Specification Languages and Models

A Term Paper By:
Tushar Chaudhary (1998428)

as a Curriculum Requirement for

CS812: High Level Design and Modelling

MODELLING:

System design is the process of implementing a desired functionality using a set of physical components. To achieve precision in the design, we need to think of a system as a collection of simpler subsystems, or pieces. There are various methods of decomposing functionality into these simpler pieces. What distinguishes these methods are the types of pieces and the rules for composing these pieces to create system functionality. Each of these methods is called a **model**.

Characteristics / required features of a model

- Formal
- Complete
- Comprehensible
- Computation
- Concurrency
- Synchronism
- Non-determinism
- State transitions
- Hierarchy
- Communication
- Synchronization
- Behavioral completion
- Exception handling
- Timings

Types of Models

- *State oriented*
- *Activity oriented*
  - *data flow*
  - *control Flow*
- *Structure oriented*
- *Data oriented*
- *Heterogeneous models*

State oriented model examples

- **FiniteStateMachine**

A finite-state machine is an example of a state-oriented model. The FSM model consists of a set of states, a set of transition between states, and a set of actions associated with these states or transitions. Formally, a FSM is a quintuple

$<S, I, O, f : S \times I \rightarrow S, h : S \times I \rightarrow O>$

where $S = \{s_1, s_2, s_{3,}$□□.., $s_l\}$ is a set of states, $I = \{i_1, i_2, i_{3,}$□□.., $i_m\}$ is a set of inputs and $O = \{o_1, o_2, o_{3,}$□□.., $s_n\}$ is a set of outputs; f is a next-state function, which determines the next state from the current state and input and h is an output function, which determines the outputs, also from the current state and input. Fig. 1 shows a finite-state machine description of an Elevator-Control System.

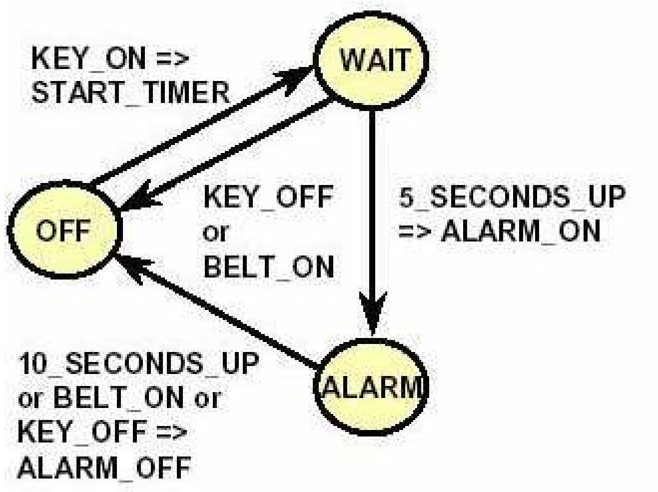


Fig. 1 □ FSM description of Elevator Control System

□ **FSMs with Data Path**

To overcome the state-explosion problem in a FSM, there is the Finite State Machine with Data Path (FSMD) which is formally described as the quintuple:

<S, I ÈSTAT, O ÈA, f, h>

§ S, I, O States, Inputs, Outputs

§ V, E, A Variables, Expressions, Assignments

§ f : S ´ (I ÈE) ® S

§ h : S ´ (I ÈE) ® (O ÈA)

□ **Petri Net**

FSMs and FSMDs cannot handle concurrency. A model specifically designed to model systems that somprise interacting concurrent tasks is the Petri-Net model.
Formally, a Petri net is a quintuple:

<P, T, I, O, u>

where P = {p_1, $p_{2,}$ $p_{3,}$□□.., p_m }is a set of places, T = {t_1, $t_{2,}$ $t_{3,}$□□.., t_n } is a set of transitions, and P and T are disjoint.

I : T ® P^+ O : T ® P^+

u : P ® N Marking

f : T ® E

h : T ® A

Petri nets can model sequencing, non-deterministic branching, synchronization, resource contention and concurrency. Fig 2. shows the transitions in a Petri net.

