Model Checking

Supratik Chakraborty

IIT Bombay
Why Model Check?

- Widespread use of complex hardware systems
  - Incorrect operation highly undesirable
    - Safety, economic … reasons
- Can we mathematically prove that a model of the system satisfies a formal property?
- Large class of practical systems
  - System model: Finite state
  - Property: Temporal logic formula

Above question answerable automatically
Method involved: Model checking
A Glimpse of Capability

- PCI Bus
  - 3 peripherals, 1 arbiter
  - Peripheral: Application controller + Bus interface controller
  - Arbiter: Fixed arbitration scheme
  - Bus: Negligible delays
- Global behaviour
  - Composition of component FSMs
- Properties verifiable using model checking:
  For arbitrary sequences of bus transactions
  - Only one peripheral can be master at any time
  - A peripheral requesting to be master eventually becomes one...
Outline

- What is Model Checking?
- Modeling basics
- Property specification in temporal logic
- CTL model checking
- Related techniques
- Conclusion
What is Model Checking?

- **Given:**
  - Finite state model of system: \( M \)
    - An abstraction of system behaviour
  - A temporal logic formula: \( \varphi \)
    - Specification of desired property

- **Determine:**
  - Does \( M \) satisfy \( \varphi \)? \( M \models \varphi \)?
    - Logic vocabulary: “Is \( M \) a model of \( \varphi \)?”
    - Hence “model checking”
      - Yes \( \Rightarrow \) System model exhibits specified property
      - No \( \Rightarrow \) System model violates property
        - Counterexample trace desirable
Model Checking Basics

- Three essential components
  - Formalism for system modeling ($M$)
  - Formalism for property specification ($\varphi$)
  - Automatic decision procedure to check if $M \models \varphi$

- Additional desirable features for practical use
  - Counterexample generation in case $M \not\models \varphi$
    - Useful for debugging, understanding errors
  - Intuitive (at least partial) nature of formalisms
  - Computational efficiency
Outline

- What is Model Checking?
- Modeling basics
- Property specification in temporal logic
- CTL model checking
- Related techniques
- Conclusion
Modeling System Behaviour

- Model: Abstraction of actual behaviour
- Choice of right level of abstraction important

System

Behaviour at low abstr level

Behaviour at higher abstr level
Modeling System Behaviour

- Choosing right level of abstraction
  - Depends on property of interest
  - Model must have details relevant for checking property
  - Details unrelated to property to be abstracted away

Property: When \( c \) is at 0.5V, \( \text{clk} \) can’t be at 1V

Property: \( a, b, c \) have logic 1 value infinitely often
Modeling System Behaviour

- Level of abstraction must be judiciously chosen

- A model is not the same as the original system
  - Model checking determines if the model satisfies a property
    - Distinct from checking if actual system satisfies property
  - With incorrect models, spurious conclusions about system behaviour may result
  - Choosing a correct model for checking a property crucial
Modeling In Design Flow

- A (pre-) design activity
  - Start with highest level model (ignoring most impl. details)
  - Model check against desired properties
    - Allows early detection of bugs
    - Exposes design errors even before implementation
    - Allows easier exploration of design space
  - If no bugs found
    - Refine model to next lower level
      - Introduce some more impl details
  - If bugs found
    - Fix bugs at current level of abstraction
    - Repeat model checking step & refine until impl obtained
Formal Modeling

- Formal Model:
  - Description of behaviour in a formal language
  - Well-defined syntax
  - Mathematically precise semantics

- Examples of formal modeling languages:
  - Finite State Machines
  - CSP, CCS, SDL, Promela, SMV (Async systems/protocols)
  - SMV, mv_blif, Verilog, VHDL (Synchr sequential circuits)
  - Esterel, Signal (Embedded controllers)
  - Statecharts, UML (System level modeling) ...
Formal Modeling

- Features of a good modeling language
  - High level data types
  - Ability to represent non-determinism & concurrency
  - Ability to model communication between components
  - Ability to represent fairness in execution

- We’ll focus on conceptually simplest formalism

  **Finite state machines**

  - Many variants
    - Edge labeled - Mealy machines
    - State labeled - Kripke structures
    - State and edge labeled - Moore machines
Models of Hardware Circuits

- Hardware blocks are reactive systems
  - Continuous interaction with environment
  - Behaviour never terminates; infinite behaviour
  - Timing and causality information important
  - Different from transformational systems

