Microelectronics and Reliability
Vol. 17, pp. 75-84, 1978

Pergamon Press
Printed in U.S.A.

STRUCTURING SOFTWARE DEVELOPMENT FOR RELIABILITY

by

Sol J. Greenspan
Department of Computer Science
University of Toronto
Toronto, Canada

Clement L. McGowan
SofTech, Inc.
460 Totten Pond Road
Waltham, Massachusetts
USA

Abstract

Software system development is viewed as a series of discrete ordered activities that produce successively more constrained models of the system by binding in additional system aspects. Treating the system aspects as separate concerns allows software engineering techniques that control production cost and enhance reliability to be applied to each step. The greatest gains, however, are due to the reliability and traceability of the system over its lifetime. An essential tool is a structured system description technique.

"To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation... all the time knowing that one is occupying oneself with only one of the aspects... The crucial choice is of course, what aspects to study 'in isolation', how to disentangle the original amorphous knot of obligations, constraints, and goals into a set of 'concerns' that admit a reasonably effective separation." [5]

Introduction

Software development can be viewed as a series of discrete ordered activities that produce successively more constrained models of the system. In this paper we describe software development as consisting of three phases: analysis, design and implementation. Analysis produces a functional model, which describes what the system must do; design produces a specification, which describes how it will be done; implementation produces an operational system. The development of reliable software requires attention during each phase.

Reliability is concerned with the probability of failure of a system and with the impact of failures. For hardware systems, failures are usually due to manufacturing errors or physical phenomena during operation such as wear and tear. In contrast, most software failures are due to analysis and design errors. Reliability must be a priority concern from the very beginning of software development. It has been said that reliability is not an add-on feature, and we concur.

Reliability is related to several other properties of systems: (1) Correctness: satisfaction by the system of the specification, which itself must accurately and completely reflect user requirements; (2) Utility: the extent to which user needs are met when a correct system is used under conditions permitted by the specification; (3) Robustness: the extent to which a correct system's behavior is insensitive to conditions that are not permitted by the specification; (4) Performance: the extent to which the requirements for response time, space usage, accuracy, etc.are satisfied.

The properties are, in a sense, prerequisites for reliability, but they have been considered as subjects in their own rights [6].

Two approaches to reliability are: (1) Fault-tolerance: keeping the system functioning in the presence of errors. Transitions to recovery states are specified in which system operation continues, although possibly at a different speed or capacity, and under different assumptions about, and reliance on, resources; (2) Fault-intolerance: preventing the occurrence of conditions that threaten normal operation. The two approaches are complementary. Both are useful and necessary, but there are always engineering tradeoffs which help determine what can be economically done. The inclusion of measures for reliability is an engineering choice. A very good treatment of this subject is in [9].

A particularly difficult problem is the incorporation of reliability requirements into the specification. This problem can be approached by structuring the design stage in a way that distinguishes and isolates three concerns of system design: structure, algorithm, and implementation properties. By focusing our attention on each of these system aspects individually, specialized software engineering techniques can be applied to each aspect as a separate concern, i.e., techniques for designing structure, specifying algorithms, and representing implementation properties.

SADT™ -- Structured Analysis and Design Technique -- provides a vehicle for structuring (and documenting) the software development process. SADT has been used extensively for analysis and design, and implementations have been derived from SADT documentation in a relatively straightforward manner. SADT abstracts the structure and algorithm aspects of a system and is a framework for expressing further information such as reliability constraints on implementations.

The most important advantage of SADT for software development is the structure its usage imposes on the development process. At every step there exists a description of the state of the system. With each step we associate (a) a methodology for constructing the next model in the series which incorporates the next aspect to be included, and (b) a set of rules for tying-back a model to the previous model.

In what follows, we first examine some reasons for advocating our highly structured approach to development. Next, we describe our three-phased view of software development. Then, a closer look at the design phase shows where the central problems of specifying implementation properties and reliability constraints are. Finally, some information about SADT, the technique proposed as a basis for easing the problems of reliability, is presented. The purpose of the paper is to define a clear framework in which software development problems can be discussed and researched, not to provide solutions to all the problems.