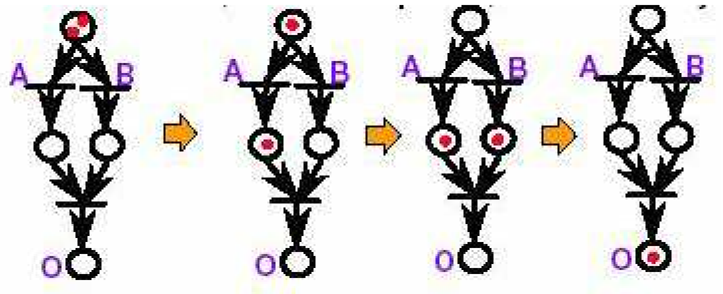


Fig. 2 □ Petri net transitions

□ **Hierarchical Concurrent FSMs**

The hierarchical concurrent finite-state machine (HCFSM) is essentially an extension of the FSM model, which adds support for hierarchy and concurrency, thus eliminating the potential for state and arc explosion that occurs when trying to describe hierarchical and concurrent systems in an FSM. Fig 3 shows an HCFSM.

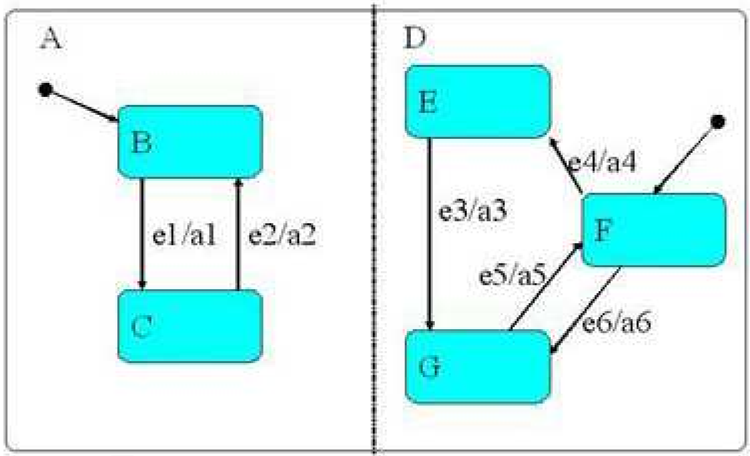


Fig. 3 □ Hierarchical Concurrent Finite State Machine

Activity oriented models

State oriented models are used mostly for reactive systems in which the system□ s state changes in response to some external events. Activity oriented models, in contrast are used mainly for transformational systems, in which the outputs are determined by a set of computations on the system□ s inputs.

□ **Data Flow Graphs**

☐ Nodes : Inputs, Outputs, Activities (computations)

☐ Edges : Flow of data

Fig. 4 shows a Data Flow graph.

