Computer Aided Verification

For Designing correct systems

Hao Zheng
zheng@cse.usf.edu
Dept. of Computer Science & Eng.
University South Florida
Outlines

• Basic concepts of verification
• Challenges to verification
• Simulation vs formal verification
• Basic concepts of formal verification/model checking
• Various model checking methods
• Current status of model checking
What is Verification?

• A process of ensuring satisfaction of design constraints.

• Can be classified based on purposes:
  – Functional verification
  – Timing verification
  – Power analysis

• Can be classified based on abstraction levels:
  – Behavioral verification
  – Logic verification
  – Hardware testing
What is Functional Verification?

• Ensuring the functional correctness.
• Scope of verification: concurrent systems are
  – Finite states
  – Reactive
• Verification of finite state reactive systems
  – Can be automated.
  – Found in many important applications as follows.
  – Hardware designs
  – Communication protocols
  – Flight control systems
  – Operating systems
Functional Verification (cont’d)

- Elements necessary for verification:
  - Test pattern generation
  - Correctness definition
  - System models: FSMs

- Verification methods
  - Simulation
  - Equivalence checking
  - Model Checking
  - Theorem proving
Why Functional Verification?

• Some numbers:
  – Verification engineers : design engineer = 3:1
  – 50% – 70% design resource is for verification.

• The reasons:
  – The longer a bug undetected, the more costly the fix.
  – A bug found early has little cost.
  – A bug found after being manufactured may require to repeat the whole design process.
  – Finding a bug in the customer's environment can cost hundreds of millions in hardware and brand image
Verification Experience

Functional testing

Time

Bugs per week

Cost to Fix Bugs
Bugs Are Costly

- **Pentium bug**
  - Intel Pentium chip, released in 1994 produced error in floating point division.
  - Cost: $475 million

- **ARIANE Failure**
  - In December 1996, the Ariane 5 rocket exploded 40 seconds after take off. A software component threw an exception.
  - Cost: $400 million payload.

- **Therac 25 Accident**
  - A software failure caused wrong dosages of x-rays.
  - Cost: Human Loss.
Challenges to Verification

• Moore’s Law:
  – Number of transistors double in every 18 months.
  – System size grows exponentially.

• Functional complexity grows exponentially.
  – System complexity, the number of states, is exponential.
  – Dynamic behavior is described as state transition sequences.
  – The number of state transition sequences may be exponential in number of states.
  – Functional complexity grows at double exponential pace!
Simulation-Based Verification

• Verification process:
  – Build a system model.
  – Drive the inputs with test patterns.
  – Check if outputs match the specification.

• Simulation scales well with system size.
  – Performance degrades polynomially as size grows.
  – Can be applied to systems with any sizes.

• Definition of functional coverage
  – The percentage of all possible behavior verified.
  – Used to measure verification quality.
Simulation-Based Verification (cont’d)

• Functional coverage degrades exponentially
  – As complexity grows exponentially.

• Bugs may exist in system behavior not verified.

• Simulation does not give much confidence in system correctness!
What Can We Do?
Formal Verification to Rescue

• Based on logic and automata foundations.

• **Exhaustively** verify system correctness.

• Gaining momentum since Intel Pentinum bug.

• Can be classified as
  – Logic equivalence checking
  – Model checking
  – Theorem proving

• Semi-formal method:
  – Combining model checking with simulation.
  – Improve functional coverage.
Equivalence Checking

• Checks for mismatches between two gate-level circuits, or between HDL and gate-level (satisfiability).

• "Formal", because it checks for all input values (solves SAT problem)

• **Acceptance**: Widely used ("It's a done deal.")

• **Limitation**: Doesn't catch functional errors in designs. (Analogy: like checking C vs. assembly language.)
Formal Verification Methods

• **Theorem Proving**
  – A system represented as a set of axioms.
  – System correctness expressed as theorems.
  – Axioms -> Inference rules -> theorems.
  – Require strong background in mathematics/logic.
  – Can verify infinite state systems.
  – Cannot tell cause of the bug if the system has one.

