SOFTWARE REQUIREMENTS AND SPECIFICATIONS:

STATUS AND PERSPECTIVES

by

C. V. Ramamoorthy and H. H. So

Memorandum No. UCB/ERL M78/44

June 1978

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Software Requirements and Specifications: Status and Perspectives

C. V. Ramamoorthy and H. H. So

ABSTRACT

This report surveys the techniques, languages and methodologies that have been and are being investigated for the specification of software throughout all phases of development from the early conception stage to the final detailed design stage. The vast scope of techniques can only be understood by providing a framework so that they can be categorized. We suggest a classification scheme based on the software system life cycle hoping that the putpose, content and requirements of a particular technique can be justified and evaluated. Summary description of significant individual techniques are included to supplement the overall category description.

Besides being an inventory of what has been done, the report is intended to provide a perspective of the area. Within our framework, we hope to spot those aspects and problems that have not been addressed adequately and suggest relevant concepts and ideas that may be used to tackle these problems and solve them, ultimately.

This report is an updated version of the paper titled "A survey of software requirements and specifications" published in the Infotech State-of-the-art Report in Software Engineering Techniques, 1977.

ACKNOWLEDGEMENT

Research sponsored in part by the Ballistic Missile Defense Advanced Technology Center (BMDATC) under contract DASG60-77-C-0138. We would like to thank specially Mr. C. R. Vick for his help throughout the preparation of this report.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

LIST OF TABLES

Software Requirements and Specifications: Status and Perspectives

1. INTRODUCTION

1.1 MOTIVATIONS AND SOFTWARE SPECIFICATION PROBLEMS

Data processing systems, either by themselves or as subsystems of a larger complex of other hardware equipment, are being entrusted more and more complex and critical functions. The cost of software in present and future data processing systems is becoming dominant, which is currently more than 50 percent and is expected to increase rapidly to a much higher percentage in less than a decade. (In some projections, e.g., [Boe 73], this figure reaches 90 percent.) This dramatic proportion indicates that software engineering has not matured proportionally to its hardware counterpart. The main problem of software systems is their complexity. We have never constructed structures with so many parts and with so many interactions among them.

An example of the most complex software system ever built is the SAFEGUARD system developed for defense against intercontinental ballistic missiles [Ste 76, SAF 75].

The total system includes, besides the data processing system, the radars and missiles systems that are controlled by the data processing functions. The system is physically located in different sites of three types: Perimeter Acquisition Radar (PAR), Missile Direction Center (MDC), and Ballistic Missile Defense Center (BMDC). The major functions are surveillance, detection, tracking, and interception of threatening missiles.

These complex functions, compounded with their critical nature, impose a severe strain on the development of the system, particularly the software.

The nature of the system and the hardware introduce additional software functions. The hardware of the data processing subsystem consists of a pool of up to 10 central processors, numerous memory modules, and other peripherals.

Under normal operation, only a subset of these resources is in use; the rest will perform auxiliary functions or just serve as standby units. If system failures are detected, the system must be dynamically reconfigured in a few seconds. Furthermore, to guarantee that the system is ready for any critical function at all times, units are periodically switched off automatically for diagnosis. A very complex software exerciser feeds simulated threats via the radar hardware interface to exercise the critical functions even though no real threats exist in the environment.

The magnitude of the system is reflected by the following statistical data. The system was developed over a span of roughly 90 months. The software consists of 735,000 real-time software instructions, 580,000 support software instructions, and 830,000 installation and maintenance instructions, and was developed at an average productivity of 418 instructions per staff year. (The corresponding figure of 146 instructions per staff year for the real-time software is considerably lower.) The allocation of resources for the whole project was as follows: (1) system engineering, 20 percent; (2) design, 20 percent; (3) code and unit test, 17 percent; and (4) integration activities, 43 percent.

The development project was considered successful in terms of productivity, cost, and time budgets, etc. However, a large number of lessons were learned, and many things could have been done better.

Generally speaking, previous experience with large-scale software development has been depressing. The symptoms of the inadequacy of our software design and development methodology are high costs, unresponsive products, slippage of production schedules, and difficulties in system operation and maintenance.

Although these symptoms are alarming, more specific knowledge is needed to rectify the problems of inadequate software development methodology. A number of studies in the analysis of software errors for both small and large systems have been conducted [End 75, Tha 76]. These analyses usually include a frequency count of various error categories, error causes, and methods for their prevention and detection. Although no specific conclusions are believed to be applicable to all software, design errors account for a high percentage of the total. There is usually a design specification that is an informal document before the implementation of the software modules. An error is classified as a design error if it involves changes in this design specification or a reinterpretation of it. In the absence of other data, design errors are all those made before the implementation process and, therefore, include specification errors. Design errors range from 36 to 74 percent [Tha 76] of the total count. The count alone does not tell the whole story because design errors are more difficult to detect and correct (about 1.5 to 3 times the effort required in dealing with an implementation error [Boe 74]).

Therefore, an investigation of methods to reduce design errors is needed. Before detailed design is possible, there is a period of activity generally referred to as requirements engineering. (No commonly accepted terminology has been used; names such as problem definition and system analysis are used interchangeably.) The activities in this phase are concerned with defining the functional needs, performance, and other requirements. Many problems may develop in this phase; e.g., inconsistent or impractical requirements, ambiguous expression of requirements, incomplete statements, etc. These are carried through and amplified in the design and later stages. For this reason, errors made in the requirements engineering phase will be difficult to fix if not discovered early. There has not been much effort in the determination of the magnitude and seriousness of the requirements problem other than the study of Bell and Thayer [Bel 76c]. Their findings affirm that requirements problems are indeed serious. Traditional means to analyze and state requirements have resulted in unsatisfactory specifications. The following figures indicate the magnitude of the problem: manual examination of two requirements specifications revealed that there were more than 50 problems in a specification document of about 48 double-spaced typewritten pages, and 972 problems in a specification containing 8248 requirements and support paragraphs (2500 pages). The most frequently occurring problems are requirements that are incorrect, incon-

sistent/incompatible, and unclear (refer to Table 1 for subcategories). These account for up to 85 percent of the total.

1.2 IMPORTANCE OF SPECIFICATION METHODOLOGY DEVELOPMENT

The primary objective of a specification methodology (a systematic engineering discipline to state and analyze specifications in software development) is to rectify the above problems. To accomplish this objective, the methodology should provide a means to precisely state the system requirements and provide analytic procedures to check for the consistency, completeness, and correctness of the specification.

Secondary objectives of a specification methodology are to:

- Provide a precise specification for software module implementation, reducing coordination problems among programmers doing the implementation.

- Support unit and integration testing with testing specifications, acceptance criteria specifications, and provide early feasibility demonstrations.

- Reduce maintenance costs by providing monitoring functions when modification and requirement changes are necessary, providing traceability when problems develop during the operational phase, and providing quality assurance without excessive testing after changes are made.

The importance of a software specification methodology and its potential impact on the reliability of future software systems have long been recognized and discussed [Lei 67, Mee 73, Vic 74]. In recent years, extensive work has been done towards establishing a scientifically based engineering discipline to design and produce computer software. A research and implementation project of immense scale is described in [DaV 76], spanning the entire software

Table 1. Listing of Requirement Problems Categories

Note: Taken from [Bel 76c].

- <u>Incorrect Requirements</u>
  - -- Requirement satisfaction probabilistic (under selected conditions)
  - -- Timing requirements not realizable with present techniques
  - -- Requirement not testable
  - -- Accuracy requirement not realizable with present techniques
  - -- Requirement (possibly) not feasible in real-time software
  - -- Required processing inaccurate
  - -- Required processing inefficient
  - -- Required processing produces negligible effect
  - -- Parameter units incorrect
  - -- Equation incorrect
  - -- Required processing not necessary
  - -- Required processing not reflective of tactical hardware
  - -- Requirement overly restrictive/allows no design flexibility (includes requirements stated at too low a level)
  - -- Physical situation to be modeled incorrect
  - -- Required processing illogical/wrong
  - -- Required processing not/not always possible
  - -- Requirement reference incorrect (i.e., other documentation)
  - -- Interpretation of requirement defferent from updated version
  - -- Requirement redundant with other requirement
- <u>Inconsistent/Incompatible Requirements</u>
  - -- Requirement information not same in two locations in Spec.
  - -- Requirement references other paragraphs that do not exist
  - -- Requirement information not compatible with other requirements
  - -- Requirements conventions (e.g., coordinate systems, definitions) not consistent with SDP understanding
- <u>Unclear Requirements</u>
  - -- Terms need definition or requirement needs restatement in other words
  - -- Requirement doesn't make sense

development spectrum from system conception to software process design, with a study of the final testing and validation of the software product being planned.

A successful specification methodology coupled with an advanced design and programming methodology should result in reducing the total cost of software development, operation and maintenance, resulting in a faster release of an operational system. These results are achieved via the following factors:

- A better understanding of the problem statement and its correct interpretation
- Proper specifications resulting in better and more verifiable designs
- Better management of complexity
- Prevention of misunderstanding between system designers, subsystem designers, and data processing designers
- Provision of a "standard" against which the final software product can be tested.

Figures 1 and 2 show a hypothetical comparison of the cost in software systems between the conventional and improved approaches. Considerable simplification has been made to illustrate the following more significant features: (1) the redistribution of time for different phases, with a considerable expansion of the specification and design phases in the improved approach; (2) shorter development time (t_1 versus t_2); (3) total reduction in cost (areas under the curves); and (4) early effort in testing and validation.

1.3 OUTLINE

This section outlines the complete scope of software specification within which the techniques surveyed in this appendix are discussed and evaluated.

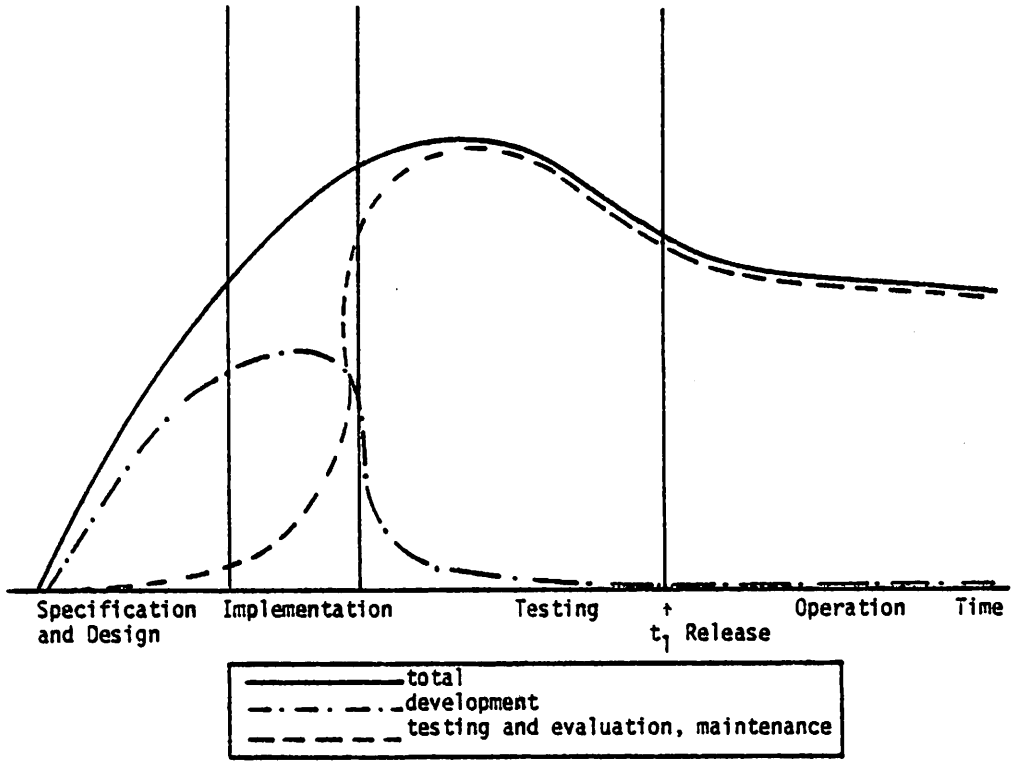


Figure 1. Conventional Software Development Cost Distribution (Hypothetical)

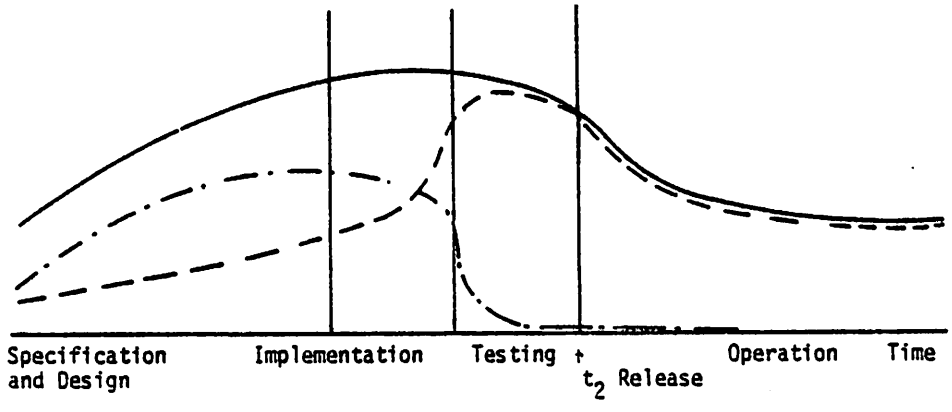


Figure 2. Improved Software Development Cost Distribution (Hypothetical)

We begin with a discussion of the nature of specification in a broader context of engineering design (Section 2). Based on our understanding of the engineering design and problem solving processes, the key principles are used to derive a methodology suitable for use in software system development. We then examine the nature of specification in the methodology. Most importantly, the principal purposes of a specification are: (1) to serve as a problem definition for the design process; (2) to serve as a means against which an implementation can be validated, and (3) to serve as an integral part in engineering a software product in general. Simultaneously, the desirable attributes of a good specification method readily surface from the discussion.

In Section 3, an outline of a model of software development is described for orientation purposes and the uniform use of terminology. This model provides a concrete background for later discussions when we try to justify why certain features and techniques are chosen and what is still missing in the state of the art in software specification.

Section 4 develops a central framework to organize the issues to be discussed and to classify the techniques surveyed. Section 5 contains a catalog of specification techniques that are referenced in the earlier section.

2. AN ENGINEERING DESIGN PARADIGM

There are extensive studies on the nature of design and design methods [Jon 63, Spi 74]. Because we cannot cover all the important facets here, a design paradigm is suggested to ser as a frame of reference for our later discussions on the specification techniq es for software system development. Here we try to outline the role of specification in a design and development project, and introduce the attributes of desirable specifications.

The general design context is the presence of a difficulty or something needed. Given the context of the problem, a solution (the design) has to be proposed. We can view this as a transformation

$$P(\text{needs}) = \text{product}.$$

In a large and complex design situation, different phases are gone through and the transformation of the needs to the final product takes on a number of distinctly recognizable forms:

$$P_{phase\text{-}n}[\ldots(P_{phase\text{-}2}(P_{phase\text{-}1}(\text{needs})))\ldots] = \text{product},$$

or alternatively:

$$\begin{aligned}
&\text{Needs} = Form_0\\
&P_{phase\text{-}1}(Form_0) = Form_1\\
&P_{phase\text{-}2}(Form_1) = Form_2\\
&\quad\vdots\\
&P_{phase\text{-}n}(Form_{n-1}) = Form_n\\
&Form_n = \text{product}.
\end{aligned}$$

The neutral terms $P_{phase\text{-}i}$ and $Form_i$ are used here to avoid specific terminology dependent on a particular design context. There can be an arbitrary number of phases.

For each equation, $P_{phase\text{-}i}(Form_{i-1}) = Form_i$, we regard $Form_{i-1}$ as the requirements or specifications for $P_{phase\text{-}i}$; and $Form_i$, the specification for $P_{phase\text{-}i+1}$; etc. Besides being purely a definition, we also intend the usual connotation of the terms--specification and requirements. Each phase of the development process, $P_{phase\text{-}i}$, is supposed to develop or elaborate on $Form_{i-1}$, given as requirements. $Form_i$, being the product of $P_{phase\text{-}i}$, must have the properties that satisfy all the requirements of $Form_{i-1}$. $Form_i$ is said to be validated against $Form_{i-1}$:

$$\text{Validate}\ (Form_i,\ Form_{i-1}) = \{\text{OK, not-OK}\}.$$

The value OK is taken provided that if the specifications of $Form_i$ are ultimately satisfied, then the $Form_{i-1}$ requirements will be satisfied.

The ideal and simplest design methodology would be:

<u>Methodology-1</u>

$Form_0 \leftarrow$ needs;

<u>Do</u> $Form_1 \leftarrow P_{phase-1}(Form_0)$ <u>until</u> Validate $(Form_1, Form_0)$. = OK

.

.

.

<u>Do</u> $Form_n \leftarrow P_{phase-n}(Form_{n-1})$ <u>until</u> Validate $(Form_n, Form_{n-1})$ = OK

Product $\leftarrow Form_n$.

However, this implies that at any phase, given any $Form_{i-1}$, $P_{phase-i}$ is able to produce $Form_i$. This assumption does not hold in most real situations. $Form_{i-1}$ may be infeasible, i.e., no final product would meet all of those requirements. The development process will abort or we can go back to a previous phase, $P_{phase-i-1}$, and generate another $Form_{i-1}$, so that Validate $(Form_{i-1}, Form_{i-2})$ is still OK and $Form_{i-1}$ is feasible.

The design methodology would be modified as follows:

<u>Methodology-2</u>

$Form_0 \leftarrow$ needs;

<u>Do</u> $Form_1 \leftarrow P_{phase-1}(Form_0)$ <u>until</u> $\left\{\begin{array}{l}\text{Validate}(Form_1, Form_0) = \text{OK } \underline{\text{or}} \\ Form_0 \text{ is found infeasible}\end{array}\right\}$;

<u>If</u> $Form_0$ is found infeasible, <u>go back</u> one phase;

.

.

.

.

.

.

<u>Do</u> $Form_n \leftarrow P_{phase-n}(Form_{n-1})$ <u>until</u> $\left\{\begin{array}{l}\text{Validate}(Form_n, Form_{n-1}) = \text{OK } \underline{\text{or}} \\ Form_{n-1} \text{ is found infeasible}\end{array}\right\}$;

<u>If</u> $Form_{n-1}$ is found infeasible, <u>go back</u> one phase;

Product $\leftarrow Form_n$.

This becomes a backtracking procedure and may not terminate. A more desirable approach, which unfortunately may not be possible at all, is to establish the feasibility of the output form at the end of each phase.

<u>Methodology-3</u>

$Form_0 \leftarrow$ needs;

Modify $Form_0$ <u>until</u> Feasible $(Form_0)$ = <u>True</u>.

<u>Do</u> $Form_1 \leftarrow P_{phase-1}(Form_0)$ <u>until</u> $\left\{\begin{array}{l}\text{Validate}(Form_1, Form_0) = \text{OK } \underline{\text{and}} \\ \text{Feasible}(Form_1) = \underline{\text{True}}\end{array}\right\}$;

.

.

.

<u>Do</u> $Form_n \leftarrow P_{phase-n}(Form_{n-1})$ <u>until</u> $\left\{\begin{array}{l}\text{Validate}(Form_n, Form_{n-1}) = \text{OK } \underline{\text{and}} \\ \text{Feasible}(Form_n) = \underline{\text{True}}\end{array}\right\}$;

Product $\leftarrow Form_n$.

The above methodology is a "straight-line program," and there is no backtracking involved. In practice, we expect a modified version of Methodology-2, in which the feasibility of the output form is not established absolutely, but any factors that may contribute to infeasibility (e.g., logical inconsistency)

are checked for as far as possible. A reduced amount of backtracking may be achieved.

Within this paradigm, several fundamental issues can be observed and are discussed below.

2.1 MULTIPHASED DEVELOPMENT

We may first ask why the whole design process is divided into phases. The main reason possibly is that there are different thought processes involved in a design corresponding to the different phases. This phenomenon is more pronounced in a large complex project requiring a long development time. The factors to be considered, the level of abstraction, the types of entities, the classes of decision and the tradeoff problems change gradually from the beginning to the end, resulting in a significant difference across the whole spectrum. The usual pattern is from general to specific, abstract to concrete, and aggregate to detailed. In other words, this phenomenon is the result of the necessity for different words; we identify this phenomenon for different groups of people with different expertise who employ a variety of different techniques and tools (e.g., application area experts, computer specialists, and programmers) at different times of the development process to cooperate with one another. This differentiation in thought process and expertise requirement has direct bearing on the language to be used for the different forms.

2.2 LANGUAGES FOR EXPRESSING THE FORMS

The two basic requirements for the language to express a given form are constructability and comprehensibility. The former allows the user to use the tool (the language) efficiently for expression, and the latter, for communication. These factors are closely related.

Apart from the linguistic issues of whether a language is theoretically capable of expressing a certain class of objects, the more immediate issue with regard to constructability is the ease of use. As pointed out in [Har 74],

the regular expressions are capable of specifying the strings recognizable by all finite automata. However, it would be quite taxing to express the finite automaton that accepts the set of binary strings with no two consecutive 1's in a regular expression [which turns out to be $(0 + 10)^* (\Lambda + 1)$, not an immediately obvious result]. The language would be useless if the user has to twist his thought to use it. The argument for comprehensibility is even more evident if effective communication among people should be provided.

Because of differences among the development phases, a single language is not adequate to express all forms. For example, take a programming language such as FORTRAN and consider how to use it to express system requirements such as the following: the system should be flexible so that an increase in external loading does not require excessive redesign of the whole system. An inevitable conclusion is that language design is very difficult because the language must be finely tuned to the particular development phase in which it is intended to be used.

2.3 VALIDATION

In our paradigm, a validation step is associated with every development phase. To support this feature, the forms must be formal and testable. By formal, we mean that a precise notation with unique interpretation of each expression is used. This requirement is mandatory if our validations are to have logical validity. The form is testable if the truth value of each requirement in it is effectively decidable, i.e., we must be able to show that each requirement in $Form_i$ is satisfied by some combination of the specifications in $Form_{i+1}$. We have just noted that it is not necessary to have a one-to-one correspondence between two forms, as is almost the case when they are expressed in two different languages. However, the traceability of each requirement must be maintained. An implication of the last condition is that we should be able to tell, for each requirement in $Form_i$, how it is satisfied in $Form_{i+1}$ and lower levels, and conversely, which requirements in $Form_{i-1}$ and above have direct bearing on it.

The strongest form of validation is to establish the logical implication between two forms. In practice, a much weaker validation is actually carried out because of a combination of the following reasons: (1) the lack of formalism in the forms so that a vigorous validation is impossible (lack of testability); (2) the complexity of the validation techniques (if they exist), especially for large-scale projects; and (3) the nonexistence of the validation techniques.

The actual validation may be as follows (in order of increasing strength):

- Adherence to prescribed standards, usually in document forms.
- Certain properties in the forms, e.g., consistency, completeness. These properties, however, are variously formulated in different contexts.
- Presence of certain invariant properties across two forms.

As has been indicated above, none of these can be accepted as "validation" in the strict sense of the word. They are, in fact, only partial validations.

The notion of consistent decomposition, which can be regarded as a special form of validation, should be introduced here. Let us for a moment call the process of obtaining $Form_{i+1}$ from $Form_i$ decomposition. They have been expressed in two different languages, or the same language in different amounts of detail. Suppose a process, called reduction, is able to homomorphically (structure preserving in some sense) transform the decomposed $Form_i$ back to its own language or level of detail (Figure 3); the symmetric difference* of $Form_i$ and $Form'_{i+1}$ reveals the anomalies of the development. In Figure 3, $Form'_{i+1} - Form_i$ stands for those features not specified in the higher level (e.g., additional requirements and assumptions); and $Form_i - Form'_{i+1}$ stands for the part of the original specification not satisfied by the decomposed version.

*The symmetric difference of two sets A and B is $(A-B)\cup(B-A)$.

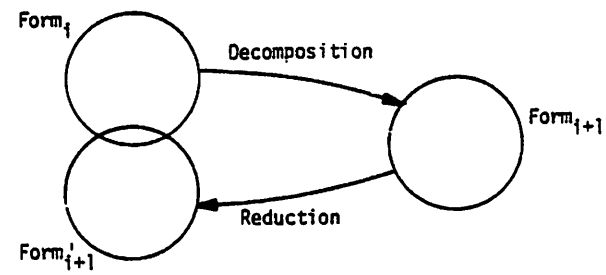


Figure 3. Validation of Decomposition

2.4 FEASIBILITY

Another element of the paradigm, demonstration of feasibility, is introduced to reduce the amount of backtracking in the development process. To be able to show that a given form (containing a set of specifications) is feasible, of course, is certainly very desirable. However, a technique must be developed to demonstrate the feasibility of a form specified in a language. We do not know of any general method with subsequently strong practical support. In most situations, feasibility demonstration is done by prototype testing, simulation, and a combination of the two.

