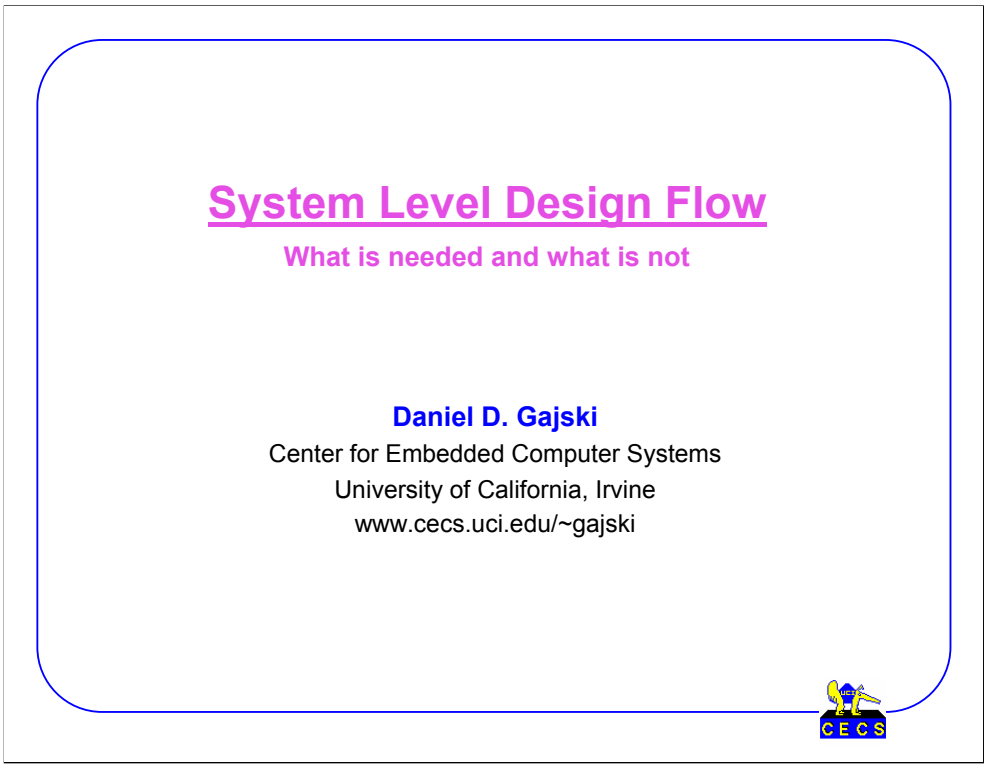
System Level Design Flow

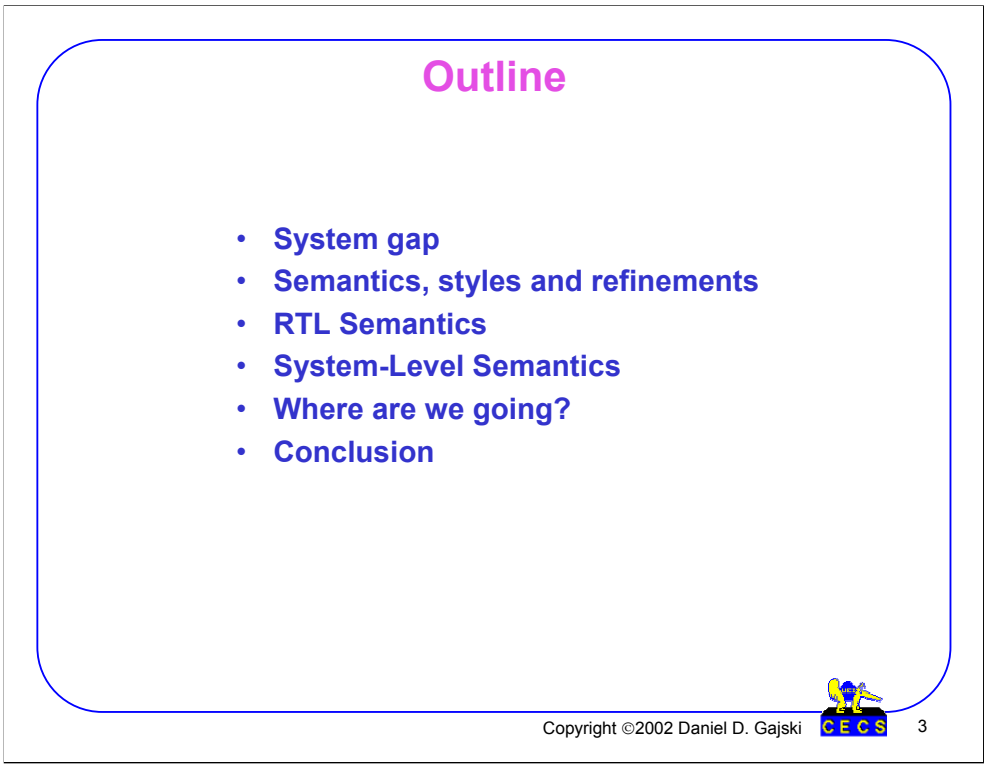
What is needed and what is not

Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
www.cecs.uci.edu/~gajski



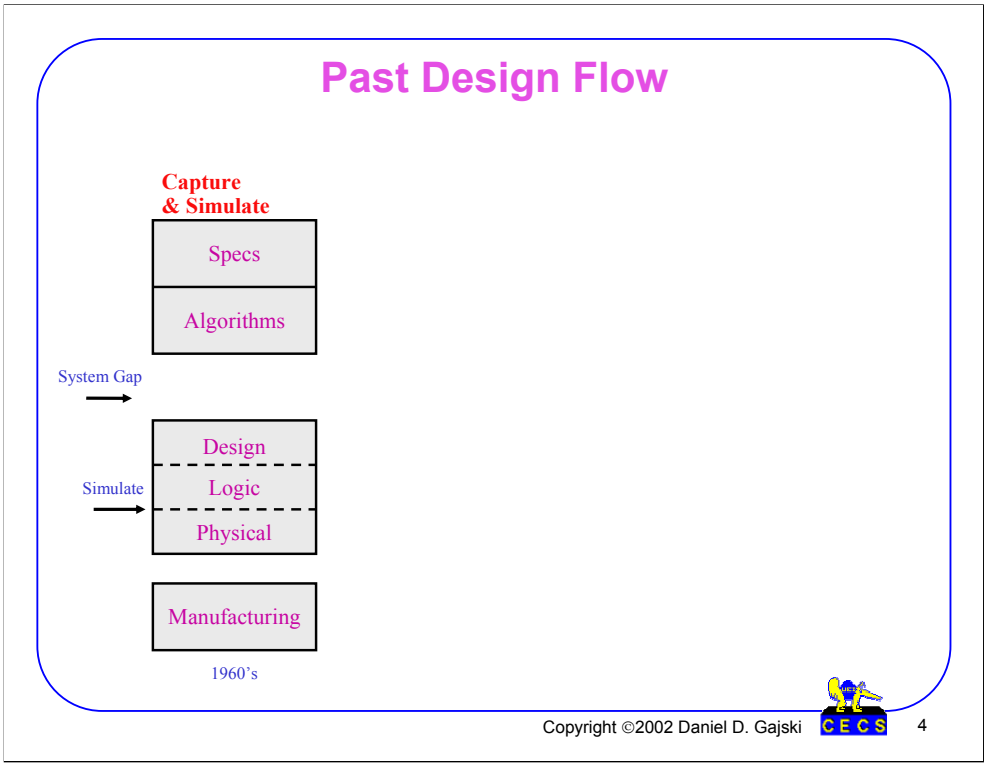
Introduction

With complexities of Systems-on-Chip rising almost daily, the design community has been searching for new methodology that can handle given complexities with increased productivity and decreased times-to-market. The obvious solution that comes to mind is increasing levels of abstraction, or in other words, increasing the size of the basic building blocks. However, it is not clear what these basic blocks should be beyond the obvious processors and memories. Furthermore, if a design consists of SW and HW the modeling language should be based on C since standard processors come only with C compilers. Unfortunately, C language was developed for describing software and not hardware. It is missing basic constructs for expressing hardware concurrency and communication among components. Therefore, we need a language that can be compiled with standard compilers and that is capable of modeling hardware and software on different levels of abstraction including cycle-level accuracy.



Outline

In order to find the solution for system-level design flow, we will look first at the system gap between SW and HW designs and then try to bridge this gap by looking at different levels of abstraction, define different models on each level and propose model refinements that will bring the specification to a cycle-accurate implementation. We will exemplify this by looking at the RTL and SL abstraction levels. From this point of view we will try to analyze the basic approaches in the academia and the industry today, and try to find out where we, as a design community, are going. We will finish with a prediction and a roadmap to achieve the ultimate goal of increasing productivity by more then 1000X and reducing expertise level needed for design of complex systems to the basic principles of design science only.

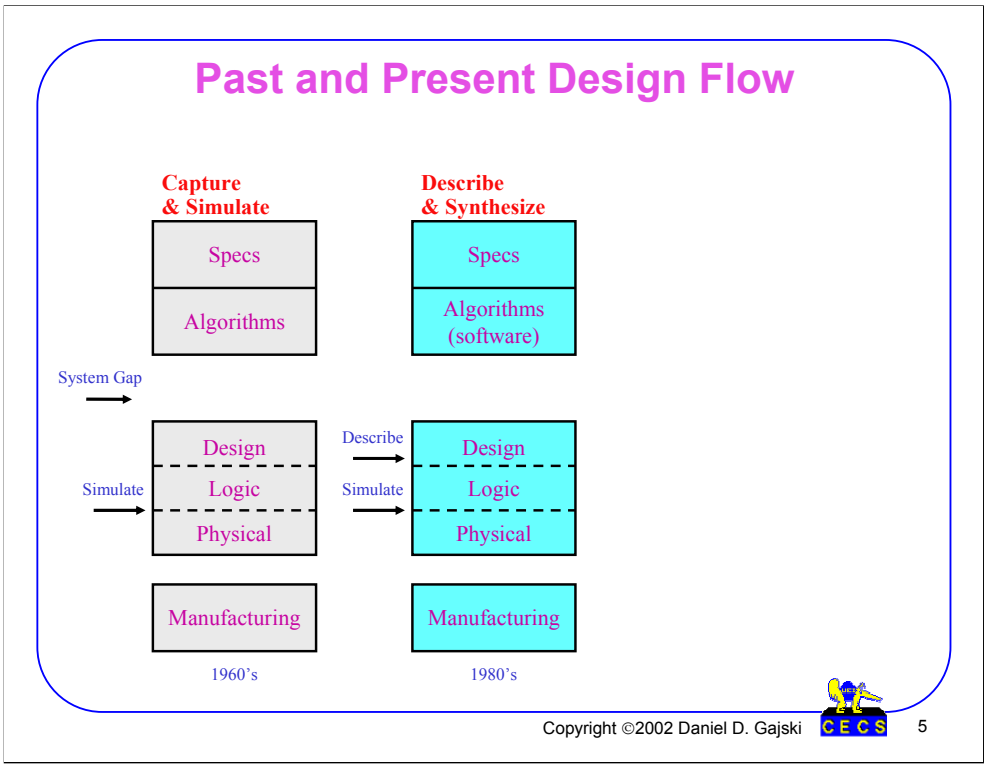


Past Design Flow

Design methodology has been changing with increase in complexity**.** We can distinguish three different phases over the last 40 years:

(a) Capture-and-Simulate Methodology (approximately from 1960s to 1980s)

In this methodology software and hardware design was separated by a system gap. SW designers tested some algorithms and possibly wrote the requirements document and initial specification. This specification was given to HW designers who read it and started system design with a block diagram. They did not know whether their design will satisfy the specification until the gate level design was produced. When gate netlist was captured and simulated designers could find whether system really worked as specified. Usually it did not work as specified, and therefore specification was changed. This approach started the myth that specification is never complete. It took many years to realize that specification is independent of its implementation. The main obstacle to close the gap was the design flow is which designers waited until the gate level design was finished to verify the system behavior. Since they captured system design once at the end of design cycle, before simulation this methodology is called capture-and-simulate.

