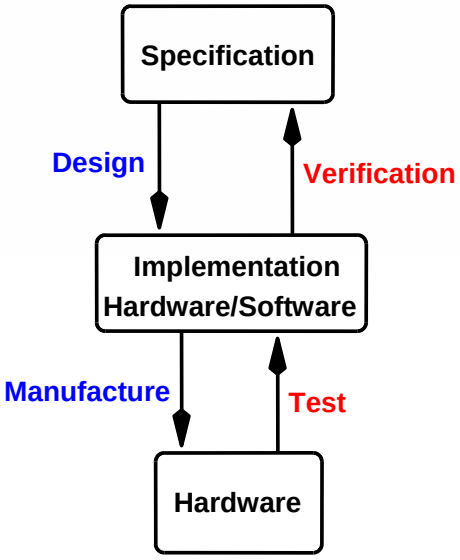
Verification of SoC Designs

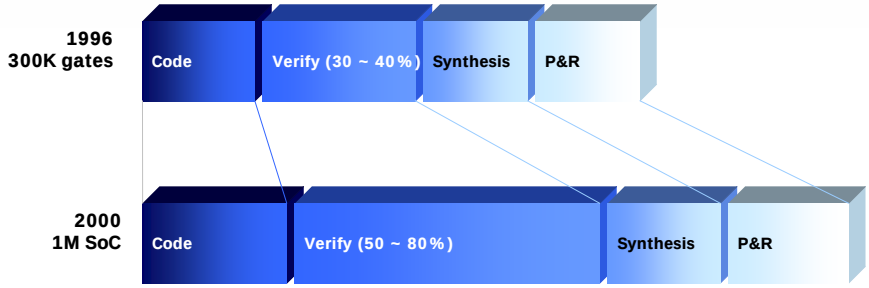
- Simulation-based techniques
- Formal analysis
- Dealing with state explosion
- Verification of embedded software

Verification versus Test



Verification Effort

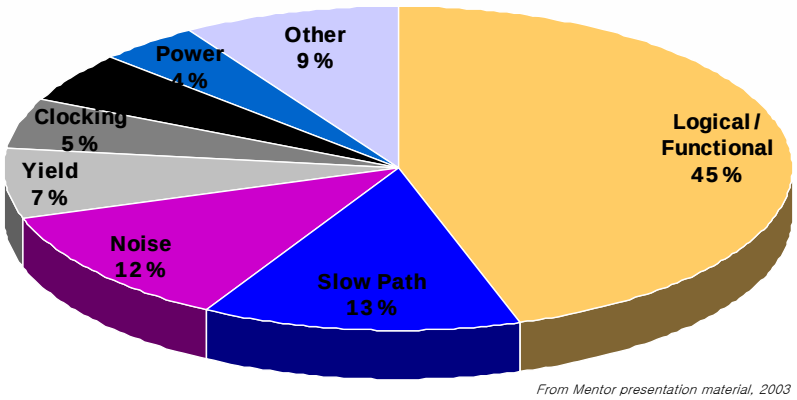
- Verification portion of design increases to anywhere from 50 to 80% of total development effort for the design.



Verification methodology manual, 2000-TransEDA

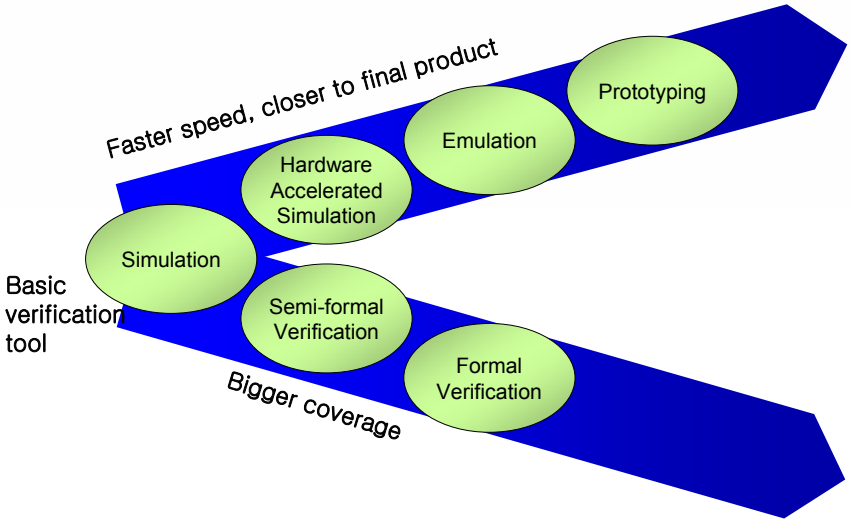
Percentage of Total Flaws

- About 50% of flaws are functional flaws.
  - Need verification method to fix logical & functional flaws



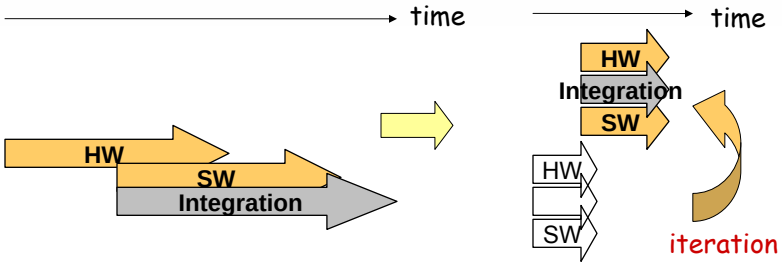
From Mentor presentation material, 2003

Verification Approaches



HW/SW Co-Design

- Concurrent design of HW/SW components
- Evaluate the effect of a design decision at early stage by "virtual prototyping"
- **Co-verification**



Verification Options

- Simulation Technologies
- Equivalence Checking
- Formal Analysis (Model Checking)
- Physical Verification and Analysis

Simulation Technologies

- Event-based Simulators
- Cycle-based Simulators
- Transaction-based Simulators
- Code Coverage
- HW/SW Co-verification
- Emulation Systems
- Rapid Prototyping Systems
- Hardware Accelerators
- AMS Simulation
- Numerical Simulation (MATLAB)

Static Technologies

- "Lint" Checking
  - Syntactic correctness
  - Identifies simple errors
- Static Timing Verification
  - Setup, hold, delay timing requirements
  - Challenging: multiple sources

Formal Techniques

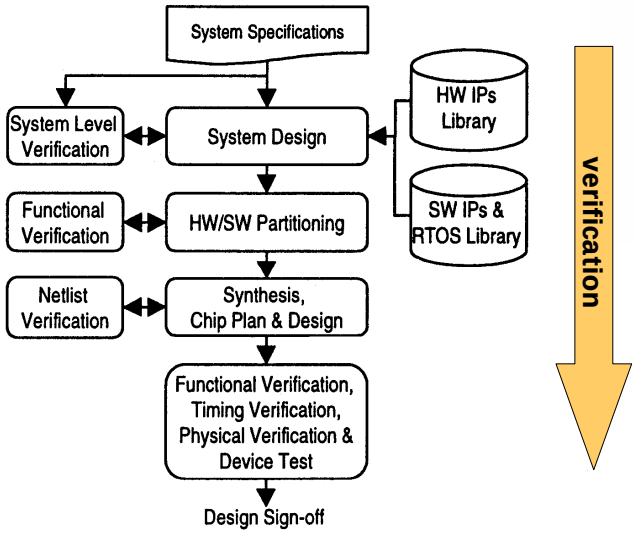
- Theorem Proving Techniques
  - Proof-based
  - Not fully automatic
- Formal Model Checking
  - Model-based
  - Automatic
- Formal Equivalence Checking
  - Reference design ←→ modified design
  - RTL-RTL, RTL-Gate, Gate-Gate implementations
  - No timing verification