3. SOFTWARE SYSTEM LIFE CYCLE

Institutionalized software development projects are usually divided into phases similar to those shown in Figure 4. We describe a typical and somewhat simplified software life cycle below to orient the later use of terminology.

The least disputable aspect of the cycle is that it begins with the conception of some needs to be fulfilled, and there are feelings and indications that a significant portion of the solution is using the information processing power of modern computer systems. It undergoes the process of development, and then the operational product is used in the actual environment, simultaneously modified, and adapted to new needs.

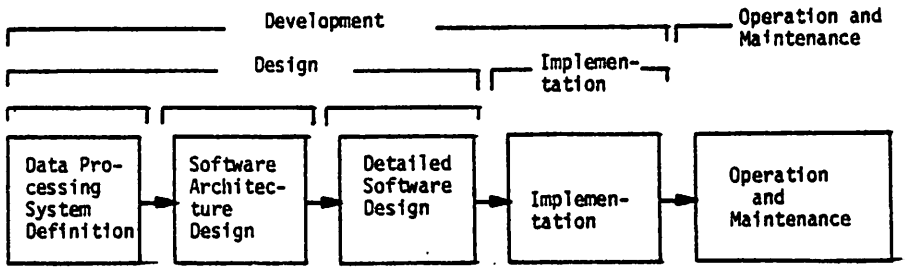


Figure 4. Software System Life Cycle

The development phase is subdivided into design and implementation. The end of design is generally regarded as the end of the more creative activities, and the rest (implementation) is a relatively straightforward procedure. In software development, a complete design is the definition of all program modules on a specific data processor, plus the support documentation.

The design process is further partitioned into three subphases of Data Processing System Definition, Software Architecture Design, and Detailed Software Design. The key phases and forms in the design paradigm are identified in Figure 5.

This figures shows only the essence of the process. In the development of a large system, the process can be much more complicated. There will be inevitable feedback from one phase bac to previous phases, when errors and other difficulties are discovered. Anc :her deviation from the simple scheme is in the incremental development approach for some systems. (A part of the system is built to the operational stages, and then some other functions and performance capabilities are added onto the existing operational parts.) This can be viewed as a series of the above basic process.

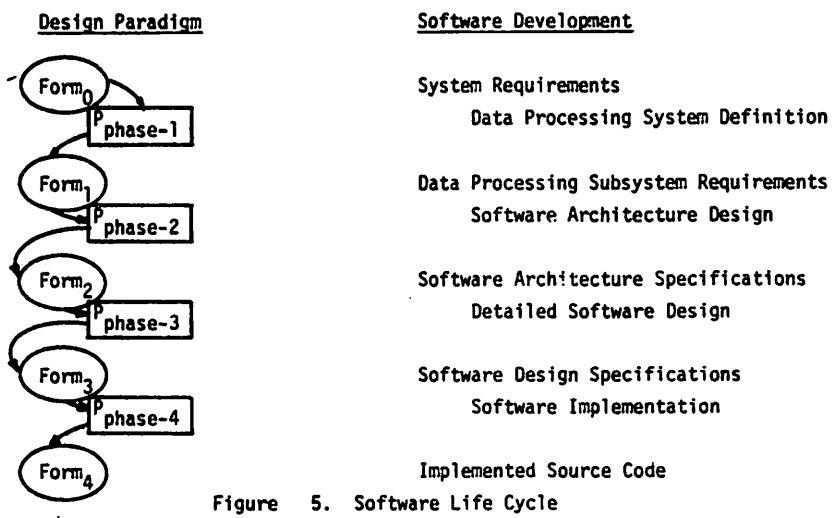


Figure 5. Software Life Cycle

The following paragraphs discuss in more detail the major activities and design decision in each of the development phases.

3.1 DATA PROCESSING SYSTEM DEFINITION

Data Processing System Definition consists of analyzing and decomposing the system needs and objectives into system engineering terms. System objectives can be very vague, and the fulfillment may not be objectively verifiable. For instance, to improve national defense can be the overall objective of a missile defense system, and to increase corporate profitability can be the objective of the real-time management information system of a manufacturing firm. The objectives must be decomposed, elaborated, and stated in precise terms. Furthermore, in most system developments we are considering, the data processing functions only constitute a subsystem of a larger complex involving other hardware facilities. In these situations, the total system requirements must be analyzed and part of them allocated to the data processing subsystem. The minimal activities to be performed are: (1) identification of all the entities

interacting between the system and its environment; (2) statements of the expected functional responses to system stimuli; (3) identification of system performance parameters and the expected system performance in terms of these parameters (e.g., response time, cost); and (4) specific characteristics of the system (e.g., flexibility) formulated formally in terms of the system elements.

3.2 SOFTWARE ARCHITECTURE DESIGN AND DETAILED SOFTWARE DESIGN

Software system architecture design has much in common with the later phase of detailed software design. The general design process, as well as software system design, has been the focus of much research. An extensive discussion of the subject is beyond our scope here. We know of no better method to design and develop large systems other than the divide-and-conquer and hierarchical structuring approaches (these include the top-down design, stepwise refinement, abstract machine layers, etc.). The most fundamental ideas appear in [Dij 68, Bri 70, Pen 72b]. Thus in the process hierarchy approach, a system is organized as a sequence of hierarchical levels of processes. In each level we observe a group of interacting processes--each of which is accomplished by yet another group at a lower level (Figure 6).

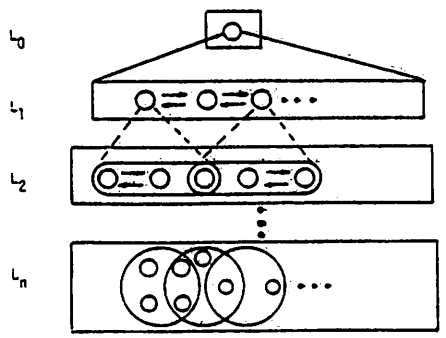


Figure 6. Hierarchical Design Approach

In these design approaches, the system is designed using a set of "subentities" whose behaviors and properties are specified and their detail is to be developed in a later stage. The specifications of these "subentities" should, therefore, include all the properties necessary for establishing the behavior and properties at the current level and only those properties necessary without restricting their construction at the later levels.

Another important concept that has been developed recently in the area of system design is the family of systems approach. The essential concept is that when we are developing a system conceptually in a hierarchy of levels, we are starting out with a broad class of systems, and as we go along and make design decisions, the class of systems is limited to meet more specific requirements. If a system is designed with this philosophy, requirements changes and, in general, system modifications can be accommodated much more easily by zeroing in the proper level at which a design decision must be changed. A new branch on the system family is developed to meet the new or additional requirements.

The distinction we make between the software architecture design and the detailed software design is that software architecture design is more concerned with the logic of the problem and, among other things, is predominantly hardware independent; whereas software design is concerned with the design decisions of realizing the system on a specific data processor. The major design decisions in software architecture design are: (1) identification of the functions the system must provide to respond to the environmental stimuli; (2) specification of the performance requirement of each system function; (3) identification of the major subsystems or subfunctions, information sets, and their interactions and coordinations; and (4) specification of requirement for reliability, maintainability, availability, etc. The major design decisions in software design, for comparison, are: (1) choosing the algorithms for implementing the system functions conforming to the functional and performance specifications; (2) choosing the major data structures (logical rather than implementational); and (3) designing operating system functions and scheduling of the functions (processes) on a specific computing system or systems.

The principles of hierarchy and abstractions apply also to the detailed software design level. Much of the effort of the software designer is occupied by the effort to coordinate the many activities in the system under stringent time constraints. The general problem is the correct and efficient coordination and synchronization of parallel processes. Most of the synchronization traditionally has been done by a general-purpose operating system. Because of efficiency considerations and the desire for decentralized control, more and more of these functions have to be performed at the application software level. System design methodologies involving parallel processes are being intensively investigated; an example can be found in [Bri 77].

3.3 SOFTWARE IMPLEMENTATION

This phase is the coding step, normally done in high-level programming languages because of its demonstrated superiority over assembly languages [Bro 75]. More and more efficient languages for the programmer are being developed, supporting structured code and data abstractions. The programming phase is actually beyond the realm of discussion in this appendix; therefore, we are not going to explore further into its issues.

3.4 OPERATION AND MAINTENANCE

System operation and maintenance refers to the phase of the life cycle after the initial release of the system. Requirements and the external environment of the system continuously change even after the system is operational. Furthermore, errors are found so that the original system must be changed. There has not been much atten on paid to this phase of the life cycle; consequently, most of the system m lifications are performed by ad hoc approaches. These have been characterized as design, implementation, and testing all over again. In principle, during the requirement definition and specification phases of the development stage, further modifications should be anticipated. In all the techniques surveyed, however, this aspect has not been addressed adequately.

4. SPECIFICATION TECHNIQUES CLASSIFICATION AND EVALUATION

This section presents a framework within which the specification techniques that have been investigated can be categorized and evaluated. The "technique" we refer to is generally a language or a scheme to express certain aspects of the system under development (e.g., specifications, requirements), and the associated tools, support, and logistics for analyzing them. Occasionally, some techniques surveyed have a much wider scope. Under those circumstances, we extract their most significant features and discuss them along with others at the proper places.

The mainframe of our classification is the design paradigm outlined in Section 2.2. We broadly classify all the techniques according to the form throughout the system development process with which it is best suited to be expressed. Thus we classify them as techniques for System Requirements, Data Processing Subsystem Requirements, Software Architecture Specifications, and Software Design Specifications. Within each category, subcategories are identified based on some of their characteristic features; they are then briefly described and evaluated as a group. References to the specific techniques, which are described in Section 5, are made whenever appropriate.

4.1 SPECIFICATION TECHNIQUES FOR SYSTEM REQUIREMENTS (OVERALL NEEDS AND OBJECTIVES)

As indicated in Section 3.1, system requirements, needs, and objectives are generally vague and ambiguous, chiefly because they are at the top-level and arise directly from the application area problems. In practice, system requirements may also include the whole design context of the system, such as environmental design constraints and criteria for evaluating alternative designs. This information is stated in free form English and sometimes not explicitly stated at all.

Formalization is most difficult at this level. It requires complete codification of all knowledge relevant to the design of the system. However, we may hope to be able to formalize certain important aspects. Research work of special relevance here is in semantic nets and fuzzy systems.

Semantic nets studies [Sim 73] are mostly within the area of artificial intelligence. The main objective is to represent in the computer a body of knowledge so that computer retrieval and manipulation of this knowlegde is possible. Information Automat [Nil 75] is the only example initiated specifically for application in system requirements analysis. The method is still at a relatively early research stage. We expect it is best applied in small- to medium-scale system projects.

Fuzzy concepts and systems, pioneered by Zadeh [Zad 71], have now grown into a large body of knowledge. The chief objective of fuzzy concepts is to precisely formulate imprecise notions and relations that are characteristic of all large and complex systems. Although no specific work has been done on software system requirements analysis using this body of knowledge, we do recognize that this is a new avenue worthwhile exploring. A design methodology suggested by Becker [Bec 73] relies quite heavily on fuzzy concepts. (Becker discusses more of the philosophy than the methodology itself.) This seems to be useful in relatively small-scale system designs. The methodology is developed within the context of structural design in civil engineering. However, it is discussed in general design terminology so that its usefulness in software system design can be assessed without much difficulty and give a favorable impression.

4.2 SPECIFICATION TECHNIQUES FOR DATA PROCESSING SUBSYSTEM REQUIREMENTS

The form and content, and hence the language, for data processing system requirements are the least agreed upon aspects compared to that of software architecture specification and software design specification. The requirements must serve two purposes. First, the user must be able to tell from the requirements whether he will accept the system and make other similar decisions such as evaluating certain specified properties of the system. Second, the system designer must be able to develop a set of software architecture specifications from the requirements and demonstrate that if a design is created satisfying the architecture specification, then the data processing system requirements are satisfied. The basic requirement is that both the user and the system designer must be able to understand the document and perform meaningful evaluation. We can narrow down the concepts and terminology in which the data processing system requirements can be stated. These must be in the overlap of the area of competence of the application-area experts and the system designer--system concepts.

Several methods have been investigated and/or used, ranging from the unstructured form used to state the system objectives and requirements discussed in Section 4.1 to highly structured forms used to describe specific properties of the target system. The formal methods will now be discussed.

To tie together the numerous methods and techniques, the notion of a general model of a large-scale, real-time system (emphasizing the data processing functions) is developed.

The most general view of any system is to take it as a relation [Mes 72]:

$$S \subset X \times Y.$$

We restrict the class of systems to functional systems to simplify the notation (without loss of generality). Thus every system is characterized by a function

$$S: X \to Y$$

where X is the set of inputs and Y is the set of outputs. To model dynamic systems, we interpret X and Y as functions of real time, T, i.e., patterns of inputs and outputs (or input and output trajectories in the terminology of Wymore [Wym 76]).

$$X: T \to \mathbf{X}$$
$$Y: T \to \mathbf{Y}$$

An example may make the abstract notation introduced so far more concrete.

Suppose system S has only one input X and one output, Y. Figure 7 shows a partial specification of S:

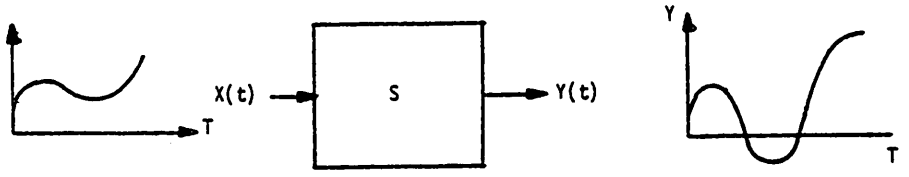


Figure 7. Functional Specification Example

A complete specification of S involves specifying all possible X(t) and the corresponding Y(t).

Performance requirements and other special attributes of the system can be specified by identifying the vector of performance parameters, W. The parameters may be implicit functions of X and Y. Thus the complete system requirements become:

$$S: X \rightarrow Y \times W.$$

There have been a number of studies conducted for methods of stating data processing system requirements (e.g., SREM [Alf 76a,b], ISDOS [Tei 74b]). None of them is satisfactory for application in really large-scale, real-time systems development. We clas fy the requirements statement methods into three broad categories: (1) .nctional requirements; (2) performance requirements; and (3) specific attributes, with further subcategories wherever appropriate. Significant techniques in each subcategory are introduced and assessed. However, both the classification and assessment are somewhat subjective.

4.2.1 Functional Requirements

Statements of functional requirements are specifications of the function $S: X \rightarrow Y$. Informally, it states the response the system should produce in reaction to a given stimulus. We identify three subclasses of methods.

4.2.1.1 Explicit Specifications of S

These methods usually employ rigorous mathematical statements. The most straightforward way is to define the domain and range of S, i.e., input and output spaces X and Y explicitly, and exhibit the function S by tabulation or a set of mathematical expressions. The methodology introduced by Wymore [Wym 76] in his book on Systems Engineering adopts this approach. The method is, of course, limited by the ability of the user to state the function, S. In any real-world system, the complete specification of S is likely to be impossible. However, an explicit statement in such a rigorous form is most readily testable. Hence, formulating the critical functions of a system in this way to the extent possible may be of significant value for system level function validation.

Because of the difficulty in stating the complete function, another approach is taken by Fitzwater [Fit 76] and Hamilton and Zeldin [Ham 76a]. The system function S is stated by a series of decompositions according to some decomposition rules. Thus S is composed of $S_1, \ldots, S_n$, and S_1 is composed of $S_{11}, \ldots, S_{1m}$, etc. The ability of the user to formulate his problem in the associated language is different, yet the rigor of precise mathematical statements permits certain properties of the function at various stages of decomposition to be checked. Apart from being less formal, the Structured Analysis and Design Technique (SADT) [Ross 77] and a few others (Section 5.1) can be included in this category. These latter ones usually employ a graphical notation for better human communication with less emphasis on rigorous formalism.

There can be certain reservations in classifying the decomposition approach in the explicit specification category. The decomposition represents certain design decisions (nonarbitrary restrictions); the later design steps may be severely dictated by these early choices. We have made an assumption that design freedom should be maintained as far as possible. With regard to this, the function decomposition approach may as well be classified in the following category of implict specifications of S.

4.2.1.2 Implicit Specification of S

Methods in this category define the function S by exhibiting a procedure or structure to show how a certain output response of the system is generated in response to system stimuli and inputs. This can be done by a definition of the subsystem structures, functional partitions, etc. Generally speaking, it is more appropriate to regard this as the software architecture specification. A detailed discussion of this large category of methods is deferred to Section 4.3.

A comment as to the suitability for requirement statements, however, is appropriate here. Specifying a system structure as system functional requirements may be regarded as a limitation of design freedom. On the other hand, we may consider whether the limitation is unnecessarily constraining or not, and whether the nature of the application problem naturally suggests a particular system structure, and at what level of detail the system structure is specified as functional requirements.

A stronger argument for accepting them as appropriate specification techniques at this level is the recognition of a class of designs as "wicked problems," and software system design is argued to be wicked [Pet 76a]. The term "wicked problem" was coined by Rittel [Rit 72], referring to design situations having the following characteristics (among others):

- Every formulation of the wicked problem corresponds to the formulation of the solution (and vice versa). The information needed to understand the problem is determined by one's idea or plan of a solution. In other words, whenever a wicked problem is formulated, there already must be a solution in mind.

- Wicked problems have no stopping rule. Any time a solution is formulated, it could be improved or worked on more. One can stop only because one has run out of resources, patience, etc.

- No wicked problem and no solution to it has a definitive test. In other words, any time a test is "successfully" passed, it is still possible that the solution will fail in some other respect.

If indeed we are dealing with wicked problems, specifying S (the problem) by exhibiting its structure (a solution) is the only means. Our position is that we should not consider "wicked" as a boolean attribute; different problems have different degrees of wickedness. It appears that in many cases, software system development is very wicked so that implicit specification of S is necessary in certain aspects but should be employed with careful considerations.

4.2.1.3 Optimization Model Formulation of S

In many cases we do not have a functional requirement in the sense that given an input X, a specific Y should be produced. Instead, a large number of outputs would be acceptable, except that we prefer one to the others.

This type of situation is best formulated as an optimization problem. An example of system requirements specification of this form is:

- The set of alternatives, M, is specified (or to be investigated).

- The outcome function, C, specifying what would be the system behavior given an alternative

  $$C: X \times M \rightarrow Y.$$

- A valuation function, V, specifying how the combination of a certain choice and behavior (output) is valued

  $$V: M \times Y \rightarrow R.$$

Then the system functional requirements S: $X \to Y$ is specified by $S(x) = y = C(m,x)$ if $\forall m_i \in M$, $V[m,C(x,m)] \leq V[m_i,C(x,m_i)]$.

Other variations of this formulation are possible [Mes 72].

The above example is in a very crude form because there is no investigation of this nature within the discipline of software engineering. Nonetheless, fruitful results may be obtained with some investigations. Another remark applicable here is that the performance requirements of the system are even more naturally formulated in this class of models (this will not be repeated in the following paragraphs).

4.2.2 Performance Requirements

Performance requirements need additional information for the complete specification of S: $X \to Y \times W$, where W is the vector of the performance parameters.

The most difficult part of the requirements statement is the definition of all the relevant performance parameters. There is essentially no notion of completeness as far as performance requirements are concerned. Functional requirements can be regarded as complete if S: $X \to Y$ is defined for every possible $x \in X$. However, W is essentially an open-ended list of parameters. For a chemical process plant control system, the pollution level can be a performance parameter. Leaving it out in the requirement specification does not constitute logical incompleteness unless it is already recognized that the factor is relevant.

In practice, there is currently no formal method of stating performance requirements except in free-form English descriptions. The most important performance requirements seem to have been timing (response time, deadline specification, etc.) and loading factor (the amount, frequency, and distribution of inputs and outputs).

Methodologies that include special provisions to state (partial) performance requirements are the System Requirements Engineering Methodology (SREM) of BMD [Bel 76b], ISDOS [Tei 74a], ADS [Syn 69], etc. In SREM, the performance requirements are stated by specifying "validation points" in the R-nets and the performance information to be collected at these validation points. Performance requirements specification in other methods are more or less similar, including the specification of the frequency and distribution of certain data, the deadline for certain outputs, etc.

As an overall remark, performance requirements specification is another area that has not been addressed adequately in the past work.

4.2.3 Specification of Special System Attributes

A typical requirement statement regarding system flexibility may be as follows: "System growth against evolving external load shall be achieved without major subsystem redesign, without major impact on other subsystem components, and without extensive system inoperability."

The above statement exhibits both the ambiguity ('major', 'extensive') and untestability of requirements statements. Attributes such as flexibility, security, relability, etc., must be formulated precisely for meaningful analysis; otherwise, the decision as to whether a system has these attributes will be highly judgmental. Work in this direction of formalizing special system attributes is now emerging, mostly originated from operating systems research. Notions such as deadlock free, security, and integrity [Neu 76] have been investigated. It is not yet certain whether this can be applied in a more general context.

4.3 SPECIFICATION TECHNIQUES FOR SOFTWARE ARCHITECTURE

Currently, there is not a generally accepted theory of what should be considered as the basic concepts and objects of a system and how they should be organized as a "structure." In any event, the software architecture specifications should be an implicit definition of the system function S as introduced in

Section 4.2. Furthermore, a method (or language) for the specification of large-scale software architecture at the system level should reflect the way a system is conceived by the requirement analysts.

In principle, the formal algorithm specification of both sequential and parallel processes, and data abstraction to be discussed in Section 4.4 may be applied at this level also. However, the current state of development of these techniques is considered as impractical at the system level.

Other software architecture specification techniques center on two dominant features of a real-time system: the data flow and the control flow. In some systems, the flow of data is so dominant that having a clear picture of the flow and interrelation of the system elements almost completely tells us all about the system. In others, with intricate synchronization and timing, the control aspect plays the more important role. Many systems, of course, consist of both aspects. Techniques for specifying the data flow type of systems are considerably more developed and numerous than the control flow type, although increasing efforts are being directed to the control flow type as real time, control, and planning systems are becoming more complex and critical. Discussions of the software architecture techniques that have been and are being investigated are divided into two categories dealing separately with the two types of systems.

We must, however, emphasize that the classification should be considered as loose, because many techniques address both features together. Features other than the above two (though not justifying separate categories) may also be included in some techniques.

4.3.1 Methods for Specifying Data Dominant Systems

The typical data flow dominant or data driven systems are business oriented information processing systems. In this type of system, a set of output documents is required to be produced at specific times or at regular intervals. The main concern of these systems is how these output documents are derived

from the input data, and internally, what data are to be retained for later use, etc. Once the process of getting the output from the input is specified, the structure and the rest of the system become apparent. An example is a simple payroll system. The required output is a weekly paycheck to every employee, and the inputs are daily work records and new employee records. The data flow requirements are shown in Figure 8. The control aspect follows naturally from the data flow requirements.

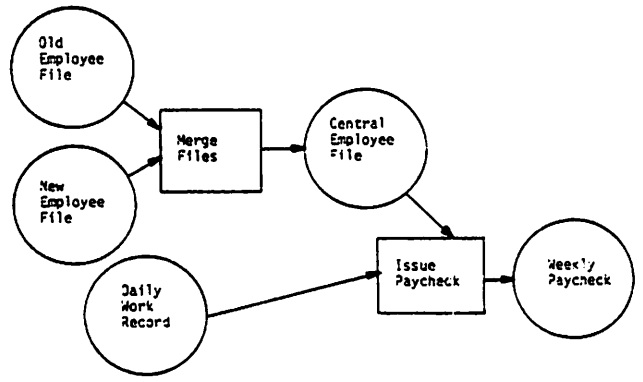


Figure 8. Simple Payroll System Data Flow

Most of the previous work in system specification deals with this class of systems. These approaches, however, have different motivations and objectives. A large portion of them are for documentation purposes only. These stem from the need for unambiguous communication among people involved in the system development and ease in systematically cross-referencing a large volume of information. There is little analysis performed on the specifications. In almost all cases, a language, a set of forms, or a discipline is developed. Previous surveys are documented in [Tei 72, Cou 73, Bur 74, Pet 76].