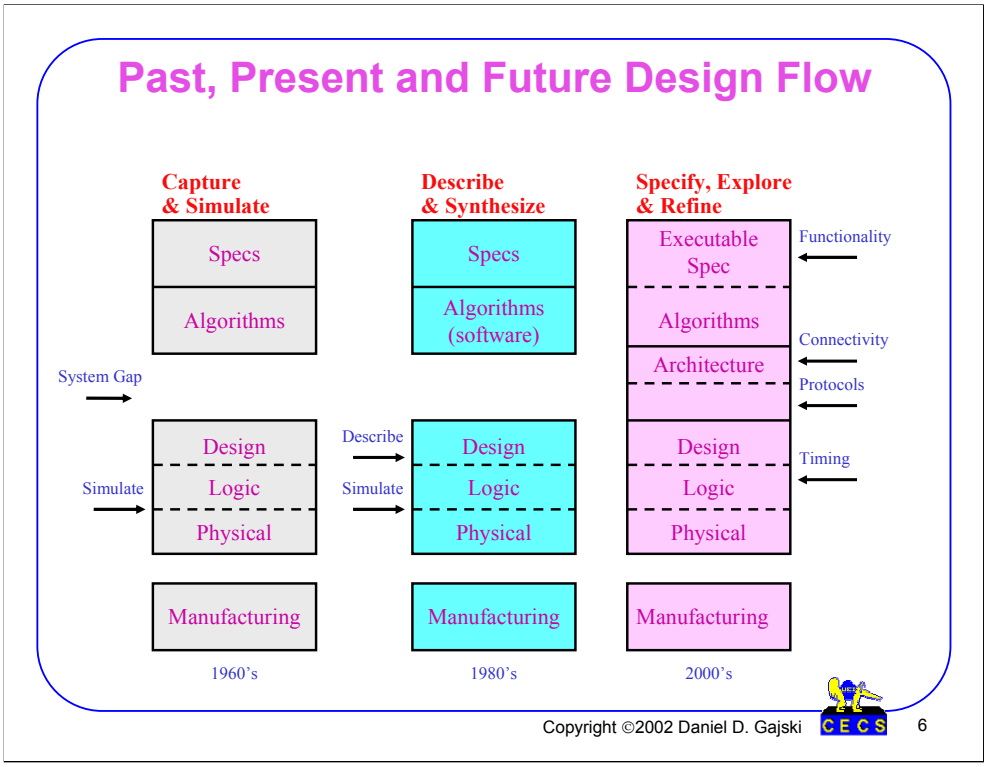


Present Design Flow

(b) Describe-and-Synthesize methodology (late 1980s to late 1990s)

1980's brought us logic synthesis, which has significantly altered the design flow, since designers first specify what they want in terms of Boolean equations or FSM descriptions and the synthesis tools generate the implementation in terms of a gate netlist. Therefore, the behavior or the function comes first and the structure or implementation next. Also, there are two models to simulate: behavior (function) and gate-level structure (netlist). Thus, in this methodology specification comes before implementation and they are both simulatable. Also, it is possible to verify their equivalence since they can be both reduced to a canonical form in principle. In practice, today's designers are too large for this kind of equivalence checking.

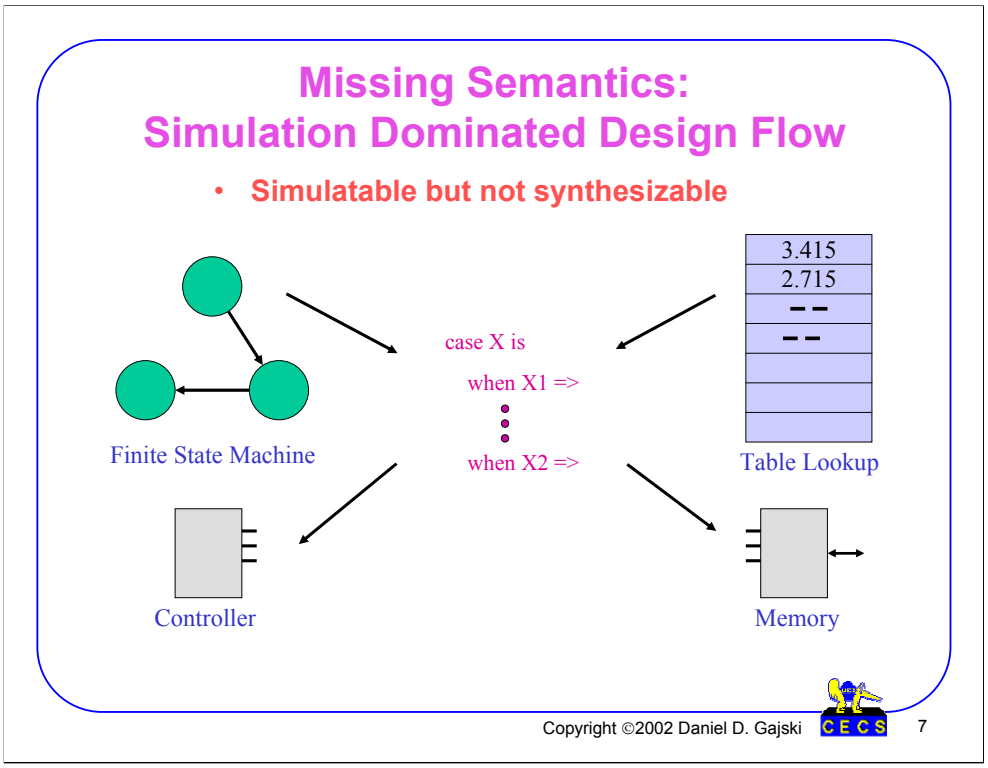
By late 1990s the logic level has been abstracted to RTL or cycle-accurate description and synthesis. Therefore, we have two abstraction levels (RTL and gate levels) and two different models on each level (behavioral and structural). However, the system gap still persists.



Future Design Flow

(c) Specify, Explore-and-Refine Methodology (from early 2000s)

In order to close the gap we must increase level of abstraction from RTL to SL. On SL level we have executable specification that represents the system behavior or function and structural models with emphasis on connectivity or communication protocols. Each model is used to prove some system property such as functionality, connectivity, communication and so on. In any case we have to deal with several models in order to close the gap. Each model can be considered to be a specification for the next level model in which more detail in implementation is added. Therefore specify-explore-refine (SER) methodology represents a sequence of models in which each model is an refinement of the previous one. Thus, SER methodology follows the natural design process where designers specify the intent first, explore possibilities and then refine the model according to their decisions. Thus, SER flow can be viewed as several interactions of the basic describe-and-synthesize methodology.

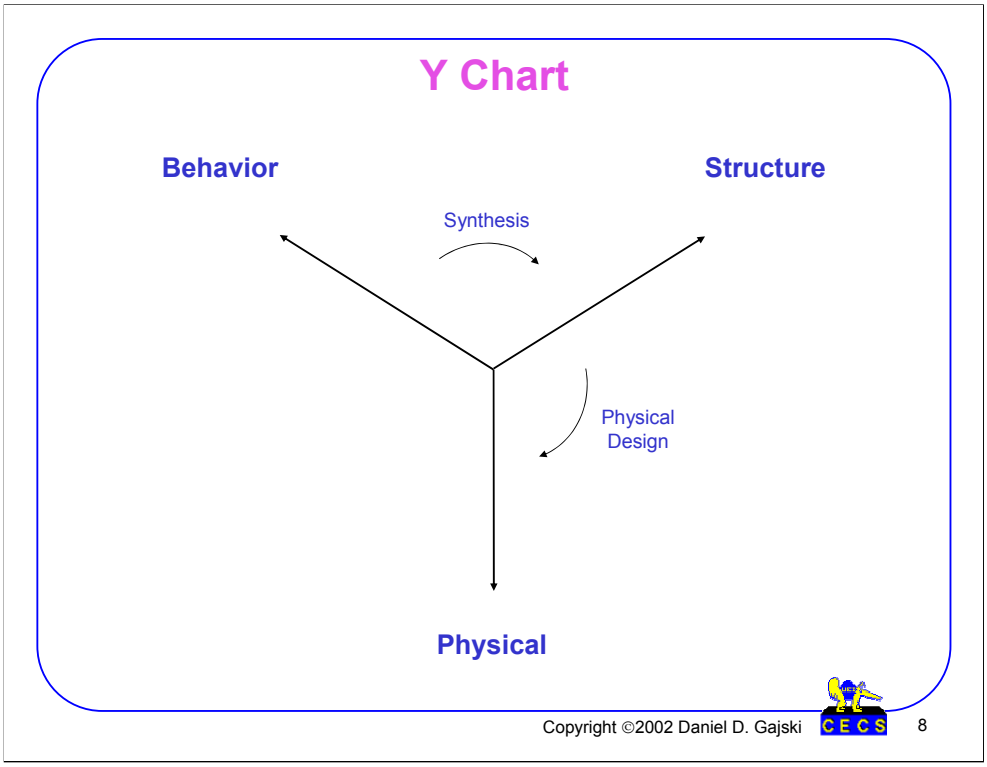


Missing Semantics

With introduction of SL abstraction, designers have to generate even more models. One obvious solution is to automatically refine one model into another. However, that requires well defined model semantics, or , in other words, good understanding what is meant by the given model. This is not as simple as it sounds, since design methodologies and EDA industry have been dominated by simulation based methodologies in the past. For example, all HDL (such as Verilog, VHDL, SystemC, and others) are simulatable but not synthetizable or verifiable.

As an example of this problem, we can look at a simple case statement in any of the languages. It can be used to model a FSM or a look-up table, for example. However, FSMs and look-up tables require different implementations: a FSM can be implemented with a controller while a look-up table is implemented with a memory. On the other hand, using memory to implement a FSM or control logic to implement a table is not very efficient and not acceptable by any designer. Therefore, the model which uses case statement to model FSMs and tables is good for simulation but not good for implementation since a designer does not know what was meant by the case statement.

