

Course Description

- An introduction to formal verification/model checking.
- Double listed course for both under/grad. students.
 - Extra requirements for graduate students.
 - Separate evaluation.
- Course topics:
 - Basic concepts
 - Specifications
 - Verification algorithms
 - Verification methodologies

Course Description (cont'd)

- Prerequisite: solid background in discrete math/logic.
 - Challenging course, but rewarding!
- Grading:
 - Assignments: 60%
 - Midterm: 20%
 - Final exam/project: 20%
- Class Policy:
 - Honesty: all work must be original!
 - **No late submission will be accepted!**

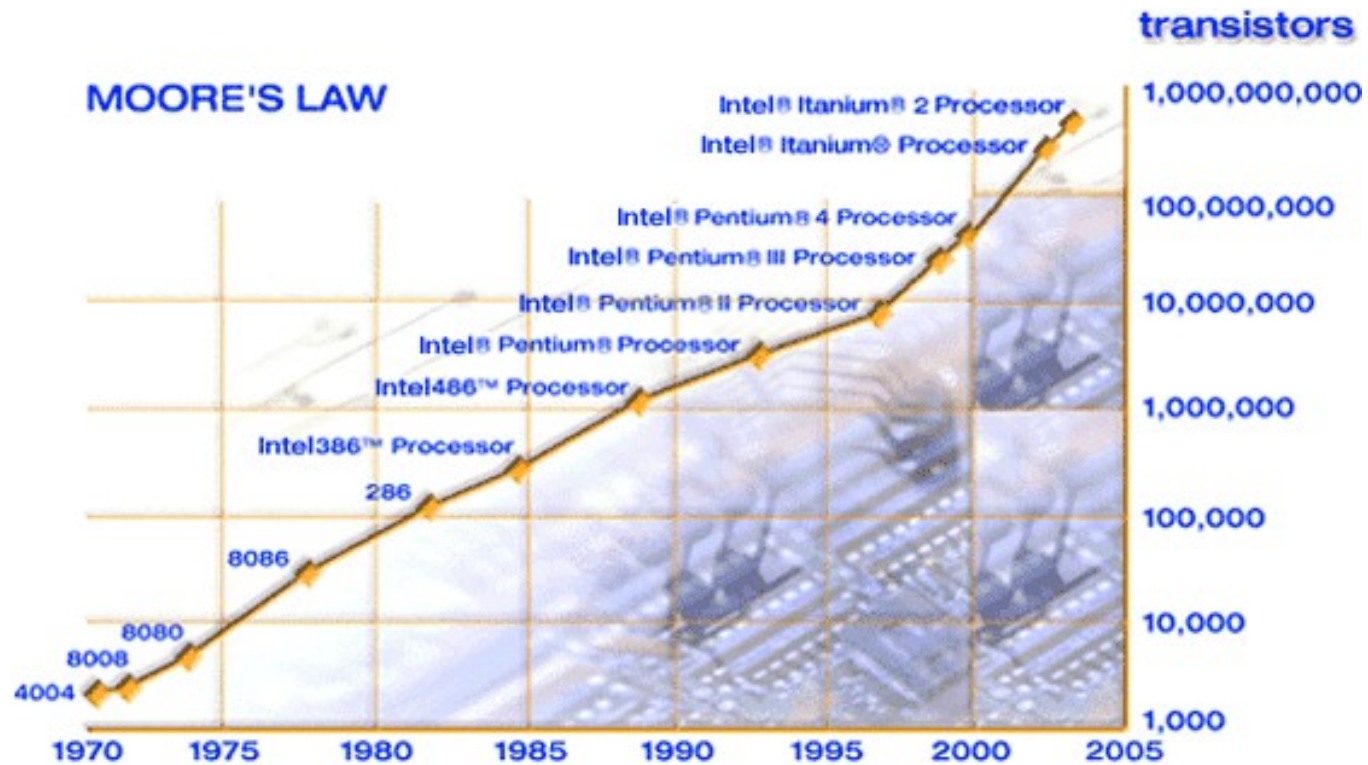
Course Description (cont'd)

- Textbook: none.
 - Handouts will be distributed.
- Other good references:
 - *Model Checking (MIT Press)*
 - *Hardware Design Verification*
 - *Logic in Computer Science*
- Office hour:
 - When: Mon., Wed. 3 - 5pm, or by appointment.
 - Where: ENB 312

