Dy

ANALYTIC TECHNIQUES FOR

EVALUATING

SOFTWARE DESIGNS AND METHODOLOGIES

SEPTEMBER 1978

by:
Paul A. Scheffer

MARTIN MARIETTA AEROSPACE
DENVER, COLORADO

I. INTRODUCTION

In recent years several methodologies have been developed to assist in the software development process at the requirements-to-design engineering phase. The objective of many of these approaches is to assist the designer in deriving, from a given set of requirements, a modular framework for the system which can be associated with qualities such as adaptability to changes in requirements, testability, maintainability, interface correctness, etc. Further, in many cases these approaches are complemented by a language designed to support the basic concept of the technique. Systems such as PSL/PSA, SREM, SSL and strategies attributed to Dijkstra, Mills, Parnas, Jackson, and Myers fall into this class and are currently receiving much attention by the software engineering community. This attention can be attributed in part to the growing recognition that rigor at the requirements and design phases tends to minimize the costly "error days" associated with software. It also allows for manageability of evolving requirements and addresses the true life cycle costs of systems. This attention, however, has yet to lead to guidelines for an intelligent selection from this wave of methodologies. With some minor exceptions, the lack of quantitative and careful qualitative evaluation of benefits and costs associated with "front end" development strategies is notable.

In this paper, several analytic techniques are discussed as they might be applied to software design. Such techniques show promise on two fronts: the ability to quantitatively measure various aspects (viz. qualities) of a given design statement; and, by using quality indexes for several designs produced under different auspices, to yield a comparative assessment of individual strategies, techniques, or personnel. The rationale behind the comparision idea is the fact that the common denominator of all design strategies, at any level, is their treatment of the structural characteristics of problems, i.e., the systematic decomposition of the original problem into a logically organized set of subproblems which contribute to the ultimate objective. The assumption is made that attributes of design quality are sufficiently manifested in the structural characteristics of the design so that they can be measured by a static analysis. (Similar to measures of complexity for code.) Further, we believe an appropriate scheme for static analysis of structure may be completely adequate for comparing different strategies, and can also provide a general tool for the development of superior quality software.

The application of analytics in this paper is not dependent on either a specific methodology or the level of development to which it applies. Since any methodology has a "product" in which its influence is contained, we simply consider associating a measure with that product and hence provide for evaluation of different methodologies or design strategies.

II. METHODOLOGY REVIEW

A. Background

Many attempts have been made at defining the crucial aspects of software development, both in terms of the finished product, viz. the design, and the activity itself, the design process. By looking at finished designs, one hopes to be able to generalize the characteristics that differentiate the better products. The problem then is determining how to effect the desirable result – the good design – within the design process. Lacking a methodology which assures a quality design. about all one is left with is an iteration and assessment cycle on designing and its end result until a satisfactory development is achieved.

A methodology for design is generally applicable to a particular phase of the total design process. As we learn more and more about software phenomena, methodology principles are discovered which can be applied earlier and earlier in the design activity. (Compare for example the precepts of structured programming from a few years ago, to Dijkstra's levels of abstraction, hierarchical systems and families, and the "top-down" approach.) Hence we now have methodologies for requirements analysis, functional analysis, and system structuring as well as detailed design and programming. However, as we shall discuss in the next section, the study of these is dependent on the manner in which they are expressed. In the short methodology review which follows, the use of design languages should be envisioned in which the design concepts can be expressed.

B. Jackson – Data Structure Orientation

The Jackson approach (7) to software design considers the input data structure as the driving force to program design. The program is viewed as the means by which input data are transformed to output data. By paralleling the structure of the input and output data, the analyst can presumably be assured of a quality design, at least if "quality" data structures exist a priori. This approach implicitly relies on the rationality of the data structures used and acknowledges a restriction to sequential files and applications which do not require a DBMS.

C. Myers and Constantine – Composite Design

Although this approach (10, 20, 23) deals more with programming principles than overall software development, the ideas can readily be generalized. In considering the design of modules, Myers differentiates the attribute of structure from those of function (purpose) and performance (behavior):

> "Structure is a description of the construction of a program, in such terms as coding structure, module structure, task (parallel-process) structure, memory layout, and module interfaces."

The composite design approach combines the structure aspects of modules, data, tasks, and their interfaces.

The methodology influences the development of programs in that it directs an iterative decomposition of structural components, called composite analysis, using defined principles of strength and coupling. Designing affects the organization of modules and module elements which produces the amount of "related-ness" among elements. The degree to which elements in the same module are related is called strength, and the methodology calls for maximizing this factor. Relationships between elements in different modules measure coupling, and are to be minimized. For any given program design, the degree to which these principles are adhered to can be measured as program stability in terms of a probability value on the impact of changes.