Physical Verification & Analysis

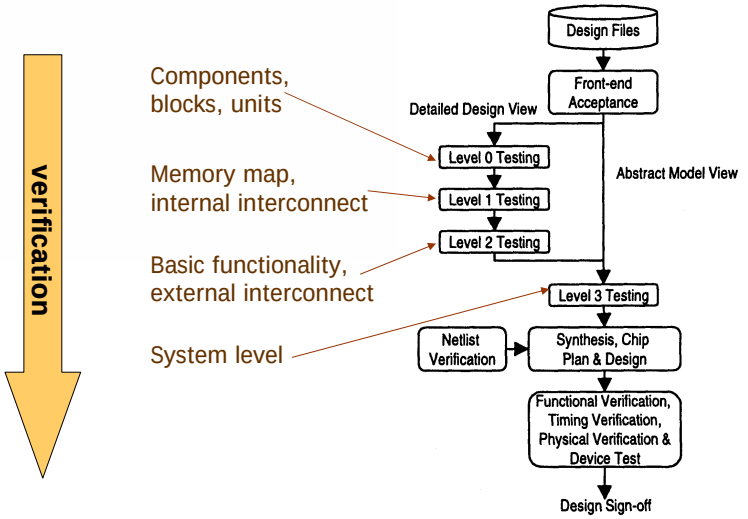
Issues for physical verification:

- Timing
- Signal Integrity
- Crosstalk
- IR drop
- Electro-migration
- Power analysis
- Process antenna effects
- Phase shift mask
- Optical proximity correction

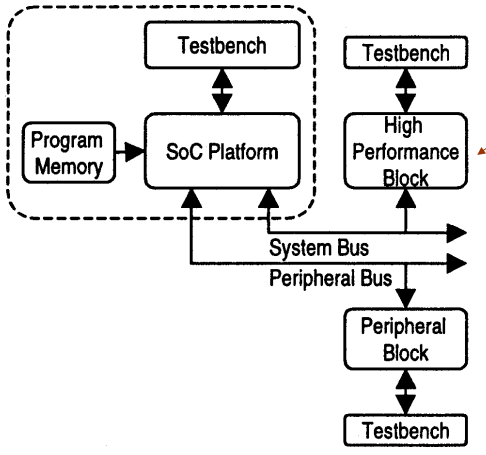
Top-Down SoC Verification



Bottom-Up SoC Verification



Platform Based SoC Verification

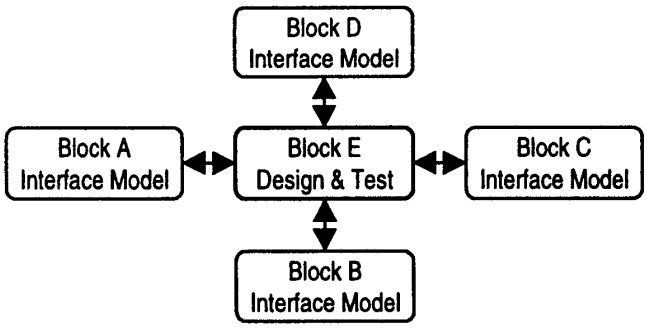


Derivative Design

Interconnect Verification between:

- **SoC Platform**
- **Newly added IPs**

System Interface-driven SoC Verification

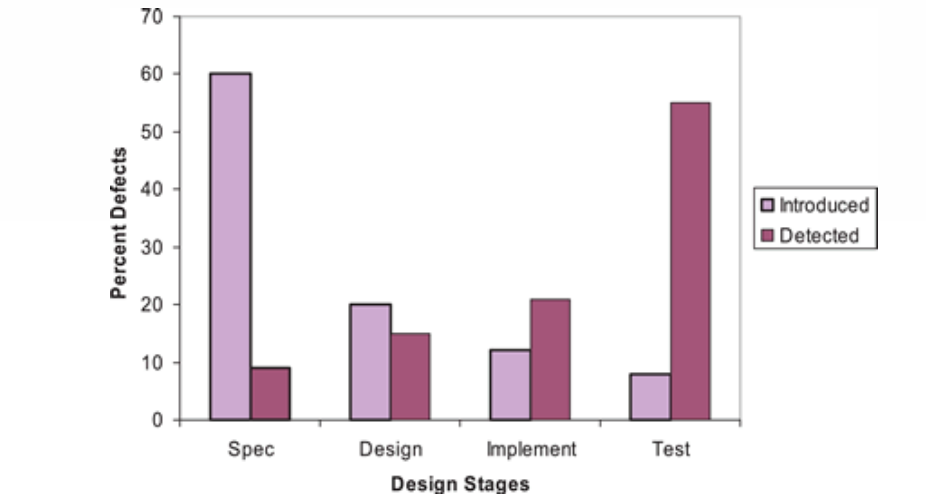


Besides Design-Under-Test, all others are interface models

Traditional Testbench

- Problems of Traditional Testbench
  - **Real-World Stimuli**
  - **System-Level Modeling**
  - **High-Level Algorithmic Modeling**
  - **Test Automation**
  - **Source Coverage**

"Bug" Introduction and Detection

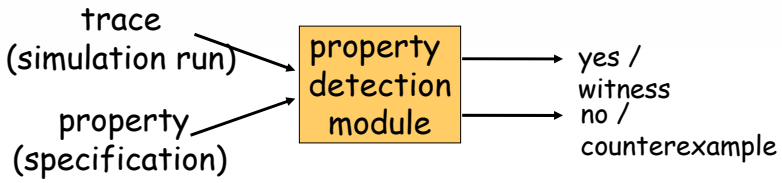


Executable Specification

- Procedural Language for Behavioral Modeling
  - Design Productivity
    - Easy to model complex algorithm
    - Fast execution
    - Simple Testbench
  - Tools
    - Native C/C++ through PLI/FLI
    - Extended C/C++ : SpecC, SystemC
- Verify it on the fly!
  - Test vector generation
  - Compare RTL Code with Behavioral Model
  - Coverage Test

Property Detection

Property detection: to decide whether a simulation run (trace) of a design satisfies a given property