SADT is a trademark of SofTech, Inc.

Life Cycle Awareness

Our collective experience has changed what is perceived as software's principle problems, where emphasis needs to be placed during its development, and how projects should be organized to optimally produce it. Perhaps nothing has motivated changed software practices more than the realization of the true costs of large computer-based systems and where these costs are incurred. Critical activities can be identified and false economies and emphases avoided by considering costs over the entire system lifetime. Four major findings have affected our attitude toward system development.

First, software rather than hardware is the dominant and critical component in computer-based systems. For example, a Rand Corporation study estimated that as of 1972 "at least 70 percent of the current Air Force investment in ADP (Automatic Data Processing) systems goes for software.... [and] software expenditures should rise to over 90 percent of ADP costs by 1985." [8]

Second, within the software realm, maintenance costs exceed development costs for large, usable systems. Boehm reported that "a recent analysis of software activities at General Motors indicated that about 75 percent of GM's software effort goes into maintenance, and that GM is fairly typical in this respect of the industry at large." [3] And DeRoze stated that a "recent DoD [Department of Defense] study showed that Air Force avionics software costs something like $75 per instruction to develop, but the maintenance of the software has shown costs in the range of $4000 per instruction."[4]

Third, errors -- their detection and correction -- account for a major portion of total system cost. Alberts of MITRE recently made the startling but "conservative estimate of almost half of the total life cycle costs (47.6%) being directly tied to error... On the cost basis of a large system [he noted], the total cost of error is in the hundreds of millions." [1]

Fourth, analysis and design errors are, by far, the most costly and crucial types of errors. The TRW Systems Group studied the relative frequency of errors found in five modifications to a large software system. They classified errors as either design or coding errors, where a "design error" was one that required changing the detailed design specification. They found that not only did design errors outweigh coding errors by "64 percent to 36 percent, but also the design errors took the longest by far to detect and correct. Of the 54 percent of the error types typically not caught until acceptance, integration, or delivery testing, only 9 percent were coding errors; the other 45 percent were design errors." [2]

The impact of these facts and of greater life cycle awareness has been truly revolutionary, and it explains why we need to focus our efforts on structuring and controlling the software development process right from the first stages.

The Software Development Process

A simplified view of the software development cycle [11] is shown in Figure 1. The "Analyze" activity yields a "Functional Architecture" which describes what functions the system must possess to satisfy the customer. The word functional connotes that information about possible implementations of

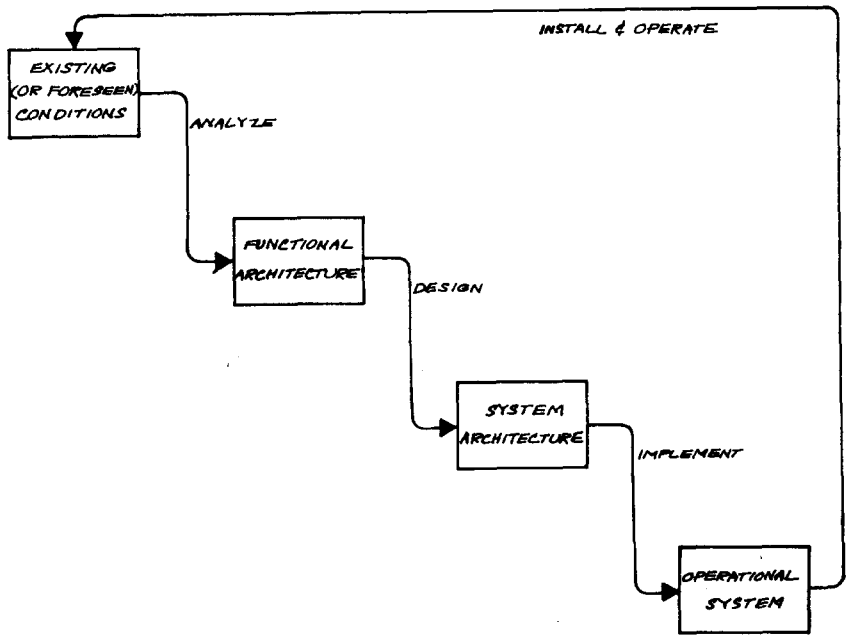


Figure 1: The System Development Cycle

the system does not appear in the document representing this phase of development. We say that we have abstracted the functional architecture and that we have postponed the binding of design and implementation decisions. The way in which we organize the system information at this stage should reflect the structure of the problem.