- Modeling with FSMs:
  - States determined by values of latches and flip-flops
  - Finite no. of latches / flip-flops
    - Finite no. of states: Could be very large though!
  - No final state; no termination of behaviour
  - Edge labels: input/condition and output/action
An Example

- A Traffic Light Controller

States: Highway green, side road red
- C - Car in side road,
- S, L - Short and long timer signals
- T, G, R - reset timer, set highway green and side road red
Non-determinism

- reqi - request from Master i
- Abstract model is nondeterministic
  - In Idle state when req1 and req2 arrive
- Non-determinism due to abstraction
- More than one behaviour for a given input: req1, req2
Concurrency

- A concurrent (and hierarchical) description of 3-bit counter (Statechart syntax)
Concurrent Descriptions

- Compact and easy to understand
- Natural model for hardware and complex systems
- Clear semantics required
  - Interleaved model and synchronous models
- Appropriate communication primitives
- Concurrent machines composed to yield a single global machine
- Global machine captures all possible executions
  - Exponential blow-up
Fairness Constraints

- Not every run of FSM model may be valid
- Example:
  - Abstract arbiter
    - Run that never grants request of master 2
- Eliminate such runs without complicating model
- Fairness constraints rule out certain runs
  - Models effect of schedulers

Fairness constraints:
- Every request eventually considered
- chg arrives infinitely often
Fairness Constraints

- Not required with a more concrete description
- But concrete description too complex to verify
- A given property may not require concrete details
- For verification, abstract designs are preferable
  - Proof is simpler
  - Proof is robust under alternate implementations
  - Fairness constraints eliminate invalid runs without complicating model
Semantics of Finite State Systems

- Semantics associates behaviors
  - Branching Time semantics
    - Tree of states obtained by unwinding the state transition graph
Semantics of Finite State Systems

- Linear Time Semantics
  - Set of all possible `runs' of the system

State transition graph

Set of infinite runs
Generating Formal Models

- Automatic translation from circuits/HDL descriptions
  - States decided by the latches/registers in the circuit
  - Exponential blow-up in size (state-explosion problem)
  - Usually abstractions required
- Tools automatically extract finite state model from high-level descriptions
  - VHDL, Verilog
  - SMV
  - mv_blif ....
Outline

- What is Model Checking?
- Modeling basics
- Property specification in temporal logic
- CTL model checking
- Related techniques
- Conclusion
Formal Specifications

- Model checking:
  - Does a design model meets its specification?
- **Specification** states **what** the system is supposed to do
- **Model** describes **how** this is done

**Specification**
- Describes unambiguously and precisely the expected behavior of a design
- In general, a list of properties
- Includes environment constraints
- Symbolic logic or automata formalisms
- Consistency and Completeness
Specification of Hardware blocks

- Properties and constraints specify possible states and transitions
  - They state set of possible valid `runs'
- Valid runs:
  - Infinite sequences or trees of states
- Formal specifications:
  - Properties over infinite sequences or trees

Classification of Properties:

- Safety properties
  - "undesirable states are never reached",
  - "desirable things always happen".
- Progress or Liveness Properties
  - "desirable state repeatedly reached"
  - "desirable state eventually reached"
Examples

Safety Properties
- A bus arbiter never grants requests to two masters
- Message received is always the message sent
- Elevator does not reach a floor unless it is requested
- At any time traffic is let either in the farm road or in the highway
- Every received message was actually sent

Liveness Properties
- Car on farm road is eventually allowed to pass
- Elevator attends to every request eventually
- Every bus request is eventually granted
- Every sent message was received
Behaviours are infinite runs or trees (reactive systems)

Need finite representation of such infinite objects

Two major formalisms:
- Temporal Logics:
  - Linear and Branching Time Temporal Logics
- Automata
Temporal Logics

- Logics well-known for precise specification
  - Amenable to symbolic manipulations
- Used in a variety of contexts:
  - Propositional Logic/Boolean algebra for combinational HW
  - Predicate logic for software
  - Higher order logics for language semantics
  - Temporal logics for hardware and protocols
- Qualitative temporal statements
- Examples:
  - If it is cloudy, eventually it will rain
  - It never rains here
Properties of Hardware Blocks

- Temporal in nature
  - At any time only one unit is accessing the bus
  - Every request to access the bus is eventually granted.

