easier. A structure of qualities is presented in Figure 2. The structure is compatible with the product concept applied to software products within the Swedish Telecommunication Administration and other companies, and imitates the way in which mechanical products are traditionally handled. Vrana[7] shows how qualities impact

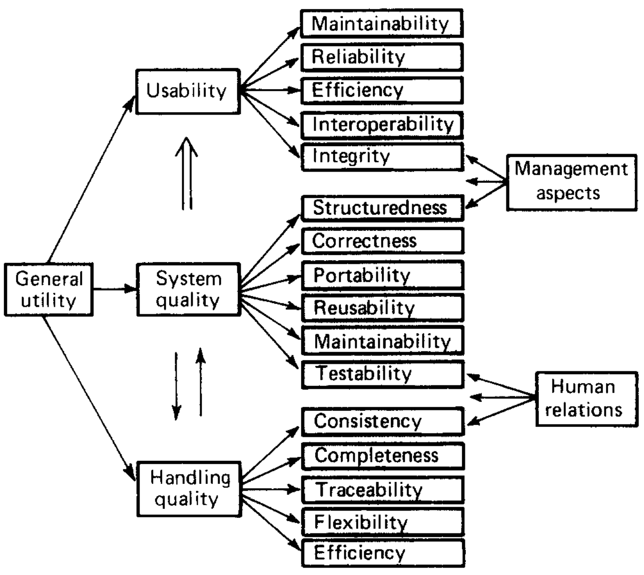


Figure 2. Criterion structure

on a system based on an 'hierarchical modular' structure. In systems built in this way almost all qualities are improved.

As pointed out above, there are many qualities that can be of interest. Which are to be considered important depends on the application and the constraints on that specific application. This area has not been fully investigated. However, some metrics and qualities have, due to their importance, been used for some time, and have also been studied in greater theoretical depth than others. To this category belongs reliability (and models of it) and complexity metrics (and different qualities derived from it). This paper will concentrate on complexity metrics and reliability models, especially for fault content estimations.

The difference between reliability and fault content should be observed. A product may have a number of faults, but if they are located in paths that are seldom executed, the product is considered to be reliable. It should be noted that the faults considered by reliability models are those that contribute to the reliability under the present conditions, and not necessarily to the total fault content.

During development it is possible to estimate the fault content from complexity metrics and reliability models. The latter estimation is possible since, in many of the models, one of the parameters is the number of faults. The models applicable during development are highly dependent on the environment and techniques used, since the behaviour of the product is dependent on, for example, the testing strategies applied. This means that the estimate during development is only an indication of possible problems during operation; this limitation is due to the problem of imitating the operation phase. During operation, however, it is possible to use reliability models to estimate the reliability and the fault content. The fault content that is estimated at this time in the life cycle is the one that the user faces, which means that these are the faults that really contribute to the total life-cycle cost.

COMPLEXITY METRICS

Complexity metrics can be used to measure how complex (or difficult) a software product is. Historically, complexity metrics have been based on source code, but if complexity metrics are to be used as a tool for management, it is necessary to measure before the coding phase. This can be done if the product is documented with a well defined and standardized description technique during the different phases of the software life cycle. An example of a well defined description language is SDL, the specification and description language standardized by CCITT[8].

Description complexity (C_d) is a measure of how complex a description document is; this is dependent on the software structure. Different parts of the software structure affect the description complexity differently. Consequently, the description complexity can be divided into complexity originating from size (C_s), complexity originating from control structure (C_{cs}), interdependencies of different descriptions (modules of a software system) (C_{co}) and so on, such that

$$C_d = f(C_s, C_{cs}, C_{co}, \ldots) \qquad (1)$$

There is no metric which covers all these aspects of description complexity for a software product today. Because of this it is very important to use different metrics to measure different attributes. With other factors (see below), these attributes affect the human handling of the descriptions (the product) throughout the software life cycle. Assuming that the influence from other factors is equal for all modules within the same product, the following relations are obtained between the complexity and the number of faults in the product (FAULT) on the one hand and the complexity and the personnel effort (EFF) needed for the development of the product on the other hand.

$$\text{FAULT} \approx k_1 C_s^{z_1} + k_2 C_{cs}^{z_2} + k_3 C_{co}^{z_3} + \ldots \qquad (2)$$

$$\text{EFF} \approx k_4 C_s^{z_4} + k_5 C_{cs}^{z_5} + k_6 C_{co}^{z_6} + \ldots \qquad (3)$$

where k_i and z_i are constants based on description language, programming language, methodology and tools (see below). Complexity measurements can for example be used in the following applications

- estimation of the initial number of faults (before the test) in the product
- identification of error-prone modules (see below)
- estimation of personnel effort for the following software development processes (this is not discussed further here, but developing a complexity model for effort estimation is very similar to developing a model for estimation of the number of faults)
- identification of methods and tools which need to be improved
- guidelines for the structuring of a system.

Empirical study

In this section we summarize an empirical study performed by Lennselius[9] of which the primary goal was to find out if metrics derived from SDL descriptions can be used as a tool for planning and controlling software projects. The relationships between the metrics and the number of faults and between the metrics and personnel effort during the coding phase were analysed in the study. In the following discussion we consider only the results obtained for metrics *versus* number of faults.

The results are based mainly on a study of 15 software modules belonging to a telecommunications switching system. The size of the modules is between one kiloline and seven kilolines of code. The system is described with SDL-like graphs. For the project (referred to as project A throughout this paper), failure data from the test phase were collected for each module.

