MODULE 2

INTRODUCTION TO SOFTWARE COST ESTIMATION

Estimating the cost of software product is one of the most difficult and error-prone tasks in software engineering. It is difficult to make an accurate cost estimate during the planning phase of software development.

A preliminary estimate is prepared during the planning phase and presented at the project feasibility review. An improved estimate is presented at the software requirements review, and the final estimate is presented at the preliminary design review. Each estimate is a refinement of the previous one, and is based on the additional information gained as a result of additional work activities.

SOFTWARE COST FACTORS

The factors that influence the cost of a software product are Programmer Ability, Product Complexity, Product Size, Available Time, Required Reliability, Level of Technology. Primary among the cost factors are the individual abilities of project personnel and their familiarity with the application area; the complexity of the product; the size of the product, the available time, the required level of reliability; the level of technology utilized, and the availability, familiarity, and stability of the system used to develop the product.

Programmer Ability

A well-known experiment conducted in 1968 by Harold Sackman and colleagues. It determines the relative influence of batch and time-shared access on programmer productivity. Twelve experienced programmers were each given two programming problems to solve, some using batch facilities and some using time-sharing.

The differences between best and worst performance were facto9rs of 6 to 1 in program size, 8 to 1 in execution time, 9 to 1 in development time, 18 to 1 in coding time, and 28 to 1 in debugging time. On very large projects, the differences in individual programmer ability will tend to average out, but on projects utilizing five or fewer programmers, individual differences in ability can be significant.

Product Complexity

There are three categories of software product: **Application Programs**, which include data processing and scientific programs; **Utility Programs**, such as compilers, linkage editors, and inventory systems; and **System Programs**, such as database management systems, operating systems, and real-time systems.

Brooks states that utility programs are three times as difficult to write as application programs, and that system programs are three times as difficult to write as utility programs. His levels of product complexity are thus 1-3-9 for applications-utility-systems programs.

Boehm uses three levels of product complexity and provides equations to predict total programmer-months of effort, PM, in terms of the number of thousands of delivered source instruction, KDSI, in the product. Programmer cost for a software project can be obtained by multiplying the effort in programmer-months by the cost per programmer-month. The equations were derived by examining historical data from a large number of actual projects. In Boehm's terminology, the three levels of product complexity are organic, semidetached, and embedded programs.

Application programs: PM = 2.4*(KDSI)**1.05

Utility programs: PM = 3.0*(KDSI)**1.12

Systems programs: PM = 3.6*(KDSI)**1.20

development time for a program is

Application programs: TDEV = 2.5*(PM)**0.38
Utility programs: TDEV = 2.5*(PM)**0.35
Systems programs: TDEV = 2.5*(PM)**0.32

Given the total programmer-months for a project and the nominal development time required, the average staffing level can be obtained by simple divisions.

For our 60 KDSI program, we obtain the following results:

Application programs: 176.6 PM/17.85 MO = 9.9 programmers
Utility programs: 294 PM/18.3 MO = 16 programmers
Systems programs: 489.6 PM/18.1 MO = 27 programmers

One of the common failures in estimating the number of source instructions in a software product is to underestimate the amount of housekeeping code required. **Housekeeping code** is that portion of the source code that handles input/output interactive user communication, human interface engineering, and error checking and error handling.

Product Size

A large software product is more expensive to develop than a small one. Boehm's equations indicate that the rate of increase in required effort grows with the number of source instructions at an exponential rate slightly greater than 1. Some investigators believe that the rate of increase in effort grows at an exponential rate slightly less than 1, but most use an exponent in the range of 1.05 to 1.83.

Available Time

Total project effort is sensitive to the calendar time available for project completion. Several investigators agree that software projects require more total effort if development time is compressed or expanded from the optimal time. The most striking feature is the Putnam curve. According to Putnam, project effort is inversely proportional to the fourth power of development time, E = k/(Td**4). This curve indicates an extreme penalty for schedule compression and an extreme reward for expanding the project schedule.

Putnam also states that the development schedule cannot be compressed below about 86% of the nominal schedule, regardless of the people or resources utilized.

In a study of 63 software projects, Boehm found that only four had compression factors less than 75% of the development time predicted by his cost estimation model. Boehm states: "There is a limit beyond which a software project cannot reduce its schedule by buying more personnel and equipment. This limit occurs roughly at 75% of the nominal schedule".

Required Level of Reliability

Software reliability can be defined as the probability that a program will perform a required function under stated conditions for a stated period of time. Reliability can be expressed in terms of accuracy, robustness, completeness, and consistency of the source code. Reliability characteristics can be built into a software product, but there is a cost associated with the increased level of analysis, design, implementation, and verification and validation effort that must be exerted to ensure high reliability. The multipliers range from 0.75 for very low reliability to 1.4 for very high reliability. The effort ratio is thus 1.87 (1.4/0.75).

Level of Technology

The level of technology in a software development project is reflected by the programming language, the abstract machine (hardware plus software), the programming practices, and the software tools used. It is well known that the number of source instructions written per day is largely independent of the language used, and that program statements written in high-level languages such as FORTRAN and Pascal expand into several machine-level statements. Use of high-level language instead of assemble language thus increases programmer productivity by a factor of 5 to 10.

The type-checking rules and self-documenting aspects of high-level languages improve the reliability and modifiability. Ada provide additional features to improve programmer productivity and software reliability. These features include strong type-checking, data abstraction, separate compilation, exception handling, interrupt handling, and concurrency mechanisms.

Modern programming practices include use of systematic analysis and design techniques, structured design notations, walkthroughs and inspections, structured coding, systematic testing, and a program development library.

Software tools range from elementary tools, such as assemblers and basic debugging aids, to compilers and linkage editors, to interactive text editors and database management system,s to program design language processors and requirements specification analyzers, to fully integrated development environments that include configuration management and automated verification tools.

The use of modern practices and the use of modern development tools can reduce programming effort to 0.67 (0.82/1.24).

SOFTWARE COST ESTIMATION TECHNIQUES

Within most organizations, software cost estimates are based on past performance. Historical data are used to identify cost factors. Cost and productivity data must be collected on current projects in order to estimate future ones. It can be done either top-down or bottom-up.

Top down estimation first focuses on system-level costs, such as computing resources and personnel required to develop the system, the costs of configuration management, quality assurance, system integration, training, and publications. Personnel costs are estimated by examining the cost of similar past projects.

Bottom up estimation first estimates the cost to develop each module or subsystem. Those costs are combined to arrive at an overall estimate.

Expert Judgement

The most widely used cost estimation technique is expert judgement, which is an top-down estimation technique. Expert judgement relies on the experience, background, and business sense of one or more key people in the organization.

An expert might arrive at a cost estimate in the following manner: The system to be developed is a process control system similar to one that was developed last year in 10 months at a cost of $1 million. The new system has similar control functions, but has 25 percent more activities to control; thus, we will increase our time and cost estimates by 25 percent. We will use the same computer and external sensing/controlling devices; and many of the same people are available to develop the new system, so we can reduce our estimate by 20 percent.

We can reuse much of low-level code from the previous product, which reduces the time and cost estimates by 25 percent. The net effect of these considerations is a time and cost estimates by 20 percent, which results in an estimate of $800,000 and 8 months development time. The customer has

budgeted $1 million and 1 year delivery time for the system. Therefore, we add a small margin of safety and bid the system at $850,000 and 9 months development time.

The biggest advantage of expert judgement, namely, **experience**, can also be a liability. The expert may be confident that the project is similar to a previous one. Groups of experts sometimes prepare a consensus estimate to minimize individual oversights and lack of familiarity.

The major disadvantage of group estimation is the effect that interpersonal group dynamics may have on individuals in the group.

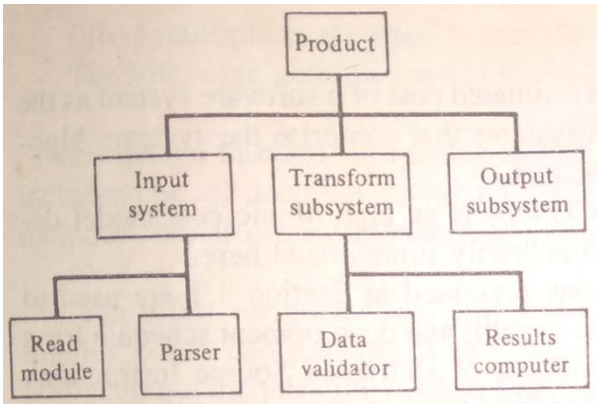
Delphi Cost Estimation

The Delphi technique was developed by Rand Corporation in 1948 to gain expert consensus without introducing the adverse side effects of group meetings. The Delphi technique can be adapted to software cost estimation in the following manner:

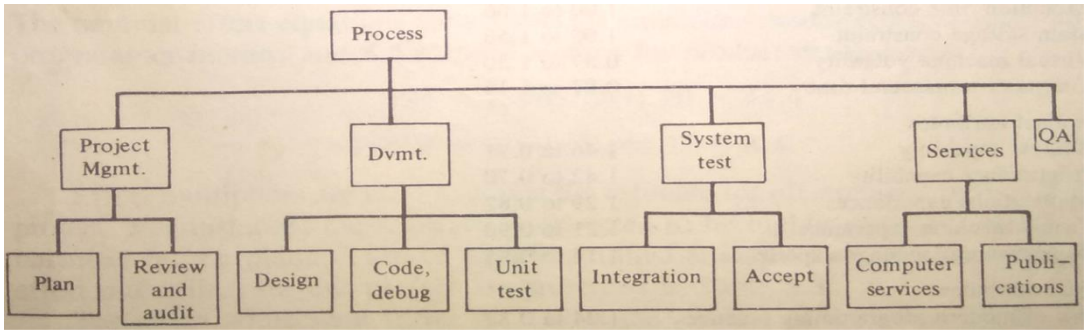
1. A coordinator provides each estimator with the *System Definition* document and a form for recording a cost estimate.
2. Estimators study the definition and complete their estimates anonymously. They may ask questions of the coordinator, but they do not discuss their estimates with one another.
3. The coordinator prepares and distributes a summary of the estimators' responses, and includes any ususual rationales noted by the estimators.
4. Estimators complete another estimate, again anonymously, using the results from the previous estimate. Estimators whose estimates differ sharply from the group may be asked, anonymously, to provide justification for their estimates.
5. The process is iterated for as many rounds as required. No group discussion is allowed during the entire process.

Work Breakdown Structures

Expert judgement and group consensus are top-down estimation techniques. The work breakdown structure method is a bottom-up estimation tool. A work breakdown structure is a hierarchical chart that accounts for the individual parts of a system. A WBS chart can indicate either product hierarchy or process hierarchy.



A product work breakdown structure



A process work breakdown structure

Product hierarchy identifies the product components and indicates the manner in which the components are interconnected. A WBS chart of process hierarchy identifies the work activities and the relationships among those activities. Using WBS technique, costs are estimated by assigning costs to each individual component in the chart and summing the costs.

Some planners use both product and process WBS chart for cost estimation. The primary advantages of the WBS technique are in identifying and accounting for various process and product factors, and in making explicit exactly which costs are included in the estimate.

Algorithmic Cost Models

Algorithmic cost estimators compute the estimated cost of a software system as the sum of the costs of the modules and subsystems that comprise the system. Algorithmic models are thus bottom-up estimators.

The Constructive Cost Model (COCOMO) is an algorithmic cost model described by Boehm in 1970 based on study of 63 projects. COCOMO is a regression model based on number of Lines of Code (LOC). COCOMO is based on procedural cost estimate model. COCOMO is used to reliably predict various parameters associated with making a project such as size, effort, cost, time and quality.
Boehm uses three levels of product complexity and provides equations to predict total programmer-months of effort, PM, in terms of the number of thousands of delivered source instruction, KDSI, in the product. Programmer cost for a software project can be obtained by multiplying the effort in programmer-months by the cost per programmer-month. The equations were derived by examining historical data from a large number of actual projects.