- Two Kinds of Temporal Logics
  - Linear Temporal Logic (LTL):
    - Time is a linear sequence of events
  - Branching time temporal logic (CTL, CTL*):
    - Time is a tree of events
Computational Tree Logic (CTL)

- CTL formulae describe properties of Computation Trees
- Computation Trees are obtained by unwinding the transition system model
- Branching structure due to nondeterminism
- CTL is the simplest branching temporal logic
- CTL* is more powerful, subsumes CTL
  - Won’t cover in this tutorial
Every atomic proposition is a CTL formula

If \( f \) and \( g \) are formulae then so are

\[ \neg f, \ (f \land g), \ (f \lor g), \ (f \rightarrow g), \ (f \leftrightarrow g) \]

\( AG \ f : \) in all paths, in all states \( f \) holds (in all future \( f \))

\( EG \ f : \) in some path, in all states \( f \) holds

\( AF \ f : \) in all paths, in some state \( f \) holds (in every future \( f \))

\( EF \ f : \) in some future, \( f \) holds

\( A(f \ U g) : \) in all paths, \( f \) holds until \( g \)

\( E(f \ U g) : \) in some path, \( f \) holds until \( g \)

\( AX \ f : \) in every next state, \( f \) holds

\( EX \ f : \) in some next state, \( f \) holds
Examples

- $AG \neg (farm\_go \land high\_go_B)$
- $AGAF (farm\_car \rightarrow AF(farm\_go))$
- $AG (mem\_wr U mem\_ack)$
- $EF (req_0 U grant_0)$
Outline

- What is Model Checking?
- Modeling basics
- Property specification in temporal logic
- CTL model checking
- Related techniques
- Conclusion
Outline

- What is Model Checking?
- Modeling basics
- Property specification in temporal logic
- CTL model checking
  - Explicit-state model checking
  - Symbolic model checking
    - Boolean function representation
    - Model checking algorithm
- Related techniques
- Conclusion
CTL model checking

[Clarke and Emerson, Clarke, Emerson and Sistla, Quielle and Sifakis]

- Given
  - Finite state transition system (Kripke structure) $M$
  - CTL formula $\varphi$

- There is an $O(|M|.|\varphi|)$ algorithm to check if $M \models \varphi$
EXAMPLE

Which of the following hold?

- $AG\ p$, $EF\neg q$, $AX\ p$, $AG\ \neg q$, $EG\ \neg q$, $AX(p \lor q)$
Explicit-state CTL Model Checking

- Iterative labeling algorithm that labels all the states with sub formulae
- Start from initial labels: atomic propositions
- Iteratively add sub formulae as labels of each state based on the following equations:

\[
EF \ p = p \lor EX \ p \lor EX(EX \ p) \lor \ldots \\
EG \ p = p \land EX \ p \land EX(EX \ p) \land \ldots \\
E (q \ U \ p) = p \lor (q \land EX \ p) \\
\quad \lor (q \land EX(q \land EX \ p)) \\
\quad \lor \ldots 
\]
CTL Model Checking

- Iteration terminates since states and subformulae are finite.
- If initial states of model are labeled with the given formula then model checking succeeds
- If it fails, counterexample can be generated
Computing $EF \ p$

To compute $EF \ p$ which is:

$EF \ p = p \lor EX(p) \lor EX(EX(p)) \lor \ldots$
Computing EF p

Iterative computation

- I step:
Computing EF p

- II step:

- III step:
Computing EF p

- Computation terminates
- $EF \ p$: Holds in all striped states
- Computation involves backward breadth first traversal and calculation of Strongly Connected Subgraphs (cycles)
Computing $EG \ p$

$EG \ p = p \land EX \ p \land EX(EX \ p) \land \ldots$
Computing $EG \, p$

Start with

I iteration
Computing EG \( p \)

II iteration

III iteration

Iteration terminates
Is This Sufficient Enough?