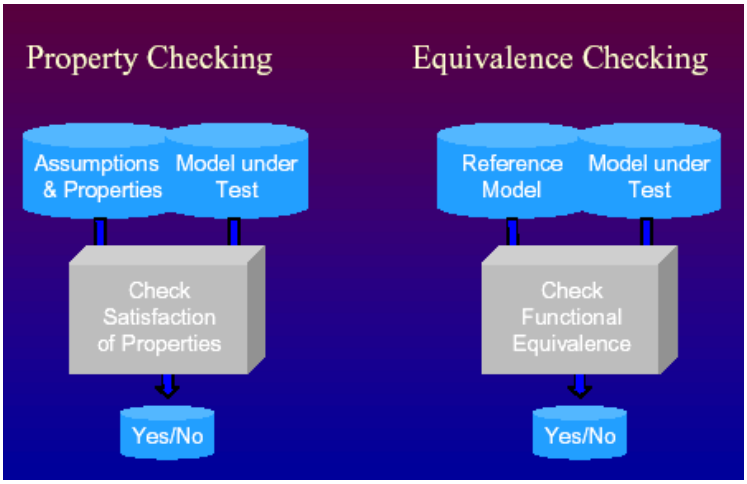
e.g., violation of mutual exclusion, $critical_1$ Æ $critical_2$

Example: Properties written in PSL/Sugar

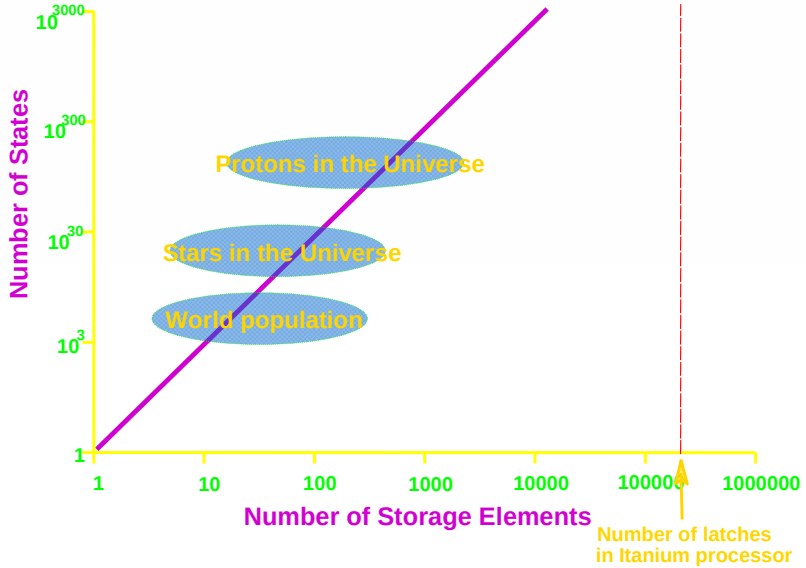
Specifying Properties (Assertions) in Industry Tools

- Open Vera Assertions Language (Synopsys)
- Property Specification Language (PSL) (IBM, based on *Sugar*)
  - Accelera driving consortium
  - IEEE Std. 1850-2005
- Accelera Open Verification Library (OVL) provides ready to use assertion functions in the form of VHDL and Verilog HDL libraries
- SystemVerilog is a next generation language, added to the core Verilog HDL
  - IEEE Std. 1800-2005

Formal Verification of SoCs



State Explosion!



Abstractions to Deal with Large State Spaces

- Model checking models need to be made smaller
- Problem: State-Space Explosion
- Smaller or "reduced" models must retain information
  - Property being checked should yield same result
- Balancing solution: Abstractions

Program Transformation Based Abstractions

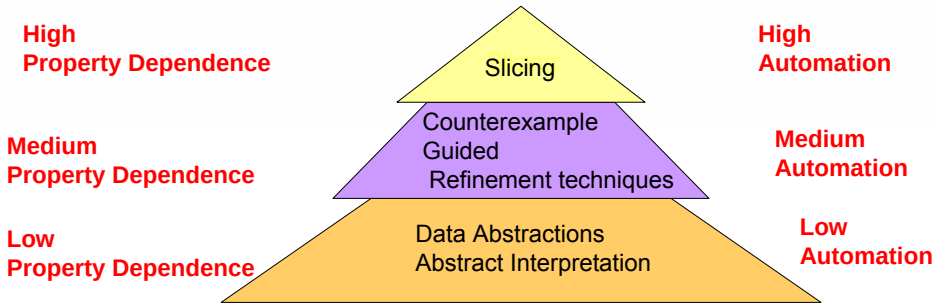
- Abstractions on Kripke structures
  - Cone of Influence (COI), Symmetry, Partial Order, etc.
  - State transition graphs for even small programs can be very large to build
- Abstractions on Program Text
  - Scale well with program size
  - High economic interest

Static Program Transformations

Types of Abstractions

- Sound
  - Property holds in abstraction implies property holds in the original program
- Complete
  - Algorithm always finds an abstract program if it exists
- Exact
  - Property holds in the abstraction iff property holds in the main program

Abstraction Landscape



Verification of challenging problems with high level static analysis

Antecedent Conditioned Slicing

- RTL abstraction technique
- Applied to LTL formulas
  - G(a =>c)
- Theoretically complex, practically effective
- USB 2.0 protocol verification

Property checking

- High level symbolic simulation
  - Symbolic simulation of antecedent
  - Symbolic simulation of all CFG nodes
- Domain aware analysis
  - Function-wise case splitting
- Decision procedure
  - Model checker

Program Slicing

- Program transformation involving statement deletion
- "Relevant statements" determined according to *slicing criterion*
- Slice construction is completely *automatic*
- Correctness is *property specific*
  - Loss of generality
- Abstractions are sound and complete

Specialized Slicing Techniques

- Static slicing produces large slices
  - Has been used for verification
  - Semantically equivalent to COI reductions
- Slicing criterion can be enhanced to produce other types of slices
  - Amorphous Slicing
  - Conditioned Slicing

Conditioned Slicing

- Slices constructed with respect to set of possible input states
- Characterized by first order, predicate logic formula
- Augments static slicing by introducing condition
  - <C, I, V>
  - Constrains the program according to condition C
- Canfora et al

Example Program

```
begin

1:          read(N);
2:          A = 1;
3:          if (N < 0) {
4:                  B = f(A);
5:                  C = g(A);
6:          } else if (N > 0) {
7:                  B = f'(A);
8:                  C = g'(A);
            } else {
9:                  B = f''(A);
10:                         C = g''(A);
            }
11:                 print(B);
12:                 print(C);

end
```


Example Program: Static Slicing wrt <11, B>

```
begin

1:          read(N);
2:          A = 1;
3:          if (N < 0) {
4:                  B = f(A);
5:                  C = g(A);
6:          } else if (N > 0) {
7:                  B = f'(A);
8:                  C = g'(A);
            } else {
9:                  B = f''(A);
10:                         C = g''(A);
            }
11:                 print(B);
12:                 print(C);

end
```

