Lect 1

- **Formal methods** are techniques used to model complex systems as mathematical entities. By building a mathematically rigorous model of a complex system, it is possible to verify the system's properties in a more thorough fashion than empirical testing.
- Formal methods are system design techniques that use rigorously specified mathematical models to build software and hardware systems. In contrast to other design systems, formal methods use mathematical proof as a complement to system testing in order to ensure correct behavior. As systems become more complicated, and safety becomes a more important issue, the formal approach to system design offers another level of insurance.
- Formal methods differ from other design systems through the use of formal verification schemes, the basic principles of the system must be proven correct before they are accepted. Traditional system design has used extensive testing to verify behavior, but testing is capable of only finite conclusions.
- Formal design can be seen as a three step process, following the outline given here:

    1. **Formal Specification:** During the formal specification phase, the engineer rigorously defines a system using a modeling language. Modeling languages are fixed grammars which allow users to model complex structures out of predefined types. This process of formal specification is similar to the process of converting a word problem into algebraic notation.

       However, formal modeling languages are more rigorously defined: in a formal grammar, there is a distinction between WFFs (well-formed formulas) and non-WFFs (syntactically incorrect statements). Even at this stage, the distinction between WFF and non-WFF can help to specify the design.

    2. **Verification:** As stated above, formal methods differ from other specification systems by their heavy emphasis on provability and correctness. By building a system using a formal specification, the designer is actually developing a set of theorems about his system. By proving these theorems correct, the formal.
       Verification is a difficult process, largely because even the simplest system has several dozen theorems, each of which has to be proven.

    3. **Implementation**: Once the model has been specified and verified, it is implemented by converting the specification into code. As the difference between software and hardware design grows narrower, formal methods for developing embedded systems have been developed. LARCH, for example, has a VHDL implementation. Similarly, hardware systems such as the VIPER and AAMP5 processors have been developed using formal approaches.

- **Benefits Of Formal Models**

  Formal methods offer additional benefits outside of provability, and these benefits do deserve some mention. However, most of these benefits are available from other systems, and usually without the steep learning curve that formal methods require.

    - **Discipline**: By virtue of their rigor, formal systems require an engineer to think out his design in a more thorough fashion. In particular, a formal proof of correctness is going to require a rigorous specification of goals, not just operation. This thorough approach can help identify faulty reasoning far earlier than in traditional design.
    - **Precision**: Traditionally, disciplines have moved into jargons and formal notation as the weaknesses of natural language descriptions become more glaringly obvious. There is no reason that systems engineering should differ, and there are several formal methods which are used almost exclusively for notation.
    -

- **Weaknesses Of Formal Methods**

  There are several reasons why formal methods are not used as much as they might be, most stemming from overreaching on the part of formal methods advocates.

  - **Expense**: Because of the rigor involved, formal methods are always going to be more expensive than traditional approaches to engineering. However, given that software cost estimation is more of an art than a science, it is debatable exactly how much more expensive formal verification is. In general, formal methods involve a large initial cost followed by less consumption as the project progresses; this is a reverse from the normal cost model for software development.
  - **Limits Of Computational Models**: While not a universal problem, most formal methods introduce some form of computational model, usually hamstringing the operations allowed in order to make the notation elegant and the system provable. Unfortunately, these design limitations are usually considered intolerable from a developer's perspective.
  - **Usability**: Traditionally, formal methods have been judged on the richness of their descriptive model. That is, 'good' formal methods have described a wide variety of systems, and 'bad' formal methods have been limited in their descriptive capacities. While an all-encompassing formal description is attractive from a theoretical perspective, it invariably involved developing an incredibly complex and nuanced description language, which returns to the difficulties of natural language. Case studies of full formal methods often acknowledge the need for a less all-encompassing approach.

Available tools, techniques, and metrics

- Larch: Unlike most formal systems, LARCH provides two levels of specification. A general high-level modeling language, and a collection of implementation dialects designed to work with specific programming languages.
- SML: Standard Meta-Language is a strongly typed functional programming language originally designed for exploring ideas in type theory. SML has become the formal methods workhorse because of its strong typing and provability features.
- HOL: HOL, short for Higher Order Logic, is an automated theorem proving system. As with most automated theorem proving systems, HOL is a computer-aided proof tool: it proves simple theorems and assists in proving more complicated statements, but is still dependent on interaction with a trained operator. HOL has been extensively used for hardware verification, the VIPER chip being a good example.
- Petri Nets: Petri Nets are a good example of a very 'light' formal specification. Originally designed for modeling communications, Petri Nets are a graphically simple model for asynchronous processes.