- Algorithm involves backward traversal
- Linear on sizes of both formula and model
- Size of model: Exponential in number of latches !!!
  - Small sized systems verifiable
- Reduction techniques:
  - Symbolic model checking
  - Compositional verification
  - Symmetry based reduction
  - Partial order reduction
Explicit vs Symbolic Model Checking

- Explicit state model checking
  - Requires explicit enumeration of states
  - Impractical for circuits with large state spaces
  - Useful tools exist: EMC, Murphi, SPIN, SMC …

- Symbolic model checking
  - Represent transition relations and sets of states implicitly (symbolically)
  - BDDs used to manipulate implicit representations
  - Scales well to large state spaces (few 100 flip flops)
  - Fairly mature tools exist: SMV, VIS, FormalCheck …
Outline

- What is Model Checking?
- Modeling basics
- Property specification in temporal logic
- **CTL model checking**
  - Explicit-state model checking
  - **Symbolic model checking**
    - Boolean function representation
    - Model checking algorithm
- Related techniques
- Conclusion
Boolean Function Representation

- Boolean logic: Foundation of reasoning strategies
- Need to represent and manipulate Boolean functions efficiently
- Common representations:
  - Truth table, Karnaugh map, Canonical sum-of-products
    - Size always $2^n$ for n-arguments
    - Operations (e.g. AND, NOT) inefficient
    - Inappropriate for practical applications
      - E.g., representing carry-out function of 64-bit adder
Binary Decision Diagrams (BDDs)

- A graphical representation [Bryant '96]
  - Allows efficient representation & manipulation of Boolean functions
  - Worst-case behavior still exponential

- Example: \( f = x_1.x_2 + x_3' \)
  - Represent as binary tree
  - Evaluating \( f \):
    - Start from root
    - For each vertex \( x_i \)
      - left branch if \( x_i = 0 \)
      - else right branch
BDDs

- Underlying principle: Shannon decomposition
  - \( f(x_1, x_2, x_3) = x_1.f(1, x_2, x_3) + x_1'.f(0, x_2, x_3) \)
    - \( = x_1. (x_2 + x_3') + x_1'. (x_3') \)
  - Apply recursively to \( f(1, x_2, x_3) \) and \( f(0, x_2, x_3) \)
  - Extend to \( n \) arguments

- Number of nodes can be exponential in number of arguments

\[ f = x_1.x_2 + x_3' \]
Restrictions on BDDs

- Ordering of variables
  - In all paths from root to leaf, variable labels of nodes in specified order
- Reduced graphs
  - No two distinct vertices represent same function
  - Each non-leaf vertex has distinct children

REDUCED ORDERED BDDs (ROBDDs): DAG

\[ f = x_1' \cdot x_2' + x_1 \cdot x_2 + x_1 \cdot x_3' \]
ROBDDs

- Example: \( f = x_1 \cdot x_2 + x_3' \)

- Properties
  - Unique representation of \( f \) for given variable ordering
    - Checking \( f_1 = f_2 \): ROBDD isomorphism
  - Shared subgraphs: size reduction
  - Every path might not have all labels
  - Every non-leaf vertex has path(s) to 0 and 1

So far good!
Variable Ordering Problem

\[ f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 \]

Order 1,3,5,2,4,6

Order 1,2,3,4,5,6
Variable Ordering Problem

- ROBDD size extremely sensitive to variable ordering
  - $f = x_1.x_2 + x_3.x_4 + \ldots + x_{2n-1}.x_{2n}$
    - $2n+2$ vertices for order 1, 2, 3, 4...2n-1, 2n
    - $2^{n+1}$ vertices for order 1, n+1, 2, n+2,...n, 2n
  - $f = x_1.x_2.x_3\ldots.x_n$
    - n+2 vertices for all orderings
- Output functions of integer multipliers
  - Exponential size for all orderings [Bryant ‘91]
Variable Ordering Problem

- Determining best variable order to minimize BDD size
  - NP-complete [Bollig, Wegener ‘96]
- Heuristics:
  - Static and dynamic ordering [Fujita et al ‘92, Rudell ‘93]
  - Sampling based schemes [Jain et al ‘98]
## Operations on BDDs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce</td>
<td>$O(</td>
</tr>
<tr>
<td>G reduced to canonical form</td>
<td></td>
</tr>
<tr>
<td>Apply</td>
<td>$O(</td>
</tr>
<tr>
<td>Any binary Boolean op: AND, XOR …</td>
<td></td>
</tr>
<tr>
<td>Compose</td>
<td>$O(</td>
</tr>
<tr>
<td>$g_1(x_1, x_2, x_5)$ composed with $g_2(x_3, x_4)$ at position of $x_2$: $g_1(x_1, g_2(x_3, x_4), x_5)$</td>
<td></td>
</tr>
</tbody>
</table>
Operations on BDDs (Contd.)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satisfy-one</td>
<td>O(n)</td>
</tr>
<tr>
<td>Assignment of $x_1, \ldots x_n$ for which $f(x_1, \ldots x_n) = 1$</td>
<td></td>
</tr>
<tr>
<td>Restrict</td>
<td>O(</td>
</tr>
<tr>
<td>ROBDD for $f(x_1, x_2, \ldots, 1, \ldots x_n)$ or $f(x_1, x_2, \ldots 0 \ldots x_n)$</td>
<td></td>
</tr>
</tbody>
</table>
Operations on BDDs