Moore's Law

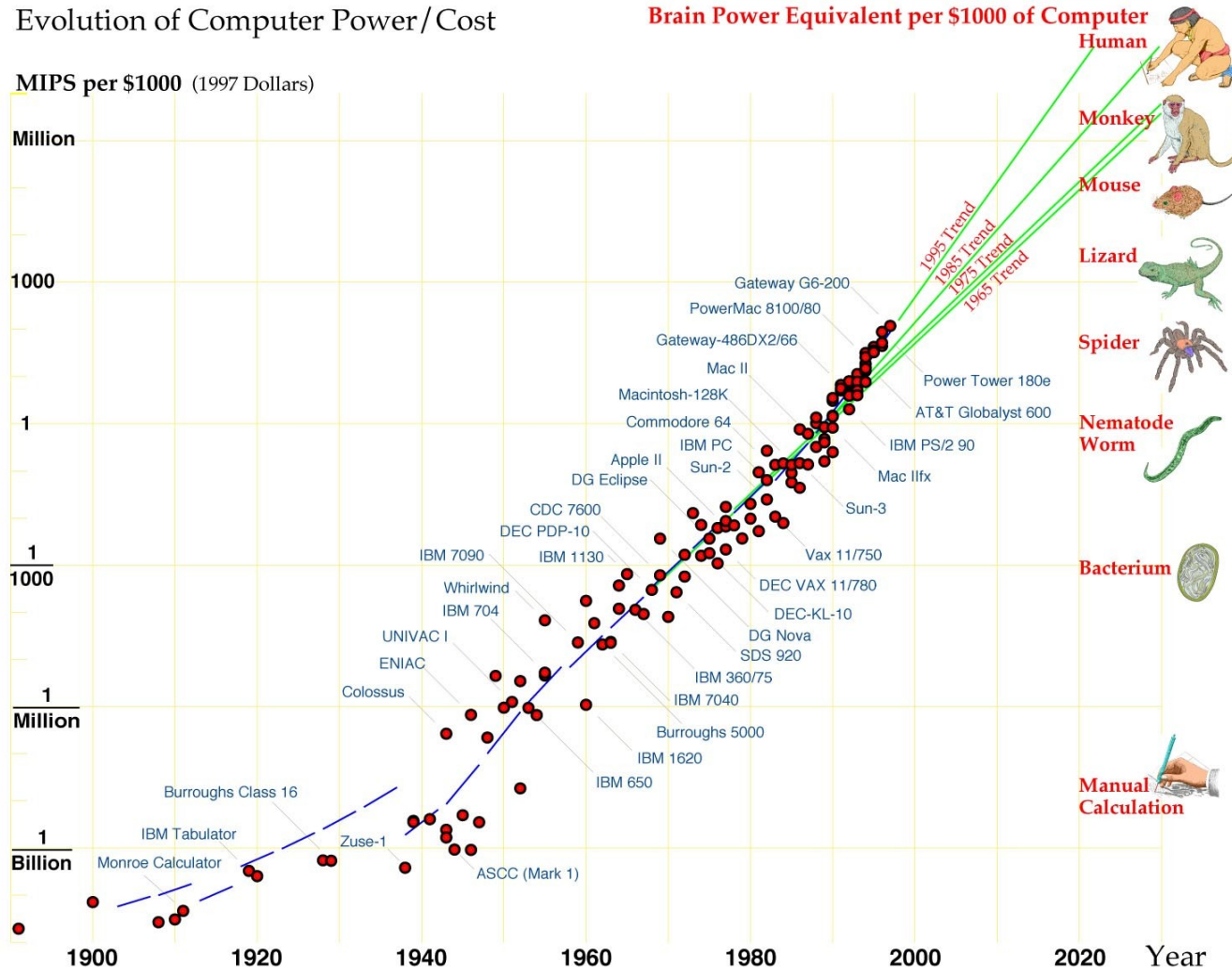
"...(T)he first microprocessor only had 22 hundred transistors. We are looking at something a million times that complex in the next generations—a billion transistors. What that gives us in the way of flexibility to design products is phenomenal."

—Gordon E. Moore



History of Computing Power

Evolution of Computer Power/Cost



Human brain power equivalent to 10^8 MIPS of computer power.

Deep Blue from IBM in 1997: 3 mil MIPS.