In Boehm's terminology, the three levels of product complexity are organic, semidetached, and embedded programs.

Application programs: PM = 2.4*(KDSI)**1.05
Utility programs: PM = 3.0*(KDSI)**1.12
Systems programs: PM = 3.6*(KDSI)**1.20

The development time for a program is

Application programs: TDEV = 2.5*(PM)**0.38
Utility programs: TDEV = 2.5*(PM)**0.35
Systems programs: TDEV = 2.5*(PM)**0.32

Given the total programmer-months for a project and the nominal development time required, the average staffing level can be obtained by simple divisions. For our 60 KDSI program, we obtain the following results:

Application programs: 176.6 PM/17.85 MO = 9.9 programmers
Utility programs: 294 PM/18.3 MO = 16 programmers
Systems programs: 489.6 PM/18.1 MO = 27 programmers

COCOMO Effort Multipliers:

Multiplier	Range of values
Product attributes	
Required reliability	0.75 to 1.40
Database size	0.94 to 1.16
Product complexity	0.70 to 1.65
Computer attributes	
Execution time constraint	1.00 to 1.66
Main storage constraint	1.00 to 1.56
Virtual machine volatility	0.87 to 1.30
Computer turnaround time	0.87 to 1.15
Personnel attributes	
Analyst capability	1.46 to 0.71
Programmer capability	1.42 to 0.70
Applications experience	1.29 to 0.82
Virtual machine experience	1.21 to 0.90
Programming language experience	1.14 to 0.95
Project attributes	
Use of modern programming practices	1.24 to 0.82
Use of software tools	1.24 to 0.83
Required development schedule	1.23 to 1.0

Effort multipliers are used to adjust the estimate for product attributes, computer attributes, personnel attributes, and project attributes.

The COCOMO equations incorporate a number of assumptions. For example, the nominal organic mode (application programs) equations apply in the following situations:

- Small to medium-size projects (2K to 32K DSI)
- Familiar applications area
- Stable, well-understood virtual machine
- In-house development effort

Effort multipliers are used to modify these assumptions. The following activities are covered by the estimates:

- Covers design through acceptance testing
- Includes cost of documentation and reviews
- Includes cost of project manager and program librarian

The effort estimators exclude planning and analysis costs, installation and training costs, and the cost of secretaries, janitors, and computer operators. The DSI estimate includes job control statements and source statements, but excludes comments and unmodified utility routines.

Other assumptions of software projects estimated by COCOMO:

- Careful definition and validation of requirements is performed by a small number of capable people.
- The requirements remain stable throughout the project.
- Careful definition and validation of the architectural design is performed by a small number of capable people.
- Detailed design, coding, and unit testing are performed in parallel by groups of programmers working in teams.
- Integration testing is based on early test planning.

- Interface errors are mostly found by unit testing and by inspections and walkthroughs before integration testing.
- Documentation is performed incrementally as part of the development process.

In other words, systematic techniques of software engineering are used throughout the development process.
COCOMO can also be used to investigate trade-offs in the development process by performing sensitivity analysis on the cost estimate.

Cost estimation procedure using COCOMO:

1. Identify all subsystems and modules in the project.
2. Estimate the size of each module and calculate the size of each subsystem and the total system.
3. Specify module-level effort multipliers for each module. The module-level multipliers are: product complexity, programmer capability, virtual machine experience, and programming language experience.
4. Compute the module effort and development time estimates for each module, using the nominal estimator equations and the module-level effort multipliers.
5. Specify the remaining 11 effort multipliers for each subsystem.
6. From steps 4 and 5, compute the estimated effort and development time for each subsystem.
7. From step 6, compute the total system effort and development time.
8. Perform a sensitivity analysis on the estimate to establish trade-off benefits.
9. Add other development costs, such as planning and analysis, that are not included in the estimate.
10. Compare the estimate with one developed by top-down Delphi estimation. Identify and rectify the differences in the estimates.

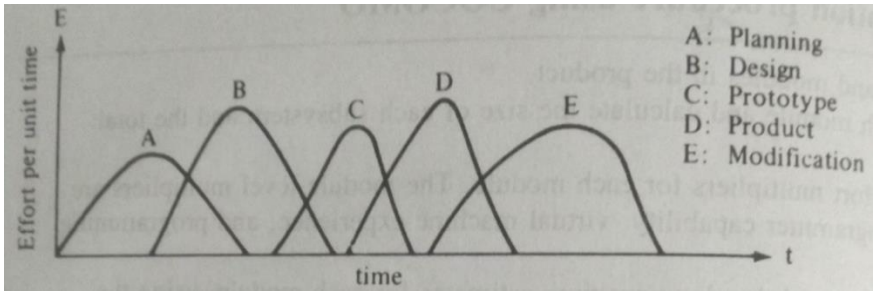
Outcomes of COCOMO:

1. **Effort**: Amount of labour that will be required to complete a task. It is measured in person-months unit.
2. **Schedule**: Amount of time required for completion of job, proportional to the effort put. It is measured in units of time such as weeks, months.

The greatest **advantage** of COCOMO is that the model can be used to gain insight into the cost factors within an organization. Data can be collected and analyzed, new factors can be identified, and the effort multipliers can be adjusted.

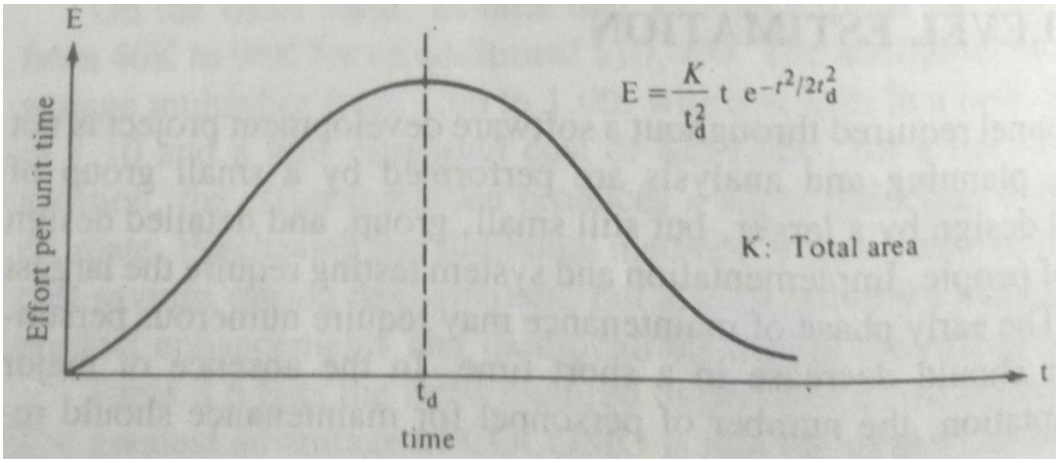
STAFFING LEVEL ESTIMATION

The number of personnel required throughout a software development project is not constant. Typically, planning and analysis are performed by a small group of people, architectural design by a larger, but still small, group, and detailed design by a larger number of people. Implementation and system testing require the largest numbers of people. The early phase of maintenance may require numerous personnel, but the number should decrease in a short time. In the absence of major enhancement or adaptation, the number of personnel for maintenance should remain small.

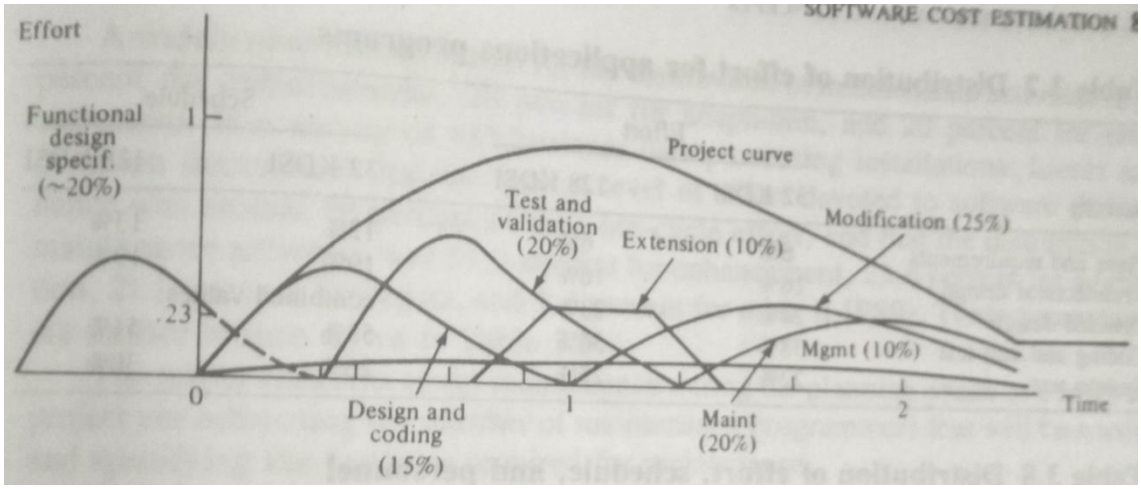


Cycles in a research and development project

In 1958, Norden observed that research and development projects follow a cycle of planning, design, prototype, development, and use, with the corresponding personnel utilization. The sum of the areas under the curves can be approximated by the Rayleigh equation. Any particular point in Rayleigh curve represents the number of full-time equivalent personnel required at that instant in time.



The Rayleigh curve of effort vs. time



Putnam's interpretation of the Rayleigh Curve

In 1976, Putnam reported that the personnel level of effort required throughout the life cycle of a software product has a similar envelope. Putnam studied 50 Army software projects and 150 other projects to determine how the Rayleigh curve can be used to describe the software life cycle.

Boehm also presents the distribution of effort and schedule in a software development project.

Activity	Effort		Schedule	
	32 KDSI	128 KDSI	32 KDSI	128 KDSI
Plans and requirements	6%	6%	12%	13%
Architectural design	16%	16%	19%	19%
Detailed design	24%	23%	combined values:	
Coding and unit test	38%	36%	55%	51%
System test	22%	25%	26%	30%

Distribution of effort for application programs

Activity	Effort		Schedule		Personnel	
	32 KDSI	128 KDSI	32 KDSI	128 KDSI	32 KDSI	128 KDSI
Plans and requirements	5 PM	24 MM	1.2 MO	3.1 MO	2.9 FSP	8 FSP
Architectural design	15 PM	63 MM	2.2 MO	4.6 MO	5.6 FSP	14 FSP
Detailed design	22 PM	90 MM	combined values:			
Implementation	34 PM	141 MM	7.7 MO	12.2 MO	7.3 FSP	19 FSP
System test	20 PM	90 MM	3.6 MO	7.2 MO	5.6 FSP	14 FSP

Distribution of effort, schedule, and personnel

ESTIMATING SOFTWARE MAINTENANCE COSTS

Software maintenance typically requires 40-60%, and in some cases 90%, of the total life-cycle effort devoted to a software product. Maintenance activities include adding enhancements to the product, adapting the product to new processing environments, and correcting problems.

A widely used distribution of maintenance activities is 60% for enhancements, 20% for adaptation, and 20% for error correction. In a survey of 487 business data processing installations, Lientz and Swanson determined that the typical level of effort devoted to software maintenance was around 50% of total life-cycle effort, and that the distribution of maintenance activities was 51.3% for enhancement, 23.6% for adaptation, 21.7% for repair, and 3.4% for other.

Activity	% Effort
Enhancement	**51.3**
Improved Efficiency	**4.0**
Improved Documentation	**5.5**
User Enhancements	**41.8**
Adaptation	**23.6**
Input data, files	**17.4**
Hardware, Operating System	**6.2**
Corrections	**21.7**
Emergency Fixes	**12.4**
Scheduled Fixes	**9.3**
Others	**3.4**

Maintenance effort distribution

Lientz and Swanson determined that the maintenance programmer in a business data processing installation maintains 32K source instructions. For real-time and aerospace software, numbers in the range of 8K to 10K are more typical.