Example Program: Conditioned Slicing wrt <(N<0),11, B>

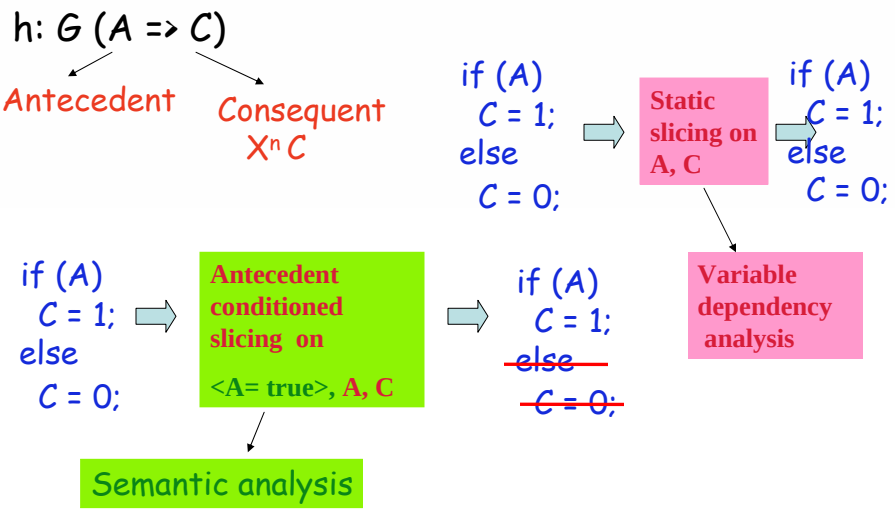
```
begin

1:          read(N);
2:          A = 1;
3:          if (N < 0) {
4:                  B = f(A);
5:                  C = g(A);
6:          } else if (N > 0) {
7:                  B = f'(A);
8:                  C = g'(A);
            } else {
9:                  B = f''(A);
10:                         C = g''(A);
            }
11:                 print(B);
12:                 print(C);

end
```


Verification Using Conditioned Slicing

- Slicing part of design irrelevant to property being verified
- Safety Properties of the form
  - G (antecedent => consequent)
- Use *antecedent* to specify states we are interested in

We do not need to preserve program executions where the antecedent is false

Property checking: antecedent conditioned slicing



Example of antecedent conditioned slicing

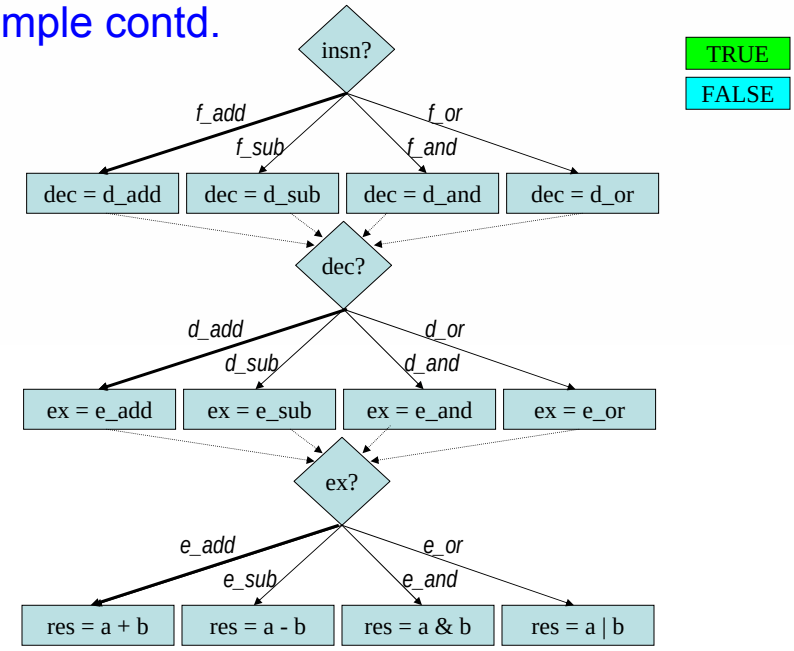
```
always @ (clk)      begin
    case(insn)
      f_add:   dec = d_add;
      f_sub:   dec = d_sub;
      f_and:   dec = d_and;
      f_or:    dec = d_or;
    endcase
end
```

```
always @ (clk)      begin
    case(dec)
      d_add:   ex = e_add;
      d_sub:   ex = e_sub;
      d_and:   ex = e_and;
      d_or:    ex = e_or;
    endcase
end
```

```
always @ (clk)      begin
    case(ex)
      e_add:   res = a+b;
      e_sub:   res = a-b;
      e_and:   res = a&b;
      e_or:    res = a|b;
    endcase
end
```

$h = [G((insn == f\_add) \Rightarrow XX(res == a+b))]$

Example contd.



Checking the truth of the antecedent

Node retained

Symbolic simulation of a node in CFG

(dec = d_add)

Antecedent symbolic expression

(insn == f_add)

dec(t+1) = ITE((insn==f_add)(t)^
~(insn == f_sub)(t)^
~(insn == f_and)(t)^
~(insn == f_or),
d_add(t), dec(t))

Rewriter

T

T: Retained
X: Retained
F: Not retained

Complexity of Antecedent Conditioned Slicing

- Symbolic simulation of all nodes in each process
- Expression computation over all processes in the program
  - Handles global predicates
- Symbolic simulation of the antecedent
- Looking forward in time
  - Depends on *n* in (A => X^nC)
- Decision procedure for checking truth of antecedent
  - Could be arbitrarily hard
- Path traversal of all processes
  - Pruning non-retained nodes

- Worst case: retain all nodes

Correctness of Antecedent Conditioned Slicing

Theorem: An LTL formula *h* of the type, where *h* is

G(a => c)

G (a => $X^{=n}$ c)

G (a => $F^{<= k}$ c)

holds on the original program *iff* it holds on the antecedent conditioned slice.

Proof intuition:

For a Kripke structure of the slice, all states satisfy a=>c.

These include states of the original Kripke structure that satisfy a.

Thus all states of the original that satisfy a must satisfy h.

All states of the original that satisy ¬a, satisfy a=>c vacuously.

Example of Antecedent Conditioned Slicing

```
always @ (clk)      begin        always @ (clk)      begin        always @ (clk)      begin
    case(insn)                       case(dec)                        case(ex)
      f_add:   dec = d_add;            d_add:   ex = e_add;   e_add:   res = a+b;

    endcase                          endcase                          endcase
end                                  end                              end
```

Single instruction behavior for f_add instruction

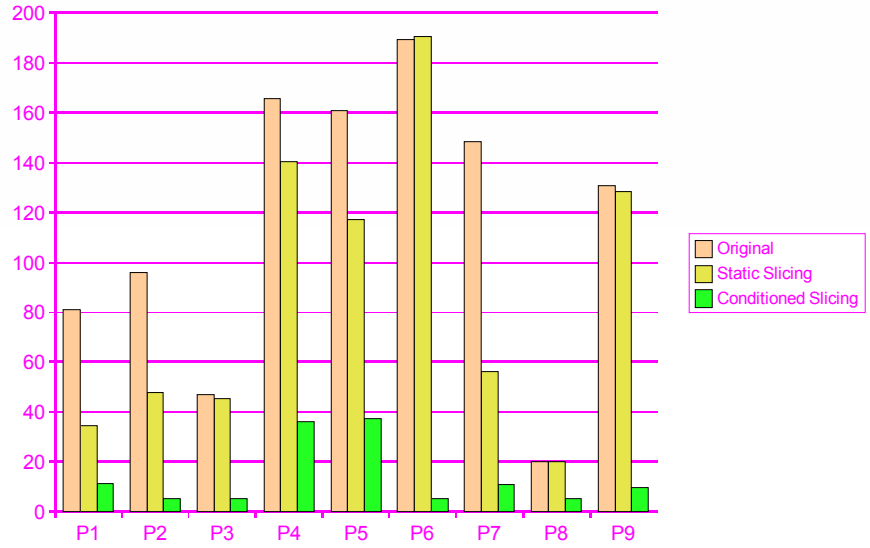
$$h = [G((insn == f\_add) \Rightarrow XX(res == a+b))]$$