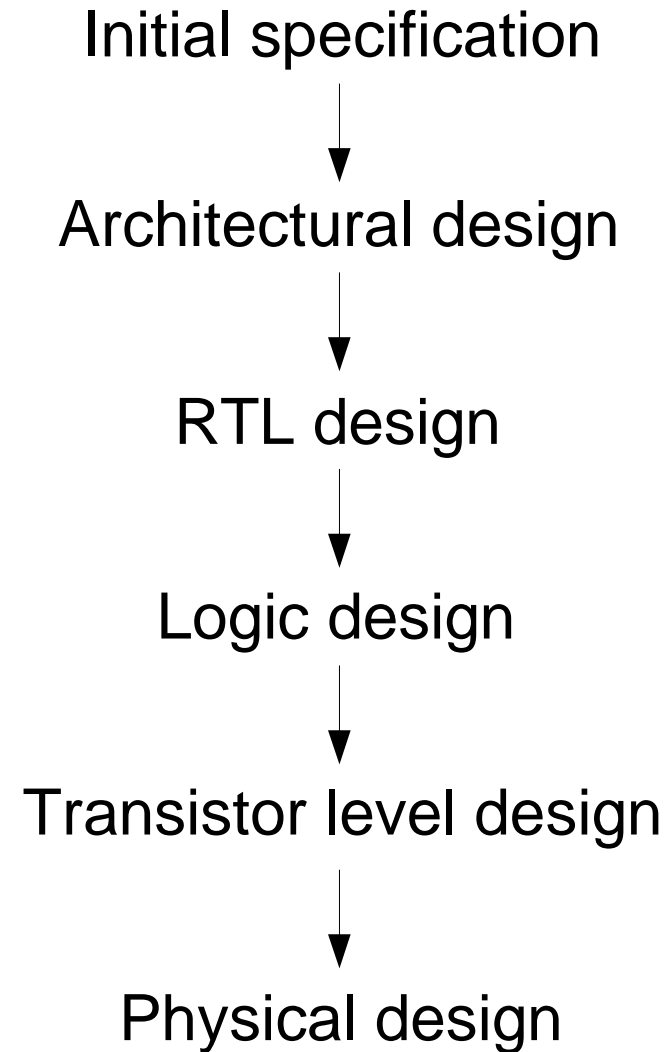
Digital Lives

- PC is a past era, and now is the Internet!
- Computing devices are everywhere.
 - PDAs/cell phones, TVs, cars, airplanes, equity trading, etc.
 - Estimate number of ucontrollers in an average car sold in the US is about 200.
- Based on Moore's law, complexity of computing devices grows exponentially. With Internet, the complexity of a networked system becomes even higher. With these systems running a significant portion of our lives, it is essential to make sure they are reliable, and do not do anything that surprises us.

What is Verification?

- A process of ensuring satisfaction of design constraints or requirements.
- 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

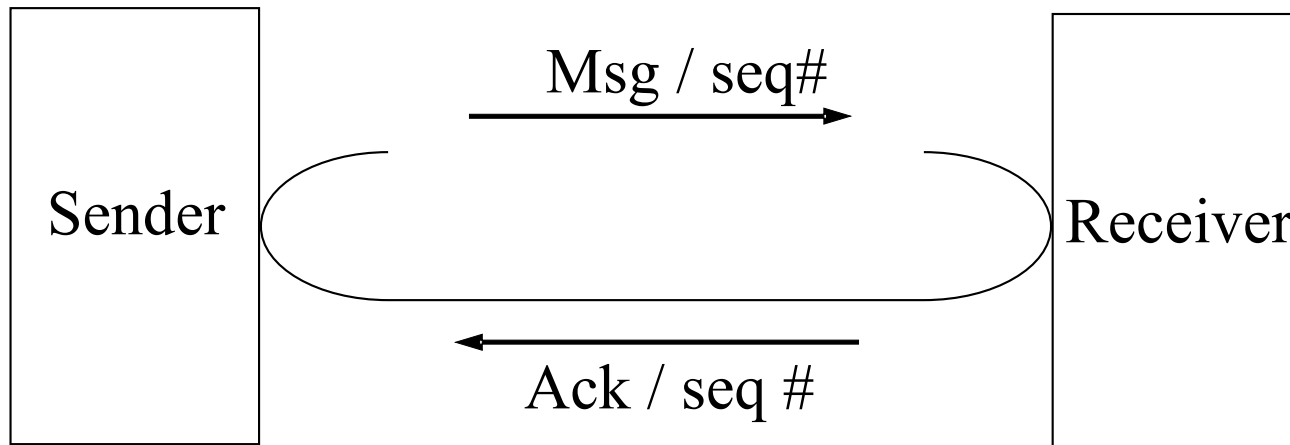
Hardware Design Flow



What is Functional Verification?