An estimate of the number of full-time software personnel needed for software maintenance can be determined by dividing the estimated number of source instructions to be maintained

by a maintenance programmer. For example, if a maintenance programmer can maintain 32 KDSI, then two maintenance programmers are required to maintain 64 KDSI:

$$FSP = (64\ KDSI)/(32\ KDSI/FSP) = 2\ FSPm$$

Boehm suggests that maintenance effort can be estimated by use of an activity ratio, which is the number of source instructions to be added or modified in any given time period divided by the total number of instructions:

$$ACT = (DSI_{added} + DSI_{modified})/DSI_{total}$$

The activity ratio is then multiplied by the number of programmer-months required for development in a given time period to determine the number of programmer-months required for maintenance in the corresponding time period:

$$PMm = ACT * MM_{dev}$$

A further enhancement is provided by an effort adjustment factor EAF, which recognizes that the effort multipliers for maintenance may be different from the effort multipliers used for development:

$$PMm = ACT * EAF * MM_{dev}$$

Heavy emphasis on reliability and the use of modern programming practices during development may reduce the amount of effort required for maintenance, while low emphasis on reliability and modern practices during development may increase the difficulty of maintenance.

THE SOFTWARE REQUIREMENTS SPECIFICATION

The analysis phase of software development involves project planning and software requirements definition. The outcome of planning is recorded in the *System Definition*, the *Project Plan*, and the preliminary *User's Manual*. The *Software Requirements Specification* records the outcome of the software requirements definition activity.

The *System Definition, Project Plan,* and preliminary *User's Manual* are concerned with the user and external view of the software product. The *Software Requirements Specification* is a technical specification of requirements for the software product. The goal of software requirements definition is to completely and consistently specify the technical requirements for the software product in a concise and unambiguous manner using formal notations.

The *Software Requirements Specification* is based on the *System Definition*. High-level requirements specified during initial planning are elaborated. The requirements specification will state the "what" of the software product without implying "how". Software design is concerned with specifying how the product will provide the required features.

The format of a requirements specification document is presented as follows:

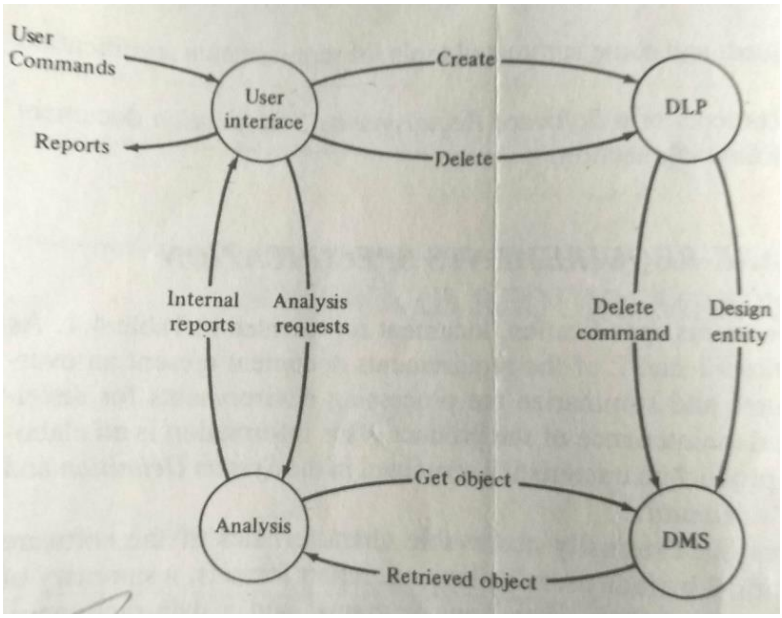
Section 1:	**Product Overview and Summary**
Section 2:	**Development, Operating and Maintenance Environments**
Section 3:	**External Interfaces and Data Flow**
Section 4:	**Functional Requirements**
Section 5:	**Performance Requirements**
Section 6:	**Exception Handling**
Section 7:	**Early Subsets and Implementation Priorities**
Section 8:	**Foreseeable Modifications and Enhancements**
Section 9:	**Acceptance Criteria**
Section 10:	**Design Hints and Guidelines**

Section 11:	**Cross-Reference Index**
Section 12:	**Glossary of Terms**

Format of a software requirements specification

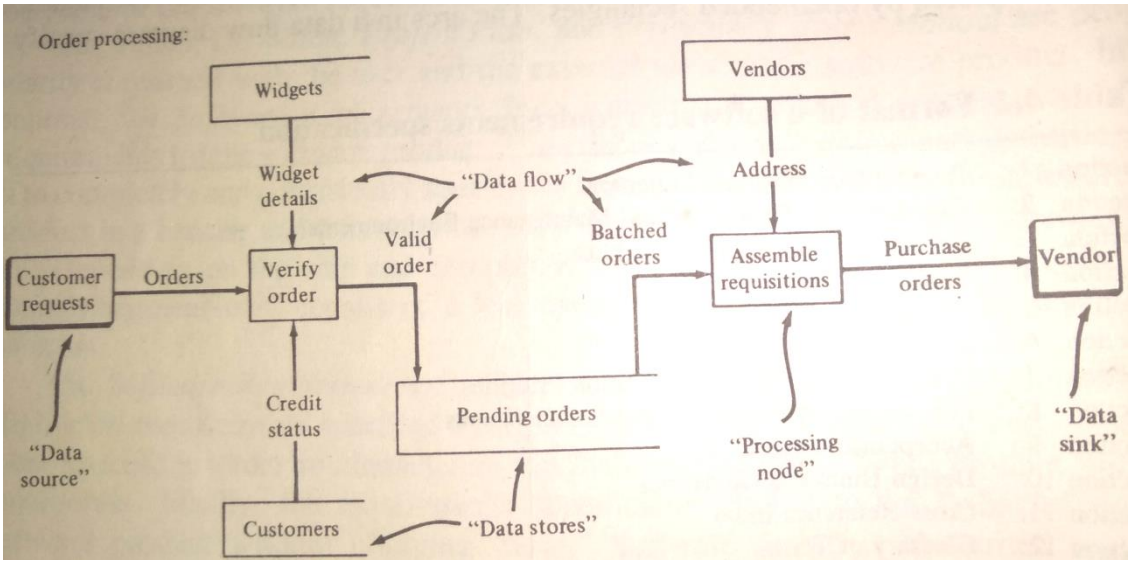
- **Sections 1 and 2 present an overview of product features and summarizes the processing environments for development, operation and maintenance of the product.**
- **Section 3 specifies the externally observable characteristics of the software product. It includes user displays and report formats, a summary of user commands and report options, data flow diagrams, and a data dictionary.**

- **Data flow diagrams specify data sources and data sinks, data stores, transformations to be performed on the data, and the flow of data between sources, sinks, transformations, and stores.**
- **A data store is a conceptual data structure, in the sense that physical implementation details are suppressed; only the logical characteristics of data are emphasized on a data flow diagram.**
- **Data flow diagram can be depicted informally or by using special notation, where data sources and data sinks are depicted by shaded rectangles, transformations by ordinary rectangles and data stores by open ended rectangles. The arcs specify data flow; they are labeled with the names of data items whose characteristics are specified in the data dictionary.**



An informal data flow diagram

- **Like flowcharts, data flow diagram can be used at any level of detail. They can be hierarchically decomposed by specifying the inner workings of the functional nodes using additional data flow diagrams. Unlike flowcharts, data flow diagrams are not concerned with decision structure or algorithmic details.**



A formal data flow diagram

NAME:	**Create**
WHERE USED:	**SDLP**
PURPOSE:	**Create passes a user-created design entity to the SLP processor for verification of syntax**
PURPOSE:	**Create passes a user-created design entity to the SLP processor for verification of syntax**
DERIVED FROM:	**User Interface Processor**
SUBITEMS:	**Name Uses Procedures References**
NOTES:	**Create contains one complete user-created design entity**

A Data dictionary entry

- **Entries in a data dictionary include the name of the data item, and attributes such as the data flow diagrams where it is used, its purpose, where it is derived from, its subitems, and any notes that may be appropriate.**
- **Section 4 specifies the functional requirements. It is expressed in relational and state-oriented notations specifying relationships among inputs, actions, and outputs.**
- **Performance characteristics such as response time for various activities, processing time for various processes, throughput, primary and secondary memory constraints, required telecommunication bandwidth, and special items such as extraordinary security constraints or unusual reliability requirements are specified in Section 5.**
- **Exception handling, including the actions to be taken and the messages to be displayed in response to undesired situations or events, is described in Section 6.**
- **Section 7 specifies early subsets and implementation priorities for the system under development.**
- **Foreseeable modifications and enhancements that may be incorporated into the product following initial product release are specified in Section 8.**
- **The software product acceptance criteria are specified in Section 9. Acceptance criteria specify functional and performance tests that must be performed, and the standards to be applied to source code internal documentation, and external documents such as the**

design specifications, the test plan, the user's manual, the principles of operation, and the installation and maintenance procedures.

- **Section 10 contains design hints and guidelines.**
- **Section 11 relates product requirements to the sources of information used in deriving the requirements. A cross-reference directory should be provided to index specific paragraph numbers in the *Software Requirements Specification* to specific paragraphs in the *System Definition* and the preliminary *User's Manual*, and to other sources of information such as people or documents.**
- **Section 12 provides definitions of terms that may be unfamiliar to the customer and the product developers.**

Desirable Properties:

There are a number of desirable properties that a Software Requirements Specification should possess:

1. **Correct**
2. **Complete**
3. **Consistent**
4. **Unambiguous**
5. **Functional**
6. **Verifiable**
7. **Traceable**
8. **Easily changed**

- **An incorrect or incomplete set of requirements result in a software product that satisfies its requirements but does not satisfy customer needs.**
- **An inconsistent specification states contradictory requirements in different parts of the document, while an ambiguous requirement is subject to different interpretations by different people.**
- **Software requirements should be functional in nature; i.e., they should describe what is required without implying how the system will meet its requirements. This provides maximum flexibility for the product designers.**
- **Requirements must be verifiable from two points of view; 1) Verify the requirements satisfy the customer's needs, 2) Verify the subsequent work products satisfy the requirements.**
- **Finally, the requirements should be indexed, segmented, and cross-referenced to permit easy use and easy modification.**
- **Every software requirement should be traceable to specific customer statements and to specific statements in the System Definition.**
- **Changes will occur, and project success depends on the ability to incorporate change without starting over. Cross-referencing can be accomplished by referring to the appropriate paragraph numbers in the appropriate documents.**

FORMAL SPECIFICATION TECHNIQUES

Specifying the functional characteristics of software product is one of the most important activities to be accomplished during requirements analysis. Formal notations have the advantage of being concise and unambiguous.

Both relational and state-oriented notations are used to specify the functional characteristics of software. Relational notations are based on the concepts of entities and attributes. Entities are named elements in a system. Attributes are specified by applying functions and relations to the named entities. Attributes specify permitted operations on entities, relationships among entities, and data flow between entities.

The state of a system is the information required to summarize the status of system entities at any particular point in time; given the current state and the current stimuli, the next state can be determined.

Relational notations include implicit equations, recurrence relations, algebraic axioms, and regular expressions. State-oriented notations include decision tables, event tables, transition tables, finite-state mechanisms, and Petri nets.

Relational Notations

Implicit Equations

Implicit equations state the properties of a solution without stating a solution method. For example, matrix inversion as

$$M \times M' = I \pm E$$

Matrix inversion is specified as the matrix product of the original matrix M and the inverse of M, M', yields the identity matrix I plus or minus the error matrix E, where E specifies allowable computational errors.

Implicit specification of a square root function, SQRT, can be stated as

$$(0 <= X <= Y)\ [ABS(SQRT(X)**2 - X) < E]$$

States that for all (real?) values of X in the closed range 0 to Y, computing the square root of X, squaring it, and subtracting X results in an error value in some range.