The "Design" activity, which produces the "System Architecture", determines what the structure of the system will be and what algorithms will perform the required functions. The structure of the system identifies how the whole system has been subdivided into parts and how the parts are interrelated; it should match the structure of the problem as much as possible. The result of the design activity is a specification which serves as a guide for implementation.

The "Implement" activity constructs an actual system in accordance with the specification. What constitutes an "Operational System" depends on what its usage will be. "Operational" could mean executable on certain specified hardware, or it could mean executable on a certain class of machines with high portability.

In order to effectively structure the development process for reliability, a less simplified view must be taken. Each of the three activities shown in Figure 1 is a part of a more extensive phase of software development in which three subjects are treated and documented. The subjects are (1) assumptions: the reasons WHY certain criteria are those that justify the action taken in this phase, (2) description: a description of WHAT the system is at this phase, and (3) constraints: a summary of the constraints on HOW the next

step may be taken. For example, the analysis activity is really part of what has been called "Requirements Definition", which consists of context analysis, functional specification, and design constraints [11]. Similarly, the design activity is not limited to preparing the specification. The design phase consists of:

1. Design constraints -- the reasons WHY the functional architecture causes certain decisions about structure and algorithm to be made and why certain design constraints are the criteria which form the boundary conditions for the system design.

2. Specification -- a description of WHAT the system is to be, in terms of algorithms and interfaces that achieve the functionality of the system architecture. Since this is a design statement, it must give only boundary conditions for the implementation (rather than giving a particular implementation).

3. Implementation constraints -- a summary of conditions specifying HOW the system is to be implemented. Requirements on error recovery, performance (storage, execution time, accuracy, etc.), and other desired behaviors of the system are stated here.

The systematic application of WHY-WHAT-HOW questions structures the system development process. The key to reliability through structured development is to carefully define reliability requirements during requirements definition, then design them into the system, and finally to implement a system that satisfies the design, and thereby satisfies the requirements.

To achieve this purpose, the system description for each phase must be structured, i.e., the division into pieces and the relationships between the pieces should be explicitly documented, and the process for going from one phase to the next must be structured, i.e., the following must be true of our development methodology:

constructible -- methods are needed for deriving code from a specification, and for guiding the writing of a specification from the requirements.