In the study the results of project A were compared with the results of prior studies by Vrana[10] and by Wohlin[11]. In these studies the relation between the number of faults and the metrics of the SDL-like descriptions in the Axe switching system were investigated for 20 and 28 modules respectively (in the following text these projects are called project B and project C respectively).

Some candidate SDL-based metrics for the investigation are presented below.

C(G). This metric is a modification of McCabe's cyclomatic complexity[12]. C(G) is a measure of the control structure of an SDL graph and is proposed by Lennselius[13]. C(G) is calculated as the number of branches plus the number of input symbols.

NOS. As a measure of the size the number of SDL symbols is counted for each module.

MNOS, MC(G). In the systems under investigation a module consists of several 'functions' (analogous with blocks and processes in SDL descriptions). If two or more functions are regarded as 'similar' (see definition in Lennselius[9]) the metrics NOS and C(G) are modified according to the number of 'similar functions'. Our assumption is that the programmer will become more and more familiar with these 'similar functions' and that we have to reduce their effect on complexity (this modification was only made in project A).

INP. As a measure of the dependencies between the modules of a system we count (for each module) the number of unique signals which are sent to the module from other modules.

For project A the SDL-based metrics were compared with the following code-based metrics: the number of lines of source code (abbreviated to LOC), including comments and the declare sector; the number of executable lines of code (EXE); and Halstead's program volume (V)[14]. For projects B and C the SDL-based measures were only compared with the number of lines of code because the measurements for these projects were done manually.

The above-mentioned metrics are primarily measures of the software structure and are not 'true' measures of the different parts of the description complexity. However, as stated above, the software structure influences the complexity and the above-mentioned metrics are considered below as simple measures of the different parts of the description complexity.

One of the main applications of complexity metrics is to identify error-prone modules. A simple way to define such modules is by using the standard deviation as a rule. An error-prone module is defined as an 'error outlier' if it lies at least one standard deviation above the mean of the error distribution of the project. For any measure, those modules which lie more than one standard deviation above the mean are referred to as 'metric outliers'. (This technique is similar to the validation technique used by Kafura[15].) We may now ask the following two questions of interest. Are metric outliers good indicators of error outliers? Are metrics derived from SDL descriptions better, worse or as good as LOC in pointing out the error outliers?

From projects A, B and C we found 11 modules out of 63 which were error outliers (within the respective project). In Table 1, each column represents an error outlier and each row corresponds to a complexity metric. An 'X' appears in a table entry if the error outlier denoted by the column in which the entry appears is also an outlier of the metric corresponding to the row in which the entry appears. The column labelled 'nonoutliers' shows how many modules were outliers of the complexity metric but were not error outliers. From Table 1 it is not a simple matter to choose the 'best metric', due to their similar results and to the small number of examined modules, but we can state that the metrics derived from SDL descriptions are as good indicators of error outliers as is the frequently used measure LOC.

Table 1. Metric outliers *versus* error outliers for projects A, B and C

Metric	Error outliers											Total	Non-
	1	2	3	4	5	6	7	8	9	10	11	outliers	outliers
LOC	X		X		X	X	X		X			6	3
NOS	X		X	X	X	X	X		X			7	3
C(G)	X		X	X	X	X	X		X			7	4
INP	X	X	X	X	X	X	X		X	X	X	10	2

By using regression analysis the dependencies (correlation) between the number of faults and each complexity metric were studied. The result was that the SDL-based measures had a higher or at least the same correlation with the number of faults as the code-based metrics. (The correlation matrix for project A is presented in Table 2.) This indicates that the SDL-based metrics can be used early in the software life cycle to estimate the number of faults.

For project A the best prediction is obtained by assuming a nonlinear relation between the complexity metrics and the number of faults (see Figure 3 and Table 3). Projects B and C also indicated an unlinear relation[10,11]. For none of the projects has it been possible to statistically determine that the relation is nonlinear or linear, due to the small number of modules in each project.

This study shows a possible way of using complexity measurements throughout the software life cycle. Before the coding phase we can use complexity metrics based on a well defined description language (SDL, for example) to estimate the number of faults and to make an early identification of error-prone modules. This may be reflected in management policies, quality assurance activities and testing effort.

The indication of nonlinearity between the number of faults and the complexity metrics points out the necessity of pinpointing the error-prone modules early in the life cycle. By doing this we can, for example, decide if the system should be redesigned[9]. Modules with extremely high complexity values will significantly increase the cost of a software product[10]. After the coding phase we can use code-based complexity metrics to make new estimations of the fault content. In particular we will study modules whose code-based estimates differ from the earlier estimates. Such a difference makes possible the identification of anomalous modules.

Development and use of complexity models

Many empirical studies (e.g. Yu[16]) have been performed in order to study the impact of complexity (measured from source code) on software development processes. No complexity metric has been shown to be superior generally (i.e. independent of development environment and product category). It is therefore very important to state that a result obtained in one environment cannot be transformed directly to another environment. The price paid for violating this rule is misleading or erroneus estimations. Consequently, we have to make complexity studies in our own development environment. In the text that follows we will give a brief description of the steps necessary in such a study and some aspects of the use of a complexity model.