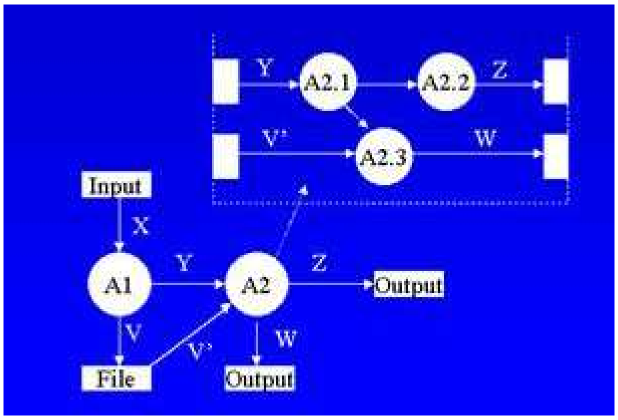


Fig. 4 ☐ Data Flow Graph

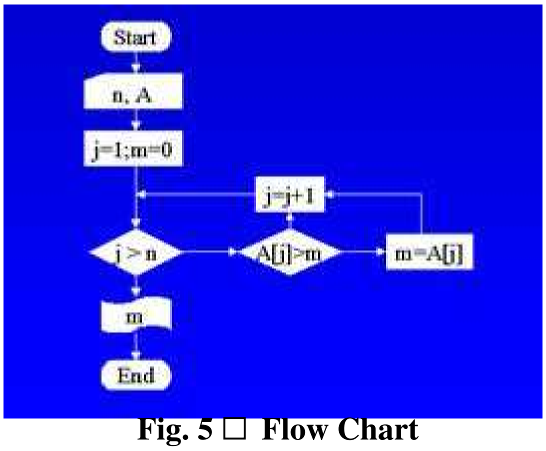


Fig. 5 ☐ Flow Chart

☐ **Flow Charts**

☐ Nodes : Start, End, Computation, Decision, Input, Output

☐ Edges : Control flow

Fig. 5 shows a Flow Chart.

Structure oriented models

□ **Component-connectivity diagram**

Component-connectivity diagrams (CCD) is a class of structure oriented models which are used to describe a system□ s physical structure as opposed to its functionality. A CCD represents a set of system components and their interconnections.
Fig. 6 shows a CCD.

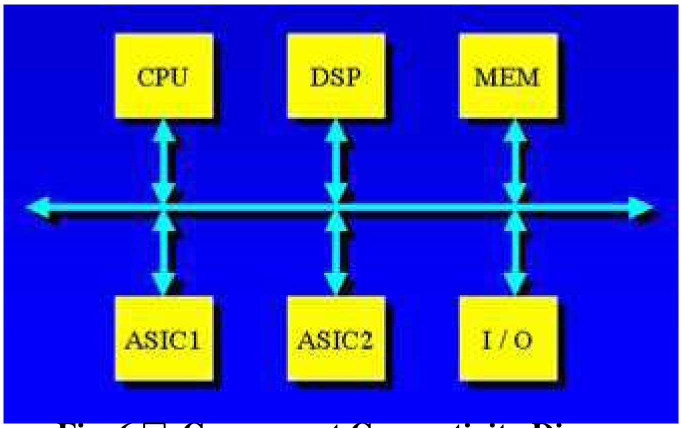


Fig. 6 □ Component Connectivity Diagram

Data-oriented models

□ Entity-Relationships diagram

  □ Entities : have attributes

  □ Relationships : relates some entities

Fig. 7 shows an Entity Relationship Diagram.

