Formal Verification for VLSI Design

Introduction
Lecture 1

Paritosh Pandya

Tata Institute of Fundamental Research, Mumbai

May 2, 2018
Program Correctness

Engineering Design

- Performance  Functionality, Efficiency, Portability, Modifiability.
- Correctness  “Product functions reliably as intended.”

Correctness in Software

Improving Reliability of Programs (Software Metrics).

- Testing
- Code walkthrough
- Managing Software Development Process (CMM Maturity Levels)
- n-version programming

Software Crisis  Nato Software Engineering Conference in 1960s.
Correctness in Engineering

Basis of Correctness

- Mathematical Modelling of Designs
- Analysis and Synthesis Techniques
- Rigorous Mathematical Specification of Products
Correctness in Engineering

Basis of Correctness

- Mathematical Modelling of Designs
- Analysis and Synthesis Techniques
- Rigorous Mathematical Specification of Products

Software: Engineering or Craft?

- No rigorous specification of the intended behaviour of the system.
- No mathematical analysis of designs for correctness. Validation is by testing.
- Correctness is implicit and uncertain. Products are unreliable.
# State of Affairs

## Disasters:

- Intel Pentium SRT Division Bug (1995)  
  (Cost 1 Billion $).
- Arian 5 Launch Failure (1996)  
  (Cost 850 Million $).
- Mars Polar lander (1999) Incomplete requirements.
- Indian PSLV Failed Launch

## Changing Face of Hardware and Software

- Multi core processors, new memory models.
- New architectures: clusters, Grids, Multi-agent systems, Service oriented computing
- Concurrent asynchronous modules with mediated interaction
Mathematical Models for program behaviours.
Logical notations for specifying properties of programs.
Methods for checking that program meets its desired specification.

Verification Problem

To check whether $M \models \phi$
- $M$ system model in programming/modelling language.
- $\phi$ property in specification notation.

Must check that all behaviours of $M$ satisfy $\phi$. 
Reactive Systems

Systems with Digital/Computerized Controllers

- Digital Hardware
- Embedded System Controllers
- Network and Mobile Telephony Protocols
- Mobile Apps

Application Domains

- Aerospace
- Defense
- Robotics
- Consumer Gadgets: AC, Washing M/c, Camera, TV

Emerging Application Domains

- Internet of Things
- Wearable computers
- Smart homes
- Smart cities, transport ...
Reactive System Features

- **Reactive**: Output depends on interaction with environment.
- **Temporal**: Current output depends on past *sequence* of inputs.
- **Real Time**: Constraint on latency, response. Time triggered.
- **Global**: Output from a component may depend on all other components due to interaction.
- **Safety Critical**: Often used in applications requiring high degree of reliability.
- **Difficult to test**.
  - Failure is not repeatable.
  - Cannot collect enough test data.

*Reactive Systems are Complex Requiring Verification and Validation.*
Model Based Design

- **High level Model** of Controller/System behaviour.
  - Precise semantics
  - Should be easy to understand
- Concurrency (logical) and Interaction.
- Predictable: **Deterministic?**
- Executable

**Uses of Model**

- Simulation (Visualization) and Testing
- **Verification and Validation of Model** (**our focus**)
- Automatic Code Generation
  
  \[ Model \leq_{\text{refine}} \text{Implementation} \]
  
  Architecture dependent
  Correct by construction
In each cycle
- a subset of requests is high.
- one of the acks must be made high.
- **Mutex** At most one of the acks must be high.

\[ AG \bigwedge_{i \neq j} \neg (ack_i \land ack_j). \]
Synchronous Bus Arbiter

In each cycle
- a subset of requests is high.
- one of the acks must be made high.
- **Mutex** At most one of the acks must be high.

\[ AG \bigwedge_{i \neq j} \neg (ack_i \land ack_j). \]
In each cycle
- a subset of requests is high.
- one of the acks must be made high.
- **Mutex** At most one of the acks must be high.

\[ \text{AG } \bigwedge_{i \neq j} \neg (ack_i \land ack_j). \]
MacMillan’s Arbiter

In each cycle
- a subset of requests is high.
- **Mutex** At most one of the acks must be high.
- **Fairness** Every cell must get a chance!
In each cycle

- a subset of requests is high.
- **Mutex** At most one of the acks must be high.
- **Fairness** Every cell must get a chance!
In each cycle

- A subset of requests is high.
- **Mutex** At most one of the acks must be high.
- **Fairness** Every cell must get a chance!
In each cycle

- a subset of requests is high.
- **Mutex** At most one of the acks must be high.
- **Fairness** Every cell must get a chance!
Invariant Properties

- Mutual exclusion \( AG \bigwedge_{i \neq j} \neg(ack_i \land ack_j) \)
- No Spurious acknowledgements \( AG \bigwedge_i (ack_i \Rightarrow req_i) \)
- No Lost Cycles: \( AG \neg Lostcycle \) where 
  \[ Lostcycle \overset{\text{def}}{=} (\lor req_i) \land \neg(\lor ack_i) \]

Questions:

- How do we state these properties precisely?
  Answer: Use Propositional logic formulas to specify set of good/bad states.