First, we have to choose a set of candidate metrics for the study. The criterion must be to choose metrics which measure different attributes — complexities originating from size, flow of control, data flow etc. Second, we have to collect data (number of faults, effort etc.) from the

Table 2. Correlation matrix for project A

Metric	INP	NOS	C(G)	MNOS	MC(G)	V	EXE	LOC
Faults	0.95	0.88	0.90	0.93	0.92	0.94	0.91	0.87
LOC	0.87	0.93	0.94	0.93	0.93	0.96	0.97	1.00
EXE	0.91	0.90	0.91	0.94	0.93	0.99	1.00	—
V	0.94	0.91	0.92	0.94	0.94	1.00	—	
MC(G)	0.93	0.93	0.96	0.99	1.00	—		
MNOS	0.93	0.94	0.95	1.00	—			
C(G)	0.91	0.99	1.00	—				
NOS	0.90	1.00	—					
INP	1.00	—						

Table 3. Analysis of the nonlinear relation between the number of faults and the complexity measures for project A

z	V^z		LOC^z		INP^z		$MNOS^z$		$MC(G)^z$	
	r^2	MRE	r^2	MRE	r^2	MRE	r^2	MRE	r^2	MRE
1.0	0.87	0.31	0.75	0.33	0.89	0.30	0.85	0.57	0.84	0.40
1.5	0.91	0.26	0.77	0.28	0.91	0.29	0.89	0.29	0.87	0.27
2.0	0.91	0.26	0.76	0.27	0.90	0.29	0.89	0.26	0.85	0.28

r^2 is the coefficient of determination
MRE is the mean-magnitude relative error between real and estimated values

projects we want to analyse. Data collection has to be done very carefully and is very time consuming without automated tools. Third, we have to identify the metrics that correlate with the fault content of our product. For example, we may find that McCabe's cyclomatic complexity correlates well with the number of faults and that we do not get significantly improved estimates if we consider this metric together with metrics which measure other complexity factors; in this case we choose the cyclomatic complexity metric for our model. Finally, we have to determine the relation between the chosen complexity metric(s) and the number of faults (or amount of effort), i.e. determine the values of the constants k_i and z_i. We now obtain a complexity model.

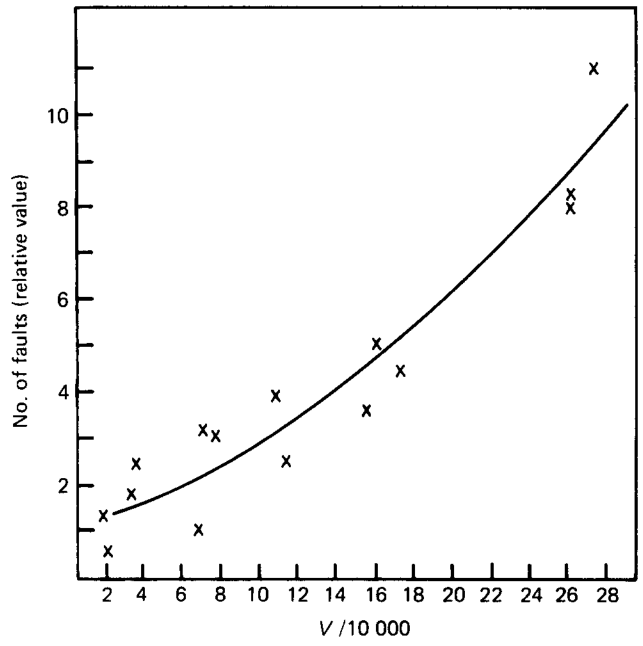


Figure 3. Estimated number of faults $\hat{y} = m + aV^{1.5}$ (solid line) and actual number of faults (marked by crosses); V is program volume[14]

When a complexity model has been developed, it is used and evaluated on new projects (products). This leads us to question whether it is enough for a new project to measure simply the chosen metric (or metrics) of our model; the authors believe not. We still have to measure a set of metrics which consider different aspects of the complexity. The reason for this is as follows.

- We want to evaluate and improve our model continually. By using several metrics we have the possibility of identifying changes in our development environment and altering the complexity model according to these changes.
- By measuring several complexity factors it is possible to identify modules in the analysed product which differ from the 'normal' structure of a module (a very small module with a very complicated flow of control, for example). In other words we can identify anomalous modules. This information is needed in the estimation process, to identify irregularities in our estimations, for example. The information is also needed in the evaluation of the project.

As mentioned above, there are other factors[5, 17] besides those already discussed that affect the personnel effort needed as well as the correctness of the product. Some of the factors are

- product category (administrative systems, operating systems, telecommunication switching systems etc.) and large differences in product size
- performance and memory constraints
- the skill of the project members and their experience of the application area

- time schedule of the project
- amount of reused design documentation and source code
- changes in the specification or the hardware during development
- use of configuration control systems and quality assurance systems or methods.

When we use complexity models we have to monitor (or measure) these factors in order to detect large differences between the project we want to analyse and the projects from which we have derived the complexity model. We need to identify those factors which significantly change the validity of the complexity model. By doing this we can recalibrate the model (the constants) according to the changes between the project which we analyse and the 'typical' project of our complexity model. The issue of how to recalibrate complexity models according to changes in the development environment will be further studied by the authors.

The method outlined above may seem very ambitious, but if we want to stay in control we need to understand the different software development processes and their impact on the qualities of the product and other quantities (e.g. personnel effort needed). As well as the development and use of a complexity model, the method outlined will build up a 'corporate memory' (or part thereof). If we do not monitor and measure we have little chance of increasing the efficiency of the software development processes in a cost-effective way.

RELIABILITY

One of the most important aspects of product quality is the reliability of the software. Software reliability can be defined as the probability that the software does what it is supposed to do throughout a prespecified time. Software reliability is dependent on the number of faults that are introduced when developing the program, since software does not wear out in the way that hardware does. Software is often modified, improved and added to throughout its lifetime, however. The relationship between reliability and the number of faults is affected by the use of the product and the location, size and type of faults are located. As discussed above, software reliability models can be used to estimate the number of faults, but we have to be aware of what fault content they estimate during different phases in the software life cycle.

