SOFTWARE ENGINEERING

UNIT-II

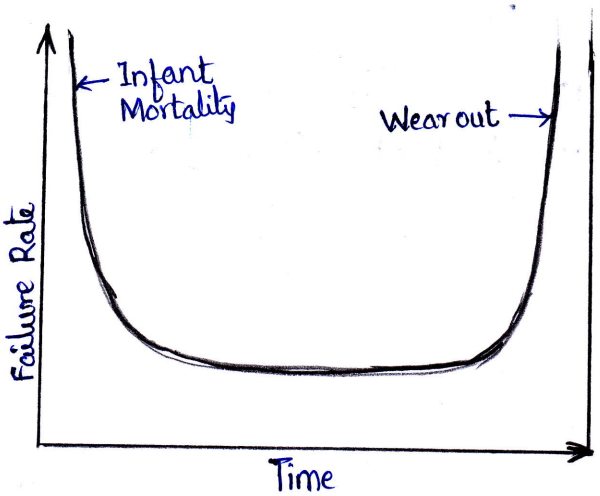
Requirements Analysis and Specification: The nature of S/W, The unique nature of Webapps, S/W Myths, Requirement gathering and analysis, S/W requirements Specification, Traceability, Characteristics of a Good SRS Document, IEEE 830 guidelines, Representing Complex Requirements using decision tables and decision trees, Overview of formal system development techniques, axiomatic specification, algebraic specification.

THE NATURE OF SOFTWARE

- Software is intangible
  - Does not exist in physical form
  - Hard to see the effort involved in its development process.
- Software is easy to reproduce
  - only costs in its development
  - Generating another copy of same S/W is easy once it is developed.
  - Building a similar S/W require less time.
- Software does not wear out
  - Deteriorates with maintenance as changes are introduced.
  - Slows down in performance as time passes.
  - Suffers when not updated regularly with respect to the surrounding environment.

Software Characteristics:

1. S/W is developed or engineered, it is not manufactured in the classical sense.
2. S/W doesn't wear out.
   The bath tub curve indicates that h/w exhibits relatively high failure rates early in its life; defects are corrected and the failure rate drops. S/W doesn't wear out but its does deteriorate.



3. Although the industry is moving toward component-based assembly, most s/w continues to be custom built.

THE UNIQUE NATURE OF WEBAPPS

In the early days of the World Wide Web (1990 to 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics.

Today, webapps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications due to the development of HTML (Hyper Text Markup Language), Java, XML (Extensible Markup Language), etc.

Attributes of WebApps:

1. Network intensiveness
2. Concurrency
3. Unpredictable load
4. Performance
5. Availability
6. Data driven
7. Content Sensitive
8. Continuous Evolution
9. Immediacy
10. Security
11. Aesthetic (Visual appearance)

SOFTWARE MYTHS

Software Myths propagated misinformation and confusion. S/w myths has a number of attributes that made them insidious; for instance, they appeared to be reasonable statements of fact, they had an intuitive feeling. Today most knowledgable professionals recognize myths for what they are - misleading attitudes that have caused serious problems for managers and technical people alike. There are two types of myths : Management and customer, and Practitioner's Myth.

1. Management Myths :- Managers with s/w responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules for slipping and improve quality. A s/w manager often grasps at belief in a s/w myth, if that belief will lessen the pressure.

Myth 1 : The standards and procedures for building s/w, won't that provide my people with everything they need to know?

Reality : It is streamlined to improve time to delivery while still maintaining a focus on quality.

Myth 2 : People have state-of-the-art s/w development tools, after all, we buy them the newest computers.

Reality : Computer Aided S/w Engg. (CASE) tools are more important than h/w for achieving good quality and productivity.

Myth 3 : If we get behind schedule, we can add more programmers and catch up.

Reality : As the new people are added, people who were working must spend time educating the new comers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well co-ordinated manner.

Myth 4 : If I decide to outsource the s/w project to the third party, I can just relax and let that firm build it.

Reality : If an organization does not understand how to manage and control s/w projects internally, it will invariably struggle when it outsources s/w projects.

2. Customer Myths : A customer who requests computer s/w may be a person at the next desk, a technical group down the hall, the marketing / sales department or an outside company, that has requested s/w under contract. In many cases, the customer believes myths about s/w because s/w managers and practitioners do little to correct misinformation. Myths leads to false expectations and ultimately dissatisfaction with the developer.