Recurrence Relations

A recurrence relation consists of an initial part called the basis and one or more recursive parts. For example, successive Fibonacci numbers are formed as the sum of the previous two Fibonacci numbers, where the first one is defined as 0, and the second as 1. This can be defined by the recurrence

$$FI(0) = 0$$
$$FI(1) = 1$$
$$FI(N) = FI(N - 1) + FI(N - 2) \text{ for all } N > 1$$

Algebraic Axioms

Mathematical systems are defined by axioms. The axioms specify fundamental properties of a system and provide a basis for deriving additional properties that are implied by the axioms. These additional properties are called theorems. In order to establish a valid mathematical system, the set of axioms must be complete and consistent; ie., it must be possible to prove desired results using the axioms, and it must not be possible to prove contradictory results.

A data type is characterized as a set of objects and a set of permissible operations on those objects. The term "abstract data type" (or "data abstraction") refers to the fact that permissible operations on the data objects are emphasized, while representation details of the data objects are suppressed.

Axiomatic specification of the last-in first-out (LIFO) property of stack objects is specified as follows:

SYNTAX:

OPERATION	DOMAIN	RANGE
NEW	( ) →	STACK
PUSH	(STACK, ITEM) →	STACK
POP	(STACK) →	STACK
TOP	(STACK) →	ITEM

EMPTY (STACK) → BOOLEAN

AXIOMS:

(stk is of type STACK, itm is of type ITEM)

1) **EMPTY(NEW) = true**
2) **EMPTY(PUSH(stk, itm)) = false**
3) **POP(NEW) = error**
4) **TOP(NEW) = error**
5) **POP(PUSH(stk, itm)) = stk**
6) **TOP(PUSH(stk, itm)) = itm**

Algebraic specification of the LIFO property

Intuitive definitions of the stack operations are:

- **NEW creates a new stack**
- **PUSH adds a new item to the top of a stack**
- **TOP returns a copy of the top item**
- **POP removes the top item**
- **EMPTY tests for an empty stack**

Operation NEW yields a newly created stack. PUSH requires two arguments, a stack and an item; it produces a stack. POP requires a stack as its arguments and yields a stack. TOP requires a stack and produces an item. EMPTY tests for an empty stack it requires a stack and provides a Boolean value.

The axioms in the above figure can be stated in English as follows:

1. **A new stack is empty.**
2. **A stack is not empty immediately after pushing an item onto it.**
3. **Attempting to pop a new stack results in an error.**
4. **There is no top item on a new stack.**
5. **Pushing an item onto a stack and immediately popping it off leaves the stack unchanged.**
6. **Pushing an item onto a stack and immediately requesting the top item returns the item just pushed onto the stack.**

Thus the axioms specify the fundamental characteristics of stacks in a precise and unambiguous manner. The intuitively defined entities NEW, PUSH, POP, TOP, and EMPTY are precisely described using a state-oriented approach as follows:

NEW

Purpose: Create a new stack
Exception: Memory_Full = true
Effects: Valid_Stack(NEW) = true
Number_Items(NEW) = 0

EMPTY(stk)

Purpose: Test stk for empty property
Exception: 'Valid_Stack(stk)' = false
Effects: if 'Number_Items(stk)' = 0 then true else false

PUSH(stk,item)

Purpose: Place item on stack
Exception: 'Valid_Stack(stk)' = false
'Number_Items(stk)' = MAX
Effects: if 'Number_Items(stk)' = MAX then error
else Number_Items(stk) = 'Number_Items(stk)' + 1

POP(stk)

Purpose: Delete top item from stk
Exception: 'Valid_Stack(stk)' = false
"Number_Items(stk)' = 0
Effects: if 'Number_Items(stk)' = 0 then error else {stk = 'stk' – TOP(stk)
Number_Items(stk) = 'Number_Items(stk)' – 1}

TOP(stk)
Purpose: Return a copy of top item on stk
Exception: 'Valid_Stack(stk)' = false
'Number_Items(stk)' = 0
Effects: if 'Number_Items(stk)' = 0 then error
Else Number_Items(stk) = 'Number_Items(stk)'}

Definition of STACK function behavior

An entity delimited by quotes [eg: 'Valid_Stack(stk)'] denotes the state of the system immediately before the operation in which it is contained. An unquoted entity refers to the state of the system immediately following the containing operation. This technique combines the advantages of the algebraic approach (precise specification of interactions among operations) and the finite-state approach (precise specification of the behavior of the individual operations).

The first-in first-out (FIFO) property of queues is specified as follows:

SYNTAX:

OPERATION	**DOMAIN**	**RANGE**
NEW	**( ) →**	**QUEUE**
ADD	**(QUEUE,ITEM) →**	**QU**EUE
FRONT	(QUEUE) →	ITEM
REMOVE	(QUEUE) →	QUEUE
EMPTY	(QUEUE) →	BOOLEAN

AXIOMS:

(que is of type QUEUE, itm is of type ITEM)

1) EMPTY(NEW) = true
2) EMPTY(ADD(que,itm)) = false
3) FRONT(NEW) = error
4) REMOVE(NEW) = error
5) FRONT(ADD(que,itm)) = if EMPTY(que) then itm else FRONT(que)
6) REMOVE(ADD(que,itm)) = if EMPTY(que) then NEW else ADD(REMOVE(que),itm)

Algebraic specification of the FIFO property

NEW creates a new queue, ADD adds an item to the rear of a queue, FRONT returns a copy of the front item in a queue without deleting it, REMOVE deletes the front item, and EMPTY tests for an empty queue.

Regular Expressions

Regular expressions can be used to specify the syntactic structure of symbol strings. Because many software products involve processing of symbol strings, regular expressions provide a powerful and widely used notation in software engineering. Every set of symbol strings specified by a regular expression defines a formal language. Regular expressions can thus be viewed as language generators.

The Rules for forming regular expressions are quite simple:

1) Atoms: The basic symbols in the alphabet of interest form regular expressions.
2) Alternation: If R1 and R2 are regular expressions, then (R1 | R2) is a regular expression.

3) Composition: If R1 and R2 are regular expressions, then (R1 R2) is a regular expression.
4) Closure: If R1 is a regular expression, then (R1)* is a regular expression.
5) Completeness: Nothing else is a regular expression.

An alphabet of basic symbols provides the atoms. The alphabet is made up of whatever symbols are of interest in the particular application. Alternation, (R1 | R2), denotes the union of the languages (sets of symbol strings) specified by R1 and R2, and composition, (R1 R2), denotes the language formed by concatenating strings from R2 onto strings from R1. Closure, (R1)*, denotes the language formed by concatenating zero or more strings from R1 with zero or more strings from R1.

Observe that rules 2, 3, and 4 are recursive; ie., they define regular expressions in terms of regular expressions. Rule 1 is the basis rule, and rule 5 completes the definition of regular expressions. Examples of regular expressions follow:

1) Given atoms a and b, then a denotes the set { a } and b denotes the set { b }.
2) Given atoms a and b, then (a | b) denotes the set { a, b }.
3) Given atoms a, b, and c, then ((a | b) | c) denotes the set { {a,b}, c}.
4) Given atoms a and b, then (a b) denotes the set { ab } containing one element ab.
5) Given atoms a, b, and c, then ((a b) c) denotes the set { abc } containing one element abc.
6) Given atom a, then (a)* denotes the set {e, a, aa, aaa, …}, where e denotes the empty string.

Complex regular expressions can be formed by repeated application of recursion rules 2, 3, and 4:

1) (a (b | c)) denotes {ab, ac}.
2) (a | b)* denotes {e, a, b, aa, bb, ab, ba, aab, …}
3) ((a (b | c)))* denotes (e, ab, ac, abab, acac, abac, acab, ababac, …}

Closure, (R1)*, denotes zero or more concatenations of elements from R1. A commonly used notation is (R1)+, which denotes one or more concatenations of elements in R1. The “*” and “+” notations are called the *Kleene star* and *Kleene plus* notations.

State-Oriented Notations

Decision Tables

Decision tables provide a mechanism for recording complex decision logic. Decision tables are widely used in data processing applications and have an extensively developed literature.

	Decision rules			
	Rule 1	Rule 2	Rule 3	Rule 4
(Condition stub)		(Condition entries)		
(Action stub)		(Action entries)		

Basic elements of a decision table

A decision table is segmented into four quadrants: condition stub, condition entry, action stub, and action entry. The condition stub contains all of the conditions being examined. Condition entries are used to combine conditions into decision rules. The action stub describes the actions to be taken in response to decision rules, and the action entry quadrant relates decision rules to actions.

In a limited-entry decision table, Y denotes “yes” , N denotes “no”, - denotes “don’t care”, and X denotes “perform action”.

	1	2	3	4
Credit limit is satisfactory	Y	N	N	N
Pay experience is favorable	-	Y	N	N
Special clearance is obtained	-	-	Y	N
Perform approve order	X	X	X	
Go to reject order				X

Limited entry decision table

Orders are approved if the credit limit is not exceeded, or if the credit limit is exceeded but past experience is good, or if a special arrangement has been made. If none of these conditions hold, the order is rejected.

If more than one decision rule has identical (Y,N,-) entries, the table is said to be ambiguous. Ambiguous pairs of decision rules that specify identical actions are said to be redundant, and those specifying different actions are contradictory. Contradictory rules permit specification of nondeterministic and concurrent actions. There are 2**N combinations of conditions in a table that has N condition entries.

	Decision rule			
	Rule 1	Rule 2	Rule 3	Rule 4
C1	Y	Y	Y	Y
C2	Y	N	N	N
C3	N	N	N	N
A1	X			
A2		X		
A3			X	X

An ambiguous decision table

C1	Y		N
C2		Y	N
C3			Y
A1		X	
A2	X		
A3			X

An incomplete and over-specified decision table

Event Tables

Event tables specify actions to be taken when events occur under different set of conditions. A two-dimensional event table relates actions to two variables; f(M, E) = A, where M denotes the current set of operating conditions, E is the event of interest, and A is the action to be taken. Tables of higher dimension can be used to incorporate more independent variables.

An event table consists of the actions to be taken are related to the current mode of operation and the events that may occur within modes. Thus, if the system is in start-up mode SU and event E13 occurs, action A16 is to be taken; f(SU, E13) = A16. Special notations can be invented to suit particular situations.

For example, actions separated by semicolons (A14; A32) might denote A14 followed sequentially by A32, while actions separated by commas (A6, A2) might denote concurrent activation of A6 and A2.

Similarly, a dash (-) might indicate no action required, while an X might indicate an impossible system configuration (eg., E13 cannot occur in steady-state mode).

Mode	Event				
	E13	E37	E45	…	…
Start-up	A16	-	A14; A32		
Steady	X	A6, A2	-		
Shut-down	…	…	…		
Alarm	…	…	…		

A two-dimensional event table

Transition Tables

Transition tables are used to specify changes in the state of a system as a function of driving forces. The state of a system summarizes the status of all entities in the system at a particular time. Given the current state and the current conditions, the next state results; if, when in state Si, condition Cj results in a transition to state Sk, we say f(Sj, Cj) = Sk.

Current State	Current Input	
	a	b
S0	S0	S1
S1	S1	S0

A simple transition table

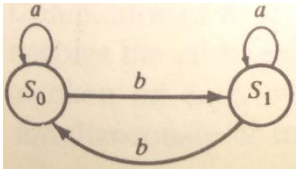
Given the current state S1 and current input b, the system will go to state S0; ie., f(S1, b) = S0. A transition table that is augmented to indicate actions to be performed and outputs to be generated in the transition to the next state.

Present state	Input	Action	Output	Next State
S0	a b			S0 S1
S1	a b			S0 S1

An augmented transition table

Transition diagrams are alternative representations for transition tables. In a transition diagram, states become nodes in a directed graph and transitions are represented as arcs between nodes. Arcs are labeled with conditions that cause transitions.

