System Debugging and Verification : A New Challenge

Daniel Gajski Samar Abdi
Center for Embedded Computer Systems
http://www.cecs.uci.edu
{gajski,sabdi}@cecs.uci.edu

Technical Report CECS-03-31
October 1, 2003
UC Irvine

Verify 2003

Abstract :

The continuous increase in size and complexity of System-on-Chip designs has introduced new modeling and verification challenges. The system designs of today need modeling at higher levels of abstraction such as transaction level. On the verification front, techniques like assertion based verification are being used to complement traditional simulation and debugging of designs. Formal methods like logical equivalence checking are becoming increasingly relevant for minimizing or even eliminating the need for costly gate-level simulations. Property checking techniques like model checking and theorem proving are being employed in high-end processor and system design.

In this talk, we will present an overview on the role of modeling and verification in the complete design flow from system level to gates. We will discuss different models and the verification techniques that apply best for validating them.

Overview

- Simulation and debugging methods
- Formal verification methods
- Comparative analysis of verification techniques
- Model formalization for SoC verification
- Conclusions

The presentation will cover various techniques in verification of systems, ranging from simulation based methods to more formal static methods. A comparison of the techniques is given based on metrics like cost, applicability and coverage. We then discuss the challenge of verifying large systems with traditional techniques and present possible directions for a solution to verifying complete systems. Our approach is based on well defined model semantics and the formalization of model construction. We show how this approach can help establish a methodology for verification of systems.

Design Verification Methods

- **Simulation based methods**
  - Specify input test vector, output test vector pair
  - Run simulation and compare output against expected output
- **Semi-formal Methods**
  - Specify inputs and outputs as symbolic expressions
  - Check simulation output against expected expression
- **Formal Methods**
  - Check equivalence of design models or parts of models
  - Check specified properties on models

Verify 2003

The verification methods available today can be broadly classified into three categories, namely simulation based, semi-formal and formal methods.

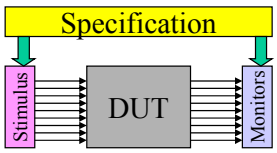
In simulation based methods, the designer writes an executable model of the design. Test vectors are applied to the inputs of the model and output values are generated after logical delays as specified in the model. The functionality of the model is tested by comparing the generated outputs to the expected outputs.

Semi-formal methods primarily use a simulation environment, but apply symbolic methods for stimulating and monitoring the design. The gain is in the reduction of test cases, however monitoring simulation results becomes more complicated. This is because the monitor has to compare generated output expressions against expected output expressions, which may be syntactically different yet evaluate to the same value.

Pure formal methods do not need a simulation environment. The models and properties are expressed in a mathematical form and mathematical formulations are used to either compare two models or check if a property holds in a model.

Simulation

- **Task : Create test vectors and simulate model**
- **Inputs**
  - Specification
    - Typically natural language, incomplete and informal
    - Used to create interesting stimuli and monitors
  - Model of DUT
    - Typically written in HDL or C or both
- **Output**
  - Failed test vectors
    - Pointed out in different design representations by debugging tools



Typical simulation environment

Simulation is the most widely used method for validation of models. The design to be tested is described in some modeling language and is referred to as design under test (DUT). The design specification is then used to generate input and out test vectors. The stimulus routine applies the input vectors to the models. The inputs are propagated through the model by the simulation tool and finally the outputs are generated. A monitor routine checks the output of the DUT against expected outputs for each input test vector. If a mismatch is found, the designer can use debugging tools to trace back and find the source of the problem. The problem arises from either incorrect design or incorrect timing. Once the problem source is identified, the designer can fix it and simulate the new model.

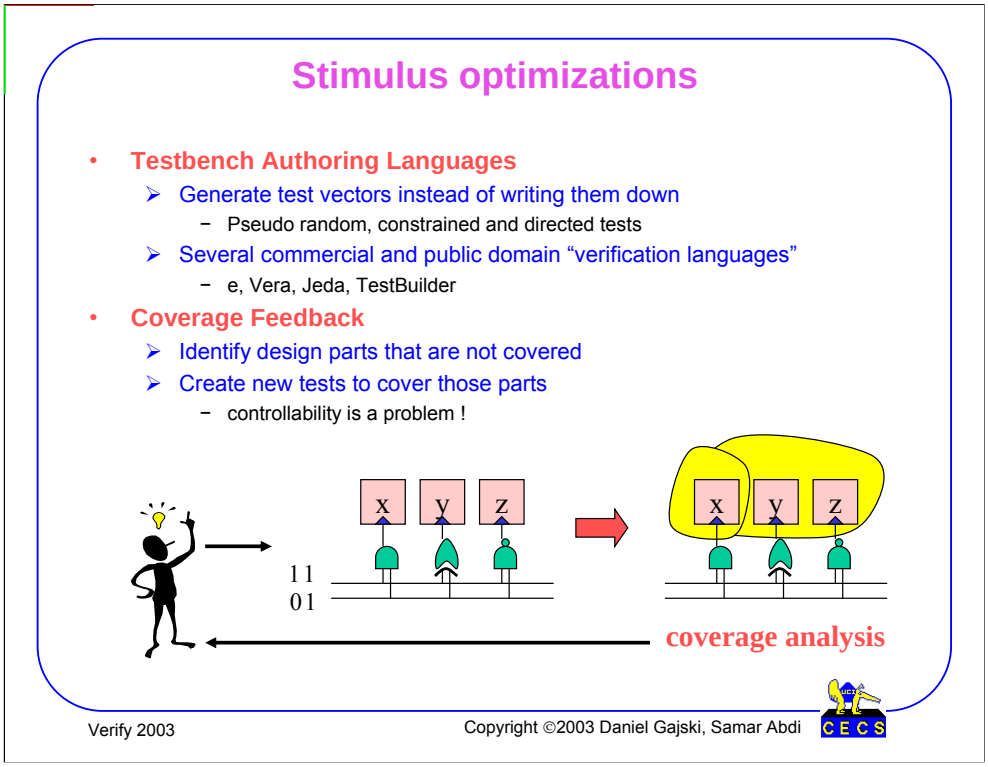
Improvements to Simulation Environment

- **Main drawback is coverage**
  - Several coverage metrics
    - HDL statements, conditional branches, signal toggle, FSM states
  - Each metric is incomplete by itself
  - Exhaustive simulation for each coverage type is impractical
- **Possible Improvements**
  - Stimulus optimizations
    - Language to specify tests concisely vs. exhaustive enumeration
    - Write tests for uncovered parts of the model
  - Monitor optimizations
    - Assertions within design to point to simulation failures
    - Better debugging aids (correlation of code, waveforms and netlist)
  - Speedup techniques
    - Cycle simulation vs. event driven
    - Hardware prototyping on FPGA
  - Modeling techniques
    - Models at higher abstraction level simulate faster

Verify 2003

The intent of the designer is to test the model for all possible scenarios. However, this would require unreasonable number of test vectors. Since only a limited number of test vectors will be used, the designer must try to choose the most useful ones. The usefulness of a test case is usually defined by the number of components and connections it can cover. Moreover, a test case that verifies an already tested part of the design does not add any value. Therefore, several coverage metrics have been invented to quantify the usefulness of a test case.

