QUANTITATIVE SOFTWARE COMPLEXITY MODELS: A PANEL SUMMARY

Victor R. Basili

Department of Computer Science
University of Maryland

Several participants at the conference formed a panel on software complexity measures. We posed four questions of ourselves:

1. What is a complexity measure and what aspects of software development does it cover?

2. How should a metric be developed and evaluated?

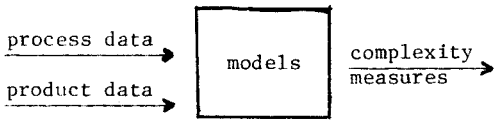
3. What are some of the uses of metrics?

4. In the final analysis, where is the area going and where are we now?

Defining A Software Complexity Measure

The panel accepted as a working definition of software complexity the following: Complexity is the measure of the resources expended by another system in interacting with a piece of software. Categories of systems that may interact with software are machines, other software, people, and even the external environment. If the interacting system is a machine, the measures deal with execution time and memory space. If the interacting system is software, the measures might focus on the number of interfaces. If the interacting system is people, the measures are concerned with human efforts to comprehend, to maintain, to change, to test, etc., that software. The external environment acts more like a set of constraints; that is, if a software development project requires travel to another site, there are certain physical limitations and expenditures in travel time that must be considered.

To calculate a complexity measure, data from the software product or process are transformed according to models into a set of complexity measures.



There are several classifications for complexity measures. We can think of the complexity of the process or the product. Process metrics would be based on time to develop, number of errors, etc. Product metrics would be based on number of decisions, number of interfaces. A second dichotomy is to think of measures as either quality metrics or invariants. A quality metric would be something that would evaluate the product as good or bad relative to a specific model. For example, reliability would be a quality metric and allow us to evaluate the product from a quality point of view. On the other hand, an invariant is a measure that is impervious to various environmental changes, for example, the relationship between effort in man months and lines of code has been shown to be invariant across many environments. A third classification scheme would be a priori vs. a posteriori metrics. An a priori metric would be used to estimate and evaluate what the product will look like eventually. An a posteriori metric is a measure of the existing product after it is completed. All metrics should be characterizable into these three classes of description. For example, the complexity metric which is defined as a count of the number of errors incurred in developing a system would be an a posteriori process quality metric. Halstead's length metric, on the other hand, would be a product metric which may very well be an invariant but again a posteriori. However, if we took Halstead's potential volume as a metric, that would be a product metric invariant to various environments and could be used as an a priori metric to estimate the actual size of the final product.

Developing A Software Measure

There are two major phases in the development of any measure. First is the analytical phase and second is the experimental phase. During the analytical phase, a model of the product or process is developed, representing a particular viewpoint. Based upon this model, a metric or set of metrics are defined which attempt to operationalize the model. In some sense, metrics are an encoding of the model that can be used to quantify software life cycle phenomena from the viewpoint of the model.

Some abstract analysis can be performed on the model and its associated metrics. The metrics should behave in a consistent way and should have reasonable boundary conditions. A model is consistent if it behaves in a similar manner, given intuitively similar data. For example, a slight change in a system's decision structure should not drastically change the value of a control flow complexity metric for the system. A model has reasonable boundary conditions if its limits correspond to intuitive expectation based on the model's viewpoint. For example, if based on the model there is no expenditure of resources in the system at certain points, then the complexity

Reprinted from WORKSHOP ON QUANTITATIVE SOFTWARE MODELS, October 1979

metric should have a value of zero. In many cases, the metric would need to be normalized and the limits made explicit. These constraints and bounds are based predominantly on analysis.

However, no matter how tractable a model we have and how nicely the metric behaves analytically, it is of no value unless we can validate its relevance via empirical experimentation. It is critical that we demonstrate that the metrics and their defining model correspond to reality.

There are several approaches to evaluating a metric empirically. The most primitive form of evaluation is called a case study--a single product being developed. Here data is collected during some phase of the life cycle of a system and the metrics are evaluated on that data. This application can be used to provide some preliminary evidence that the metrics correspond to the model of what we are studying and can be used to fine tune the model and metrics. A second form of evaluation is called a quasi-experiment. In this case, several products of a similar nature are developed and compared based on the data collected. It is an environment in which causal relationships can be suggested but not proved, but it gives us much more insight and empirical confidence than a single case study would. A third type of data collection environment would be a controlled experiment, that is, a duplication of identical developments in a controlled environment. Clearly this is the most ideal data collection and metric evaluation environment. Unfortunately, it is very expensive and difficult to achieve. This type of experiment can be performed when we have gained confidence from case studies and quasi-experiments.