Transition diagrams and transition tables are representations for finite state automata. The theory of finite state automata is rich, complex, and highly developed.



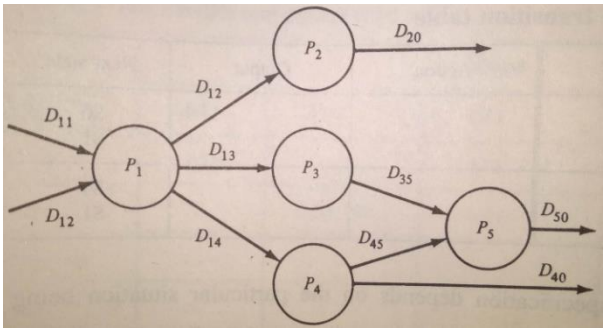
Transition diagram

Decision tables, event tables, and transition tables are notations for specifying actions as functions of the conditions that initiate those actions. Decision tables specify actions in terms of complex decision logic, event tables relate actions to system conditions, and transition tables incorporate the concept of system state.

Finite State Mechanisms

Data flow diagrams, regular expressions, and transition tables can be combined to provide a powerful finite state mechanism for functional specification of software systems. The following figure depicts

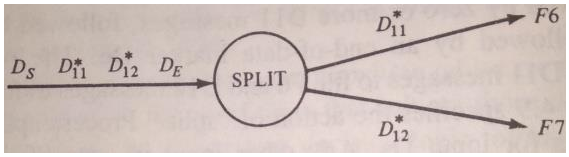
the data flow diagram for a software system consisting of a set of processes interconnected by data streams. Each of the data streams can be specified using a regular expression and each of the processes can be described using a transition table.



A network of data streams and processes

Regular expressions have been quite simple; one can describe highly complex data streams using regular expressions. For example, a data stream might be specified as

(X(((X Y))* | Z) | ((((X | Y) Z)) + Y))



Present state	Input	Actions	Outputs	Next state
S0	D_S	Open F6 Open F7		S1
S1	D_{11}	Write F6	D_{11}:F6	S1
	D_{12}	Close F6 Write F7	D_{12}:F7	S2
	D_E	Close F6 Close F7		S0
S2	D_{12}	Write F7	D_{12}:F7	S2
	D_E	Close F7		S0

Specification of the "Split" process

The following figure specifies a system for which the incoming data stream consists of a start marker Ds, followed by zero or more D11 messages, followed by zero or more D12 messages, followed by an end-of-data marker De. The purpose of process "split" is to route D11 messages to file F6 and D12 messages to file F7. The transition table in the above figure specifies the action of "split". Process split starts in initial state S0 and waits for input Ds.

Any other input in state S0 is ignored (alternatively, other inputs could result in error processing). Arrival of input Ds in state S0 triggers the opening of files F6 and F7 and transition to state S1. In S1, D11 messages are written to F6 until either a D12 message or a De message is received (note that a Ds De data stream with no D11 messages and no D12 message is possible). On receipt of a D12 message, process split closes Fchnique6, writes the message D12 to F7 and goes to state S2.

In state S2, split writes zero or more D12 messages to F7, then, on receipt of the end-of-data marker, De, closes F7 and returns to state S0 to await the next transmission.

One drawback of fine-state mechanisms is the so-called **state explosion phenomenon**. Complex systems may have large numbers of states and many combinations of input data. Specifying the

behavior of a software system for all combinations of current state and current input can become unwieldy.

Hierarchical decomposition is one technique for controlling the complexity of a finite-state specification. Using hierarchical decomposition, higher level concepts are given names, the meaning and validity of which are established at a lower level.

Petri Nets

Petri nets were invented in the 1960s by Carl Petri at the University of Bonn, West Germany. Petri nets have been used to model a wide variety of situations; they provide a graphical representation technique, and systematic methods have been developed for synthesizing and analyzing Petri nets. Petri nets were invented to overcome the limitations of finite state mechanisms in specifying parallelism.

Concurrent Systems are designed to permit simultaneous execution of the software components, called tasks or processes, on multiple processors. Alternatively, execution of tasks can be interleaved on a single processor. Concurrent tasks must be synchronized to permit communication among tasks that operate at differing execution rates, to prevent simultaneous updating of shared data, and to prevent deadlock. Deadlock occurs when all tasks in the system are waiting for data or other resources that can only be supplied by tasks that are waiting on other tasks. The fundamental problem of concurrency are thus synchronization, mutual exclusion, and deadlock.

A Petri net is represented as a bipartite directed graph. The two types of nodes in a Petri net are called places and transitions. Places are marked by tokens; a Perti net is characterized by an initial marking of places and a firing rule. A firing rule has two aspects; a transition is enabled if every input places has at least one token. An enabled transition can fire; When a transition fires; each input places of that transition loses one token, and each output places of that transition gains one token. A marked Petri net is formally defined as a quadruple, consisting of a set of places P, a set of transitions T, a set of arcs A, and a marking M. C = (P,T,A,M), where

$p = \{p1, p2, \ldots Pm\}$

$T = \{t1, t2, \ldots tn\}$

$A \subset \{P \times T\} \cup \{T \times P\} = \{(pi, tj) \ldots (tk, pl) \ldots\}$

$M: P \rightarrow I$; ie., $M(p1, p2, \ldots pm) = (i1, i2, \ldots im)$

Marking M associates an integer number of token ik with each places pk. When a transition fires, the marking of places p changes from M(p) to M'(p) as follows:

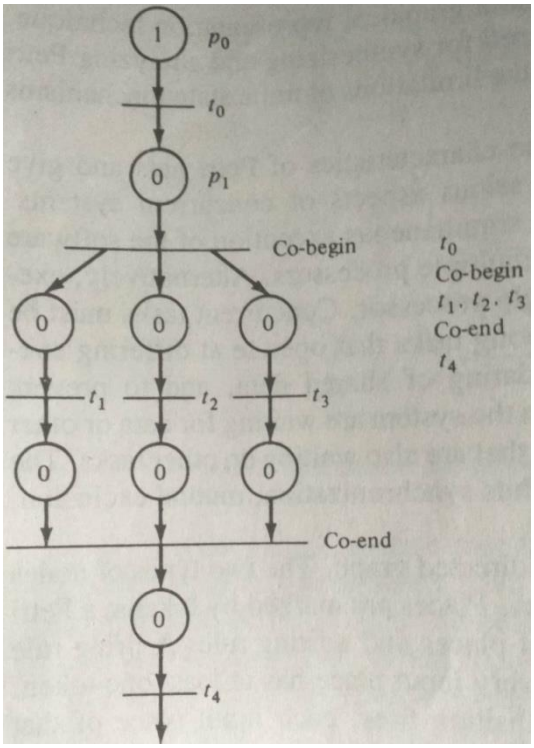
$M'(p) = M(p) + 1$ if $p \in O(t)$ and $p \notin I(t)$

$M'(p) = M(p) - 1$ if $p \notin I(t)$ & $p \in O(t)$

$M'(p) = M(p)$ otherwise

where I(t) is the set of input places of transition t and O(t) is the set of output places of transition t.

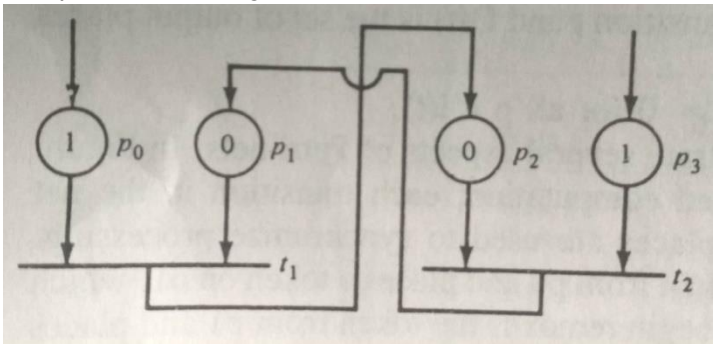
A transition t is enabled if $M(p) > 0$ for all $p \in I(t)$.



Petri net model of concurrent processes t1, t2, t3 (initial marking)

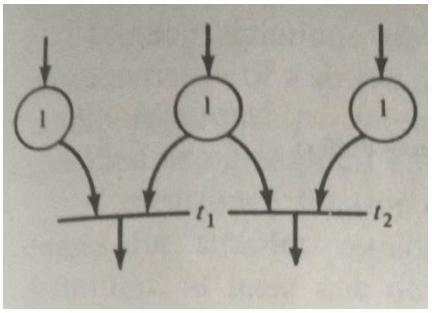
In the above figure, the Petri net models the indicated computation; each transition in the net corresponds to a task activation and places are used to synchronize processing. Completion of t0 removes the initial token from p0 and places a token on p1, which enables the co-begin. Firing of the co-begin removes the token from p1 and places a token on each of p2, p3, and p4.

This enables t1, t2, and t3; they can fire simultaneously or in any order. This corresponds to concurrent execution of tasks t1, t2, and t3. When each of tasks t1, t2, and t3 completes its processing, a token is placed on the corresponding output place, p5, p6, or p7. Co-end is not enabled until all three tasks complete their processing. Firing of co-end removes the tokens from p5, p6, and p7 and places a token on p8, thus enabling t4 which can then fire.



A deadlocked Petri net

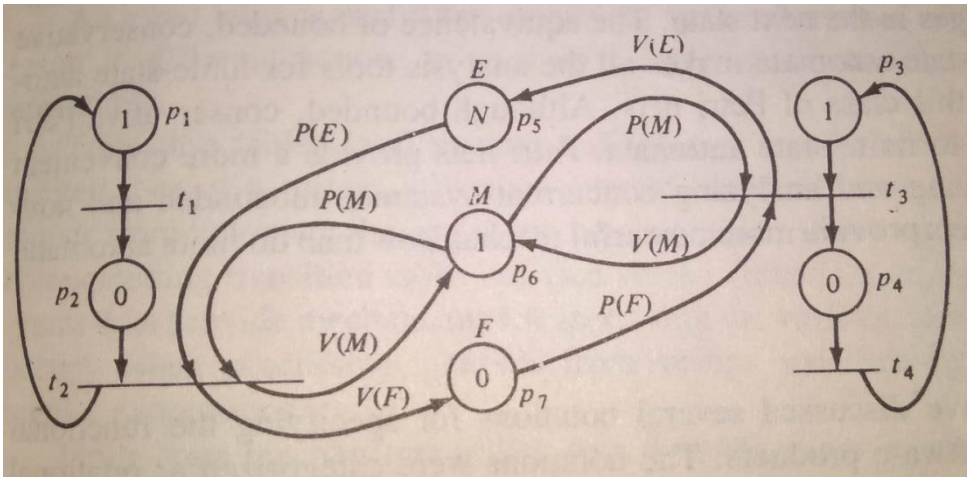
The above figure illustrates a deadlock situation. Both t1 and t2 are waiting for the other to fire and neither can proceed.



A conflict situation

Conflict in Petri nets provides the basis for modeling of mutual exclusion. Conflict is illustrated in above figure; both t1 and t2 are enabled, but only one can fire. Firing one will disable the other.

As an example of mutual exclusion, a Petri net of the producer/consumer problem is illustrated as:



Initial marking for the producer/consumer Petri Net

```
SEMAPHORE      E, F, M;
INITIALLY      E := N, F := 0, M := 1;

LOOP: t1 (Produce)
      P(E), P(M)
      t2 (Place in buffer)
      V(F), V(M)
END_LOOP

LOOP: P(F), P(M)
      t3 (Remove from buffer)
      V(E), V(M)
      t4 (Consume)
END_LOOP

P(K) ≡ if K > 0 then K := K - 1 else wait
V(K) ≡ K := K + 1;
       Initiate one waiting process
```

Semaphore solution to the producer/consumer problem

LANGUAGES AND PROCESSORS FOR REQUIREMENTS SPECIFICATION

A number of special-purpose languages and processors permit concise statement and automated analysis of requirements specification for software. Some are graphical in nature, while others are textual. Some are manually applied, and others have automated processors.