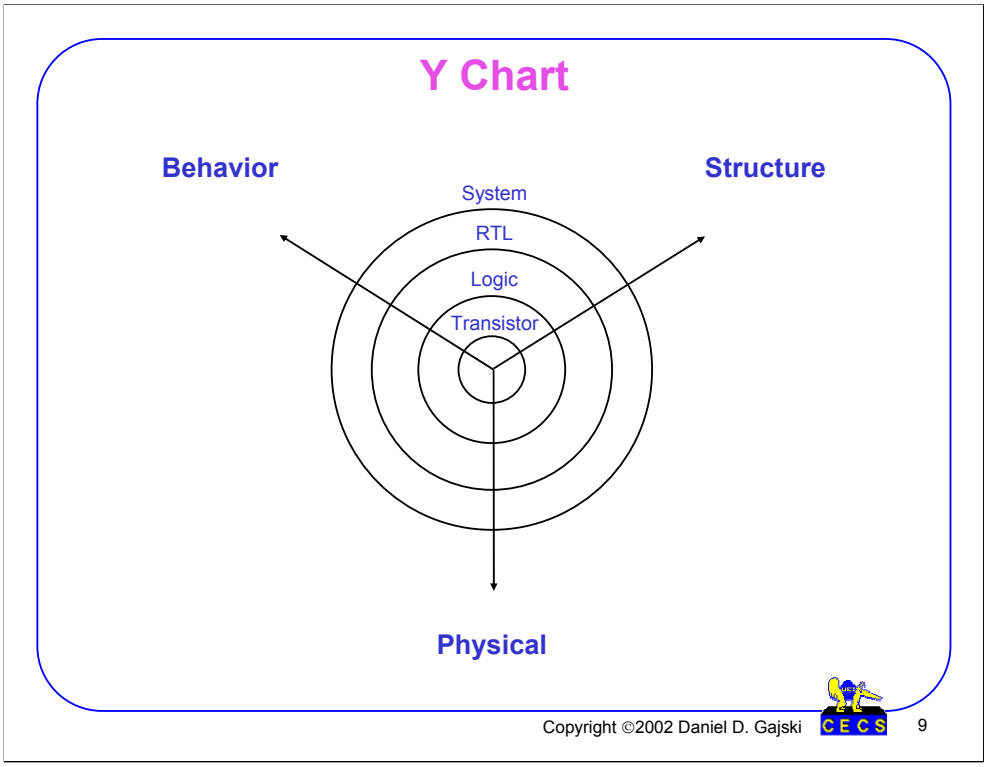
Thus, clean and unambigues semantics is needed for refinement, synthesis and verification. This semantics is missing from most of the simulation-oriented languages.



Y-Chart

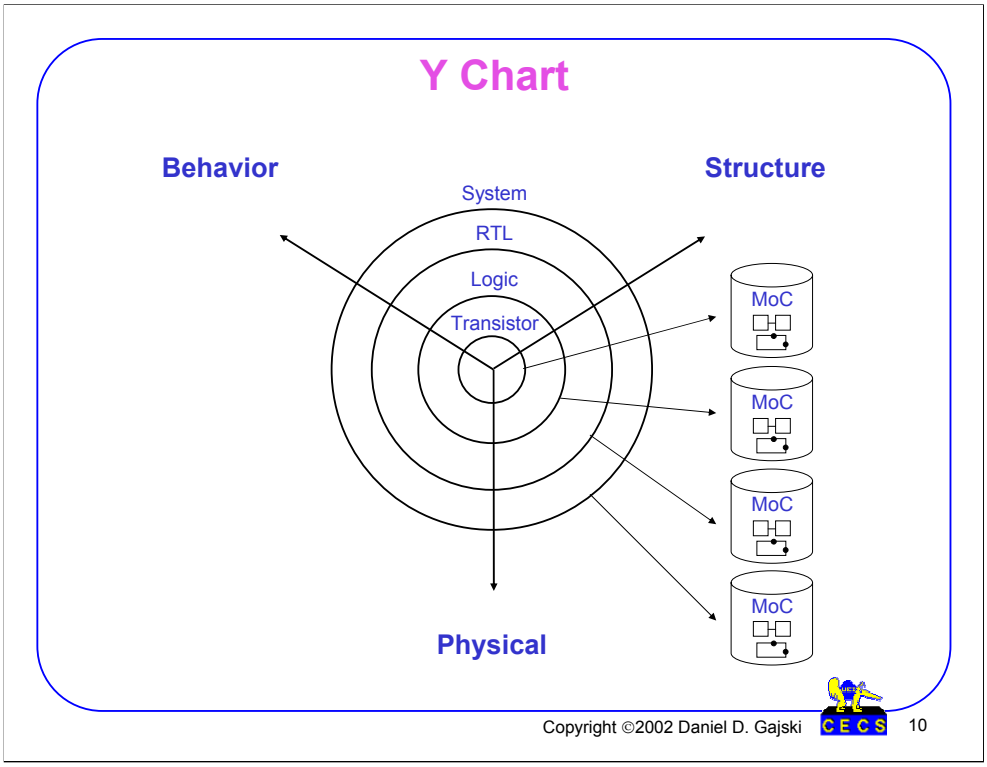
In order to explain the relationship between different abstraction levels, design models and design methodologies or design flow we will use Y-Chart, which was developed in 1983 to explain different design methodologies or design flows. The Y-Chart makes the assumption that each design, no matter how complex, can be modeled in three basic ways emphasizing different properties of the same design.

Therefore, Y-Chart has three axes representing design behavior (function, specification), design structure (connected components, block diagram), and physical design (layout, boards, packages). Behavior represents a design as a black box and describes its outputs in terms of its inputs and time. The black-box behavior does not indicate in anyway how to implement the black box or what its structure is. That is given on the structure axis, where black box is represented as a set of components and connections. Although, behavior of the black box can be derived from its component behaviors such an obtained behavior may be difficult to understand. Physical design adds dimensionality to the structure. It specifies size (hight and width) of each component, the position of each component, each port and each connection on the silicon substrate or board or any other container.



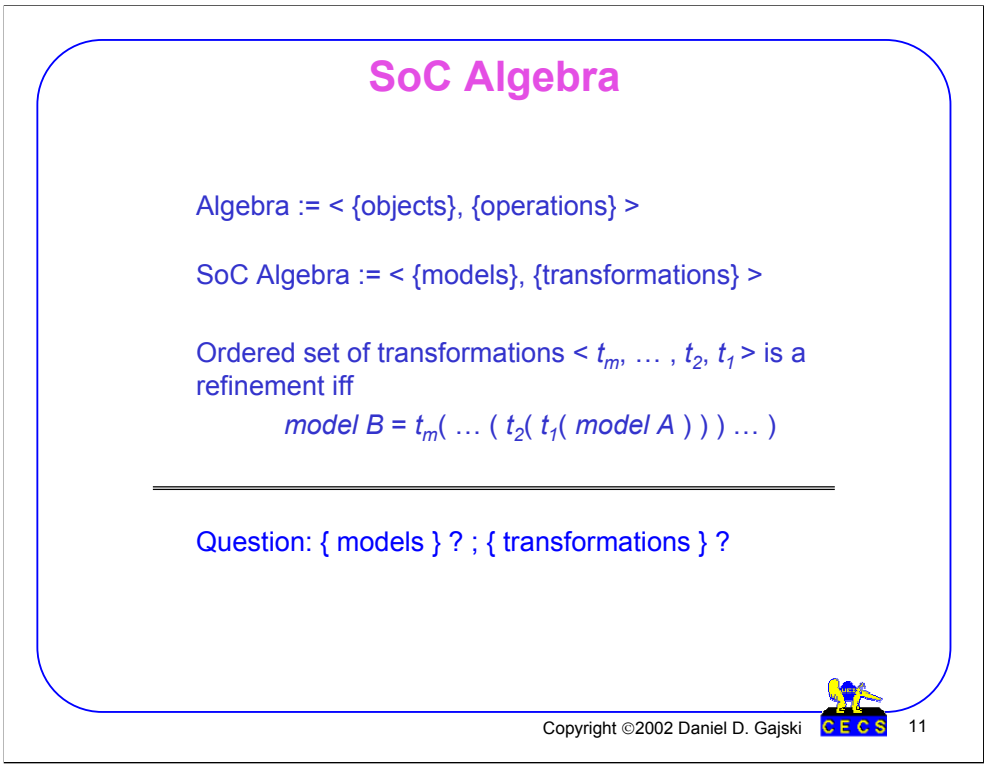
Y-Chart (continued)

Y-Chart can also represent design on different abstraction levels identified by concentric circles around the origin. Usually, four levels are used: Transistor, Logic, Register-transfers and System levels. The name of the abstraction level is derived by the main component used in the structure on this abstraction level. Thus, the main components on Transistor level are N or P-type transistors, while on Logic level they are gates and flip-flops. On the Register-transfers level the main components are registers, register files and functional units such as ALUs. While on the System level they are processors, memories and buses.



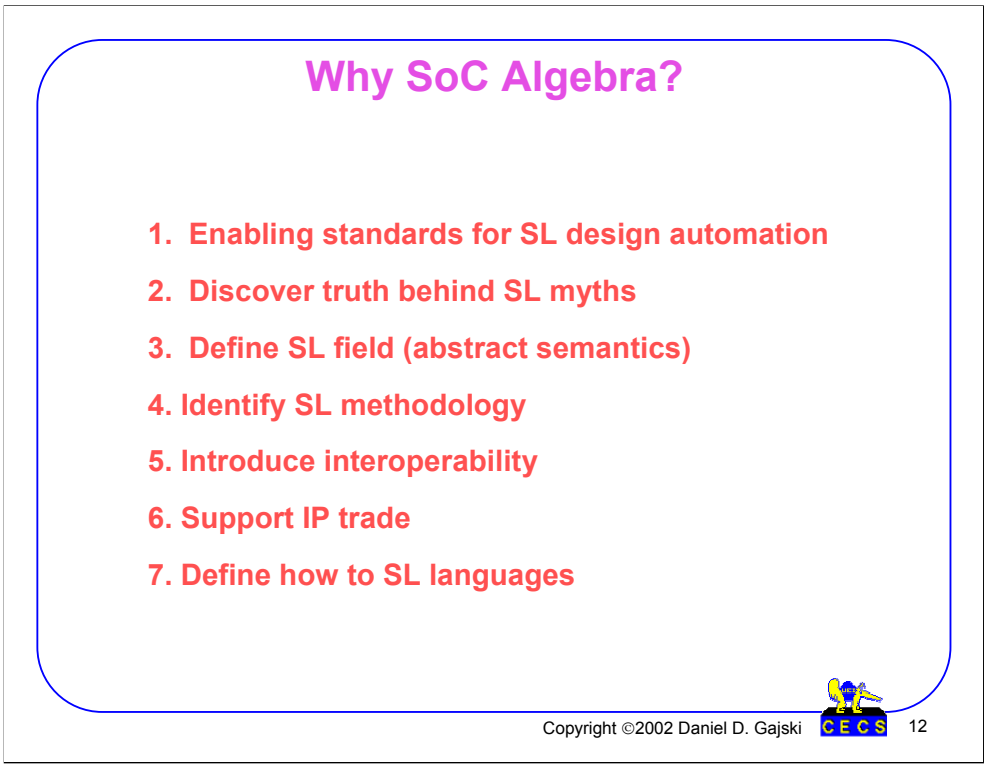
Y-Chart (continued)

Each abstraction level needs also a database of components on this level. Each component in the database has tree models representing three different axes in the Y-Chart: behavior or function (sometimes called Model of Computation (MoC)), structure of components from the lower level of abstraction and the physical layout or implementation of the structure. These components are IPs for each abstraction level.