The following provides an overview of their basic common features, rather than emphasizing their differences. References to a specific approach are made when its unique features are discussed. Special attention is placed on the techniques for analysis of the specifications that have not been emphasized in previous surveys.

A typical approach usually has provision for specifying one or more of three groups of information: (1) the data flow network, (2) process or function specifications, and (3) other information.

4.3.1.1 Data Flow Network

The specification or requirement statement language provides the basic capabilities for stating what data or documents are required of the system, and the data requirements and processes for the derivation of each output. For example, in Figure 9, c is the required output and a specification statement may be as follows: Process P, with inputs a and b, is needed to generate c.

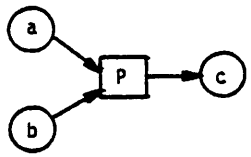


Figure 9. Example Data Flow Requirement

Some of the inputs to the process may be internal file data (usually referred to as history-type data). This procedure is continued until conceptually a network of data flow is constructed (Figure 10).

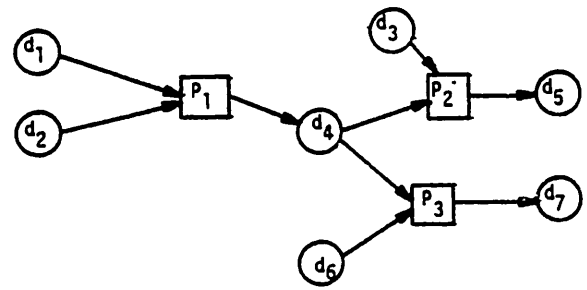


Figure 10. Example Data Flow Network

In Figure 10, the squares represent processes; and circles, data items. A graphical representation of their relationship is amenable to a number of basic analyses--typically connectivity and completeness (all data not feeding to some processes must be external output and all data with no source must be external inputs or history data). Other analyses that can be performed include identification of strongly interacting subsystems, restructuring, dynamics, etc. These are based on criteria that are less universally adopted.

4.3.1.2 Process or Function Specification

To specify the process or function is to define the functional relationship between the inputs and outputs of the process. If these are simple data elements, the functional specifications are easy, especially in management information systems where the relationships are typically arithmetical.

More complicated situations arise when the inputs or outputs are aggregates of simple data elements (e.g., a file of records). For instance, a process has to calculate the total net pay of an employee from a file of pay rate and a file of work time.

There are attempts to develop a language that is capable of expressing these relationships among data (without specifying implementation). A formal approach is the information algebra developed by the CODASYL Development Committee [COD 62].

Among the key concepts introduced are lines, areas, bundles, glumps, and functions on them. Lines correspond (roughly) to records and areas to files. A bundle or a glump specifies a new file based on data in other files. These descriptions given here are vague because the original notions are built up from a rather elaborate basis. A detailed example is included in paragraph 5.2.1.3.

Some additional concepts are yet to be developed beyond simple lines and areas for more general applications. In the end, this can be extended to a comprehensive data base language. The possibility of using such a language for specification of data flow and manipulation should not be ignored.

4.3.1.3 Other Information

Besides the data flow and processing function specification, some additional information can be specified within the existing framework. Among these, the most common ones are timing and system load information. Timing specification deals with requirements of the following types: (1) some output documents must be provided before a certain deadline, (2) certain outputs have to be produced periodically, and (3) certain outputs have to be produced after an input event within a specified response time. System load specifications include size of file, maximum, mean, minimum, frequency of usage of data, etc.

Analysis of this information may be important in some systems. Information such as a calendar of events or the volume of data flow among subsystems can be extracted from these specifications. More elaborate analyses are based on a graphical model of the static flow and volume information. We illustrate the approach with an example. Given a network of data flow as in Figure A-10, let $P_1, \ldots, P_n$ be the set of processes and $d_1, \ldots, d_k$ be the set of data sets. An incidence matrix of the processes and data sets can be obtained, which is defined as follows:

$$e_{ij} = \begin{cases} 1 \text{ if } d_j \text{ is an input to } p_i \\ -1 \text{ if } d_j \text{ is an output from } p_i \\ 0 \text{ if there is no direct incidence between } d_j \text{ and } p_i \end{cases}$$

Let v_j be the volume (e.g., storage size) of d_j; e_i be the number of inputs and outputs for p_i; and m_j be the number of times d_j is used either as input or output, then

$$e_i = \sum_{j=1}^{k} |e_{ij}| , \quad i = 1, \ldots, n$$

$$m_j = \sum_{i=1}^{n} |e_{ij}| , \quad j = 1, \ldots, k$$

The transport volume for d_j is $t_j = m_j v_j$, and the transport volume for the whole network is $T = \sum_{j=1}^{k} m_j v_j$. This may be taken as a measure of the amount of data movement necessary between the main and secondary storages.

4.3.2 Methods for Specifying Control Dominant Systems

In some real-time systems, the control flow or the sequence of operations of the system becomes the dominant features for consideration. Many important aspects of a system are characterized by its control flow. The operation sequence, the interaction pattern of the subsystems, the deadlock situation, etc., are among the more important ones.

In designing a system, it is desirable to have these features validated before commitment to further design and implementation. Therefore, specification of the control flow of a system is important even at the earliest stage of the design.

An approach in this category generally develops a graph model of the system abstracting the structural aspects, so that some formal properties of the

system can be analytically examined, and desirable and undesirable features identified.

Here we will examine the theoretical background of this group of specification techniques without going into details of their specific setup.

Graph models have been developed [Bae 68, Cer 71, Den 70, Hol 70, Luc 68, Pat 70, Rod 67, Slu 68] to study various aspects of computer systems and other more general systems. Originally, the more important issues considered are assignment and sequencing of computations, harmonious cooperation of processes, memory allocation, transformation of sequential processes into parallel ones, etc. Currently, much interest has developed for extending these studies to model system level requirements of large-scale software. Examples of research effort along this line are [Ros 72, DaV 76, Alf 76a, Bal 76, Sal 76, Bel 76].

The major advantages of using a graph model for the representation and analysis of system specifications and requirements are fourfold: (1) many graph models are capable of modeling very general features of large systems, especially the asynchronous interactions of parallel subsystems; (2) abstractions--a graph model retains certain properties and leaves out irrelevant detail (this makes them particularly attractive for formal analyses); (3) the existence of a variety of theoretical techniques for studying the models; and (4) visual convenience--a graph can be an excellent aid for visual understanding, which is very useful for human analysts.

To improve the concreteness of the discussion, we will use the Petri net as an example for sampling the available analysis techniques on the model. (Many of the theoretical considerations have a direct analog in other models.)

A Petri net [Pet 62] is a bipartite directed graph, which means that it consists of two types or sets of nodes (usually represented by circles and bars) and a set of directed arcs going from a member of one set of nodes to a member in the other set. The two sets of nodes are places (circles) to be interpreted

as conditions or events, and transitions (bars) to be interpreted as processes or actions. A dot (token) may or may not be present at each place. When there is a token in a particular place, the place is referred to as a condition being true or the condition holds. Figure 11 shows a Petri net. The places are $c_1, \ldots, c_5$ and the transitions are $t_1, \ldots, t_4$. Conditions c_2, c_3 are holding while the rest are not. The pattern of tokens in a net is called the marking. A net models the following situation: for each transistion, if every input place (places with a directed arc to the transition) is holding, then the transition is enabled and can fire. After it fires, the output conditions will hold, i.e., each of the output place receives a token. Thus a net represents all the potential asynchronous actions.

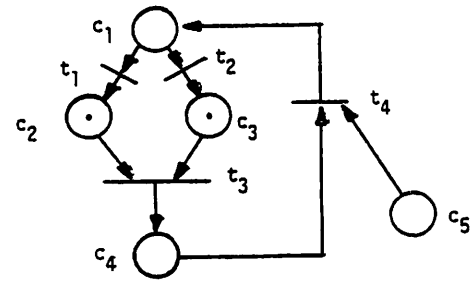


Figure 11. Petri Net Example

Many properties of a net can be studied. A net is live if all its transitions are live. A transition is live given a particular marking if there exists a sequence of firings that fire it for every marking reachable from the given marking. A marking is reachable from another marking if there exists a sequence of firings transforming the latter into the former. A live net roughly corresponds to the situation of the absence of deadlocks. A net is safe if all places are safe. A place is safe given a particular marking if every marking reachable from the given marking has at most one token on that place. Other theoretical problems pertaining to Petri nets and generalized Petri nets are

the reachability problem and the liveness problems that address the problem of whether a particular marking can be reached and whether a given marking of a net is live, respectively. Many of these problems are difficult decidability problems. More thorough discussions are found in [Kar 69, Hac 74].

Lauer and Campbell [Lau 75] developed a hierarchical classification of Petri net structures for system descriptions. A system is described in terms of a path and a program specification corresponding to the synchronization and processing sequence, respectively. Certain semantic correctness properties of a system can be inferred from its syntactic classification in the hierarchy.

Gostelow developed the notions of proper termination and module substitution [Gos 71, Gos 75]. Proper termination is a formal description of the property of being able to "reach the end" of a net in a "well-behaved" and "structured" manner. Effective procedure for the determination of proper termination exists [Gos 72]. The module substitution idea is developed around the notion of two nets "behaving identically" if a part of one of them is replaced by another. The substitutability property is defined in terms of proper termination and the desirable property can be proved.

Postel attempts to define the notion of a proper module [Pos 74]--a net structure that has the desirable properties of a system unit for independent development or for substituting one another. Proper module is a weaker characterization of proper termination.

Gostelow [Gos 71], Cerf [Cer 72], and Postel [Pos 74] developed and applied the reduction concept to analyze Petri nets. Reduction is a procedure that homomorphically transforms a set of transformation expressions (characterizing a net) to another while preserving some properties of the original set. Proper termination can be partially determined by the reduction procedure, and similarly for a number of auxiliary notions developed around proper termination such as proper module and substitutability.

There is vast literature on the various graph models that will be exhaustively summarized here. Although most of the work is not done specifically for system specification analysis, many of the results may be applicable in this area.

4.3.3 Specification Analysis System

The techniques already covered in this section are intended to be used over the extended period of software architecture design. For a large project, very voluminous information is generated among a large group of system designers. Automated aids are needed for managing this situation. In more recent research, it usually takes the form of a specification analysis system so that the system designer can input their specifications, which are deposited in a central data base. A collection of automated tools will be available to perform various analyses on the specifications. The most representative of this type of system is found in ISDOS [Tai 74b] and SREM [Bel 76b]. In the following we will briefly describe some of the technical features of a typical specification analysis system--a schematic diagram of which is shown in Figure 12.

4.3.3.1 Monitor

The monitor is the interface between the user and the analysis system itself. Instrumental to the success of a specification system is the nature and extent of interaction provided by the system. For this reason, considerable attention must be given to the development of suitable feedback mechanisms and formats. An important design consideration of the monitor is that it should be able to guide the user through a complete specification analysis, while the system automatically maintains a partially entered or changing set of specifications.

Other supplementary functions performed by the monitor may include configuration and management control of the development project, and automatic generation of documentation in human readable form, etc.

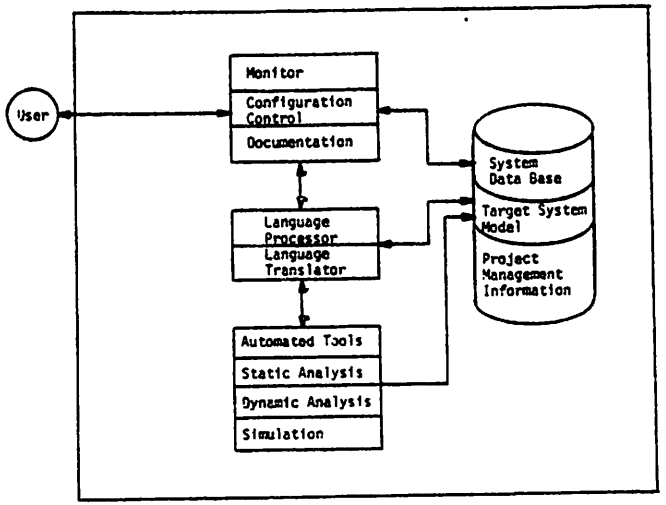


Figure 12. Specification Analysis System

4.3.3.2 Language Processor

The language processor will perform the usual function of analyzing the input statements written in the specification language accepted by the system. Syntactic analysis and other diagnosis results are immediately feed back to the user. The accepted input is deposited to the central data base via the language translator.

4.3.3.3 System Data Base

The most important part of th system data base should contain a computer representation or model of th target system under development. This model will be continuously modified under appropriate configuration controls, including addition, deletion, modification, and entry of alternative versions. The representation is directly submitted for analysis.

4.3.3.4 Automated Tools

A collection of analysis tools of the system data base is usually the key component of specification analysis systems. These tools, however, are strongly dependent on the specific techniques used for specification and their underlying concepts. The general features include the static analysis tools that check out the necessary and desirable features of the data base such as consistency and completeness. A variety of dynamic analyses of the target system can be performed depending on the system model use. The most widely used analysis technique for studying the dynamic behavior and performance of a system at its specification stage is simulation. Simulation can be generated either automatically or manually. The specific analysis tools have been discussed along with the specification techniques in paragraphs 4.3.1 and 4.3.2.

4.4 SPECIFICATION TECHNIQUES FOR DETAILED SOFTWARE DESIGN

Because the level of detail in specifying a detailed software design is much lower when compared to specifying the software architecture, the techniques in this section are applicable to higher levels if not limited by the complexity and by minute details. We will examine those techniques that try to describe the entities concerned down to the most primitive details, although they may be built up hierarchically by a number of levels.

There are, loosely speaking, two aspects for specification--a unit of action and a class of objects. A unit of action, in current high-level programming language terminology, may be a procedure, block, or subroutine. The overall system functions are ultimately built up of these basic units of action, e.g., a sorting routine, a Kalman filter procedure, etc.

Specification of a unit of action, which we will later refer to as process specification, is to specify succinctly all the effects that the action will have on its environment. The primary objective is to satisfy the minimality criterion that all the relevant, and only that, information needed to use the unit is stated. In general, the implementation information is not relevant.

The second criterion is comprehensibility, i.e., the user should be able to understand all the effects explicitly. This also requires that the specification be unambiguous. The specification should also allow a unique interpretation. The two aspects are simultaneously satisfied only by a formal but natural notation.

The third criterion is whether the method allows formal manipulation, in particular, proving that an implementation satisfies the specification and that the specification itself satisfies some consistency requirements.

A unit of action may belong to one of the two classes requiring very different specification techniques because of a fundamental difference between them. These two classes are sequential process specification and parallel process specification. A sequential process can be thought of as one that will be in action alone (although some other processes may be active concurrently, they have no influence on one another). On the other hand, in specifying parallel processes, specific attention must be paid to their interactions with one another.

The idea of "data type" or data abstraction is to treat the objects in programming as abstract entities instead of their physical implementation. The objects are treated as primitives and their semantics are completely defined in terms of the set of primitive operations and relations on them. These are similar to the definition of mathematical objects, treated as undefined concepts, and constructed inductively by primitive constructors.

In the following, specific classes of techniques are outlined under the headings of methods for specifying sequential processes, parallel processes, and data types.

4.4.1 Methods for Specifying Sequential Processes

We have identified three approaches. A more detailed survey has been performed by Liskov and Berzins [Lis 76].

4.4.1.1 Enumeration of Input/Output Pairs

In this method, the function is stated as a list of input/output pairs. For example, to define the square function on the integers: f = ...(-1,1),(0,0),(1,1),(2,4).... Although this form is regarded as a specification, it is impractical in almost all situations and is theoretically impossible in some cases (e.g., for infinite input domain functions).

4.4.1.2 Exhibiting a Procedure to Obtain the Output From the Input

A procedural description may be used as a specification with emphasis on clarity of expression, whereas the actual implementation emphasizes the efficiency of the process so that a completely different algorithm may be used. The specification (possibly simpler in structure) is taken as "correct" or as "what is desired." The correctness of the actual implementation is validated against the specification.

In terms of the desirable criteria of a specification method, minimality is not satisfied since the specification of a procedure includes much information that is not needed. This is the major criticism of the procedural specification method. The user, say a programmer, may make use of some specific detail of the procedure, which may be changed in a new version. Substantial changes in a large portion of the system are necessary in this situation.

Furthermore, rigorous verification cannot begin until the whole program or system has been completed. This implies that the verification effort is complex and no corrective action can be taken until the very end.

With regard to comprehensibility, it depends on the programming style, structure, and more importantly, the size of the procedure. In general, this is not a critical problem because we are becoming aware of some principles of program clarity and readability [Dah 72].

4.4.1.3 Input/Output Assertion

The actual procedure of how to derive the output from the input is not stated. Instead, the relation between the input and output variables is specified in terms of more primitive relations and notions.

Example:

Specification of a sort routine
Sort (A: int array; N: integer)
ENTRY: $N \geq 1$
EXIT: $\forall_i((1 \leq i \leq N) \Rightarrow Sort(A,N)[i] = AMAX(Sort(A,N),1,i))$
and APERM(Sort(A,N),A,1,N))

This states that for an input array with one or more elements, the resulting array would be a permutation of the input array, and, furthermore, the output array elements are in ascending order.

The more primitive notions assumed to be defined are array indexing; AMAX, which finds the index of the maximum among the first N elements of an array; and APERM, a boolean function indicating whether an array is a permutation of another.

In the approaches within this category, the actual implementation detail is left out. Only the entry and exit conditions are stated; the user only needs to ascertain that the input condition is satisfied on the invocation of the procedure, and it is guaranteed that the exit condition is satisfied on termination. Verification involves showing that the implementation has this property.

Because the conditions are stated in terms of more primitive notions, the criterion of minimality may not be satisfied in that the choice of the particular primitive notions to implement the algorithms is arbitrary and may be a specific one from among many possibilities. In the example above, it is

assumed that the array is implemented with that particular indexing mechanism as used in the conditions.

The comprehensibility of this method again depends on the complexity of the procedure and also on the choice of the suitable level of the primitive notions used.

There are two fundamental limitations to the approach: (1) The growth in the complexity of the condition when the system or procedure to be specified becomes complex. Even with the best design of a hierarchy of the primitive notions, the complexity may still get out of hand; (2) The "naturalness" of stating the entry and exit conditions. While it seems ratural to express the "sort" procedure as in the example, for other functions it may not. A good example is the factorial function; the only natural way seems to be to exhibit a procedure. Similarly for a "differentiate" function, it is more natural to show how to do it than telling what it is all about. On the other hand, stating the entry and exit conditions is far simplier in the case of matrix inversion, etc.

4.4.2 Methods for Specifying Parallel Processes

Specification techniques for parallel processes have not been studied as extensively as those for sequential processes. Their importance is now apparent since actual parallel hardware is gradually being employed. More importantly, a system viewed as a society of parallel processes is conceptually cleaner and more elegant. Principato has performed an extensive survey of these techniques [Pri 77]. The techniques are classified into three categories.

4.4.2.1 Parallel Constructs for Programming Languages

Program languages are extended with special primitives to indicate interactions among parallel processes. Thus, for example,

<u>cobegin</u> $P_1, \ldots, P_n$ <u>coend</u>

indicates that the processes P_1 through P_n will be executed concurrently. If any of these processes refers to some common data, the computation results are usually indeterminate. For a specific application, these random interactions must be restricted. The most common example cited is the reader and writer problem. Two groups of processes, designated readers and writers, are working on a shared data base; the readers only refer to information in the data base while a writer may modify it. To guarantee predictable results, whenever a writer is working on the data base, no other processes may be permitted to use it, even for reading only. However, any number of readers may use the data base simultaneously. This and other problems of synchronization and communication, and other criteria such as fair and deadlock free scheduling, etc., lead to the development of the semaphore [Dij 68a], critical region and conditional critical region [Bri 73], and monitor [Hoa 74] primitives.

Using these primitives to specify parallel processes is analogous to the techniques described in paragraph 4.4.1.2 for the sequential case. They share the same comment that an actual solution is exhibited rather than stating just what is intended.

4.4.2.2 Event Approach

Techniques in the last category integrate the sequential processing aspects with the parallel synchronization aspects. It is possible to isolate the latter so that they can be examined separately. A method using the event approach, such as Habermann's path expressions [Cam 74, Hab 75], tries to identify the allowable sequences of events from all the processes. For example, in the reader/writer problem described in the last paragraph, the sequences of events allowed are those in which an exit from the shared data by a writer i immediately follows an entry of the same process i.

These techniques have the merit of exhibiting the synchronization explicitly, so that their properties can be proved formally. Deadlocks and other undesirable situations can be uncovered. The approaches of Shaw [Sha 76], Greif [Gre 75], and Reed [Ree 76] are in this category.

4.4.2.3 State Variable Approach

In the state variable approaches, variables are introduced sometimes additional to the normal program variables to specify the states of the system of the parallel processes. Constraints on the possible states that can be taken are stated as a set of invariants involving the state variables. These invariants are supposed to characterize the synchronization. A simple example is the synchronization of a producer and consumer sharing a buffer. A set of state variables may be the number of units produced, p, number of units consumed, c, and number of units in the buffer, b. Assuming that the buffer size is N, the constraints are: $b \leq N$ and $c \leq p-b$.

Although formal proofs of desirable properties of the synchronization and the correctness of implementation can be obtained, it may be difficult to specify correctly the desired synchronization intended.

Specific techniques in this category are those of Robinson and Holt [Rob 75] and Owicki [Owi 75].

4.4.3 Methods for Specifying Data Types

There are a large number of researchers in this area [Gut 76, Zil 75, Gog 75, Par 72, Hoa 72a, Ear 71]. An excellent survey of existing techniques in data-type specification can be found in [Lis 75, Lis 76].

An abstract data type is a collection of objects with a set of permissible operations specified. Any manipulations on these objects must be done via the permissible operations, and conversely, the characteristics of the objects are completely defined by these operations alone. The exact representation of the object is not part of the specification.

An illustrative example is the definition of a stack (Figure 13) [Gut 76]. The primitive notions are stack, item, boolean; and the primitive operations are MTSTACK (which creates an empty stack), PUSH, POP, TOP, and ISMTSTACK. The complete specification of the stack consists of six axioms (Figure 13).

Note: Taken from [Gut 76].

```
declare   MTSTACK()→stack
          PUSH(stack, item)→stack
          POP(stack)→stack
          TOP(stack)→item
          ISMTSTACK(stack)→boolean;
for all   s ε stack, i ε item,
Axiom 1   ISMTSTACK(MTSTACK) = true
      2   ISMTSTACK(PUSH(s,i)) = false
      3   POP(MTSTACK) = MTSTACK
      4   POP(PUSH(s,i)) = s
      5   TOP(MTSTACK) = undefined
      6   TOP(PUSH(s,i)) = i
```

Figure 13. Algebraic Axiom Specification of Stack

The most salient features of the stack are: POPing a stack results in the one just before the last PUSH operation, and examining the stack gives the last item pushed in.

These are captured minimally by the axiom system. It is minimal because it only involves those concepts that are relevant at the level of consideration of the stack.

The comprehensibility of thi form of definition depends on whether the object being specified has operatio, with simple relations among them. It can be very complex, difficult to understand, and may involve more than it requires to state explicitly the algorithms of the individual operations themselves.

Currently we know of no effective (general) method either to verify whether an implementation of a given data structure satisfies the specification or to verify automatically certain "correctness" or "consistency" properties of a set of axioms.

4.4.4 Assessment of Software Design Specification Techniques

After this survey of the variety of techniques, a summary statement about the whole class can be made. So far we have been dealing with the specification of units of relatively small sizes. The information specified is complete and precise in the sense that everything needed is given. Because of this property, the complexity and comprehensibility become a stumbling block.

The same problem of complexity exists even in a greater magnitude in the verification of the specification. Special techniques by relaxing the preciseness and completeness must be used in large systems. Some aspects of all the information is given up for concentration on more important features, namely, the structural aspects of the system.

Another major problem is in the construction of larger and larger units from basic blocks. We have seen roughly how this can be achieved in a hierarchical fashion in the discussion of algorithm specification. However, emphasis is not placed on the coordination and performance aspects of the subsystems of a system. All techniques discussed so far do not address these aspects. They become the prominent feature of a large system, and special techniques have to be developed for them, as have been discussed in Sections 4.1 through 4.3.

5. SPECIFICATION TECHNIQUES SUMMARY

This section provides a summary description of the individual techniques to highlight their significant features that have not been mentioned in the last section. The organization of the techniques is closely parallel to the classification framework developed in Section 4.

Table 2 gives a condensed summary of the techniques for quick comparison and cross references between Section 4 and 5 via the Category and Summary Section columns. Additional information regarding the nature and status of the techniques not specifically mentioned in the summary sections is also included in the table.

