Computer-Aided Verification

Introduction

Pao-Ann Hsiung

National Chung Cheng University

Contents

- Case Studies
  - Therac-25 system software bugs
  - Ariane 501 software bug
  - Mars Climate Orbiter, Mars Polar Lander
  - Pentium FDIV bug
  - The Sleipner A Oil Platform
  - USS Yorktown
- Motivation for CAV
- Introduction to Formal Verification
- Introduction to Model Checking

Therac-25

1985 ~ 1987

AECL Development History

- Therac-6: 6 MeV device,
  - Produced in early 1970's
  - Designed with substantial hardware safety systems and minimal software control
  - Long history of safe use in radiation therapy
- Therac-20: 20 MeV dual-mode device
  - Derived from Therac-6 with minimal hardware changes, enhanced software control
- Therac-25: 25 MeV dual-mode device
  - Redesigned hardware to incorporate significant software control, extended Therac-6 software

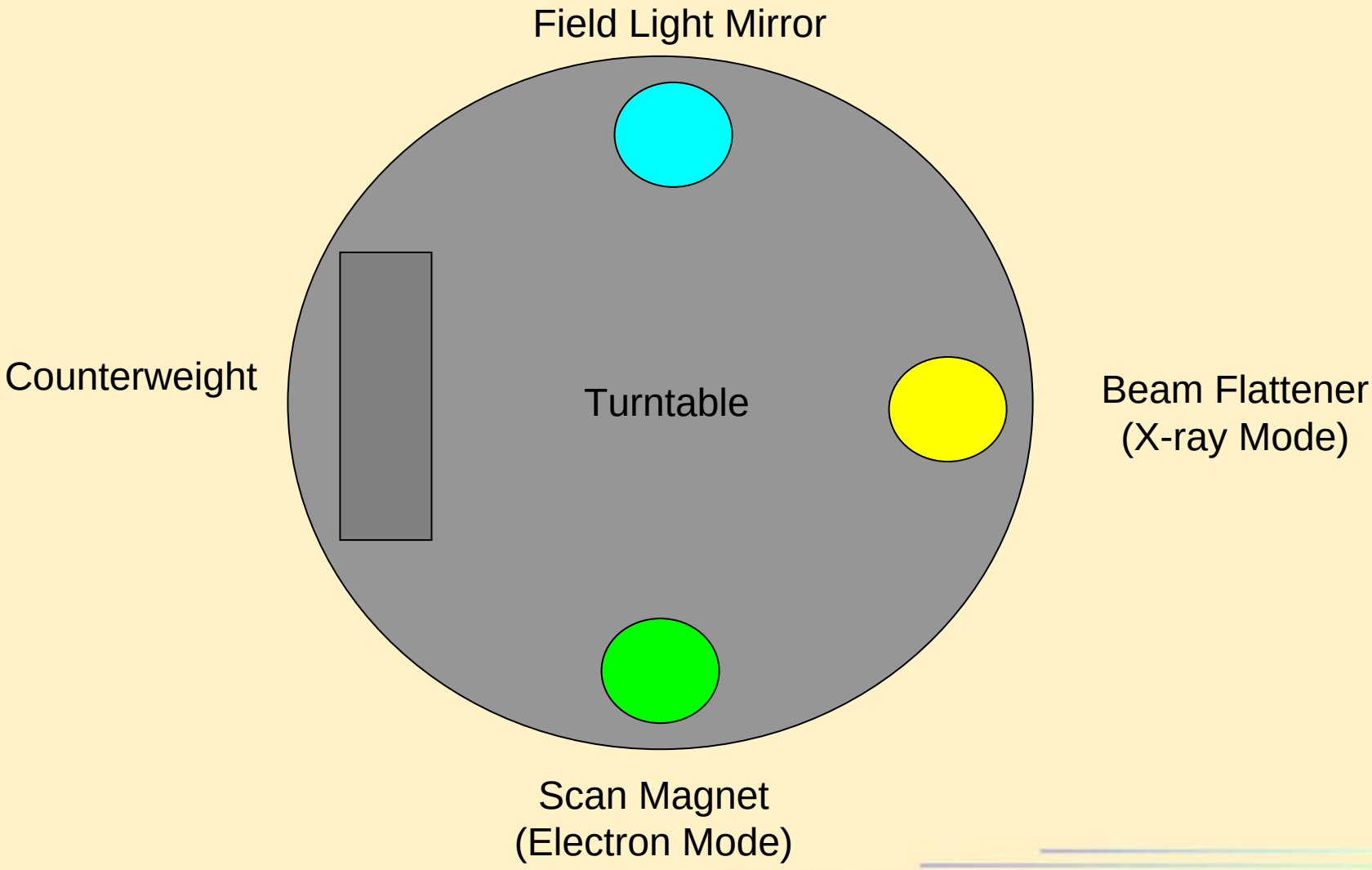
Therac-25

- Medical linear accelerator
  - Used to zap tumors with high energy beams.
    - Electron beams for shallow tissue or x-ray photons for deeper tissue.
- Eleven Therac-25s were installed:
  - Six in Canada
  - Five in the United States
- Developed by Atomic Energy Commission Limited (AECL).

Therac-25

- Improvements over Therac-20:
  - Uses new “double pass” technique to accelerate electrons.
  - Machine itself takes up less space.
- Other differences from the Therac-20:
  - Software now coupled to the rest of the system and responsible for safety checks.
    - Hardware safety interlocks removed.
  - “Easier to use.”

Therac-25 Turntable



Accident History

- June 1985, Overdose (shoulder, arm damaged)
  - Technician informed overdose is impossible
- July 1985, Overdose (hip destroyed)
  - AECL identifies possible position sensor fault
- Dec 1985, Overdose (burns)
- March 1986, Overdose (**fatality**)
  - “Malfunction 54”
  - Sensor reads underdosage
  - AECL finds no electrical faults, claims no previous incidents

Accident History (cont.)

- April 1986, Overdose (**fatality**)
  - Hospital staff identify race condition
  - FDA, CHPB begin inquiries
- January 1987, Overdose (burns)
  - FDA, CHPB recall device
- July 1987, Equipment repairs Approved
- November 1988, Final Safety Report

What Happened?