D. Parnas – Information Hiding

On a more abstract level, David Parnas (12) has presented a way of designing software to produce a system structure which is more stable in the sense of being adaptable and flexible when it comes to localizing the effects of a change. The Parnas methodology is based on the concept of "information hiding". The design decomposition criteria concentrates on the decisions which must be addressed in the software development process. The decomposition method entails the maximal containment of information by each module so that its interface and definition reveals as little as possible about its inner workings. Modules, i.e., software systems, developed in this manner will have a decomposition structure much less mechanically arrived at.

With this approach, modules that are components of the same system know only what is necessary about one another, no more and no less. An important aspect here is the emphasis on the hierarchical organization of a system and the levels of abstraction which alone helps to minimize module dependencies and complexities. By ordering levels from the most primitive to the most abstract, each level appears as a virtual machine for the level above it. This in itself restricts the information a module on one level needs about the operation of modules on lower levels.

III THE ANALYSIS STRATEGY

A. Approach

The study of design strategies is dependent on two important factors. One is that some "tangible" or machine readable representation of a problem must be available which lends itself to use computer aids in studying strategies. The second is that a "canonical form" of structural representation can be derived upon which certain assessment metrics can be based. These two factors are generally realized by high level design languages and (tree) graphs. Most analytic techniques rely on the ability to transform any methodology "product" to an equivalent graph structure of nodes and links representing design elements and element relationships. This is especially true when considering aspects of structure and structural decomposition. However, there are less complex forms of analysis which can be derived directly from the syntactic constructs provided by a formal expression medium.

In Figure 1, a model for studying design strategies is presented. It basically summarizes the concepts and relationships involved and can be used as a focal point for the discussion which follows. The process described in Figure 1 reflects the assignment of a measurement characteristic as a goal for a selected problem statement or expression. The center of the figure identifies the analytic techniques which can be used – applied to a given expression to measure a specific characteristic. The bottom of the figure illustrates this for the older source code analysis technology. The majority of the techniques listed are based on the notion of structural decomposition. The decomposition problem considers the partitioning of a set, in which "interdependencies" among elements have been defined, into a collection of subsets (clusters) characterized by:

1. Strong interdependencies among elements with a cluster.

2. Weak interdependencies between elements in different clusters.

The decomposition problem is readily handled by using a tree graph model of the set which sonsiders interdependencies as node links (edges) and connected sub-trees for the partioning. The characterization then appears as a strength and coupling measure on the clusters.

B. Expression

The expression of a given problem can take many "forms", depending on the vehicle used to state it. In traditional software developments this has simply been natural language. However, it is the recent availability of specialized tools – the specification and design languages and methodologies – which allows formalized approaches to expression. This in turn is what has provided us with machine processible data representing some design level. It is the formalization provided by syntactically structured and unambiguous methods and vocabularies which provides this "data base".