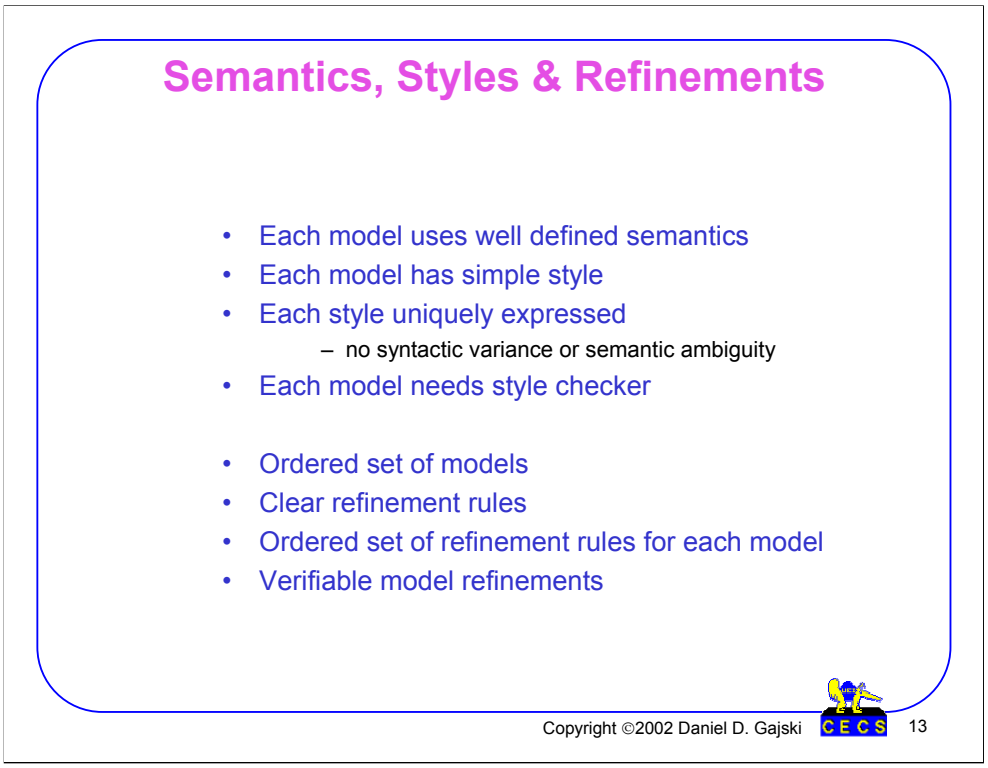
Design Formalism

In order to define a design methodology or a design flow we must define first a set of different models and a set of transformations that will generate one model from the other. This is similar to a abstract algebra that consists of a set of objects and a set of operations on those objects. Each object and operations may have certain properties. For example, we say that on operations is commutative if the order of objects for this operation is not important. Similarly to an abstract algebra, we can define SoC algebra which consists of a set of models (objects) and a set of transformations (operations) with the following property: for each model in the set we can find a ordered set of transformations that will generate another model in the set. If the models in the set are ordered by the complexity or the level of detail we say tha SoC algebra is ordered. More formally, for each model A we can find a ordered set of transformations < t_m, … , t_2, t_1 > such that the next (more detailed) model B can be derived by applying this set of transformations to A. In other words: $B = t_m(\ldots (t_2(t_1(A))) \ldots)$. Every ordered SoC algebra defines a design methodology or design flow.



Why SoC Algebra?

Proper definition of abstraction levels and models for design will enable standards in system-level (SL) design automation. It will also introduce some science into ad hoc methods used in approaches to create tools for SL simulation, synthesis and verification. It will also help define SL field and identify SL methodology. Formal definition of models (model semantics) will introduce interoperability and establishment of IP trading. It will also allow us to properly use SL languages such as SystemC, SpecC and others.



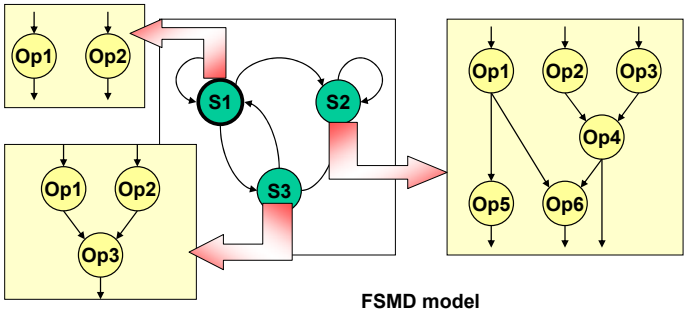
Semantics, Styles and Refinements

The main requirements for success of SoC algebra and SL design automation is formal definition of models and transformations. Thus, each model needs well defined semantics which can be expressed with a very simple syntax and modeling style which is free of syntactic variance or semantic ambiguity. Syntactic variance allows same meaning to be expressed (or described) syntactically in several different ways, which makes all synthesis or verification algorithms extremelly complicated. On the other hand, semantic ambiguity allows same syntax to have different meanings as shown previously in the case of missing semantics. Since each language can be use for describing different models and each model may require different style, we need a style checkers to help designers comply with the model style.

In order to automatize SoC design methodology we need ordered set of models and a ordered set of transformations or refinement rules for transforming higher-level model into lower-level ones. Also, the model transformations should be verifiable. We will explain these concepts for RTL and SL levels of abstraction.

RTL Computational Models

- **Finite State Machine with Data (FSMD)**
  - Combined model for control and computation
    - FSMD = FSM + DFG
  - Implementation: controller plus datapath

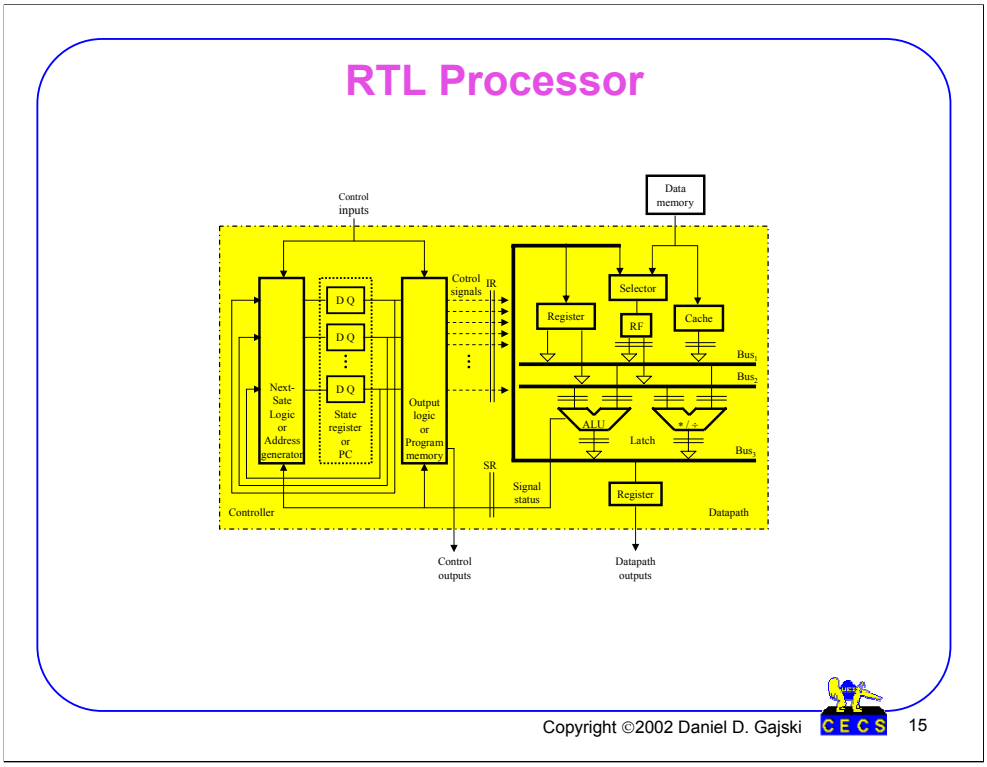


FSMD model

RTL Computational Model

The RTL behavior or computational model is given by a Finite-state-machine with Data (FSMD). It combines finite-state-machine (FSM) model for control and data-flow-graph (DFG) for computation. FSM has a set of states and a set of transitions from one state into other depending on the value of some of the input signals. In each state FSMD executes a set of expressions represented by a DFG. FSMD model is clock-accurate if each state takes a single clock-cycle.

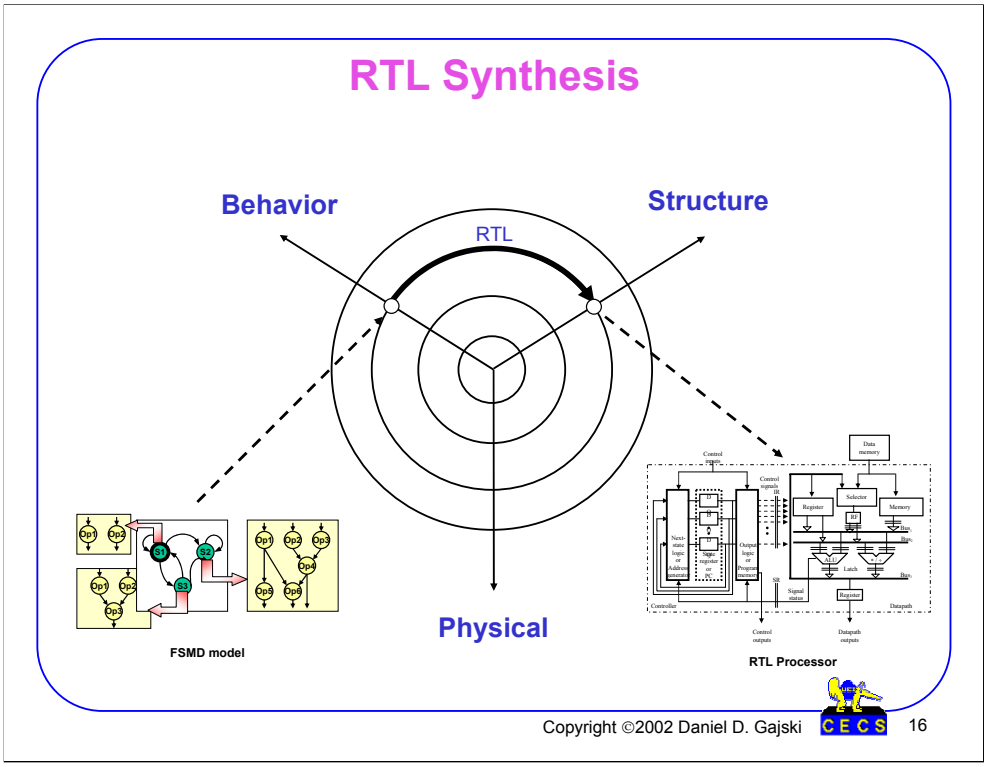
It should be noted that FSMD model encapsulates the definition of the state-based (Moore-type) FSM in which the output is stable during duration of each state. It also encapsulates the definition of the input-based (Mealy-type) FSM with the following interpretation: Input-based FSM transitions to a new state and outputs data conditionally on the value of some of FSM inputs. Similarly, FSMD executes set of expressions depending on the value of some FSMD inputs. However, if the inputs change just before the clock edge there may be not enough time to execute the expressions associated with that particular state. Therefore, designers should avoid this situation by making sure the input values change only early in the clock period or they must insert a state that waites for the input value change. In this case if the input changes too late in the clock cycle, FSMD will stay in the waiting state and proceed with a normal operation in the next clock cycle.