Myth 5 : A general statement of objectives is sufficient to begin writing programs - We can fill in the details later.

Reality : It is true that s/w requirements change, but the impact of change varies with the time at which it is introduced. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential.

Myth 6 : Project requirements continually change, but change can be easily accommodated because s/w is flexible.

Reality : It is true that s/w project requirements continually change. If serious attention is given to up-front definition, early requests for change can be accommodated easily.

3. Practitioner's Myth : Myths that are still believed by s/w practitioners have been fostered by 50 years of programming culture. During the early days of s/w programming was viewed as an art form. Old ways and attitudes die hard.

Myth 7 : Once we write the program and get it to work, our job is done.

Reality : Between 60 to 80 percent of all effort expended on s/w will be expended after it is delivered to the customer for the first time.

Myth 8 : Until I get the program "running" I have no way of assessing its quality.

Reality : S/w reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of s/w defects.

Myth 9 : The only deliverable work product for a successful project is the working program.

Reality : A working program is only part of a s/w configuration that includes many elements.

Myth 10 : S/w Engg. will make us create voluminous and unnecessary documentation and invariably slow us down.

Reality : S/w Engg. is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced work results in faster delivery times.

REQUIREMENTS GATHERING & ANALYSIS

The analyst starts the requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system. Then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to removing all ambiguities and inconsistencies from the initial customer perception of the problem.

Requirements gathering : This activity typically involves interviewing the end-users and customers and studying the existing documents to collect all possible information regarding the system. However, in the absence of a working system, much more imagination and creativity on the part of the system analyst is required.

Analysis of gathered requirements :- The main purpose of this activity is to clearly understand the exact requirements of the customer. 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 a problem ?
- Why is it important to solve the problem?
- What are the possible solutions ?
- What exact input, output data required ?
- What are the likely complexities?
- What exactly would the data interchange formats ?

The problems of ~~anomalies~~ anomalies :

a. Anomaly :- An ambiguity in the requirement.

b. Inconsistency :- might arise in the process control application.

c. Incompleteness :- Some of the requirements have overlooked.

SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

The SRS document usually contains all the user requirements in an informal form. Writing the SRS document is toughest. The reason behind is SRS document is expected to cater to the needs of a wide variety of audience. Some of the important categories of users of the SRS document and their needs are as follows :

a. Users, customers, and marketing personnel :- The goal of this set of audience is to ensure that the system will meet their needs.

b. S/w developers :- The s/w developers refers to the SRS document to make sure that they develop exactly what is required by the customer.

c. Test Engineers :- Their goal is to ensure that the requirements understandable from a functionality point of view, so that they

can test the s/w and and validate its working.

d. User documentation writers :- Their goal is in reading the SRS document is to ensure that they understand the document well enough to be able to write the users' manuals.

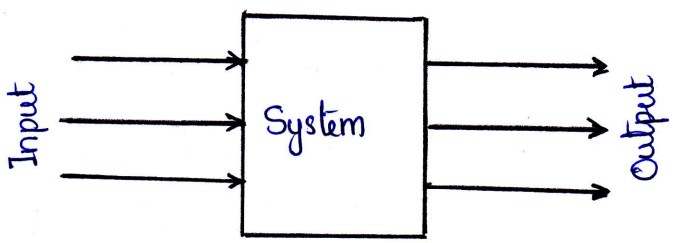
e. Project managers :- They want to ensure that they can estimate the cost easily by referring to the SRS document and that it contains all the information required to plan the project well.

f. Maintenance engineers :- The SRS document helps the maintenance engineers to understand the functionality of the system. The requirements knowledge would enable them to determine modifications.

Contents of the SRS Document :-

An SRS document should clearly document the following aspects of a system :

- Functional requirements
- Nonfunctional requirements
- Goals of implementation.



The functional requirements part should discuss the functionalities required from the system. Each function of the system (f_1) 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 functional requirements of the system as documented in the SRS document should clearly describe each function which the system would support along with the corresponding input and output data set.