  Problems with Conventional Specification:

  - Contradictions
  - Ambiguities
  - Vagueness
  - Incompleteness
  - Mixed levels of Abstraction

Symbols in mathematics terms:

Sets:

$S : \mathbb{P}\, X$	S is declared as a set of Xs.
$x \in S$	x is a member of S.
$x \notin S$	x is not a member of S.
$S \subseteq T$	S is a subset of T: Every member of S is also in T.
$S \cup T$	The union of S and T: It contains every member of S or T or both.
$S \cap T$	The intersection of S and T: It contains every member of both S and T.
$S \setminus T$	The difference of S and T: It contains every member of S except those also in T.
$\varnothing$	Empty set: It contains no members.
$\{x\}$	Singleton set: It contains just x.
$\mathbb{N}$	The set of natural numbers 0, 1, 2, ...
$S : \mathbb{F}\, X$	S is declared as a finite set of Xs.
max (S)	The maximum of the nonempty set of numbers S.

Functions:

$f : X \nrightarrow Y$	f is declared as a partial injection from X to Y
dom f	The domain of f: the set of values x for which $f(x)$ is defined.
ran f	The range of f: the set of values taken by $f(x)$ as x varies over the domain of f.
$f \oplus \{x \mapsto y\}$	A function that agrees with f except that x is mapped to y.
$\{x\} \lhd f$	A function like f, except that x is removed from its domain.

Logic:

$P \wedge Q$	P and Q: It is true if both P and Q are true.
$P \Rightarrow Q$	P implies Q: It is true if either Q is true or P is false.
$\theta S' = \theta S$	No components of schema S change in an operation.

Functional Specification

- Black Box
  - S-> [f: S*->R]->R
- State Box
  - S->[T, g:S*×T*->R×T]->R
- Clear Box
  - S->[T, g_{11}->[c_{g1}: g_{12}, g_{13}]]->R

Formal Methods Concepts

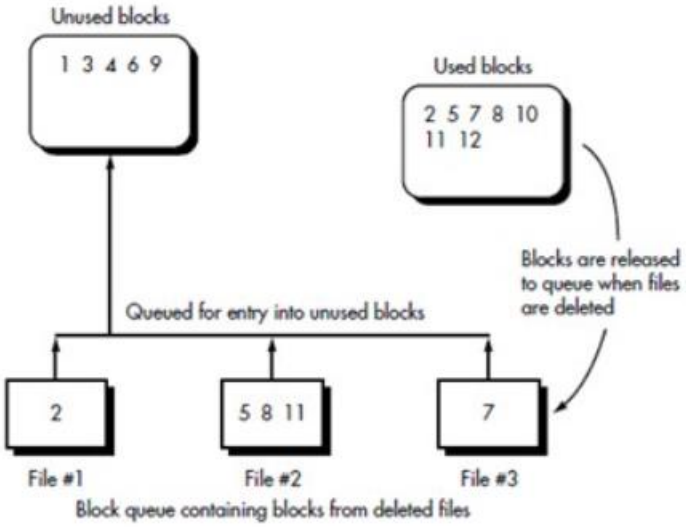
- *data invariant*—a condition that is true throughout the execution of the system that contains a collection of data
- *state*
  - Many formal languages, such as OCL (Section 21.7) , use the notion of states as they were discussed in Chapters 7 and 8, that is, a system can be in one of several states, each representing an externally observable mode of behavior.
  - The Z language (Section 21.7) defines a *state* as the stored data which a system accesses and alters
- *operation*—an action that takes place in a system and reads or writes data to a state
  - *precondition* defines the circumstances in which a particular operation is valid
  - *postcondition* defines what happens when an operation has completed its action

One of the known examples for defining the formal specifications:

Example 2: A Block Handler. One of the more important parts of a computer's operating system is the subsystem that maintains files created by users. Part of the filing subsystem is the *block handler*. Files in the file store are composed of blocks of storage that are held on a file storage device. During the operation of the computer, files will be created and deleted, requiring the acquisition and release of blocks of storage. In order to cope with this, the filing subsystem will maintain a reservoir of unused (free) blocks and keep track of blocks that are currently in use. When blocks are released from a deleted file they are normally added to a queue of blocks waiting to be added to the reservoir of unused blocks. This is shown in Figure 25.2. In this figure, a number of components are shown: the reservoir of unused blocks, the blocks that currently make up the files administered by the operating system, and those blocks that are waiting to be added to the reservoir. The waiting blocks are held in a queue, with each element of the queue containing a set of blocks from a deleted file.