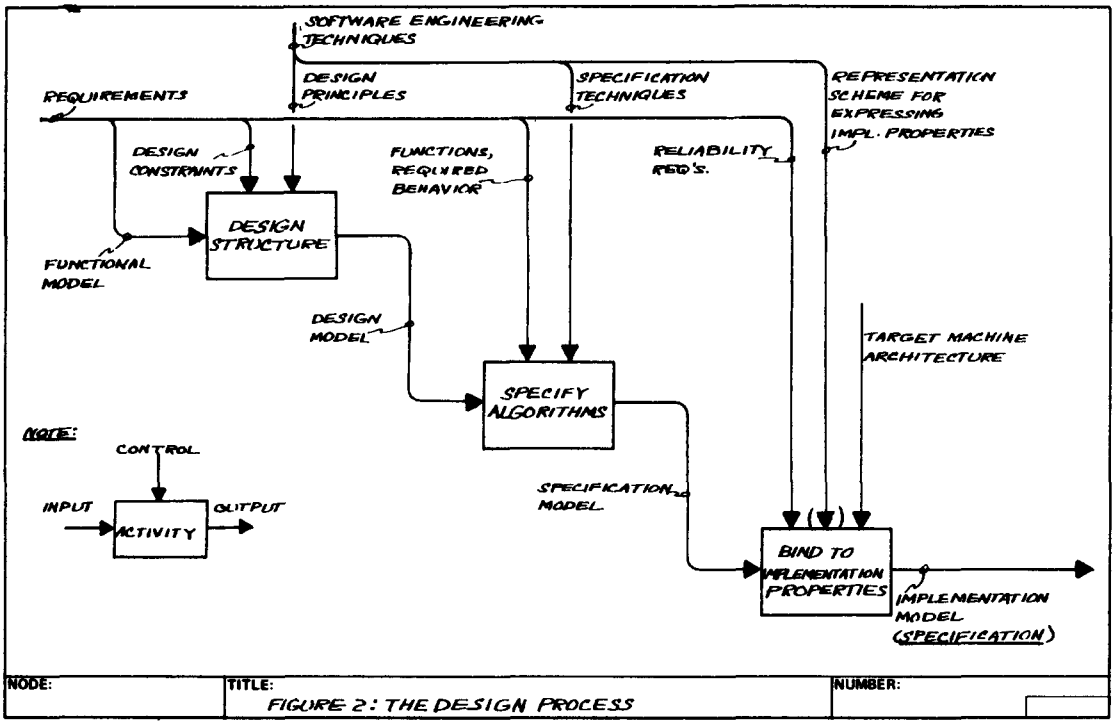
confirmable -- information needed to tie back each phases's description to the previous one should be presented.

A major goal for software, attainable through these properties is:

traceability -- This is the property of a system description technique which allows changes in one of the three system descriptions -- requirements, specification, implementation -- to be traced to the corresponding portions of the other descriptions. The correspondence should be maintained throughout the lifetime of the system.

Structuring the Design Phase

The progression through the three phases of software development was seen to be a process of imposing successively stronger constraints on the contents of the eventual system. At each step, new boundary conditions are established which restrict the class of acceptable systems by binding to new determining aspects. In this section we apply the same structuring idea, in a more detailed view, to the design phase.



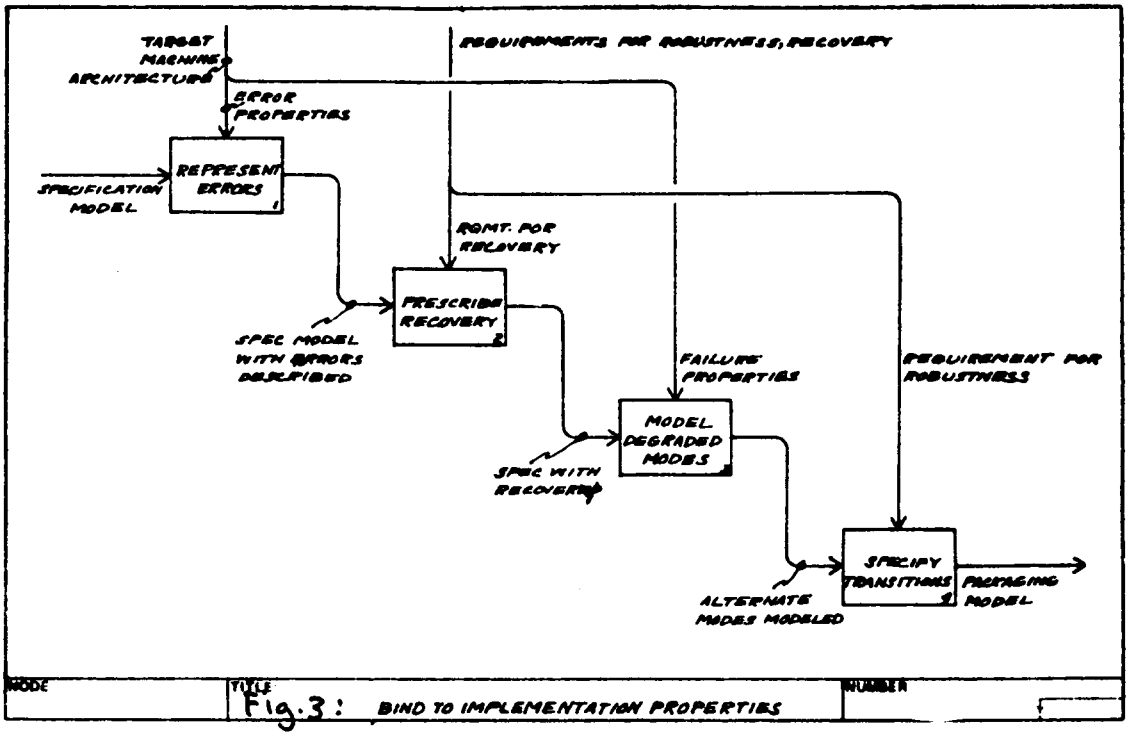
We separate design into three steps, treating separately the aspects of structure, algorithm, and implementation properties. Each step results in a model of the system. The input to the design phase is the Functional Model produced during Requirements Definition. The output from the design phase is an Implementation Model, the description of which is the system specification.