Table 2. Specification Techniques Summary Table (Sheet 1 of 3)

Category	Methodology	Principal Developer	Nature*	Status	Principal Concepts	Reference	Summary Section
System Requirements (Overall Needs and Objectives)	Semantic Nets		C			Sim 73	
	IA	Wilson	M-Computerized	Research	Semantic representation of design context	Wil 75	
		Becker	M-manual, computer assisted	Applied to example	Semantic representation of design context Fuzzy concepts	Bec 73	
		Zadeh	C		Precise concepts to deal with fuzzy notions and complex systems	Zad 71	
Data Processing Subsystem Requirements		Wymore	M-manual	Applied to example	Input/output trajectories specification of system functions	Wym 76	
	HOS	Hamilton, Zeldin	MC-computerized	Research	Axioms for system decomposition Specification language-AXES	Ham 76a, b, c	5.1.1
		Fitzwater	C	Research	Formal elaboration Interaction primitives	Fit 76	5.1.2
	SADT	Ross	MC-manual	In use	Structured design Graphical notations	Ros 77	5.1.3
	F^2D^2	RCA	M-manual	In use		RCA 72	5.1.4

* C -- Conceptual notions
M -- Methodology
L -- Language

Table 2. Specification Techniques Summary Table (Sheet 2 of 3)

Category	Methodology	Principal Developer	Nature*	Status	Principal Concepts	Reference	Summary Section
Software Architecture Specification: Data Dominant Systems	Decision table		M	In use		Lon 72	5.2.1.1
		Young, Kent	M	In use		You 58	5.2.1.2
	Information Algebra	CODASYL Committee	L	Inactive	Formal language primitives for complex data operation involving files	COD 62	5.2.1.3
		Langefors	C	Not known		Lan 63	5.2.1.4
	Systematics	Grindley	M	Not known		Gri 66	5.2.1.5
	ADS	NCR	M	In use	Use of predefined forms	Lyn 69	5.2.1.6
	TAG	IBM	M-computer assisted	In use		IBM 71	5.2.1.7
	HIPO	IBM	M-manual	In use		IBM 73, Sta 76	5.2.1.8
	ISDOS	Teichoew	M-computerized	In use	PSL/PSA System data base	Tei 74b, 77	5.2.1.9
	SODA	CWU	M	Not known	Evaluation of alternative designs Optimization of design	Nun 71	5.2.1.10
	BDL	IBM	M-computerized	Development	Direct implementation of system from user specs	Gol 75	5.2.1.11
		Ho, Numamaker	L	Designed	Formal specification of complex data relations	Ho 74	5.2.1.12
		Bridge, Thompson	L-extension	Designed	Classification and declaration of data usage	Bri 74	5.2.1.13
	Structured Design	Constantine	M-manual	In use	Design guidelines Structure chart, bubble chart	HUG 75	5.2.1.14

* C -- Conceptual notions
M -- Methodology
L -- Language

Table 2. Specification Techniques Summary Table (Sheet 3 of 3)

Category	Methodology	Principal Developer	Nature*	Status	Principal Concepts	Reference	Summary Section
Software Architecture Specification:	LOGOS	Rose	M	Not known	Formal graph model and analysis	Ros 72	5.2.2.1
Control Dominant Systems	CSC-Threads	CSC	M-manual	In use		CSC 73	5.2.2.2
	SREM	TRW	M-computerized	In use	Extensible RSL R-nets REVS--Analysis System	Alf 76a, b Bel 76a, b	5.2.2.3
	FSM	AFC	M-computerized	Research	Finite state machine model of system	Sal 76	5.2.2.4
	VG	CSC	M-computerized	Research	Graph model	Bel 76	5.2.2.5
	GRC-Petri net	GRC	M-computerized	Research	Petri net model and analysis	Bel 76	5.2.2.6
		Booth, et al.	M	Research	Computation structures Performance modeling	Sho 76	5.2.2.7
Detailed Software Design Specification:		McGowan	L-manual	In use		McG 75	5.3.1.1
Sequential Processes		Parnas	L-manual	In use	Effect specification of functions--implementation independent	Par 72	5.3.1.2
		Good, et al.	L-computer verification	Research	Formal input/output specifications	Goo 75	5.3.1.3
		Hewitt, Smith	L-computer verification	Research	Contracts (specifications) Meta-evaluation	Hew 75	5.3.1.4
Detailed Software Design Specification:	V-Graph	Earley	L	Research	Graphic specification of data types	Ear 71	5.3.2.1
Data Types		Guttag, et al.	L	Research	Axioms of data properties Formal verification	Gut 76	5.3.2.2

* C -- Conceptual notions
M -- Methodology
L -- Language

5.1 SPECIFICATION TECHNIQUES FOR DATA PROCESSING SUBSYSTEM REQUIREMENTS

5.1.1 Higher Order Software

The Higher Order Software (HOS) methodology [HAM 76a,b] is an example of the axiomatic approach. Desirable features of a system are characterized by a set of axioms that can be checked on a specific system specification. The HOS methodology is shown in Figure 14.

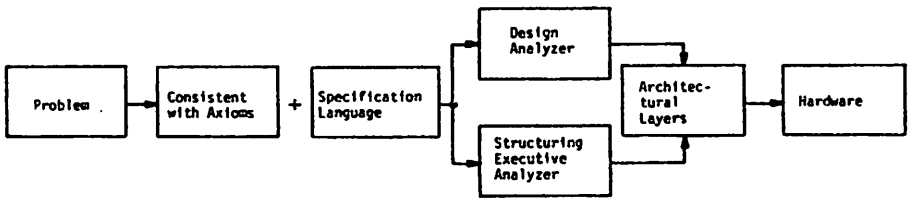


Figure 14. HOS Methodology

The key feature is to establish a formal basis. A system is defined as a process tree of modules or subfunctions (Figure 15). Each of the nodes is a "function" in the mathematical sense.

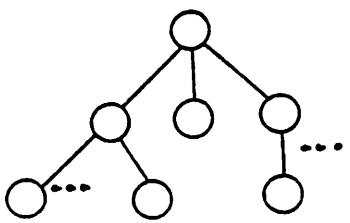


Figure 15. Subfunction Process Tree

The relationship between the nodes in the trees is one being a controller of the other, and being a subfunction (submodule) in the reverse direction (Figure 16).

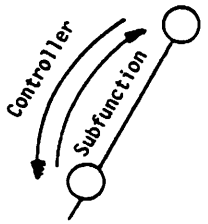


Figure 16. Process Tree Node Relationship

The decomposition of a module into subfunctions must obey the following rules (Figure 17), corresponding to a partition of the input space, a partition of output components, and a composition function.

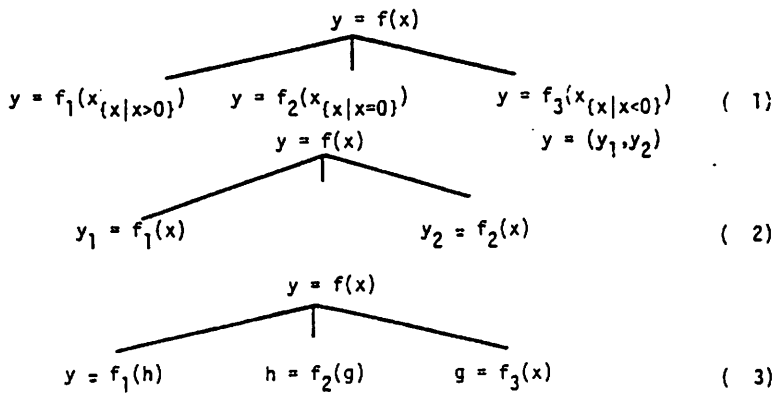


Figure 17. Module Decomposition

The functions and their parameters must satisfy a set of six axioms governing six aspects of control. Figure 18 summarizes the definitions of the controls and axioms.

Theorems can be derived from these axioms asserting many desirable properties that a system should have, such as correct interfaces.

A specification language is being developed so that a system specified by the language can be checked against the axioms. If these are satisfied, the system to be implemented will possess these desirable properties.

5.1.2 Fitzwater

Fitzwater [Fit 76] introduces a philosophy and methodology for developing and analyzing distributed data processing systems. The main emphasis is on formal decomposition. The originating requirements of a system development are always stated informally. The steps in developing the system obtain a successive set of approximations to the final system. These intermediate systems represented by these approximate sets may be formally analyzable to determine whether they produce the desired behavior (Figure 19).

To effect the decomposition, a formalism is introduced based on a state-machine description of the system functions. We shall outline the salient features in two paragraphs--functional and interaction specifications.

5.1.2.1 Functional Specification

The system is represented by an ordered pair $(\sum, f)$, where $\sum$ is referred to as the state space, and f, the state successor function. $\sum$ is a cross product of the component state space $\sum_1 \times \sum_2 \times \ldots \times \sum_m$. Each of the components $\sum_i$ is the set of subsets of a value space V_i; f, (with argument $\sigma \in \sum$) is a nondeterministic state successor function.

The state successor funtion f can be represented by a state grapn resembling a state transition diagram. The complete precise definition of the state transi-

Note: Taken from [Ham 76c].

DEFINITION: Invocation provides for the ability to perform a function.

Axiom 1: A given module controls the invocation of the set of functions on its immediate, and only its immediate, lower level.

DEFINITION: Responsibility provides for the ability of a module to produce correct output values.

Axiom 2: A given module controls the responsibility for elements of only its own output space.

DEFINITION: An output access right provides for the ability to locate a variable, and once located, the ability to place a value in the located variable.

Axiom 3: A given module controls the output access rights to each set of variables whose values define the elements of the output space for each immediate and only each immediate, lower level function.

DEFINITION: An input access right provides for the ability to locate a variable, and once located, the ability to reference the value of that variable.

Axiom 4: A given module controls the input access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate, lower level function.

DEFINITION: Rejection provides for the ability to recognize the improper input element in that if a given input element is not acceptable, null output is produced.

Axiom 5: A given module controls the rejection of invalid elements of its own, and only its own, input set.

DEFINITION: Ordering provides for the ability to establish a relation in a set of functions so that any two function elements are comparable in that one of said elements precedes the other said element.

Axiom 6: A given module controls the ordering of each tree for the immediate, and only the immediate, lower level.

Figure 18. HOS Control Aspects and Axioms

Note: Taken from [Fit 76].

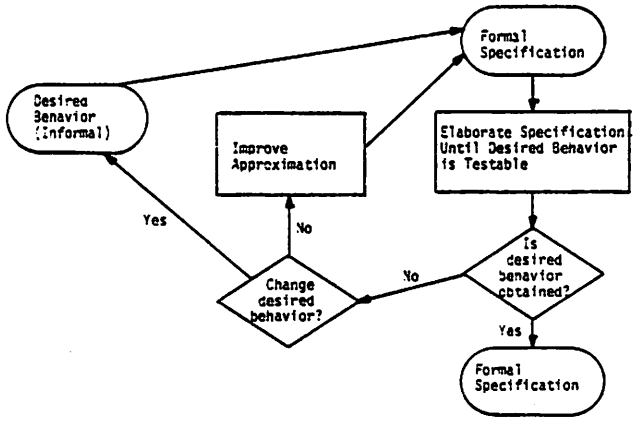


Figure 19. Requirement Specification Process

tion function is done via a series of decompositions. In the first level, f is decomposed into component successor functions f_i. We give an illustrative example rather than the mathematical definition of the f_i's.

Suppose we have a state successor function $f:\Sigma \to \Sigma$ where $\Sigma = \Sigma_1 \times \Sigma_2 \times \Sigma_3 \times \Sigma_4$; f can be decomposed into component successors f_1, f_2, f_3, given by:

$$f_1: \Sigma_2 \times \Sigma_4 \to \Sigma_1 \times \Sigma_3$$

$$f_2: \Sigma_1 \times \Sigma_2 \to \Sigma_2$$

$$f_3: \Sigma_3 \to \Sigma_4$$

An f_i is further decomposed into value successor functions f_{ij}. f_{ij}'s are distinguished from the f_i's as being value space valued functions where the

f_i's are set valued functions. Further carrying on with the example, f_1 may be decomposed into

$$f_{11}: V_2 \times V_4 \to V_1$$

$$f_{12}: V_2 \times V_4 \to V_3$$

The f_i's and f_{ij}'s can be further decomposed using composition, subtree selection, and primitive recursion until all the components are regarded as primitive functions.

5.1.2.2 Interactive Specifications

A set of interaction primitives called exchange functions are introduced to specify interactions and synchronization among parallel processes. These exchange functions will exchange values of arguments with a matching exchange function elsewhere in the specification. The exchange of arguments between a pair of matching exchange functions is accomplished by having each of them evaluate to the argument of the other. Exchange functions are labelled with subscripts and only exchange functions with the same label can match. The set of exchange functions with a given subscript is referred to as a class. Thus exchanges may only occur between members of the same class.

The three exchange functions XC, XA, XS are defined as follows:

$XC_i(\alpha) = \beta$ if there is an outstanding $XC_i(\beta)$ or $XA_i(\beta)$ that has been waiting for a matching exchange function

or

if this $XC_i(\alpha)$ has been waiting for a matching exchange function and an $XC_i(\beta)$, $XA_i(\beta)$, or $XS_i(\beta)$ is evaluated.

$XA_i(\alpha) = \beta$ if there is an outstanding $XC_i(\beta)$ that has been waiting for a matching exchange function to be evaluated

or

if this $XA_i(\alpha)$ has been waiting for a matching exchange function and an $XC_i(\beta)$ or $XS_i(\beta)$ is evaluated.

$XS_i(\alpha) = \beta$ if there is an outstanding $XC_i(\beta)$ or $XA_i(\beta)$ that has been waiting for a matching exchange function to be evaluated.

$= \alpha$ otherwise.

These are all the primitives that are needed to specify process interactions. Rules are introduced to restrict te use of these primitives so that certain properties of the interactions can be maintained such as ensuring that the path from stimulus to response is bounded.

5.1.2.3 Assessment

The method proposed is a formal approach capable of stating a system function with absolute preciseness. However, whether the system architecture designer is capable of specifying what he wants with the approach is questionable. All other formal methods, e.g., HOS, share this problem. Furthermore, the interaction primitives introduced may lack intuitive appeal to requirements engineers; the properties of an interaction pattern may not be transparent. This may result in a hindrance to their actual use in system specification.

5.1.3 Structured Analysis and Design Techniques

The Structured Analysis and Design Technique (SADT), developed at Softech, is essentially a graphic language supplemented by verbal language labels and other notations to desc ibe a system structure hierarchy [Ros 77].

Figure 20 shows dynamically the hierarchical decomposition of system specifications in different levels.

The basic entities that the language is able to express are activities and data and their relationships (e.g., input/output). An activity or data item is represented by a box, and its relation with the rest of the world is expressed by arrows going into or out of the box, labelled to convey the appropriate meanings. Some examples are shown in Figure 21.

Note: Taken from [Ros 77].

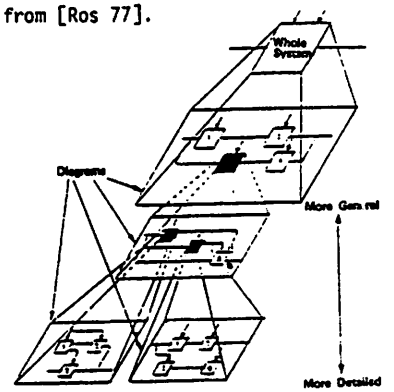


Figure 20. Structured Decomposition

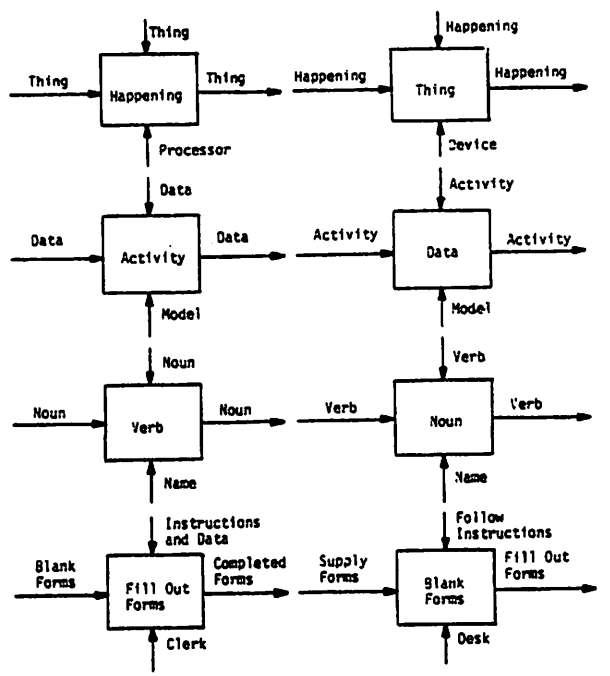


Figure 21. SADT Graphical Language

Each box, of course, can be expanded to a level of greater detail consisting of a network of boxes. The complete language consists of many more notations and conventions than can be described concisely here.

It has been reported that SADT has been applied manually to a number of system development projects. The results were subjectively evaluated to be very successful. The language notation is among the most elaborate used in software development, resembling a language for designing blueprints in electrical equipment or architectural design. It is useful as a visual aid. Computer assistance to SADT is obviously needed and possibly easily implemented. However, the important issue of analyzing the specifications beyond simply checking for the correct use of language notations has not been emphasized. It is apparent that the language design was not based heavily on the analysis consideration.

5.1.4 Functional Flow Diagram and Description

Functional Flow Diagram and Description (F^2D^2) is a proprietary corporate approach of RCA used for system decomposition and audit purposes. We are not aware of any published literature on the technique. The essential feature of the technique is to serve as a graphic language representation quite similar to SADT.

Each functional unit is represented by a box, and the related items: input, output, controls, etc., are specified. Part of the symbol usage is shown in Figure 22.

The function can be connected with others to create a network showing the complete system structure. Each function may be decomposed into more details. The levels are called "tiers" in F^2D^2 terminology. In Tier 0, the system functions are identified. Tier 1 identifies the hardware, software, and manual functions. Tier 2 proceeds down to a lower level of software functions, e.g., identifying the data records and display formats.

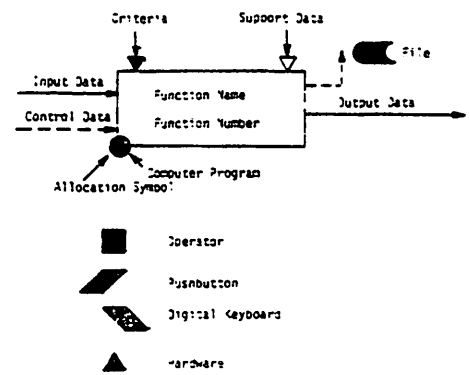


Figure 22. Functional Flow Symbols

F^2D^2 has been used internally by RCA for a number of projects, mostly manually, and serves as precise documentation. Analysis of the diagrams is limited to visual inspections.

5.2 SPECIFICATION TECHNIQUES FOR SOFTWARE ARCHITECTURE

5.2.1 Data-Dominant Systems

5.2.1.1 Decision Tables

A large body of literature on decision tables has been published. For a general introduction, see [Lon 72]. The main use of decision tables is in the programming of information-based decision problems.

The primary aim of decision tables is to represent man/man and man/machine communications of decision-making processes. The information is presented in a tabular form. It is not specifically intended to be implementation-free or nonprocedural. Automatic processors have been implemented to translate decision tables directly to executable object codes.

The basic features of the decision tables are described below. There are a large number of extensions of these basic features. The reader is referred to the bibliography section of [Lon 72] for a detailed exposition.

A decision table corresponds to a module and consists of a condition section and an action section (Figure 23). The interpretation (using the example) is as follows. Column 1: if not "promoted" and "minimum 10 percent raise," then "I stay"; Column 2: if not "promoted" and not "minimum 10 percent raise," then "I quit"; Column 6: "promoted" and no "more work interest" and not "own office," then "I quit."

It is readily observed that a hierarchy of decisions can be constructed so that an action entry in a table can be expanded to a decision table by itself in a lower level. However, it is equally apparent that the processing problems must be relatively simple and easily expressible as basic decisions, which may not be the characteristic of large, complex problems. Furthermore, the decision tables only express system logic, ignoring performance requirements.

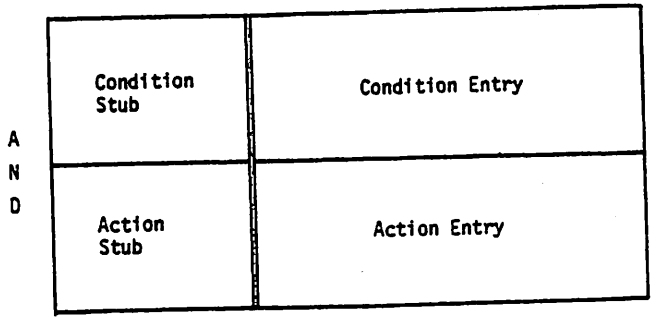
5.2.1.2 Young and Kent

A notation is provided for a precise and abstract specification of the informational and time characteristics of a data processing problem [You 58]. The intention is to avoid machine dependence, file structures, and a sequence of operations that is not considered necessary for logical correctness.

The language describes a system in terms of four kinds of basic components: (1) information set--a list of all possible items belonging to the same class that will be used in the system, (2) documents--a collection of related information items that is almost the same as a file of records, (3) relationships among information sets, and (4) operational requirements--the volume of input documents, the number of copies of outputs, response time, etc.

The presentation of the system specification is in the form of a graphical display with very elaborate symbols. All the information is put in the graph.

Note: Taken from [Lon 72].



a. Decision Table Structure

Title of Table
"To Quit or Not to Quit"

	1	2	3	4	5	6		8				12
Promoted?	N	N	Y	Y	Y	Y						
More Work Interest?	-	-	Y	Y	N	N						
Minimum 10% Raise?	Y	N	Y	N	-	-						
Own Office?	-	-	-	-	Y	N						
I Stay	X		X		X							
I Quit		X		X		X						

b. Decision Table Example

Figure 23. Decision Tables

However, the scheme is not easily understood. A detailed description of the notion is impossible here; Figure 24 is a highly condensed summary of the graphical notation with minimal explanation.

5.2.1.3 Information Algebra

The objective of Information Algebra is to define a language to specify information flow of a data processing system abstracting from hardware and other implementation constraints. The language is based on a formal theory drawing concepts from modern algebra and point set theory.

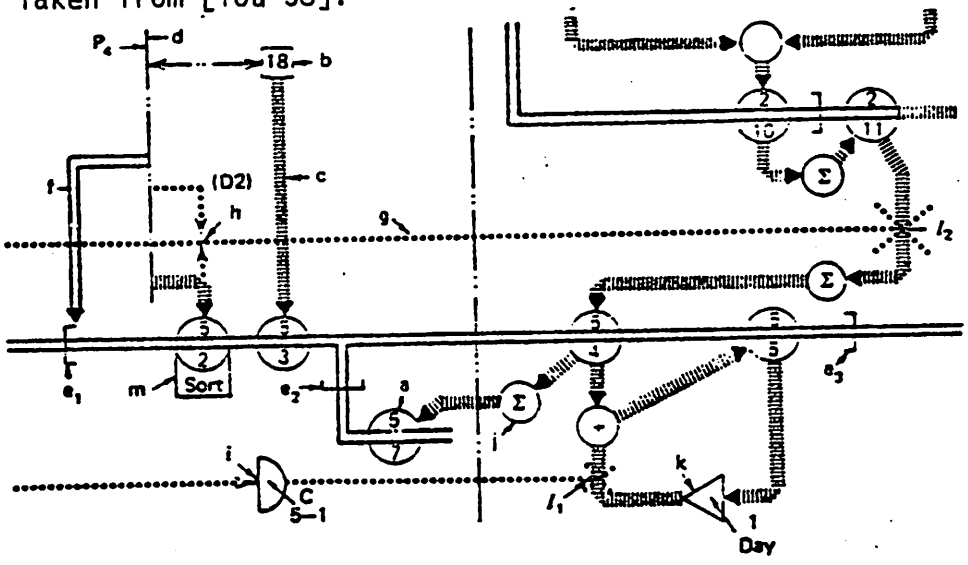
The basic features of the language are described in paragraph 4.3.1.2. The key concepts introduced in the language are: lines, areas, bundles, glumps, and the corresponding functions on them. Lines correspond (roughly) to records and areas to files. A bundle or a glump specifies a new file based on the data in other files. With these language primitives, a processing function involving complex operations on file data can be succinctly specified without stating a procedure to perform the operations. An example to illustrate the features of the language with example is given in Figure 25. The language, however, is difficult to use, and no support or anlaysis of the language has yet been developed. The language has been applied to a limited number of cases.