TRACEABILITY

Traceability means that it would be possible to tell which design component corresponds to which requirement, which code corresponds to which design component, and which test case corresponds to which requirement, etc. Traceability analysis is an important concept and is frequently used during s/w development. A basic requirement to achieve traceability is that each functional requirement should be numbered uniquely and consistently. Proper numbering of the requirements makes it possible to uniquely refer to specific requirements in different documents

CHARACTERISTICS OF A GOOD SRS DOCUMENT

The characteristics and the skill of writing a good SRS document usually comes from the experience gained from writing similar documents for many problems. However, the analyst should be aware of the desirable qualities that every good SRS document should possess. Some of the identified desirable qualities of the SRS documents 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: The SRS document should be well-structured. A well structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope 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 :- The SRS document should specify the

external behavior of the system and not discuss the implementation issues. It should view the system to be developed as a 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 :- The SRS document should characterize acceptable responses to undesired events. These are called system responses to exceptional conditions.

Response to undesired events :- The SRS document should exhibit integrity so that the reader can easily understand the contents.

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. Requirements such as 'the system should be user friendly' are not verifiable. On the other hand, "When the name of the book is entered, the s/w should display the location of the book, if the book is available", is verifiable. Any feature of the required system that is not verifiable should be listed separately in the 'goals of the implementation' section of the SRS document.

Examples of Bad SRS Documents :

SRS document written by practitioners in industry and by students as classroom exercises, frequently suffer from a variety of problems. Some of the important categories of problems that many SRS documents suffer from are as follows:

- Over specification
- Forward references
- Wishful thinking

IEEE 830 GUIDELINES

- Help s/w customers to accurately describe what they wish to obtain.
- Help s/w suppliers to understand exactly what the customer wants.
- Help participants to -
  - Develop a template (format and content) for the s/w requirements specification (SRS) in their own organizations.
  - Develop additional documents such as SRS quality checklists or an SRS writer's handbook.
  - Establish the basis for agreement between the customers and the suppliers on what the s/w product is to do.
  - Reduce the development effort.
  - Forced to consider requirements early
  - Reduces later redesign, recoding and retesting
  - Provide a basis for realistic estimates of costs and schedules
  - Provide a basis for validation and verification
  - Facilitate transfer of the s/w product to new users or new machines.
  - Serve as a basis for enhancement requests.

REPRESENTING COMPLEX REQUIREMENTS USING DECISION TABLES & DECISION TREES

A simple text description, in cases where complex conditions would have to be checked and large number of alternatives might exist, would be difficult to comprehend and analyze. In such a situation, a decision tree or a decision table can be used to represent the logic and processing involved.

There are two main techniques available to analyze and represent complex processing logic : decision trees and decision tables. Once the decision-making logic is captured in the form of trees or tables, the test cases to validate the logic can be automatically obtained. For Instance, decision trees and decision tables find applications in information theory and switching theory.

Decision Tree :-

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. Decision table specify which variables are to be tested, and based on this what actions is to be taken depending upon the outcome of the decision-making logic, and the order in which decision making is performed.

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. Instead of discussing let us understand through below example:

Example :- A Library Membership Software (LMS) should support the following three options : new member, renewal and cancel membership. When the "new member" option is selected.

the s/w should ask for the member's name, address and phone number. If proper information is entered, the s/w should create a membership record for the new member and print a bill for the annual membership charge and the security deposit payable. If the "renewal" option is chosen, the LMS should ask for the member's name and the membership number. If the members details entered are valid, then the membership expiry date in the membership record should be updated and the annual membership details entered are invalid, an error message should be displayed. If the "cancel membership" option is selected and the name of the valid member is entered, then the membership is cancelled, a cheque for the balance amount due to the member is printed and his membership record is deleted.