PSL/PSA

The Problem Statement Language (PSL) was developed by Professor Daniel Teichrow at the University of Michigan. The Problem Statement Analyzer (PSA) is the PSL processor. PSL and PSA were developed as components of ISDOS and so this system is sometimes referred to as the ISDOS system. PSL is based on a general model of systems. This model describes a system as a set of objects, where each object may have properties, and each property may have property values. Objects may be interconnected; the connections are called relationships.

The objective of PSL is to permit the information appears in a *Software Requirements Specification*. In PSL, system descriptions can be divided into eight major aspects:

1. System input/output flow
2. System structure
3. Data structure
4. Data derivation
5. System size and volume
6. System dynamics
7. System properties
8. Project management

The system input/output flow aspect deals with the interaction between a system and its environment. System structure is concerned with the hierarchies among objects in a system. The data structure aspect includes all the relationships that exist among data. The data derivation specifies which data objects are involved in particular processes in the system. Data derivation describes data relationships that are internal to a system.

The system size and volume aspect is concerned with the size of the system and those factors that influence the volume of processing required. The system dynamics presents the manner in which the system “behaves” over time. The project management aspect requires that project-related information, as well as product-related information. This involves identification of the people involved, their responsibilities, schedules, cost estimates, etc.

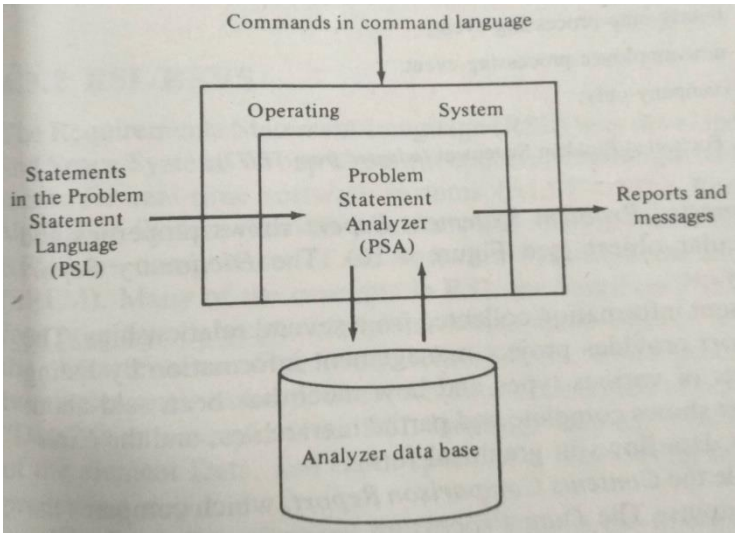
PSA is an automated analyzer for processing requirements stated in PSL. PSA operates on a database of information collected from a PSL description. The PSA system can provide reports in four categories: database modification reports, reference reports, summary reports, and analysis reports.

Database Modification Report list changes that have been made since the last report, together with diagnostic and warning messages for error correction and recovery.

Reference Reports include the *Name List Report*, which lists all the objects in the database with types and dates of last change. The *Formatted Problem Statement Report* shows properties and relationships for object. The *Dictionary Report* provides a data dictionary.

Summary Reports present information collected from several relationships. The Data Base Summary Report provides project management information by listing the total number of objects. The *Structure Report* shows complete and partial hierarchies, and the *External Picture Report* depicts data flows in graphical form.

Analysis Reports include the *Contents Comparison Report*, which compares the similarity of inputs and outputs. The *Data Processing Interaction Report* can be used to detect gaps in information flow and unused data objects. The *Processing Chain Report* shows the dynamic behavior of the system.



Structure of the Problem Statement Analyzer

PROCESS	hourly-employee-processing
DESCRIPTION:	this process performs those actions needed to interpret time cards to produce a pay statement for each hourly employee.;
GENERATES:	pay-statement, error-listing, hourly-employee-report;
RECEIVES:	time-card;
SUBPARTS ARE:	hourly-paycheck-validation, hourly-emp-update, h-report-entry-generation, hourly-paycheck-production;
PART OF:	payroll-processing;
DERIVES:	pay-statement
USING:	time-card, hourly-employee-record;
DERIVES:	hourly-employee-report
USING:	time-card, hourly-employee-record;
DERIVES:	error-listing
USING:	time-card, hourly-employee-record;

PROCEDURE:
1. compute gross pay from time card data.
2. compute tax from gross pay.
3. subtract tax from gross pay to obtain net pay.
4. update hourly employee record accordingly.
5. update department record accordingly.
6. generate paycheck.

note: if status code specifies that the employee did not work this week, no processing will be done for this employee;

HAPPENS:	number-of-payments TIMES-PER pay period;
TRIGGERED BY:	hourly-emp-processing-event;
TERMINATION-CAUSES:	new-employee-processing-event;
SECURITY-IS:	company-only;

Example of a PSL Formatted Problem Statement

Advantages:

- PSL/PSA is a useful tool for documenting & communicating software requirements.
- PSL/PSA not only supports requirements analysis, but also supports design.
- PSL/PSA has been used from commercial data processing applications to air defense systems.

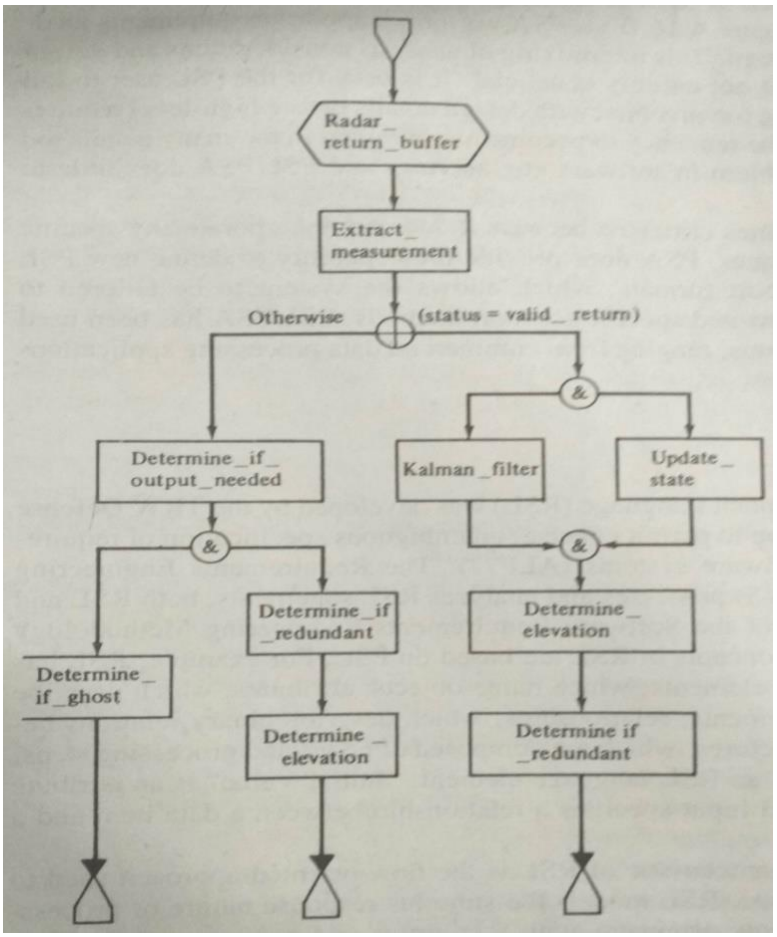
RSL/REVS (Requirements Statement Language/Requirements Engineering Validation System)

The Requirements Statement Language (RSL) was developed by TRW Defense and Space Systems Group to permit concise, unambiguous specification of requirements for real-time software systems.

Requirements Engineering Validation System (REVS) processes and analyzes RSL statements; both RSL/REVS are components of Software Requirements Engineering Methodology (SREM).

Many of the concepts in RSL are based on PSL. RSL has four primitive concepts: Elements which name objects; Attributes which describe the characteristics of elements; Relationships which describe binary relationship between elements; and Structures, which are composed of nodes and processing steps.

The fundamental characteristic of RSL is the flow-oriented approach used to describe real-time systems. RSL models the Stimulus-Response nature of process control systems. Flow approach provides direct testability of requirements. Flows are specified by Requirements Networks (R-NETS). R-NETS have both graphical & textual representations. AND node (&) specify parallel operations. OR node (+) have an Otherwise path. RSL includes predefined element types, relationships, attributes and structures.



Graphical Representation of an R-NET

```
R_NET: PROCESS_RADAR_RETURN
  STRUCTURE:
    INPUT_INTERFACE_RADAR_RETURN_BUFFER
    EXTRACT_MEASUREMENT
    DO (STATUS = VALIDRETURN)
      DO UPDATE_STATE AND KALMAN_FILTER END
      DETERMINE ELEVATION
      DETERMINE_IF_REDUNDANT
      TERMINATE
    OTHERWISE
      DETERMINE_IF_OUTPUT_NEEDED
      DO DETERMINE_IF_REDUNDANT
        DETERMINE_ELEVATION
        TERMINATE
      AND DETERMINE_IF_GHOST
      TERMINATE
      END
    END
  END
```

Textual Representation of an R-NET

Predefined elements include Alpha, Data, & R_NET. Alpha specifies functional characteristics of processing step in R-NET. Alphas are described by Inputs, Outputs, & Descriptions. Data specifies data elements at conceptual level: Input_To & Output_From. Attributes – Initial Value & Includes. Elements – Description.

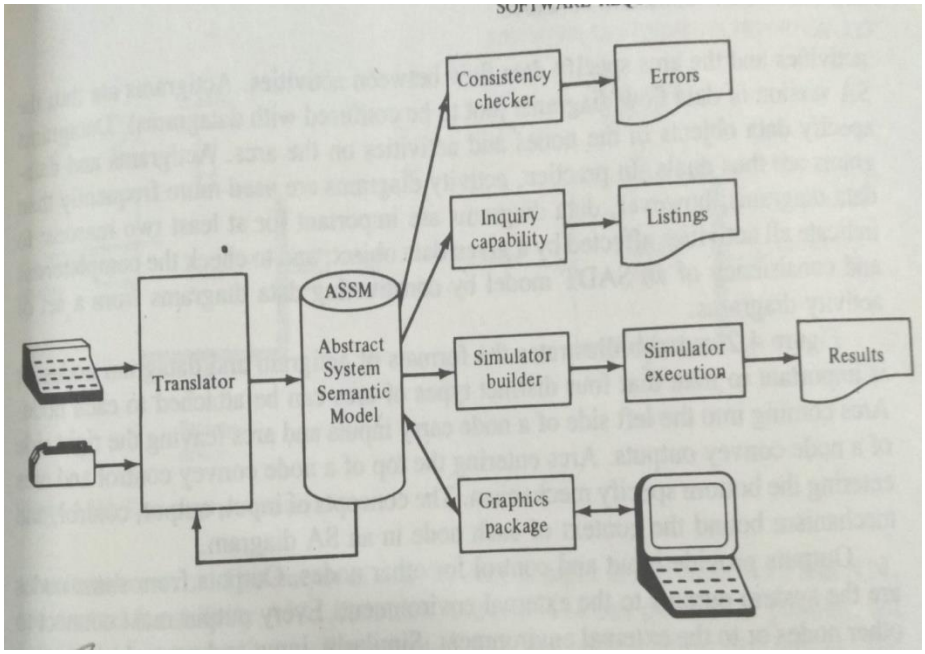
The three major components of REVS: A translator for RSL, A centralized data base, the Abstract System Semantic Model (ASSM), A set of automated tools for processing information in ASSM

```
ALPHA: EXTRACT_MEASUREMENT.
  INPUTS: CORRELATED_RETURN
  OUTPUTS: VALID_RETURN, MEASUREMENT.
  DESCRIPTION: "DOES RANGE SELECTION PER CISS REFERENCE 2-7."
ALPHA: DETERMINE_IF_REDUNDANT.
  INPUTS: CORRELATED_RETURN
  OUTPUTS: REDUNDANT_IMAGE.
  DESCRIPTION: "THE IMAGE OF THE RADAR RETURN IS ANALYZED TO
        DETERMINE IF IT IS REDUNDANT WITH ANOTHER IMAGE."
DATA: MEASUREMENT.
  INCLUDES: RANGE_MARK_TIME, AMPLITUDE,
      RANGE_VARIANCE, RD_VARIANCE,
      R_AND_RD_CORRELATION.
  OUTPUT FROM: ALPHA EXTRACT_MEASUREMENT.
  DESCRIPTION: "THIS IS THE ESSENCE OF THE INFORMATION IN THE RETURN."
ORIGINATION_REQUIREMENT:
  DPSPR_3.2.2.A_FUNCTIONAL.
  DESCRIPTION: "ACTION: SEND RADAR ORDER INFORMATION: INFORMATION:
      RADAR ORDER, IMAGE(REDUNDANT)."
  TRACES TO:  ALPHA COMMAND_PULSES
      ALPHA DETERMINE_IF_REDUNDANT
      MESSAGE RADAR_ORDER_MESSAGE
      DATA REDUNDANT_IMAGE
      ENTITY_CLASS IMAGE
```

