Module 3

Requirements Analysis and Specification

Lesson 7

Algebraic Specification

Specific Instructional Objectives

At the end of this lesson the student will be able to:

- Explain algebraic specification and its use.
- Explain how algebraic specifications are represented.
- Develop algebraic specification of simple problems.
- Identify the basic properties that a good algebraic specification should satisfy.
- State the properties of a structured specification.
- State the advantages and disadvantages of algebraic specifications.
- State the features of an executable specification language (4GL) with suitable examples.

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 [1980, 1985] in specification of abstract data types. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages.

Representation of algebraic specification

Essentially, 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; {I, +, -, *, /}. In contrast, alphabetic strings together with operations of concatenation and length {A, I, con, len}, is not 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. To define a heterogeneous algebra, we first need to specify its signature, the involved operations, and their domains and ranges. Using algebraic specification, we define the meaning of a set of interface procedure by using equations. An algebraic specification is usually presented in four sections.

Types section:-

In this section, the sorts (or the data types) being used is specified.

Exceptions section:-

This section gives the names of the exceptional conditions that might occur when different operations are carried out. These

exception conditions are used in the later sections of an algebraic specification. For example, in a queue, possible exceptions are novalue (empty queue), underflow (removal from an empty queue), etc.

Syntax section:-

This section defines the signatures of the interface procedures. The collection of sets that form input domain of an operator and the sort where the output is produced are called the signature of the operator. For example, the append operation takes a queue and an element and returns a new queue. This is represented as:

$$append : queue \times element \rightarrow queue$$

Equations 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. For example, a rewrite rule to identify an empty queue may be written as:

$$isempty(create()) = true$$

By convention each equation is implicitly universally quantified over all possible values of the variables. Names not mentioned in the syntax section such as ‘r’ or ‘e’ are variables. The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them as either basic constructor operators, extra constructor operators, basic inspector operators, or extra inspection operators. The definition of these categories of operators is 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 element of the type being specified. For example, ‘create’ and ‘append’ are basic construction operators for a FIFO *queue*.

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 for a FIFO queue 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. For example, 'isempty' is a basic inspection operator because it does not modify the FIFO queue type.

4. **Extra inspection operators.** These are the inspection operators that are not basic inspectors.

A good rule of thumb while writing an algebraic specification, is to first establish which are the constructor (basic and extra) and inspection operators (basic and extra). Then write down an axiom for composition of each basic construction operator over each basic inspection operator and extra constructor operator. Also, write down an axiom for each of the extra inspector in terms of any of the basic inspectors. Thus, if there are m1 basic constructors, m2 extra constructors, n1 basic inspectors, and n2 extra inspectors, we should have m1 $\times$ (m2+n1) + n2 axioms are the minimum required and many more axioms may be needed to make the specification complete. Using a complete set of rewrite rules, it is possible to simplify an arbitrary sequence of operations on the interface procedures.

Develop algebraic specification of simple problems

The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them into different catgories.

A simple way to determine whether an operator is a constructor (basic or extra) or an inspector (basic or extra) is to check the syntax expression for the operator. If the type being specified appears on the right hand side of the expression then it is a constructor, otherwise it is an inspection operator. For example, in a FIFO queue, 'create' is a constructor because the data type specified 'queue' appears on the right hand side of the expression. But, 'first' and 'isempty' are inspection operators since they do not modify the *queue* data type.

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, integer

Exceptions:
underflow, novalue

Syntax:
1. *create : $\phi \rightarrow$ queue*
2. *append : queue x element $\rightarrow$ queue*
3. *remove : queue $\rightarrow$ queue + {underflow}*
4. *first : queue $\rightarrow$ element + {novalue}*
5. *isempty : queue $\rightarrow$ boolean*

Equations:
1. *isempty(create()) = true*
2. *isempty((append(q,e)) = false*
3. *first(create()) = novalue*
4. *first(append(q,e)) = is 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 this example, there are two basic constructors (*create* and *append*), one extra construction operator (*remove*) and two basic inspectors (*first* and *empty*). Therefore, there are 2 x (1+2) + 0 = 6 equations.