To obtain a good estimate of the reliability, factors such as the number of faults remaining and the time between failures must be determined. By this means we should be able to predict how failures will occur in the future. An early estimate of the number of faults can be obtained through complexity metrics (see above). This estimate can be used as input to software reliability models, to obtain better estimates of the reliability factors mentioned above as the project proceeds. The results from the reliability models are improved through collection of failure data. This makes it possible to refine the estimates of the model parameters. By predicting failure occurrences it is possible, for example, to predict a suitable time to release the software product and to determine the allocation of resources, so that a product which meets the quality constraints can be delivered on the target date.

Need to compare software reliability models

A realistic prediction of software failure occurrences is vital if we wish to be able to draw any conclusions from it. It is therefore necessary that software reliability models well suited to the situation (i.e. the environment and the application) are used. A software reliability model is used to predict the behaviour of the product but, especially during development, its behaviour is dependent on the environment and techniques used, e.g. testing strategies. This means that the models have to be reasonable with regard to development environment, tools, application, test environment etc. A thorough investigation, classification and comparison of existing models as well as a study of our own environment is needed to identify possible models for our products.

A number of software reliability models exist; some examples are presented in Jelinski[18], Goel[19], Schneidewind[20], Littlewood[21] and Musa[22]. But do we really need this diversity of models?

To answer this question we have to compare models and study how they relate to each other. First of all we have to study the models and select those that are best suited for our environment, techniques and applications according to the assumptions made by the models. Having done this we are probably left with either a number of models or none at all. If we are left with none we have to take an approach similar to the one described below (under 'Environment-adapted models') and presented in Wohlin[23], i.e. to develop a model which is tailored to the environment and the techniques used. Let us suppose that we still have at least two possible models.

An easy (but time-consuming and expensive) method is to use multimodelling, i.e. to use all models and then choose the best one at the end. The criterion for choosing a model is hard to decide. Should we take the worst case, or the model which makes the management happy, or what? Even if a couple of models give the same result, can we be sure that it is the right result? There is a possibility that the result is the same because the models are similar to each other, and this result might not be accurate. The authors suggest another approach, which we call the pre-evaluation approach. We wish to stress the need to compare and evaluate the software reliability models before we use them in a given environment and organization[24, 25]. Our suggestion is that, instead of spending money buying expensive program packages for various models and then running them, effort should be put into investigation and classification. Having understood the models, tools can be selected for some of these models which are best suited to the individual application. Some classifications already exist, but they are mostly concerned with classifying the models according to the approach taken in developing them. This information is of little interest to the user of the models. The user is, in most cases, not especially interested in the mathematical background but rather in the usefulness of the results.

We have to compare the information we get from the models. If one model gives us all the information we need, why use the others? A thorough study of the available models, the information obtained from them and its accuracy has to be conducted. We have to catalogue the models and find out where they overlap. Once this is done we can use the results as guidelines on which model(s) to choose depending on the environment, applications and information needed. The model has first of all to be realistic with respect to the product's behaviour, which is highly influenced by the environment both during development and operation. The behaviour of the product is influenced by, for example, the testing strategies during development and the use of the product during operation. When the model is considered to be realistic and applicable, then the next step is to consider another important aspect when comparing models — the accuracy of estimates obtained from the models[26].

It is no use applying a more sophisticated model than is necessary. If we need a lot of information, then we can use more than one model, but we have to be sure that the models complement each other, otherwise effort will simply be wasted. It is the authors' conviction that an extensive investigation of all available software reliability models must be carried out.

Classification of software reliability models based on the failure process studied

The best classification of software reliability models from the user's point of view would be based on the application area and information obtained from the models. The classification based on the failure process studied, however, is important because the models must be compared based on the failure process studied. According to Goel[27], the software reliability models can be placed in four classes. Goel defines the classes as follows.

'Times between failures' models. The general approach for this class of models is to assume that the time between failure number $(i - 1)$ and failure number (i) follow a probability distribution, whose parameters depend on the number of faults remaining. One of the first and most commonly used models is the Jelinski-Moranda de-eutrophication model[18].

'Failure count' models. This class of models assumes that the number of detected failures in an interval follows a stochastic process with a time-dependent discrete or continuous failure rate. A well known model from this class is the Goel-Okumoto nonhomogeneous Poisson process model[19].

'Fault seeding' models. In this class we 'seed' a known number of faults into our product; when testing the product we find both seeded and unseeded faults. If we look at the proportion of seeded faults found compared to the number of unseeded faults found, we can estimate the total number of faults in the software product. The estimate is used to assess software reliability. The most widely spread model of this class is probably Mills' seeding model[28].

'Input domain based' models. The basic approach in this class is to generate a set of test cases from a distribution. The distribution should be chosen so that it is representative of the operation of the software product. Models in this class estimate the reliability from the outcome of the test cases. A model in this class is presented in Nelson[29].

Most models so far developed fall into the first two of these classes; the authors feel that this is no coincidence. These two classes are time dependent, which gives us an opportunity to estimate the forthcoming software failure occurrences. The latter two classes only give a stationary value, e.g. the number of faults in the software product or the reliability, but we obtain no information on how that number will decrease. The different classes of models will be suitable at different times during the project and for different applications, but this problem is not considered here.