- Six patients were delivered severe overdoses of radiation between 1985 and 1987.
  - Four of these patients died.
- Why?
  - The turntable was in the wrong position.
  - Patients were receiving x-rays without beam-scattering (光散射).

What would cause that to happen?

- Race conditions.
  - Several different race condition bugs.
- Overflow error.
  - The turntable position was not checked every 256th time the “Class3” variable is incremented.
- No hardware safety interlocks.
- Wrong information on the console.
- Non-descriptive error messages.
  - “Malfunction 54”
  - “H-tilt”
- User-override-able error modes.

Cost of the Bug

- To users (patients):
  - Four deaths, two other serious injuries.
- To developers (AECL):
  - One lawsuit
    - Settled out of court
  - Time/money to investigate and fix the bugs
- To product owners (11 hospitals):
  - System downtime

Source of the Bug

- Incompetent engineering.
  - Design
  - Troubleshooting
- Virtually no testing of the software.
  - The safety analysis excluded the software!
  - No usability testing.

Bug Classifications

- Classification(s)
  - Race Condition (System Level bug)
  - Overflow error
  - User Interface
- Were the bugs related?
  - No.

Testing That Would Have Found These Bugs…

- Design Review
- System level testing
- Usability Testing

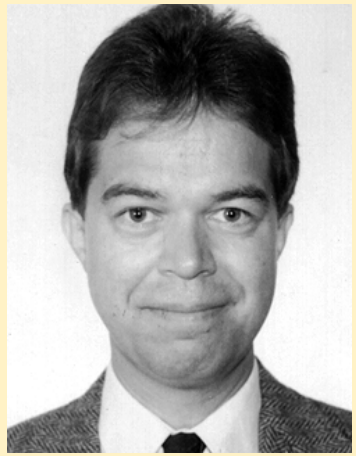
- Cost of testing… worth it?
  - Yes. It was irresponsible and unethical to not thoroughly test this system.

Sources

- **Leveson, N., Turner, C. S., An Investigation of the Therac-25 Accidents.** *IEEE Computer*, Vol. 26, No. 7, July 1993, pp. 18-41. http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html
  - Information for this article was largely obtained from primary sources including official FDA documents and internal memos, lawsuit depositions, letters, and various other sources that are not publicly available.

The authors:

Nancy Leveson



Clark S. Turner

Ariane 501

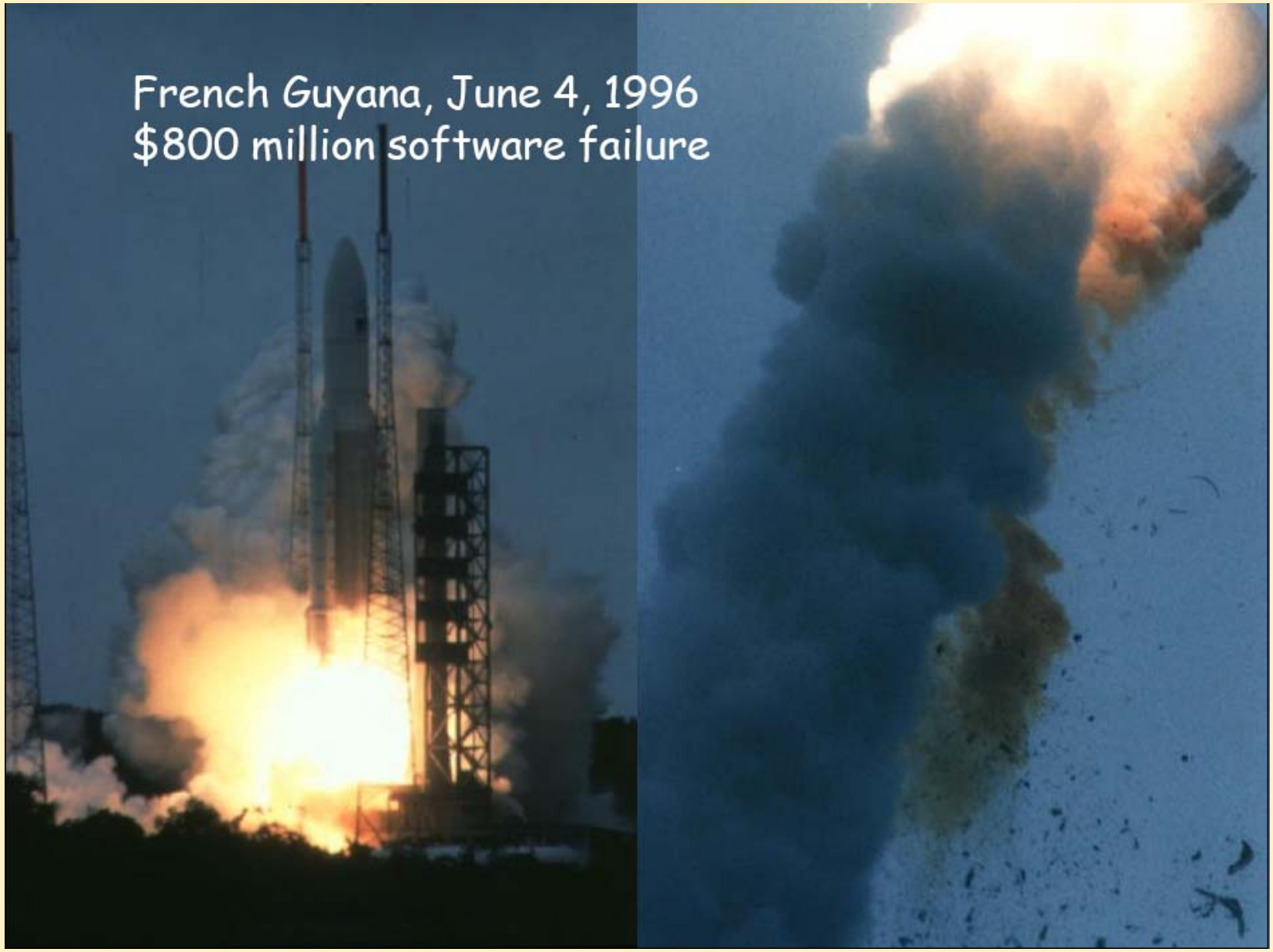
1996

Ariane 501

- On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure.
- Only about **40 seconds** after initiation of the flight sequence, at an altitude of about **3700 m**, the launcher veered off its flight path, broke up and exploded.
- Investigation report by Mr Jean-Marie Luton, ESA Director General and Mr Alain Bensoussan, CNES Chairman
  - ESA-CNES Press Release of 10 June 1996

Ariane 501 Failure Report

- Nominal behavior of the launcher up to H0 + 36 seconds;
- Simultaneous failure of the two inertial reference systems;
- Swivelling into the extreme position of the nozzles (尾噴管) of the two solid boosters (助推器) and, slightly later, of the Vulcain engine, causing the launcher to veer abruptly;
- Self-destruction of the launcher correctly triggered by rupture of the electrical links between the solid boosters and the core stage.


