

MODULE – II

Requirements gathering and analysis—

The system analyst starts *requirements gathering and analysis* activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to remove all ambiguities and inconsistencies from the initial customer perception of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems. The most important requirements problems that the analyst has to identify and eliminate are the problems of anomalies, inconsistencies, and incompleteness. When the analyst detects any inconsistencies, anomalies or incompleteness in the gathered requirements, he resolves them by carrying out further discussions with the end-users and the customers.

Parts of a SRS document--

The important parts of SRS document are:

Functional requirements of the system

Non-functional requirements of the system, and

Goals of implementation

Functional requirements:-

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions $\{f_i\}$. The functional view of the system is shown in fig. 5.1. Each function f_i of the system can be considered as a transformation of a set of input data (i_i) to the corresponding set of output data (o_i). The user can get some meaningful piece of work done using a high-level function.

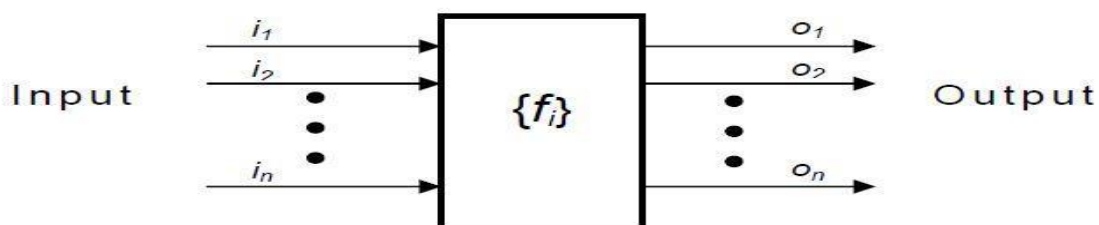


Fig. View of a system performing a set of functions

Nonfunctional requirements:-

Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Goals of implementation:-

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

Identifying functional requirements from a problem description

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high-level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

Example: - Consider the case of the library system, where –

F1: Search Book function

Input: an author's name

Output: details of the author's books and the location of these books in the library

So the function Search Book (F1) takes the author's name and transforms it into book details.

Functional requirements actually describe a set of high-level requirements, where each high-level requirement takes some data from the user and provides some data to the user as an output. Also each high-level requirement might consist of several other functions.

Documenting functional requirements

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.

Properties of a good SRS document

The important properties of a good SRS document are the following:

Concise. The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.

Structured. It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.

Black-box view. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.

Conceptual integrity. It should show conceptual integrity so that the reader can easily understand it.

Response to undesired events. It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.

Verifiable. All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

Problems without a SRS document

The important problems that an organization would face if it does not develop a SRS document are as follows:

Without developing the SRS document, the system would not be implemented according to customer needs.

Software developers would not know whether what they are developing is what exactly required by the customer.

Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.

It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

Problems with an unstructured specification

- It would be very much difficult to understand that document.
- It would be very much difficult to modify that document.
- Conceptual integrity in that document would not be shown.
- The SRS document might be unambiguous and inconsistent.

DECISION TREE--

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the condition.

Example: -

Consider Library Membership Automation Software (LMS) where it should support the following three options:

- ☐ **New member**
- ☐ **Renewal**
- ☐ **Cancel membership**

New member option-

Decision: When the 'new member' option is selected, the software asks details about the member like the member's name, address, phone number etc.

Action: If proper information is entered then a membership record for the member is created and a bill is printed for the annual membership charge plus the security deposit payable.

Renewal option-

Decision: If the 'renewal' option is chosen, the LMS asks for the member's name and his membership number to check whether he is a valid member or not.

Action: If the membership is valid then membership expiry date is updated and the annual membership bill is printed, otherwise an error message is displayed.

Cancel membership option-

Decision: If the 'cancel membership' option is selected, then the software asks for member's name and his membership number.

Action: The membership is cancelled, a cheque for the balance amount due to the member is printed and finally the membership record is deleted from the database.