The simulation performance can be improved either by speeding up the simulator or by choosing test cases intelligently to maximize coverage with minimal simulation runs. One optimization is to reduce test generation time by giving constraints to stimuli and testing with only valid inputs. Monitoring non-primary output variables in the model reduces debug time by pointing out the error closer to its source. Testing the model by implementing it on hardware provides much faster functional testing. Finally, rewriting the model by abstracting away low level details also reduces simulation time, thereby finding errors earlier.

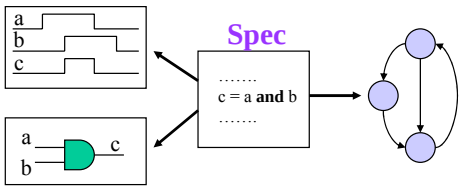


Writing down test vectors for simulation can be a painful task. Also, generating test vectors randomly might result in a lot of invalid vectors. Since the model is typically constrained to work for only select scenarios, we can use this knowledge to generate valid test vectors only. The test scenario can thus be written in some language and a tool can be used to generate valid test vectors for that scenario.

Using the results from coverage is another way to minimize the number of test vectors. For instance, the code coverage feedback technique can be visualized in the given figure. A simulation run with vector "11" results in only block "x" being covered. The designer looks at the coverage result and comes up with a vector "10" to cover blocks "y" and "z". Note that vector "00" would not cover block "y" and is thus not used. Such a feedback strategy can be used with other coverage metrics as well.

Monitor optimizations

- **Assertions in the model**
  - Properties written as assertions in design
    - Example : signals **a** and **b** are never '**1**' at the same time
    - Errors detected before reaching primary output (helps debugging)
  - Several methods of inserting assertions
    - Assertion languages, e.g. PSL, SystemVerilog, e
      - assert always !(a & b)
    - Pragmas
- **Debugging aids**
  - Correlation between different design representations
    - Waveforms, schematic, code, state machines



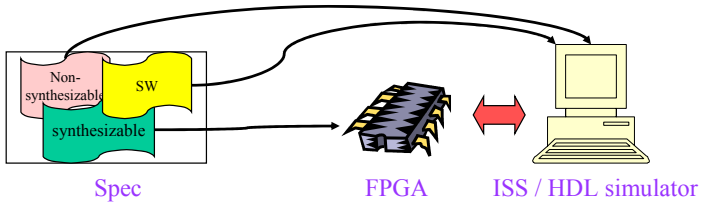
Verify 2003

Monitoring only the primary outputs of a design during simulation lets us know if a bug exists. Tracing the bug to its source can be difficult for a complex designs. If the source code of the model is available, assertions can be placed on internal variables or signals in the model. For example, we can specify that the two complementary outputs of a flip-flop never evaluate to the same value. Not only does this improve understanding of the design, it also points out the bug much closer to the source. Assertions can also be used to check validity of properties over time, like protocol compliance. It must be ensured that the assertions do not get synthesized along with the design. Therefore, they must be written either in a different language than the design, or as special comments that can be ignored by the synthesis tool.

Graphical visualization of the structure and behavior of a design also helps debugging. Specifically, correlation between different representations, such as waveforms, net lists, state machines and code, allows the designer to easily identify the bug in a graphical representations and locate the source code for buggy part of the model.

Speedup techniques

- **Cycle simulation**
  - Observe signals once per clock cycle
  - Cannot observe glitches within a clock cycle
- **Emulation**
  - Prototype hardware model on FPGAs
  - Much faster than software simulation
  - In-circuit emulation
    - FPGA is inserted on board instead of real component
  - Simulation acceleration
    - Emulate parts of hardware by interfacing with software simulator



Overall simulation time can be reduced by simply increasing the simulation speed. The two common speedup techniques are cycle simulation and emulation. Cycle simulation is used when we are only concerned about the signals at clock boundaries. This allows improving the simulation algorithm to update signal values at clock boundaries only. On the other hand event driven simulation needs to keep track of all events, even between the clock edges, and is thus much slower.

Another speedup technique is the use of reconfigurable hardware to implement the DUT. If the designer wants to simulate a component in a larger available system, the FPGA implementation can be hardwired in the system. This technique is called in-circuit emulation. A different scenario in which emulation is used is dubbed software acceleration. The synthesizable part of the hardware is implemented on an FPGA. The SW and the unsynthesizable HW runs on a software simulator, which talks to the emulation tool via remote procedure calls.

Modeling techniques

- **Use higher levels of abstraction for faster simulation**
  - Untimed functional / Specification model
    - Executable specification to check functional correctness
    - Simulates at the speed of C program execution but no timing
  - Timed architecture model
    - Used to evaluate HW/SW partitioning
    - Computation distributed onto system components
  - Transaction level model
    - Used to evaluate system with abstract communication
    - Transactions vs. bit toggling (data abstraction)
  - Bus functional model
    - Communication modeled at pin-accurate / time accurate level
    - Computation modeled at functional level
  - Cycle accurate model
    - HW and SW at cycle accurate level
    - Communication at cycle accurate level

Verify 2003

A different approach to reduce functional verification time is by modeling the system at higher abstraction levels. By abstracting away unnecessary implementation details, the model not only becomes more understandable, but also simulates faster. For instance, models with bus transactions at word level simulate faster than those at bit level because the simulator does not have to keep track of bit-toggling on bus wires. Similarly, models with coarse timing result in fewer events during simulation. There are several abstract models that can be used depending on the size and nature of the design as well as the design methodology.