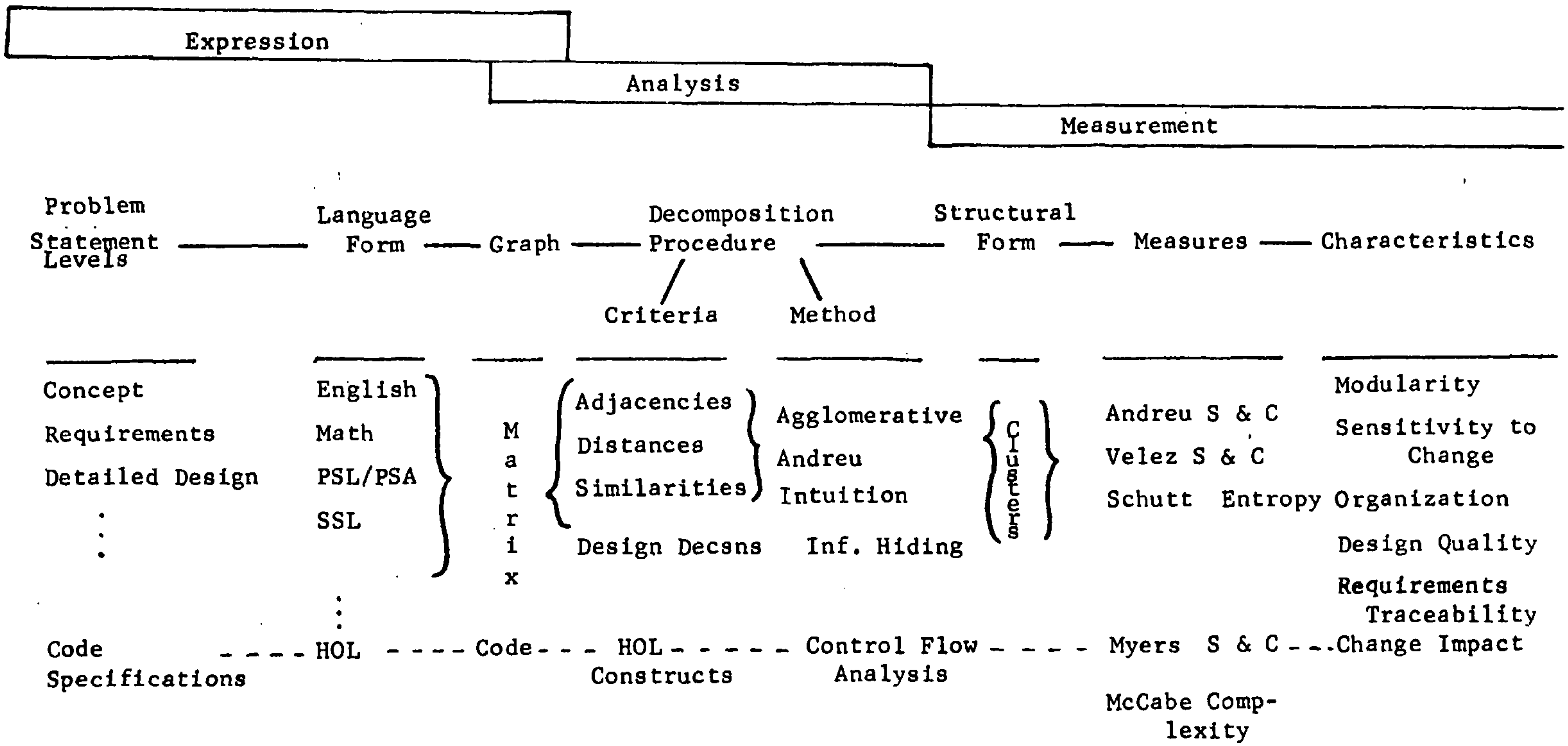

Figure 1. Formulation of the Study of Design Strategies

Several language forms are listed in Figure 1. Each has its peculiarities as to the design level it best expresses. For example, English is better for conceptualization, mathematics for algorithmic specifications, and Higher Order Languages for detailed design representation. The PSL/PSA scheme was originally intended as a documentation tool but continuing developments are making it widely applicable to many design levels. At Martin Marietta/Denver, we are developing a specialized requirements language (MEDL/R) to fill what we feel is a void for that level.

C. Analytic Techniques

1. The Andreu Approach – In (2), Andreu describes how the graph decomposition problem can be approached with both classical and hueristic cluster analysis techniques. Algorithmic schemes are based on a matrix of coefficients which be an array of adjacencies, minimum-path distances, or similarities; each provides a distance function representation of how a node is related to all the other nodes in a graph. Algorithmic methods are then described which use these metrics for the decomposition criterion in producing different structural clusterings of graph nodes.

The basic hueristic algorithm defines the "nearest-neighbor" set of nodes for each component of the graph. A distance count d_{ij} identifies the number of links between any two nodes. The distance metric used is what determines the meaning of "nearest". Hence for each node i, a neighborhood set is defined:

$$N_i = \left\{ n_j \mid d_{ij} < T \right\}$$

Clusters are produced by considering the subsets for which $|N_i|$ is greater than some threshold parameter k. The heuristic assumption is that the code notes n_i of these k neighborhoods form "kernels" of importance over the entire graph. By forming successive intersections of neighborhoods and considering kernal clusters as nodes, the process becomes iterative. Inter-cluster linkages are determined, leaf nodes and "unimportant" clusters get merged (set union) and a decomposition is formed. A strength and coupling measure is used to evaluate a particular decomposition; strength is increased by node counts and linkages within a cluster, coupling is reduced by fewer inter-cluster linkages.

2. The Agglomerative Techniques – These are procedures which start with a set of n one-member clusters and try to reduce the number as dictated by some meaningful criteria. Typically, agglomerative techniques are measure independent in that they proceed until the number of clusters is reduced to one (the entire graph). The order in which elements are assigned to clusters that will eventually merge into the complete original set is then used to identify a "reasonable" set of clusters. Variations on these techniques are possible by using different decomposition criteria (e.g. dissimilarity matrix instead of similarity; various definitions of node-node distance). The major disadvantage of agglomerative techniques is that early decisions which categorize (cluster) a node cannot be changed at later stages – hence an initially poor assignment can never be reconsidered or recovered from. Also, some measure must be used to describe which of the interim partitionings is the most useful.