- Operators: Take ROBDD arguments, return ROBDD result.

- Complexity polynomial in BDD size
  - BDD size limiting factor in most applications
  - Ongoing research on avoiding BDD blowup
    - Variable ordering, Partitioned BDDs, Implicitly conjoined BDDs etc.

- Quantification with BDDs
  - \( \forall x_1. f(x_1, x_2, x_3) = f(0, x_2, x_3) \cdot f(1, x_2, x_3) \)
  - \( \exists x_1. f(x_1, x_2, x_3) = f(0, x_2, x_3) + f(1, x_2, x_3) \)
  - Useful in Symbolic Model Checking
BDD Packages/Libraries Out There

- CUDD package (Colorado University)
  http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html
- Carnegie Mellon BDD package
- TiGeR BDD library (commercial package)
- CAL (University of California, Berkeley)
  http://www-cad.eecs.berkeley.edu/Respep/Research/bdd/cal_bdd/
- BuDDy
  http://www.itu.dk/research/buddy
BDD Packages/Libraries Out There

- ABCD
- BDDNOW
  - http://www.diku.dk/students/lordtime/bddnow.tar.gz
- PPBF
  - http://www-2.cs.cmu.edu/~bwolen/software/ppbf/
- ...

Outline

- What is Model Checking?
- Modeling basics
- Property specification in temporal logic
  - **CTL model checking**
    - Explicit-state model checking
    - **Symbolic model checking**
      - Boolean function representation
      - Model checking algorithm
- Related techniques
- Conclusion
Symbolic Model Checking of Hardware Sequential Circuits

- **Given:**
  - A sequential circuit
    - Finite state transition graph
    - Flip-flops with next-state logic
    - Transition relation between present and next states
  - A property in temporal logic

- **Prove that** MODEL satisfies SPECIFICATION
  - In case of failure, counterexample desirable
Example: 3-bit Counter

Model
State transition graph defined by
\[ X_0 = \text{NOT}(x_0) \]
\[ X_1 = \text{XOR}(x_1, x_0) \]
\[ X_2 = \text{XOR}(x_2, x_0 \cdot x_1) \]

Property
State \( x_0, x_1, x_2 = 111 \) is reached infinitely often starting from state 000
Symbolic Model Checking

- Reachability analysis
  - Find all states reachable from an initial set $S_0$ of states
  - Check if a safety condition is violated in any reachable state

- CTL property checking
  - Express property as formula in Computation Tree Logic (CTL)
  - Check if formula is satisfied by initial state in state transition graph
Symbolic Model Checking

- For 3-bit counter, set of states $x_0, x_1, x_2 = \{000, 010, 011, 001\}$ can be represented by $S(x_0, x_1, x_2) = S(x) = x_0'$.

BDD:

- Set of state transitions can be represented by $N(x_0, x_1, x_2, X_0, X_1, X_2) = N(x, X) = (X_0 \Leftrightarrow x_0') \land (X_1 \Leftrightarrow x_1 \oplus x_0) \land (X_2 \Leftrightarrow x_2 \oplus (x_1 \land x_0))$.