RTL Processor

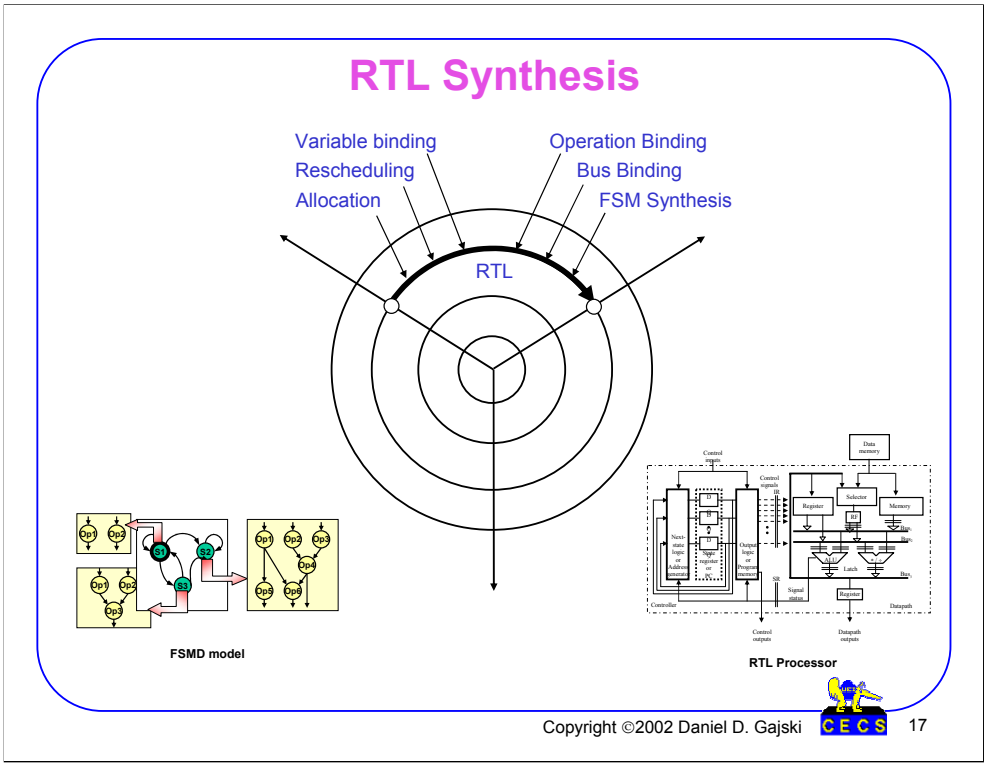
The behavior represented by a FSMD model can be implemented by the structure of a RTL processor, which consists of a Controller and the Datapath. Datapath consists of set of storage elements (registers, register files, memories), set of functional units (ALUs, multipliers, shifters, custom functions) and set of busses. All these RTL components may be allocated in different quantities and types and connected arbitrarily through busses. Each component may take one or more clock cycles to execute, each component may be pipelined and each component may have input or output latches or registers. The entire Datapath can be pipelined in several stages in addition to components being pipelined themselves. The Controller defines the state of the RTL processor and issues the control signals for the Datapath.

The RTL processor may represent an implementation of a standard processor (such as Pentium, PowerPC, or a DSP) or a custom processor or IP specifically synthesized for a particular function. In the former case, the Controller is programmable with a Program memory, PC and an Address generator. In the later case, the Controller is hardwired with a State register, Next-state logic and Output logic providing the control signals to the Datapath.



RTL Synthesis

RTL synthesis starts with the FSMD model in the behavior axes of the Y-chart and ends up with a custom RTL processor containing any number and type of components connected as required by the FSMD model. Note, that FSMD model can be obtained easily from a programming language code such as C by grouping all the consecutive statements into basic blocks (BB) and introducing two states for each **if** statement or **loop** statement , where each state executes a BB. Such a FSMD is sometimes called super-state FSMD since each BB may be considered to be executed in one super state. This generation is very simple. Note, that each BB will be partitioned into several states during RTL synthesis, where the number of states depends on the resources allocated to the RTL processor.



RTL Synthesis

RTL synthesis consists in converting a FSMD model into a RTL processor that generates the same result. It consists of several tasks;

(a) Allocation of components from the RTL library,

(b) Rescheduling of computation in each state since some components may need more than one clock cycle,

(c) Binding of variables, operations and register transfers to storage elements, functional units and busses,

(d) Synthesis of programmable or hardwired controller.

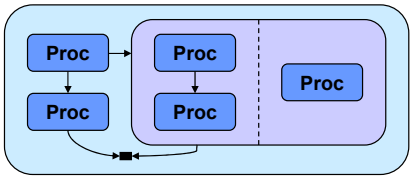
(e))Generation of refined model representing the RTL processor.

Any of the above tasks can be performed manually or automatically. If all of them are done automatically, we call the above process RTL synthesis. On the other hand, if (a) to (d) are performed by designer and only (e) is done automatically, we call the process model refinement. Obviously, many other strategies are possible as exemplified by available EDA tools that may perform each of the above tasks only partially in automatic fashion and leave the rest to the designer.

System Behavioral Model

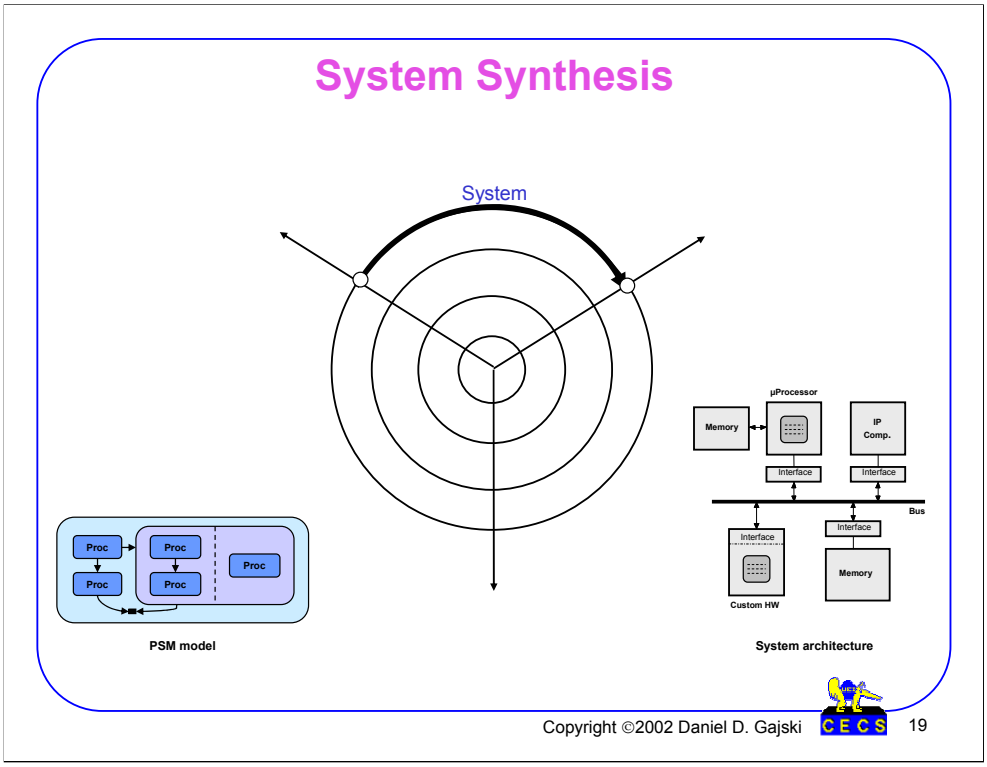
- **Program State Machine**
  - States described by procedures in a programming language
    - Example: SpecC! (SystemC!)

PSM model



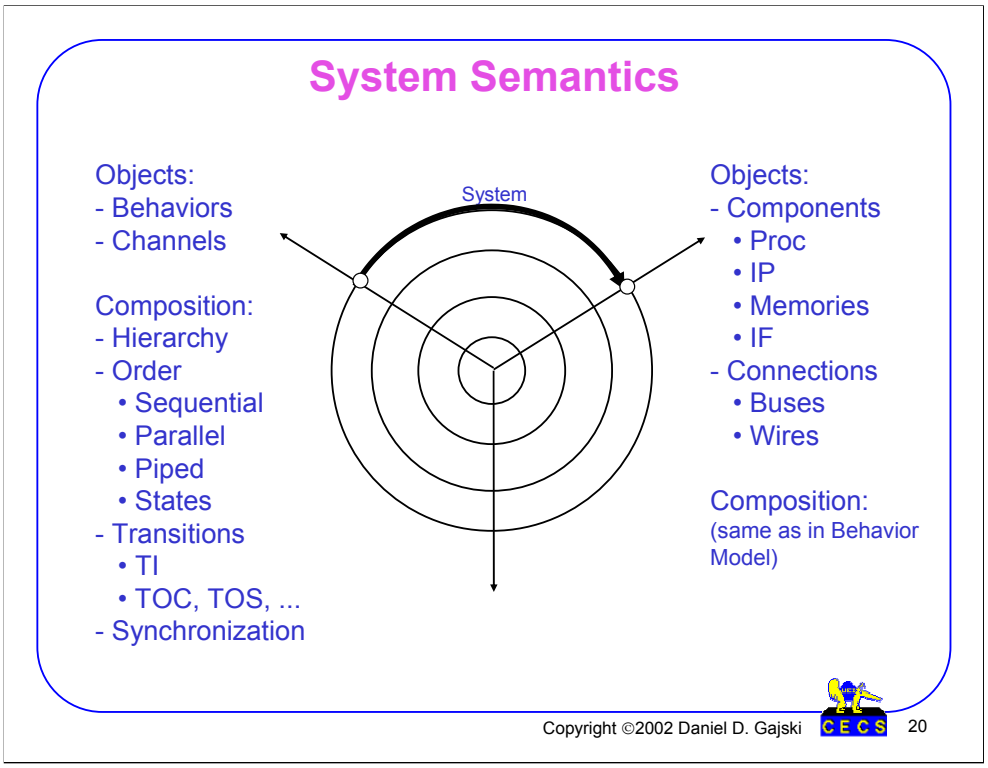
System Behavioral Model

Since systems consists of many communicating RTL processors the FSMD model will not suffice. Furthermore, such model must represent SW and HW. The easiest way is to retain the concept of states and transitions as in a FSM but to extend the computation in each state to include any procedure in a programming language such as C/C++. Furthermore, in order to represent an architecture with several processor working in parallel or in pipelined mode we must introduce concurrency (two states running in parallel) and pipelining (several states running in parallel with data passed between them sequentially). Since states are running concurrently we need a synchronization mechanism for data exchange. Furthermore, we need a concept of channel to encapsulate data communication. Also, we need to support hierarchy to allow humans to write easily complex systems specifications.