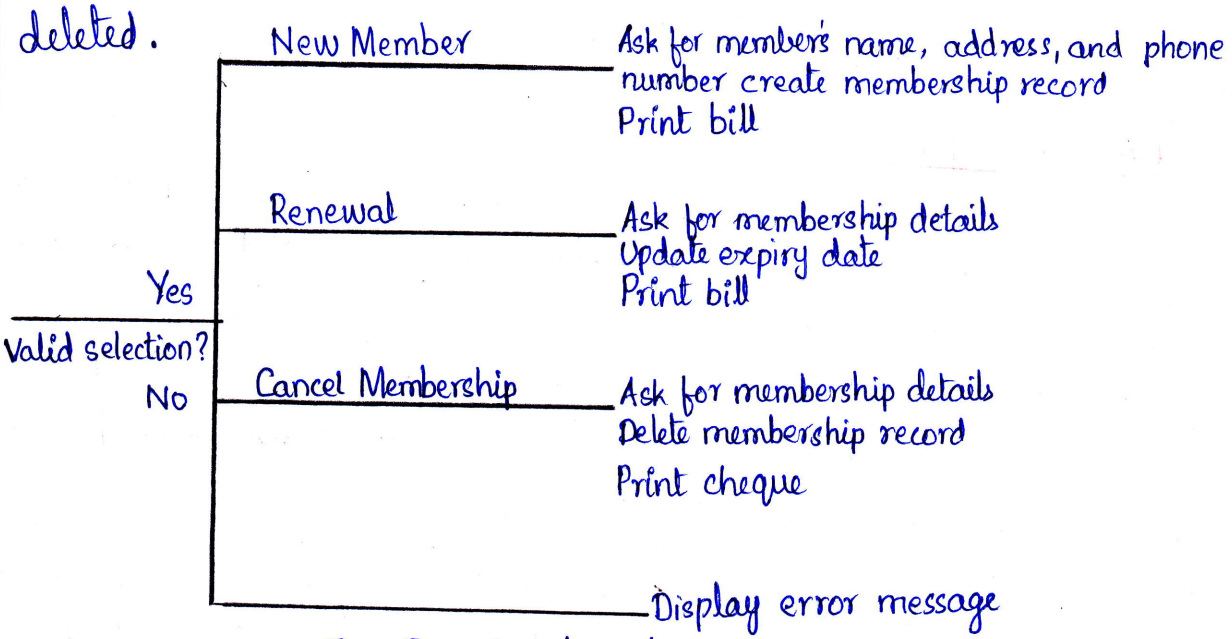


Fig: Decision tree for LMS

Decision Table

A decision table shows the decision making logic and the corresponding actions taken in a tabular or matrix form. The upper rows of the table specify the variables or conditions to be evaluated and the lower rows specify the actions to be taken when an

evaluation test is satisfied. A column in the table is called a rule. A rule implies that if a condition is true, then the corresponding action is to be executed. The decision table for the LMS problem is as follows:

CONDITIONS				
Valid selection	NO	YES	YES	YES
New members	-	YES	NO	NO
Renewal	-	No	YES	NO
Cancellation	-	No	NO	YES
Actions				
Display error message	X			
Ask for member's name etc.	X			
Build customer record		X		
Generate bill		X	X	
Ask for membership details			X	X
Update expiry date			X	
Print cheque				X
Delete record				X

Even though both decision tables and decision trees can be used to represent complex program logic, decision trees are easier to read and understand when the number of conditions are small. On the other hand, a decision table causes the analyst to look at every possible combination of conditions which he might otherwise omit. A decision tree is more useful in a situation where multilevel decision making is required. Decision trees can represent multilevel decision making hierarchically, whereas decision tables can only represent a single decision to select the appropriate action for execution.

Overview of Formal System Development Techniques

In recent years, formal techniques have emerged as a central issue in s/w engineering. This is not accidental; the importance of precise specification, modeling and verification is recognized in most engineering disciplines. Formal methods provide us with tools to precisely describe a system and show that a system is correctly implemented. The specification of a system can be given either as a list of its desirable properties or as an abstract model of the system. These two approaches are discussed, will first highlight some important concepts in formal methods, and then examine the merits and demerits of using formal techniques.

1. What is a Formal Technique :- A formal technique is a mathematical method used to: specify a h/w and/or a s/w system. The mathematical basis of formal method is provided by its specification language. More precisely, a formal specification language consists of two sets of syn and sem, and a relation sat between them. The set syn is 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 sat, then syn is said to be the specification of sem, and sem is said to be the specificand of syn.

The different stages in the system development activity are requirements specification, functional design, architectural design, detailed design, coding, implementation, etc. In general, formal techniques can be used at every stage of the system development activity.

Semantic Domains: Formal techniques can have 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.

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 formulas from the alphabet. The well-formed formulas are used to specify a system.

Satisfaction relation :- It is determined by using a homomorphism known as semantic abstraction function. There can be different specifications describing different aspects of a system model, possibly using different specification languages.

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

In a model-oriented approach, we would 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.

3. Operational Semantics : The operational semantics of a formal method constitute the ways computations are represented. There are different types of operational semantics according what is means by a single run of the system and how the runs are grouped together to describe the behavior of the system. Commonly used semantics are :

- Linear Semantics :- a sequence of events or states.
- Branching Semantics :- represented by a directed graph. The nodes of the graph represent the possible states in the evolution of a system.
- Maximally parallel Semantics :- all concurrent actions enabled at any

state are assumed to be taken together.