French Guyana, June 4, 1996
$800 million software failure

Sequence of Events on Ariane 501

- At 36.7 seconds after H0 (approx. 30 seconds after lift-off) the computer within the back-up inertial reference system, which was working on stand-by for guidance and attitude control, became inoperative. This was caused by **an internal variable** related to the horizontal velocity of the launcher exceeding a limit which existed in the software of this computer.
- Approx. 0.05 seconds later the active inertial reference system, identical to the back-up system in hardware and software, failed for the same reason. Since the back-up inertial system was already inoperative, correct guidance and attitude information could no longer be obtained and loss of the mission was inevitable.
- As a result of its failure, the active inertial reference system transmitted essentially diagnostic information to the launcher's main computer, where it was interpreted as flight data and used for flight control calculations.

Sequence of Events on Ariane 501

- On the basis of those calculations the main computer commanded the booster nozzles, and somewhat later the main engine nozzle also, to make a large correction for an attitude deviation (偏航) that had not occurred.
- A rapid change of attitude occurred which caused the launcher to disintegrate at 39 seconds after H0 due to aerodynamic forces (空氣動力).
- Destruction was automatically initiated upon disintegration, as designed, at an altitude of 4 km and a distance of 1 km from the launch pad.

Post-Flight Analysis (1/4)

- Inertial reference system of Ariane 5 is same as in Ariane 4
- In Ariane 4
  - Used before launch
  - For realignment of system in case of late hold in countdown
- In Ariane 5
  - No use!!!
  - Retained for commonality reasons
  - Operates for 40 seconds after lift-off
- Horizontal velocity variable
  - Decided not to prevent overflow of values
  - Did not analyze which values would the variable have after lift-off

Post-Flight Analysis (2/4)

- In Ariane 4
  - During first 40 seconds of flight
    - No value overflow possible for the horizontal velocity variable
- In Ariane 5
  - High initial acceleration
  - Horizontal velocity is FIVE times more rapid than Ariane 4
    - Horizontal velocity variable value overflow occurred within 40 seconds!!!

Post-Flight Analysis (3/4)

- In the review process
  - Limitations of alignment software not fully analyzed
  - Possible implications of allowing it to continue to function during flight were not realized
- In the specification and test plans
  - Ariane 5 trajectory data were not included
    - Not tested under simulated Ariane 5 flight conditions
    - Design error was not discovered

Post-Flight Analysis (4/4)

- In overall system simulation
  - Decided to use simulated output of inertial reference system, not the system itself or its detailed simulation
  - Could have included entire inertial reference system in overall system simulation
- In post-flight simulations
  - Software in inertial reference system + actual trajectory of Ariane 501 flight
  - Faithfully reproduced the chain of events leading to the failure of inertial reference systems

Mars Climate Orbiter

1998 ~ 1999

Mars Climate Orbiter

- Launched December 1998
- Arrived at Mars 10 months later
- Slowing to enter a polar orbit in September 1999
- Flew to close to the planet's surface and was lost

Mars Climate Orbiter

- “The prime contractor for the mission, Lockheed Martin, measured the thruster (推沖器) firings in pounds even though NASA had requested metric measurements. That sent the Climate Orbiter in too low, where the $125-million spacecraft burned up or broke apart in Mars' atmosphere.”

http://www4.cnn.com/TECH/space/9911/10/orbiter.03/#3

Mars Climate Orbiter

- Wow!
- And whilst all this was occurring the Mars Polar Lander was on its way to the red planet
- "*That incident has prompted some 11th hour considerations about how to safely fly the Polar Lander. "Everybody really wants to make sure that all the issues have been looked at", says Karen McBride, a member of the UCLA Mars Polar Lander science team.*"

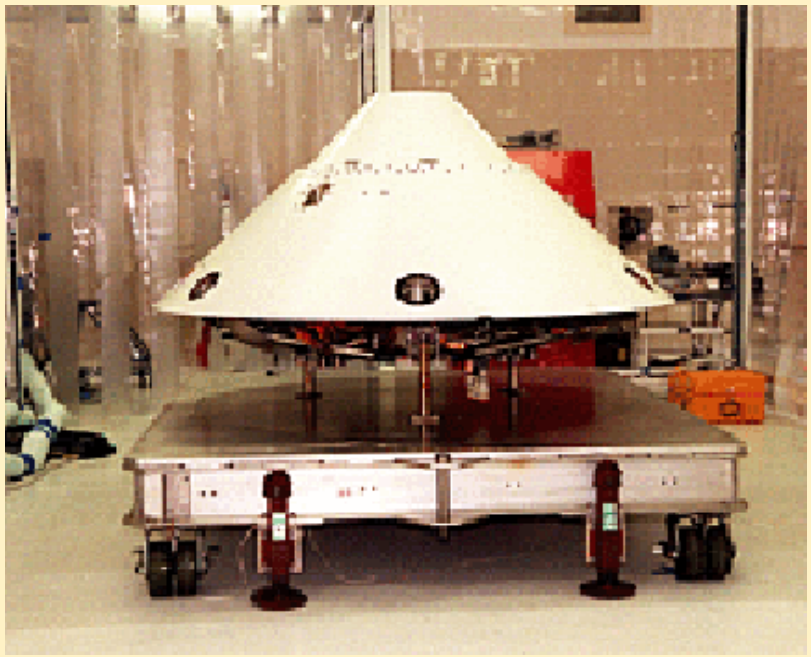
http://www4.cnn.com/TECH/space/9911/10/orbiter.03/#3