• **Model Checking**
  – Model a system as a state transition graph.
  – Check logic properties on system models.
  – Fully automated.
  – Produce a counter-example for any bug found.
Temporal Logic Model Checking

- Model checking is an **automatic verification technique** for **finite** state concurrent systems.
- Developed independently by Clarke, Emerson, and Sistla and by Queille and Sifakis in early 1980’s.
- **Specifications** are written in some **temporal logic**, CTL, LTL, etc.
- A finite state model is built for the system through **exhaustive search of the state space**.
- Verification checks that specification is satisfied on the model.
Advantages of Model Checking

- **Exhaustiveness**: compared to simulation.
- **No proofs.**  
  - In general, *why* a system is correct is not important.
- **Fast**: compared to theorem proving.  
  - minutes instead of months.
- **Counter-examples** to speed debugging.  
  - Pinpoint source of the bug.
- Specification logics can easily express many concurrency properties.
A Model Checking System
A Model of Systems

State transition graph
Kripke Structure
An Example of Design Model
Linear Time Logic

• Express properties of “Reactive Systems”
  – interactive, nonterminating
• LTL formulas defined on infinite state sequences.

$$\sigma = s_0, s_1, s_2 \ldots$$

• Temporal operators
  – “Globally”: $$G p \text{ at } t \iff p \text{ for all } t' \geq t.$$
Linear Time Logic (cont'd)

- "Future": $F p$ at $t$ iff $p$ for some $t' \geq t$.

- "Until": $p U q$ at $t$ iff
  - $q$ for some $t' \geq t$ and
  - $p$ in the range $[t, t')$

- "Next-time": $X p$ at $t$ iff $p$ at $t+1$
LTL Model Checking

• A path in $M = (S, R, L)$ is a sequence of states

$$\sigma = s_0, s_1, s_2 \ldots \in S^*$$

such that $(s_i, s_{i+1}) \in R$.

$$M, s_0 \models f \text{ iff for all paths } \sigma = s_0, s_1, s_2 \ldots \text{ of } \sigma, s_0 \models f$$
Computation Tree Logic (CTL)

- **Temporal Operators**
  - $\text{AF}p$ - $p$ holds sometime in the *future*.
  - $\text{AG}p$ - $p$ holds *globally*.
  - $\text{AX}p$ - $p$ holds *next* time.
  - $\text{A}(pUq)$ - $p$ holds *until* $q$ holds.
  - $\text{EF}p$ - $p$ holds sometime in the *future*.
  - $\text{EG}p$ - $p$ holds *globally*.
  - $\text{EX}p$ - $p$ holds *next* time.
  - $\text{E}(pUq)$ - $p$ holds *until* $q$ holds.
- $p$ and $q$ are some temporal logic formulas.
- Ex.: $\text{A}(\text{req} \rightarrow \text{AF} \text{ack})$, $\text{AG} (\neg\text{grant}_1 \lor \neg\text{grant}_2)$
CTL Definition

- Every operator $F$, $G$, $U$, $X$ preceded by $A$ or $E$.
- $A$: universal quantifier.
CTL Definition (cont'd)

- **E**: existential quantifier.
CTL Model Checking

Let $M$ be a labeled state-transition graph (Kripke structure).

$$M = \{S, R, I, L\}$$

Let $f$ be a CTL formula.

Find all states $S_f$ of $M$ such that $M, S_f \models f$.

$f$ is satisfied on $M$ if $I \subseteq S_f$, denoted as $M \models f$. 
CTL Model Checking Algorithms

- We need only EX, EU, EG
  - AG\( p = \neg E(\text{true U } \neg p) \);  AF\( p = \neg EG\neg p \)
- Checking E(p U q) by backward BFS

- Checking EG\( p \)
Example: Microwave Over Controller

- The oven doesn’t **heat up** until the **door is closed**.
  - $E(\sim\text{heat} \lor \text{close})$
Microwave Oven Controller (cont’d)

\[ M \models E(\sim \text{heat} \cup \text{close}) \]

First, find states labeled with \textit{close}. 