- How do we verify these properties?
  Must ensure that under all possible inputs and at all steps invariant holds.
  This course will cover several algorithms for reachability/invariance checking.
Complex Properties

- **Fairness**: If request is kept continuously high, eventually there will be acknowledgment.

- **Timing Properties**: Time is measured in number of clock cycles and count of number of cycles for which a proposition holds.

- **3-cycle response time**: The minimum number of cycles for which $req_i$ must be continuously high to guarantee 3 $ack_i$ signals.

- **Dead-time**: The maximum number of consecutive lost cycles.

$$Lostcycle \overset{\text{def}}{=} (\lor req_i) \land \neg(\lor ack_i)$$
Temporal Logics: Stating Requirements

Linear Temporal Logic (*LTL*) (Pnueli 1977)
- **Formal notation** for unambiguous specification of properties of interest.
- Mathematical semantics.
- Specifies properties of an *execution* as a whole.
- Example: In any execution, at any cycle if *req* is true then *ack* is true at some later cycle.

\[ \Box (req \Rightarrow \Diamond ack) \]
- In this course, LTL and its extensions will be covered.

Founders of Temporal Logic and Model Checking

<table>
<thead>
<tr>
<th>Amir Pnueli</th>
<th>Ed Clark</th>
<th>Emerson</th>
<th>Sifakis</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Amir Pnueli" /></td>
<td><img src="image2.png" alt="Ed Clark" /></td>
<td><img src="image3.png" alt="Emerson" /></td>
<td><img src="image4.png" alt="Sifakis" /></td>
</tr>
</tbody>
</table>
Model checker is an algorithm which, given a system model $M$ and property $\phi$, determines whether $M \models \phi$.

A good model checker will produce a counter example if $M \not\models \phi$. 

System $M$ \hspace{1cm} Model Checker \hspace{1cm} Yes \hspace{1cm} No + Counter Example

Property $\phi$ \hspace{1cm} Yes \hspace{1cm} No + Counter Example
Model Checking: Verifying properties

- Model checker is an **algorithm** which, given a system model $M$ and property $\phi$, determines whether $M \models \phi$.
- A good model checker will produce a **counter example** if $M \not\models \phi$.

System M $\rightarrow$ Model Checker $\rightarrow$ Yes $\rightarrow$ No + Counter Example

Property $\phi$ $\rightarrow$ Model

**Applicability**
- Hardware verification
- Verification of Embedded systems controllers
- Protocol verification (networks, mobile telephony)
Automata as Models
Example: Two bit counter

Global State $s_1, s_0$

An Execution:

$\begin{array}{c}
0,0 \xrightarrow{c_0} 0,1 \xrightarrow{!c_0} 0,1 \xrightarrow{c_0/c_1} 1,0 \xrightarrow{c_0} 1,1 \xrightarrow{c_0/c_1, c_2} 0,0 \rightarrow \ldots
\end{array}$

Property: Invariantly $(c_2 \Rightarrow c_0)$
Global State Graph

\[ G = (S, R) \]
\[ S \text{ set of global states} \]
\[ R \text{ transition relation (edges)} \]

Property: Invariantly \( (c2 \Rightarrow c0) \)

Verification by **State Enumeration**: Search the whole state graph. Check that for each state/transition of the graph the property holds.
Asynchronous Concurrency with Shared Variables

Mutual Exclusion
Initially $y := 0$

Asynchronous parallelism
Guarded assignments.

$s, g \rightarrow \text{act, t}$
Properties

- **Mutual exclusion**: System will not reach a state where both processes are in critical region.

- In *each execution*, whenever either process is in critical region the value of $y = 0$.

- **Liveloak absense** In *each execution*, some process will access critical region infinitely often.