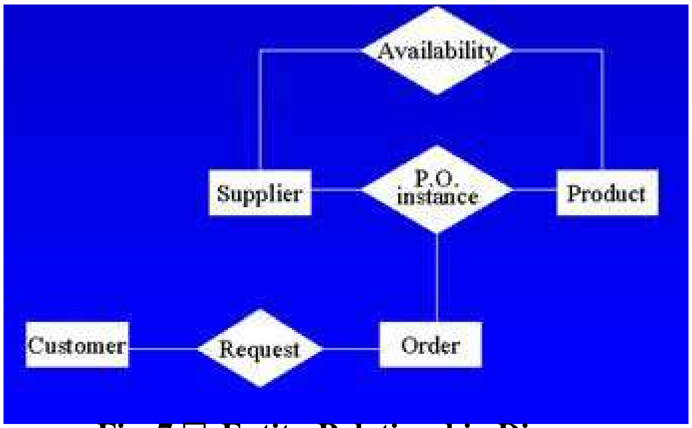


Fig. 7 □ Entity-Relationship Diagram

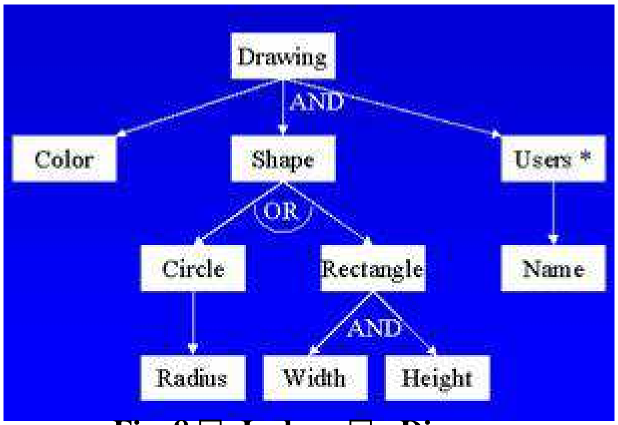


Fig. 8 □ Jackson□ s Diagram

□ **Jackson□ s Diagram**

A Jackson□ s Diagram is useful for representing data that would need to be decomposed in a way that is much more complex than that appropriate for simple records. The data in a Jackson□ s Diagram is decomposed by means of a tree-type structure, in which the leaf nodes are the basic data types and the non-leaf nodes are the composite data types of data that have been obtained through various operations like composition (AND), selection (OR), and iteration(*).

Fig. 8 shows a Jackson□ s Diagram.

Heterogeneous models

- **Control /Data Flow Graph (CDFG)**

The CDFG is a heterogeneous model, designed to combine the advantages of a flowchart (Control Flow Graph) with those of DFG models. Fig. 9 shows a CDFG. The primary advantage of a CDFG is that it corrects the inability of a DFG to represent the control of a system, as well as the inability of a CFG to represent data dependencies.

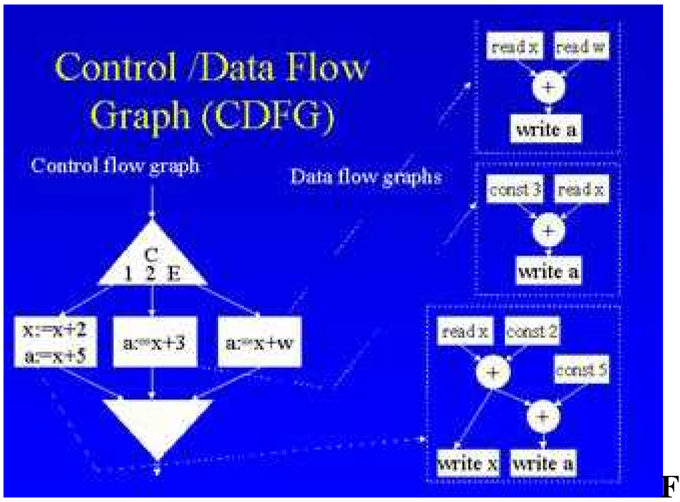


Fig. 9 □ A Control/Data Flow Graph

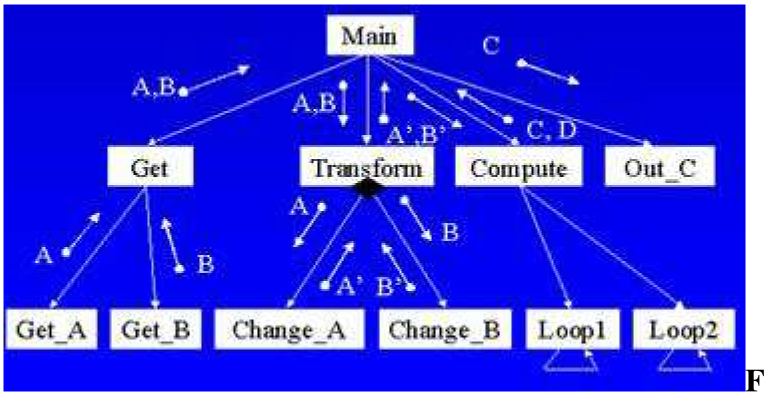


Fig. 10 □ Structure Chart

- **Structure Chart**

  The Structure Chart model is another heterogeneous model, in this case designed to specify the data, activities and control accesses of a system in a single representation. The structure chart consists of a set of nodes, representing activities, and a set of edges, which represent the procedure or function calls in a programming language. The data passed between activities is indicated on the edges. Fig. 10 shows a Structure Chart.