Experimental Results

- Verilog RTL implementation of USB 2.0 function core
- Properties taken from specification document
  - Safety properties expressed in LTL
  - Mostly control based, state machine related
- Used Cadence SMV-BMC
  - Circuit too big for SMV
  - Used a bound of 24
- 450 MHz, Ultra Sparc dual processor with 1 GB RAM

Results on USB G(a=>c)Properties

CPU Seconds, 450 MHz dual UltraSPARC-II with 1 GB RAM



Results of Antecedent Conditioned Slicing

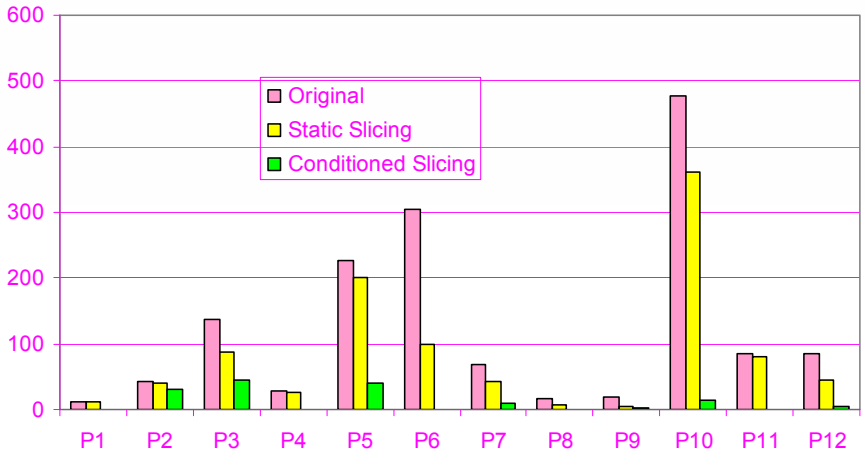
- Temporal property verification for USB 2.0
- Safety properties of the form
  - G(a => Xc)
  - G(a => a U_s c)
- Liveness Properties
  - G(a => Fc)
- USB has many interacting state machines
  - Approximately 10^{33} states
- Bound of 50

Example Properties of the USB

- G((crc5err) V ¬(match) => ¬(send_token))
  - If a packet with a bad CRC5 is received, or there is an endpoint field mismatch, the token is ignored
- G((state == SPEED_NEG_FS) => X((mode_hs) ^ (T1_gt_3_0ms) => (next_state == RES_SUSPEND))
  - If the machine is in the speed negotiation state, then in the next clock cycle, if it is in high speed mode for more than 3 ms, it will go to the suspend state
- G((state == RESUME_WAIT) ^ ¬(idle_cnt_clr) =>F(state == NORMAL))
  - If the machine is waiting to resume operation and a counter is set, eventually (after 100 mS) it will return to normal operation

Results on Temporal USB Properties

CPU Seconds, 450 MHz dual UltraSPARC-II with 1 GB RAM



Verification of challenging problems with high level static analysis

Pipelined Processor Verification

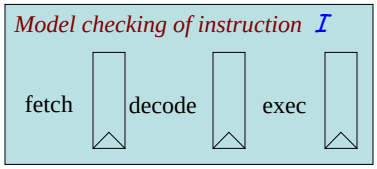
- Reason with the entire state of the machine (Burch and Dill)
- Enhancements use theorem proving techniques
  - –Significant manual component
  - –Construct complicated invariants
  - –High-level model based

Automatic techniques do not scale to instruction level verification

- Antecedent conditioned slicing
- Domain aware analysis
  - Instruction wise case splitting
- Decision procedure
  - Model checker

Single instruction verification

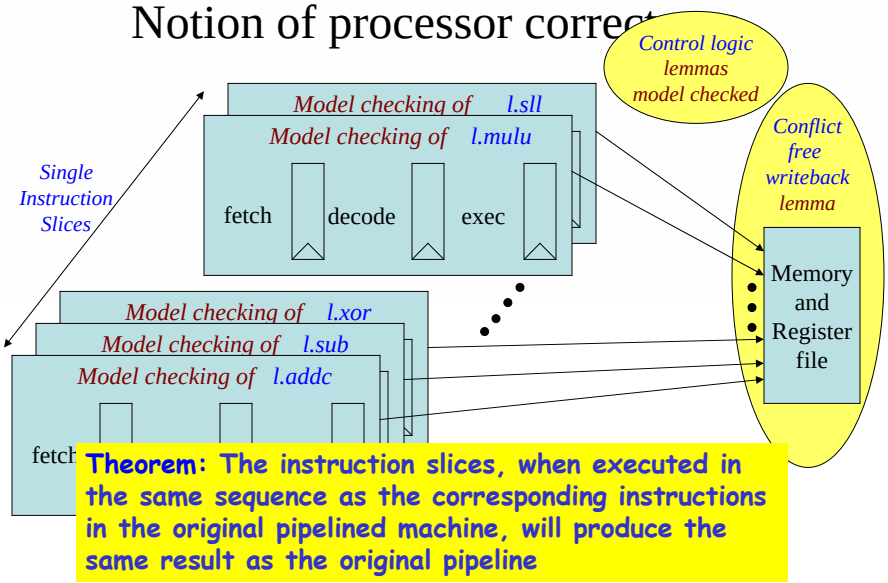
- Obtain single instruction machine by antecedent conditioned slicing
  - Antecedent is instruction word
- Property is $G(I \Rightarrow R)$ where
  - $I = i_1 \wedge Xi_2 \wedge XXi_3 \ldots X^n i_n$
    - i_t represents the antecedent in pipeline stage t
  - R is the result of I in terms of its target register values



Interaction between instructions

Lemma: Instructions should write back only to target register only on writeback stage

Notion of processor correct



Theorem: The instruction slices, when executed in the same sequence as the corresponding instructions in the original pipelined machine, will produce the same result as the original pipeline

Results of OR1200 verification

Class	Insn	SMV Time(s) SLICED	Memory usage (KB)	SMV Time(s) UNSLICED
ALU	l.add	25.65	23796	DNF
ALU	l.sub	24.7	24018	DNF
ALU	l.addi	21.6	19658	DNF
ALU	l.xor	24.84	24831	DNF
ALU	l.and	23.28	21727	DNF
ALU	l.or	24.01	22761	DNF
MAC	l.mul	25.28	49831	DNF
MAC	l.mulu	26.63	22801	DNF
BRANCH	l.bf	132.63	44281	DNF
BRANCH	l.bnf	139.47	46350	DNF
BRANCH	l.j	57.36	31969	DNF
BRANCH	l.jalr	54.09	31073	DNF
BRANCH	l.cmov	159.64	30094	DNF