Example 2: A Block Handler

- One of the more important parts of a computer's operating system is the subsystem that maintains files created by users. Part of the filing subsystem is the *block handler.*



States and Data Invariant

- No block will be marked as both unused and used.
- All the sets of blocks held in the queue will be subsets of the collection of currently used blocks.
- No elements of the queue will contain the same block numbers.
- The collection of used blocks and blocks that are unused will be the total collection of blocks that make up files.
- The collection of unused blocks will have no duplicate block numbers.
- The collection of used blocks will have no duplicate block numbers.

Operations

- add(): An operation which adds a collection of blocks to the end of the queue,
- remove(): An operation which removes a collection of used blocks from the front of the queue and place them in the collection of unused blocks
- check(): An operation which checks whether the queue of blocks is empty

Pre- & Postconditions

For the first operation:

Precondition: the blocks to be added must be in the collection of used blocks.

Postcondition: the collection of blocks is now found at the end of the queue.

For the second operation:
Precondition: the queue must have at least one item in it.
Postcondition: the blocks must be added to the collection of unused blocks.

For the third one:
Precondition: no
Postcondition: delivers the value of true if the queue is empty and false otherwise.

Sets and Constructive Specification

- A *set* is a collection of objects or elements and is used as a cornerstone of formal methods.
  - Enumeration
    - {C++, Pascal, Ada, COBOL, Java}
    - #{C++, Pascal, Ada, COBOL, Java} implies *cardinality* = 5
  - Constructive set specification is preferable to enumeration because it enables a succinct definition of large sets.
    - $\{x, y : N \mid x + y = 10 \bullet (x, y^2)\}$

Mathematical Concepts

- sets and constructive set specification
- set operators $\cup \cap \subseteq \subset$
- logic operators $\forall \exists \wedge \vee \neg$
  - e.g., $\forall i, j \in N (i > j) \rightarrow (i^2 > j^2)$
  - which states that, for every pair of values in the set of natural numbers, if i is greater than j, then i^2 is greater than j^2.
- Sequences: an enumerated collection of objects in which repetitions are allowed.

Set Operators

- A specialized set of symbology is used to represent set and logic operations.
  - Examples
    - The ϵ operator is used to indicate membership of a set. For example, the expression
      - x ϵ X
      - y ∉ X
    - The union operator, U, takes two sets and forms a set that contains all the elements in the set with duplicates eliminated.
      - {File1, File2, Tax, Compiler} U {NewTax, D2, D3, File2} is the set
      - {Filel, File2, Tax, Compiler, NewTax, D2, D3}
    - ∩, ×, ℙ,

Sequences

- Sequences are designated using angle brackets. For example, the preceding sequence would normally be written as
  - < Jones, Wilson, Shapiro, Estavez>
- Catenation, ⌢ is a binary operator that forms a sequence constructed by adding its second operand to the end of its first operand. For example,
  - <2, 3, 34, 1>⌢<12, 33, 34, 200 > = < 2, 3, 34, 1, 12, 33, 34, 200 >
- Other operators that can be applied to sequences are *head, tail, front,* and *last.*
  - *head* < 2, 3, 34, 1, 99, 101 > = 2
  - *tail* < 2, 3, 34, 1, 99, 101 > = <3, 34, 1, 99, 101 >
  - *last* < 2, 3, 34, 1, 99, 101 > = 101
  - *front* < 2, 3, 34, 1, 99, 101 > = <2, 3, 34, 1, 99>

Formal Specification Languages

- A formal specification language is usually composed of three primary components:
  - a syntax that defines the specific notation with which the specification is represented
  - semantics to help define a "universe of objects" [WIN90] that will be used to describe the system
  - a set of relations that define the rules that indicate which objects properly satisfy the specification
- The syntactic domain of a formal specification language is often based on a syntax that is derived from standard set theory notation and predicate calculus.
- The *semantic domain* of a specification language indicates how the language represents system requirements.

Formal Specification in Z Notation

- The block handler
  - The block handler maintains a reservoir of unused blocks and will also keep track of blocks that are currently in use. When blocks are released from a deleted file they are normally added to a queue of blocks waiting to be added to the reservoir of unused blocks.
    - The state

      used, free: P *BLOCKS*

      BlockQueue: *seq* P *BLOCKS*
    - Data Invariant

      $used \cap free = \Phi \wedge$

      $used \cup free = AllBlocks \wedge$

      $\forall i$: dom *BlockQueue* • $BlockQueue\ i \subseteq used \wedge$

      $\forall i, j$: dom *BlockQueue* • $i \neq j \Rightarrow BlockQueue\ i \cap BlockQueue\ j = \Phi$
  - Remove:
    - Precondition

      #BlockQueue > 0
    - Postcondition

      $used' = used \setminus head\ BlockQueue \wedge$

      $free' = free \cup head\ BlockQueue \wedge$

      $BlockQueue' = tail\ BlockQueue$

Pages 578-579