Module

7

Software Engineering Issues

Lesson

34

Requirements Analysis and Specification

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Get an overview of Requirements Gathering And Analysis
- Identify the important parts and properties of a SRS document
- Identify and document the functional and non-functional requirements from a problem description
- Identify the problems that an organization would face if it does not develop SRS documents
- Identify the problems that an unstructured specification would create during software development
- Understand the basics of decision trees and decision table
- Understand formal techniques and formal specification languages
- Differentiate between model-oriented and property-oriented approaches
- Explain the operational semantics of a formal method
- Identify the merits and limitations of formal requirements specification
- Explain and develop axiomatic specification and algebraic specification
- Identify the basic properties of a good algebraic specification
- 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

1. Introduction

The requirements analysis and specification phase starts once the feasibility study phase is complete and the project is found to be technically sound and feasible. The goal of the requirements analysis and specification phase is to clearly understand customer requirements and to systematically organize these requirements in a specification document. This phase consists of the following two activities:

- Requirements Gathering And Analysis
- Requirements Specification

2. Requirements Gathering And Analysis

The 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 analyses 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.

3. SRS Document

After the analyst has collected all the requirements information regarding the software to be developed, and has removed all the incompleteness, in consistencies, and anomalies from the specification, he starts to systematically organize the requirements in the form of an SRS document.

The important parts of SRS document are:

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

3.1.1. Functional Requirements

The functional requirements part discusses the functionalities required from the system. Here we list all high-level functions $\{f_i\}$ that the system performs. Each high-level function f_i, as shown in fig. 34.1, is considered as a transformation of a set of input data to some corresponding output data. The user can get some meaningful piece of work done using a high-level function.

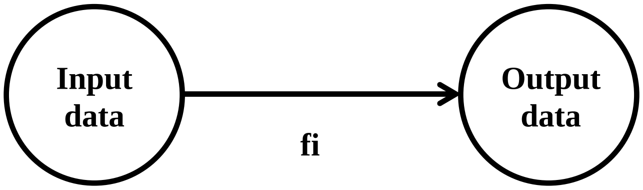

Fig. 34.1 Function f_i

3.1.2. Non-Functional Requirements

Non-functional 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.

Non-functional requirements may include:

- Reliability issues
- Accuracy of results
- Human-computer interface issues
- Constraints on the system implementation, etc.

3.1.3. 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.

3.1.4. Identify Functional Requirements

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.

Here we list all functions $\{f_i\}$ that the system performs. Each function f_i, as shown in fig. 34.1, is considered as a transformation of a set of input data to some corresponding output data.

Example

Consider the case of the library system, where –

F1: Search Book function (fig. 34.2)

Input: An author's name

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

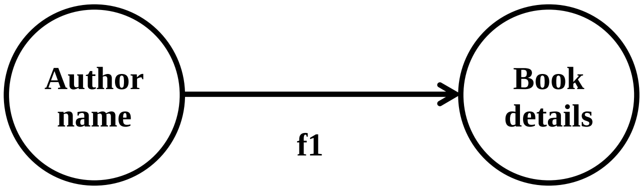

Fig. 34.2 Book Function

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.

3.1.5. Document 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.

Example: Withdraw Cash from ATM

R1: withdraw cash
Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

R1.1: select withdraw amount option
Input: "withdraw amount" option
Output: user prompted to enter the account type

R1.2: select account type
Input: user option
Output: prompt to enter amount

R1.3: get required amount
Input: amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.
Output: The requested cash and printed transaction statement.
Processing: the amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed.

3.1.6. 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 behaviour 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 behaviour 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.

3.1.7. Problems without a SRS Document

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 is required by the customer.
- Without SRS document, it will be very difficult for the maintenance engineers to understand the functionality of the system.
- It will be very difficult for user document writers to write the users' manuals properly without understanding the SRS document.

3.1.8. Identify Non-Functional Requirements

Non-functional requirements may include:

- Reliability issues
- Performance issues
- Human - computer interface issues
- Interface with other external systems
- Security and maintainability of the system, etc.

3.1.9. Problems with An Unstructured Specification

The problems that an unstructured specification would create during software development are as follows:

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

3.1.10. Techniques for Representing Complex Logic

A good SRS document should properly characterize the conditions under which different scenarios of interaction occur. Sometimes such conditions are complex and several alternative interaction and processing sequences may exist.

There are two main techniques available to analyze and represent complex processing logic: **decision trees** and **decision tables**.

1. Decision Trees

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 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.

Decision tree representation of the above example

The following tree (fig. 34.3) shows the graphical representation of the above example. After getting information from the user, the system makes a decision and then performs the corresponding actions.