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 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 result 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 number? Checking the unique termination property is a very difficult problem.

Structured specification

Developing algebraic specifications is time consuming. Therefore efforts have been made to device ways to ease the task of developing algebraic specifications. The following are some of the techniques that have successfully been used to reduce the effort in writing the specifications.

- **Incremental specification.** The idea behind incremental specification is to first develop the specifications of the simple types and then specify more complex types by using the specifications of the simple types.
- **Specification instantiation.** This involves taking an existing specification which has been developed using a generic parameter and instantiating it with some other sort.

Advantages and disadvantages of algebraic specifications

Algebraic specifications have a strong mathematical basis and can be viewed as heterogeneous algebra. Therefore, they are unambiguous and precise. Using an algebraic specification, the effect of any arbitrary sequence of operations involving the interface procedures can automatically be studied. A major shortcoming of algebraic specifications is that they cannot deal with side effects. Therefore, algebraic specifications are difficult to interchange with typical programming languages. Also, algebraic specifications are hard to understand.

Executable specification language (4GLs).

If the specification of a system is expressed formally or by using a programming language, then it becomes possible to directly execute the specification. However, executable specifications are usually slow and inefficient, 4GLs[3] (4^{th} Generation Languages) are examples of executable specification languages. 4GLs are successful because there is a lot of commonality across data processing applications. 4GLs rely on software reuse, where the common abstractions have been identified and parameterized. Careful experiments have shown that rewriting 4GL programs in higher level languages results in up to 50% lower memory usage and also the program execution time can reduce ten folds. Example of a 4GL is Structured Query Language (SQL).

The following questions have been designed to test the objectives identified for this module:

1. Identify at least four roles of a system analyst.

Ans.: - 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 removing 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.

2. Identify three important parts of an SRS document.

Ans.: - The important parts of SRS document are:

- Functional requirements of the system
- Non-functional requirements of the system, and
- Goals of implementation

The functional requirements part discusses the functionalities required from the system. Nonfunctional requirements deal with the characteristics of the system which can not be expressed as functions -

such as the maintainability of the system, portability of the system, usability of the system, etc. 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.

3. Without developing an SRS document an organization might face severe problems. Identify those problems.

Ans.: - The important problems that an organization would face if it does not develop an 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.

4. Identify the non-functional requirement-issues that are considered for any given problem description?

Ans.: - Nonfunctional requirements are the characteristics of the system which can not be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Nonfunctional requirements may include:

reliability issues,
performance issues,
human - computer interface issues,
interface with other external systems,
security and maintainability of the system, etc.

5. Mention at least five problems that an unstructured specification would create during software development.

Ans.: - Some problems that might be created by an unstructured specification are as follows:

- 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.

6. Identify the necessity of using formal specification technique in the context of requirements specification.

Ans.: - A formal specification 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. There are also some advantages of formal specification technique. Those advantages are:

- 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 behavior 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 behavior of the specified system, and is especially useful in checking the completeness of specifications.

7. Identify at least two disadvantages of formal technique.

Ans.: - It is clear that formal methods provide mathematically sound frameworks within large, complex systems can be specified, developed and verified in a systematic rather than in an ad hoc 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 formulas is difficult to comprehend.

8. Identify at least two differences between model-oriented and property-oriented approaches in the context of requirements specification.

Ans.: - 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 behavior 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 behavior is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy.

  Let us consider a simple producer/consumer example. In a property-oriented style, it is probably started 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 → S, S + c → 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.

- 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. They do not support logical conjunctions (AND) and disjunctions (OR).

Property-oriented approaches are suitable for requirements specification because they can be easily changed. They specify a system as a conjunction of axioms and you can easily replace one axiom with another one.