- **Starvation-freedom**: If process 1 is trying to enter the critical region, it will eventually succeed.
Properties

- **Mutual exclusion**: System will not reach a state where both processes are in critical region.
  \[ AG \neg (pc_1 = C \land pc_2 = C) \]

- In *each execution*, whenever either process is in critical region the value of \( y = 0 \).
  \[ AG ((pc_1 = C \lor pc_2 = C) \Rightarrow y = 0) \]

- **Livelock absense** In *each execution*, some process will access critical region infinitely often.
  \[ \square \Diamond (pc_1 = C \lor pc_2 = C) \]

- **Starvation-freedom**: If process 1 is trying to enter the critical region, it will eventually succeed.
  \[ \square (pc_1 = T \Rightarrow (\Diamond pc_1 = C)) \]
Global Automaton for Mutual Exclusion Problem

The diagram represents the states and transitions of a global automaton for the mutual exclusion problem. The states are labeled with triplets (P, Q, R) where P, Q, and R represent different conditions or variables. The arrows indicate the transitions between states, and the state labels follow the pattern: 

- N,N,1
- T,N,1
- N,T,1
- C,N,0
- T,T,1
- N,C,0
- C,T,0
- T,C,0

These transitions illustrate the dynamic behavior of the automaton in response to changes in the system's conditions.
Aim: To prove $M \models AG P$

- Construct Global state graph of $M$
- Check that all reachable states satisfy $P$.

**Explicit state enumeration method**

- Explicitely construct global state graph in computer memory. Search state graph for invariance violation.
- Useful for small state graphs.
- **Disadvantage:** Number of states in global state graph grows exponentially with number of interacting components. State space explosion problem.
Efficient representation and exploration of explicit state graphs.

- on-the-fly construction of state graphs
- partial-order reductions
- symmetry reduction
- hashing: one bit per state!!
- efficient cycle detection

Can typically explore systems with $10^6$ to $10^9$ states.

Given **ACM systems software award** for year 2001.

- for developing system software that has lasting influence reflected in concepts, commercial aspects or both.
Symbolic Model Checking

Can very often explore systems with $10^{120}$ states.

**Technique:** Represent set of states by boolean formulae. Iteratively compute the set of reachable states.

$S_i$: set of states reachable from $Init$ in $i$ or less steps.

$$S_{i+1} = S_i \cup F(S_i)$$

Chain $S_0 = Init \subset S_1 \subset \ldots \subset S_m = S_{m+1}$

$S_m$ is the set of reachable states:

$$M \models invariants(P) \text{ iff } S_m \subset P.$$
State of the Art

Initial Idea: [Bryant, MacMillan] BDDS, CTL model checking by iterative computing of fixed points.
SAT solving [Mallik (Princeton)].

Advanced techniques:
- Partitioning and image computation
- Variable ordering (BDD Heuristics)
- Bounded Model Checking
- SAT solving.
- Abstraction and Induction
- Rich specification languages

Leading Symbolic Model Checkers from Universities
- Z3 (Microsoft), SMV (CMU), ABC and VIS (UC Berkeley, UC Colorado), NuSMV (FBK, Trento), EUCLID (CMU), ICS (SRI, Stanford)
Diversity of Modelling Language

- Hardware Description Languages: Verilog, AIGER (ABC, VIS).
- Symbolic Transition Systems (SMV, SAL)
- Synchronous Data Flow Networks (Lustre/SCADE)
- State Charts
- Multi-Threaded C-like programs with Interaction (SPIN)
State of the Art

**Initial Idea:** [Bryant, MacMillan] BDDS, CTL model checking by iterative computing of fixed points.

SAT solving [Mallik(Princeton)].

**Advanced techniques:**
- Partitioning and image computation
- Variable ordering (BDD Heuristics)
- Bounded Model Checking
- SAT solving.
- Abstraction and Induction
- Rich specification languages

**Leading Symbolic Model Checkers from Universities**
- Z3 *(Microsoft)*
- SMV *(CMU)*
- NuSMV *(IRST, Trento)*
- ABC *(UC Berkeley, UC Colorado)*
- UCLID *(CMU)*
- SAL *(SRI, Stanford)*
Industrial Use

Hardware and Computer Architecture Companies have inhouse model checking tools. (E.g. Microsoft, Intel, IBM, Motorola, Siemens, Cadence, Synopsis.) IEEE standard Temporal Logic *PSL/SUGAR*.

- Intel announced formal verification of floating point unit of Pentium Pro processor (1999). Possibly used formal verification to check parts of design of Pentium IV processor.
- Siemens used Equivalence Checking in validation of ASICs with upto a Million Gates.
- Microsoft provides driver development kit where every third party driver is model checked for health.
- Intel, Motorola, IBM routinely use Model Checking in their design process.

**Some model checkers from India**

- DCVALID *(TIFR)* (Released since 1997),
- Word Level ABC verifier *(IITB)* (Web interface).
- Bebop and SLAM *Microsoft Research India*
Thank You.