- **Object Oriented Model**

  The Object Oriented model is characterized by its tendency to view a system as a set of objects. In terms of this model, each object would consist of a set of data and a set of operations for transforming that data.
  Object-oriented models have several features not found in other models.

  - data abstraction or information hiding
  - concurrency

  Fig. 11 shows describes the object oriented model.

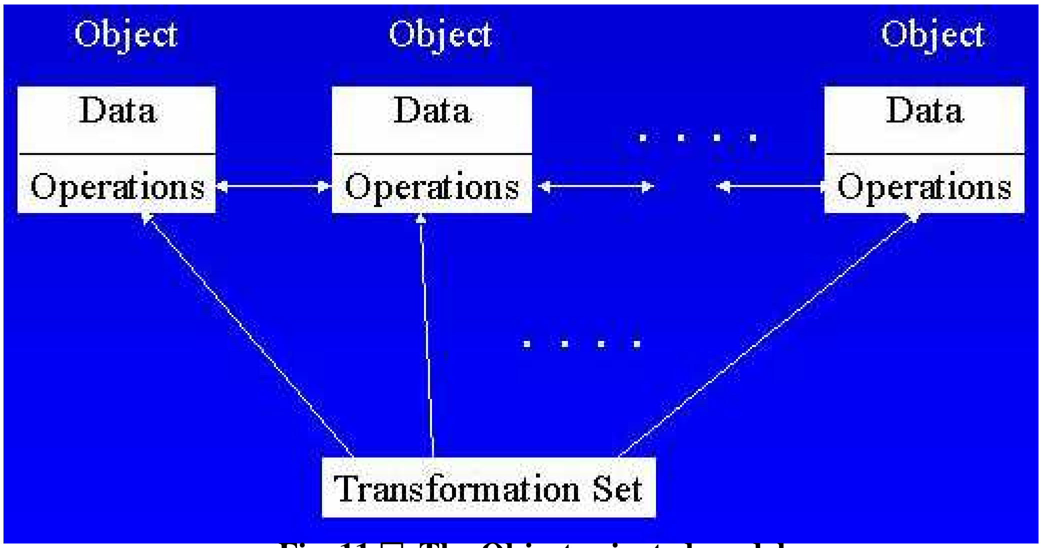
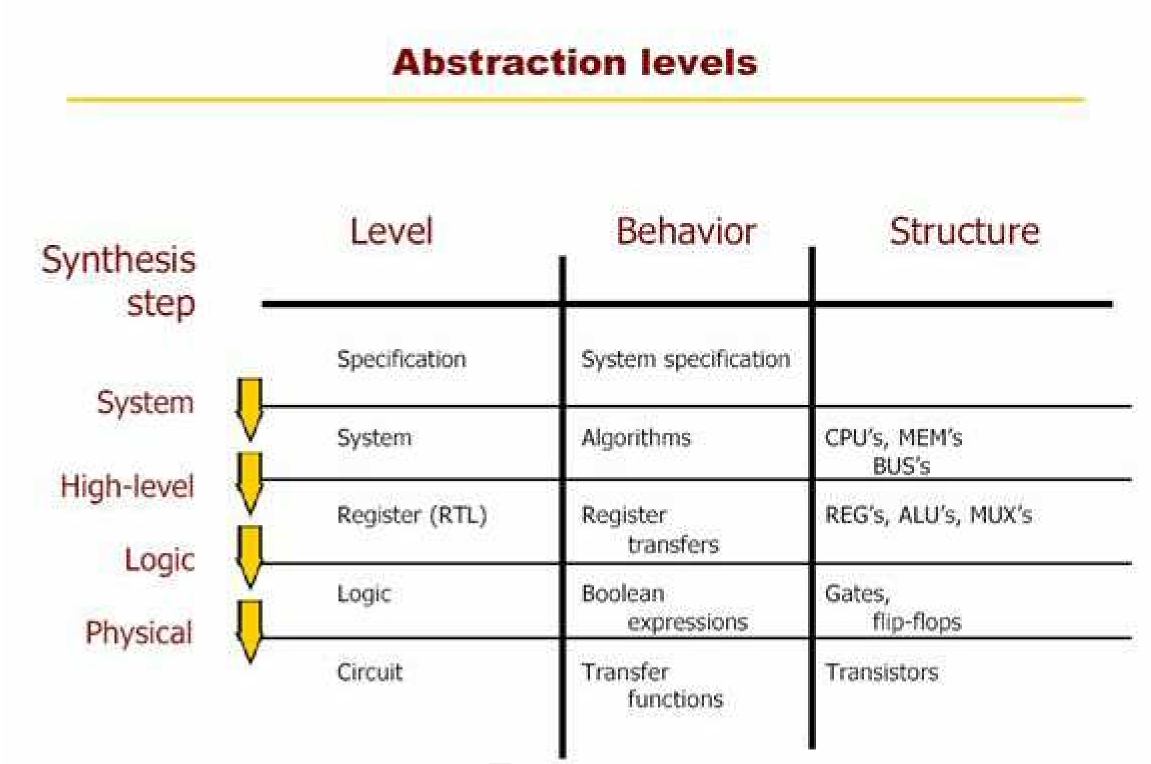