3. Static Analyzers – Another class of design decomposition schemes is based on the linguistic expression of a design. The ready availability of a "tangible" form of a design represented by its collection of "source" statements has prompted the development of many analysis tools analagous to source code analyzers. Most prominent in this activity are the measurement schemes which result in "complexity scores". These are developed from some static form of analysis of statements which generally assess:

- keyword occurrances and relationships

- module linkages, e.g., calling forms, parameter lists, common areas, etc.
- syntactic structures, e.g., assignments, DO's, CASE, etc.
- control structures, e.g., IF ....., CALL, Branches, etc.

Graph structures are not directly involved in these cases; a total decomposition (where each node is a cluster) is essentially assumed. Hence static analyzer programs use statistical and control flow analysis techniques to produce measures of quality, complexity, or structuredness.

Most noteworthy in this area is the approach of Myers (10) who develops structure criteria based on probability measures of dependencies between modules. The probability measures are developed from applying module strength and coupling measures to design structures which are easily envisioned in the source code implementation. Even though heuristically formulated, this approach seems most practical in the determination of a software system's sensitivity to change brought about by simple maintenance of fluctuating requirements.

D. Measurement

The ability to compute a "quality of design" score is fundamental in the scheme for strategy comparison. Recent research has developed design quality metrics and measures from various viewpoints. Andreu (1) uses a strength and coupling measure applied to a graph representation of requirements and their inter-relationships for preliminary design. Myers (10) has developed a model in terms of probability measures applied to discrete strength and coupling factors of program modules which is used to assess the ramifications of making program changes. Schutt, et. al. (3,17) have applied an information entropy measure to hypergraph representations of computer processes and data structures. And, McCabe (8) shows a method for determining quality as a function of module "structured-ness". Each of these approaches relates in some way to a system measure.

We feel that the concept of subset strength and coupling (S&C) as used by Myers, Constantine, Andreu, et. al. is most appropriate as a quality measurement. The S&C concept assumes that it is the links which give structure to the entire graph. Consequently a structural evaluation of a partitioning involves the determination of how tightly coupled (linked) the nodes within a cluster area, as well as the extent to which two different clusters are related (number of linkages between).

In Figure 1 we listed several explicit ways of producing a "quality of design" score. While each of these has its own merits, the actual formulation of a quality measure is not as much at issue as what it means and what we can do with it. For example, the Andreu Strength and Coupling measure associates "goodness" with both modularity and sensitivity to change, in the sense that he derives his input from requirements statements and is concerned with the impact of fluctuating requirements. The Myers S&C measure is probability based using source code relationships as input. The interpretation here is that a programming change in any one module will affect all other parts with some probability as a function of how strongly the two modules are joined (coupled) and how isolated the changed module is (strength). The McCabe measure on source code produces a value of module complexity which characterizes a degree of structured-ness, i.e., how well the module conforms to precepts of structured programming. This is related to the Schutt, Gileadi, et. al. approach which deals in information entropy and software work. Using agglomerative clustering schemes, each node or additional nodes can be associated with a probability of misclassification for each existing cluster which can be used to determine some facet of modularity, organization, design quality, or sensitivity to change – obviously highly interpretive.

We see then, that with the various measures of design that can be used, each has a different shade of "quality" associated with it. Ideally, we would like to have a separate measure for each clearly identifiable software quality characteristic. To achieve this end, much research and experimentation is needed to calibrate and validate specific techniques to insure that the measures produced accurately reflect the attributes of the design problem.

APPENDIX

Measurements of a Requirements Expression

1. Completeness – The ultimate goal of this measure is to provide the analyst some means of determining whether or not more work needs to be done to a requirements expression to make it usable in determining a design expression, i.e., are the requirements that I presently have available of sufficient information content that a meaningful design activity can commence? A measure for completeness was derived by using two MEDL-R constructs and applied to sample "problem" sets of requirements. The measure is a percentage value which shows the degree to which all requirements are either RESOLVED in an appropriate manner or DERIVE others. (A greater percentage implies more completeness.) Appropriate resolution is determined by checking NATURE keywords against the type of RESOLUTION; e.g., NATURE: DATA implies that a DATA-RESOLUTION descriptor should be given

Sample Data:

Version	Pop.n	Derives	Resolved	%
I – Baseline	100	11	31	42
II – Baseline	7	1	0	14
II – A	12	1	0	8
II – B	15	1	5	40

The reduction in completeness in Version II-A from its Baseline is due to more requirements being added to the A revision. In Version II-B, further information was imparted to the requirements set by providing additional resolution of existing requirements.