5.2.1.4 Langefors

The goal of the work [Lan 63] is stated as trying to arrive at a proper structure for a machine-independent problem defining language at the system level of data processing. A set of notations and graphical symbols are developed to achieve this purpose.

The basic feature of the approach is to specify the precedence relationships among data. A data item a preceeds b if it is required to calculate b. A matrix of these relations can be obtained and standard analyses can be performed. Specifying the memory requirement of the data elements allows further analyses of the total system memory requirement.

Note: Taken from [You 58].



Explanation of graphical notation.

(a) This circle represents an information item on a document, viz., D_{5-7}, with the document number above the line, and the item number within the document, below. The heavy double line (generally horizontal and possibly with branches) connects all the items on one document.

(b) Each information set, P_i, is shown as a square, in this case P_{18}.

(c) An item on a document which is an element of some P_i is connected to it: the arrow runs out of the document for an input, into it for an output (i.e., one may think of the flow of information to and from the system as following the arrows).

(d) Isomorphism (or homomorphism) between information sets is shown by a double (or single) headed arrow. In this case we are saying that $P_{18} = P_4$, with the square standing for P_4 having been extended downward by the broken vertical line.

(e) The small square brackets enclose those items making up a line item of a document, in this case $D_{5-2,3,4,\text{and}\,5}$. Note that D_{5-7} is not a line item and is excluded by the bracket e_2.

(f) The producing relationship is shown by a double heavy line with an arrow pointing to the document or line item produced, e.g., P_4 produces a line item on D_5. The line runs from the extension (the vertical broken line) of the square box representing P_4 to the bracket representing a line item on D_5.

(g) The dotted line is used to connect various elements of a condition.

(h) The condition at h is that D_{5-2} is equal to P_4 (D_2), represented by condition lines from D_{5-2} and P_4 with arrows to the condition line g. That D_2 is involved is shown by writing in (D_2).

(i) This is the standard symbol for a gate and states the condition involved, viz., C_{5-1}. The input to the gate (the flat surface) is connected to the item or items involved in the condition. The output goes to the operation affected by the condition.

(j) The summation sign indicates that D_{5-7} is equal to the sum of D_{5-4}.

(k) The triangle represents a delay equal to the time indicated within it. Thus yesterday's D_{5-5} is added to D_{5-4} to produce today's D_{5-5}.

(l_1 & l_2) The dotted cross through an information transfer line (l_1) or a sum (l_2) (or any other operation or production) indicates that this transfer or operation is performed if the condition is true.

(m) The sort notation implies that the document or line item must be sorted by the item indicated.

Figure 24. Young and Kent Graphical Notation

TABLE A Payroll Problem–System Information

Properties	Value Set	Areas: Old Pay File CP	Daily Work File DW	New Employee File NE	New Pay File NP
q_1 = File ID	PF, DW, NE	X (always PF)	X (always DW)	X (always NE)	X (always PF)
q_2 = Man ID	00000 ··· 99999	X	X	X	X
q_3 = Name	20 alphanumeric characters	X	Ω	X	X
q_4 = Rate	00.00 ··· 99.99	X	Ω	X	X
q_5 = Hours	00 ··· 24	Ω	X	Ω	Ω
q_6 = Day number	0 ··· 7	Ω	X	Ω	Ω
q_7 = Total salary	00000.00 ··· 99999.99	X	Ω	Ω	X
q_8 = Pay period number	00 ··· 52	X	Ω	X	X
q_9 = Salary	000.00 ··· 999.00	X	Ω	Ω	X

(1) $NP = F_1[q_{12} = q_{22}, H(q_2, DW), OP]$

$$\cup\ F_2[q_{12} = q_{22}, H(q_2, DW), NE]$$

$$H = \begin{cases} q_2' = q_2 \\ q_5' = \Sigma\,[q_5 \leftarrow (q_5 < 8) \rightarrow 1.5 * q_5 - 4] + f_1 \\ f_2 = \Sigma\,[q_5 \leftarrow (q_5 < 8) \rightarrow 8] \\ f_1 = 0 \leftarrow (f_2 < 40) \rightarrow 0.5 * f_2 - 20 \end{cases}$$

$$F_1 = \begin{cases} q_1' = q_{27} + q_{15} * q_{26} \\ q_6' = q_{20} + 1 \\ q_9' = q_{15} * q_{26} \end{cases}$$

$$F_2 = \begin{cases} q_1' = PF \\ q_7' = q_{15} * q_{26} \\ q_6' = q_{20} + 1 \\ q_9' = q_{15} * q_{26} \end{cases}$$

The terms of equation (1) are identified in Figure A. Let us examine each element of this equation more closely. Consider first the expression $H(q_2, WD)$. The Daily Work File (DW) consists of the daily hours-worked entries for all employees. Using q_2 (Man ID) as a glumping function, each glump element contains the entries for a single employee. $H(q_2, DW)$ is a function of a glump which calculates the values of certain properties for a set of points where each point corresponds to a single glump element. These values are defined by

$$H = \begin{cases} q_2' = q_2 \\ q_5' = \Sigma\,[q_5 \leftarrow (q_5 < 8) \rightarrow 1.5 * q_5 - 4] + f_1 \\ f_2 = \Sigma\,[q_5 \leftarrow (q_5 < 8) \rightarrow 8] \\ f_1 = 0 \leftarrow (f_2 < 40) \rightarrow 0.5 * f_2 - 20. \end{cases}$$

In the definition of H, f_1 and f_2 are functions of areas. Note that $f_2 = 40$ unless $q_5 < 8$, in which case $f_2 = \Sigma\, q_5$. From this, it follows that $f_1 = 0$ if $f_2 < 0$; and $f_1 = f_2/2 - 20$ = half of excess time over 40, otherwise. The overtime rule used in q_5' states that a man is paid time and a half for each day's excess over 8, and also is paid time and a half for the excess over 40.

All other properties are undefined for this function of a glump.

The set of points obtained from $H(q_2, DW)$ constitutes an area which is the first area of the area set $[H(q_2, DW), OP]$. A bundle is defined over this area set. The bundling function for this bundle is a match on the Man ID's in each area, i.e., $q_{12} = q_{22}$. q_{12} is the property *Man ID* in the first area of the bundle; q_{22} is the corresponding property in the second area of the bundle.

$F_1[q_{12} = q_{22}, H(q_2, DW), OP]$ is a function of a bundle which maps each line of the bundle into a single point for each employee. The definition of F_1 is:

$$F_1 = \begin{cases} q_1' = q_{27} + q_{15} * q_{26} \\ q_6' = q_{20} + 1 \\ q_9' = q_{15} * q_{26} \end{cases}$$

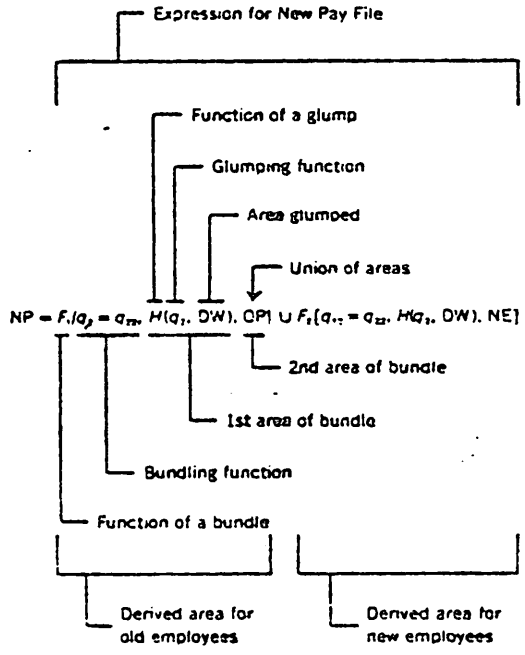


FIGURE A Identification of terms of equation (1).

The subscripts of the q-primes denote the property in the new area which is being defined by the equation. The first subscript of each unprimed q indicates the area of the bundle and the second subscript identifies the particular property. Thus, q_{27} is the property *Total Salary* for points in the *second* area of the bundle.

Each of the nonstated q-primes is understood to have the same value as the corresponding q in the last (in this case, the second) area of the bundle. Thus,

q_1' = PF	q_3' = Name	$q_8' = \Omega$
q_2' = Man ID	q_4' = Rate	$q_6' = \Omega$

while the remaining q-primes are defined by F_1.

Similarly, a second function of a bundle is defined for the area set $[H(q_2, DW), NE]$:

$$F_2[q_{12} = q_{22}, H(q_2, DW), NE]$$

$$\text{where } F_2 \text{ is defined by } F_2 = \begin{cases} q_1' = PF \\ q_7' = q_{15} * q_{26} \\ q_6' = q_{20} + 1 \\ q_9' = q_{15} * q_{26} \end{cases}$$

and the remaining q-primes are understood to be taken from the New Employee File:

q_2' = Man ID	q_4' = Rate	$q_6' = \Omega$
q_3' = Name	$q_8' = \Omega$	

Now two areas of up-to-date pay information have been derived. All that remains is to join them by the set operation *union* to form the New Pay File.

Figure 25. Information Algebra Language Example

The process or computation can also be specified and the incidence matrix of data on processes can be obtained. The matrix permits analysis for grouping of processes and data to reduce the volume of flow.

5.2.1.5 Systematics

Systematics [Gri 66] is a language solely concerned with techniques and concepts useful to system analysts in designing information models to meet the user's requirements. It is a tool for specifying solutions to information systems problems. Neither a compiler to convert the specifications into code nor analysis of the requirements is intended.

The principal features of Systematics are:

1. Specification of alternative conditions: all the potential system conditions are tabulated specifying in each case what action must be taken. These actions are expressed in a tabular form similar to decision tables.

2. Definition of qualities: for each data item, the variability and range possible values are specified.

3. Classification of information: each item of information is classified according to its role within the system.

Classification is based on permanency, i.e., whether the data remain unchanged, updated, or destroyed, etc.

5.2.1.6 Accurately Defined Systems (ADS)

Accurately Defined Systems (ADS) [Lyn 69] was developed at the National Cash Register (NCR) Company to be used in the development of business-oriented management information systems. ADS is a technique for communication and establishment of a working discipline in the definition of objectives, criteria, and specifications for systems being designed for EDP processing.

The presentation of a specification is in five standard forms:

1. Report layout form: Specifies the output reports required and their format.

2. Input media layout form: Specifies input records, field length, and miscellaneous information about input data.

3. Computation form: Specifies how a certain system element is computed from the others.

4. History definition form: Specifies the data necessary to be retained for later use.

5. Logic definition form: Specifies the decisions to be made for various computations.

These five forms are tied together by cross reference information on each of them. Figure 26 shows the format and detail of the forms.

ADS was not processed by machines. Later, automation eliminated a great deal of clerical manipulation by providing various directories and cross reference tables. Incidence and precedence matrices permit analysis of the structure of the system. However, it remains primarily as a documentation means.

5.2.1.7 Time Automated Grid

Time Automated Grid (TAG) [IBM 71] is a general-purpose technique designed to be applicable to the design of data processing systems in the commercial environment. With TAG, the user can systematize his study effort while reducing study time and maximizing the creative utilization of personnel. The technique was originally developed at IBM.

Note: Taken from [Cou 73].

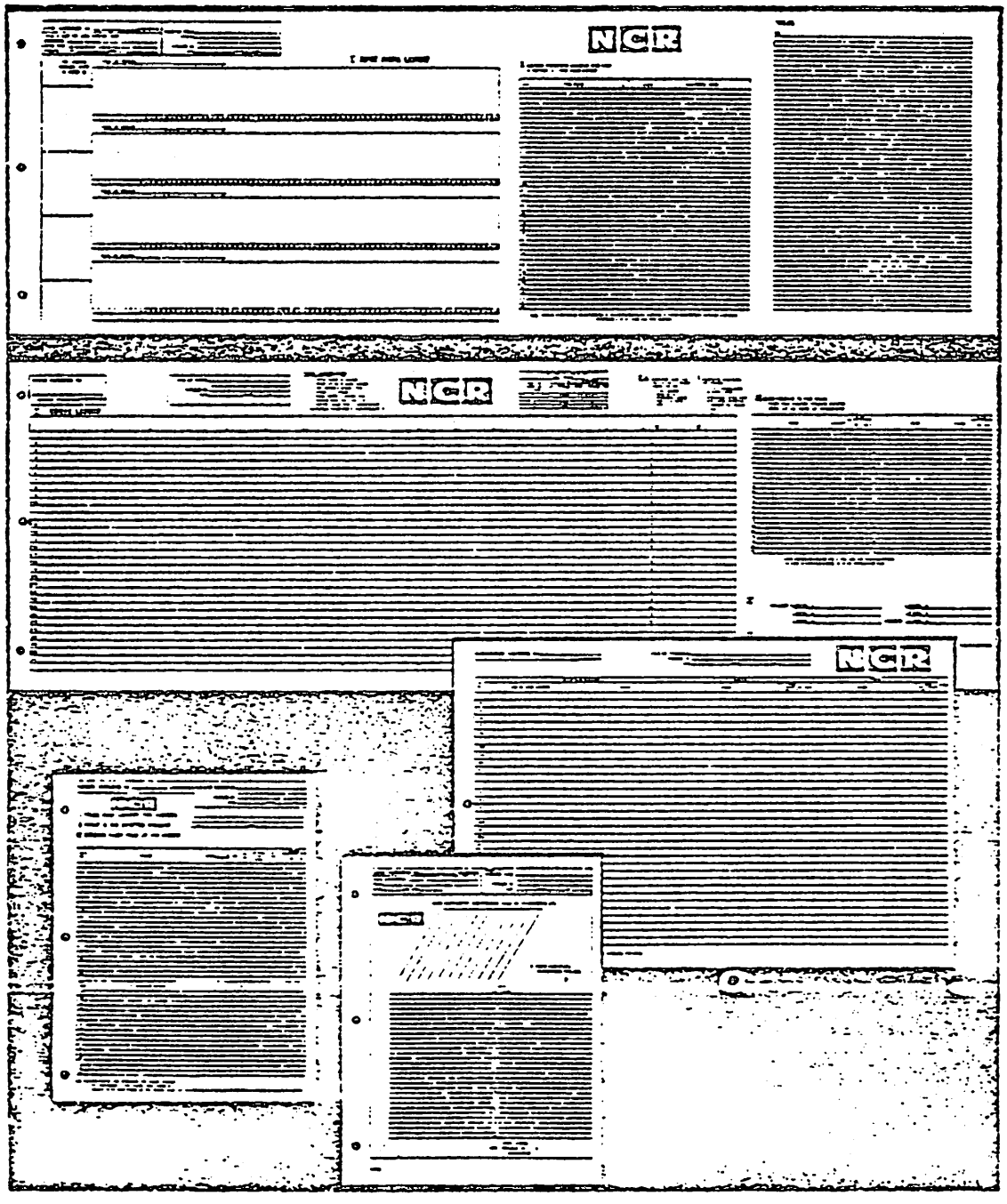
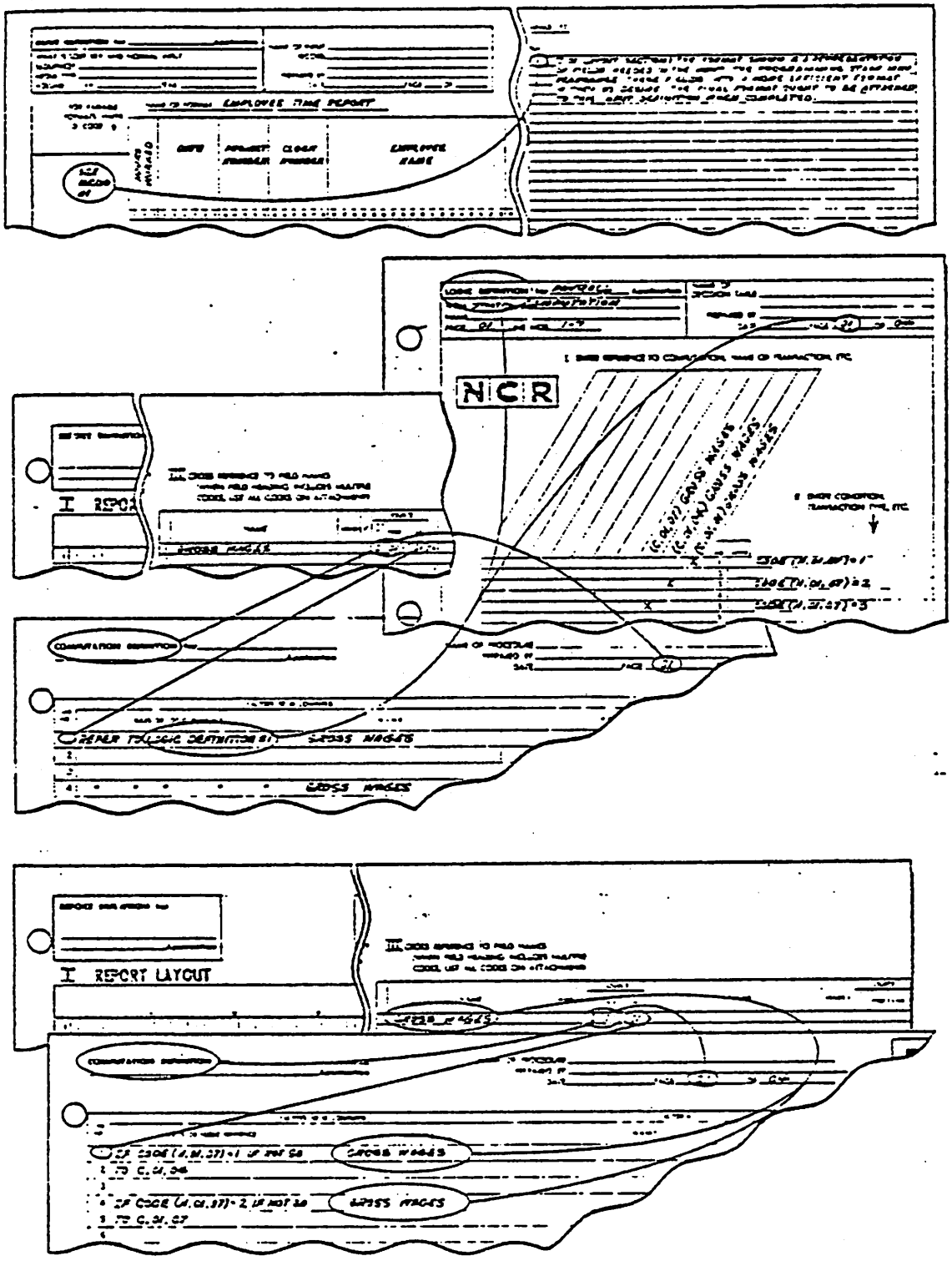


Figure 26. ADS Forms (Sheet 1 of 2)

Note: Taken from [Cou 73].



Inter-reference

Figure 26. ADS Forms (Sheet 2 of 2)

The main feature of the technique is the input-output anlaysis form. The output requirements are fed to the system, which then works backward to derive the inputs and determine at what time they are required. The definition of data requirements includes peak, average and minimum volume, and frequency.

The original system was manual and later automated in 1966. TAG generates a series of 10 reports from source file data to a variety of checking results.

5.2.1.8 Hierarchy plus Input-Process-Output

Hierarchy plus Input-Process-Output (HIPO) [IBM 73, Sta 76], developed at IBM, has been applied for system definition in other places. The HIPO method provides a graphical device for the development of a system in a hierarchical manner. It was originally developed as a documentation tool. The stress is on the presentation of relationships between the process and its subprocesses, and the flow of the input and output of each process.

The layout of the system is in graphical form. The structure of the system is depicted in a tree in Figure 27.

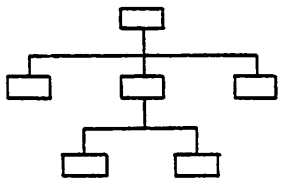


Figure 27. HIPO Structure

The immediate successor nodes in the tree of any node are its subprocesses. For each of the processes, input, process, and output are specified (Figure 28).

Note: Taken from [Sta 76].

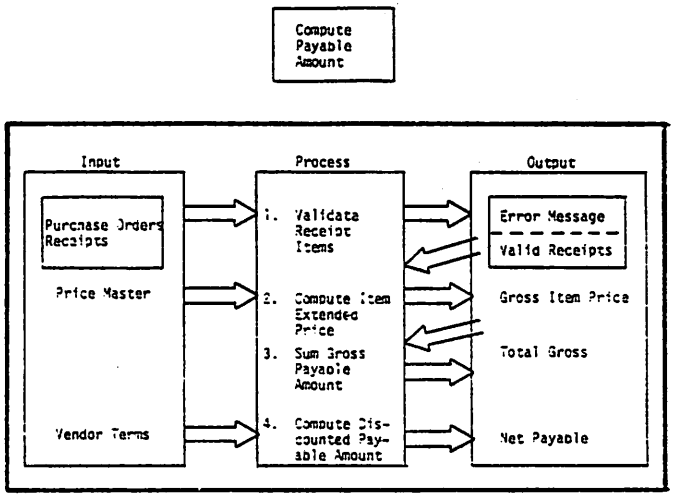


Figure 28. HIPO Specification Form

5.2.1.9 Information System Development and Optimization System

The Information System Development and Optimization System (ISDOS) [Tei 74a, Tei 71, Tei 74b, Tei 77] was developed at the University of Michigan for the analysis and documentation of requirements and the preparation of functional specifications for information processing systems. A large portion of the system is in operation now.

The specification of the target system to be built is entered into the system in a machine readable language, the Problem Statement Language (PSL). This language describes the system in terms of the relationships between the various PSL objects. The relation descriptions can be grouped into eight categories:

1. System Input/Output Flow: specifies the relation of the system with its environment in terms of the input and output entities.

2. System Structure: specifies the hierarchical relations of the basic functions or processes making up the target system.

3. Data Structure: specifies the relationships that exist among data used and/or manipulated by the system.

4. Data Derivation: specifies how certain data objects are obtained through what processes, etc.

5. System Size and Volume: specifies the loading, amount of data, etc.--in general, those factors that influence the volume of processing required.

6. System Dynamics: specifies certain performance, timing, etc., characteristics of the target system to be satisfied.

7. System Properties: specifies the classification or type of entities to which certain system elements belong.

8. Project Management Aspects: maintains information on the author, version, schedule, etc.

All this information is maintained in a system data base in the form of binary relationships between system elements. For example, if A and B are input and output, respectively, to a process P, and A, B, P are each decomposed into a_1, a_2; b_1, b_2, p_1, p_2; then their internal representation conceptually is as shown in Figure 29.

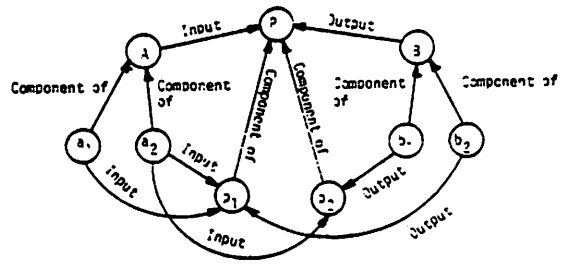


Figure 29. ISDOS System Element Relationships

PSL statements are machine processible, and the relations specified can be checked by a Problem Statement Analyzer. Both static and dynamic analyses on the target system representation can be performed.

Statis analyses include some bookkeeping functions and checking for consistency of relations. Dynamic analyses include timing and deadline checking, i.e., whether the system will meet those requirements. There are certain analyses, classified as volume analysis, that deal with those aspects regarding the frequency of use of certain data items and the amount of data traffic (e.g., between main and secondary storages).

The system is capable of automatically generating hard-copies of documentation for human references. The various reports are: Data Base Modification Reports, Reference Reports, Summary Reports, and Analysis Reports.

ISDOS is currently fully operational. Although there are limitations on the applicability of the system for the development of systems with complex interactions, ISDOS has demonstrated a viable approach. Extensions of some of the concepts in ISDOS can be found in SREM [Bel 76b] and Konynski's Model of System Analysis [Kon 76].

5.2.1.10 Systems Optimization and Design Algorithm

The Systems Optimization and Design Algorithms (SODA) [Nun 71] was developed at Case Western Reserve University. The objective of this system is to provide a computer facility for the system designer to specify his design and performance optimization on alternative systems based on the specified information. SODA is not so much concerned with the completeness or correctness of the requirements as with the engineering of an efficient system from its problem statements. An assumption is made that the problem designer is able to accurately identify the requirements.