Mars Polar Lander

1999

Mars Polar Lander

- Launched January 3, 1999
- Two hours prior to reaching its Mars orbit insertion point on December 3, 1999, the spacecraft reported that all systems were good to go for orbit insertion
- There was no further contact
- US120,000,000



Mars Polar Lander

- “The most likely cause of the lander’s failure, investigators decided, was that a spurious sensor signal associated with the craft’s legs falsely indicated that the craft had touched down when in fact it was some 130-feet (40 meters) above the surface. This caused the descent engines to shut down prematurely and the lander to free fall out of the Martian sky.”

http://www.space.com/businesstechnology/technology/mpl_software_crash_000331.html

Mars Polar Lander

- Spurious signals – hard to test
  - By the way – this is an example of the type of requirement that might be covered in the external interfaces section (range of allowable input etc)
- But surely there had to be a better way to test for touch-down than vibrations in the legs

The Sleipner A Oil Platform

1991

The Sleipner A Oil Platform

- Norwegian Oil company's platform in the North Sea
- "When [it] sank in August 1991, the crash caused a seismic event registering 3.0 on the Richter scale, and left nothing but a pile of debris at 220m of depth."
- The failure involved a total economic loss of about $700 million."

http://www.ima.umn.edu/~arnold/disasters/sleipner.html

The Sleipner A Oil Platform

- Long accident investigation
- Traced the problem back to an incorrect entry in the Nastran finite element model used to design the concrete base. The concrete walls had been made too thin.
- When the model was corrected and rerun on the actual structure it predicted failure at 65m
- Failure had occurred at 62 m

The Pentium FDIV Bug

1994

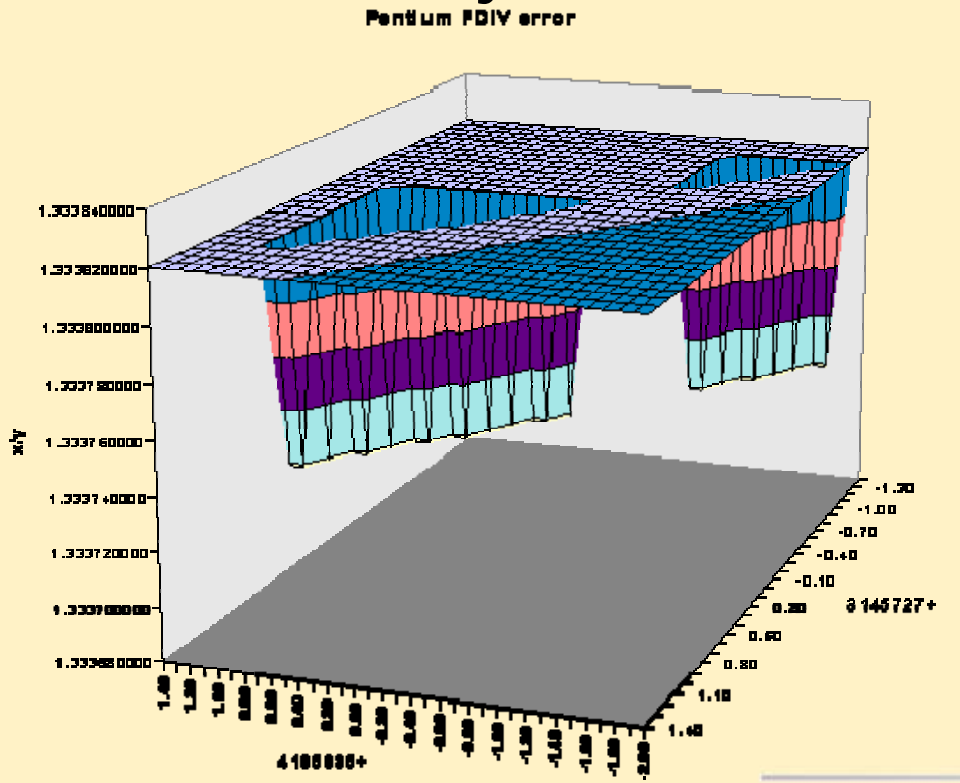
The Pentium FDIV Bug

- A programming error in a for loop led to 5 of the cells of a look-up table being not downloaded to the chip
- Chip was burned with the error
- Sometimes (4195835 / 3145727) * 3145727 – 4195835 =-192.00 and similar errors
- On older c1994 chips (Pentium 90)