BDD:
Forward Reachability

- Start from set S0 of states
- Set of states reachable in at most 1 step:
  \[ S1 = S0 \cup \{ X \mid \exists x \text{ in } S0 \land N(x, X) = 1 \} \]

Expressed as Boolean functions:
- Given \( S0 (x_0, x_1, x_2) \),
  \[ S1 (X_0, X_1, X_2) = S0 (X_0, X_1, X_2) \lor \exists x_0, x_1, x_2. [S0 (x_0, x_1, x_2) \land N(x_0, x_1, x_2, X_0, X_1, X_2)] \]

*Given BDDs for \( S0 \) and \( N \), BDD for \( S1 \) can be obtained*
Forward Reachability

- Compute S1 from S0, S2 from S1, S3 from S2, ...
  - Predicate transformer F: $S_{i+1} = F(S_i)$
- Continue until $S_{k+1} = F(S_k) = S_k$
  - Least fixed point of F
- $S_k = \text{Set of all states reachable from S0}$
  - Computed symbolically -- using BDDs
- Very large state sets can be represented compactly
Backward Reachability

- Give a set $Z_0$ of states
  - Compute set of states from which some state in $Z_0$ can be reached.
  - Analogous to forward reachability with minor modifications
Checking Safety Conditions

- Safety condition must **ALWAYS** hold
  - E.g. Two bits in 1-hot encoded state can’t be 1 simultaneously
- \( Z \) = set of states violating safety condition
- Given \( S_0 \) = set of initial states of circuit,
  - Compute \( R \) = set of all reachable states
  - Determine if \( Z \) intersects \( R \), i.e. \((Z \land R) \neq 0\)
    - If YES, safety condition violated
      - Satisfying assignment of \((Z \land R)\): counterexample
    - If NO, circuit satisfies safety condition
- All computations in terms of BDDs
Checking Safety Conditions

- Start from $Z =$ set of “bad” states
- Find by backward reachability set of states $B$ that can lead to a state in $Z$
- Determine if $S_0$ intersects $B$

**Forward Reachability**

**Backward Reachability**
CTL Properties

- “Once req goes high, grant eventually goes high”
  - Not expressible as safety property
- Use formulae in Computation Tree Logic (CTL)
- CTL formulae at state S0
  Atomic proposition: $x_1 = x_2 = x_3 = 0$
  AG f: In all paths from S0, f holds globally
  AF f: In all paths from S0, f holds finally
  AX f: In all paths from S0, f holds in next state
  A[f U g]: In all paths from S0, g holds finally, and f holds until then
Recap of CTL

- EG f, EF f, EX f, E [f U g] defined similarly
  - “There exists a path from current state …”
  - f and g can themselves be CTL formulae
  - E.g., AG AF (x1 \lor x2)
    - x1 or x2 is satisfied infinitely often in the future

- Recall 3-bit counter example:
  - “The state \( x0, x1, x2 = 111 \) is reached infinitely often starting from 000”
  - \( \neg x0 \land \neg x1 \land \neg x2 \rightarrow AG AF (x0 \land x1 \land x2) \)
Symbolic CTL Model Checking

- Burch, Clarke, Long, McMillan, Dill gave algorithm for CTL model checking with BDDs [Burch et al’94]
- Suffices to have algorithms for checking $EG f$, $EX f$, and $E [f U G]$
  - Other formulae expressed in terms of these
    - $EF f = E [true U f]$
    - $AF f = \neg (EG (\neg f))$
Symbolic CTL Model Checking

- Given a model with set S0 of initial states and a CTL formula f
  - To determine if f is satisfied by all states in S0
- Convert f to g that uses only EX, EG, E[p U q]
- CHECK(g) returns set of states satisfying g
  - If g = atomic proposition (e.g., x1. x2 + x3), CHECK returns BDD for g
  - If g = EX p, EG p, E[p U q], CHECK uses reachability analysis to return BDD for set of states
  - Worst-case exponential complexity
- Finally, determine if S0 ⊆ CHECK(g)
State of the Art

- Techniques to address memory/runtime bottlenecks
  - Partitioned transition relations
    - Addresses BDD blowup in representing transitions
  - Early quantification of variables
    - Addresses BDD blowup during image computation
  - Iterative squaring
    - Exponential reduction in number of steps to fixed point
State of the Art

- Techniques to address memory/runtime bottlenecks (contd.)
  - Use domain knowledge to order BDD variables and order quantified variables
  - Modified breadth first search
    To explore state space of loosely coupled systems
  - Active ongoing research …
State of the Art

- Symbolic model checkers can analyze sequential circuits with ~ 300 flip flops
  - For specific circuit types, larger state spaces have been analyzed
  - Frontier constantly being pushed
  - Abstract, Avant!, IBM, Cadence, Intel. Fujitsu, Motorola (internal) ...
State of the Art