Microwave Oven Controller (cont’d)

\[ M \models E( \sim \text{heat} \cup \text{close} ) \]

Next, find states labeled with \( \sim \text{heat} \) in backward direction from the states found in the first step.
Microwave Oven Controller (cont’d)

\[ M \models E(\sim \text{heat \ U close}) \]

Next, find states labeled with \(\sim \text{heat}\) in backward direction from the states found in the first step.
Microwave Oven Controller (cont’d)

\[ M \models E( \sim heat \cup close ) \]

Formula \( f \) holds on \( M \).
Main Disadvantage

State Explosion Problem:
- Too many current processes
- Data Paths

Much progress has been made on this problem recently!
- Symbolic model checking
- Partial order reduction
- Abstraction
- Compositional verification
- Bounded model checking
Symbolic Model Checking

- States and transitions are encoded as Boolean formulas.
- Use BDDs (binary decision diagrams) to represent state spaces symbolically.
- MC manipulates Boolean formulas.
- BDD
  - Is often very compact.
  - Exist efficient algorithms.
- SMV problems
  - Exponential BDD size in worst case.
  - BDD size unpredictable.
  - Unacceptable in production.

\[(x_1 \lor x_2) \land x_3\]
Symbolic Breadth-First Search

Each layer of breadth-first search is represented by a BDD.
Fixpoint Characterization of EU

- It is the limit of the increasing series

\[ \text{EF } p = \text{E}(\text{true } \cup p) \]

...which we can compute entirely using BDD operations
Symbolic Model Checking

• Computer fixpoints for TL formulas using BDDs.
  – A fixpoint is a set of states satisfying a property.
  – Then, check if initial states included in the fixpoint.

• Acceptance:
  – There have been major successes on some industrial projects.
  – Used on particular projects in huge companies (e.g. IBM, Intel)
  – Commercially supported products.
  – But <5 % use overall.
Partial Order Reduction

• Targeted for asynchronous system verification.
• Async. components run in parallel.
  – There is no clock for synchronization.
• All orderings of concurrent events in an async. design are considered.
  – To avoid discrimination of a particular ordering.
• Ex:. consider $n$ events executed concurrently.
  – there are $n!$ orderings and $2^n$ states.
• Orderings among independent events are irrelevant to verification
  – Major source of state explosion in an async. system.
Partial Order Reduction (cont'd)

- If the specification does not distinguish these orderings, ordering $n! \rightarrow l$, and states $2n \rightarrow n+1$.
- In a sync. design, all events execute at the same time.
- The reduce model contains same information for MC.
Abstraction

• Abstraction produces an abstract model by abstracting irrelevant information to verification.

• Essential to verification of practical systems.

• Two methods:
  – Cone of influence reduction
  – Data abstraction
  – Predicate abstraction
Cone of Influence Reduction

- Usually, the specification is not complete
  - It refers to a subset of all state variables.
  - Cone of influence of a variable is the set of variables driving the former one.

- Cone of influence (COI) reduction
  - Remove variables not in COI of variables in the spec.

### 3-bit counter

\[ v'_0 = \neg v_0 \]
\[ v'_1 = v_0 \oplus v_1 \]
\[ v'_2 = (v_0 \land v_1) \oplus v_2 \]

\[ \text{COI}(v_0) = \{ v_0 \} \]
\[ \text{COI}(v_1) = \{ v_0, v_1 \} \]
\[ \text{COI}(v_2) = \{ v_0, v_1, v_2 \} \]
Data Abstraction

• Abstraction produces an abstract model by abstracting irrelevant information to verification.
• Targeted for systems with large datapath.
• Verification involves only simple logic relations over the data space.
• Partition the data space into equivalence classes according to the logic relations.
• Ex.: input data is checked if it is larger than 5.
  – \( D_A = \{D \leq 5, D > 5\} \)
Example of Data Abstraction

\{\text{green, red, yellow}\} \quad \Downarrow \quad \Downarrow
\{\text{go, stop}\}

⇒

\{\text{go, stop}\} \Downarrow
⇒

\{\text{green, red, yellow}\}
\{\text{go, stop}\}
Compositional Verification

- Parallel composition leads to exponential growth of state space.
  - Impose an upper limit on the system size.