Requirements are stated in a problem statement language. Problem statements include data description, processing requirements, and operational requirements. The data description is defined by "elementary data sets" and "data sets." Processing requirements specify the formulas for transforming inputs to outputs via processes, and the operational requirements specify the volume, timing, and frequency of input and output characteristics. The language also provides the capability for handling changes in the requirements.

The other components of the system (Figure 30) are the Problem Statement Analyzer (PSA), generation of Alternative Designs (ALT), and Optimization and Performance Evaluation (OPT).

5.2.1.11 Business Definition Language

The Business Definition Language (BDL) [Gol 75] was developed to provide a language for the noncomputer-specialist user in a business institution. Language primitives are designed to be similar to the concepts a would-be programmer should be familiar with. The basic elements of the language are:

1. Steps, which are used to represent the basic elements in the bureaucracy, such as departments, sections, and clerks.

2. Documents, which represent the paper that the steps consume and produce.

Note: Taken from [Nun 71].

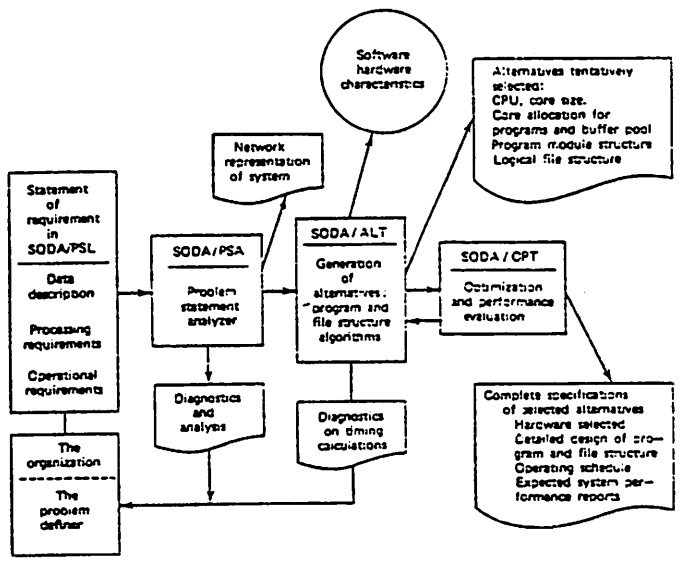


Figure 30. The Systems Optimization and Design Algorithm

3. Paths, which represent the established communication links between the various operating entities and along which the documents flow.

The user can directly specify his information processing need in terms of the graphical interrelationship of the elements. An example is shown in Figure 31.

A step, e.g., BILLING in the example, can be elaborated individually, as shown in Figure 32.

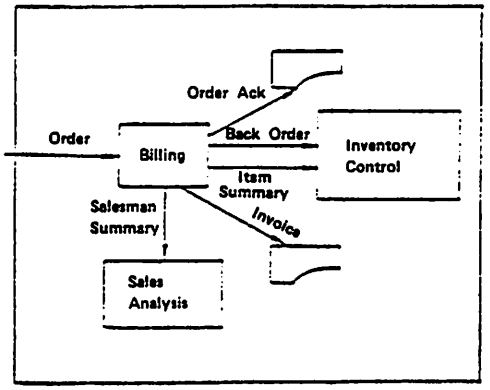


Figure 31. Business Definition Example

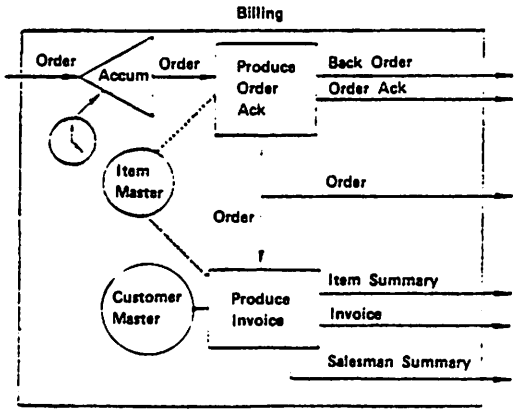


Figure 32. Elaboration of Business Definition Language Example

These specifications, or diagrams, are automatically processed to produce a machine executable program to perform the functions. There is no concern regarding the efficiency of implementation because it is assumed that all the process steps at the lowest level are implemented by a few standard procedures in a straightforward manner. Because of the automatic programming feature, the class of applications is necessarily limited to small ones.

5.2.1.12 Ho and Nunamaker

The principal objective of Ho and Nunamaker in [Ho 74] is to provide a language to state information requirements of the system. This is envisioned as the first step towards automatic construction of information processing systems. Hence, a consistent and rigorous formalism capable of being machine processed is the main criterion of the language.

The language specifies the following aspects of an information system: (1) logical data structures and their identifiers, (2) time and volume parameters, and (3) processing logic. Data are structured via the concepts of item, group, record, file, and data base; and their relationships are specified using a algebraic/set theoretic formalism. Explicit and precise analysis on the problem statements can be specified and automatically processed on the machine. The language and notation, however, are very complex, with a great deal of effort required to master them. The mathematical orientation defeats the intention of explicitness and general comprehensibility.

5.2.1.13 Bridge and Thompson

Bridge and Thompson [Bri 74] proposed a scheme intended to be used to augment the scope rules of block structured languages. Specifications of the different types of uses of data in a program, particularly concerning the flow of data among calling and called modules, are dealt within this scheme. Figure 33 summarizes the specification of data flow into and through modules and the specification of data manipulation.

Note: Taken from [Bri 74].

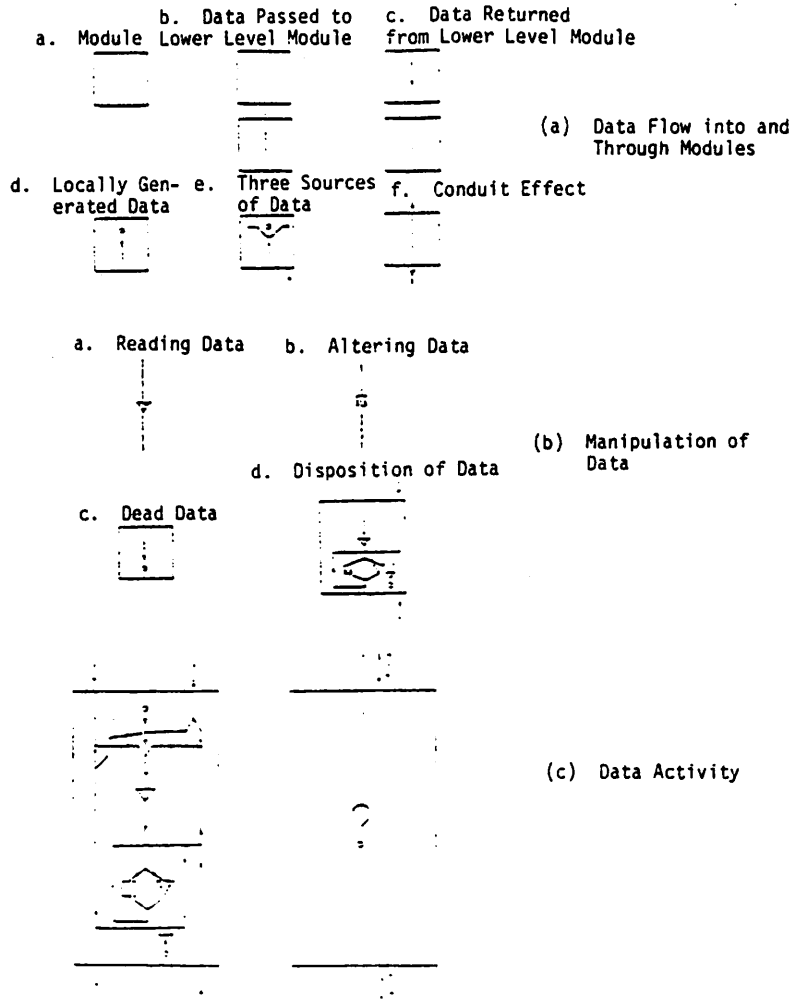


Figure 33. Data Flow Specification

Every data item used in any program module is used in one of the ways shown in the figure. A specification scheme is devised to specify the use of each variable in a module. The scheme is based on a three level categorization of all the variables used (Figure 34).

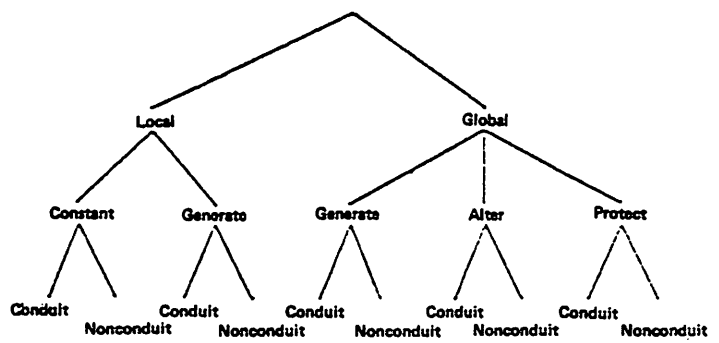


Figure 34. Three-Level Categorization of Variables

The conduit attribute is attached to a variable if it appears in a module, but direct use is not made of it (i.e., the module serves only as a pathway). In the data declaration section of each module, a variable must be categorized according to its intended use.

5.2.1.14 Structured Design

Structured Design is a set of software design philosophy and guidelines [HUG 75] based on the research and concepts of Constantine, et al. [Ste 74]. It is currently applied by the software development group at Hughes Aircraft Company, with emphasis on applications in the military real-time systems.

Central to the methodology is a "design process" and the visual aids of the structure and bubble charts. The design process is divided into three phases: the First Cut Design, Intermediate Design, and Final Design. At a purely

conceptual level in the First Cut Design, only the main system function and data flow are considered, ignoring hardware, data implementation, etc. In the second phase, consisting of partly conceptual and partly real world considerations, the actual data base, system control, etc., are given considerations. The last phase primarily addresses real-word issues. Codible modules are produced; the data base is finalized; the design is packaged for hardware; and coding documentation is produced. Throughout all phases, a set of guidelines is prescribed to guide the designers. Other than that, however, the designer has to rely heavily on his experience, familiarity with the system, and ingenuity.

The bubble chart and the structure chart, tnough an integral part of the above three-phase design process, can be understood by themselves.

The bubble chart shows the conceptual data flow of the system (Figure 35). The "bubbles" represent basic system functions and the labelled arc indicates the flow of information among these functions.

The structure chart shows how the bubble chart is to be implemented by a hierarchy of modules. The control/data flow, conditional call, and iteration can also be indicated in the chart. Figure 36 shows the corresponding structure chart for the bubble chart of Figure 35.

These two charts may help the designer to manually evaluate his design, based on the design guidelines and other criteria such as the understandability and complexity of the design. There are no considerations to automatically analyze properties of any bubble or structure charts. Other than those general properties discussed in paragraph 4.3.1, the charts do not contain additional information of a design amenable to automatic analysis.

5.2.2 Control-Dominant Systems

Note: Taken from [HUG 75].

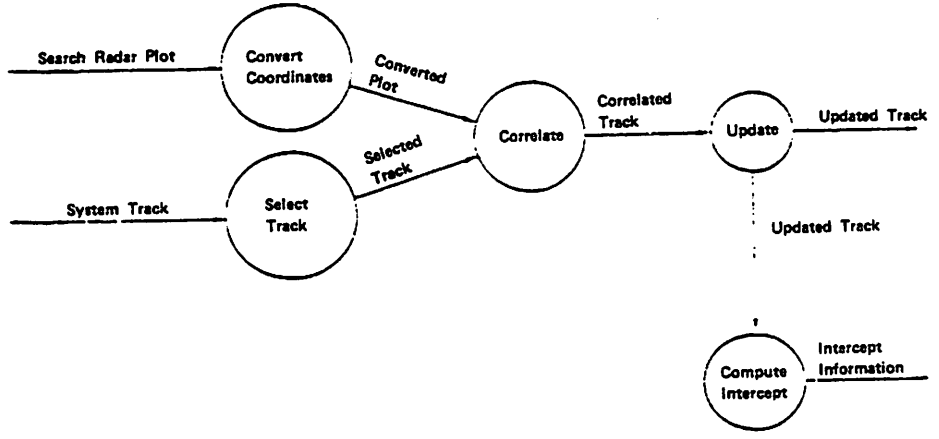


Figure 35. Bubble Chart Example

Note: Taken from [HUG 75].

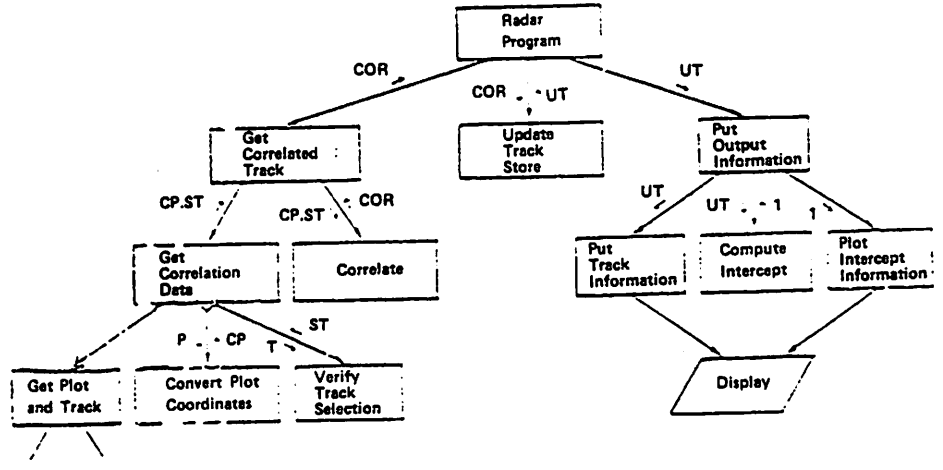


Figure 36. Structure Chart Example

5.2.2.1 LOGOS

LOGOS [Ros 72] was developed at the Case Western University to provide an aid to the development of computer systems. It imposes a discipline for the description of the system in hierarchical refinement while intermediate steps can be checked for certain desirable properties.

The key feature of LOGOS is to describe a system in two graphs: the control graph and the data group (Figure 37). The control graph is based on familiar graph models of [Pet 62, Kar 69, Hol 70]. The control graph consists of two types of nodes: the squares are control variables and the remaining symbols are control operators. The graph represents the static template of a task. The data graph consists of two types of nodes: the hexagons are data operators, and the squares are data elements. The graph represents input and output relations. The nodes in the graph can be decomposed into a complex structure as the system is developed. This provides a scheme for stepwise development and local analysis.

Uninterpreted and interpreted types of analyses can be performed. Uninterpreted analysis procedures performed on these graphs are based on graph-theoretic techniques and properties such as parallelism and determinacy developed in [Kar 69].

Interpreted analysis deals with the correctness of the algorithm used in implementing the nodes in the graph. These are not readily automatible with the present techniques.

LOGOS pays particular attention to the control flow of a system in addition to the information flow, which is the focus of most other systems in this category. However, the symbols and language developed in LOGOS are too specialized and implementation oriented, and as such, it is not very applicable for general requirement statements.

Note: Taken from [Ros 72].

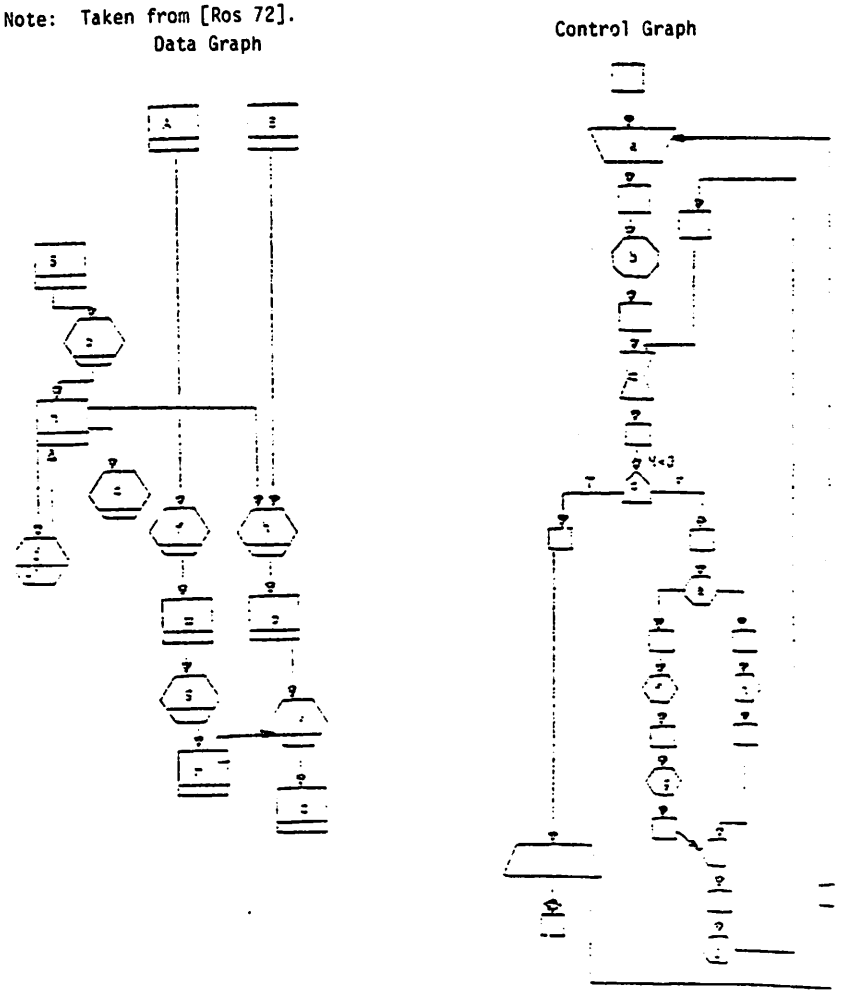


Figure 37. LOGOS Example

5.2.2.2 CSC Threads

CSC Threads was developed by Computer Sciences Corporation [CSC 73] to improve the effectiveness of the system development project--mostly oriented towards nonreal-time systems.

A system is viewed as a collection of subsystems--each subsystem a collection of tasks; and each task a collection of program units. In response to a system stimulus, a number of subsystems are invoked in a certain sequence. This sequence is referred to as a thread. Figure 38 shows an example of a thread through the Command and Control Subsystem.

Note: Taken from [CSC 73].

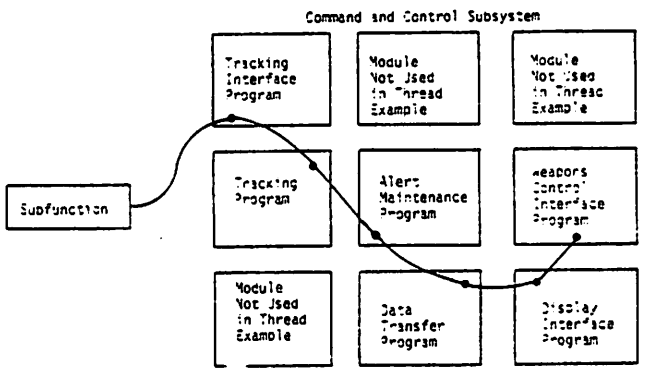


Figure 38. Thread Example

A thread must be specified for each functional requirement.

Threads are identified in three levels: system level threads, subsystem level threads, and task level threads to include software, hardware, manual operations, and their combinations in the system. Figure 39 shows this concept of a hierarchy of threads.

Note: Taken from [CSC 73].

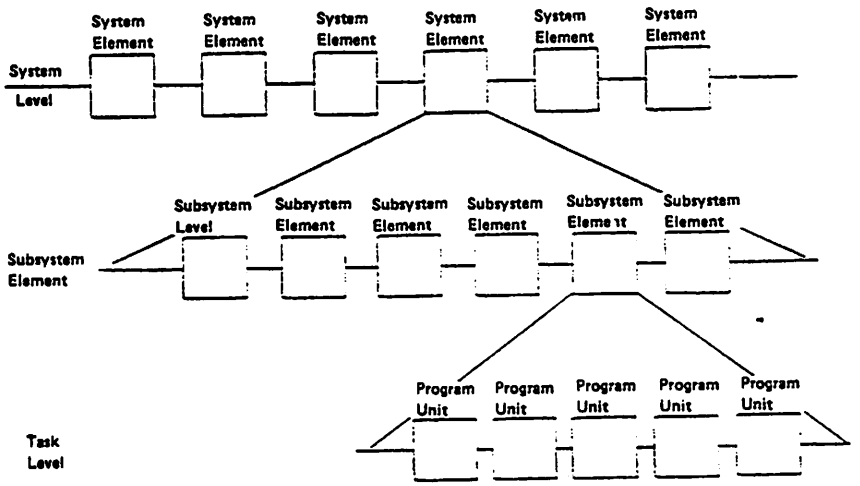


Figure 39. Hierarchical Thread Concept

As indicated before, threads are used for management of system development projects. Their integration to the system development steps is shown in Figure 40.

This technique is very limited in that it restricts the system development to identify all the subsystems and tasks, etc., and then to identify all the processing paths, i.e., threads. For complex systems and real-time operations, the number of threads is usually too large to be handled effectively. TRW, in developing the SREM, discovered this problem early in their research; and a modification of the thread concept was made, resulting in the development of R-nets.

Note: Taken from [CSC 73].

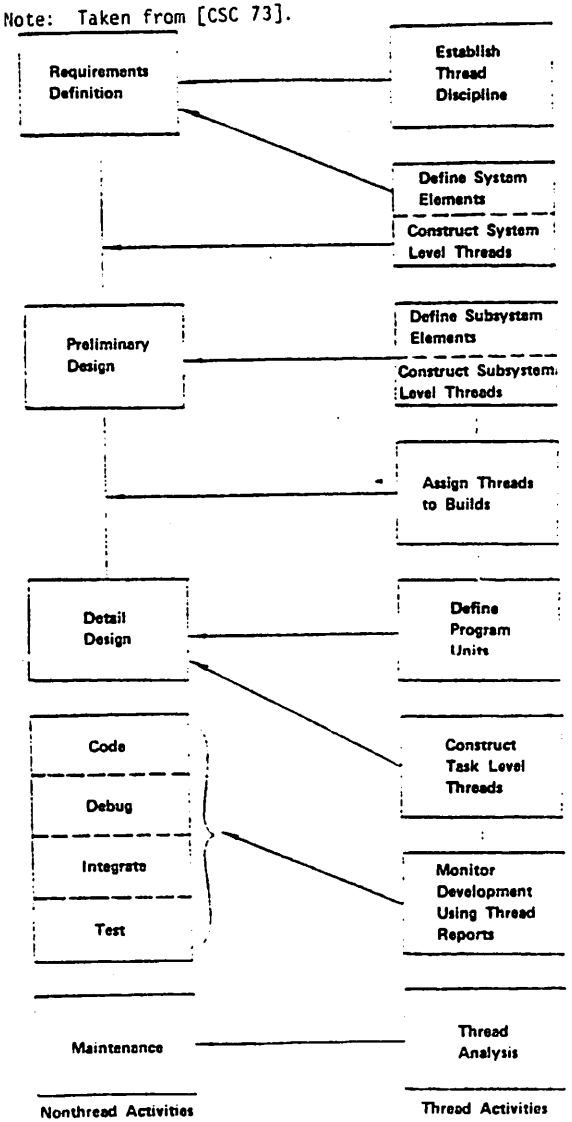


Figure 40. Thread Usage in System Development

5.2.2.3 Requirements Nets

The requirement net (R-net) representation of system requirements was developed at TRW [Alf 76a,b, Bel 76a,b], partially derived from the CSC threads [CSC 73]. A thread identifies the sequence of operations of the system in reaction to a specific stimulus. Specifying the functional requirements would theoretically involve identifying all the possible threads. The number is enormous for any nontrivial system. This difficulty is resolved by representing these threads as a structure known as the requirement net. The constructs used in the net are shown in Figure 41 and briefly explained below.

▽ Event--specifies the condition under which the net of processing actions is activated.

⬡ External interface--specifies the interaction between the system and the external environment.

○ Validation points--performance requirements such as timing and accuracy are specified and validated during simulation.

▭ Alpha--represents a basic unit of processing action.

▭ Subnet-substitutes for a net to be expanded.

System data base--specified with the information of which the alphas will reference and modify a particular data base. The dotted arrows indicate the potential accesses.

"AND" nodes--both rejoining and nonrejoining AND node, in which the processing sequences are to be followed in parallel. All actions will be delayed at the rejoining points until all branches have been finished and then proceed to the next alpha.