Fig. 11 □ The Object oriented model

LANGUAGES:

Characteristics of an Ideal HDL:

- Supports Multiple levels of Abstraction
- Concise
- Describes Functional Blocks and their interconnections
- Well suited for:
  - Synthesis
  - Verification



Abstraction levels

Synthesis step

- System
- High-level
- Logic
- Physical

Level	Behavior	Structure
Specification	System specification	
System	Algorithms	CPU's, MEM's BUS's
Register (RTL)	Register transfers	REG's, ALU's, MUX's
Logic	Boolean expressions	Gates, flip-flops
Circuit	Transfer functions	Transistors

Fig. 12 □ Abstraction Levels

Time models

- Continuous time
- Discrete time, cycle based
- Multi-rate discrete time, cycle based
- Totally-ordered discrete events
- Partially-ordered events

VLSI System Design:

- Hardware abstraction levels, timing and data flow computations
- Hardware Description Languages (HDLs)
- E.g., VHDL, HardwareC, SpecCharts, SpecC

Protocol specification:

- Formal description to enable verification
- LOTOS
  - Based on process algebra and abstract data types
  - Specification is executable
- SDL
  - Based on extended FSMs
  - Both graphical and textual modeling
- ESTELLE
  - Pascal-like programming language
  - Implementation details necessary

Reactive (real-time) system design:

- Need to guarantee (timely) response to events
- ESTEREL
    - Based on events
    - Synchronous time model
- LUSTRE, SIGNAL
    - Based on programmable automaton
    - Simple time aspects in LUSTRE but more advanced in SIGNAL
- Petri net tools
    - Based on Petri nets
    - Not always formally defined

Programming languages:

- Often lacking constructs for concurrency and timing
- Extensions break the language standards
- E.g., C, Ada, Java, Fortran

Formal methods:

- Offers high abstraction but perhaps not all necessary concepts
- VDM, Z
    - Based on set theory and predicate logic
- B
    - Based on Abstract Machine Notation

Structural Analysis:

- Systematic approach for structuring code and data in software systems
- ☐ Divide and conquer☐
- E.g., OO, UML

Continuous languages:

- High-level modeling based on differential equations
- Used for DSP, mechanical and hydraulic design
- Large expressiveness makes verification and synthesis hard
- E.g., Matlab, Matrixx, Mathematica

VHDL

Positive

- PL + concurrency
- VHDL: Shared memory with signals
- VHDL: Limited behavioral hierarchy

Negative

- Formal
- State transitions
- No completion of processes
- VHDL: Limited Exceptions
- VHDL: Only one level of
- concurrent hierarchy