Formal Verification Methods

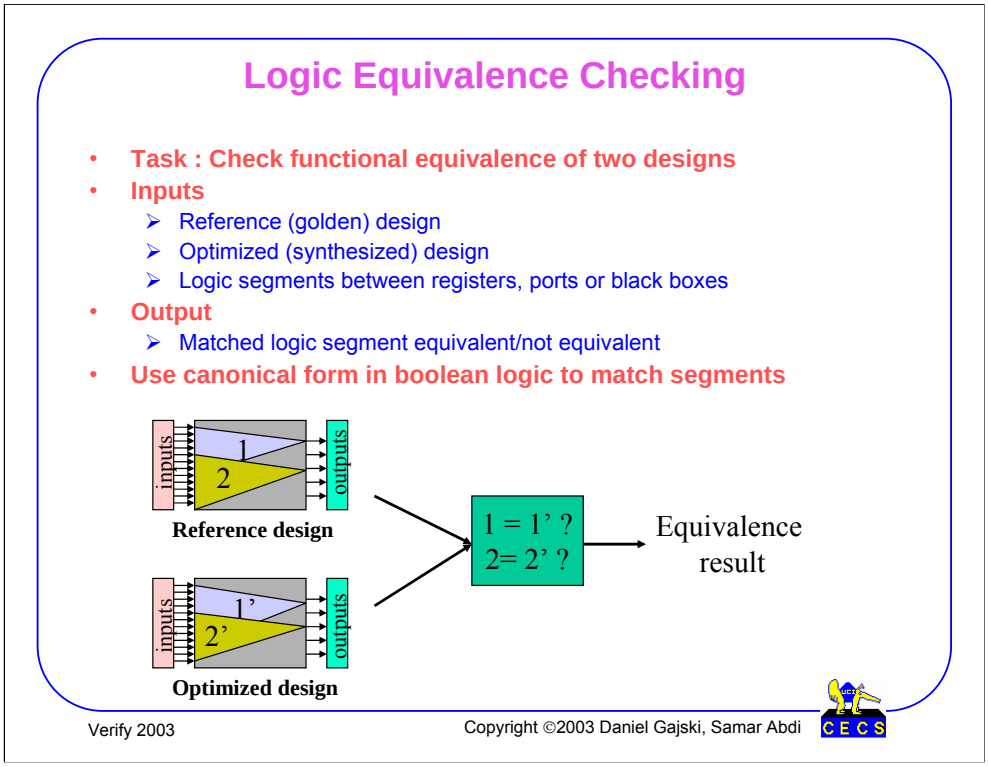
- **Equivalence Checking**
  - Compare optimized/synthesized model against original model
- **Model Checking**
  - Check if a model satisfies a given property
- **Theorem Proving**
  - Prove implementation is equivalent to specification in some formalism

Verify 2003

Formal verification techniques use mathematical formulations to verify designs. In order to check for correctness of synthesis and optimization of models, we can use equivalence checking. We define some notion of equivalence like logic equivalence or state machine equivalence and the equivalence checker proves or disproves the equivalence of original and optimized/synthesized models.

Model checking, on the other hand, takes a formal representation of both the model and a given property, and checks if the property is satisfied by the model. Assertions that have been used for simulation can also be used as properties for model checking.

Theorem proving takes formal representations of both the specification and implementation in a mathematical logic and proves their equivalence.



During synthesis or optimization of logic circuits, the design is optimized to reduce the number of gates or circuit delay. The designer is responsible for the logical correctness of any such transformation. A logic equivalence checker checks that the result of the synthesis or optimization is equivalent to the original design. This is achieved by dividing the model into logic cones between registers, latches or black-boxes. The corresponding logic cones are then compared between original and optimized models.

Logic cones can be described with boolean expressions and thus represented as boolean decision diagrams (BDDs). Since BDDs have a canonical form, we can reduce the original and optimized cones to their respective canonical forms and check if they are the same. This technique is possible at the GATE/RTL level because of the available formalism for boolean algebra.

FSM Equivalence Checking (1/2)

- **Finite State Machine**
  - M : < I, O, Q, Q0, F, H >
    - I is the set of inputs
    - O is the set of outputs
    - Q is the set of states
    - Q0 is the set of initial states
    - F is the state transition function Q × I → Q
    - H is the output function Q → O
- **FSM as a language acceptor**
  - Define Qf to be the set of final states
  - M *accepts* string S of symbols in I if
    - applying symbols of S to a state in Q0 leads to a state in Qf
  - Set of strings accepted by M is its language
- **Product FSM**
  - Define product FSM as a parallel composition of two machines
    - M1: < I, O1, Q1, Q01, F1, H1 > , M2: < I, O2, Q2, Q02, F2, H2 >
    - M1×M2 : <I, O1×O2, Q1×Q2, Q01×Q02, F1× F2, H1×H2 >

Verify 2003

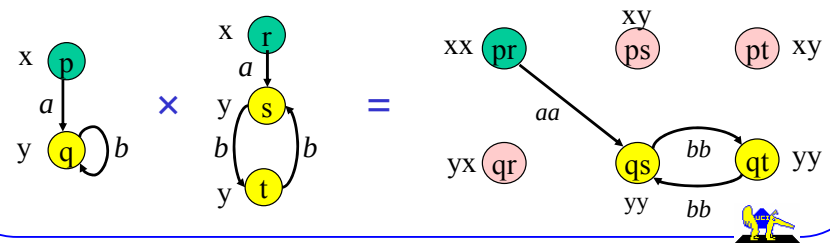
Logic equivalence checker checks only the equivalence of the combinational part of the circuit. There are also techniques to check equivalence of the sequential part of the design. In order to understand those techniques, we have to define the notion of a finite state machine. A finite state machine (FSM) is a tuple consisting of a set of inputs, a set of outputs and a set of states. Some of the states are designated as initial states and some as final states. Transitions between states are defined as a function of current state and the input. An output is also associated with every state.

We can think of a FSM as a language acceptor. If we start from an initial state, supply input symbols from a string S and reach a final state, then S is said to be accepted by the FSM. The set of all acceptable strings forms the language of the FSM.

We also define the notion of a product FSM. The product of two finite state machines M1 and M2 has the same behavior as if M1 and M2 were running in parallel.

FSM Equivalence Checking (2/2)

- **Task : Check if implementation is equivalent to spec**
- **Inputs**
  - FSM for specification (Ms)
  - FSM for implementation (Mi)
- **Output**
  - Do Mi and Ms give same outputs for same inputs ?
- **Idea (Devadas, Ma, Newton '87)**
  - Compute Mi×Ms
  - Qf(Mi×Ms) = States which have different outputs for Mi and Ms
  - Check if any state in Mi×Ms is reachable (language emptiness)



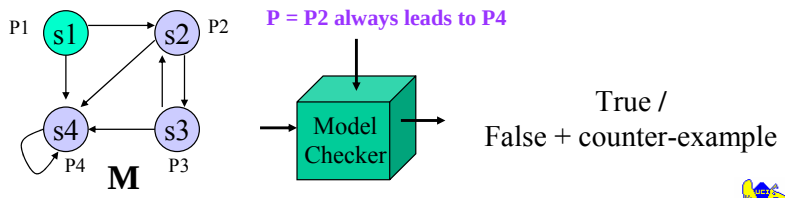
We can now define equivalence of FSM models by using the previously discussed notions and concepts. The specification and its implementation are both represented as FSMs Ms and Mi respectively. It must be ensured that the input and output alphabet of the two machines should be the same.

We derive the product machine Ms x Mi. Now all the states in Ms x Mi that have pair of differing outputs are labeled as final states. In the given figure, the states ps, pt and qr have output pairs with non-identical symbols (xy or yx) and are thus labeled as final states. We also keep only those transitions that have the same symbols in the input pair. What we are trying to prove is that for the same sequence of inputs, Ms and Mi would produce the same sequence of outputs. In other words, any state with a pair of non-identical outputs should never be reached. Since such states are the final states in the product FSM, they should never be reached. Therefore the product FSM should not accept any language. This notion is called language emptiness.

Showing language emptiness means starting from the set of initial states in Ms x Mi and performing a reachability analysis. If any of the final states is reachable, then the specification and implementation are not equivalent.