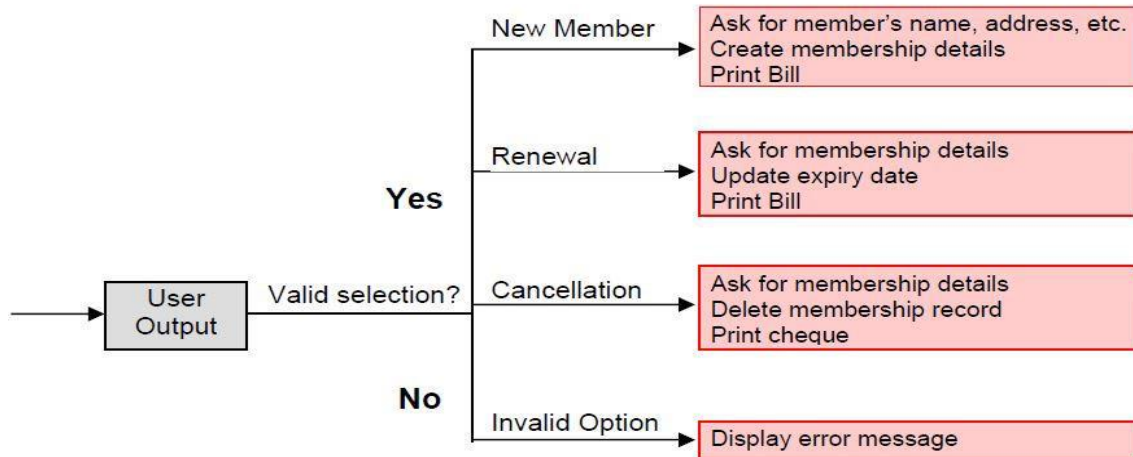


Fig. Decision Tree of LMS

DECISION TABLE--

A decision table is used to represent the complex processing logic in a tabular or a matrix form. The upper rows of the table specify the variables or conditions to be evaluated. The lower rows of the table specify the actions to be taken when the corresponding conditions are satisfied. A

column in a table is called a *rule*. A rule implies that if a condition is true, then the corresponding action is to be executed.

Example: -

Consider the previously discussed LMS example. The following decision table (figure) shows how to represent the LMS problem in a tabular form. Here the table is divided into two parts, the upper part shows the conditions and the lower part shows what actions are taken. Each column of the table is a rule.

Conditions				
Valid selection	No	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes
Actions				
Display error message	x	-	-	-
Ask member's details	-	x	-	-
Build customer record	-	x	-	-
Generate bill	-	x	x	-
Ask member's name & membership number	-	-	x	x
Update expiry date	-	-	x	-
Print cheque	-	-	-	x
Delete record	-	-	-	x

Fig. Decision table for LMS

From the above table you can easily understand that, if the valid selection condition is false then the action taken for this condition is 'display error message'. Similarly, the actions taken for other conditions can be inferred from the table.

The software requirements document--

The software requirements document (sometimes called the software requirements specification or SRS) is the official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements. The requirements document has a diverse set of users, ranging from the senior management of the organization to the engineers responsible for developing the system.

IEEE standard suggests the following structure for requirements documents:

1. Introduction

1.1 Purpose of the requirements document

- 1.2 Scope of the product
- 1.3 Definitions, acronyms and abbreviations
- 1.4 References
- 1.5 Overview of the remainder of the document
- 2. General description
 - 2.1 Product perspective
 - 2.2 Product functions
 - 2.3 User characteristics
 - 2.4 General constraints
 - 2.5 Assumptions and dependencies
- 3. Specific requirements cover functional, non-functional and interface requirements.
- 4. Appendices
- 5. Index

OR

Guidelines for Software Requirement Specification

CONTENTS

1. INTRODUCTION

- 1.1 Purpose
- 1.2 Scope
- 1.3 Definitions, Acronyms, And Abbreviations
- 1.4 References
- 1.5 Overview

2. OVERALL DESCRIPTION

- 2.1 Product Perspective

2.1.1 System Interfaces

2.1.2 User Interfaces

2.1.3 Hardware Interfaces

2.1.4 Software Interfaces

2.1.5 Communications Interfaces

2.1.6 Memory Constraints

2.1.7 Operations

2.1.8 Site Adaptation Requirements

2.2 Product Functions

2.3 User Characteristics

2.4 Constraints

2.5 Assumptions and Dependencies

3. SPECIFIC REQUIREMENTS

3.1 External Interface Requirements

3.1.1 User Interfaces

3.1.2 Hardware Interfaces

3.1.3 Software Interfaces

3.1.4 Communications Interfaces

3.2 Software Product Features

3.3 Performance Requirements

3.4 Logical Database Requirements

3.5 Design Constraints

3.6 Software System Attributes

3.6.1 Reliability

3.6.2 Availability

3.6.3 Security

3.6.4 Maintainability

3.6.5 Portability

3.7 Other Requirements

Formal Systems Specification--

A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realizable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc. The mathematical basis of a formal method is provided by the specification language.