- OpenRISC 1200
- 32-bit scalar RISC processor
- 5 stage integer pipeline
- Publicly available
- Intended for portable/embedded applications

Results of OR1200 verification

Class	Insn	SMV Time(s) SLICED	Memory (KB)	SMV Time(s) UNSLICED
COMPARE	l.sfeq	157.29	30004	DNF
COMPARE	l.sfne	183.01	51731	DNF
COMPARE	l.sfgt	194.43	53801	DNF
LSU	l.ld	35.85	63112	DNF
LSU	l.lws	33.91	29104	DNF
LSU	l.sd	38.32	30941	DNF
SHF/ROT	l.sll	26.81	23771	DNF
SHF/ROT	l.srl	27.83	24865	DNF
SHF/ROT	l.ror	27.93	26919	DNF
SPRS	l.mfspr	226.97	50696	DNF
SPRS	l.mtspr	212.27	48627	DNF

- 3GHz Pentium4
- 1GB RAM
- Bolstered use of several Boolean level engines
  - Model checkers, SAT, BDD based engines

All instructions of a pipelined processor were verified

Verification of challenging problems with high level static analysis

Sequential equivalence checking

- High level symbolic simulation of RTL implementation
- High level symbolic simulation of System level spec
- Domain aware analysis
  - Sequential compare points obtained using heuristics
- Decision procedure
  - SAT solver

Term Rewriting for Arithmetic Circuit Checking

- Significant success with RTL Term level reductions
- Verification of arithmetic circuits at the RTL level using *term rewriting*
- RTL to RTL equivalence checking
- Verified large multiplier designs like Booth, Wallace Tree and many optimized multipliers using this rewriting technique

Term Rewriting Systems: Example

- Terms: GCD(x,y)
- Rewrite rules:
  - GCD(x,y) ⟩ GCD(y,x) if $x > y$, $y \neq 0$
  - GCD(x,y) ⟩ GCD(x,y-x) if $x \cdot y$, $y \neq 0$
- Initial term: GCD(initX, initY)

$$GCD(6, 15) \overset{R_2}{\Rightarrow} GCD(6, 9) \overset{R_2}{\Rightarrow} GCD(6, 3) \overset{R_1}{\Rightarrow}$$
$$GCD(3, 6) \overset{R_2}{\Rightarrow} GCD(3, 3) \overset{R_2}{\Rightarrow} GCD(3, 0)$$

VERIFIRE

- Dedicated Arithmetic Circuit Checker
- Vtrans: Translates Verilog designs to Term Rewriting Systems
- Vprover: Proves equivalence of Term Rewriting Systems
  - Iterative engine
  - Returns error trace if proof not found
  - Maintains an expanding rule base for expression minimization
  - Incomplete, but efficient engine

Results on Multipliers

Wallace Tree	VERIFIRE	Commercial Tool 1	Commercial Tool 2
4 X 4	14s	10s	9s
8 X 8	18s	18s	16s
16 X 16	25s	Unfinished	Unfinished
32 X 32	40s	Unfinished	Unfinished
64 X 64	60s	Unfinished	Unfinished

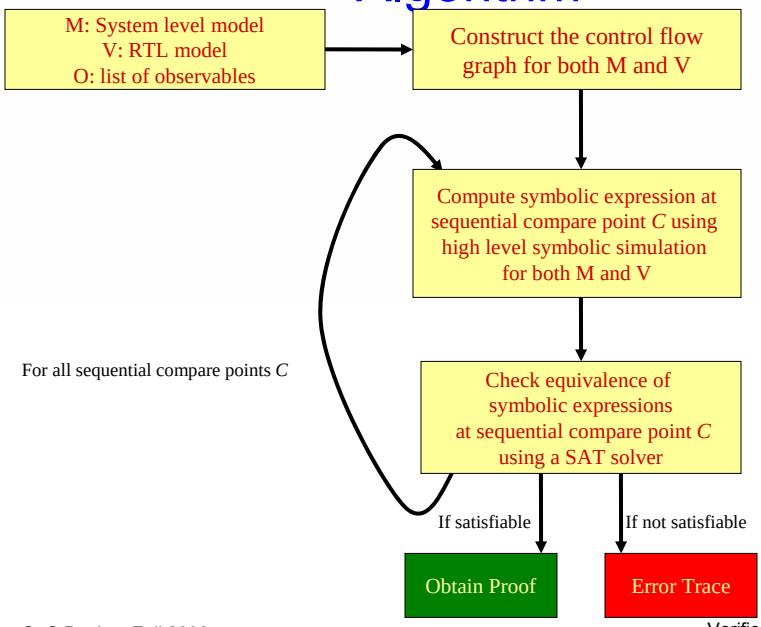
System Level Equivalence Checking

- Sequential equivalence checking
  - Verifying two models with different state encodings
- System specifications as system level model (SLM)
  - Higher level of abstraction
  - Timing aware models
- Design concept in RTL needs checking
  - Retiming, power, area modifications
  - Every change requires verification against SLM
- Simulation of SLM
  - Tedious to develop
  - Inordinately long running times

Equivalence Checking Using Sequential Compare Points

- Variables of interest (observables) obtained from user/block diagram
  - Primary outputs / Relevant intermediate variables
- Symbolic expressions obtained for observables assigned in a given cycle (high level symbolic simulation)
- Introduce notion of sequential compare points
  - Identification with respect to relative position in time
  - Identification with respect to space (data or variables)
- Symbolic expressions compared at sequential compare points
- Comparison using a SAT solver in this work
  - Other Boolean level engines can also be used

Algorithm



Correctness theorem

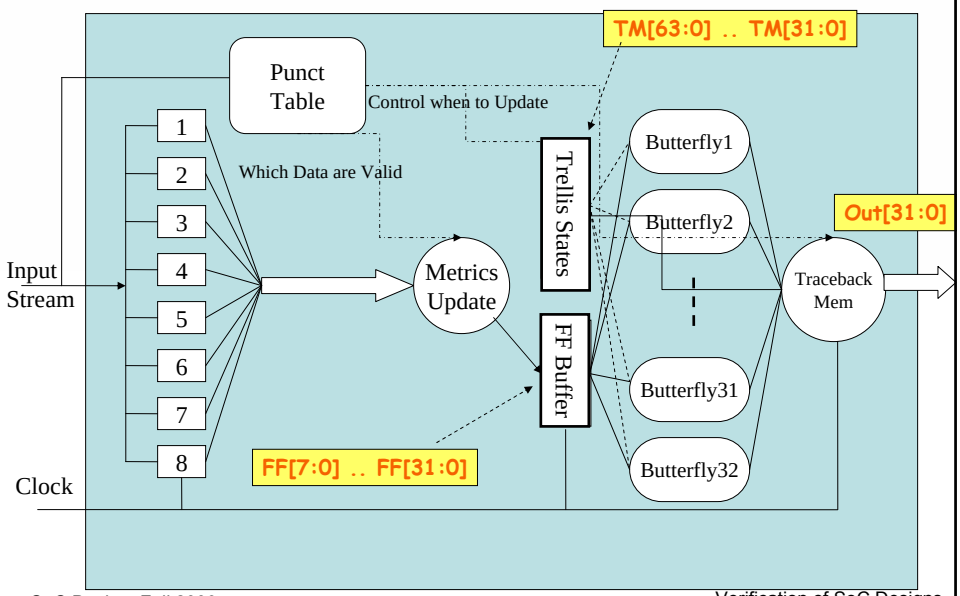
Theorem:

Let two systems *M* and *V* such that, *PI(M) = PI(V)* and *PO(M) = PO(V) = PO*.
Let *n* be the longest cycle length taken to obtain all primary outputs in both systems.
Let *M* and *V* be compared at every point *C = (t,d)* such that *t <= n*.
Let $\sim_c$ be the simulation relation that denotes the symbolic expression equality at *C*.
Then, for all *C*, $V \sim_c M \Rightarrow V \sim_{PO} M$.

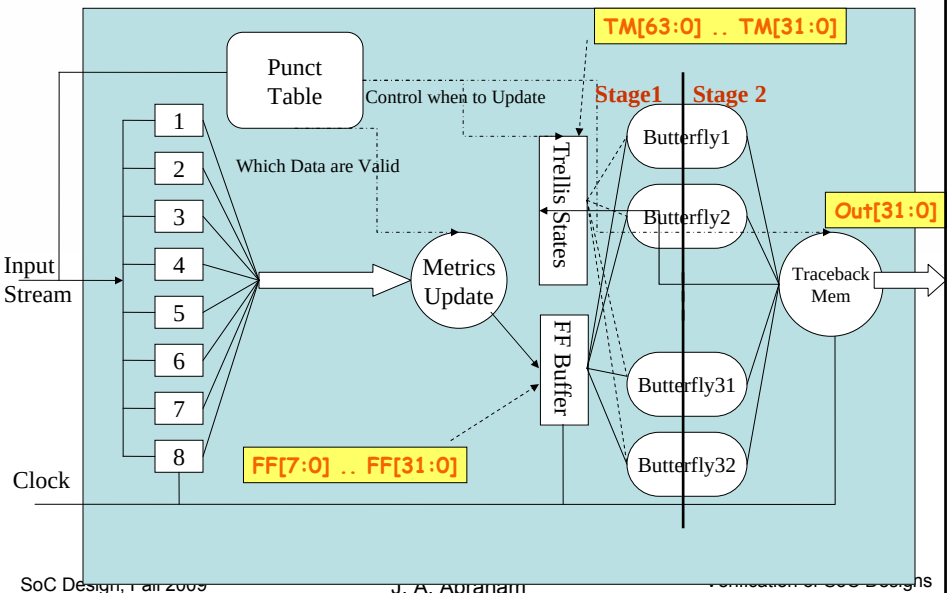
Proof intuition:

The base case is at time *t=0*, at initial state.
The induction hypothesis is relieved using a lemma that proves that
at any cycle *t*, if the two systems have
the same value for the symbolic expression of all variables *d*,
$\sim_c$ holds at that cycle.
If all primary outputs are generated by cycle *n*, the relation holds.