Model Checking (1/2)

- **Task : Property P must be satisfied by model M**
- **Inputs**
  - Transition system representation of M
    - States, transitions, labels representing atomic properties on states
  - Temporal property
    - Expected values of variables over time
    - Causal relationship between variables
- **Output**
  - True (property holds)
  - False + counter-example (property does not hold)
    - Provides test case for debugging



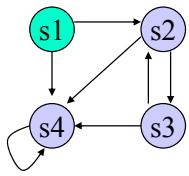
Model checking is a formal technique for property verification. The model is represented as a state transition system, which consists of a finite set of states, transitions between states and labels on each state. The state labels are atomic properties that hold true in that state. These atomic properties are expressed as a boolean expression of the state variables in the model. The property to be verified on the model is expressed as a temporal formula. The temporal formula is formed using state variables and time quantifiers like “always” or “eventually”.

For example, in the model of a D-flip flop the state variables would be the input, the clock, the output, its complement, and the reset. The states would be all possible values of the state variables. A simple property might be that if the reset signal is 0, then eventually the output will be 0.

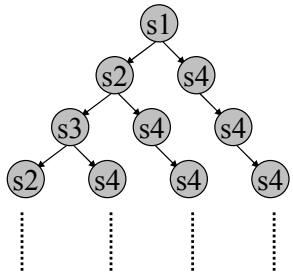
The model checker works on the state transition system of the model and the given property and produces a result TRUE is the property holds in the model. If the property does not hold, the checker gives a counter-example to show that the property is violated. This feature of model checking is very helpful in debugging because it provides a readymade test case. In the given figure, we see the state transition system of M and a temporal property stating that if P2 is true then eventually P4 will be true.

Model Checking (2/2)

- **Idea (Clarke, Emerson '81)**
  - Unroll transition system to an infinite computation tree
    - Start state is the root (S1)
  - Define properties using
    - On all paths (A)
    - On some path (E)
    - Always / Globally (G)
    - Eventually (F)
  - Some examples
    - EG p
    - AG p
    - EF p
    - AF p
- **State space explosion**
  - What next ?



Transition system



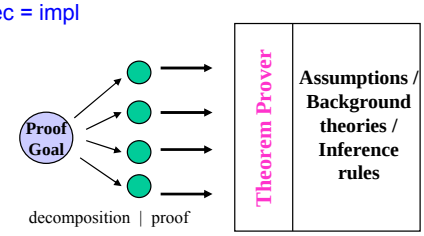
Computation Tree

The idea behind model checking can be visualized by unrolling the transition system. We start with the initial state and form an infinite tree (called the computation tree). Temporal properties can by graphically visualized on this computation tree.

The major problem with model checking is the state space explosion problem. The state transition system grows exponentially with the number of state variables. Therefore, memory for storing the state transition system becomes insufficient as the design size grows.

Theorem Proving (1/2)

- **Task : Prove implementation is equivalent to spec in given logic**
- **Inputs**
  - Formula for specification in given logic (spec)
  - Formula for implementation in given logic (impl)
  - Assumptions about the problem domain
    - Example : Vdd is logic value 1, Gnd is logic value 0
  - Background theory
    - Axioms, inference rules, already proven theorems
- **Output**
  - Proof for spec = impl



Manual **Automated**

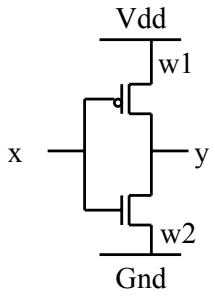
An alternative approach to formal verification is verification by deductive reasoning. The specification and implementation models are written as formulas in some mathematical logic. Then a theorem is established and proven for the equivalence of these formulas. If a proof is found, the models are equivalent. However, if a proof is not found then the equivalence of models is inconclusive.

The proof uses certain assumptions about the problem domain and axioms of the mathematical logic. In the domain of circuit design, an assumption might be that power supply is always at logic level 1 while ground is logic 0. The proof is constructed by breaking down a complex proof goal into smaller goals. The smaller goals are then simplified using assumptions and then passed onto automatic theorem prover.

Theorem proving is still a largely manual process. Several steps of simplifying and breaking down proof goals may be required before an automatic prover can solve it.

Theorem Proving (2/2)

- **Example**
  - CMOS inverter (Gordon'92)
  - Using higher order logic
- **Assumptions**
  - Vdd(y) := (y=T)
  - Gnd(y) := (y=F)
  - Ntran(x,y1,y2) := (x->(y1=y2))
  - Ptran(x,y1,y2) := (¬ x->(y1=y2))
- **Impl(x,y) :=** ∃ w1, w2. Vdd(w1) ∧ Ptran(x,w1,y) ∧ Ntran(x,y,w2) ∧ Gnd(w2)
- **Spec(x,y) :=** (y=¬ x)
- **Proof**
  - Impl(x,y) = ….. (assumption / thm / axiom)
    = ….. (assumption / thm / axiom)
    = ….. (assumption / thm / axiom)
    = Spec(x,y)



CMOS inverter

We present a simple example of the use of theorem proving for verifying circuits. Suppose we have to prove that the CMOS inverter circuit inverts the input logic. We start with the basic assumptions about voltage levels and logic levels and the behavior of P and N transistors. The formula for the implementation is derived by conjunction of the various components of the inverter. The specification formula simply states that the output is logical inverse of input. The proof process takes the implementation formula and reduces it to the specification formula by a number of steps. Each proof step uses either an assumption, an axiom or an already proven theorem.

Drawbacks of formal methods

- **Equivalence checking**
  - Designs to be compared must be similar for LEC
    - Correlated logic segments are identified by design structure
    - Drastic transformations may force manual identification of segments
  - FSM EC requires spec and implementation to
    - Be represented as finite state machines
    - Have same input and output symbols
- **Model Checking**
  - State explosion problem
    - Insufficient memory for designs with > 200 state variables
  - Limited types of designs
    - Design should be represented as a finite transition system
- **Theorem Proving**
  - Not easy to deploy in industry
    - Most designers don't have background in math logic (esp. HOL)
    - Models must be expressed as logic formulas
  - Limited automation
    - Extensive manual guidance to derive proof sub-goals

Verify 2003

Formal verification methods have not been as well accepted in the industry as simulation based methods because of several drawbacks. Logical equivalence checking works only for combinational logic and FSM equivalence checking requires both specification and implementation machines to have the same set of inputs and outputs.

Model checking, besides suffering from the state explosion problem, is not suitable for all types of designs. Since it needs a state transition system, it works best for control intensive designs like protocol compliance etc.

Automatic theorem proving has not become very popular in the industry because of several reasons. The foremost reason is the amount of manual intervention required in running the theorem proving. Since different applications have different kinds of assumptions and proof strategies, it is infeasible for a theorem proving tool to generate the entire proof automatically. Secondly, most designers lack a background in mathematical logic. Therefore, it requires a huge investment and long training time for them to start using theorem proving efficiently.