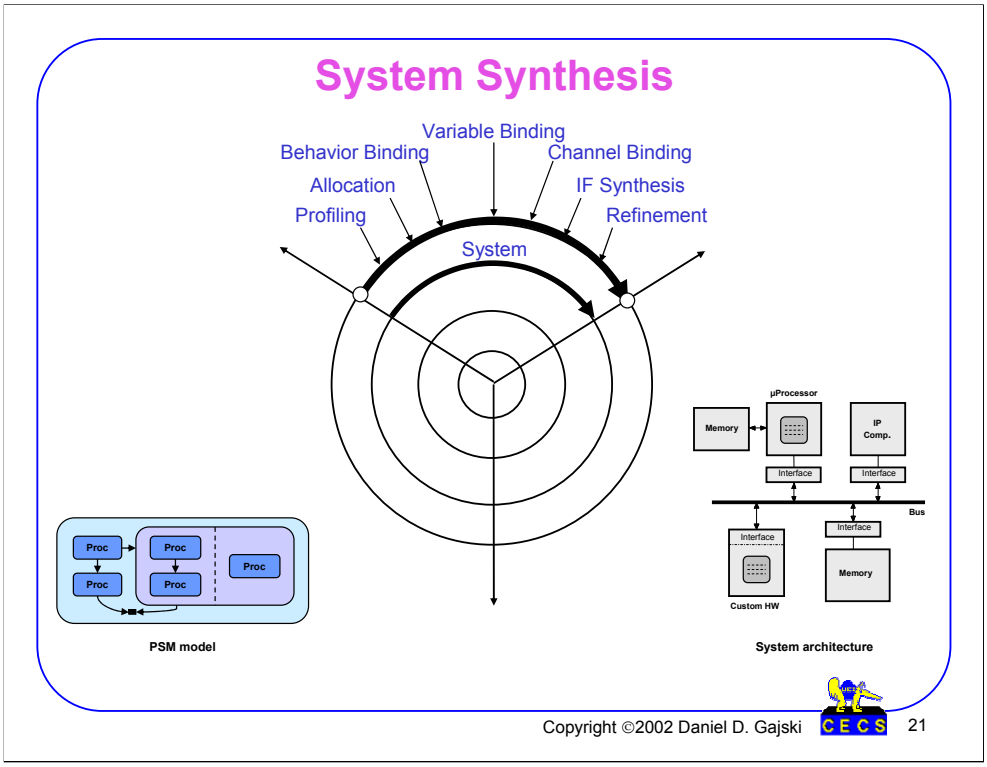
System Synthesis

System synthesis starts with a behavioral model of the system such as PSM model and generates the system architecture such as bus-functional model which describes components, their behavior and connectivity among components. Such model describes the operation of each component as a set of functions or procedures with lumped time assigned to each function. The communication is described with channels including time accurate protocols.



System Semantics

In system semantics we must transform a behavioral model into a structural model. Following the SoC algebra the behavioral model is a composition of two objects: behaviors and channels. They can be composed hierarchically and ordered through sequential, parallel pipelined and state operators. The transition from state to state may be accomplished on interrupt, completion, or specific variable value. For concurrent processing we can use some type of synchronization such as waiting for event generated by another process. The structural model uses different object: behaviors are replaced by components (processors, IPs, etc.) and channels are replaced by buses or wires with well defined protocols. The composition rules stay the same.



System Synthesis

PSM model can be synthesized into a arbitrary architecture by the following set of tasks:

(a) Profiling of code in each behavior and collecting statistics about computation, communication, storage,traffic, power consumption, etc.,

(b) Allocating components from the library of processors , memories , IPs and custom RTL processors,

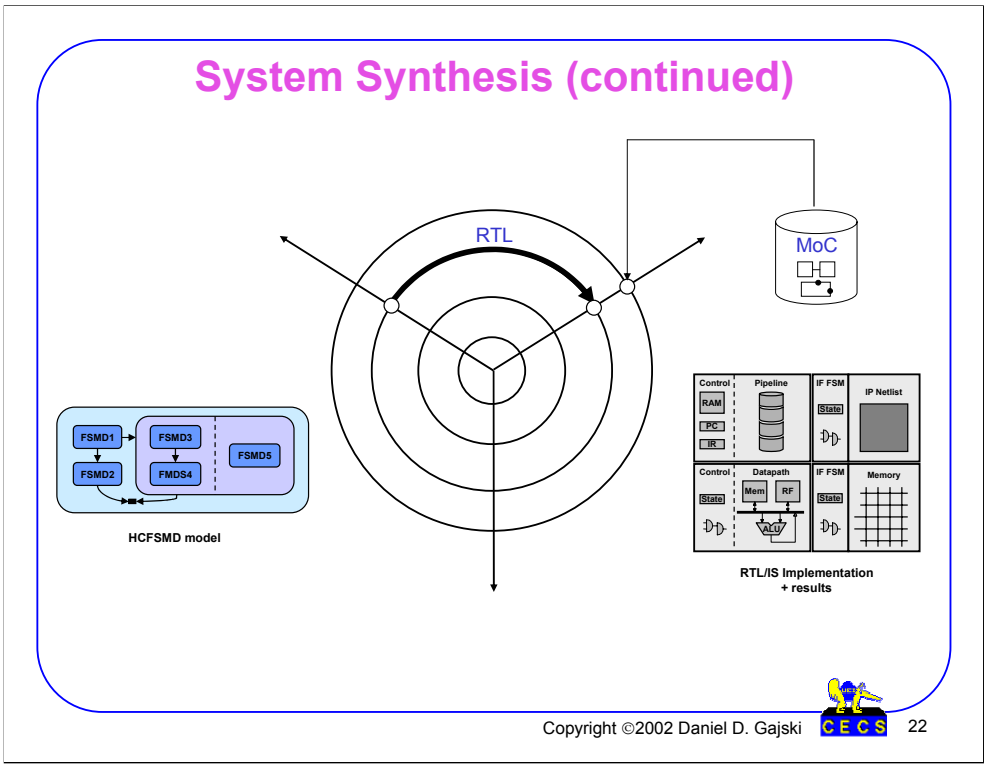
(c) Binding behaviors to processing elements, variables to storage elements (local and global), and channels to busses,

(d) Synthesizing IF between components and busses with incompatible protocols,

(e) Refining the PSM model into a architecture model that reflect allocation and binding decisions.

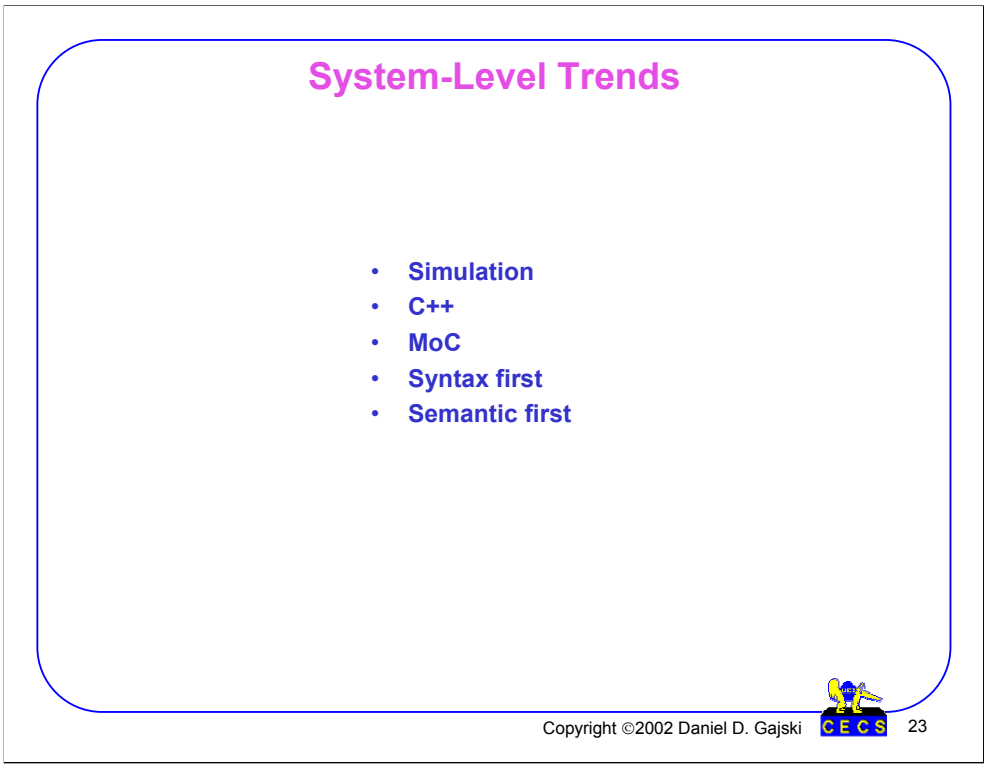
The above tasks can be performed automatically or manually. Tasks (b)-(d) are usually performed by designers while tasks (a) and (e) are better done automatically since they require lots of mundane effort.

Once the refinement is performed the architecture model can be validated by simulation quite efficiently since all the component behaviors are described by high level functions.