The design phase is described in Figure 2 by an SADT diagram. Although SADT has not been discussed, the ideas communicated by the diagram are virtually self-explanatory. Each box represents an activity and is a part of the design activity. The arrows are all "things" and show how the activities are interrelated. An activity box transforms the input (arrow entering from the left) to output (arrow leaving from the right) under circumstances imposed by control (arrow entering at the top). The unconnected arrows are the interfaces between design and the other phases of software development.

The "Design Structure" step uses design principles to convert the functional model to a design model. The aspect which is bound in this step is structure, i.e., the decomposition of a whole into parts along with the relationships between the parts. Various so-called "structured design" techniques [7] impose structure in characteristic ways. The way that SADT represents structure is exemplified by Figure 2, in which the boxes represent parts and the arrows relationships. The design model is optimized for understandability, modifiability, and reusability.

The "Specify Algorithm" step binds the design structure of modules to an algorithm or procedure. The design model constrained the choice of an algorithm but has not (in general) chosen one. Algorithms can be specified in an appropriate specification language.

The "Bind to Implementation Properties" step is where properties of the target machine and resources are introduced into the specification model. The



properties take the form of additional elements in the model or additional relations among model elements. Some of the properties that need to be expressed are to reflect the reliability requirements: error recovery, machine errors, performance and other requirements on the structure and behavior of the hardware/software system (which may be anything from a microcomputer to a complex avionics system). Implementation properties have been expressed in SADT by transforming diagrams into a notation appropriate for the implementors who build the systems, e.g., PASCAL and Directed Flowgraphs [10].

This third step is of special importance to designing-in reliability, and so we will take a closer look. Figure 3 is a detailed view of Box 3 of Figure 2. Note that the unconnected arrows in Figure 3 correspond exactly with the arrows impinging on the "parent" box, (except for a control arrow entering Box 3 of Figure 2 -- the parentheses about the arrow's head indicates that it has been omitted from the "child" diagram). The first box represents the process of annotating the specification with the detection of errors. The second box involves specifying some action to correct the error at the same level as it was detected. The third box denotes the modelling of degraded modes, entered upon failure, in which the same function is accomplished but perhaps at different speed, capacity, and assumptions about resources. The fourth box is concerned with procedures for moving from normal to degraded modes, (and back again if use of failed resources is regained). Techniques for expressing implementation properties (viz., the nature of the control arrow omitted from Figure 3) must be developed in order to produce reliable software.

SADT -- A Technique for Structuring Software Development

SADT has an integrated methodology for requirements definition and the design and specification of software systems. The methodology is build on the SADT graphical language [12] which structures both the system and the development process.