Improvements to Formal Methods

- **Symbolic Model Checking (McMillan '93)**
  - Represent states and transitions as BDDs
    - Allows many more states (~10^20) to be stored
    - Compare sets of states for equality using SAT solver
- **Bounded Model Checking (Biere et.al. '99)**
  - Restricted to bugs that appear in first K cycles of model execution
    - Unfolded model and property are written as propositional formula
    - SAT solver or BDD equivalence used to check model for property
- **Partial Order Reduction (Peled '97)**
  - Reduces model size for concurrent asynchronous systems
    - Concurrent tasks are interleaved in asynchronous models
    - Check only for 1 arbitrary order of tasks
- **Abstraction (Long, Grumberg, Clarke '93)**
  - Cone of influence reduction
    - Eliminate variables that do not influence variables in spec

Verify 2003

There have been several improvements to formal techniques, particularly in model checking. Symbolic model checking encodes the state transition system using BDDs, which is much more compact than exhaustively enumerating the states and transitions. Since BDDs represent sets of states, the model checking algorithm can operate on sets of states rather than individual states.

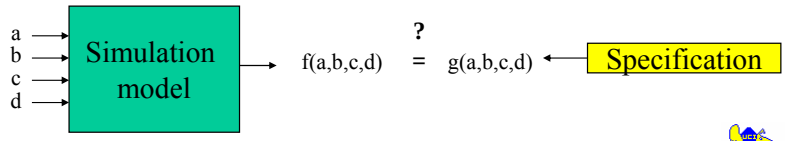
Bounded model checking checks if a model satisfies a property on paths of length at most K. The number K is incremented until a bug is found or the problem becomes intractable.

Partial order reduction techniques are usually used in model checking of asynchronous systems, where concurrent tasks are interleaved rather than being executed simultaneously. It uses the commutativity of concurrently executed transitions, which result in the same state when executed in different orders.

Abstraction techniques are used to create smaller state transition graphs. The specified property is described using some state variables. The variables that do not influence the specified property are eliminated from the model, thereby preserving the property while reducing the model size.

Semi-formal Methods (Symbolic Simulation)

- **Task : Check if implementation satisfies specification**
- **Inputs**
  - Simulation model of the circuit
  - Specification of expected behavior (as boolean expressions)
- **Output**
  - Expression for the signals in design
- **Idea (Bryant '90)**
  - Encode set of inputs symbolically (using BDD)
  - Evaluate output expressions during simulation
  - Compare simulation output with expected output
    - using BDD canonical form



The idea behind symbolic simulation is to significantly minimize the number of simulation test vectors, for the same coverage, by using symbols.

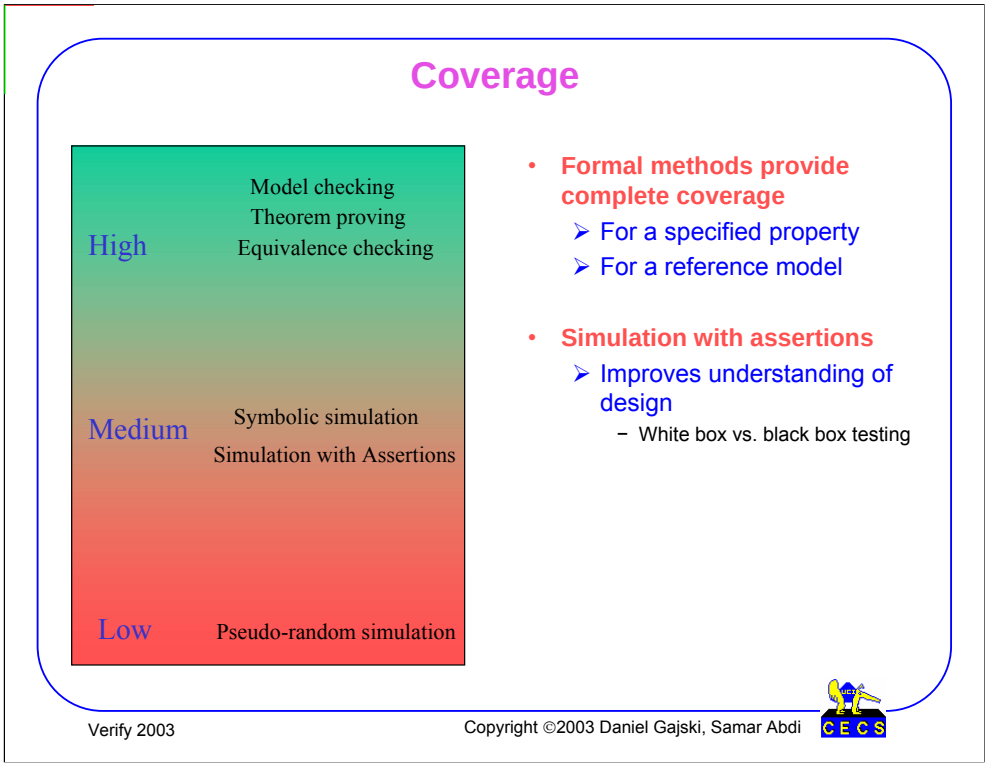
In symbolic simulation, the stimulus applies boolean variables as inputs to the simulation model. During simulation, the internal variables and outputs are computed as booelan expressions. In order to check for correctness, the output expression is compared with the expected output expression for logic equivalence. BDDs can be used to store the boolean expressions. Since, BDDs of equivalent boolean expressions can be reduced to the same canonical form, the equivalence of specified output expression to simulated output expression can easily be checked. For larger circuits, where the BDD size may blow up, SAT solvers are being increasingly used.

Evaluation Metrics

- **Coverage**
  - How exhaustive is the technique ?
    - % of statements covered
    - % of branches taken
    - % of states visited / state transitions taken
- **Cost and Effort**
  - How expensive is the technique ?
    - Dollars spent per simulation / emulation cycle
    - Training time for users
- **Scalability**
  - How well does the technique scale with design size / abstraction ?
    - Tool capacity
    - Tool applicability for various modeling abstraction levels