The possibilities for comparing models within the classes are many, but the main problem is how to compare models when they study different failure processes. To compare models from different classes some common parameters have to be found; this is especially important for the two time-dependent classes, i.e. the first two listed above. It is well known from probability theory and queueing theory[30] that there is a relationship between the distribution of times between two consecutive events and the distribution of numbers of events in an interval. Thus, if this relationship could be used to compare one model from the 'times between failures' and one from the 'failure count' class, then it would be possible to use these two models as reference models for their classes. The reference models would work as a bridge between the classes, i.e. by comparing all models within a class with the reference model in that class, the models will be compared with the models from the other classes too. This bridge can be constructed between the time-dependent classes by comparing two frequently referenced models.

Comparison of two models

The two models which will be compared are only briefly described here. For more information on them, see Jelinski[18], Goel[19] and Goel[27].

Jelinski-Moranda de-eutrophication model (J-M model). This is one of the first models and probably one of the most commonly used for assessing software reliability. The model assumes that the time between failures follows an exponential distribution with a failure rate that is proportional to the number of remaining faults. By applying the maximum likelihood method, when we have observed a number of times between failures, we can estimate the initial fault content and the proportionality factor.

Goel-Okumoto nonhomogeneous Poisson process model (G-O model). This model was proposed after a study of actual failure data from many systems. Using various assumptions it was concluded that the number of faults detected by time t follows a Poissonian distribution with a time-dependent mean value

$$m(t) = N[1 - \exp(-zt)] \tag{4}$$

where N is the expected number of faults to be detected and z is the proportionality factor.

The two models above study two different failure processes. The J-M model is derived from the times between occurrences and the G-O model is developed from the number of occurrences in a given time interval. The relationship between the times between occurrences and the number of events in an interval is used to compare the two models presented above. The relationship càn be used for developing the distribution of the number of detected faults over time t for the J-M model; since this is known for the G-O model, it is possible to compare probability distributions, mean values, variances etc. for the two models.

The result is that the number of failure occurrences over time t for the J-M model follows a binomial distribution, with exactly the same time-dependent mean value as the G-O model. This means that we should compare a Poissonian and a binomial distribution with the same mean value. We also observe that the variances for these distributions are well known and can be compared with each other. Thus we can link the two time-dependent model classes.

The models have been compared thoroughly by Wohlin[31]. They are first compared analytically: mean values, variances and confidence intervals are considered, taking both time-dependent and asymptotic values. Then the two models are compared with collected failure data from two large software projects. The results are as one could reasonably expect. It is possible to build different realizations of a system, and the number of errors made when developing the system will consequently vary. This means that we cannot initially be certain which of these realizations we have in fact developed. Therefore the G-O model is suitable in the early stages of development because it treats the initial number of faults as a stochastic variable. When almost all faults have been removed, however, we know which of the realizations we have built and consequently the J-M model is the best one to use; this is because the J-M model treats the initial number of faults as a constant.

We have shown that it is possible to compare different models with each other and to see how they fit collected failure data. It is by no means certain that the same models are always the best — it is more likely that we will need different models during different phases of the software life cycle, e.g. for different test strategies during a project. This is backed up by the fact that the G-O and J-M models are suitable at different times.

Environment-adapted models

The two software reliability models compared in the previous section have been used successfully in earlier telecommunication projects[10]. The models were adopted, adjusted and successfully applied to several projects during the operational phase of the system. They were then applied to projects during the functional test stage; unfortunately they did not work.

The problem during the functional test phase is that some of the assumptions made for the models are far from valid[32]. There are some assumptions we have to accept and some we do not. One assumption made by almost all models, which we probably have to accept, is that either the times between failures are independent of each other or the number of faults detected during nonoverlapping intervals are independent of each other. This independence is usually not present, but experience shows that this violation does not significantly affect the results. Without assuming independence in probability theory we soon end up with very complex problems that are difficult, if not impossible, to solve.

Since the models used by Vrana[10] did not work during parallel testing, i.e. functional testing, of a system with an hierarchical structure, a model for the system structure and test environment used by Vrana was sought. No software reliability model was found that fulfilled our requirements and we therefore had to develop a model of our own, but before we could do this we had to examine in more detail how functional testing is performed. While doing this it is possible to identify the assumptions that are violated, and hence develop a model more suitable for functional testing than the existing ones.

An investigation of various different telecommunication projects was undertaken[23]. Managers, programmers, test groups etc. were interviewed to build up a picture of what happens during the different test stages, especially during functional testing. Based on this investigation a process model similar to the one presented by Huff[33] was developed, which led to the conclusion that the assumptions in software reliability models that were not fulfilled could be identified. By doing this it was possible to develop an environment-adapted model, i.e. a model whose assumptions better modelled the product's behaviour during functional testing[23].

The results show that we are able to get a good picture of how faults are detected during functional testing using the model developed. The predictions improve if we keep track of the number of hours used for testing each day. The data used for prediction in this example was collected from failure reports, so we do not know exactly how effective the testing was over different periods. We are, however, convinced that if data are collected directly for the model, we will get very good estimates of the times concerned. Before we can state this definitely we have to evaluate the model carefully on some other projects.

Conclusions on reliability models

Above we have established the possibility of comparing different software reliability models and of developing models adapted for a specific environment. It is essential to try to understand the stochastics of how software failures occur, to obtain an efficient, reliable, maintainable and manageable software product.

The main point of the comparison of reliability models is to indicate the need to choose software reliability models that are well suited to the environment and applications instead of taking the multimodelling or 'simply using' approach. This choice has been shown to be possible. Once realistic models have been identified, they must be compared to find the model which gives the most accurate estimate; this entails an investigation of available models and the information obtained from them. There has to be a criterion, objective and well known to everyone, that can be used when choosing the model to use in different projects. Different models may even have to be applied during different phases of the software life cycle.