http://www.mathworks.com/company/pentium/index.shtml

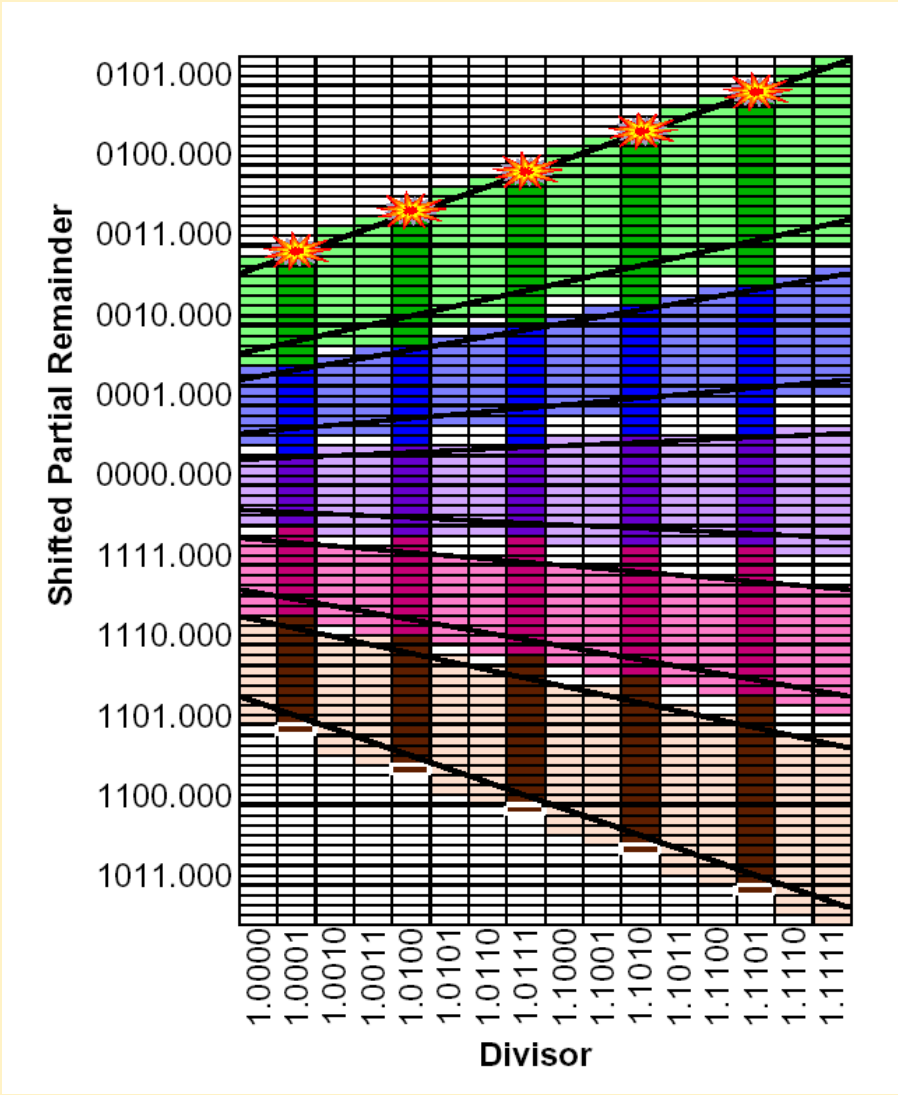
The Pentium FDIV Bug

- 4195833.0 <= x <= 4195836.4
- 3145725.7 <= y <= 3145728.4



The correct values all would round to 1.3338 and the incorrect values all would round to 1.3337, an error in the 5th significant digit.

Look-up Table



USS Yorktown

1998

USS Yorktown

- The Yorktown lost control of its propulsion system because its computers were unable to divide by the number zero, the memo said. The Yorktown's Standard Monitoring Control System administrator entered zero into the data field for the Remote Data Base Manager program.
- The ship was completely disabled for several hours

USS Yorktown

- This is such a dumb bug there is little need to comment!
- All input data should be checked for validity
- If you have a zero divide risk then trap it
- Particularly if it might bring down an entire warship
- And, even if a zero divide gets through, how robust is a system where a single user input out of range error can crash an entire ship?

Patriot

1991

Patriot

- On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soldiers.

Patriot

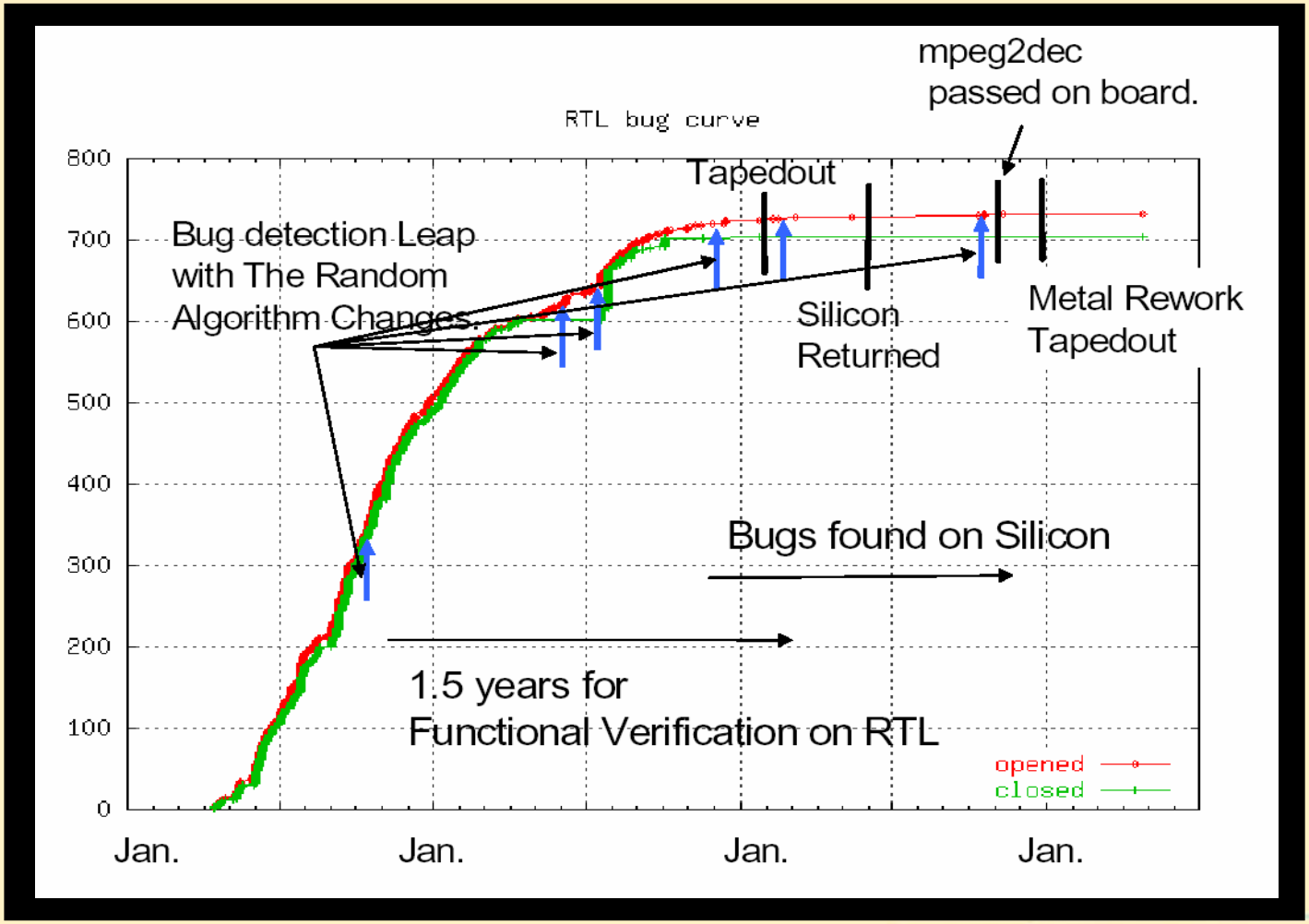
"The range gate's prediction of where the Scud will next appear is a function of the Scud's known velocity and the time of the last radar detection. Velocity is a real number that can be expressed as a whole number and a decimal (e.g., 3750.2563...miles per hour). Time is kept continuously by the system's internal clock in tenths of seconds but is expressed as an integer or whole number (e.g., 32, 33, 34...). The longer the system has been running, the larger the number representing time. To predict where the Scud will next appear, both time and velocity must be expressed as real numbers. Because of the way the Patriot computer performs its calculations and the fact that its registers are only 24 bits long, the conversion of time from an integer to a real number cannot be any more precise than 24 bits. This conversion results in a loss of precision causing a less accurate time calculation. The effect of this inaccuracy on the range gate's calculation is ***directly proportional to the target's velocity and the length of the system has been running****. Consequently, performing the conversion after the Patriot has been running continuously for extended periods causes the range gate to shift away from the center of the target, making it less likely that the target, in this case a Scud, will be successfully intercepted."*