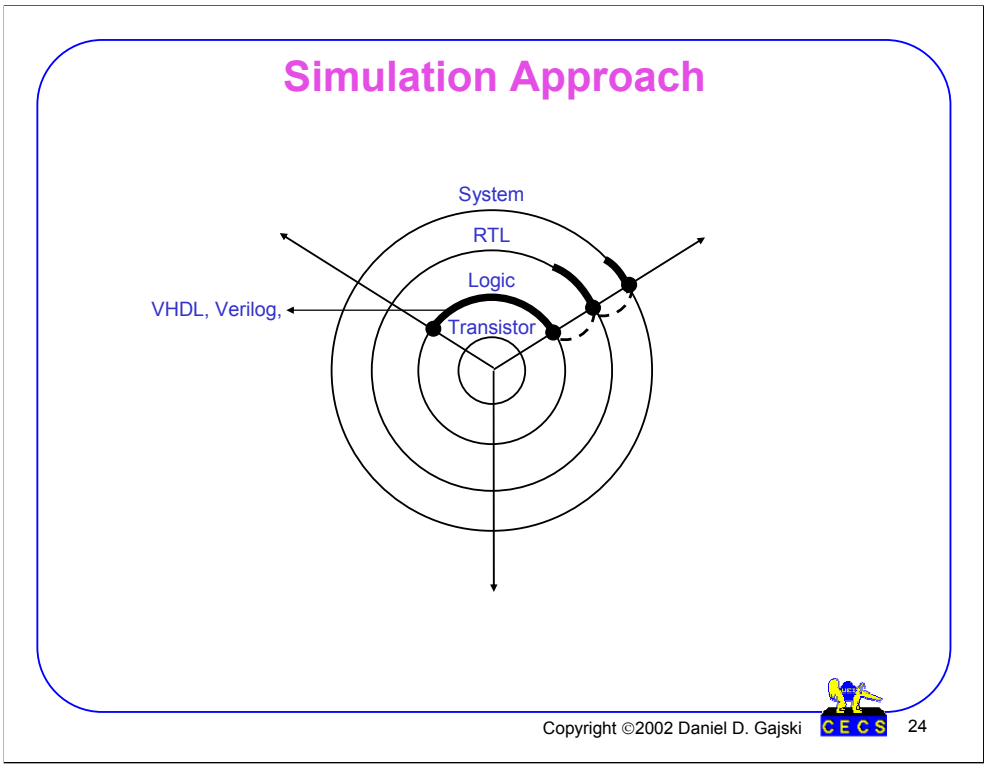
System Synthesis (continued)

In order to generate cycle –accurate model, we must replace each component functional model with a FSMD model for custom HW or IS model for standard processors executing SW. Once we have bus-functional model, we can refine it further to cycle-accurate model by performing RTL synthesis for custom RTL processors or custom Ifs and compiling behaviors assigned to standard processors to instruction-set level and inserting IS simulator to execute the compiled instruction stream. RTL synthesis can start from super-state FSMD that is obtained through two different mechanism. On one hand, we can replace a behavior assigned to an IP with a super-state FSMD model from IP library. On the other hand, we can perform RTL synthesis on any behavior assigned to RTL processor after refining the behavior to BB super-state FSMD. After RTL/IS refinement we end up with cycle-accurate model of the entire system.



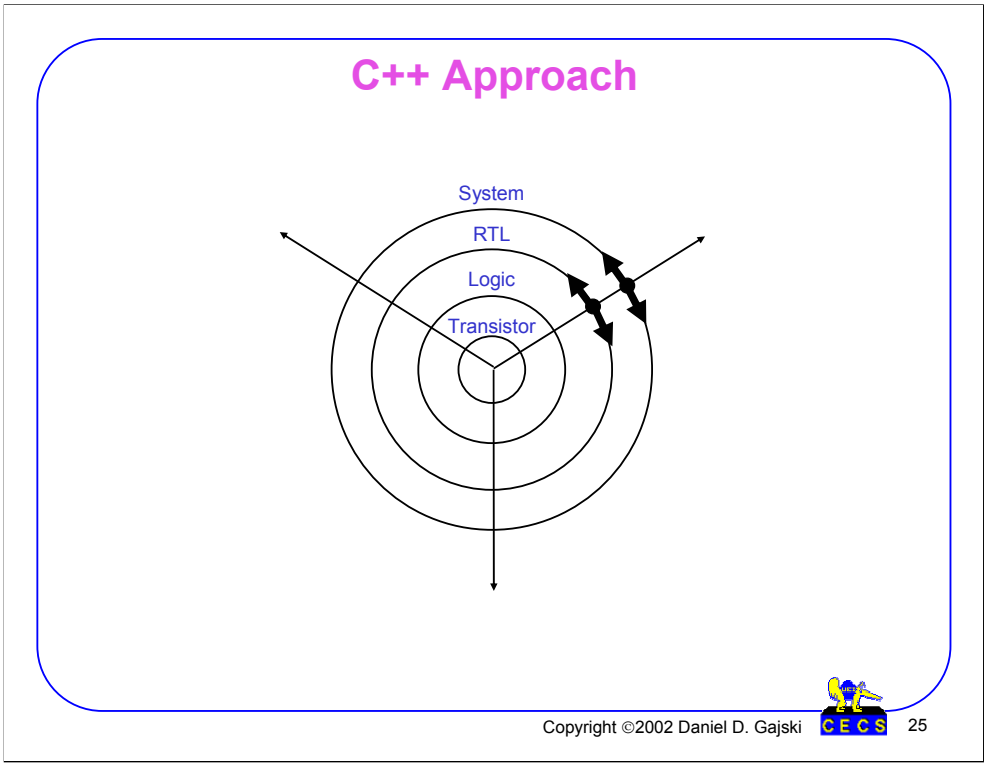
System-Level Trends

System-level abstraction arrived suddenly as a consequence of large increases in chip complexity. Since the methodology and tools are not available yet. Many research groups are experimenting with different approaches to solve the complexity issues. We can identify several more popular approaches. Simulation is the most popular among EDA community. Some people are trying to extend C++ for their particular needs, while others are developing new MoCs for particular applications. Others are trying to develop multi purpose system-level languages. Two trends can be identified in this group depending whether syntax or semantics is developed first. We will look at each trend in detail in the rest of this report.



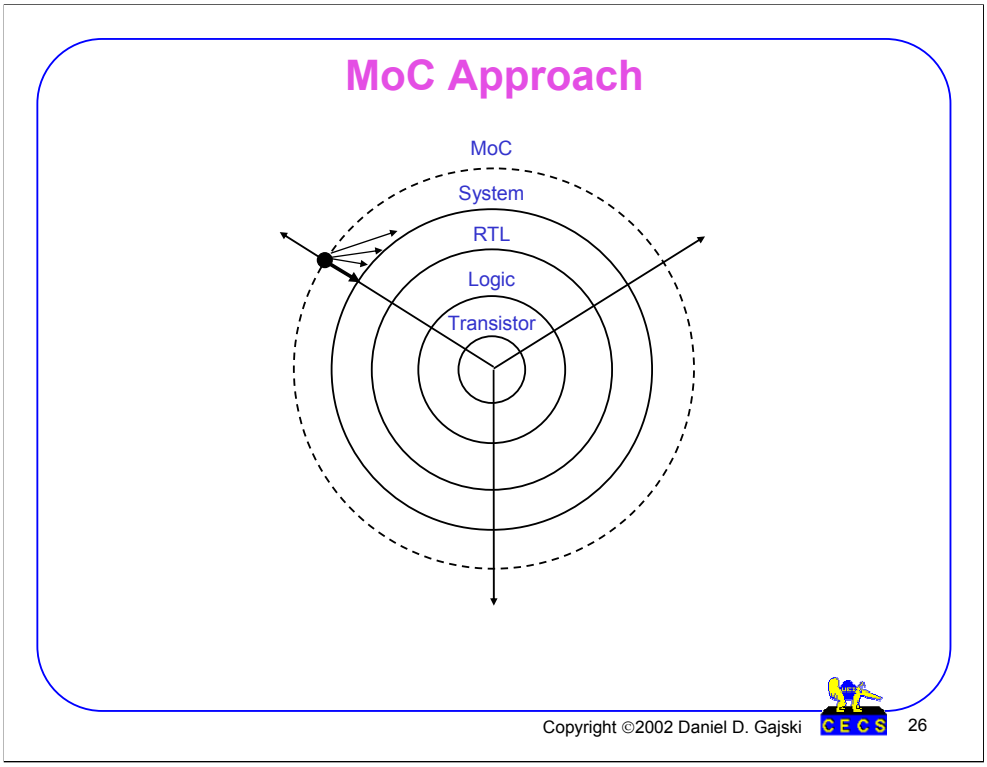
Simulation Approach

Simulation approach focuses design methodology on simulation. Designers develop design on different levels and simulate its behavior by writing simulation models. Formal verification or equivalence checking is almost impossible since simulation models are ambiguous and not strict enough for synthesis and verification. Usually, simulation model must be restricted to a language subset or a particular style to be synchronizable and even more restricted to be verifiable. This simulation approach reverse the trend established by logic synthesis where design methodology focuses on synthesis whilecreating simulatable models is automatic consequence of synthesis approach.



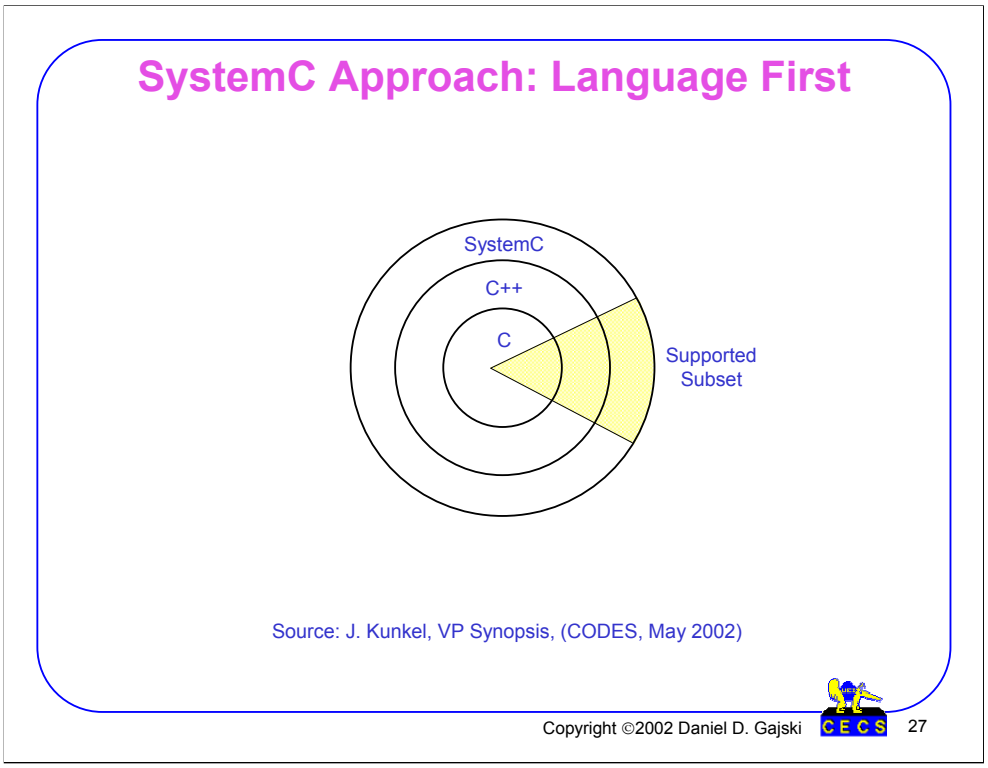
C++ Approach

C++ is an extensible language. Providing a standard set of C++ classes and a free-source simulator is very appealing to all faculty ad their students as well as to large number of industrial researchers and managers since they can add some classes and adapt the standard set to their particular application, methodology and modeling style. However, these additional classes make their methodology and models useless for others. Therefore, C++ is good for experimentation but is improbable to become a standard modeling language.