The identification of realistic software reliability models to a specific environment or application can only be done if we make a survey of software projects, find the critical parts, examine the main points, and get an overall view of the behaviour of the underlying processes. This survey has already been undertaken for hardware, but it is also necessary for software. Systems are getting continually larger and more complex. If we do not want to end up with a software product that is 'out of control', it is necessary to adopt or adapt a realistic existing software reliability model or develop an environment-adapted model.

QUALITY CONSTRAINT MODEL

It is well known that it costs more to correct a software fault in the operational phase than in the test phase, but it is not economical to make the test phase very long simply to get rid of all faults. This means that there has to be an economical minimum at some time[34, 35].

The cost function is hard to elucidate; it depends on factors connected with the system, the environment, testing methods, the market, maintenance routines etc. But if we could find the cost function and assume that each fault contributes equally to the cost, then we would be able to find out how many faults we should try to correct during the test phase — before we release the software product — to obtain the economical optimum.

A logical approach to the problem, would be to

- decide how many faults we should remove in the test phase to minimize the cost function
- develop a model for deriving the mean and variance of the time we have to spend in the test phase to minimize the cost function
- plan for different cases, from best to worst, depending on how the test phase goes (a company should always be prepared for the worst and should not be surprised if it happens).

The model

We want to develop a model for determining the mean and variance of the time to spend in the test phase to minimize the cost function. A sensible way to do the testing would be to put a quality constraint, based on the number of faults to remove in order to minimize costs, on the product and allow the time before the product is put into operation to vary. The problem is solved using the following three assumptions.

- The initial number of faults when the test phase is started are estimated to a known accuracy. (Estimation of the number of faults from complexity metrics was discussed above. An improved estimate can be obtained from reliability models.)
- The failure time distribution is known. (Failure time distribution is known when realistic reliability models are used. The distribution could be found through measurements on earlier projects, for example. The models are used to estimate the software failure occurrences in the future; in this application we are not considering reliability itself. The use of reliability models has been discussed above.)
- If a fault is found, it is corrected and no new faults are introduced. (This assumption is quite natural, even though it is not always true, because it is always the aim when correcting faults. Deriving results will be easier with this assumption, but it is possible to find some results if we assume that the fault correction has a certain probability of success.)

It is possible to calculate the mean and variance of the time when a specific number of faults remain[36]. It should be observed that the formulae for doing this do not assume an equal contribution to the cost from all faults. This assumption only influences the quality constraint on the number of faults that should remain, and it could be partly overcome by putting a quality constraint on the failure intensity; this means, however, that we have to assume that the size of a fault is proportional to the cost of correcting it.

The results obtained here for numbers of faults against time give a good idea of the amount of time to test the product while fulfilling the quality constraints, and to see how this time varies. We should keep in mind that the results are examples of the inevitable consequences that the formulae[36] lead to. When using the formulae we should be aware of the following.

- Complexity metrics are needed to estimate the initial number of faults at an early stage.
- Failure time distributions must be found that are applicable to our own specific problems.
- There will probably be different failure time distributions during different phases of the software life cycle.

The principal results are shown in Figure 4, which is valid for all failure time distributions where the number of faults remaining decreases. If we consider a number of distributions with the same mean value, the difference in Figure 4 will be the variance (i.e. the length of the arrows in the figure).

We have estimated the initial number of faults to be N_0 and we would like to test the software product until we reach N_{opt}; this happens at T_e if we do not consider the variations in N_0 and T_e. Now let us consider variations of both N_0 and T_e and see what happens. If we assume that we have estimated N_0 to be somewhere within the interval ($N_0 - \Delta N_0$, $N_0 + \Delta N_0$), then we observe that we have a best and a worst case, remembering that N_0 was the initial number of faults. If we study the best and the worst case and assume that T_e varies as marked in Figure 4 by the broken arrows, then we obtain an interval ΔT, marked in the figure by the solid arrow, in which T_e lies. This interval is rather large for most current realistic failure time distributions, and to get a good estimate of T_e we have to make ΔT much smaller; otherwise we get a software product that is at least partly out of control.

Perhaps the most important aspect of the model discussed above is to understand the inevitable consequences of software failure stochastics. There are two

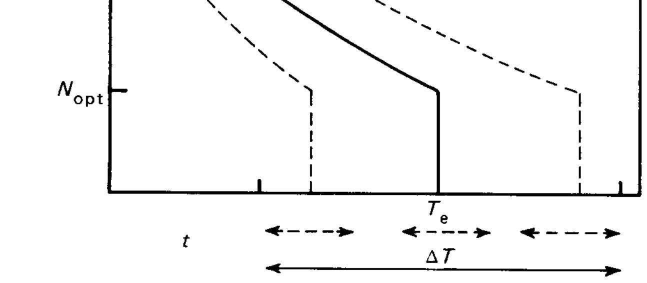


Figure 4. Principal results: number of faults N_t against time t. $N_0 \pm \Delta N_0$ is the estimated initial number of faults; N_{opt} is the 'ideal' level of faults at the end of the test period, i.e. at time T_e (which varies as shown by the broken arrows); ΔT is the total time span within which T_e may lie

primary and two secondary ways of improving the situation. The two primary ways, dealing with the problem of making the test phase shorter, are

- to reduce the number of initial faults, perhaps by developing better methods and tools for the design and construction
- to develop better methods and tools for the test phase, i.e. more systematic testing routines

The two secondary ways of making the situation better deal with the problem of making the variances smaller; they are

- to obtain a better estimate of the number of initial faults (i.e. reduce N_0)
- to decrease the variance in T_e, again using better methods and tools during testing and more systematic testing routines

If we do not make the estimated interval for T_e smaller, two things, both of them uneconomical, could happen: we could stay in the test phase longer than necessary, just to be on the safe side; or we could leave the test phase too early, which could lead to the software product being very expensive to maintain and, in the worst case, to an inefficient, unreliable, unmanageable and unmaintainable software product.