- Ensuring the correct logical behavior of systems.
- Scope of verification: concurrent systems that are
 - Finite states and Reactive
 - Makes verification decidable.
- Verification of finite state reactive systems
 - Can be automated.
 - Can be found in many important applications as follows.
 - Hardware designs
 - Communication protocols
 - Flight control systems
 - Operating systems

An Example of Reactive Systems



Some requirements of protocol:

- Every message sent is eventually received.
- A message is not received unless one is sent .

Some requirements of components:

- Sender keeps resending a message until Ack is received.
- Receiver does not produce Ack before a message is received.

Sequential vs Combinational Sys.

- A sequential system has memory of the past.
 - It produces outputs based on current inputs and current state.
 - Sequential verification is much harder since all possible orderings of inputs need to be checked. Different sequences of the same inputs often produce different output sequences.
- A combinational system is memory-less.
 - The same output is produced given the same inputs.
 - Orderings of inputs do not need to be considered.

Functional Verification: Why

- Some numbers related to HW design:
 - Verification engineers : design engineer = 3:1
 - 50% - 70% design resource is for verification.
- The reasons: with various computing devices running almost everywhere, human lives and smooth operation of the society depend largely and critically on the reliability and correct operations of these devices. Defects of the systems can cause huge economic or even human life losses.
 -