- Partial order semantics :- constitute a structure of states satisfying a partial order relation among the states or events.

Merits and Limitations of Formal Methods :-

- Formal specifications encourage rigour.
- Formal methods usually have a well founded mathematical basis.
- Formal methods have well-defined semantics.
- The mathematical basis of the formal methods facilitates automating the analysis of specifications.
- Formal specifications can be executed to obtain immediate feedback on the features of the specified system.

AXIOMATIC SPECIFICATION

In axiomatic specification of a system, the first-order logic is used to write the pre and post-conditions in order to specify the operations of the system in the form of axioms. The pre-conditions basically capture the conditions that must be satisfied before an operation can be successfully invoked. In essense, the pre conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when a function complete execution for the function to be considered to have executed successfully. Thus, the post-conditions are essentially the constraints on the result produced for the function execution to be considered successful.

How to develop an axiomatic specification ?

The following are the sequence of steps that can be followed to systematically develop the axiomatic specifications of a function.

- Establish the range of input values over which the function

should behave correctly. Establish the constraints on the input parameters as a predicate.

- Specify a predicate defining the condition which must hold on the output of the function if it behaved properly.
- Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type of assertion is not necessary for pure functions.
- Continue all of the above into pre and post-conditions of the function.

Example 1: Specify the pre and post-conditions of a function that takes a real number as argument and returns half the input value if the input is less than or equal to 100, or else returns double the value.

$$f(x : real) : real$$
$$pre : x \in R$$
$$post : \{(x \leq 100) \wedge (f(x) = x/2)\} \vee \{(x > 100) \wedge (f(x) = 2 * x)\}$$

Example 2: Axiomatically specify a function named search which takes an integer array and an integer key value as its arguments and returns the index in the array where the key value is present.

$$Search(X : IntArray, key : Integer) : Integer$$
$$pre : \exists i \in [X.first \ldots X.last], X[i] = key$$
$$post : \{(X'[Search(X, key)] = key) \wedge (X = X')\}$$

ALGEBRAIC SPECIFICATION

In the algebraic specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type. It was first brought into prominence by Guttag, in the specification of abstract data type. Algebraic specifications define a system as a heterogeneous algebra. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations $(1, +, -, *, /)$. In contrast

alphabetic strings together with operations of concatenation and length (A, 1, Con, len), do not form a homogeneous algebra, since the range of the length operation is the set of integers.

Each set of symbols in the algebra, in turn, is called a sort of the algebra. Using algebraic specification, we define the meaning of a set of interface procedures by using equations. An algebraic specification is usually presented in four sections.

1. Types section :- In this section, the sorts (or the data types) being used are specified.
2. Exception Section :- This section gives the names of the exceptional conditions that might occur when different operations are carried out.
3. Syntax section :- This section defines the signatures of the interface procedures. The collection of sets that form the input domain of an operator and the sort where the output is produced are called the signature of the operator. For example, PUSH takes a stack and an element and returns a new stack.
4. Equation Section :- This section gives a set of rewrite rules (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to contain conditional expressions.

The first step in defining an algebraic specification is to identify the set of required operations. After identified the required operators, it is helpful to classify them as either basic constructors, extra constructors, basic inspectors, or extra inspectors. The definition of these categories of operators are as follows:

1. Basic construction operators :- These operators are used to create or modify entities of a type. The basic construction operators are essential to generate all possible elements of the type being specified. For example, 'create' and 'append' are basic construction operators.

2. Extra Construction Operators :- These are the construction operators other than the basic construction operators. For example, the operator 'remove' is an extra construction operator, because even without using 'remove' it is possible to generate all values of the type being specified.

3. Basic inspection operators :- These operators evaluate attributes of a type without modifying them. e.g. eval, get etc.. Let S be the set of operators whose range is not the data type being specified. The set of the basic operators S1 is a subset of S, such that each operator from S-S1 can be expressed in terms of the operators from S1.

4. Extra Inspection operators :- These are the inspection operators that are not basic inspectors.

A simple way to determine whether an operator is a constructor or an inspector is to check the syntax expression for the operator. A good rule of thumb while writing an algebraic specification, is to first establish which are the constructor and inspection operators. Then write an axiom for the composition of each basic construction operator over each basic inspection operator and extra constructor operator.