CONCLUSIONS

The need for software metrics will continue to grow as the systems being developed become larger and more complex. Software metrics will be one way to ensure that the system development process is under control, that the project proceeds as planned, and that the quality constraints are fulfilled. Software metrics have to be introduced into organizations developing software products in order to cope with the demands put on the systems.

The emphasis in this paper has been on fault content and reliability estimations, but software metrics are needed for all aspects concerning quality, resources, economy etc. Fault content estimations are very important when developing software products because time schedules, quality and the overall economic result of the project are often highly dependent on the fault content.

It has been shown here that complexity measurements are valuable in a number of applications. To plan and control software projects, it is essential to measure complexity early in the software life cycle. This summarized study shows an important property of a well defined description language such as SDL — the possibility of measuring, estimating and controlling before the coding phase. We have also emphasized the need for complexity studies within the actual project environment rather than adapting models and results from other environments.

Reliability models give us the possibility of estimating the reliability of the product, and they can be a useful tool in the process of controlling and planning a software project both during development and operation. It is essential, however, that the models used are realistic. By this we do not mean that the models have to be perfectly matched to the environment and techniques used, but we have to be aware of the differences between them. To find suitable models for different applications the models have to be compared with each other and evaluated; if no model is found to be realistic for the application it is, in most cases, possible to develop an environment-adapted model.

The possibility of using complexity metrics and reliability models together to control and plan the release of the software product has been discussed.

In the future software metrics will be a natural ingredient in the development and operation of software products. We will have tools for data collection, databases with information from earlier projects, handbooks for software metrics and expert systems for different aspects of a system, e.g. software reliability models. This will make it possible to develop larger and more complex systems of high quality within set time schedules and budgets.

The ideas and techniques presented are beginning to be used within the Swedish Telecommunication Administration (STA). Tools and methods are developed at Telelogic AB, a subsidiary of the STA. The software reliability group at Ellemtel Telecommunication System Laboratories, a subsidiary of the STA and Ericsson, has been testing metrics and models for several years, and research at Lund Institute of Technology continues. Since the field is quite new it is important to exchange experiences and results on all aspects of software metrics and models, both research results and practical experiences; this is an integral part of the programme at the Lund Institute.

REFERENCES

1 **DeMarco, T** *Controlling software projects* Yourdon Press, New York, NY, USA (1982)

2 **Conte, S, Dunsmore, H and Shen, V** *Software engineering metrics and models* Benjamin/Cummings, Menlo Park, CA, USA (1986)

3 **Basili, V R and Rombach, H D** 'Tailoring the software process to project goals and environments' *Proc. 9th Int. Conf. Software Engineering, Monterey, CA, USA (IEEE Cat. No 87CH2432-3)* (1987) pp 345–357

4 **Kitchenham, B and Walker, J** 'The meaning of quality' in **Barnes, D and Brown, P (eds)** *Software engineering '86* Peter Peregrinus, Stevenage, UK (1986) pp 393–406

5 **Boehm, B** *Software engineering economics* Prentice-Hall, Englewood Cliffs, NJ, USA (1981)

6 **Rapp, D and Vrana, C** 'Systems and systems management environments' *NT Symposium, Turku, Finland* (1984)

7 **Vrana, C** 'S/W engineering economics — models for systems with a hierarchical modular structure' *NT Symposium, Turku, Finland* (1984)

8 *CCITT Recommendations Z101–Z104 — Red book Vol VI Fascicle 10–11* (1984)

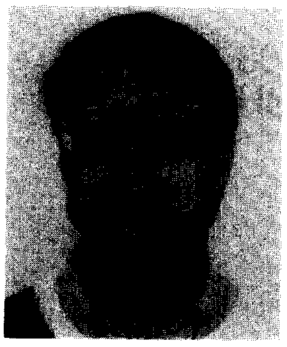
9 **Lennselius, B** 'Software complexity and its impact on different software handling processes' *Proc. 6th Int. Conf. Software Engineering for Telecommunication Switching Systems (IEE 259), Eindhoven, The Netherlands* (1986) pp 148–153

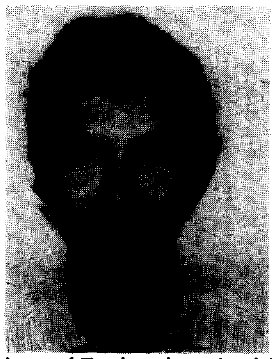
10 **Vrana C and Wallander, A** 'S/W quality and complexity — different aspects and measurements results' *Proc. 5th Int. Conf. Software Engineering for Telecommunication Switching Systems (IEE 223), Lund, Sweden* (1983) pp 121–127

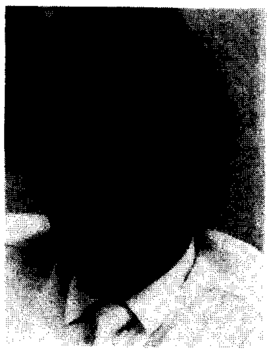
11 **Wohlin, C** 'A study of software complexity' *MS Thesis, Lund Institute of Technology, Lund, Sweden* (1983)

12 **McCabe, T J** 'A complexity measure' *IEEE Trans. Software Eng.* Vol SE-2 No 4 (1976) pp 308–320