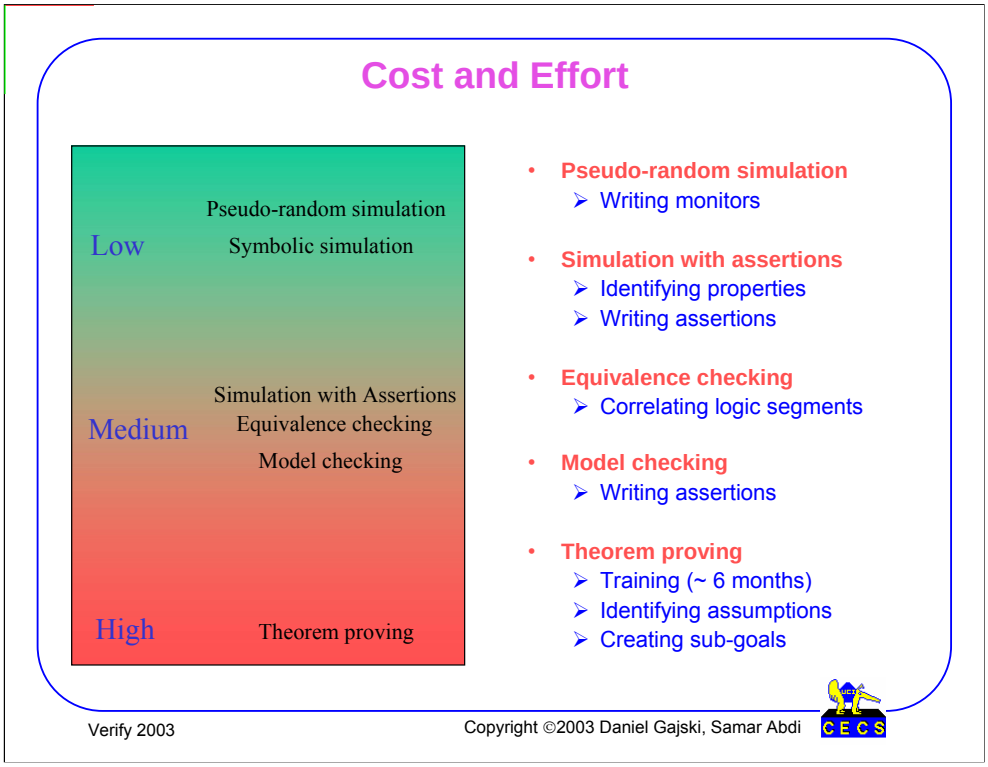
Verify 2003

In order to determine the most suitable verification method, one can define some metrics to evaluate them. The three most common metrics that we discuss here are coverage, cost and scalability. Coverage of a verification method determines how much of the design's functionality has been tested. Cost includes the money spent on purchase of tools, hiring of experts and the training of users. Scalability of the technique shows if there are any limitations on the size or type of design that we are verifying.

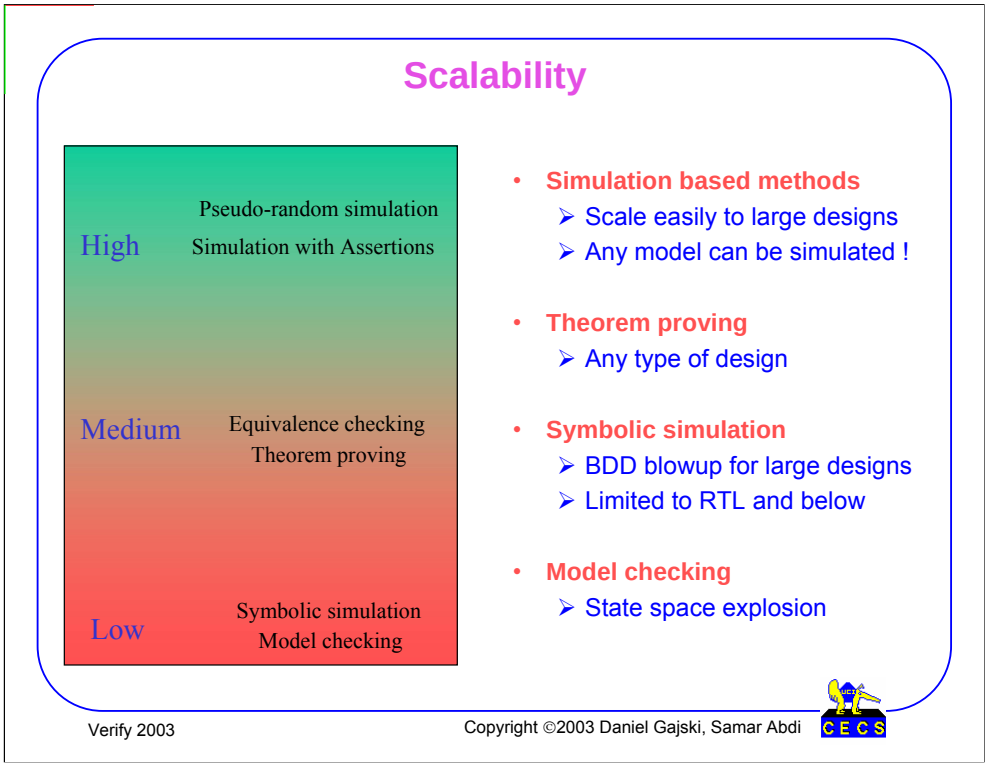


Formal verification claims to provide complete coverage. However, the coverage is limited to the given property and the model representation. For instance, model checking covers all possible states in the state transition representation of the model for a given property. Logic equivalence checking covers the combinational part of the model only. Nevertheless, the coverage of formal methods, if they are applicable, is significantly more than that of simulation methods for the same run-time.

Using assertions in the design can help make better test cases that exercise the assertions, thereby ensuring that the tests are useful and valid. Pseudo random testing, on the other hand, would generate a lot of test inputs that are invalid for the design, and hence wasted.



Cost and effort of a verification method influences the design phase in which it is used. For instance, the preliminary phase usually employs simulation to uncover most of the easy bugs. This is because most designers have experience with simulation tools and debuggers and it is thus cost effective. As the verification process continues and bugs become harder to find, specialized and more expensive techniques like model checking or theorem proving may be used. Assertions are also used to generate more directed tests and to verify correctness on corner cases.



The performance of a verification method on different sizes and types of models determines its scalability. Some methods like logic equivalence checking may be limited to RTL models or below. Similarly, model checking is constrained by the number of state variables in the model. Compared to other techniques, simulation scales very well in this department. Almost any executable model at any level of abstraction can be simulated.

Evaluating Verification Techniques

Technique \ Metric	Coverage	Cost and Effort	Scalability
Pseudo random simulation	L	L	H
Simulation w/ assertions	M	M	H
Symbolic simulation	M	L	L
Equivalence checking	H	M	M
Model checking	H	M	L
Theorem proving	H	H	M

- **Well accepted techniques in industry**
  - Simulation with assertions
  - Equivalence checking

Verify 2003

If we look at the trend in the acceptance of verification techniques in the industry, we find that methods with a severe drawback have been generally avoided. Model checking suffers from poor scalability and theorem proving is way too expensive, thereby making equivalence checking the most commonly used technique in the industry.

Similarly, assertion based techniques may require extra cost but they are replacing pseudo random simulation because of their better coverage. A number of new verification and assertion languages are testimony to this fact.

