FORMAL SPECIFICATION

A formal software specification is a statement expressed in a language whose vocabulary, syntax, and semantics are formally defined. The need for a formal semantic definition means that the specification languages cannot be based on natural language; it must be based on mathematics.

The advantages of a formal language are:

- The development of a formal specification provides insights and understanding of the software requirements and the software design.
- Given a formal system specification and a complete formal programming language definition, it may be possible to prove that a program conforms to its specifications.
- Formal specification may be automatically processed. Software tools can be built to assist with their development, understanding, and debugging.

- Depending on the formal specification language being used, it may be possible to animate a formal system specification to provide a prototype system.
- Formal specifications are mathematical entities and may be studied and analyzed using mathematical methods.
- Formal specifications may be used as a guide to the tester of a component in identifying appropriate test cases.

Formal specifications sometimes are not used because:

- Software management is conservative and unwilling to adopt new techniques whose payoff is not immediately obvious.
- Most software engineers have not been trained in formal specification techniques.
- Some classes of systems are difficult to specify using existing specification techniques. Interactive and interrupt driven systems might be examples.

Relational and State-Oriented Notations

Relational notations are used based on the concept of entities and attributes. Entities are elements in a system; the names are chosen to denote the nature of the elements (e.g., stacks, queues). Attributes are specified by applying functions and relations to the named entities. Attributes specify permitted operations on entities, relationships among entities, and data flow between entities.

Relational notations include implicit equations, recurrence relations, and algebraic axioms.

State-oriented specifications use the current state of the system and the current stimuli presented to the system to show the next state of the system. The execution history by which the current state was attained does not influence the next state; it is dependent only on the current state and the current stimuli.

State-oriented notations include decision tables, event tables, transition tables, and finite-state tables.

SPECIFICATION PRINCIPLES

Principle 1: Separate functionality from implementation.

A specification is a statement of what is desired, not how it is to be realized. Specifications can take two general forms. The first form is that of mathematical functions: Given some set of inputs, produce a particular set of outputs. The general form of such specifications is find [a/the/all] result such that P(input), where P represents an arbitrary predicate. In such specifications, the result to be obtained has been entirely expressed in a “what”, rather than a “how” form, mainly because the result is a mathematical function of the input (the operation has well-defined starting and stopping points) and is unaffected by any surrounding environment.

Principle 2: A process-oriented systems specification language is sometimes required. If the environment is dynamic and its changes affect the behavior of some entity interacting with that environment (as in an embedded computer system), its behavior cannot be expressed as a mathematical function of its input. Rather a process-oriented description must be employed, in which the “what” specification is achieved by specifying a model of the desired behavior in terms of functional responses to various stimuli from the environment.

Principle 3: The specification must provide the implementor all of the information he/she needs to complete the program, and no more.
In particular, no information about the structure of the calling program should be conveyed.

Principle 4: The specification should be sufficiently formal that it can conceivably be tested for consistency, correctness, and other desirable properties.

Principle 5: The specification should discuss the program in terms normally used by the user and implementor alike.

SOME SPECIFICATION TECHNIQUES

1. Implicit Equations

Specify computation of square root of a number between 0 and some maximum value Y to a tolerance *E*.

$$(0<=X<=Y)\{ABS\_VALUE[(WHAT(X))^2-X]\}<=E$$

2. Recurrence Relation

Good for recursive computations. Example, Fibonacci numbers 0, 1, 1, 2, 3, 5, 8,...
FI(0) = 0, FI(1) = 1,
FI(n) = FI(n-1) + FI(n-2),n>=1.

3. Decision Tables Good for state-oriented specifications. Example, credit approval.

CONDTION	1	2	3	4
Credit Limit is Satis.	y	n	n	n
Pay exp. is favorable	-	y	n	n
Special Clear.	-	-	y	n
ACTION				
Approve order	x	x	x	
Reject order				x

4. Event Table

Specify action to be taken when events occur under different sets of conditions. f(M,E)=A, where M denotes the current set of operating conditions, E is the event of interest, and A is the action to be taken (two-dimension table). Example:

	EVENT		
MODE	E13	E37	E45
Start-up	A16	-	A14; A32
Steady	x	A6, A2	

(Start-up, E13) = A16

x means impossible
- means no action required
A14; A32 means A14 followed by A32
A6, A2 means A6 and A2 concurrently

5. Transition Table

Used to specify changes in the state of a system as a function of driving forces.

$f(s_i,c_j) = s_k$
means if
f is in state s_i with the condition c_j then the next state is s_k

Menu-driven software systems normally are based on such tables.

Example, RS flip-flop behavior.

R	S	Q	Q^n
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	-
1	1	1	-

AXIOMATIC SPECIFICATIONS

Use a set of stateless functions; each function has a set of pre- and post-conditions. Pre- and post-conditions are predicates over the inputs and outputs of a function. A predicate is a Boolean expression which is true or false and whose variables are the parameters of the function being specified.

Some predicates are:

1**. A>B and C>D**

2. *exists* **i, j, k** *in* **M...N:** $i^2 = j^2 + k^2$

3. *for-all i* in **1...10**, exists *j* in **1...10**: squares (i)$=j^2$

Stages of axiomatic specification of a function:

1. Establish the range of the input parameters over which the function is intended to behave correctly. Specify the input parameter constraints as a predicate.
2. Specify a predicate defining a condition which must hold on the output of the function if it behaves correctly.

3. Establish what changes (if any) are made to the function's input/output parameters. (note: in a purely mathematical function, no parameters would be changed. However, some programming languages modify parameters through call by name or call by reference.)
4. Combine these into pre- and post-conditions for the function.

Consider a search function that accepts an array of integers and some integer key of its parameters. The array members are sorted in increasing order. The function returns the array index of the member of the array whose value is equal to that of the key. The original input is unchanged. If the key is not matched, return an array index that is one number larger than the array's maximum index.

A specification is:

```
    function Search (X: INTEGER-ARRAY; Key: INTEGER)
        return INTEGER;
Pre:   exists i in X'FIRST...X'LAST:
       X(i)=Key and for-all i,j in X'FIRST...X'LAST:
       i<j implies X(i)<=X(j)
Post:  X''(Search (X, Key))=Key and X=X''
Error: Search (X, Key)=X'LAST + 1
```

X'' refers to the value of the array *X* after the function has been evaluated.

Formal Specification Homework Problem

Consider a function that accepts an array and then returns the greatest member of the array and the index of that member in the original array. The original input is unchanged.

If the array is empty, return an array index of -1.

Assume that X’FIRST and X’LAST return the indices of the first and last array members, respectively.