Government Accounting Office Report http://www.fas.org/spp/starwars/gao/im92026.htm

Patriot

- This bug is typical of a requirements deficiency caused by reuse
- Patriot was originally an anti-aircraft (防空) system designed to remain “up” for short periods of time and to track slow (~mach 1-2) targets
- It was moved into a missile defence (反飛彈) role where it now had to be on station for many days and to track much faster targets

Design Productivity Crisis Hardware



Design Productivity Crisis Software

MARCH 13, 2003 THE DAILY YOMIURI

NATIONAL

FS

handed
ison term
himbun
son guard was
1½ years in
ee years, for ac-
or supplying
iba Prison in

rt also ordered
hi Hashimoto, 24,

oto gave an in-
e and other
r 2000 and April

aki Kuwabara, a
organized crime
al at the time,
ing a person.

dents face
es in N.Z.
e Japanese stu-

Air traffic control bug found

NEC neglected to fix defect

The Yomiuri Shimbun

A software bug left unfixed by NEC Corp. was found to be partly responsible for a computer breakdown that badly disrupted domestic flight schedules on March 1 and 2, Construction and Transport Ministry sources said Wednesday.

NEC Corp. had incorrectly programmed the computer system of an air traffic control facility in Tokorozawa, Saitama Prefecture, the sources said. Although the company knew about the defect, it did not take any measures to fix it, the sources said.

The trouble occurred in a device called the flight data processing system (FDP), which processes information on all flights from all the nation's airports, the sources said.

The computer is made up of about 1,000 data storage units. A preinstalled data processing program to read the data was replaced in September. But with the new program, the whole system can crash if the data in one unit is misread.

NEC discovered the defect at the end of January, but since the FDP system had been running without any problems since September, NEC allegedly considered the defect unlikely to cause serious trouble. The company therefore did not take any measures to fix the problem.

But when another computer program was updated on March 1, the location of the data in the storage units was changed. Soon after the data processing program was restarted, the system went down, according to the sources.

"In addition to NEC's programming error, our negligence in not fully checking the functioning of the renewed program caused the glitch," a Construction and Transport Ministry official said.

The official said the ministry would consider whether to claim compensation from NEC for damages resulting from the breakdown after fully reviewing the contents of their contract with the company.

The breakdown took place at the ministry's Air Traffic Control Center in Tokorozawa, Saitama Prefecture. On March 1 and 2, a total of 215 flights were canceled, and 1,462 flights were delayed for 30 minutes or more. At least 270,000 people flying on Japan Airlines, All Nippon Airways and Japan Air System were affected by the failure.

An NEC official said, "We'll reflect on the failure in our programming, which seriously inconvenienced many people, and do our best to prevent any recurrence of the trouble."

Eiichiro Sekigawa, a critic of aviation affairs, said: "The incident could have been worse. This time it only affected departing flights. That kind of trouble could affect air planes in flight in the future. NEC has reportedly said that it didn't report the trouble to the ministry, assuming that the problem was not a serious one. But I wonder how the company reached that decision. I urge them to take a strict approach when it comes to aviation safety and to thoroughly review their operations for improvements."

Ex-Nippon Food

Design Productivity Crisis
Internet Security

- Microsoft's Passport bug leaves 200 million users vulnerable
  - Passport accounts are central repositories for a person's online data as well as acting as the single key for the customer's online accounts.
  - The flaw, in Passport's password recovery mechanism, could have allowed an attacker to change the password on any account to which the username is known.
  - BBC, CNET news May 8, 2003

Conclusions

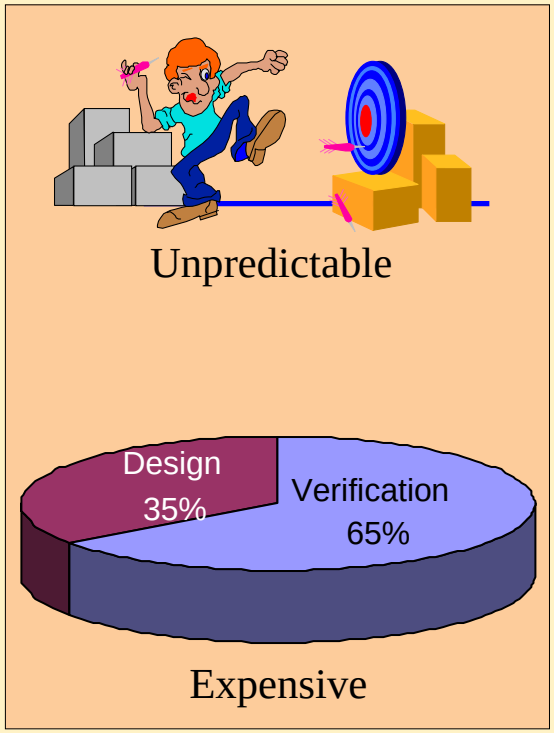
- According to Dept of Commerce's National Inst. of Standards and Technology (NIST), 2002
  - Software errors cost US economy $59.5 billion annually
  - An estimated $22.2 billion could be saved by improved testing infrastructures
  - Half of the errors are not found until “downstream” in the software design process