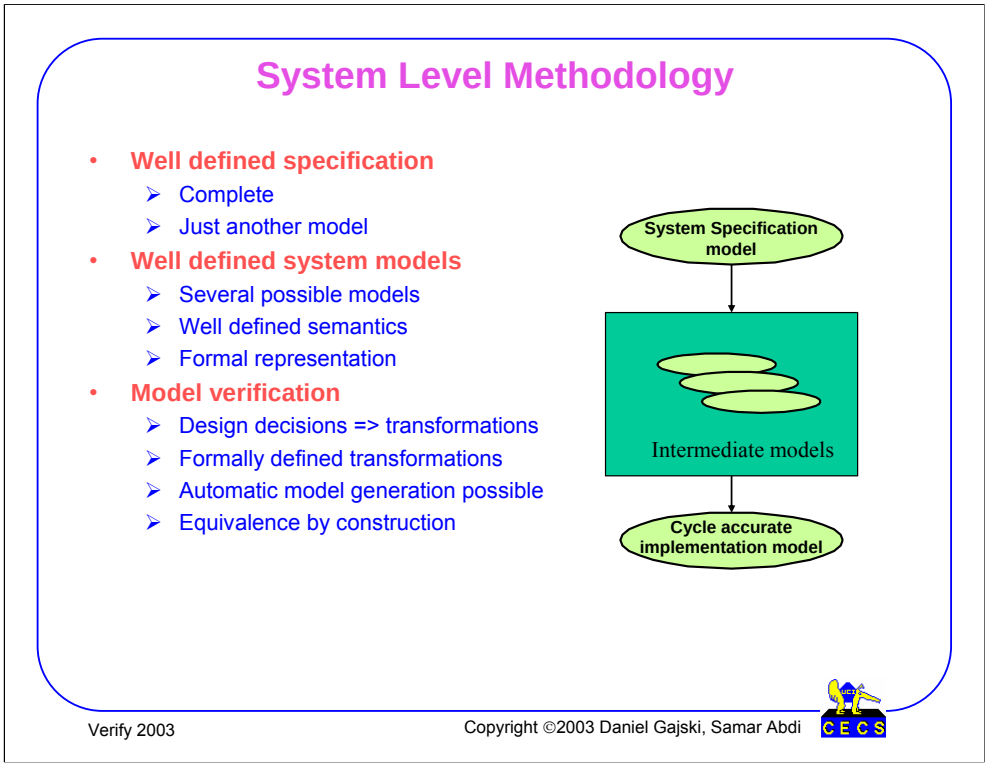
New Verification Challenges for SoC Design

- **Design complexity**
  - Size
    - Verification either takes unreasonable time (eg. Logic simulation)
    - Or takes unreasonable memory (eg. Model Checking)
  - Heterogeneity
    - HW / SW components on the same chip
    - Interface problems
    - Interdependence of both design teams
- **Possible directions**
  - Methodology
    - Unified HW/SW models
    - Model formalization
    - Automatic model transformations

Verify 2003

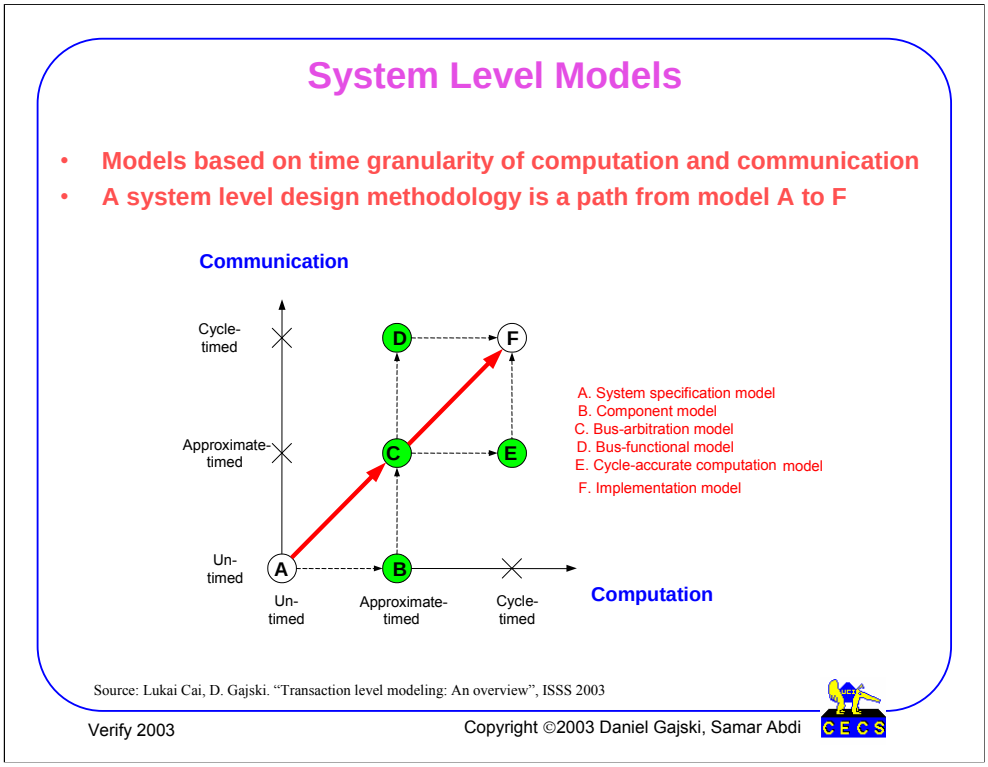
The new challenges to verification of systems comes from the growth in size and complexity of designs. Individually verified components do not work together due to interface issues. Also the sheer size of designs makes modeling and verification too expensive and time consuming.

To answer this challenge, we look towards system design methodology. To design systems on chip, the level of model abstraction has been raised. If the semantics of system level models is well defined, then they can be formalized. Consequently, we can define transformations from models at one abstraction level to another. The transformations can be proven to produce equivalence models. Thus, the traditional methods can still be used at higher levels of abstraction while correct transformations will avoid the need to verify lower level models as well.



A system level methodology starts with a well defined executable specification model that serves as the golden reference. The specification is gradually refined to a cycle accurate model that can be fed to traditional simulation and synthesis tools. The gradual refinement produces some intermediate models depending on the choice of methodology.

The details that are added to models during refinement depend on the designer decisions. Each decision corresponds to one or more model transformations. If all the transformations are formally defined, the refinement process can be automated.

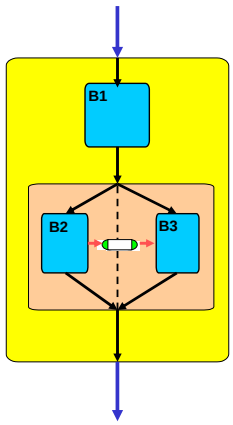


In general, system level models can be distinguished by the accuracy of timing for their communication and computation. In the graph, the two axes represent the granularity of communication and computation. The functional specification at the origin is untimed, with only a causal ordering between tasks. On the other end is the cycle accurate model.

A system level methodology takes the untimed specification to its cycle accurate implementation. The path through the intermediate models determines the refinements that need to be performed.