Examples of ALPHA, DATA, and ORIGINATING_REQUIREMENT

```
DEFINE ELEMENT_TYPE: INPUT_INTERFACE
        (*A port between the data processing system and the
        rest of the system which accepts data from another
        part of the system*).
DEFINE ELEMENT_TYPE: MESSAGE
        (*An aggregation of DATA and FILES that PASS through
        an interface as a logical unit.*).
DEFINE RELATIONSHIP: PASSES
        (*An INPUT_INTERFACE "PASSES" a logical aggregation
        of data called a MESSAGE from the outside system into
        the data processing system*).
COMPLEMENTARY RELATIONSHIP: PASSED_BY.
SUBJECT: INPUT_INTERFACE.
OBJECT: MESSAGE.
```

Examples of the DEFINE attribute from RSL



Structure of the REVS Processor

ASSM is a relational database. ASSM include an interactive graphics package to specify flow paths. Static checkers to check completeness and consistency of the information used throughout the system. An automated simulation package that generates & executes simulation models of the system. REVS is a large, complex software tool. REVS is cost-effective only for large, complex real-time systems.

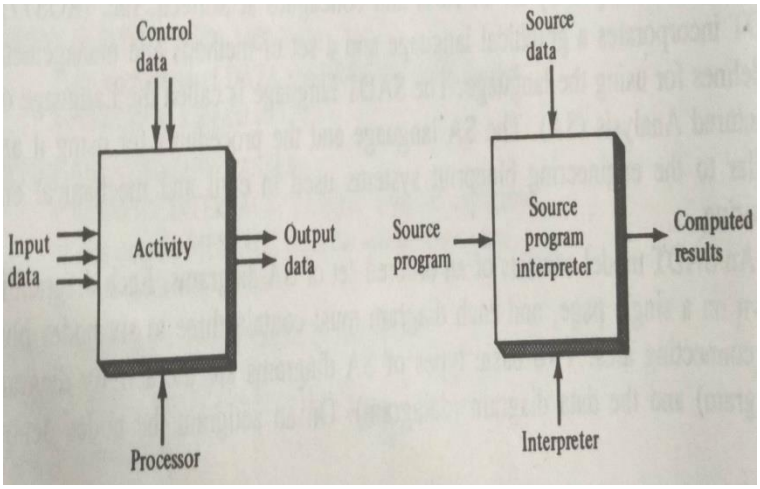
SADT (Structured Analysis and Design Technique)

SADT was developed by D.T.Ross and colleagues @ Softech, Inc. SADT incorporates a graphical language & a set of methods & management guidelines for using the language called the Language of **Structured Analysis (SA)**.

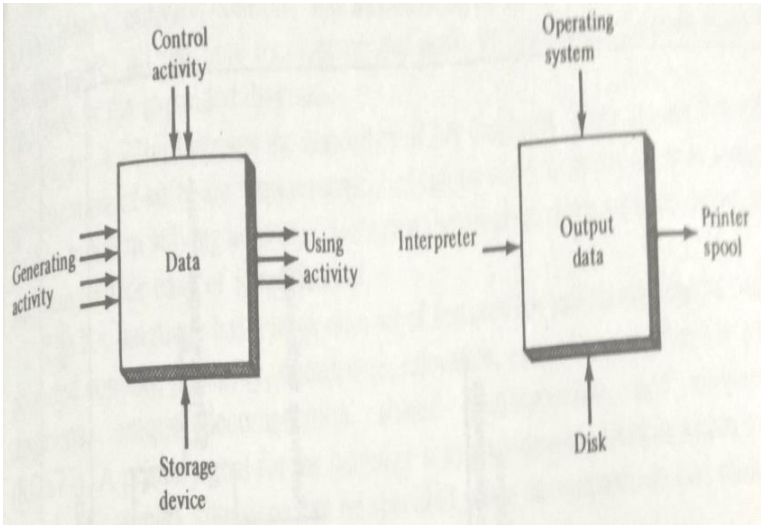
SA language are similar to engineering blueprint systems used in civil & mechanical engineering. SADT model consists of an ordered set of SA diagrams. Each diagram is drawn on a single page, & must contain 3-6 nodes plus interconnecting arcs. Two basic types of SA diagrams: Activity diagram (Actigram) and Data diagram (Datagram).

Actigram: nodes denote activities & arcs specify data flow. Actigram is the SA version of Data Flow Diagrams (DFD). Datagrams specify data objects in the nodes and activities on the arcs. Activity diagrams are used more frequently than data diagrams. Data diagrams are important for two reasons:

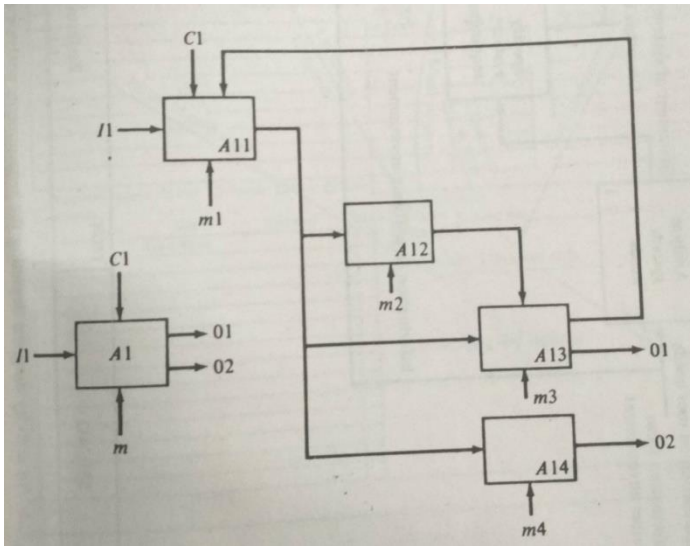
1. To indicate all activities affected by a given data object
2. To check the completeness and consistency of an SADT model