13 **Lennselius, B and Vrana, C** 'Complexity in the SDL-graph-description and its impact on different handling activities' *Proc. 2nd SDL Users and Implementors Forum, Helsinki, Finland* (1985)
14 **Halstead, M** *Elements of software science* Elsevier, New York, NY, USA (1977)
15 **Kafura, D and Canning, J A** 'Validation of software metrics using many metrics and two resources' *Proc. 8th Int. Conf. Software Engineering (IEEE Cat. No 85CH2139-4), London, UK* (1985) pp 378-385
16 **Yu, T J** 'The static and dynamic models of software defects and reliability' *PhD Thesis, Dept of Computer Science, Purdue University, West Lafayette, IN, USA* (1985)
17 **Takahashi, M and Kamayachi, Y** 'An empirical study of a model for program error prediction' *Proc. 8th Int. Conf. Software Engineering (IEEE Cat. No 85CH2139-4), London, UK* (1985) pp 330-336
18 **Jelinski, Z and Moranda, P** 'Software reliability research' in **Freiburger, W (ed.)** *Statistical computer performance evaluation* Academic Press, New York, NY, USA (1972) pp 465-484
19 **Goel, A and Okumoto, K** 'A time dependent error-detection rate model for software reliability and other performance measures' *IEEE Trans. Reliability* Vol R-28 No 3 (1979) pp 206-211
20 **Schneidewind, N** 'Analysis of error processes in computer software' *Proc. Int. Conf. Reliable Software, Los Angeles, CA, USA* (1975) pp 337-346
21 **Littlewood, B and Verral, J** 'A Bayesian reliability growth model for computer software' *Appl. Statist.* Vol 22 (1973) pp 332-346
22 **Musa, J D and Okumoto, K** 'A logarithmic Poisson execution time model for software reliability measurement' *Proc. 7th Int. Conf. Software Engineering, Orlando, FL, USA* (1983) pp 230-237
23 **Wohlin, C** 'Software testing and reliability for telecommunication systems' in **Barnes, D and Brown, P (eds)** *Software engineering '86* Peter Peregrinus, Stevenage, UK (1986) pp 27-42
24 **Goel, A** 'Software reliability models: assumptions, limitations and applicability' *IEEE Trans. Software Eng.* Vol SE-11 No 12 (1985) pp 1411-1423
25 **Goel, A, Basili, V and Valdes, P** 'When and how to use a software reliability model' *Proc. 7th Software Engineering Workshop, Greenbelt, MD, USA* (1983)
26 **Abdel-Ghaly, A A, Chan, P Y and Littlewood, B** 'Evaluation of competing software reliability predictions' *IEEE Trans. Software Eng.* Vol SE-12 No 9 (1986) pp 950-967
27 **Goel, A** 'A guidebook for software reliability assessment' *Rep. RADC-TR-83-176, Syracuse University, Syracuse, NY, USA* (1983)
28 **Mills, H** 'On the statistical validation of computer programs' *Rep. 72-6015, IBM Federal Systems Division, Gaithersburg, MD, USA* (1972)
29 **Nelson, E** 'Estimating software reliability from test data' *Microelectron. Rel.* Vol 28 (1978) pp 428-443
30 **Kleinrock, L** *Queueing systems, Vols 1 and 2* Wiley, New York, NY, USA (1975-76)
31 **Wohlin, C** 'The possibilities of comparing software reliability models' *Tech. Report, Lund Institute of Technology, Lund, Sweden* (1986)
32 **Ehrlich, W K and Emerson, T J** 'Modeling software failures and reliability growth during system testing' *Proc. 9th Int. Conf. Software Engineering (IEEE Cat. No. 87CH2432-3)*, Monterey, CA, USA (1987) pp 72-82
33 **Huff, K E, Sroka, J V and Struble, D D** 'Quantitative models for managing software development processes' *Software Eng. J.* No 1 (1986) pp 17-23
34 **Křtěn, O and Levy, D** 'Software modelling for optimal field entry' *Proc. Ann. Reliability and Maintainability Symp.* (1980) pp 410-414
35 **Wohlin, C and Körner, U** 'Software faults: spreading, detection and costs' *Tech. Report, Lund Institute of Technology, Lund, Sweden* (1987)
36 **Wohlin, C and Vrana, C** 'A quality constraint model to be used during the test phase of the software life cycle' *Proc. 6th Int. Conf. Software Engineering for Telecommunication Switching Systems (IEE 259), Eindhoven, The Netherlands* (1986) pp 136-141


Bo Lennselius was awarded an MSc in electrical engineering in 1983 at the Lund Institute of Technology, Lund, Sweden. He was a research student at the Department of Communication Systems, Lund Institute of Technology from 1983 to 1986, and he is currently working in the same department in the field of software engineering, specializing in software metrics and complexity measurements. He also works as a consultant.


Claes Wohlin received an MSc in electrical engineering in 1983 from the Lund Institute of Technology, Lund, Sweden, and was a research student at the Department of Communication Systems, Lund Institute of Technology from 1983 to 1986. During this time he worked in the field of software reliability, and he was awarded a Licentiate of Technology in this field in 1986. He has been a research engineer at the Department of Communication Systems at Lund since 1986. He is currently working in the field of software engineering, specializing in software performance, especially reliability and effectiveness.


Ctirad Vrana received an MSc in electrical engineering in 1976 from Lund Institute of Technology, Lund, Sweden, where he remained until 1982. He worked with methodologies and quality issues for software at the headquarters of the Swedish Telecommunication Administration during 1982-1983, and was awarded a Licentiate of Technology in software engineering at the Department of Communication Systems at Lund in 1984. He is currently head of the Department of Education at Telelogic AB, Sweden.