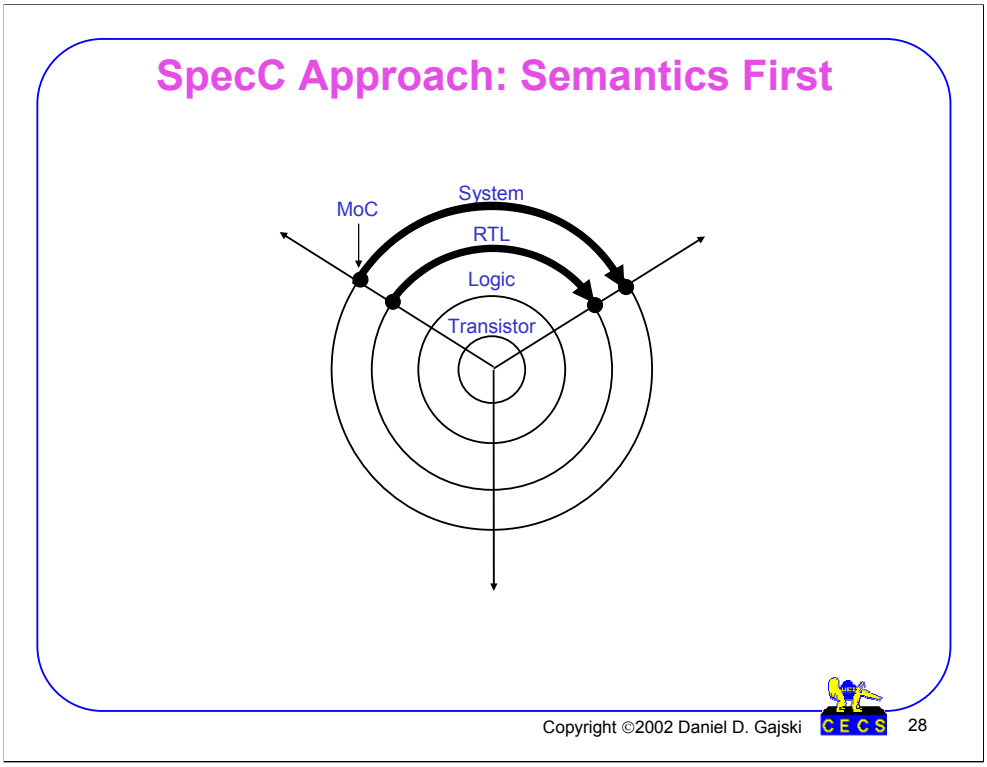
MoC Approach

Developing new models of computation (MoCs) that are tuned to different applications simplifies the specification captive for a particular application. However, this approach leaves a huge gap between MoC and working SW and HW on a SoC. This gap is difficult to bridge with today's algorithms and methods for synthesis, verification and test in order to generate an efficient design.



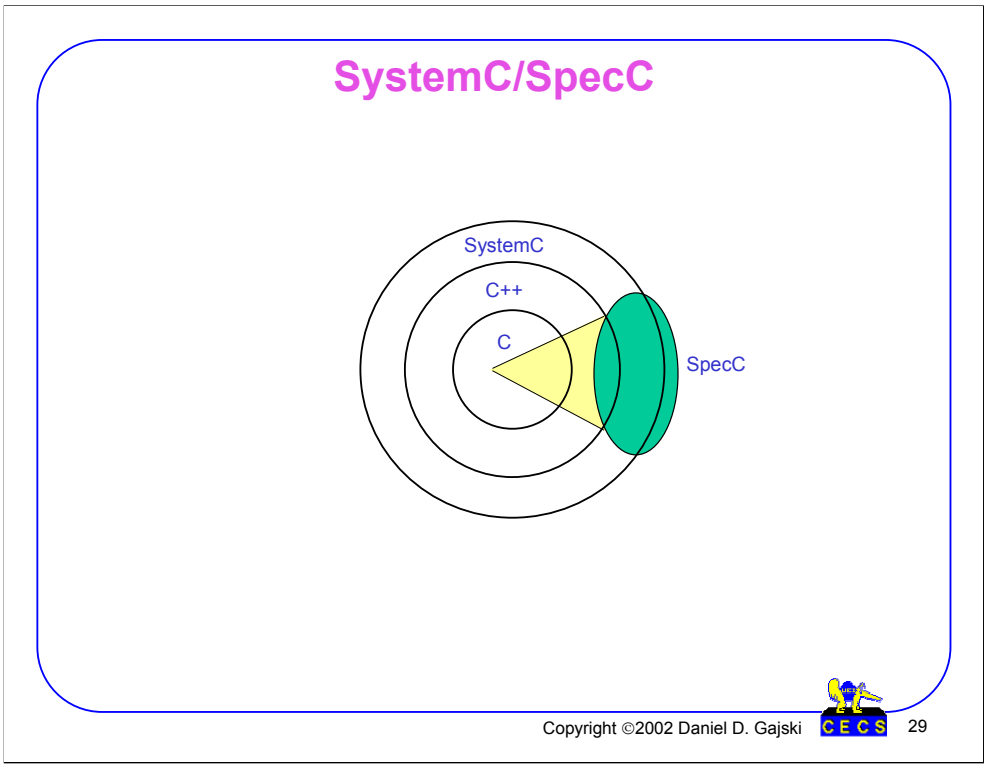
SystemC Approach: Language First

There are two approaches in developing system methodology. One approach is to develop a language first and then experiment with it to find how to use it in different applications. Such a language is accompanied with a simulator to support simulation of the models written in the language. SystemC is such a language based on C++. It is appealing to many researchers who can experiment with it by adding new classes that they optimize for their own application. However this extensibility does not help. On the contrary, it creates incompatibility, since everyone has his or her own classes that are not compatible with anyone else. At this moment, SystemC already has too many classes that are not easy too support. That will eventually result in a subset for synthesis and another subset for verification. It is difficult to predict when these subsets will emerge, how they will look like and who will define them. Similar situation occurred with other simulation- oriented languages such as VHDL and Verilog.



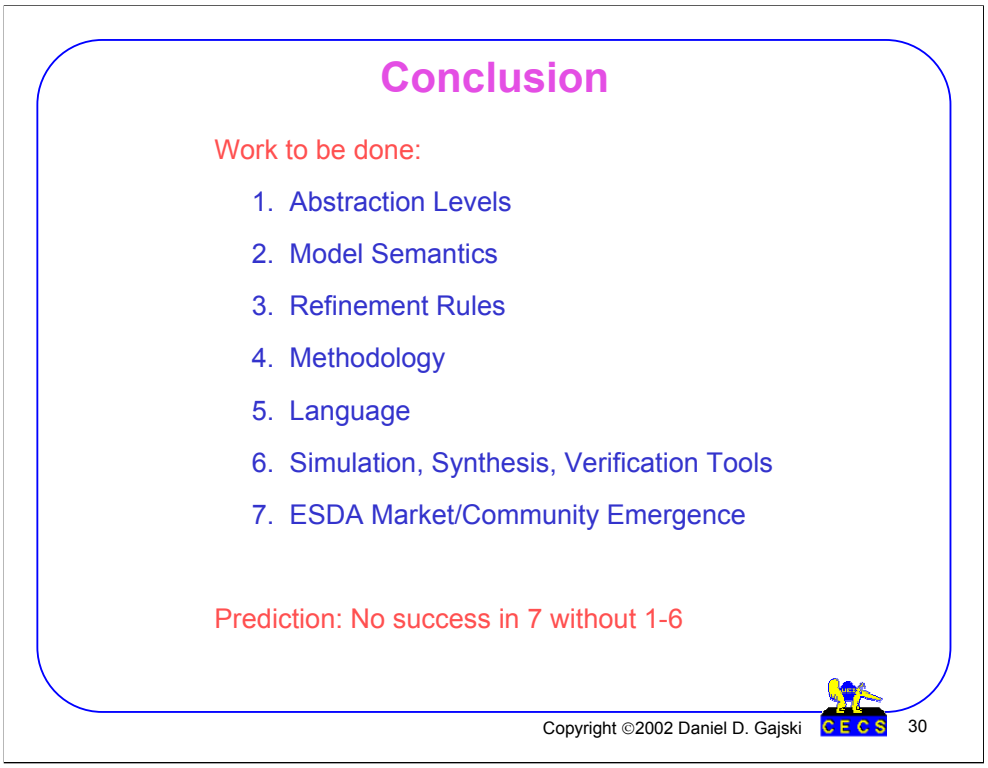
SpecC Approach: Semantics First

The other approach to language definition is to define abstraction levels and define behavioral and structural models and transformation rules for deriving one model from the other. This approach is more limited since there is less flexibility later but it results into interoperability of models and tools. With well defined semantics the designer education is easier and IP trade can develop sooner. Since SpecC was designed to support minimal and orthogonal set of concepts for modeling SW and HW it is easy to use by designers as well as tool makers and it does not need subsetting since the SpecC models are synthetizable and verifiable. The SpecC version 2.0 supports automatic refinement of a PSM model into architecture/bus-functional model, into FSMD model, and finally into cycle-accurate RTL model ready for synthesis with available standard EDA tools.



SystemC/SpecC Standing

In the moment, SystemC looks like a popular simulation language that is looking for synthesis and verification subsets. On the other, SpecC is more restricted, synthesizable and verifiable. Therefore, the obvious conclusion is that SpecC may become synthesizable and verifiable subset of SystemC, since it compiles to C++ for simulation anyhow. This outcome may become a real possibility if both group came together and smooth some minor differences in the syntax and semantics of both languages.



Conclusion

From the above discussion it is obvious that old strategy of developing a language and subsetting it for different design task is not acceptable for SL design. The strategy for success is to solve the following issues:

(1) Define the abstraction levels for SoC design flow,

(2) Define semantic for each model without synthetic variance or semantic ambiguity,

(3) Define refinement rules for deriving a refined model from a more abstract one,

(4) Define the methodology including models, tasks and necessary tools,

(5) Design the language to support the above models, refinements and methodology,

(6) Develop tools for simulation, synthesis and verification of different models,

(7) Stimulate the organization of SL community, and ESDA market.

It is obvious that issues 1-6 must be resolved before a community or market will emerge.

SL academic community has made sizable progress toward this goal in the past, but more work is needed in the future before we can claim that SL design flow is understood.