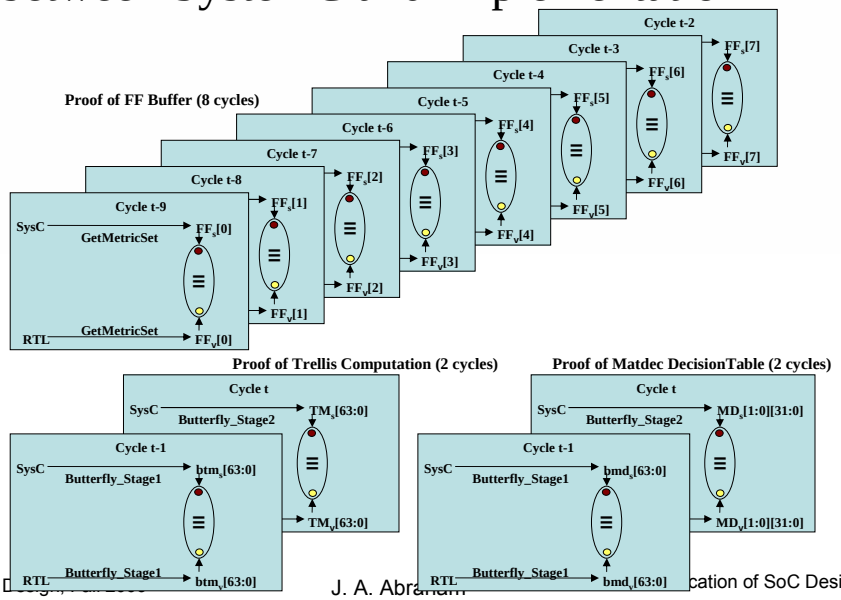
Viterbi Decoder: SystemC Specification



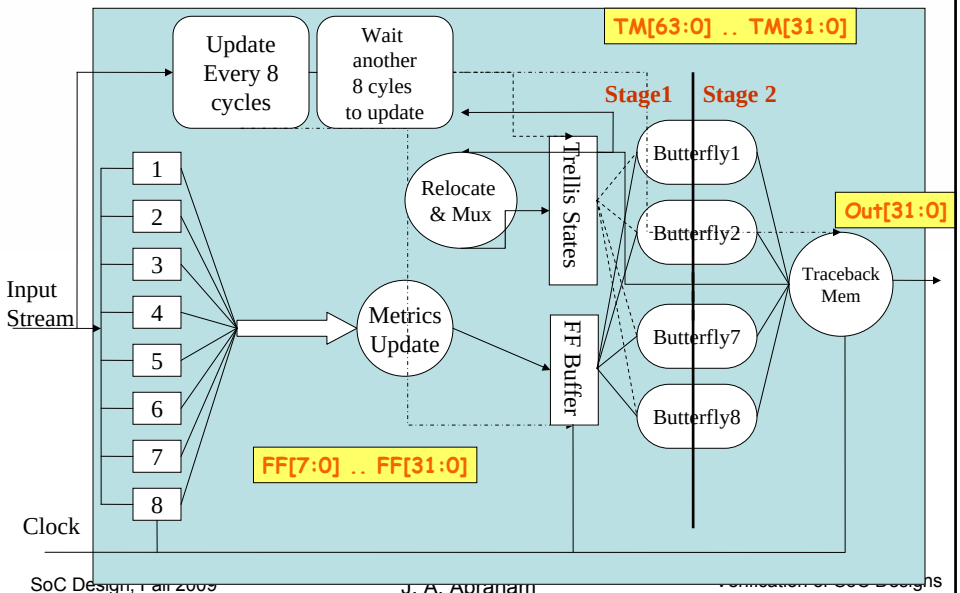
Viterbi Decoder Implementation 1



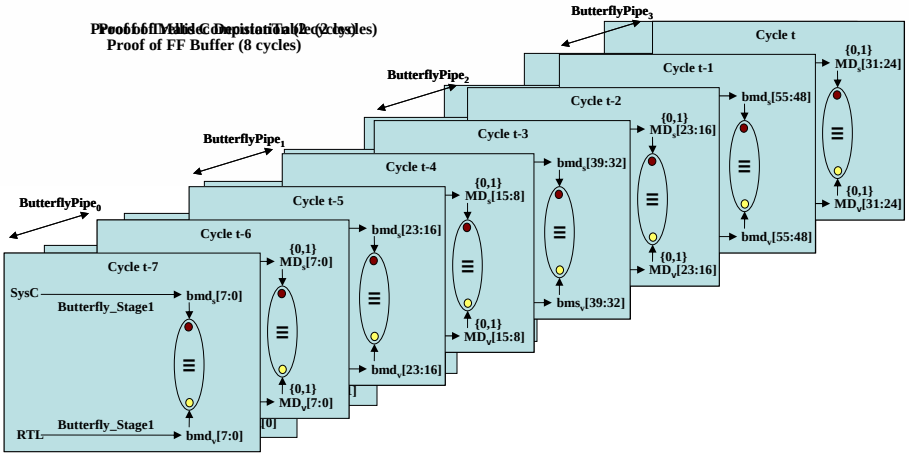
Decomposition of Equivalence Checking between SystemC and Implementation - 1



Viterbi Decoder Implementation 2



Decomposition of Equivalence Checking between SystemC and Implementation - 2



Results of using a SAT solver

Block/Function	Number of clauses in the CNF formula
PLUS	448
LESSTHAN	32
Trellis Condition in the butterfly	14336
Trellis computation in each stage of butterfly	28672
Trellis per butterfly	57344
MatDec each stage of butterfly	896
MatDec per butterfly	1792

Design	Number of clauses in the CNF formula
Monolithic Trellis	1892352
RTL decomposition (Design 1)	59136
RTL decomposition (Design 2)	59136

Results

Block/Function	Number of variables	Number of symbolic variables generated
PLUS	64	2
Butterfly	128	66
Trellis (monolithic)	2304	2112
Trellis (decomposed)	128	66

Verifying Embedded Software

- Software Testing
  - Execute software for test cases
  - Analogous to simulation in hardware
- Testing Criteria
  - Coverage measures
- Formal analysis of software
  - Model Checking
  - Theorem Proving

Path Testing

- Assumption: bugs affect the control flow
- Execute all possible control flow paths through the program
  - Attempt 100% path coverage
- Execute all statements in program at least once
  - 100% statement coverage
- Exercise every branch alternative during test
  - Attempt 100% branch coverage

Software Verification

- Formal analysis of code
- Result, if obtained, is guaranteed for all possible inputs and all possible states
- Example of software model checker:

  SPIN
- Problem: applicable only to small modules

) State Explosion

Data Abstractions

- Abstract data information
  - Typically manual abstractions
- Infinite behavior of system abstracted
  - Each variable replaced by abstract domain variable
  - Each operation replaced by abstract domain operation
- Data independent Systems
  - Data values do not affect computation
  - Datapath entirely abstracted

Data Abstractions: Examples

- Arithmetic operations
  - Congruence modulo an integer
    - *k replaced by k mod m*
- High orders of magnitude
  - Logarithmic values instead of actual data value
- Bitwise logical operations
  - Large bit vector to single bit value
    - *Parity generator*
- Cumbersome enumeration of data values
  - Symbolic values of data

Abstract Interpretation

- Abstraction function mapping concrete domain values to abstract domain values
- Over-approximation of program behavior
  - Every execution corresponds to abstract execution
- Abstract semantics constructed once, manually

Abstract Interpretation: Examples

- Sign abstraction
  - Replace integers by their sign
    - *Each integer K replaced by* one of {> 0, < 0, =0}
- Interval Abstraction
  - Approximates integers by maximal and minimal values
    - *Counter variable i replaced by lower and upper limits of loop*
- Relational Abstraction
  - Retain relationship between sets of data values
    - *Set of integers replaced by their convex hull*

Counterexample Guided Refinement

- Approximation on set of states
  - Initial state to bad path
- Successive refinement of approximation
  - Forward or backward passes
- Process repeated until fixpoint is reached
  - Empty resulting set of states implies property proved
  - Otherwise, counterexample is found
- *Counterexample can be spurious because of over-approximations*
- Heuristics used to determine spuriousness of counterexamples

Counterexample Guided Refinement

- Predicate Abstraction
  - Predicates related to property being verified (User defined)
  - Theorem provers compute the abstract program
  - Spurious counterexamples determined by symbolic algorithms
  - Some techniques use error traces to identify relevant predicates

Counterexample Guided Refinement

- Lazy Abstraction
  - More efficient algorithm
  - Abstraction is done on-the-fly
  - Minimal information necessary to validate a property is maintained
    - Abstract state where counterexample fails is “pivot state”
    - Refinement is done only “from the pivot state on”

Specialized Slicing for Verification

- Amorphous Slicing
  - Static slicing preserves syntax of program
  - Amorphous Slicing does not follow syntax preservation
  - Semantic property of the slice is retained
  - Uses rewriting rules for program transformation

Example of Amorphous Slicing

```
begin
   i = start;
   while (i <= (start + num))
        {
        result = K + f(i);
        sum = sum + result;
        i = i + 1;
        }
end
```

LTL Property: G sum > K
Slicing Criterion: (end, {sum, K})

Example of Amorphous Slicing

Amorphous Slice:

```
begin
   sum = sum + K + f(start);
   sum = sum + K + f(start + num);
end
```

Program Transformation rules applied

- Induction variable elimination
- Dependent assignment removal

- Amorphous Slice takes a fraction of the time as the real slice on SPIN

Amorphous Slicing for Verification

- Similar to term rewriting
  - Used by theorem provers for deductive verification
- What is different?
  - Theorem provers try to prove entirely by rewriting
  - Hybrid approach
    - Rewriting only part of the program, based on slicing criterion
    - Model checking the sliced program

Conditioned Slicing

- Theoretical bridge between static and dynamic slicing
- Conditioned Slices specify initial state in criterion
  - Constructed with respect to set of possible inputs
  - Characterized by first order predicate formula
- Yields much smaller slices than static slices

Example Results – Conditioned Slicing

- Group Address Registration Protocol (GARP) and X.509 authentication protocol
- SPIN model checker
  - Memory limit of 512 MB given
  - Max search depth of 2^{20} steps
- All properties were in the form

  Antecedent => Consequent

Experimental Results

Property	Unsliced*	Conditioned Sliced	Property Proved
P1	91.65	1.72	Yes
P2	145.78	8.44	Yes
P3	145.36	8.41	Yes
P4	154.96	1.95	Yes
P5	117.81	10.23	Yes

*Static slicing in SPIN was enabled