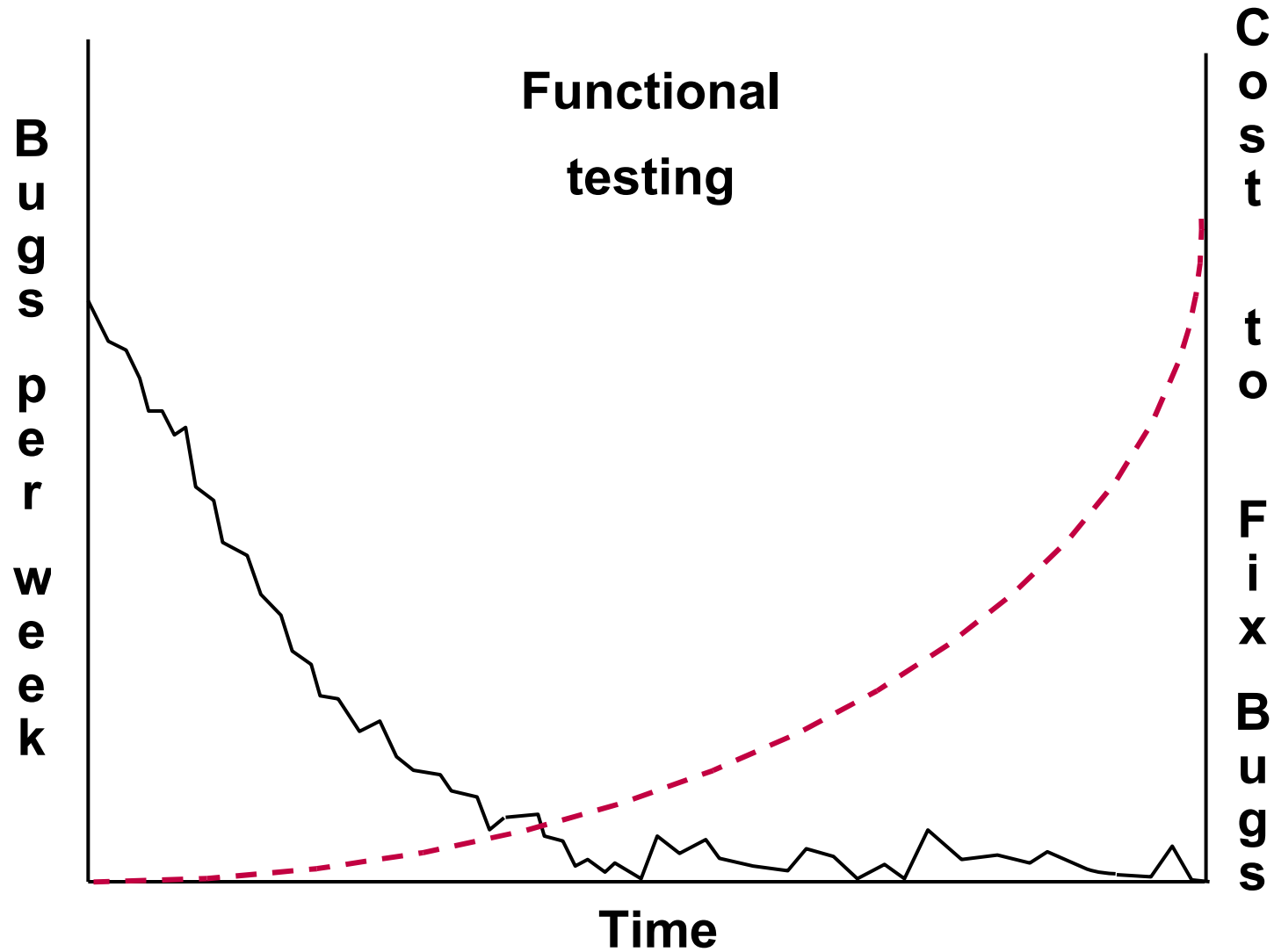
Bugs Are Costly

- **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 components threw an exception
 - Cost : \$400 million payload.
- **Therac 25 Accident**
 - A software failure caused wrong dosages of x-rays.
 - Cost: Human Losses.

Functional Verification: When

- The sooner the better!
 - The longer a bug goes undetected, the more costly the fix.
- A general design flow: model -> implementation -> tapeout
 - Implementation inherits bugs in its model.
- A bug found early (prototyping) has little cost.
- A bug found after being implemented may require to repeat the whole design process.
- Finding a bug in the customer's hands can cost hundreds of millions in hardware and brand image.
- Even worse, some bugs may not be fixed when the system is deployed.

Cost to Debugging

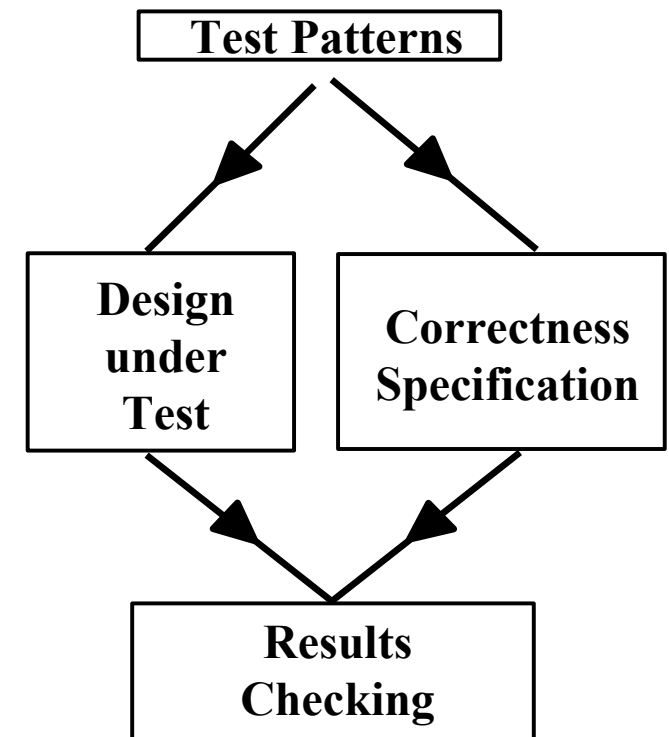


Challenges to Verification

- System complexity grows exponentially, and
 - System re-spin adds cost!
- Competition requires short time-to-market (TTM).
 - Late entry = no market!
- There is delicate trade-off between verification quality and cost.
- For mission critical systems, we need the guarantee of the system correctness at any cost.
- For non-critical soft systems, bugs need to be uncovered as many as possible without delaying TTM.

Functional Verification: How-to

- Elements necessary for verification:
 - System models: FSMs
 - Correctness definition
 - Test pattern generation
- Verification methods
 - Simulation
 - Theorem proving
 - Model Checking



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 system size grows.
 - Can be applied to systems with any sizes.
- Definition of functional coverage
 - The system model is regarded as a FSM.
 - The percentage of all states checked.