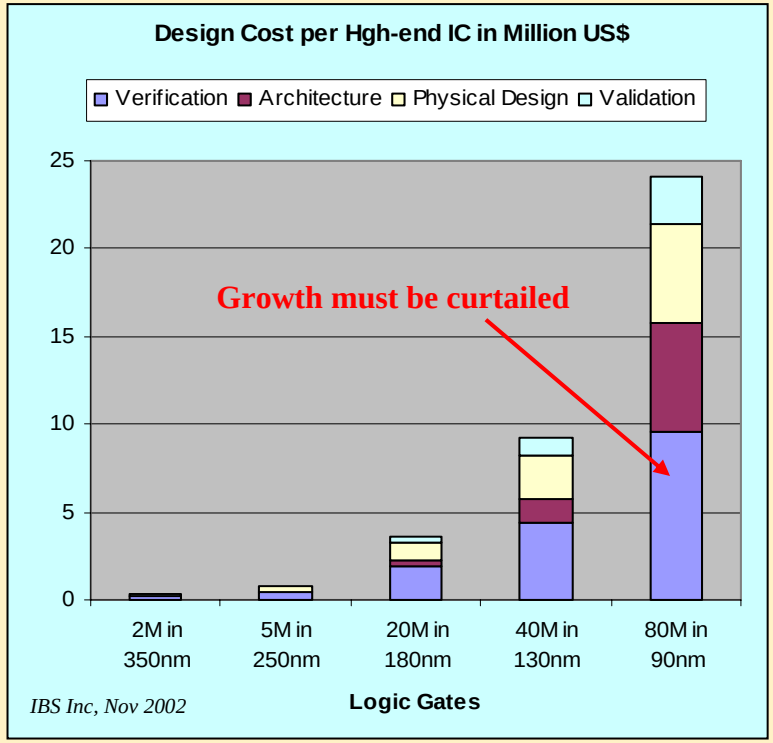
http://www.nist.gov/public_affairs/releases/n02-10.htm
http://www.nist.gov/director/prog-ofc/report02-3.pdf

Reality in System Design

- Computer systems are getting more complex and pervasive
  - Testing takes more time than designing
  - Automation is key to improve time-to-market
- In safety-critical applications, bugs are unacceptable
  - Mission control, medical devices
- Bugs are expensive
  - FDIV in Pentium: $4195835/3145727$

Reality in System Verification





Why Study Computer-Aided Verification?

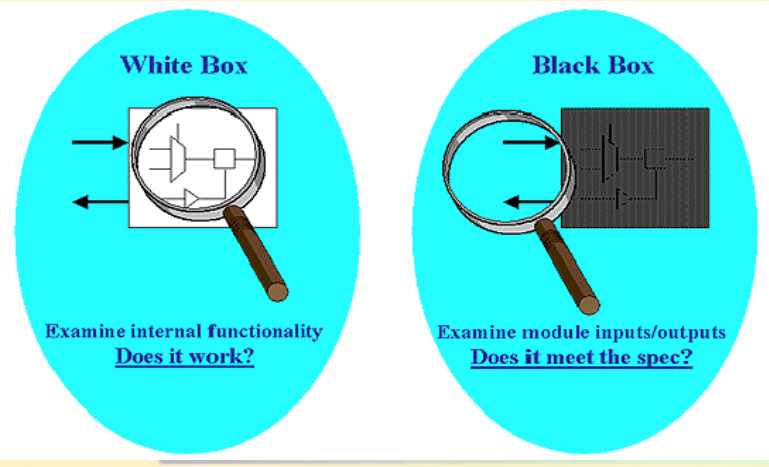
- A general approach with applications to
  - Hardware/software designs
  - Network protocols
  - Embedded control systems
- Rapidly increasing industrial interest
- Interesting mathematical foundations
  - Modeling, semantics, concurrency theory
  - Logic and automata theory
  - Algorithms analysis, data structures

Traditional Methods

- **White Box Testing**

1. Validate the implementation details with a knowledge of how the unit is put together.
2. Check all the basic components work and that they are connected properly.
3. Give us more confidence that the adder will work under all circumstances.

Example: Focus on validating an adder unit inside the controller.



Traditional Methods

- **Black Box Testing**

1. Focus on the external inputs and outputs of the unit under test, with no knowledge of the internal implementation details.
2. Apply stimulus to primary inputs and the results of the primary outputs are observed.
3. Validate the specified functions of the unit were implemented without any interest in how they were implemented.
4. This will exercise the adder but will not check to make sure that the adder works for all possible inputs

Example: Check to see if the controller can count from 1 to 10.

Traditional Methods

- Static Testing

1. Examine the construction of the design
2. Looks to see if the design structure conforms to some set of rules
3. Need to be told what to look for

- Dynamic Testing

1. Apply a set of stimuli
2. Easy to test complex behavior
3. Difficult to exhaustively test
4. It does not show that the design works under all conditions

Traditional Methods

- Random Testing

1. Generate random patterns for the inputs
2. The problems come from not what you know but what you don't know
3. You might be able to do this for data inputs, but control inputs require specific data or data sequences to make the device perform any useful operation at all

Formal Verification

- Goal: provide tools and techniques as design aids to improve reliability
- Formal: correctness claim is a precise mathematical statement
- Verification: analysis either proves or disproves the correctness claim

Formal Verification Approach

- Build a model of the system
  - What are possible behaviors?
- Write correctness requirement in a specification language
  - What are desirable behaviors?
- Analysis: check that model satisfies specification

Why Formal Verification?

- **Testing/simulation** of designs/implementations may not reveal error (e.g., no errors revealed after 2 days)
- **Formal verification** (=exhaustive testing) of design provides 100% coverage (e.g., error revealed within 5 min).
- **TOOL** support.
- No need of testbench, test vectors

Interactive versus Algorithmic Verification

- Interactive analysis
  - Analysis reduces to proving a theorem in a logic
  - Uses interactive theorem prover
  - Requires more expertise
  - E.g. Theorem Proving

Interactive versus Algorithmic Verification

- Algorithmic analysis
  - Analysis is performed by an algorithm (tool)
  - Analysis gives counterexamples for debugging
  - Typically requires exhaustive search of state space
  - Limited by high computational complexity
  - E.g. Model Checking, Equivalence Checking

Theorem Proving

- Prove that an implementation satisfies a specification by mathematical reasoning.
- Implementation and specification expressed as *formulas* in *a formal logic* .
- Relationship (logical equivalence/ logical implication) described as *a theorem* to be proven.
- **A proof system**:

  A set of axioms(facts) and inference(deduction) rules (simplification, rewriting, induction, etc.)