9. Explain the use of operational semantic.

Ans.: - Informally, the operational semantics of a formal method is the way computations are represented. There are different types of operational semantics according to what is meant by a single run of the system and how the runs are grouped together to describe the behavior of the system. 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 interleaving of the automatic actions. For example, a concurrent activity a‖b is represented by the set of sequential activities a;b and b;a. This is simple but rather unnatural representation of concurrency. The behavior of a system in this model consists of the set of all its runs. To make this model realistic, usually justice and fairness restrictions are imposed on computations to exclude the unwanted interleavings.

Branching Semantics:-

In this approach, the behavior of a system is represented by a directed graph as shown in the fig. 3.7. 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. Although this semantic model distinguishes the branching points in a computation, still it represents concurrency by interleaving.

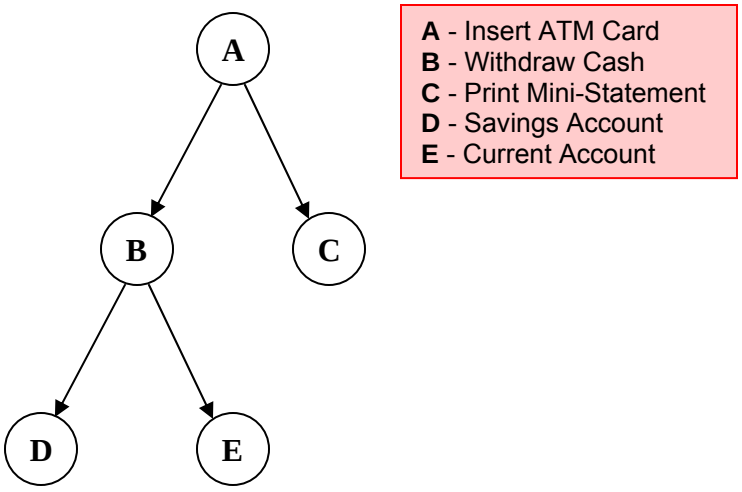

Fig. 3.7: Branching semantics

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 constraints 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. This fact identifies concurrency as a phenomenon not translatable to any interleaved representation.

For example, from figure (fig. 3.8), we can see that node Ingredient can be compared with node Brew, but neither can it be compared with node Hot/Cold nor with node Accepted.

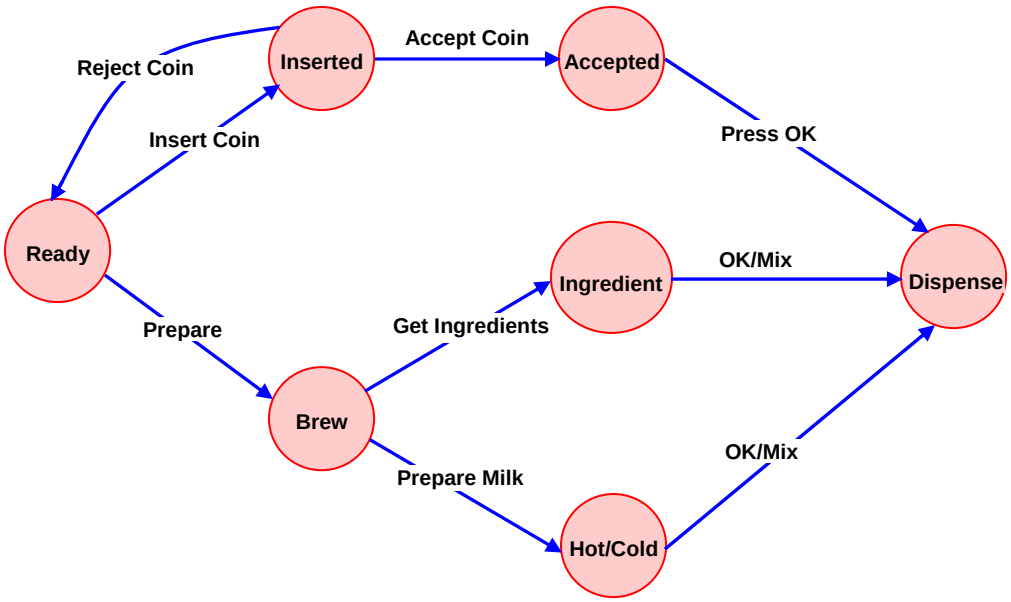

Fig. 3.8: Partial order semantics implied by a beverage selling machine

10. Identify the requirements of algebraic specifications in order to define a system.

Ans.: - In the algebraic specification technique an object class or type is specified in terms of relationships existing between the operations defined on that type. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages.