Simulation-Based Verification (cont'd)

- Functional coverage degrades exponentially.
 - A moderate design with 2^{100} states.
 - A simulator handles 100 states in 1 ns.
 - It takes more than 2^{89} seconds to check all states, long than the life of the universe ($4e17$ seconds).
- Even covering a very small portion of the entire state space requires unbearable amount of time.
 - Debugging is like searching for a diamond lost in an ocean!
 - Simulation does not give much confidence in system correctness!
- Hardware emulation: implement design in FPGAs.
 - Run design very similarly to real hardware implementation.
 - Speed of emulation is close to real implementation.

Formal Verification to Rescue

- Based on mathematical and logical foundations.
 - Formal verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property, using formal methods of mathematics.
- **Exhaustively** verify system correctness.
 - 'Formal' = Correctness proved to hold for all possible inputs.
- Gaining momentum since the Intel Pentium bug.
- Can be classified as
 - Logic equivalence checking
 - Theorem proving
 - Model checking
- Semi-formal method to improve functional coverage.
 - Combining model checking with simulation.

Logic 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 Functional Verification Methods

- Theorem Proving
 - Represent a system as a set of axioms.
 - Prove system correctness with a set of inference rules.
 - Can verify infinite systems.
 - Require human intervention with expertise. This is troublesome because "Manual verification is at least as likely wrong as the program itself"
 - Only suitable for very simple systems.
 - Proof process can be length, if terminates at all.
 - Cannot tell cause of the bug if the system has one.
- Model Checking
 - Exhaustively enumerates all possible systems and check that the logic properties holds on all system states.
 - Applies to finite state systems which are decidable.

System Correctness Specification

- Formal specification of system correct is essential to formal verification.
 - “ A design without requirements cannot be right or wrong, it can only be surprising. ”
 - Two methods: Logic- based and automata-based.
- Formal verification should start with defining the system correctness using some mathematical logic.
 - Natural languages are too vague.
- Two types of correctness requirements:
 - Safety: nothing bad produced by a system.
 - Liveness: something good eventually produced.