Formal Specification Language-- A formal specification language consists of two sets syn and sem, and a relation sat between them. The set syn is called the syntactic domain, the set sem is called the semantic domain, and the relation sat is called the satisfaction relation. For a given specification syn, and model of the system sem, if $\text{sat}(\text{syn}, \text{sem})$ as shown in fig., then syn is said to be the specification of sem, and sem is said to be the specificand of syn.

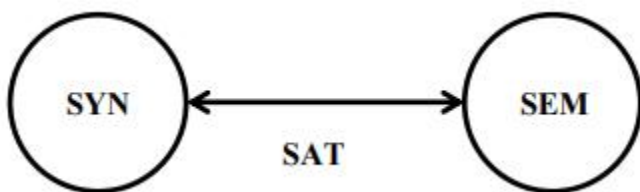


Fig. $\text{sat}(\text{syn}, \text{sem})$

Syntactic Domains-- The syntactic domain of a formal specification language consists of an alphabet of symbols and set of formation rules to construct well-formed formulae from the alphabet. The well-formed formulae are used to specify a system.

Semantic Domains-- Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values. Concurrent and distributed system specification languages are used to specify state sequences, event sequences, state transition sequences, synchronization trees, partial orders, state machines, etc.

Satisfaction Relation--- Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as semantic abstraction function. The semantic abstraction function maps the elements of the semantic domain into equivalent classes.

There can be different specifications describing different aspects of a system model, possibly using different specification languages. Two broad classes of semantic abstraction functions are defined: those that preserve a system's behaviour and those that preserve a system's structure.

Model-Oriented Vs. Property-Oriented Approach-- Formal methods are usually classified into two broad categories – model – oriented and property – oriented approaches. In a model-oriented style, one defines a system's behaviour directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc. In the property-oriented style, the system's behaviour is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy.

Example Let us consider a simple producer/consumer example. In a property-oriented style, we would probably start by listing the properties of the system like: the consumer can start consuming only after the producer has produced an item, the producer starts to produce an item only after the consumer has consumed the last item, etc. Examples of property-oriented specification styles are axiomatic specification and algebraic specification. In a model-oriented approach, we start by defining the basic operations, p (produce) and c (consume). Then we can state that $S1 + p \rightarrow S$, $S + c \rightarrow S1$. Thus the model-oriented approaches essentially specify a program by writing another, presumably simpler program. Examples of popular model-oriented specification techniques are Z, CSP, CCS, etc.

In the property-oriented style, the system's behaviour is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy.

Model-oriented approaches are more suited to use in later phases of life cycle because here even minor changes to a specification may lead to drastic changes to the entire specification.

Operational Semantics-- Informally, the operational semantics of a formal method is the way computations are represented. Some commonly used operational semantics are as follows:

Linear Semantics--- In this approach, a run of a system is described by a sequence (possibly infinite) of events or states. The concurrent activities of the system are represented by non-deterministic inter-leavings of the atomic actions. For example, a concurrent activity $a \parallel b$ is represented by the set of sequential activities $a;b$ and $b;a$. This is a simple but rather unnatural representation of concurrency. The behaviour of a system in this model consists of the set of all its runs.

Branching Semantics--- In this approach, the behaviour of a system is represented by a directed graph as shown in the fig.. The nodes of the graph represent the possible states in the evolution of a system. The descendants of each node of the graph represent the states which can be generated by any of the atomic actions enabled at that state.