StateCharts

Positive
- HCFSM
- Formal

Negative
- PL
- Structure
- Behavioral completion
- Dataflow

SDL

Positive
- Communication oriented
    - Asynchronous, blocking read, non-blocking write
- Formal

Negative
- PL
- Data flow
- Exceptions

SILAGE(DFL)

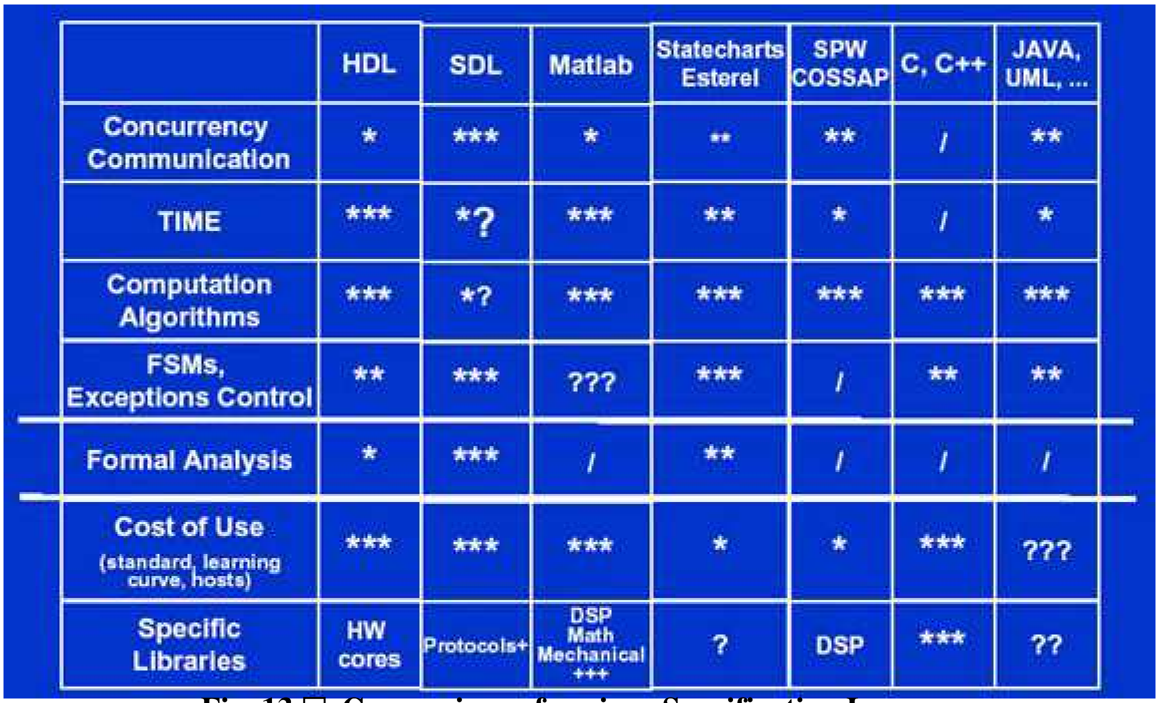
Positive
- Single assignment (data flow)
- Formal
- DSP applications

Negative
- Imperative PL
- State transitions
- Behavioral hierarchy
- Exceptions

ESTEREL

Positive
- Synchronous PL
    - perfect synchronous
- Communication
    - signal broadcasting
- Formal

Negative
- State transitions
- Timing
- Data flow



	HDL	SDL	Matlab	Statecharts Esterel	SPW COSSAP	C, C++	JAVA, UML, ...
Concurrency Communication	*	***	*	**	**	/	**
TIME	***	*?	***	**	*	/	*
Computation Algorithms	***	*?	***	***	***	***	***
FSMs, Exceptions Control	**	***	???	***	/	**	**
Formal Analysis	*	***	/	**	/	/	/
Cost of Use (standard, learning curve, hosts)	***	***	***	*	*	***	???
Specific Libraries	HW cores	Protocols+	DSP Math Mechanical +++	?	DSP	***	??