Thus, if there are m1 basic constructors; m2 extra constructors, n1 basic ~~instructors~~ inspectors, and n2 extra inspectors, we should have $m1 \times (m2 + n1) + n2$ axioms. However, it should be clearly noted that these $m1 \times (m2 + n1) + n2$ axioms are the minimum required and many more axioms may be needed to take the specification complete.

While developing the rewrite rules, different persons can come up with different sets of equations. However, while developing the equations one has to be careful to see that the equations are able

to handle all meaningful composition of operators, and they should have the unique termination and finite termination properties.

Example :- Let us specify a data type point supporting the operations create, xcoord, ycoord, isequal, where the operations have their usual meaning.

Types :
defines point
uses boolean, integer

Syntax :-
1. create : integer x integer → point
2. xcoord : point → integer
3. ycoord : point → integer
4. isequal : point x point → boolean

Equations :
1. xcoord (create (x,y)) = x
2. ycoord (create (x,y)) = y
3. isequal (create (x1, y1), create (x2, y2)) = ((x1 = x2) and (y1 = y2))

In this example, we have only one basic constructor (create), and three basic inspectors (xcoord, ycoord, isequal). Therefore, we have only three equations.

The rewrite rules let you determine the meaning of any sequence of calls on the point type. Consider the following expression:

isequal (create (xcoord (create (2,3)), 5), create (ycoord (create (2,3)), 5),

By applying the rewrite rule 1 you can specify the given expression as

isequal (create (2,5), create (ycoord (create (2,3)), 5)).

By using the rewrite rule 2, you can further simplify this as

isequal (create (2,5), create (3,5)).

This is false by rewrite rule 3.

Properties of algebraic specifications

Three important properties that every algebraic specification should possess are :

- Completeness. This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. There is no simple procedure to ensure that an algebraic specification is complete.
- Finite termination property. This property essentially addresses the following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable. But, if the right-hand side of each rewrite rule has fewer ~~rule has~~ terms than the left, then the rewrite process must terminate.
- Unique termination property : This property indicates whether application of rewrite rules in different orders always results in the same answer. Essentially, to determine this property, the answer to the following question needs to be checked : can all possible sequence of choices in application of the rewrite rules to an arbitrary expression involving the interface procedures always give the same answer? Checking the unique terminations property is a very difficult problem.

Example :- Let us specify a FIFO queue supporting the operations create, append, remove, first and isempty, where the operations have their usual meaning.

Types :
    defines queue
    uses boolean, element
Exception :
    underflow, novalue
Syntax :
1. create : $\phi \rightarrow$ queue

Properties of algebraic specifications

Three important properties that every algebraic specification should possess are :

- Completeness. This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. There is no simple procedure to ensure that an algebraic specification is complete.
- Finite termination property. This property essentially addresses the following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable. But, if the right-hand side of each rewrite rule has fewer ~~rule has~~ terms than the left, then the rewrite process must terminate.
- Unique termination property : This property indicates whether application of rewrite rules in different orders always results in the same answer. Essentially, to determine this property, the answer to the following question needs to be checked : can all possible sequence of choices in application of the rewrite rules to an arbitrary expression involving the interface procedures always give the same answer? Checking the unique terminations property is a very difficult problem.

Example :- Let us specify a FIFO queue supporting the operations create, append, remove, first and isempty, where the operations have their usual meaning.

Types :
defines queue
uses boolean, element
Exception :
underflow, novalue
Syntax :
1. create : $\phi \rightarrow$ queue

2. append : queue x element → queue
3. remove : queue → queue + (underflow)
4. first : queue → element + (novalue)
5. isempty : queue → boolean

Equations :

1. isempty (create()) = true
2. isempty (append (q, e)) = false
3. first (create()) = novalue
4. first (append (q, e)) = if isempty (q) then e else first (q)
5. remove (create()) = underflow
6. remove (append (q, e)) = if isempty (q) then create() else append (remove (q), e)

In example, we have two basic construction operators (create and append). We have one extra construction operator (remove). We have considered remove to be an extra construction operator because all values of the queue can be realized, even without having the remove operator. We have two basic inspectors (first and isempty). Therefore, we have 2 x 3 = 6 equations.

END OF UNIT-II