- In general, systems have structural hierarchy.
  - Divide-and-conquer is the natural solution.

- Compositional verification handles each component.
  - Verify specification for each component.
  - Compose component specification for the complete system.
Assume-Guarantee Reasoning

• Make assumptions on environment first.
• Verify component with the assumptions.
• Verify that environment satisfies assumptions.

\[ M = P \ || \ Q \]

If \( P \ || \ f \models g \) and \( Q \models f \)
then \( P \ || \ Q \models g \)

• Currently, machine learning algorithms are used to derive assumptions automatically.
Compositional Verification (cont’d)

$\Sigma_p$ is the set of I/O of $P$.

$Q' = Q \setminus \Sigma_p$

If $P || Q' \models \phi_P$

Then $P || Q \models \phi_P$

Requirement:
The interface behavior of $Q'$ must preserve that of $Q$. 
Bounded Model Checking

• Targeted for finding bugs.
• Based on SAT solvers.
  – Assume a counter-example of length $k$, then generate a propositional formula from $M$ and check its satisfiability.
  – In BMC, LTL model checking is reduced to a SAT problem in polynomial time.

• Advantages
  – Finding counter-examples: fast, and minimal length.
  – Less space, no variable orderings (vs BDD).
Bounded Model Checking Overview

System Model → BMC → Boolean formula

LTL formula

Formula violated ← Satisfied

Formula holds ← Unsatisfied

SAT Solver
Consider $M | - \lnot G f$ as BMC problem with $k=3$. All paths in $M$ with length 3 is

$$[M]_3 = S_0 \land R(S_0, S_1) \land R(S_1, S_2) \land R(S_2, S_3)$$

$G f$ is defined over paths with loops. Loop condition is defined as

$$L = R(S_3, S_0) \lor R(S_3, S_1) \lor R(S_3, S_2) \lor R(S_3, S_3)$$

$G f$ over the above paths is defined as

$$[f]_3 = f(S_0) \land f(S_1) \land f(S_2) \land f(S_3)$$
Put everything together, the BMC for $M |\models \neg G f$ with $k=3$ is defined as follows:

$$[M, f]_3 = [M]_3 \land L \land [f]_3$$
Real-Time System Verification

• Correct logic behavior within certain amount of time.
• Real-time systems can be found in
  – flight control
  – Nuclear power plant control
  – Robots
• Model timing at higher-level
  – Timed Automata
  – Timed Petri-nets
• Timing representaion in state space.
Timed Automata

- Timed automata are finite-state automata augmented with real-valued clocks.

\[ x, y : \text{clocks} \]

\[ y < 4; \ a, \ x := 0 \]

\[ x = 5; \ b \]

\[ c; \ y := 0 \]
Timed Petri-Nets

$CW, CP...$ are events.
$t1, ... t6$: bounded clocks.

An event executes when

- all preceeding places are marked
- all preceeding clocks are in their bounds.
Timed Petri-Nets (cont'd)

\{1, t6=0}\}

\[
\begin{align*}
&\text{CW} \rightarrow t1: [2, 3] \rightarrow CP \\
&\quad \rightarrow t2: [2, 3] \rightarrow WA \\
&\quad \rightarrow t3: [5, \infty] \rightarrow PA \\
&\quad \rightarrow t4: [2, 3] \rightarrow WP \\
&\quad \rightarrow t5: [2, 3] \\
&\quad \rightarrow t6: [2, 3]
\end{align*}
\]
Timed Petri-Nets (cont'd)

\[\{1, t6=0\} \rightarrow \{1, t6=2\} \xrightarrow{CW} \{2,3\}, t1=t2=0 \rightarrow \{2,3\}, t1=t2=2 \cdots \]


**Time Representation**

- **Continuous**: the value of clocks is a real number.
  - There are infinite timed states
  - Value(clock) = integer + fraction.
  - Region = integer values of all clocks and relations between their fractions.