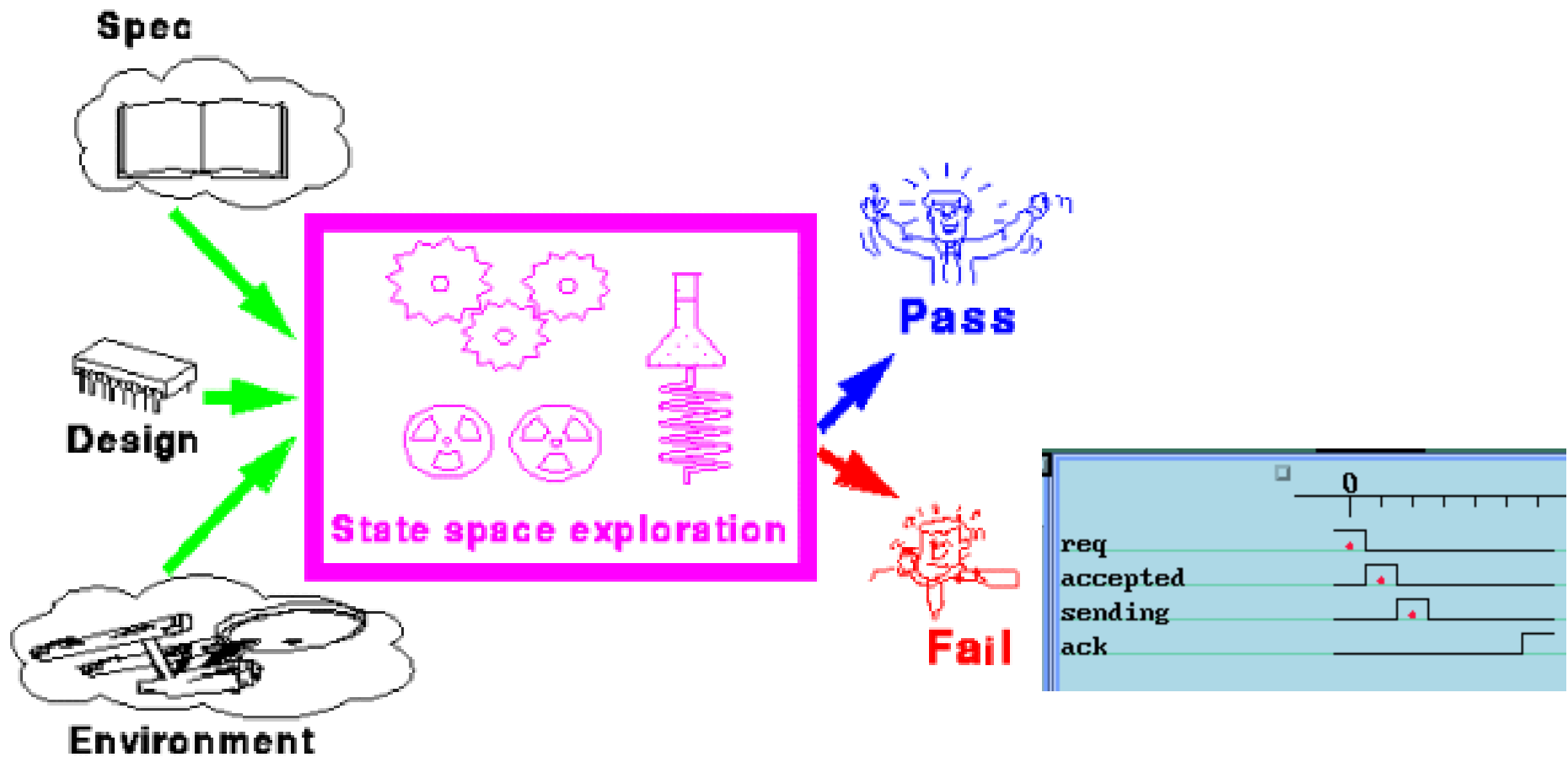
Advantages of Model Checking

- Exhaustiveness (compared to simulation).
 - All system states are checked.
- No proofs.
 - In general, **why** a system is correct is not important, and does not reveal much useful information.
- Full automated and fast.
 - Allows easy integration into the existing design flow.
- Diagnostic counter-examples to speed debugging.
 - Pinpoint source of the bug.
- Specification logics can easily express many concurrency properties.
 - Allows partial verification.