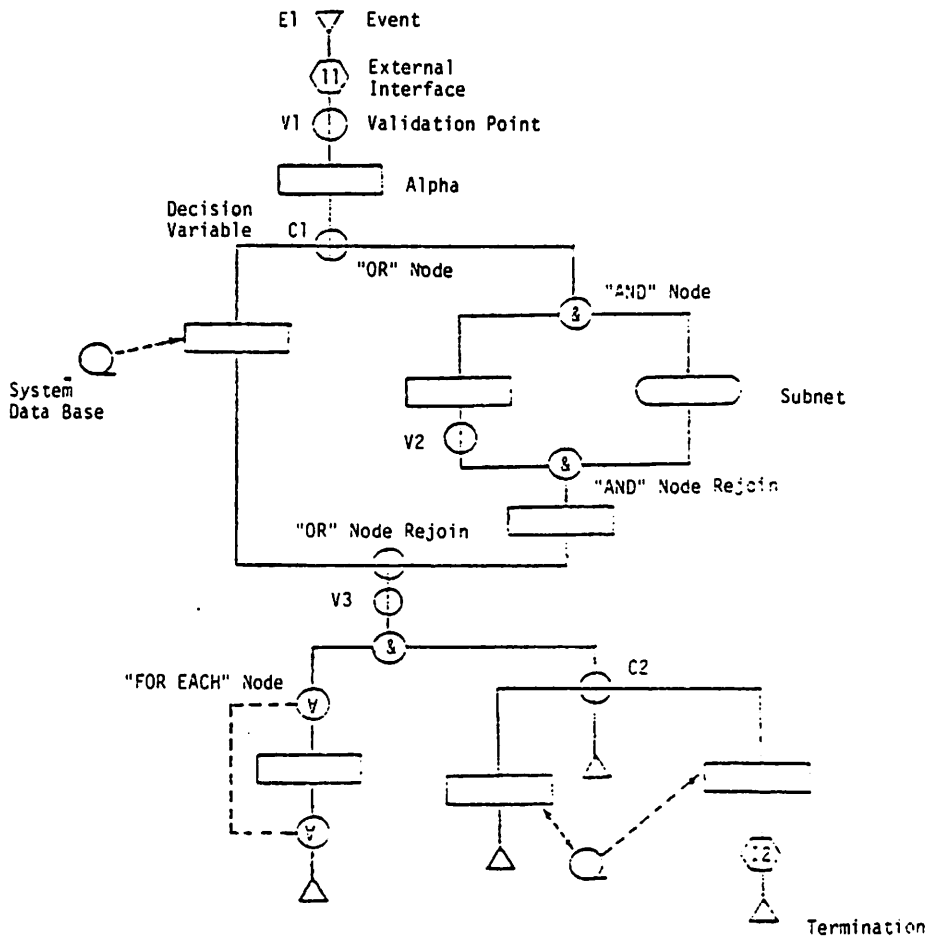


Figure 41. R-net Terminology

"OR" node--a condition is attached to each path at an OR node. The one path with a true condition is processed; the others are ignored. Each OR node has an OTHERWISE path that is processed if none of the conditions are true.

"FOR EACH" node--one path is associated with the node, which is processed once for each element of a set of data processing system entities that is currently present.

Termination--indicates termination of a processing path.

The R-net specification, together with other information such as the hierarchical relation of data elements, traceability information, and other management information, constitutes the complete process performance specifications. These are maintained in a data base known as the Abstract System Semantic Model (ASSM). A set of analysis tools is designed to perform various analyses on this data base. The composite is the Requirements Engineering and Validation System (REVS) which is shown in Figure 42.

Note: Taken from [Bel 76b].

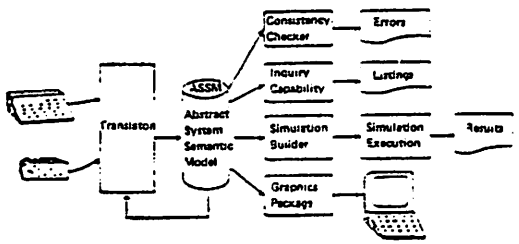


Figure 42. Requirements Engineering and Validation System

There are three classes of analysis tools besides the basic language translator and data extraction facilities from the data base.

The first class is a graphic capability with which an R-net can be generated or modified interactively with a graphics terminal. This provides a more natural graphic form for the requirement engineer to work with, rather than using a one-dimensional textual language. Extensive functions for positioning, selecting, and zooming are available.

The second class is static analysis, which checks the structure of the R-nets for consistency and correctness, such as proper branching and rejoining of paths. The data flow of the R-nets is also analyzed, obtaining information on the life time and use of individual data items.

The third and most extensive class of analysis is simulation. In REVS, discrete simulation models are generated automatically from the ASSM by translating the R-nets into Pascal code. Simulation execution is also controlled by the system with its initiation, data gathering, and data analysis functions.

5.2.2.4 The Finite State Machine (FSM) Approach

This approach was developed by the Aeronutronic Ford Corporation [Sal 76] as a research and development project. The salient feature of the methodology is to identify four essential elements of a system in the process of decomposing system requirements into data processing requirements. These are the control, functions, data, and functional flows. Each of these is modeled by a structure that allows formal mechanical analysis. The decomposition methodology is illustrated in Figure 43.

5.2.2.4.1 Control. The control of a data processing system is the mechanism that activates system functions in the desired sequence; control is modeled as an FSM. An arbitrary FSM can be reorganized in a special form, the structured FSM, which is a hierarchy of FSMs whose only cycles are loops. The structured FSM allows certain automated formation checks, such as: (1) consistency--for a given state and any given input, no more than one transition is possible; (2) completeness--for any given state and any given input, a transition is defined; and (3) reachability--there is a path to a given state of the FSM from the START state and a path from it to an END state.

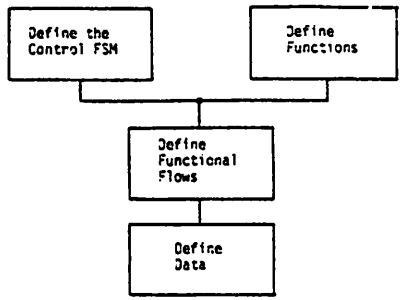


Figure 43. System Decomposition Methodology

5.2.2.4.2 Functions. The interdependence among system functions is modeled by a directed graph whose nodes represent the functions, and an arc from function A to function B indicates that function B requires an input generated by function A. The relation is referred to as function A impacting function B. The directed graph model readily lends itself to connectively and similar graph theoretic structural analyses. Properties such as reachability and completeness with respect to the system requirements can be verified. In short, questions of the following forms are addressed by these analyses:

- What functions depend on a given function?
- What functions influence a given function?
- Are two functions independent, so that there may be reason to implement them in distinct, parallel processors?
- Are there cycles of mutually dependent functions?

5.2.2.4.3 Functional Flow. A functional flow is the output of one or more states of the control FSM, causing an ordered execution of data processing

functions and non-data processing subsystems. Specification of functional flows can be checked for: (1) consistency--if each impact in a functional flow corresponds to an impact in the system function structure; (2) completeness--if each data processing function is contained in at least one functional flow; and (3) reachability--if each functional flow is the output of at least one state in the control FSM.

5.2.2.4.4 Data. The data specification of the data processing subsystem consists of indicating, for each data object, the functions to which it is an input and from which it is an output. This is used to relate the several functions within a functional flow, to drive the control mechanism, and to link control to the functional flows.

Data elements are said to be consistent with the function structure if each element of data has both a source and a destination, and if every impact defined by the data is an impact in the system impact relationship. Data elements are said to be complete if each impact in the system impact relationship is defined by the data. An element of data is said to be reachable if its value is defined before being used. Mechanical tests can be provided to measure consistency and completeness of data. An open question is that of how to determine reachability.

It is the intention of the methodology that the integration of these elements with the verification of the previous properties should guarantee at least a consistent and well-formed definition of the data processing subsystem.

5.2.2.5 The Verification Graph (VG) Method

The specification methodology, developed by Computer Sciences Corporation [Bel 76], consists of three sequential phases as shown in Figure 44: (1) decomposition, (2) static analysis, and (3) dynamic analysis. Each phase is executed for each level of requirements in the specification. For the first level of requirements, the system specification is decomposed and then statically analyzed. The first two phases are iterated until a consistent set of system

Note: Taken from [Bel 76].

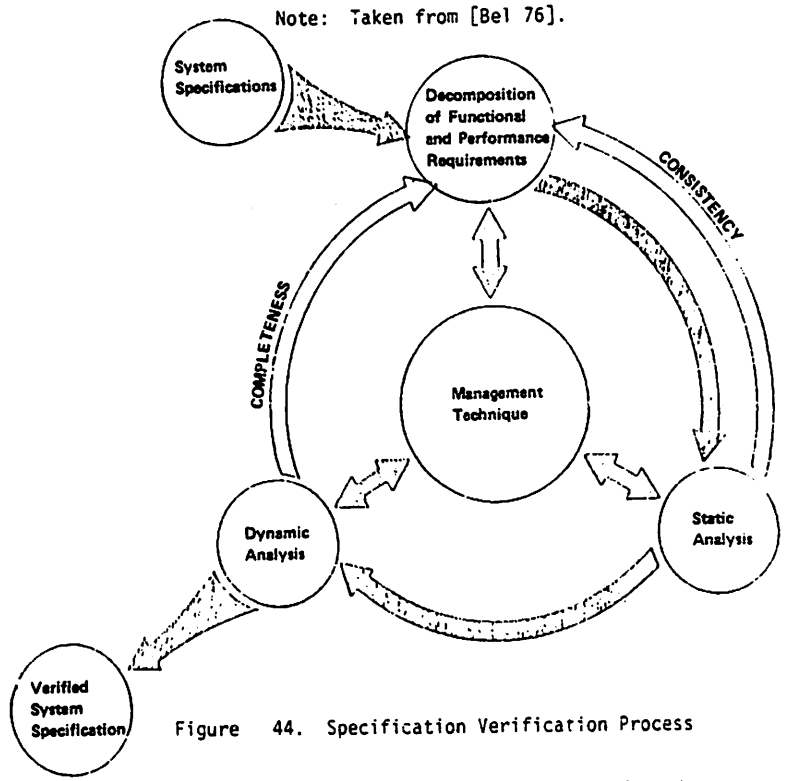


Figure 44. Specification Verification Process

requirements exists. Finally, dynamic analysis is performed, and the three phases are iterated until a verified, validated, and complete set of requirements exists.

The decomposition of system requirements is achieved through the specification of the interaction of system functions using the notion of a Decomposition Element (DE). The DE represents a functional requirement and associated performance characteristics from the system specification. A functional requirement includes a description of a function, the information or action

input to the function, and the information or action output from the function. This is expressed as:

Functional Requirements =

Input	Transformation	Output
Stimulus	Functional Argument	Response
	Performance Characteristics	

A stimulus is the initiator of an action; it is the input to the functional argument. The functional argument is the action or transformation performed as a result of the initiating action of the stimulus. A response is the result or output of the functional argument. The performance characteristics from the system specification are associated with each stimulus, functional argument, and response. The special form for entering the information of each DE is shown in Figure 45.

The VG is automatically generated when the DEs are specified. It is represented as a directed graph. The directed arcs in the VG represent the stimulus/response pairs of the requirements in the specification as decomposed in DEs. Each functional node of the graph represents functional requirements in the specification. Decision nodes represent either branching requirements or logical requirements.

The directed graph model is amenable to familiar analyses such as the following:

- Consistency. Each node in the graph must be either functional or logical; each arc in the graph must be either local or global; and the set of all nodes and arcs in the graph must be connected.

- Traceability. The global stimulus arcs and global response arcs in sublevel graphs must map one-to-one in quantity, direction, and value to simply connected arcs in the next-higher-level graph.

Note: Taken from [Hel 76].

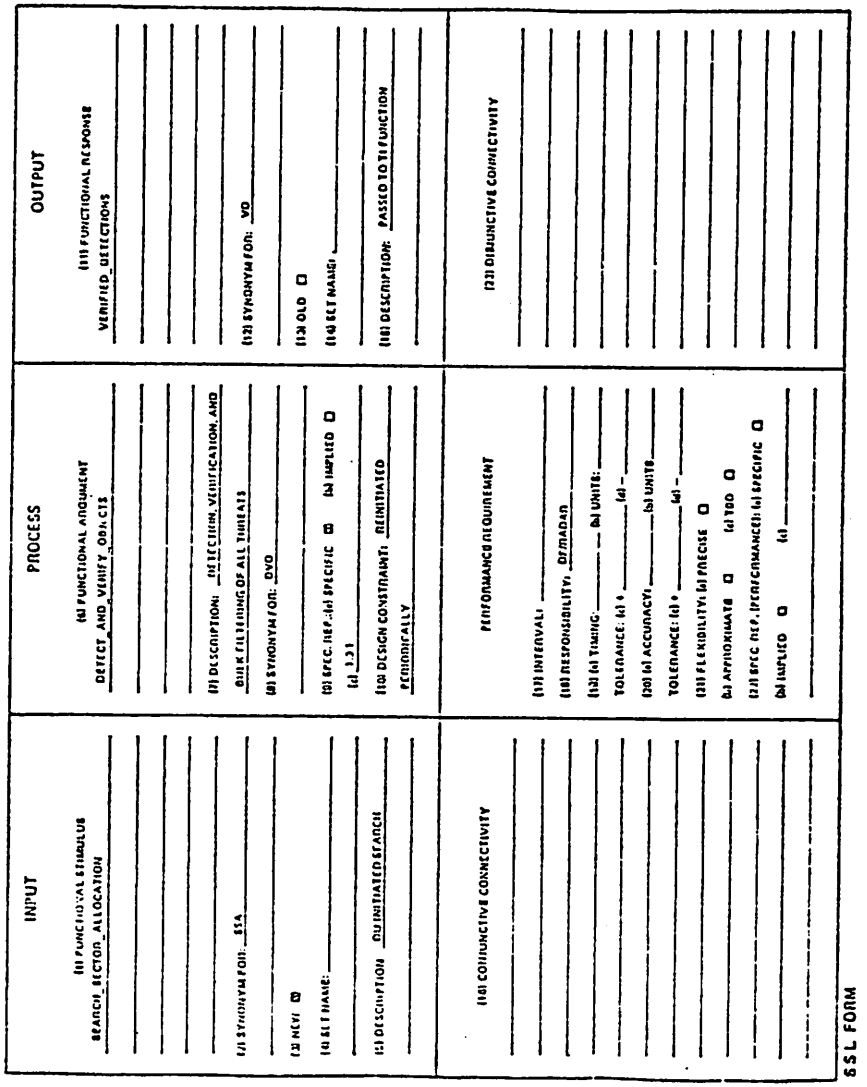


Figure 45. DE Entry on a System Specification Language Form

- Completeness. All sublevel graphs must be traceable to functional nodes in the next-higher-level graph, and these functional nodes must exhaust the set at the level.

Besides these static analyses, simulation models are constructed from the specification for analysis of dynamic behavior and performance characteristics.

A mechanical translator of the DE specification and support management software has been developed for the methodology.

5.2.2.6 GRC Petri Net Approach

The Petri net approach was developed in a research program carried out by GRC [Bal 76]. Data processing subsystem requirement specification involves, among other things, the description of the operating rules of the system as conditions and events. Specification of this aspect is considered to be best done by the Petri net model.

Figure 46 illustrates how Petri nets can be used to express operating rules for a system. A Petri net is a directed graph that distinguishes two types of vertices: (1) events or transitions (represented by vertical bars) and (2) conditions or places (represented by circles).

An event is said to be enabled if all of its input conditions are satisfied (marked by a token). The behavior implied by the network is derived by a selection rule that specifies the selection of one enabled event. The selected event is then fired (a token is removed from each input place, and a token is put on each output place). Repeated application of the selection rule produces a simulation of possible event sequences.

Petri nets are adequate as descriptive devices for the operating rules of a system. There are known analytic techniques to identify forms of undesirable behavior that may be allowed by a particular formulation of a net (paragraph 4.3.2).

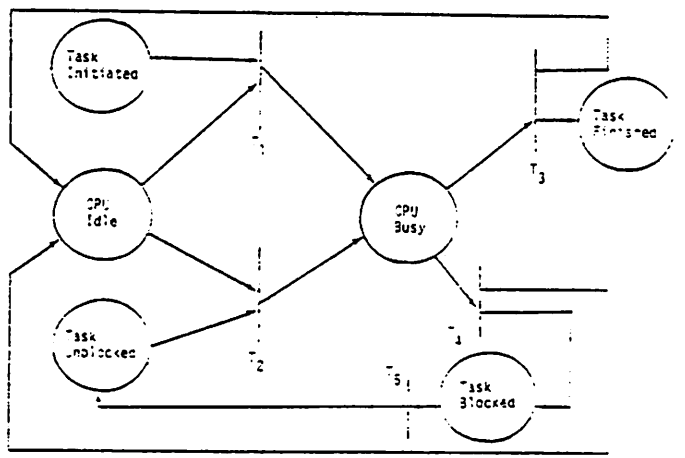


Figure 46. Example of Petri Net Model

The design methodology using the Petri net allows for the successive refinement of detail in the performance specifications. This capability introduced the need for techniques to verify that refinements or decompositions of specifications were consistent with earlier high-level specifications. Formal verification is made possible by the fact that a Petri net can be represented as the conjunction of conditional expressions describing the condition for firing individual transitions. For example, transitions T_3 and T_4 of Figure 46 are captured by:

IF CPU BUSY THEN [(TASK FINISHED AND CPU IDLE) XOR
(TASK BLOCKED AND CPU IDLE)]

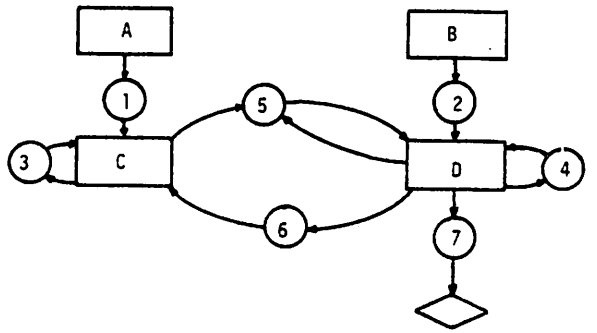
The formal logic translation of the Petri net permits the use of automated theorem proving techniques to verify the consistency of the decomposition.

Although this model is successful with respect to operating rule specification, performance requirements are not adequately described or evaluated by the network. Models, in the form of procedures, are then used to estimate performance. Constructing a model of the system requires that the Petri net tokens be assigned "attributes." Attributes can be accessed by "transition procedures" associated with transitions. Thus, in the example of Figure 46, transitions T_1 and T_2 would access the attributes of the tokens occupying the "Task Initiated" and "Task Unblocked" conditions. For example, if priority is the criterion being modeled by the selection rule, then selection would be based on the token with the highest priority (determined by the transition procedures).

Simulation, as the chief means for performance evaluation, requires a model of the environment and the performance achieved by the implementation of the rules. The designer must select models with greater fidelity than those that have been employed by the systems engineer, but ones that do not go as far as prototype software. The Petri net serves to relate the models to the specified operating rules to form a simulation. The Petri net description of the requirements can be converted automatically into the framework for an event-based simulation. The addition of models associated with events and conditions, and the environment simulator, form a simulation of the system that allows the analyst to verify that the stated performance requirements meet system objectives.

5.2.2.7 Computation Structures and Performance Modeling

Taylor Booth, et al. [Sho 75], introduces a software performance modeling scheme based on a graph model known as a computation structure. A computation structure consists of two directed graphs called a data flow graph and a precedence graph. In the data flow graph, the relations between the storage cells and the set of operators for processing the information are specified. For example in Figure 47, the circles, labeled 1 through 6, are operators and boxes A through D are storage cells. The directed arrows represent the input/output relation of a storage cell to the operator. In particular, operator 5 requires storage cells C and D as inputs and produces D as output; the decision element 7 is based on information in storage cell D.



N = {S, P}
S = {A, B, C, D} Storage Cells
P = {1, 2, 3, 4, 5, 6} Operator
{7} Decision Element

Figure A-47. Data Flow Graph

The precedence graph specifies the flow of control in a computation. Figure 48 shows an example based on the data flow graph of Figure 47.

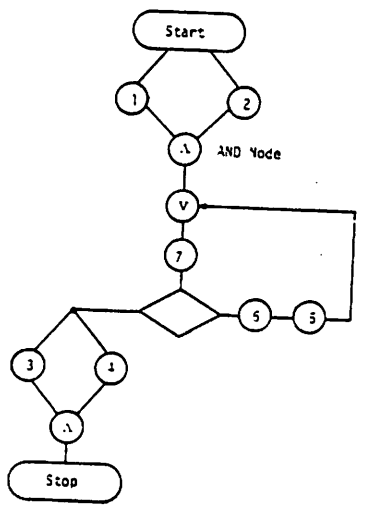


Figure 48. Precedence Graph

The semantics of the graph are similar to the other graph models.

The third element of the model is the set of performance parameters for each operator. Thus, for each operator i, a number of cost factors are assigned, e.g., t_i--execution time, m_i--memory requirement, etc.

To estimate or predict the performance of the computation, the input must be modeled. Let $x \in X$ be the input space; then $p(x)$, the probability distribution of the inputs is appropriately modeled, and $c(x)$, the cost for input x is calculated from the computation structure. Then the expected cost of the system is given by:

$$c = \sum_{x \in X} p(x)\, c(x).$$

or

$$c = \lim_{\text{length}(x) \to \infty} \left| \frac{c(x)}{\text{length}(x)} \right|$$

Calculating a cost c from the basic cost parameter t and m can be specified by a set of rules. For example, for sequential execution of two operators:

$$t_{12} = t_1 + t_2$$

$$m_{12} = \max(m_1, m_2)$$

and $c_{12} = f(t_{12}, m_{12})$, a predefined function of weighting time and space requirements. Or, for parallel execution of two operators:

$$t_{12} = \max(t_1, t_2) + t_{overhead}$$

$$m_{12} = m_1 + m_2$$

$$c_{12} = f(t_{12}, m_{12}).$$

The modeling technique is a simple tool; however, it can be very effective if the environment can be modeled (especially the input distribution). Under these situations, the performance of the specified system can be evaluated. Furthermore, additional cost or performance parameters can be introduced as long as the combination rules for a computation structure can be specified. For most other parameters, this may turn out to be difficult; an example may be the leakage factor of detection of threatening objects in a BMD system. The two dimensions of difficulty are the assignment of a factor for each basic system function and the combination of the individual factors in a sequence of system functions. Performance specification and analysis beyond simple aspects remain a difficult problem.

5.3 SPECIFICATION TECHNIQUES FOR DETAILED SOFTWARE DESIGN

5.3.1 Sequential Processes

5.3.1.1 Informal Description

The effect or function of the procedure is described using a natural language and whatever formalism or discipline the author thinks necessary [McG 75].

Comprehensibility may be good; however, ambiguity and the lack of formalism is a major problem. An example of this technique is shown in Figure 49.

The merit of the approach is for documentation purposes and enhancement of understanding. Some manual verification may be possible. However, the major effect that cannot be overlooked is that merely requiring a programmer to work through the exercise of producing such a specification should increase understanding and productivity substantially.

5.3.1.2 Parnas' Module Specification

This specification [Par 72a] belongs to both the algorithms and data-type specification categories. Parnas' notion of a module includes abstract data type and a collection of related functions. For example, the module "stack"

Note: Taken from [McG 75].

```
Function Name:  GET_A_CHAR, a PL/I function procedure

Domain:         A sequence of characters from a logical record; each character
                is

                (1)  an end-of-line, or
                (2)  a delete-line, or
                (3)  a regular

                character.  The last character in the sequence (i.e., the end
                of the logical record) is an end-of-line character.

Range:          The function returns a character of type (1), (2), or (3) with
                each call.

Rule-of-Correspondence:
                GET_A_CHAR returns the next character in the input sequence
                (i.e., the first character on the first call, the second
                character on the second call, . . .).
```

Figure 49. Informal Functional Specification

includes the functions PUSH, POP, VAL, and DEPTH. Two types of functions are distinguished: V-function, which returns the value of the states and O-function, which causes state transition.

The specification of a module consists of specifying, for each function, the following: (1) possible values, (2) initial values, (3) parameters, and (4) effect. In the effect section, the minimal amount is stated about what would happen when the function is executed. Figure 50 shows an example of the stack specification.

Where PUSH and POP are the O-function and VAL and DEPTH are the V-function, the minimality is achieved in specifying the effect entirely in terms of the operations, types, and constant values. Mutual reference to each other is

required in stating the effect of some operations. For instance, the sequence PUSH(a); POP has to be stated either in PUSH(a) or POP.