- Specifying properties in specialized logic often daunts engineers
  - Better interfaces needed for property specification
- Monitor-based model checking
  - Monitor observes system states and flags when something “bad” happens
  - Property to check: “Does monitor ever raise flag?”
Outline

- What is Model Checking?
- Modeling basics
- Property specification in temporal logic
- CTL model checking
- Related techniques
- Conclusion
Related techniques

- Model checking for bugs
  - Prioritize state space search to direct it towards bugs
    - Start from error state and current state
    - Compute pre-image of error states & image of current state
    - Choose states for further expansion in order of their “proximity” to pre-image of error states
      - Proximity metrics: Hamming distance, tracks, guideposts [Yang, Dill ‘98]
  - Helps find bugs in erroneous circuits quickly
  - No advantages if circuit is bug-free
Related techniques

- Approximate Model Checking
  Representing exact state sets may involve large BDDs
  Compute approximations to reachable states
    - Potentially smaller representation
    - Over-approximation:
      - No bugs found \(\Rightarrow\) Circuit verified correct
      - Bugs found may be real or false
  - Under-approximation:
    - Bug found \(\Rightarrow\) Real bug
    - No bugs found \(\Rightarrow\) Circuit may still contain bugs
Related techniques

- Bounded model checking
  - Check property within k steps from given set $S_0$ of states
  - $S_0 \rightarrow F(S_0) \rightarrow F^2(S_0) \rightarrow \ldots \rightarrow F^k(S_0)$
  - Unroll sequential machine for k time steps

- To check property $Z$, test satisfiability of
  $$(S_0 \land Z) \lor \neg(S_0 \land Z) \land (S_1 \land Z) \lor \ldots \lor (S_k \land Z)$$
- Leverages work done on SAT solvers
Outline

- What is Model Checking?
- Modeling basics
- Property specification in temporal logic
- CTL model checking
- Related techniques
- Conclusion
Where do we stand?

Based on Dill and Tesiran’99

Model checking
FSM-based generation
Symbolic simulation
Manual test w/ coverage
Random simulation

Coverage

Scale (gates)

1 FSM
50K
250K
2M
Research Issues

- Fundamental complexity hurdles in scaling formal verification methods with design sizes
- Approximate verification methods, semi-formal methods will be important tools in future
- Approximate methods:
  - Model approximations should be driven by property being checked
  - Available computing resources can be used to guide nature of approximation
  - Verification should give some coverage metric on termination
    - “Out of memory” after 48 hours not of any use!
Research Issues

- Abstraction of system behavior crucial for analyzing large designs
  - Must employ right degree of abstraction
    - Can this be done automatically?
    - More research needed
- Approximate and semi-formal methods
  - “Test of the pudding” is on large real examples
  - Most published examples are small
  - A lot of ideas -- no clear winners so far
  - An active area of research
Research Issues

- With IP based designs, supplier of IP must provide hints on how to verify IP core
  - RTL with assertion checks
  - Hints on exercising combinations of inputs
  - IP verification test suite with coverage metric
  - Formal verification of IP-based System-on-Chips will be challenging
    - Success constrained by verifiability, controllability and observability of IP-core
Research Issues

- Runtime/memory bottlenecks of existing formal verification methodologies must be addressed to scale them to larger designs
  - Active field of ongoing research

- Suitable combination of techniques (random simulation, model checking, theorem-proving …) to be selected for each verification task
  - Understanding which techniques work well under what circumstances
  - Can this be automated?
Future Research Directions

- A verification framework involving different cooperating verification methods
  - What should be the data structure?
- Design for Verifiability -- is it viable and practical?
- How do we integrate verification tools with synthesis tools?
- How do we generate counter-examples for sequential, concurrent, and real-time systems verifiers?
- Scalability of existing methods to larger designs
Summary

- Formal verification has brought hard-core engineers and theoreticians on a common platform
- Model checking most successful so far
- The gap between VLSI advancements and advancements in FV techniques can be filled to some extent by semi-formal methods
  - Hard to measure “success” for such techniques?
  - I might not have detected one out of 100000 bugs, but this might be the killer bug
Summary

- Rigorous verification will be an important area in future
- Simulation and emulation still predominant techniques used in industry and justifiably so
- However, formal methods ARE NEEDED when applicable and when one can’t afford to miss a bug
  - Success stories: Protocols, FSMs, specific processors, floating point arithmetic etc.