Fig. 13 □ Comparison of various Specification Languages

Use of Programming Languages for High Level System Modeling:

Java has several characteristics that make it suitable for system specification. However, static control and dataflow analysis of Java programs is problematic because Java classes are dynamically linked. This can however be taken care of by using a technique that pre-allocates most class instances and aggressively resolves memory aliasing using global analysis. The output of the analysis is a control dataflow graph for the input specification.

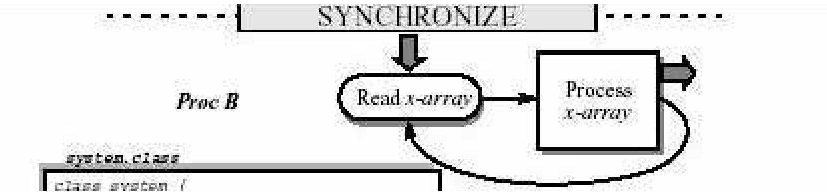
Hardware-software specification languages currently used by system designers can be divided into software programming languages and hardware description languages. Software languages such as C or C++ generate high-performance executable specifications of system behavior for functional validation. Software languages are traditionally based on a sequential execution model derived from the execution semantics of general purpose processors. However, software languages generally do not have support for modeling concurrency or dealing with hardware issues such as timing or events. These deficiencies can be overcome by providing the designer with library packages that emulate the missing features. A more serious problem is that software languages allow the use of indirect memory referencing which is very difficult to analyze statically. This makes it difficult for static analysis to extract implicit concurrency within the specification. Hardware description languages such as Verilog and VHDL are optimized for specifying hardware with support for a variety of hardware characteristics such as hierarchy, fine-grained concurrency, and elaborate timing constructs. Esterel is another specification language similar to Verilog with more constructs for handling exceptions. SpecCharts builds on a graphical structural hierarchy while using VHDL to specify the implementations of the various structures in the hierarchy. These languages do not have high-level programming constructs, and this limits their expressiveness and makes it difficult to specify software. Furthermore, these languages are based on execution models that require a great deal of run-time interpretation such as event-driven semantics. This results in low-performance execution compared to software languages.

As a solution, there have been analyses on the use of Java as the specification language for hardware-software systems by identifying key language characteristics that enable both efficient functional validation and concurrency exploration by the hardware-software partitioner. Java is a general-purpose, concurrent, object-oriented, platform-independent programming language.

Since Java classes are predominantly linked at run-time, references to class instances cannot be resolved at compile time. This presents a challenge to static analyzers in determining data flow through data field accesses and control flow through method calls.

It is desirable for the hardware-software co-design process to use a single specification language for design entry because specifications using different languages for software and hardware combine different execution models. This makes these specifications difficult to simulate and to analyze. Some researchers begin with a software programming language usually C++ and extend this language with constructs to support concurrency, timing, and events by providing library packages or by adding new language constructs. Examples of this approach are Scenic and V++. We take a slightly different approach. Instead of requiring the designer to specify the hardware implementation details in the specification, in our approach the designer models the complete system in an algorithmic or behavioral fashion. Software languages are well-suited for this type of modeling. Once the specification is complete, an automatic compilation process is used to analyze the specification to identify the coarse-grained concurrency described by the designer and uncover the finer-grained concurrency implicit in the specification. The partitioning and synthesis steps of the hardware-software co-design process use the concurrency uncovered by this analysis to create an optimized hardware-software system. The specification language used with this approach must have the ability to specify explicit concurrency and make it easy to uncover the implicit concurrency.

Coarse-grained concurrency is intuitive for the designer to specify because hardware-software systems are often conceptualized as sets of concurrent behaviors [2]. Java is a multi-threaded language and can readily express this sort of concurrency. Such concurrent behaviors can be modeled by sub-classing the *Thread* class and overriding its *run* method to encode the thread behavior as shown in Figure 14.


SYNCHRONIZE
Proc B
Read x-array
Process x-array
system.class
class system {