Note: Taken from [Par 72].

```
Function PUSH(a)
Possible Values:  None
Integer:  a
Effect:  Call ERR1 if a > p2∨ a < 0∨'DEPTH' = p1
         else [Val = a; DEPTH = 'DEPTH'+1;]

Function POP
Possible Values:  None
Parameters:  None
Effect:  Call ERR2 if 'DEPTH' = 0
         The sequence "PUSH(a); POP" has no effect if no error calls occur.

Function VAL
Possible Values:  Integer initial; value undefined
Parameters:  None
Effect:  Error call if 'DEPTH' = 0

Function DEPTH
Possible Values:  Integer; initial value 0
Parameters:  None
Effect:  None
p1 and p2 are parameters.  p1 is extended to represent the maximum depth of
the stack and p2 the maximum width or maximum size for each item.
```

Figure 50. Specification of STACK Module

The notation used in [Par 72] is semi-informal and a large part can conceptually be formalized. Procedures for proving the correctness of an implementation have not yet been developed.

5.3.1.3 Formal Logic Approach

A typical example is [Goo 75], in which a system for automatically verifying the correctness of a program is described. Specification of the program takes

the form of an assertion on the ENTRY and EXIT conditions. (Figure 51 shows a sample program with specification.) The verification process is to show that the ENTRY assertion always implies the EXIT assertion if the program terminates. Additional ASSERT statements are used to assist automatic proving. From the standpoint of a programmer, only the ENTRY and EXIT assertions are needed to understand all the behavior of the procedure.

Note: Taken from [Goo 75].

```
FUNCTION LOCMAX(A:INTARRAY;I,J:INTEGER):INTEGER;
ENTRY I LE J;
EXIT (I LE LOCMAX(A,I,J))
  AND (LOCMAX(A,I,J) LE J)
  AND (A[LOCMAX(A,I,J)] = AMAX(A,I,J));;

FUNCTION SORT(A:INTARRAY;N:INTEGER):INTEGER;
ENTRY N GE 1;
EXIT ALL I (1 LE I) AND (I LE N) IMP
     SORT(A,N)[I] = AMAX(SORT(A,N),1,I);
  AND APERM(SORT(A,N),A,1,N);
VAR B:INTEGER;
BEGIN
  B := A;
  K := N;
  ASSERT ALL I ((K+1 LE I) AND (I LE N) IMP B[I] = AMAX(B,1,I))
    AND APERM(B,A,1,N)
    AND (K GE 1) AND (K LE N);
  WHILE K > 1 DO
    BEGIN
      B := ASWAP(B,LOCMAX(B,1,K),K);
      K := K - 1;
    END;
  SORT := B;
END;.
```

Figure 51. Sample Sorting Program with Specifications

Although the method is formal, the assertions themselves may be difficult to understand or even to construct. Furthermore, the specification may involve specific implementation details that should not be known to the user of the procedure.

5.3.1.4 Hewitt and Smith Contracts

The term contract used in [Hew 75] has the same meaning as specification in our sense. The system devised by Hewitt and Smith is intended to give a rigorous behavioral specification of actors (processes or procedures) at various levels of detail. It embodies both algorithm and data-type specifications. We will not develop the formalism here. The following brief outline should be sufficient to convey the flavor of the system to the reader.

The syntax for a contract (in LISP language) is:

```
(➡<pattern-for-incoming-message>
    (require: <assumptions>
     use: <relevant-knowledge>
     rider-on-the continuation: <entailments>
     rider-on-the-complaint-department: <entailments-on-complaints>)).
```

<pattern-for-incoming-message> specifies the form of the input to the procedure. The <assumptions> specifies what the user assumes to be true on entry. <relevant-knowledge> specifies such information as what type of knowledge is likely to help in understanding why the contract is true. The last two items specify the conditions expected to be true after the actor is executed in normal and error situations.

There is also a provision to state "intentions" at any point inside a particular implementation of the actor, which serve as check or validation points.

The notion of "meta-evaluation," in which a process verifies that a program fulfills its contract, is also developed.

5.3.2 Data Types

5.3.2.1 V-Graph Method

Earley's V-graph [Ear 71] for data structure is constructed from atoms, nodes, and links. Each node of the graph represents a part of the data structure.

Links are labelled indicating the access paths. For example, a linked list of the atoms a,b,c is shown in Figure 52.

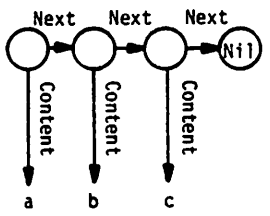


Figure 52. Linked List Structure

At the leftmost node, for instance, there are two accesses possible: obtaining the next item (NEXT) or getting the atom (a in this case). Operations on data structures can also be represented by the V-graphs. Figure 53 shows the operation PUSH on a stack.

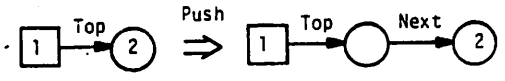


Figure 53. Specification of PUSH Operation

This graphical notation has the merit of being easily understood. Formal analyses of the graphs are possible. However, the scheme suffers from the special emphasis on access paths. If the access paths are not the chief concern of a particular data structure, they would be considered as over-specification, violating the minimality criterion.

5.3.2.2 Algebraic Axioms

The specification of a data structure is stated in a set of axioms that governs all the permissible operations involving the data structure.

Guttag et al. [Gut 76] developed a set of notations that only allows the use of the following features: (1) free variables, (2) if-then-else expressions, (3) boolean expressions, and (4) recursion. Figure 13 is a simple example using this notation, and the following specification of the QUEUE illustrates the full use of all the features (Figure 54).

Note: Taken from [Gut 76].

```
type    Queue[item]
 1.     declare NEWQ( ) - Queue
 2.          ADDQ(Queue,item) - Queue
 3.          DELETEQ(Queue) - Queue
 4.          FRONTQ(Queue) - item
 5.          ISNEWQ(Queue) - Boolean
 6.          APPENDQ(Queue,Queue) - Queue;
 7.     for all q,r ε Queue, i ε item let
 8.          ISNEWQ(NEWQ) = true
 9.          ISNEWQ(ADDQ(q,i)) = false
10.          DELETEQ(NEWQ) = NEWQ
11.          DELETEQ(ADDQ(q,i)) =
12.               if ISNEWQ(q) then NEWQ
13.                    else ADDQ(DELETEQ(q),i)
14.          FRONTQ(NEWQ) = UNDEFINED
15.          FRONTQ(ADDQ(q,i)) =
16.               if ISNEWQ(q) then i else FRONTQ(q)
17.          APPENDQ(q,NEWQ) = q
18.          APPENDQ(r,ADDQ(q,i)) = ADDQ(APPENDQ(r,q),i)
19.     end
end          Queue
```

Figure 54. Axiomatic Specification of QUEUE

Specifications of this moderate size and for such well known structures as in the examples are quite easy to understand. More complex structures may pose problems in comprehensibility. Besides, there is the formidable task of validating an implementation against the axioms, and validating the consistency and independence of the axiom system itself. Presently, effective techniques have not been developed along this line.

6. REFERENCES

[Alf 76a] Alford, M. W., "R-Nets: A Graph Model for Real-Time Software Requirements," Proc. MRI Symposium on Computer Software Engineering (April 1976).

[Alf 76b] Alford, M. W., "A Requirement Engineering Methodology for Real-Time Processing Requirements," Proc. 2nd International Conference on Software Engineering (October 1976).

[Bae 68] Baer, J. L., "Graph Models of Computations in Computer Systems," Ph.D. Dissertation, Report No. 68-46, University of California, Los Angeles, UCLA-10P14-51 (1968).

[Bal 76] Balkovich, E. and Engleberg, G., "Research Towards a Technology to Support the Specification of Data Processing System Performance Requirements," Proc. 2nd International Conference on Software Engineering (October 1976).

[Bec 73] Becker, J. M., "A Structural Design Process Philosophy and Methodology," Ph.D. Dissertation, University of California, Berkeley (1973).

[Bel 76] Belford, P. C., Bond, A. F., Henderson, D. G., and Sellers, L. S., "Specifications: A Key to Effective Software Development, Proc. 2nd International Conference on Software Engineering (October 1976).

[Bel 76a] Bell, T. E. and Bixler, D. C., "A Flow Oriented Requirement Statement Language," Proc. MRI Symposium on Computer Software Engineering (April 1976).

[Bel 76b] Bell, T. E., Bixler, D. C., and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," Proc. 2nd International Conference on Software Engineering (October 1976).

[Bel 76c] Bell, T. E. and Thayer, T. A., "Software Requirements: Are They Really a Problem?: Proc. 2nd International Conference on Software Engineering (October 1976).

[Boe 73] Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," Datamation, pp. 48-59 (May 1973).

[Boe 74] Boehm, B. W., "Some Steps Toward Formal and Automated Aids to Software Requirements Analysis and Design," IFIPS 74, pp. 192-197 (1974).

[Bri 70] Brinch-Hansen, P., "The Nucleus of a Multiprogramming System," Comm. ACM, Vol. 13, No. 4, pp. 238-241, 250 (April 1970).

[Bri 73] Brinch-Hansen, P., Operating System Principles, Prentice-Hall, Inc., (1973).

[Bri 77] Brinch-Hansen, P., The Architecture of Concurrent Programming, Prentice-Hall, Inc. (1977).

[Bri 74] Bridge, R. F. and Thompson, E. W., "A Module Interface Specification Language," Information Systems Research Laboratory, University of Texas at Austin, Technical Report No. 163 (December 1974).

[Bro 75] Brooks, F. P., Jr., The Mystical Man-Month, Addison-Wesley Publishing Company (1975).

[Bur 74] Burns, F., et al., "Current Software Requirements Engineering Technology," TRW Systems Group, Huntsville, Alabama (August 1974).

[Cam 74] Campbell, R. H. and Habermann, A. N., "The Specification of Process Synchronization by Path Expressions," Lecture Notes on Computer Science, Vol. 16 (1974).

[Cer 71] Cerf, V., Fernandez, E., Gostelow, K., and Volansky, S., "Formal Properties of a Graph Model of Computation," UCLA Technical Report No. UCLA-ENG-7178, University of California, Los Angeles (December 1971).

[Cer 72] Cerf, V. G., "Multiprocessors, Semaphores, and a Graph Model of Computation," Ph.D. Dissertation, UCLA-ENG-7223 (April 1972).

[COD 62] CODASYL Development Committee, "An Information Algebra Phase I Report," Comm. ACM, Vol. 5, No. 4, pp. 190-204 (April 1962).

[Cou 73] Couger, J. D., "Evolution of Business System Analysis Techniques," Computing Surveys, Vol. 5, No. 3, pp. 167-198 (September 1973).

[CSC 73] Computer Sciences Corporation, "A Users Guide to the Threads Management System," (November 1973).

[Dah 72] Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, U.S.A. (1972).

[DaV 76] Davis, C. G. and Vick, C. R., "The Software Development System," Proc. 2nd International Conference on Software Engineering (October 1976).

[Den 70] Dennis, J. B. and Patil, S., "Computation Structures," Course Notes for Course 6.232 (MIT), Department of Electrical Engineering, MIT (1970).

[Dij 68] Dijkstra, E. W., "The Structure of the Multiprogramming System," Comm. ACM, Vol. 11, No. 5, pp. 341-346 (May 1968).

[Dij 68a] Dijkstra, E. W., "Cooperating Sequential Processes," Programming Languages, (ed. Genuys, F.) Academic Press, New York, pp. 43-112 (1968).

[Ear 71] Earley, J., "Toward an Understanding of Data Structures," Comm. ACM, Vol. 14, No. 10, pp. 617-627 (October 1971).

[End 75] Enders, A., "An Analysis of Errors and Their Causes in System Programs," Proc. International Conference on Reliable Software, pp. 327-336 (April 1975).

[Fit 76] Fitzwater, D. R., "The Formal Design and Analysis of Distributed Data Processing Systems," Report CSTR 279, University of Wisconsin Computer Science Department (October 1976).

[Gog 75] Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B., "Abstract Data-Types as Initial Algebras and Correctness of Data Representations," Proc. Conference on Computer Graphics, Pattern Recognition and Data Structure (May 1975).

[Gol 75] Goldberg, P. C., "Structured Programming for Non-Programmer," IBM Report RC-5318 (March 1975).

[Goo 75] Good, D. I., London, R. L., and Bledsoe, W. W., "An Interactive Program Verification System," IEEE Trans. on Software Engineering, Vol. SE-1, No. 1, pp. 59-67 (March 1975).

[Gos 71] Gostelow, K. P., "Flow of Control, Resource Allocation, and the Proper Termination of Programs," Ph.D. Dissertation, UCLA-ENG-7179 (December 1971).

[Gos 72] Gostelow, K. P., Cerf, V. G., Volansky, S., and Estrin, G., "Proper Termination of Flow Control in Programs Involving Concurrent Processes," Proc. ACM National Conference, pp. 742-754 (August 1972).

[Gos 75] Gostelow, K. P., "Computation Modules and Petri Nets," Proc. 3rd Milwaukee Symposium on Automatic Computation and Control, pp. 345-353 (April 1975).

[Gre 75] Greif, I., "Semantics of Communicating Parallel Processes," Ph.D. Dissertation, MAC TR-154, MIT (September 1975).

[Gri 66] Grindley, C. B. B., "Systematic - A Nonprogramming Large for Designing and Specifying Commercial System for Computers," Computer Journal, pp. 124-128 (August 1966).

[Gut 76] Gutlag, J., Horowitz, E., and Musser, D., "The Design of Data Structure Specifications," Proc. 2nd International Conference on Software Engineering (October 1976).

[Hab 75] Habermann, A. N., "Path Expressions," Department of Computer Science, Carnegie-Mellon University (1975).

[Hac 74] Hack, M., "Decision Problems for Petri Nets and Vector Addition Systems," Computation Structures Group Memo 95, Project MAC, MIT (March 1974).

[Ham 76a] Hamilton, M. and Zeldin, S., "Higher Order Software - A Methodology for Defining Software," IEEE Trans. on Software Engineering, Vol. SE-2, No. 1, pp. 9-32 (March 1976).

[Ham 76b] Hamilton, M. and Zeldin, S., "The Foundations for AXES: A Specification Language Based Upon Completeness of Control," The Charles Stark Draper Laboratory, Inc., R-964 (March 1976).

[Ham 76c] Hamilton, M., and Zeldin, S., "Integrated Software Development System/Higher Order Software Conceptual Description," TR-3, Higher Order Software, Inc. (November 1976).

[Har 74] Harrison, M. A., "Some Linguistic Issues in Design," in [Spi 74], pp. 405-416 (1974).

[Hew 75] Hewitt, C. E. and Smith, B., "Towards a Programming Apprentice," IEEE Trans. on Software Engineering, Vol. SE-1, No. 1, pp. 26-45 (March 1975).

[Ho 74] Ho, T. I. M. and Nunamaker, J. F., Jr., "Requirements Statement Language Principles for Automatic Programming," Proc. ACM National Conference, pp. 279-288 (1974).

[Hoa 72] Hoare, C. A. R., "Proof of Correctness of Data Representation," Acta Informatica, Vol. 1, pp. 271-281 (1972).

[Hoa 74] Hoare, C. A. R., "Monitors: An Operating System Structuring Concept," Comm. ACM, Vol. 17, pp. 549-557 (October 1974).

[Hol 70] Holt, A. W. and Commer, F., "Events and Conditions: An Approach to the Description and Analysis of Dynamic Systems," Third Semi-Annual Technical Report Part II for the Project Research in Machine-Independent Software Programming Applied Data Research Inc. (April 1970).

[HUG 75] Hughes Aircraft Company, "1975 IR&D Structured Design Methodology, Vol. II: Structured Design," FR 76-17-289 (1975).

[IBM 71] IBM, "The Time Automated Grid System (TAG): Sales and Systems Guide," Publication No. GY20-0358-1, pp. 1-12 (May 1971).

[IBM 73] IBM, "HIPO: Design Aid and Documentation Tool," IBM, SR20-9413-0 (1973).

[Jon 63] Jones, J. C., (ed.), Conference on Design Methods, The MacMillan Company, New York (1963).

[Kar 69] Karp, R. M. and Miller, R. E., "Parallel Program Schemata," Journal of Computer and System Science, Vol. 3, pp. 147-195 (1969).

[Kon 76] Konsynski, B. R., "A Model of Computer Aided Definition and Analysis of Information System Requirements," Ph.D. Dissertation, Purdue University (December 1976).

[Lan 63] Langefors, B., "Some Approaches to the Theory of Information Systems," BIT, Vol. 3, pp. 229-254 (1963).

[Lau 75] Lauer, P. E. and Campbell, R. H., "Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes," Acta Informatica, Vol. 5, pp. 297-332 (1975).

[Lei 67] Leibowitz, H. "The Technical Specification - Key to Management Control of Computer Programming," AFIPS Conference Proceeding, Vol. 30, pp. 51-59 (1967).

[Lis 75] Liskov, B. H. and Zilles, S. N., "Specification Techniques for Data Abstractions," IEEE Trans. on Software Engineering, Vol. SE-1, No. 1, pp. 7-19 (March 1975).

[Lis 76] Liskov, B. H. and Berzins, V. "An Appraisal of Program Specifications, Computations Structures Group Memo 141, Laboratory for Computer Science, MIT (July 1976).

[Lon 72] London, K. R., Decision Tables, Auerback Publishers, Inc. (1972).

[Luc 68] Luconi, F. L., "Asynchronous Computational Structures," Ph.D. Dissertation, MIT-TR-49, MIT (February 1968).

[Lyn 69] Lynch, H. J., "ADS: A Technique in Systems Documentation," Database, Vol. 1, No. 1, pp. 6-18 (Spring 1969).

[McG 75] McGowan, C. L. and Kelly, J. R., Top-down Structured Programming Techniques, Petrocelli/Charter, New York (1975).

[Mee 73] Meekers, R. E., Jr. and Ramamoorthy, C. V., "A Study in Software Reliability and Evaluation," TML No. 39, University of Texas, Austin (February 1973).

[Mes 72] Mesarovic, M. D., "A Mathematical Theory of General Systems," in Trends in General Systems Theory (ed. Klir, G. J.) Wiley-Interscience (1972).

[Neu 76] Neumann, P. G., Feiertag, R. J., Levit, K. N., and Robinson, L., "Software Development and Proofs of Multi-level Security," Proc. Second International Conference on Software Engineering, pp. 421-428 (October 1976).

[Nun 71] Nunamaker, J. F., Jr., "A Methodology for the Design and Optimization of I formation Processing Systems," AFIPS Conference Proceedings, Vol. 38, pp. 283-293 (May 1971).

[Owi 75] Owicki, S. S., "Axiomatic Proof Techniques for Parallel Programs," Ph.D. Dissertation, TR75-251 Cornell University (July 1975).

[Par 72a] Parnas, D. L., "A Technique for Software Module Specification With Examples," Comm. ACM, Vol. 15, No. 5, pp. 330-336 (May 1972).

[Par 72b] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," Comm. ACM, Vol. 15, No. 12, pp. 1053-1058 (December 1972).

[Pat 70] Patil, S., "Coordination of Asynchronous Events," Ph.D. Dissertation, MAC-TR-72, MIT (1970).

[Pet 62] Petri, C. A., "Kommunikation mit automaten," Schriften des Reinsch-West Falischen Inst. Instrumentelle Math und der Universitat Bonn Nr 2 Bonn (1962).

[Pet 76] Peters, L. J. and Tripp, L. L., "Design Representative Schemes," Proc. MRI Symposium on Computer Software Engineering (April 1976).

[Pet 76a] Peters, L. and Tripp, L., "Is Software Design Wicked?", Datamation, pp. 127-129 (May 1976).

[Pos 74] Postel, J. B., "A Graph Model Analysis of Computer Communications Protocols," Ph.D. Dissertation, UCLA-ENG-7410 (January 1974).

[Pri 77] Principato, R., "A Survey of Specification Techniques for Parallel Processes," The Charles Stark Draper Laboratory (January 1977).

[RCA 72] RCA, "Functional Flow Diagrams and Description for AEGIS--A Systems Engineering Management Tool," Corporate Engineering Services, Reprint RE-19-1-6 (1972).

[Ree 76] Reed, D. P., "Processor Multiplexing in a Layered Operating System," S. M. Thesis, MAC TR-164, MIT, (July 1976).

[Rit 72] Rittel, H. W. J, "On the Planning Crisis: System Analysis of the First and Second Generations," Bedriftsokonomen, No. 8, pp. 390-396.

[Rob 75] Robinson, L. and Holt, R. C., "Formal Specifications for Solutions to Synchronization Problems," Computer Science Group, Stanford Research Institute (1975).

[Rod 67] Rodriguez, J., "A Graph Model for Parallel Computation," Sc. D. Dissertation, Department of Electrical Engineering, MIT (1967).

[Ros 72] Rose, C. W., "LOGOS and the Software Engineer," AFIPS Conference Proceedings, Vol. 41, Part I, pp. 311-323 (1972).

[Ros 77] Ross, D., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1 (January 1977).

[SAF 75] "SAFEGAURD Data Processing System," The Bell System Technical Journal, Special Supplement (1975).

[Sal 76] Salter, K., "A Methodology for Decomposing System Requirements Into Data Processing Requirements," Proc. 2nd International Conference on Software Engineering (October 1976).

[Sha 76] Shaw, A., "A Path Notation for Describing Software," University of Washington (1976).

[Sho 75] Sholl, H. A. and Booth, T. L., "Software Performance Modeling Using Computation Structures," IEEE Trans. on Software Engineering, Vol. SE-1, No. 4, pp. 414-420 (December 1975).

[Sim 73] Simmons, R. F., "Semantic Networks: Their Computation and Use for Understanding English Sentences," in Computer Models of Thought and Language (ed. Schank, R. C.), W. H. Freeman and Company (1973).

[Slu 68] Slutz, D. R., "The Flow-Graph Schemata Model of Parallel Computation," Ph.D. Dissertation, MAC-TR-53, MIT (September 1968).

[Spi 74] Spillers, W. R., (ed.) Basic Questions of Design Theory, North-Holland Publishing Company (1974).

[Sta 76] Stay, J. F., "HIPO and Integrated Program Design," IBM System Journal, Vol. 15, No. 2, pp. 143-154 (1976).

[Ste 74] Stevens, W. P., et al., "Structured Design," IBM Systems Journal, Vol. 13, No. 2, pp. 115-139 (1974).

[Ste 76] Stephenson, W. E., "An Analysis of the Resources Used in the SAFEGUARD System Software Development," Proc. Second International Conference on Software Engineering, pp. 312-321 (October 1976).

[Tei 71] Teichroew, D. and Sayani, H., "Automation of System Building," Datamation, pp. 25-30 (August 15, 1971).

[Tei 72] Teichroew, D., "A Survey of Languages for Stating Requirements for Computer-Based Information Systems," AFIPS Conference Proceedings, Vol. 41, Part II, pp. 1203-1224 (Fall 1972).

[Tei 74a] Teichroew, D., "Problem Statement Analysis: Requirements for the Problem Statement Analyzer (PSA)," in Systems Analysis Techniques, (eds., Couger, J. D. and Knapp, R. W.,) John Wiley and Sons (1974).

[Tei 74b] Teichroew, D., Hershey, E. and Bastarache, M., "An Introduction to PSL/PSA," ISDOS Working Paper No. 86, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor (March 1974).

[Tei 77] Teichroew, D. and Hershey, E. A. III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, pp. 16-33 (January 1977).

[Tha 76] Thayer, T. A., et al., "Software Reliability Study: Final Technical Report," TRW Report 75-2266-1.9-5 (March 1976).

[Vic 74] Vick, C. R., "Specification for Reliable Software," Proc. EASCON '74 (October 1974).

[Wil 75] Wilson, M. L., "The Information Automat Approach to Design and Implementation of Computer-Based Systems," IBM, Report IBM-FSD, (June 1975).

[Wym 76] Wymore, A. W., "System Engineering Methodology for Inter-Disciplinary Teams," Wiley-Interscience (1976).

[You 58] Young, J. W. and Kent, H. K., "Abstract Formulation of Data Processing Problems," Journal of Industrial Engineering, pp. 471-479 (November-December 1958).

[Zad 71] Zadeh, L. A., "Toward a Theory of Fuzzy System," in Aspects of Network and Systems Theory (eds. Kalman, R. E.. and DeClaris, N.), Holt, Rinehart and Winston, Inc. (1971).

[Zil 75] Zilles, S. N., "Data Algebra: A Specification Technique for Data Structures," Ph.D. Dissertation, Project MAC, MIT (1975).