Essentially, 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; {I, +, -, *, /}. In contrast, alphabetic strings together with operations of concatenation and length {A, I, con, len}, is not a homogeneous algebra, since the range of the length operation is the set of integers. To define a heterogeneous algebra, firstly it is needed to specify its signature, the involved operations, and their domains and ranges. Using algebraic specification, it can be easily defined the meaning of a set of interface procedure by using equations. An algebraic specification is usually presented in four sections.

Types section:-
In this section, the sorts (or the data types) being used is specified.

Exceptions section:-
This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification.

Syntax section:-
This section defines the signatures of the interface procedures. The collection of sets that form 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.

$$stack \times element \rightarrow stack$$

Equations 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.

By convention each equation is implicitly universally quantified over all possible values of the variables. Names not mentioned in the syntax section such 'r' or 'e' is variables. The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them as either basic constructor operators, extra constructor operators, basic inspector operators, or extra inspection operators. The definition of these categories of operators is as follows:

- **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 element of the type being specified. For example, 'create' and 'append' are basic construction operators in a FIFO queue.

- **Extra construction operators.** These are the construction operators other than the basic construction operators. For example, the operator 'remove' is an extra construction operator in a FIFO queue because even without using 'remove', it is possible to generate all values of the type being specified.

- **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.

- **Extra inspection operators.** These are the inspection operators that are not basic inspectors.

A good rule of thumb while writing an algebraic specification, is to first establish which are the constructor (basic and extra) and inspection operators (basic and extra). Then write down an axiom for composition of each basic construction operator over each basic inspection operator and extra constructor operator. Also, write down an axiom for each of the extra inspector in terms of any of the basic inspectors. Thus, if there are m1 basic constructors, m2 extra constructors, n1 basic inspectors, and n2 extra inspectors, we should have m1 $\times$ (m2+n1) + n2 axioms are the minimum required and many more axioms may be needed to make the specification complete. Using a complete set of rewrite rules, it is possible to simplify an arbitrary sequence of operations on the interface procedures.

11. Identify the use of algebraic specifications in the context of requirements specification.

Ans.: - The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them as either basic constructor operators, extra constructor operators, basic inspector operators, or extra inspector operators. A simple way to determine whether an operator is a constructor (basic or extra) or an inspector (basic or extra) is to check the syntax expression for the operator. If the type being specified appears on the right hand side of the expression then it is a constructor, otherwise it is an inspection operator. For example, in case of the following example, create is a constructor because point appears on the right hand side of the expression and point is the data type being specified. But, xcoord is an inspection operator since it does not modify the point type.

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 × integer → point
2. xcoord : point → integer
3. ycoord : point → integer
4. isequal : point × 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, there is only one basic constructor (create), and three basic inspectors (xcoord, ycoord, and isequal). Therefore, there are only 3 equations.

12. Identify the three important properties that every good algebraic specification should possess.

Ans.: - 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 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 result 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 number? Checking the unique termination property is a very difficult problem.

13. Identify at least two properties of a structured specification.

Ans.: - Two properties of a structured specification are as follows:

- **Incremental specification.** The idea behind incremental specification is to first develop the specifications of the simple types and then specify more complex types by using the specifications of the simple types.
- **Specification instantiation.** This involves taking an existing specification which has been developed using a generic parameter and instantiating it with some other sort.