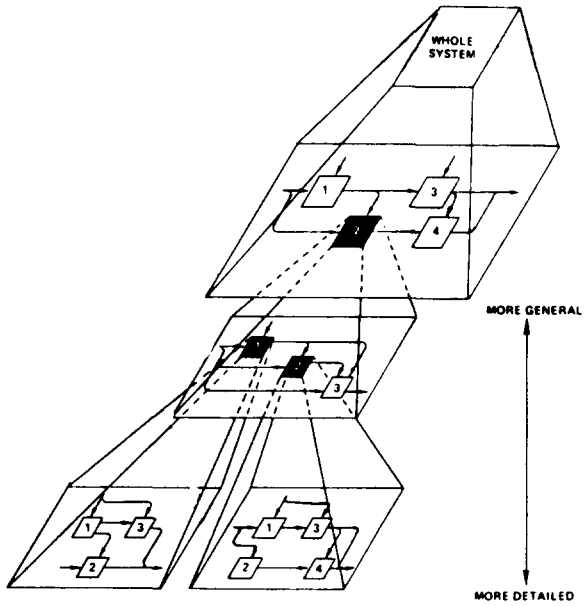


Figure 4: SADT Model

An SADT system description is a hierarchically organized set of diagrams each representing a limited amount of detail. The rules of language usage enforce unfolding a system's structure a piece at a time from the top down (Figure 4). An SADT model is a set of diagrams that describe a system in some bounded context from an identified viewpoint and for a particular purpose. An SADT system description usually consists of multiple models from different viewpoints (e.g., user, manager, builder, etc.). These models are interconnected where details are shared. The process of interconnecting or "tying" models together is a major vehicle for checking the consistency and completeness of the system.

Most important system concepts such as feedback, data flow, sequencing, state transition, and component interconnection can be expressed easily in clear graphic form. Design principles such as top down decomposition, modularity, coupling and cohesion, stepwise refinement, abstract data types, monitors, levels of abstraction, and information hiding are recognizable and expressible in SADT.

SADT was developed initially by Douglas T. Ross during the period 1969-1973. It has been further developed and refined at SofTech as a result of extensive use. Thus far it has been applied successfully to a variety of systems, data base design, military training, policy planning, and financial management. It has been used to design and operating system, a production control system, a data base system used to maintain and modify records containing data on over five million business establishments, and a private automatic branch exchange (PABX) telephonic system. The last project was to produce a detailed software design and implementation specification for a PABX system. It had to accommodate a wide range of installation size (100 lines up to 10,000 lines) over 275 special features. The functional specifications were given using a program language specially tailored for specifying switching programs.

REFERENCES

1. Alberts, D. "The Economics of Software Quality Assurance," Proc. of the National Computer Conference, AFIPS, Vol. 45 (1976) pp. 433-442.

2. Boehm, B., R. McClean, D. Urfrig, "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," Proc. of International Conference on Reliable Software, Los Angeles (April 1975) pp. 105-113.

3. Boehm, B., "Software Engineering Education: Some Industry Needs," in Software Engineering Education, A.I. Wasserman and P. Freeman (eds), Springer-Verlag, New York, 1976.

4. DeRoze, B.C., "Managing the Development of Weapon System Software," presented at the Symposium on Computer Software Engineering, New York (April 1976) (by J.S. Gansler).

5. Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, 1976

6. Goodenough, J.B., "A Survey of Program Testing Issues," to be published in Infotech State of the Art Report on Software Reliability, Infotech Intl., Ltd, Maidenhead, England, 1977; and Research Directions in Software Technology, Wegner, P., and Wulf, W. (eds), MIT Press, Cambridge, Massachusetts. 1977

7. Infotech State of the Art Conference on Structured Design, London (14-16 February 1977)

8. Kosy, D. Air Force Command and Control Information Processing in the 1980s: Trends in Software Technology, Rand Corp. Report R-1012-PR, 1974

9. Myers, G.J., Software Reliability, Wiley and Sons, New York, 1976

10. Rodriguez, J.E., "A Graphic Model for Parallel Computations," Ph.D. Thesis, M.I.T. (September 1969)

11. Ross, D.T., and K.E. Schoman, Jr., "Structured Analysis for Requirements Definition", IEEE Trans. on Software Engineering, Vol. SE-3, No. 1 (January 1977) pp. 6-15

12. Ross, D.T., "Structured Analysis: A Language for Communicating Ideas," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1 (January 1977) pp. 16-33