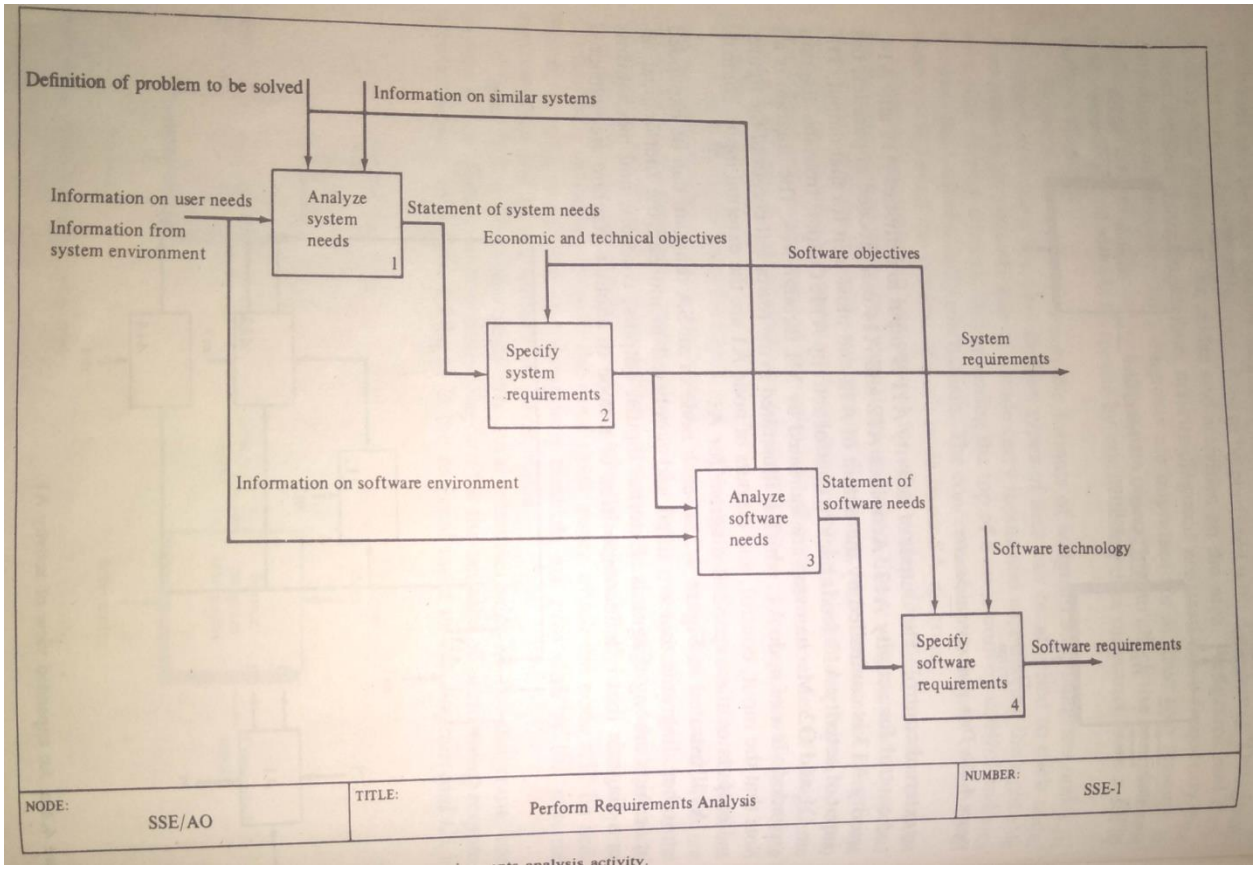
Activity Diagram Components



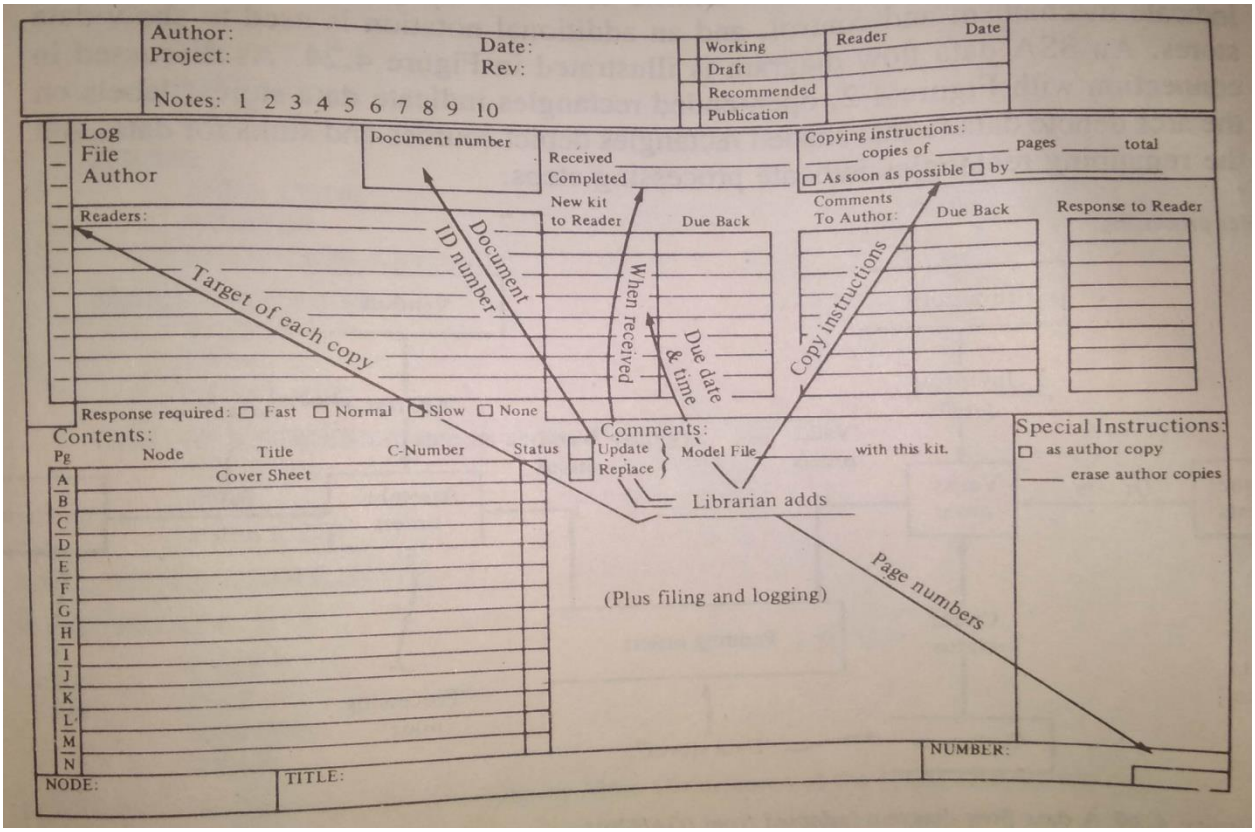
Data Diagram Components



An expanded view of Activity A1



An Activity Diagram Depicting the Requirements Analysis Activity



Cover Sheet for an SADT Model

Provides notations and set of techniques for understanding and recording complex requirements in clear and concise manner. Top-down methodology in decomposing high-level nodes into subordinate diagrams

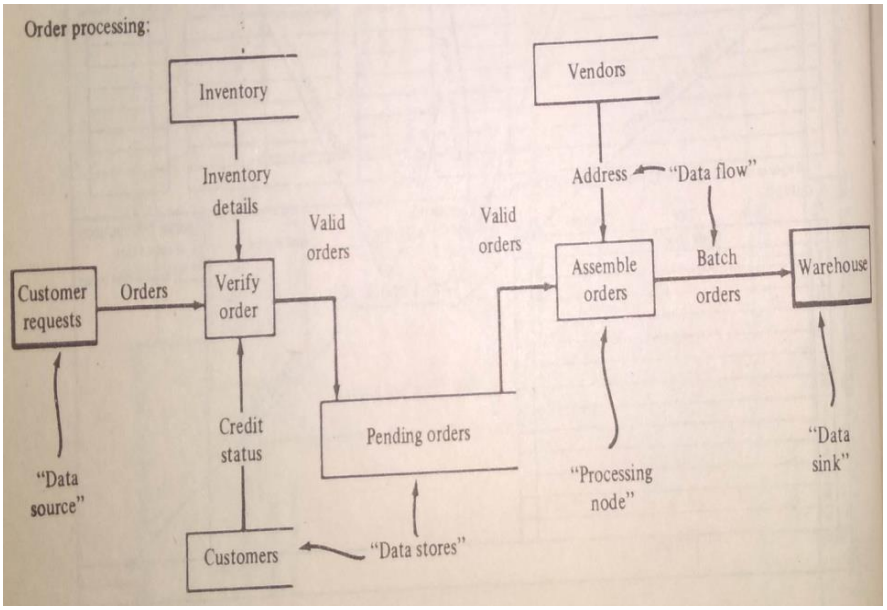
SSA (Structured System Analysis)

Two similar versions of SSA described by Gane and Sarson and by DeMarco. DeMarco version is similar to Gane and Sarson version, but it does not incorporate database concept. SSA is used in traditional data processing environments.

SSA Uses a graphical language to build models of systems. Unlike SADT, it uses database concepts. SSA does not provide many structural mechanisms available in SADT. Four basic **features** in SSA:

1. Data flow diagrams
2. Data dictionaries
3. Procedure logic representations
4. Data store structuring techniques

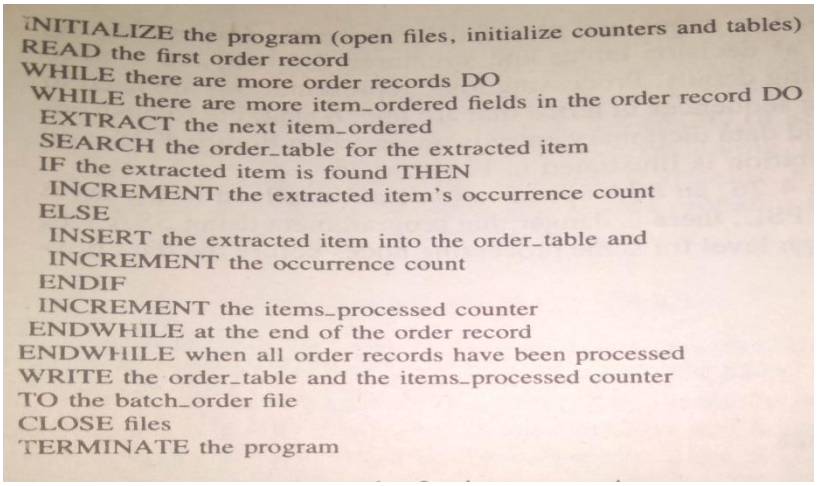
SSA data flow diagrams are similar to SADT actigrams. Open-ended rectangles indicate data stores. Labels on the arcs denote data items. Shaded rectangles depict sources and sinks. Rectangles indicate processing steps. Data dictionary is used to define and record data elements. Processing logic representations viz. decision tables & structured English – used to specify algorithmic processing details. Process logic representations precisely specify processing sequences that are understandable to customers and developers.



A Data Flow Diagram

```
DATA FLOW: ORDER
COMPOSITION:
        CUSTOMER_IDENTITY
        ORDER_DATE
        ITEM_ORDERED +
                CATALOG_NUMBER
                ITEM_NAME
                UNIT PRICE
                QUANTITY
                TOTAL_COST
DATA STORE: CUSTOMER_IDENTITY
COMPOSITION:
        NAME
                FIRST
                MIDDLE
                LAST
        PHONE
                AREA_CODE
                NUMBER
                EXTENSION (Optional)
        SHIP_TO_ADDRESS
                STREET
                CITY
                STATE-ZIP
        BILL_TO_ADDRESS (Same as above if empty)
                STREET
                CITY
                STATE-ZIP
DATA STORE: ORDER_DATE
COMPOSITION:
                TIME
                DAY
                MONTH
                YEAR
NOTE: The " + " Notation Denotes One or More Occurrences of the ITEM_ORDERED Field
```

Data Dictionary Entries in SSA



```
INITIALIZE the program (open files, initialize counters and tables)
READ the first order record
WHILE there are more order records DO
 WHILE there are more item_ordered fields in the order record DO
  EXTRACT the next item_ordered
  SEARCH the order_table for the extracted item
  IF the extracted item is found THEN
   INCREMENT the extracted item's occurrence count
  ELSE
   INSERT the extracted item into the order_table and
   INCREMENT the occurrence count
  ENDIF
  INCREMENT the items_processed counter
 ENDWHILE at the end of the order record
ENDWHILE when all order records have been processed
WRITE the order_table and the items_processed counter
TO the batch_order file
CLOSE files
TERMINATE the program
```

An SSA Processing Logic Representation

SSA can be refined to level of detailed design before the DFD & Data dictionary are completed. Important feature of SSA is use of a relational model to specify data flows and data stores. Relations are composed from fields of data records. Fields are called domains of the relation. If a record has N fields, the relation is called an N-tuple.

GIST

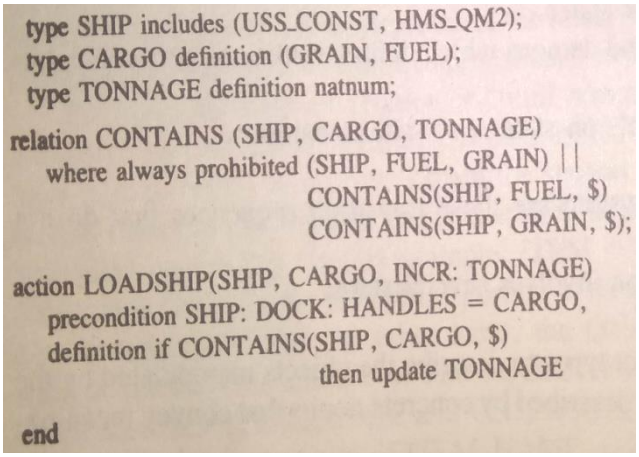
Gist is a formal specification language developed at the USC/Information Sciences Institute by R.Balzar & Colleagues. Gist is a textual language based on a relational model of objects and attributes.

A Gist specification is a formal description of valid behaviors of a system. Specification is composed of 3 parts:

1. Object types and & relationships between them (determines set of possible states).
2. Actions and demons (define transitions between possible states).
3. Constraints on states and state transitions.

Steps in preparing Gist specification:

1. Identify a collection of object types manipulated by process, described by concrete nouns.
2. Identify individual objects within types. This is not a variables, used to describe constraints or dynamic aspects of a process.
3. Identify relationships in which objects can be related to one another (include “connects to”, “is part of” & “derived from”).
4. Identify types & relationships.
5. Identify static constraints on types & relationships. Static constraints identify processing states that never arise.
6. Identity actions that can change state of the process.
7. Identify dynamic constraints.
8. Identify active participants in the process and group them into classes.



```
type SHIP includes (USS_CONST, HMS_QM2);
type CARGO definition (GRAIN, FUEL);
type TONNAGE definition natnum;

relation CONTAINS (SHIP, CARGO, TONNAGE)
   where always prohibited (SHIP, FUEL, GRAIN) ||
                           CONTAINS(SHIP, FUEL, $)
                           CONTAINS(SHIP, GRAIN, $);

action LOADSHIP(SHIP, CARGO, INCR: TONNAGE)
   precondition SHIP: DOCK: HANDLES = CARGO,
   definition if CONTAINS(SHIP, CARGO, $)
                           then update TONNAGE

end
```

A Gist Specification

Initial Operating Capability (IOC) is a prototype testing facility for software specifications. Purpose of IOC is to validate functional specifications by "executing" them on real test data. IOC consists of evaluator capable of executing specifications expressed in a subset of Gist, & programs that permit entering, editing, & displaying of Gist specifications. IOC permits state initialization, displaying of states, and interactive breakpointing and tracing of test case evaluation. Gist has well-defined, but rather complex syntax.

Conclusion:

- SADT & SSA do not have automated processors

PSL/PSA	Originally developed for data processing applications. Widely used in other applications.
RSL/REVS	Real-time process control systems.
SADT	Interconnection structure of any large, complex system. Not restricted to software systems.
SSA	Gane and Sarson version used in data processing applications that have database requirements. DeMarco version suited to data flow analysis of software systems.
GIST	Object-oriented specification and design. Refinement of specifications into source code.