2, Consistency Measurement – This measure should provide the analyst a means of determining the extent to which requirements in a set are isolated from one another. The term "isolation" means that property of a requirement that determines the extent to which it is bound to the total requirements set. The consistency measure used represents an average "strength" value, i.e., a large value implies a large degree of dependence among requirements. The value is determined by counting the number of names the requirements have in common.

Sample Data:

Version	Pop.n	Names	Strength
I – Baseline	100	47	.47
II – Baseline	7	4	.57
II – A	12	7	.68
II – B	15	9	.60

3. Complexity Measurement – This measure should provide a means to determine the inherent complexity of a set of requirements where the term "complexity" is still vague. However, it is clear that the greater the complexity of a set of requirements, the more difficult those requirements are to understand and hence more difficult to verify that they have been satisfied. So we chose a method that produces a probability that a node in a DERIVES tree is a non-terminal node. The higher this probability, the more complex the requirements expression.

Sample data:

Version	Pop.n	Non-Terms.	Probability
I – Baseline	100	27	.73
II – Baseline	7	6	.14
II – A	12	11	.08
II – B	15	13	.13

REFERENCES

1. Andreu, R. C., "A Systematic Approach to the Design of Complex Systems: An Application to DBMS Design and Evaluation," Center for Information Systems Research, Report #32. MIT, 1977.

2. Andreu, R. C., "Set Decomposition: Cluster Analysis and Graph Decomposition Techniques," CISR Preliminary Report, MIT/Sloan School, June 1977.

3. Gileadi, A. N. and Ledgard, H. F., "On a Proposed Measure of Program Structure," SIGPLAN Notices; May 1974.

4. Halstead, M., Elements of Software Science, Elsevier 1977.

5. Hamilton, M. and Zeldin, S., "HOS – A Methodology for Defining Software," IEEE Transactions, SE-2, March 1976.

6. Hartigan, J., Clustering Algorithms, Wiley, 1975.

7. Jackson, M., Principles of Program Design, Academic Press, 1975.

8. McCabe, T. J., "A Complexity Measure," IEEE Trans SE-2 No. 4, December 1976.

9. Myers, G. J., "An Extension to the Cyclomatic Measure of Program Complexity," SIGPLAN Notices, October 1977.

10. Myers, G. J., Reliable Software Through Composite Design, Petrocelli/Charter, New York, NY, 1975.

11. Paige, M. R., "On Partitioning Program Graphs, " IEEE Transactions, SE-3, No. 6, November 1977, p. 386.

12. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM 15, 12 (December 1972), p. 1053.

13. Robinson, L., "The Relationship of System Families to HDM" (Hierarchical Development Methodology) – Stanford Research Institute, TR:CSL-50, June 1977.

14. Roubine, O., "The Design and Use of Specification Languages," SRI Tech Report CSL-48, October 1976 (AD/A 038-783).

15. Scheffer, P. A., "Computer-Aided Software Design," Martin Marietta Internal Report D-22R, December 1977.

16. Scheffer, P. A., and Velez, C. E. "On the Problem of Software Design and Measuring Quality," Proceedings NAECON, May 1978, p. 223.

REFERENCES

17. Schütt, D., "On a Hypergraph Oriented Measure for Applied Computer Science," CompCon Proceedings, September 1977, p. 295.

18. Silver, A. N., "On the Structural Decomposition and Heirarchical Recombination of Non-Directed Linear Graphs ...," Carnegie-Mellon Symposium on Constructive Approaches to Mathematical Models, July 1978.

19. Silver, A. N., "Structural Decomposition using Entropy Metrics", Proceedings of the 1978 conference on Information Sciences and Systems, JHU March 1978.

20. Stevens, W. P., Myers, G. J., and Constantine, L. L. Structural Design, Yourdan, Inc., 1975.

21. Teichroew, D. and Hershey, E. A. III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, SE-3/1, January 1977.

22. TRW, "Software Requirements Engineering Methodology," Report 27332-6291-024, September 1976; Ballistic Missile Defense Advanced Technology Center, Contract DASG60-75-C-0022.

23. Yourdan, E. and Constantine, L. L., Structured Design, Yourdan, Inc., 1975.