- **Discrete**: value of clocks is an integer number.
  - If it is not important to check the value of clocks strictly greater/less than the timing bounds.

- **Zone**: the values of clocks that enable the execution of an event are defined using inequality formulas.
  - $0 \leq t_1 \leq 2$, $0 \leq t_2 \leq 2$, $t_1 - t_2 \leq 0$, and $t_2 - t_1 \leq 0$. 
Past Experience

- Using formal verification is an *economic* decision.

- **Costs:**
  - Requires expensive, skilled labor.
  - May delay time-to-market.
  - Users must *need* formal verification.

- Look where the bugs are:
  - Interacting state machines
  - Memory systems (uni/multi-processor)
  - Floating point
• *Bug hunting* is valuable

• Easier:
  – doesn't require full verification
  – liberal abstractions work (e.g. downscaling)
  – may find error before looking at all states

• Value is more evident
  – Designs believed to be bug-free by default.
  – Cost of bugs is approximately quantifiable
    (= value of verification)
Model Checking Systems

• There are many model checking systems for hardware and protocol verification.
  – Software verification tools are coming!

• Industry (Intel, IBM, Motorola) has been using MC more widely.
  – Obvious reason!

• **SMV**: first symbolic model checker

• **SPIN**: an explicit model checker for SW verification.

• **Verus/Kronos/ATACS**: real-time system verification.

• **HyTech**: hybrid system verification.
Model Checking Systems (cont'd)

• **Cospan/FormalCheck**: w-automata/language inclusion.

• **SteP/PVS**: combination of model checking and theorem proving.

• **VIS**: combines model checking with logic synthesis and simulation.

• **NuSMV**: latest implementation of SMV including a bounder model checker.
Model Checking Examples

• IEEE Futurebus+
  – First time it is verified formally
  – Found unexpected errors.

• IEEE SCI
  – Found some errors in a “correct” design.

• My experience in IBM
  – Verified data link layer design of Infiniband protocol.
  – 1 vs. 4 verification engineers (two more later)
  – 1/3 of errors found by 1 person with SMV.
Model Checking Performance

- Model checkers today can routinely handle systems with between 100 and 1000 state variables.
- Systems with $10^{120}$ reachable states have been checked. (Compare approx. $10^{78}$ atoms in universe.)
- By using appropriate abstraction, systems with an essentially unlimited number of states can be checked.
- By combining compositional approach with abstraction, most finite state systems can be verified.
- Rationale of model-checking
  - More problems found by exploring all behavior of a downscaled system than by testing some behavior of the full system.
Near-term opportunities

• Security (Cryptographic protocols)
  – Model checking (Lowe, Clarke, Mitchell, Wing)
  – Theorem proving (Paulson)
  – Very important (e.g. e-commerce)
  – Protocols are reasonably small

• Distributed algorithms (Fault tolerance, Synchronization, Agreement)
  – People are willing to prove them \textit{manually}.
  – … but they make mistakes.
  – Computer assistance for case analysis, debugging.
Near-term opportunities (cont'd)

• High-level specifications (Statecharts, UML, RSML, SCR, Z)
  – Smaller than implementations.
  – If concept is wrong, can we get correct product?
  – “Most bugs are specification errors” (?)
  – Bugs can be serious, conceptual problems.
  – Model checking (NRL, Atlee, Uwash
  – Satisfiability (Jackson)
  – “Semantic checking” (Tablewise, NRL)

• Embedded software is (sometimes) more like hardware than software
Near-Term Challenges

- **Capacity**: design sizes that can be handled is limited.
  - Requires a lot of human intervention.
- **Robustness**: Whether and when verification can be finished cannot be predicted.
  - Unaccepted in production environment!
- **Verification metrics**: measure verification quality.
  - Enough specification?
  - Enough environment modeling?
- **Reuse**: save verification time
  - how to take local verification and reuse it in global setting.
Questions?

问题？