Module 3

Requirements Analysis and Specification

Lesson 6

Formal Requirements Specification

Specific Instructional Objectives

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

- Explain what a formal technique is.
- Explain what a formal specification language is.
- Differentiate between model-oriented and property-oriented approaches in the context of requirements specification.
- Explain the operational semantics of a formal method.
- Identify the merits of formal requirements specification.
- Identify the limitations of formal requirements specification.
- Develop axiomatic specification of simple problems.

Formal technique

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

Formal specification language

A formal specification language consists of two sets syn and sem, and a relation sat between them. The set syn is called the syntactic domain, the set sem is called the semantic domain, and the relation sat is called the satisfaction relation. For a given specification syn, and model of the system sem, if sat (syn, sem), as shown in fig. 3.6, then syn is said to be the specification of sem, and sem is said to be the specificand of syn.

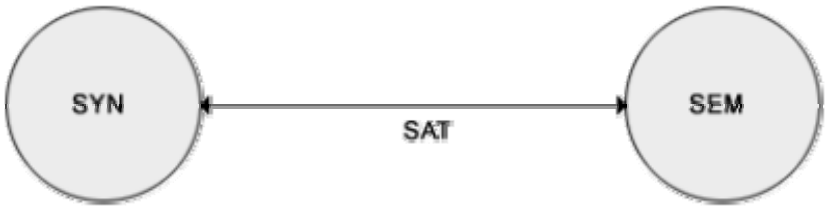

Fig. 3.6: sat (syn, sem)

Syntactic Domains

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

Semantic Domains

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

Satisfaction Relation

Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as semantic abstraction function. The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages. Some of these specifications describe the system's behavior and the others describe the system's structure. Consequently, two broad classes of semantic abstraction functions are defined: those that preserve a system's behavior and those that preserve a system's structure.

Model-oriented vs. property-oriented approaches

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.

Example:-

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. A good example of a producer-consumer problem is CPU-Printer coordination. After processing of data, CPU outputs characters to the buffer for printing. Printer, on the other hand, reads characters from the buffer and prints them. The CPU is constrained by the capacity of the buffer, whereas the printer is

constrained by an empty buffer. 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.

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 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 interleavings 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. An example involving the transactions in an ATM is shown in fig. 3.7. Although this semantic model distinguishes the branching points in a computation, still it represents concurrency by interleaving.

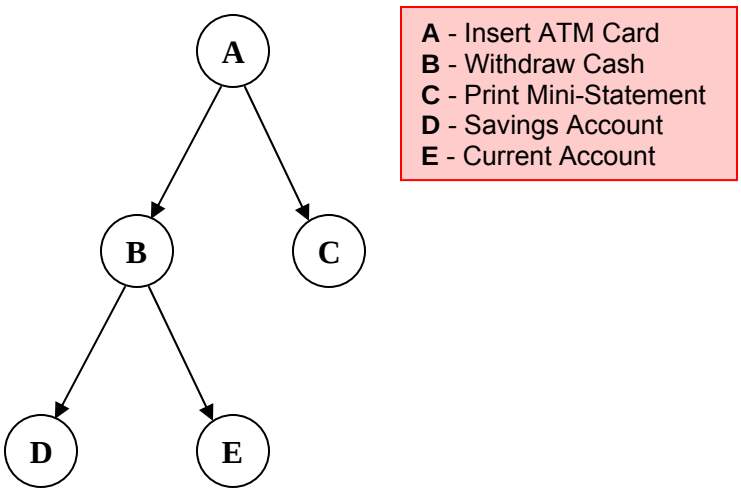

Fig. 3.7: Branching semantics

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, figure (fig. 3.8) shows the semantics implied by a simplified beverage selling machine. From the figure, we can infer that beverage is dispensed only if an inserted coin is accepted by the machine (precedence). Similarly, preparation of ingredients and milk are done simultaneously (concurrence). Hence, node Ingredient can be compared with node Brew, but neither can it be compared with node Hot/Cold nor with node Accepted.

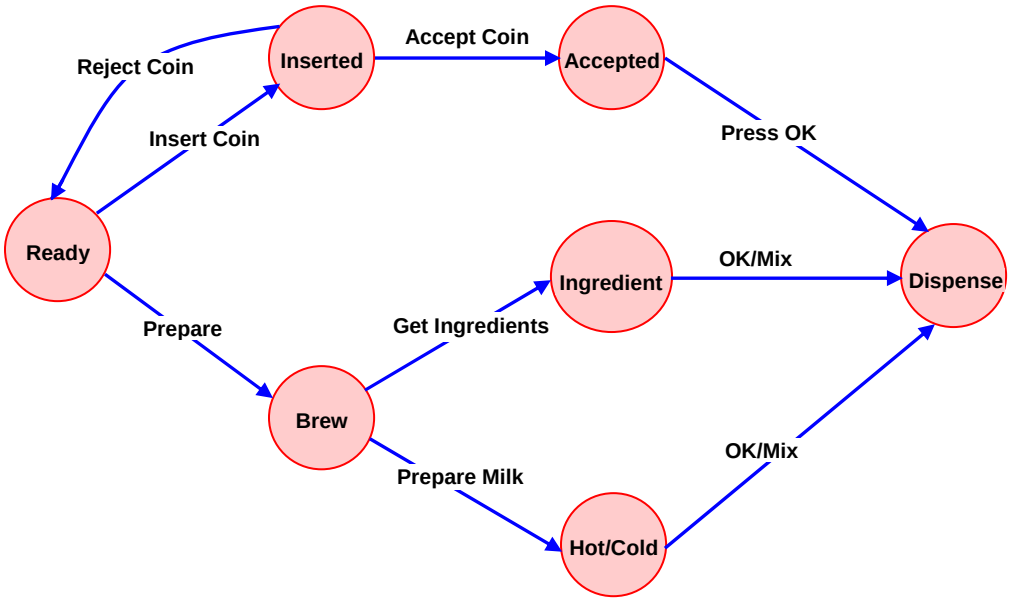

Fig. 3.8: Partial order semantics implied by a beverage selling machine

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 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.

Limitations of formal requirements specification

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.

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 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.

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 it 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.

Example1: -

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 : $\{(x \leq 100) \wedge (f(x) = x/2)\} \vee \{(x > 100) \wedge (f(x) = 2 * x)\}$

Example2: -

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 : $\exists\ i \in [Xfirst \ldots . Xlast], X[i] = key$

post : $\{(X'[search(X, key)] = key) \wedge (X = X')\}$

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