Model Definition

- **Model = < {objects}, {composition rules} >**
- **Objects**
  - Behaviors
    - tasks / computation / function
  - Channels
    - communication between behaviors
- **Composition rules**
  - Sequential, parallel, FSM
  - Behavior / channel hierarchy
  - Behavior composition also creates execution order
    - Relationship between behaviors in the context of the formalism
- **Relations amongst objects**
  - Connectivity between behaviors and channels

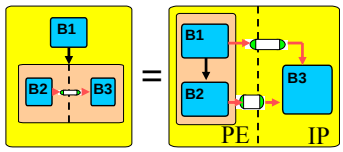


Verify 2003

Formally, a model is a set of objects and composition rules defined on the objects. A system level model would have objects like behaviors for computation and channels for communication. The behaviors can be composed as per their ordering . The composition creates hierarchical behaviors that can be further composed. Interfaces between behaviors and channels or amongst behaviors themselves can be visualized as relations.

Model Transformations (1/2)

- **Design Decision**
  - Map behaviors to PEs
- **Model Transformations**
  - Rearrange object composition
    - Distribute computation over PEs
  - Replace objects
    - Import IP components
  - Add / Remove synchronization
    - Transform sequential composition to parallel and vice-versa



Distribution of behaviors (tasks) over components

analogous to……

a*(b+c) = a*b + a*c

Distributivity of multiplication over addition

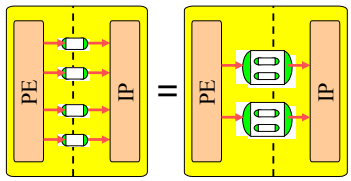
Verify 2003

A transformation on a model can be expressed using the concept of rearranging and replacing objects. For instance, in order to distribute the behaviors in a specification onto components of the system architecture, we need to rearrange the behaviors into groups. In order to use IP components, we need to replace behaviors in the model with an IP from the library. Each of these transformations has to be proven correct in a formal context.

Intuitively, we can draw an analogy between distributive law for natural numbers and distribution of behaviors on components as shown in the figure. Just as the expression on LHS is equal to that on RHS in the distributive law equation, we have a model on the LHS equal to a model on the RHS. The equality is determined by the order in which the behaviors execute.

Model Transformations (2/2)

- **Design Decision**
  - Map channels to buses
- **Model Transformations**
  - Rearrange object composition
    - Group channels according to bus mapping
    - Slice complex data into bus words
  - Replace objects
    - Import bus protocol channels



Mapping of channels to buses

analogous to……

a+b+c+d = (a+b) + (c+d)

Associativity of addition

Verify 2003

Another designer decision would be to map the abstract data channels to system buses in order to implement the inter-component communication. To reflect these decisions, we need to perform certain model transformations. These transformations would include the grouping on abstract channels as per the bus mapping and creation of hierarchical channels. The hierarchical channels represent the system level bus architecture. Eventually, these hierarchical channels need to be replaced with bus protocol channels and drivers need to be added in components to implement the data transfer.

The grouping transformation can be seen as analogous to associative rule for addition of natural numbers. No matter how we group the summation terms, the result would always be the sum of all the numbers. Similarly, no matter how the abstract channels are grouped in the transformed model, they would perform the same data transfer as in the original model.

Model Refinement

- **Definition**
  - Ordered set of transformations < t_m, … , t_2, t_1 > is a refinement
    - *model B* = t_m(… (t_2(t_1(*model A*))) …)
- **Equivalence verification**
  - Each transformation maintains functional equivalence
  - The refinement is thus correct by construction
- **Derives a more detailed model from an abstract one**
  - Specific sequence for each model refinement
  - Not all sequences are relevant
- **Refinement based system level methodology**
  - Methodology := < {models}, {refinements} >

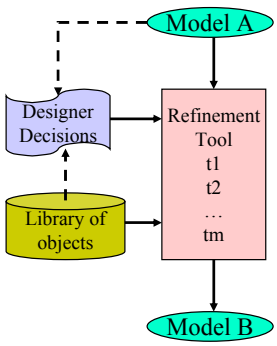
Verify 2003

A model refinement can be expressed as a well defined sequence of transformations. Since each transformation is shown to be correct, the refinement also produces an output model equivalent to the input model.

A refinement based methodology can be defined as a set of well defined models and the refinements between them.

System Verification through Refinement

- **Set of models**
- **Designer Decisions => transformations**
  - Select components / connections
    - Import behaviors / protocols
  - Map behaviors / channels
    - Synchronize behaviors / slice data
- **Transformations preserve equivalence**
  - Same partial order of tasks
  - Same input/output data for each task
  - Same partial order of data transactions
  - Equivalent replacements
- **All refined models will be "equivalent" to input model**
  - Still need to verify
    - First model
    - Correctness of replacements



Verify 2003

In a refinement based system level methodology, each model produced by a refinement is equivalent to the input model. As shown in the figure, designer decisions are used to add details to a model to refine it to the next lower level of abstraction. Each designer decision corresponds to a transformation in the model. The transformations would either rearrange the computation and communication objects or replace an object in the model with one from the library.

The notion of model equivalence comes from the simulation semantics of the model. Two models are equivalent is they have the same simulation results. This translates to the same (or equivalent) objects in both models and the same partial order of execution between them. Correct refinement, however, does not mean that the output model is bug free. We also need to use traditional verification techniques on the specification model and prove equivalence of objects that can be replaced with each other.

Conclusion

- **Variety of verification techniques available**
  - Several tools from industry and academia
  - Each technique works well for specific kind / level of models
- **Challenges for verification of large system designs**
  - Simulation based techniques take way too long
    - Time to market issues
  - Most formal techniques cannot scale
    - Memory requirement explosion
    - Too much manual effort required
- **Modeling is pushed to system level**
- **Future design and verification**
  - Complete and executable functional specification model
  - Well defined semantics for models at different abstraction levels
  - Well defined transformations for design decisions
    - Verify transformations
    - Automate refinements
- **Formalism helps system verification !**

Verify 2003

In conclusion, we have looked at several verification techniques available from both the industry and the academia. As the size and complexity of designs increase, traditional techniques might not be able to keep pace. A system design methodology will well defined model semantics may be a possible solution to the problem.

Specifying the design at a higher level of abstraction would make traditional verification and debugging tractable because of smaller model size. Well defined model semantics would make it possible to define and prove correct transformations for automatic model refinement. Therefore, formalisms would make complete system verification much faster.

References

- Devadas, Ma, Newton, “On the verification of sequential machines at different levels of abstraction”, 24th DAC, pp.271-276, June 1987
- Clarke, Grumberg, Peled, “Model Checking” , MIT Press
- K.L. McMillan, “Symbolic Model Checking: An approach to the State Explosion Problem” , Kluwer Academic 1993
- McFarland, “Formal Verification of Sequential Hardware: A tutorial”, IEEE Transaction on CAD, pp. 633-653, May 1993
- Thomas Kropf, “Introduction to Formal Hardware Verification” Springer, 1999
- Gordon, “Specification and Verification of Hardware”, University of Cambridge, October 1992
- Lionel Bening, Harry Foster, “Principles of Verifiable RTL Design”, Kluwer 2000