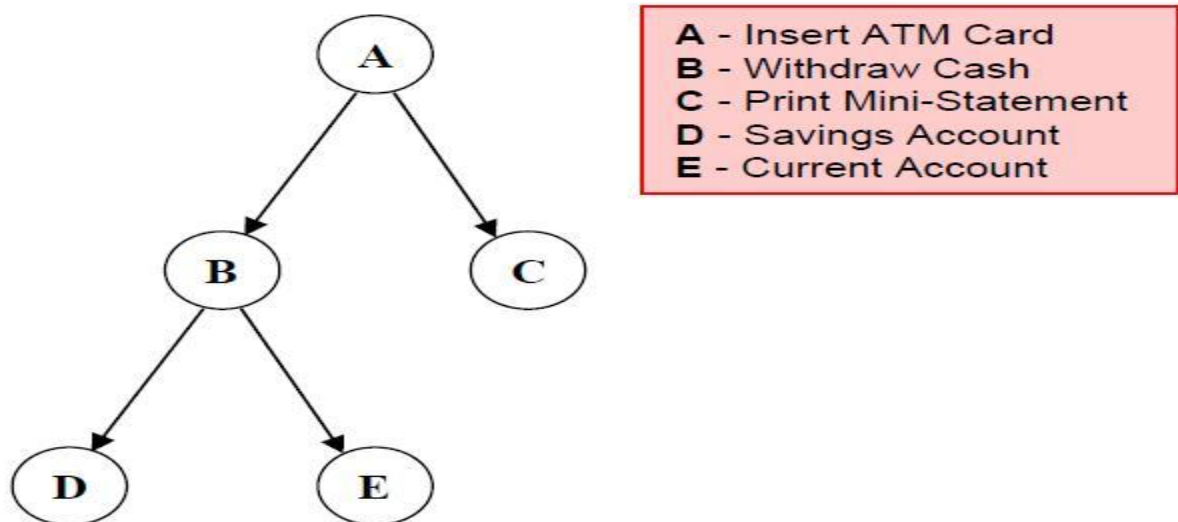


Fig. Branching semantics

Maximally Parallel Semantics--- In this approach, all the concurrent actions enabled at any state are assumed to be taken together. This is again not a natural model of concurrency since it implicitly assumes the availability of all the required computational resources.

Partial Order Semantics--- Under this view, the semantics ascribed to a system is a structure of states satisfying a partial order relation among the states (events). The partial order represents a precedence ordering among events, and constrains some events to occur only after some other events have occurred; while the occurrence of other events (called concurrent events) is considered to be incomparable. For example, Fig. shows that we can compare node B with node D, but we can't compare node D with node A. A B C D E F

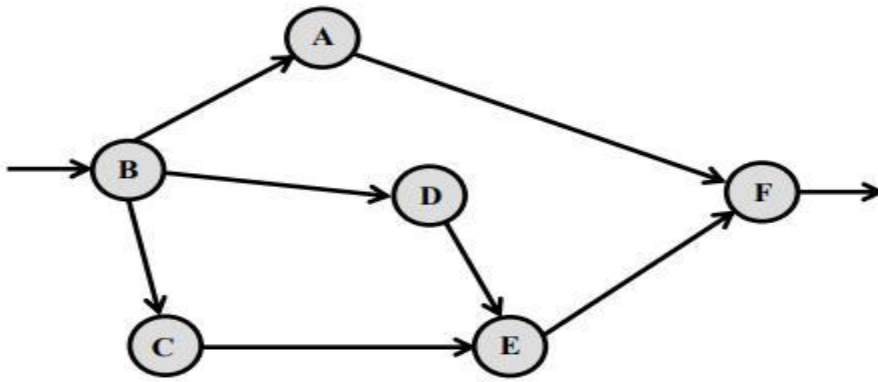


Fig. Partial order semantics

Merits of Formal Requirements Specification ----

Formal methods possess several positive features, some of which are discussed below.

- Formal specifications encourage rigour. Often, the very process of construction of a rigorous specification is more important than the formal specification itself. The construction of a rigorous specification clarifies several aspects of system behaviour that are not obvious in an informal specification.
- Formal methods usually have a well-founded mathematical basis. Thus, formal specifications are not only more precise, but also mathematically sound and can be used to reason about the properties of a specification and to rigorously prove that an implementation satisfies its specifications.
- Formal methods have well-defined semantics. Therefore, ambiguity in specifications is automatically avoided when one formally specifies a system.
- The mathematical basis of the formal methods facilitates automating the analysis of specifications. For example, a tableau-based technique has been used to automatically check the consistency of specifications. Also, automatic theorem proving techniques can be used to verify that an implementation satisfies its specifications. The possibility of automatic verification is one of the most important advantages of formal methods.
- Formal specifications can be executed to obtain immediate feedback on the features of the specified system. This concept of executable specifications is related to rapid prototyping. Informally, a prototype is a “toy” working model of a system that can provide immediate feedback on the behaviour of the specified system, and is especially useful in checking the completeness of specifications.

Limitations of Formal Requirements Specification ----

It is clear that formal methods provide mathematically sound frameworks using which systems can be specified, developed and verified in a systematic manner. However, formal methods suffer from several shortcomings, some of which are the following:

- Formal methods are difficult to learn and use.
- The basic incompleteness results of first-order logic suggest that it is impossible to check absolute correctness of systems using theorem proving techniques.
- Formal techniques are not able to handle complex problems. This shortcoming results from the fact that, even moderately complicated problems blow up the complexity of formal specification and their analysis. Also, a large unstructured set of mathematical formulae is difficult to comprehend.