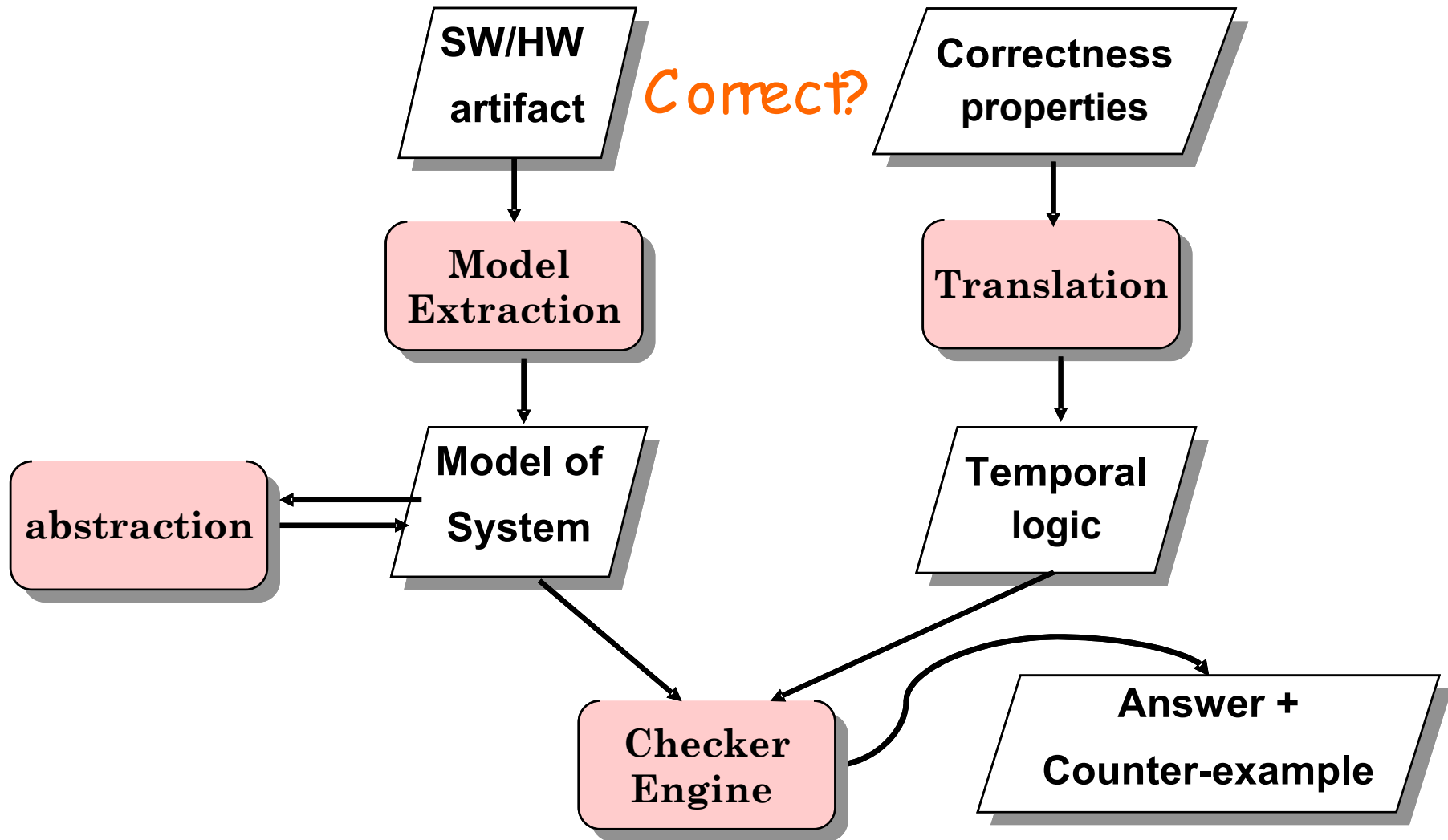
Model Checking

- Model checking (MC) 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.
- **Temporal logic MC:**
 - **Specifications** are written in **temporal logic, CTL, LTL, etc.**
 - A finite state model is built for the system through **exhaustive search of the state space.**
 - Checks that model satisfies specification.
- **Automata-theoretic MC:**
 - both design and specification are automata.
 - Checks that design behavior conforms to that of specification.

A Model Checking System



A Detailed View of Model Checking



Main Disadvantage of MC

- **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
 - Combining theorem proving with MC.

Implications of Verification

- Verification deals with models of the systems, not the actual implementations.
- The actual implementations can still fail.
 - System models lack of physical information about the actual implementation, therefore, some behavior that is not be checked during verification can cause failures in real operation.
 - Correctness definition is wrong such that wrong behavior is falsely regarded as being correct.
 - Test inputs during verification do not cover the entire set of inputs appearing in the real operation resulting in some behavior not being checked.
 - What if verification procedure itself has bugs?
 - Verification is more valuable for **debugging** in commercial setting!

Model Checking Tools

- 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 Tools (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.
- **SLAM**: a project done at Microsoft for device driver verification.

Model Checking Examples

- IEEE Futurebus+: a cache coherence protocol.
 - First time it is verified formally
 - Found unexpected errors.
- IEEE SCI: another cache coherence protocol.
 - A small instance of the system was verified.
 - Found some errors in a "correct" design.
- ISDN/IUPP: a comm. protocol.
 - 7500 lines SDL code were verified.
 - Found 112 requirement violations.
- 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 MC.

Model Checking Performance

- Model checkers today can routinely handle systems with between **100** and **2000 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.

Industrial View of MC

- MC is an effective debugging tool that it can accelerate design process and reduce TTM.
 - “Effective” is because hard or deeply hidden bugs can be found by MC fairly easily.
 - “Reduce TTM” is because MC is fast and can be applied early in the design flow.
- In practice, showing bugs is more valuable.
 - Even the bugs are false!
- MC is mainly applied to control-intensive applications.
 - Needs to identify the objects for MC.
 -

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 *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 concepts are 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.
- See http://www.itrs.net/Links/2004Update/2004_01_Design.pdf

The Verification Cycle

- Functional specification
 - Design interface and functionalities.
- Verification plan
 - What to verify, when to complete, what is needed
- Development
- Debugging
- Regression
- System test
- Escape analysis

Course Objectives

- Lay ground for future advanced research.
- Study basic concepts of formal verification.
- Learn the formal specification of concurrent systems.
- Learn various verification algorithms.
- Gain hands-on experience with some formal verification tools.
- Understand the challenges of formal verification and study some of the methods addressing them.

Topics to be Covered

- Modeling and specification.
- Explicit model checking
- Symbolic model checking
- Bounded model checking
- Symbolic Simulation
- Logic equivalence checking
- Abstraction and compositional verification
- Some other topics