One possible solution to the problem of validating software metrics outside of a controlled environment is to conduct many case studies and quasi-experiments. In this way, a wealth of independent experiences can be assembled to generate confidence in the value of the metric. In order to do this, however, results must be published using agreed-upon, or at least well-defined, terms and explicitly-stated environmental constraints so that the next experimenter can be sure he is testing the same thing.

It is clear that measures must be evaluated across the entire life cycle. That is, although most experiments have been done during the coding phase only, we are interested in collecting data and developing metrics that encompass the system from early inception and planning all the way through the maintenance phase. A measure may then be evaluated by first isolating its direct effect on the data and then correlating that effect with experience. These properties and results are then evaluated with respect to the original model that was developed during the analysis phase.

Using A Software Measure

There are several uses for metrics and the panel suggested three possibilities: (1) they can be used to evaluate the software process and product, (2) they can be used as a tool for software development, and (3) they can be used to monitor the stability and quality of an existing product. Better understanding of the software development process and the software development product is a critical need. Metrics can help in that understanding by allowing us to compare different products and different development environments and providing us with insights regarding their characteristics. Too often we think of all software as the same. Metrics can be used to delineate the various software products and environments.

Many metrics have as a major goal the evaluation of the quality of the process or product in a quality assurance environment. Thus a low score, on a metric like the number of errors, indicates something desirable about the quality of the process while a high score on the same metric indicates something quite undesirable about the product.

A second use of metrics would be as a tool for development. In this case, the metric can act as feedback to the developer, letting him know how the development is progressing. It can be used to predict where the project is going by estimating future size or cost, or it may tell him his current design is too complicated and unstructured. Metrics should certainly be used across the entire life cycle and as early as possible to facilitate estimation as well as evaluation.

A third use of metrics is to monitor the stability and quality of the product through maintenance and enhancement; that is, we can periodically recalculate a set of metrics to see if the product has changed character in some way. It can provide a much needed feedback during the maintenance period. If we find over a period of time that more and more control decisions have entered the system, then something may have to be done to counteract this change in character. This last use of metrics is relativistic, requiring only a simple partial ordering to give us an indication of what is changed. A relative measure is clearly easier to validate than an absolute measure. The first two uses of metrics--the evaluation of the process and product and the tool of development--are predominantly absolute metrics; that is, there is nothing to compare them to within the same project. You may only compare their values with the values of the metrics on other projects. The drawback to an absolute metric is that we need some normalization and calibration factor to tell us what is good and what is bad.

The Effect of Software Metrics

Metrics should affect software methodology. Assuming we learn from their use more about what a good product is and what a good process is, we will gain valuable insights into changing, refining, and developing new methodology. On the other hand, surely this changing methodology and technology will affect what we model and measure. For example, in an environment where all code is written in a structured programming language, a metric that evaluates control flow structure is useless, although measures of modular structure

would still be useful.

One important issue is the current effect metrics are having in contracts. A combination of models and associated metrics would be of great value for independent validation and verification and quality assurance. In fact, metrics are already being used in contract-related issues, such as award, acceptance, and budget incentives. Unfortunately, it is not clear that they are minimally sufficient for the purpose yet. They may often be misleading. For example, one specific contract that was written required a 47-year mean time to failure. This is clearly beyond the current state of technology. Most metrics have not been tested enough in different environments to assure the kind of rigor that is required in satisfying contract requirements. They should be used cautiously until a stronger basis has been established for their validity. What we need is a set of metrics that are well understood, have been validated through empirical study, and that help in developing and monitoring contracts.

The final evaluation of whether complexity measures are worthwhile, however, will lie in their cost effectiveness; that is, metrics will survive only if they prove to be beneficial from a cost point of view. If a metric measures only a microscopic aspect of the software development process or product, then clearly it will not be cost beneficial. But if it gives us some panoramic insight into the process, if it has some effect on the really important issues of cost estimation and quality control, then the work on metrics will have been worthwhile.