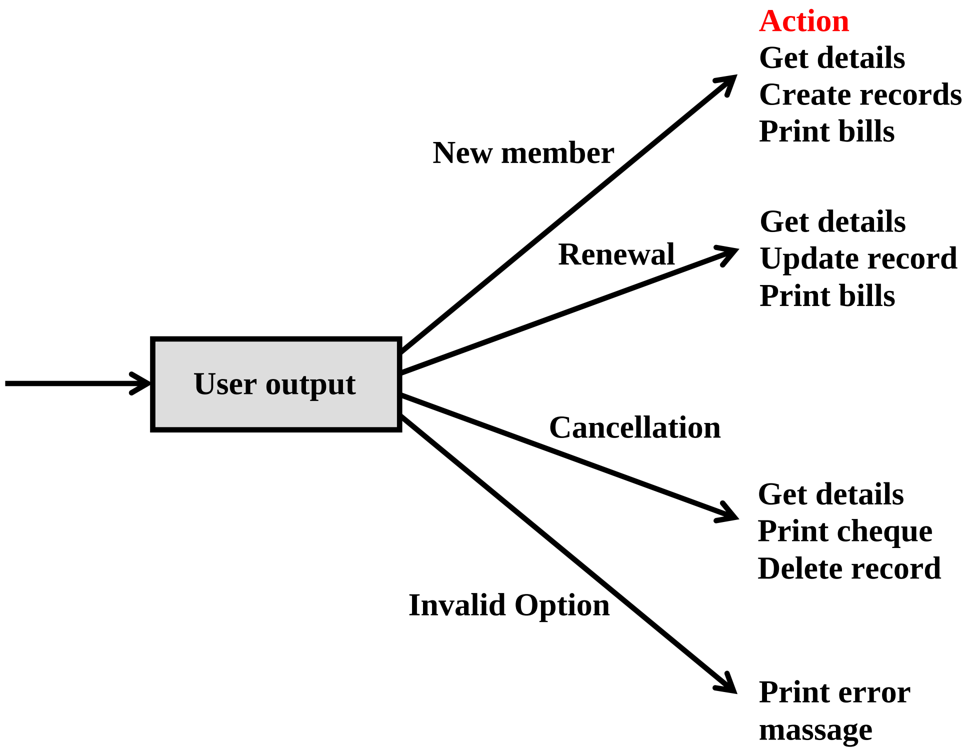

Fig. 34.3 Decision tree for LMS

2. Decision Tables

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.

Example

Consider the previously discussed LMS example. The decision table shown in fig. 34.4 shows how to represent the 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. 34.4 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' and so on.

4. Formal Requirements 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.

4.1. 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 *sat(syn, sem)* as shown in fig.34.5, then *syn* is said to be the specification of *sem*, and *sem* is said to be the specificand of *syn*.

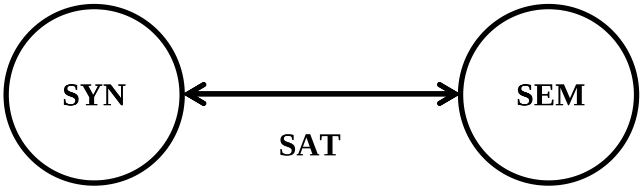

Fig. 34.5 sat (syn, sem)

4.1.1. 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.

4.1.2. 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.

4.1.3. 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. Some of these specifications describe the system’s behaviour and the others describe the system’s structure. Consequently, two broad classes of semantic abstraction functions are defined: those that preserve a system’s behaviour and those that preserve a system’s structure.

4.2. 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. 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.

4.3. Operational Semantics

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 behaviour of the system. Some commonly used operational semantics are as follows:

4.3.1. 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 automatic actions. For example, a concurrent activity a‖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. To make this model realistic, usually justice and fairness restrictions are imposed on computations to exclude the unwanted interleaving.

4.3.2. Branching Semantics

In this approach, the behaviour of a system is represented by a directed graph as shown in the fig. 34.6. 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.

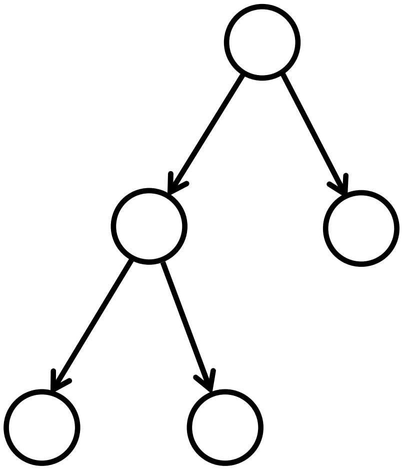

Fig. 34.6 Branching semantics

4.3.3. 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.