14. Identify at least two advantages of algebraic specification.

Ans.: - Algebraic specifications have a strong mathematical basis and can be viewed as heterogeneous algebra. Therefore, they are unambiguous and precise. Using an algebraic specification, the effect of any arbitrary sequence of operations involving the interface procedures can be automatically studied.

15. Identify at least two disadvantages of algebraic specification.

Ans.: - A major shortcoming of algebraic specifications is that they cannot deal with side effects. Therefore, algebraic specifications are difficult to interchange with typical programming languages. Also, algebraic specifications are hard to understand.

16. Write down at least two features of an executable specification language with examples.

Ans.: - If the specification of a system is expressed formally or by using a programming language, then it becomes possible to directly execute the specification. However, executable specifications are usually slow and inefficient, 4GLs[3] (4^{th} Generation Languages) are examples of executable specification languages. 4GLs are successful because there is a lot of commonality across data processing applications. 4GLs rely on software reuse, where the common abstractions have been identified and parameterized. Careful experiments have shown that rewriting 4GL programs in higher level languages results in up to 50% lower memory usage and also the program execution time can reduce ten folds. Example of a 4GL is Structured Query Language (SQL).

For the following, mark all options which are true.

1. An SRS document normally contains

□ Functional requirements of the system √
□ Module structure
□ Configuration management plan
□ Non-functional requirements of the system √
□ Constraints on the system √

2. The structured specification technique that is used to reduce the effort in writing specification is

□ Incremental specification
□ Specification instantiation
□ Both of the above √
□ None of the above

3. Examples of executable specifications are

□ Third generation languages
□ Fourth generation languages √
□ Second-generation languages
□ First generation languages

Mark the following as either True or False. Justify your answer.

1. Functional requirements address maintainability, portability, and usability issues.

Ans.: - False.

Explanation: - The functional requirements of the system should clearly describe each of the functions that the system needs to perform along with the corresponding input and output dataset. Non-functional requirements deal with the characteristics of the system that cannot be expressed functionally e.g. maintainability, portability, usability etc.

2. The edges of decision tree represent corresponding actions to be performed according to conditions.

Ans.: - False.

Explanation: - The edges of decision tree represent conditions and the leaf nodes represent the corresponding actions to be performed.

3. The upper rows of the decision table specify the corresponding actions to be taken when an evaluation test is satisfied.

Ans.: - False.

Explanation: - The upper rows of the table specify the variables or conditions to be evaluated and the lower rows specify the corresponding actions to be taken when an evaluation test is satisfied.

4. A column in a decision table is called an attribute.

Ans.: - False.

Explanation: - A column in a decision table is called a rule. A rule implies that if a condition is true, then execute the corresponding action.

5. Pre – conditions of axiomatic specifications state the requirements on the parameters of the function before the function can start executing.

Ans.: - True.

Explanation: - The pre-conditions basically capture the conditions that must be satisfied before an operation can successfully be invoked. In essence, the pre-conditions capture the requirements on the input parameters of a function.

6. Post – conditions of axiomatic specifications state the requirements on the parameters of the function when the function is completed.

Ans.: - True.

Explanation: - The post-conditions are the conditions that must be satisfied when a function completes execution for the function to be considered to have executed successfully. Thus, the post-conditions are essentially constraints on the results produced for the function execution to be considered successful.

7. Homogeneous algebra is a collection of different sets on which several operations are defined.

Ans.: - False.

Explanation: - A heterogeneous algebra is a collection of different sets on which several operations are defined. But homogeneous algebra consists of a single set and several operations; {I, +, -, *, /}.

8. Applications developed using 4 GLs would normally be more efficient and run faster compared to applications developed using 3 GL.

Ans.: - False.

Explanation: - Even though 4th Generation Languages (4 GLs) reduce the effort for development; it is normally inefficient as these are more general-purpose languages. If somebody rewrite 4 GL programs in higher level languages (i.e. 3GLs), it might result in upto 50% lower memory requirements and also the program can run upto 10 times faster.