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

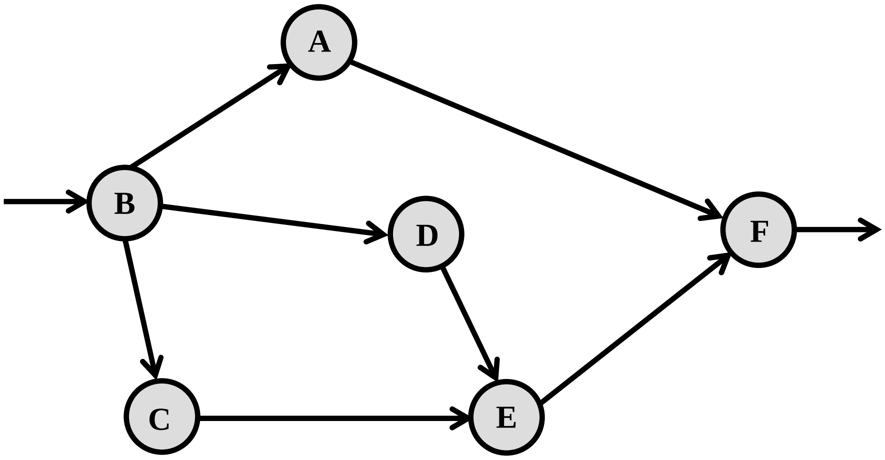

Fig. 34.7 Partial order semantics

For example, Fig. 34.7 shows that we can compare node B with node D, but we can't compare node D with node A.

4.4. 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.

4.5. 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.

5. Axiomatic Specification

In axiomatic specification of a system, first-order logic is used to write the pre and post-conditions 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 successfully be invoked. In essence, 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 completes execution and the function is considered to have been executed successfully. Thus, the post-conditions are essentially the constraints on the results produced for the function execution to be considered successful.

5.1. Steps 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. Also find out other constraints on the input parameters and write them in the form of a predicate.
- Specify a predicate defining the conditions 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.
- Combine all of the above into pre and post conditions of the function.

5.2. Examples

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 : $\exists\, i \in [Xfirst....Xlast], X[i] = key$
post : $\{(X'[search(X, key)] = key) \wedge (X = X')\}$

Here, the convention that has been followed is that, if a function changes any of its input parameters, and if that parameter is named X, then it has been referred that after the function completes execution as X′.

6. 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.

6.1. 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. 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.

1. Types section
In this section, the sorts (or the data types) being used are specified.

2. 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.

3. 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 x element → stack

4. 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.

6.2. Operators

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' 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 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.

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.

6.3. Writing Algebraic Specifications

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 × (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.

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 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,* and *isequal* where the operations have their usual meaning.

Types:

defines point
uses boolean, integer

Syntax:

create : integer × integer → point
xcoord : point → integer
ycoord : point → integer
isequal : point × point → Boolean

Equations:

xcoord(*create*(x, y)) = x
ycoord(*create*(x, y)) = y
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,* and *isequa*l). Therefore, we have only 3 equations.

6.4. Properties of Algebraic Specifications

Three important properties that every good 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.

6.5. 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.

6.6. Pros and Cons 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.

7. Executable Specification Language (4GL)

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, 4GLs3 (4th 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 by ten folds. Example of a 4GL is Structured Query Language (SQL).

8. Exercises

1. Mark the following as True or False. Justify your answer.
    a. All software engineering principles are backed by either scientific basis or theoretical proof.
    b. Functional requirements address maintainability, portability, and usability issues.
    c. The edges of decision tree represent corresponding actions to be performed according to conditions.
    d. The upper rows of the decision table specify the corresponding actions to be taken when an evaluation test is satisfied.
    e. A column in a decision table is called an attribute.
    f. Pre-conditions of axiomatic specifications state the requirements on the parameters of the function before the function can start executing.
    g. Post-conditions of axiomatic specifications state the requirements on the parameters of the function when the function is completed.

h. Homogeneous algebra is a collection of different sets on which several operations are defined.
i. Applications developed using 4 GLs would normally be more efficient and run faster compared to applications developed using 3 GL.

2. For the following, mark all options which are true.
   j. 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

   k. The structured specification technique that is used to reduce the effort in writing specification is
      - Incremental specification
      - Specification instantiation
      - Both the above
      - None of the above
   l. Examples of executable specifications are
      - Third generation languages
      - Fourth generation languages
      - Second-generation languages
      - First generation languages
3. Identify the roles of a system analyst.
4. Identify the important parts of an SRS document. Identify the problems an organization might face without developing an SRS document.
5. Identify the non-functional requirement-issues that are considered for a given problem description.
6. Discuss the problems that an unstructured specification would create during software development.
7. Identify the necessity of using formal technique in the context of requirements specification.
8. Identify the differences between model-oriented and property-oriented approaches in the context of requirements specification.
9. Explain the use of operational semantic.
10. Explain the use of algebraic specifications in the context of requirements specification. Identify the requirements of algebraic specifications to define a system.
11. Identify the essential sections of an algebraic specification to define a system. Explain the steps for developing algebraic specification of simple problems.
12. Identify the properties that every good algebraic specification should possess.
13. Identify the basic properties of a structured specification.
14. Discuss the advantages and disadvantages of algebraic specification.
15. Write down the important features of an executable specification language with examples.