Theorem Proving

- **Some known theorem proving systems:**

  HOL PVS Lambda

- **Advantages:**

  High abstraction and powerful logic expressiveness

  Unrestricted applications

  Useful for verifying datapath- dominated circuits

- **Limitations:**

  Interactive (under user guidance)

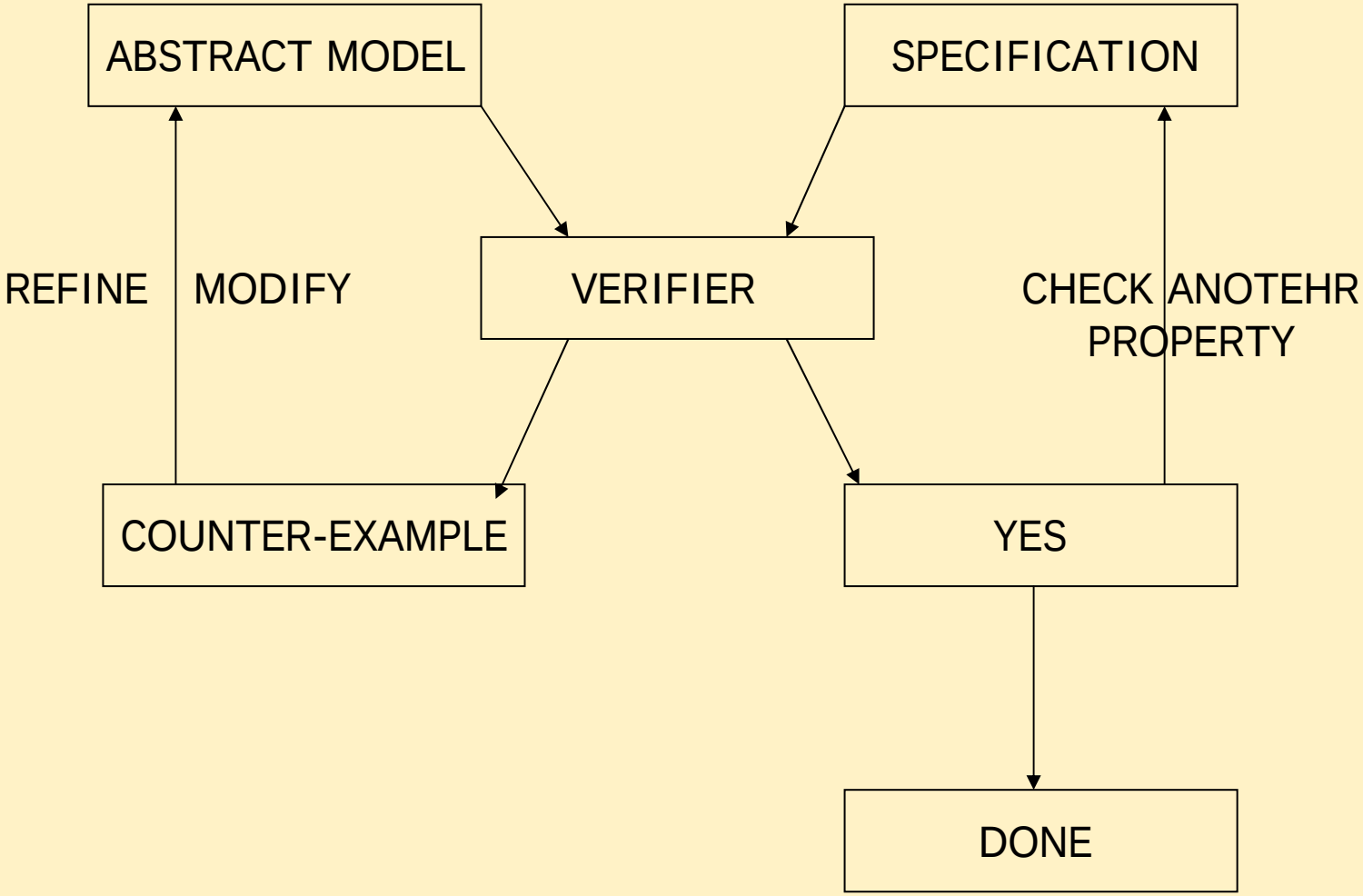
  Requires expertise for efficient use

  Automated for narrow classes of designs

Model Checking

- Term coined by Clarke and Emerson in 1981 to mean checking a finite-state model with respect to a temporal logic
- Applies generally to automated verification
  - Model need not be finite
  - Requirements in many different languages
- Provides diagnostic information to debug the model

Verification Methodology

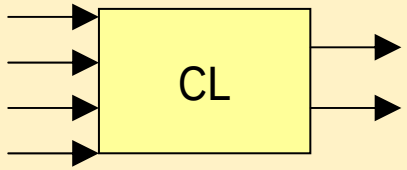


Equivalence Checking

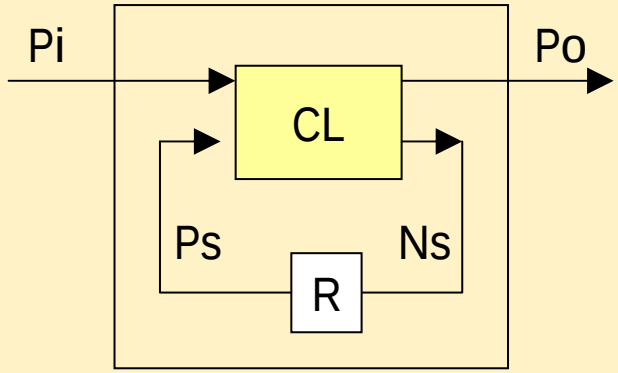
- Checks if two circuits are equivalent
  - Register-Transfer Level (RTL)
  - Gate Level
- Reports differences between the two
- Used after:
  - clock tree synthesis
  - scan chain insertion
  - manual modifications

Equivalence Checking

- Two circuits are functionally equivalent if they exhibit the same behavior
- Combinational Circuits
  - For all possible input **values**

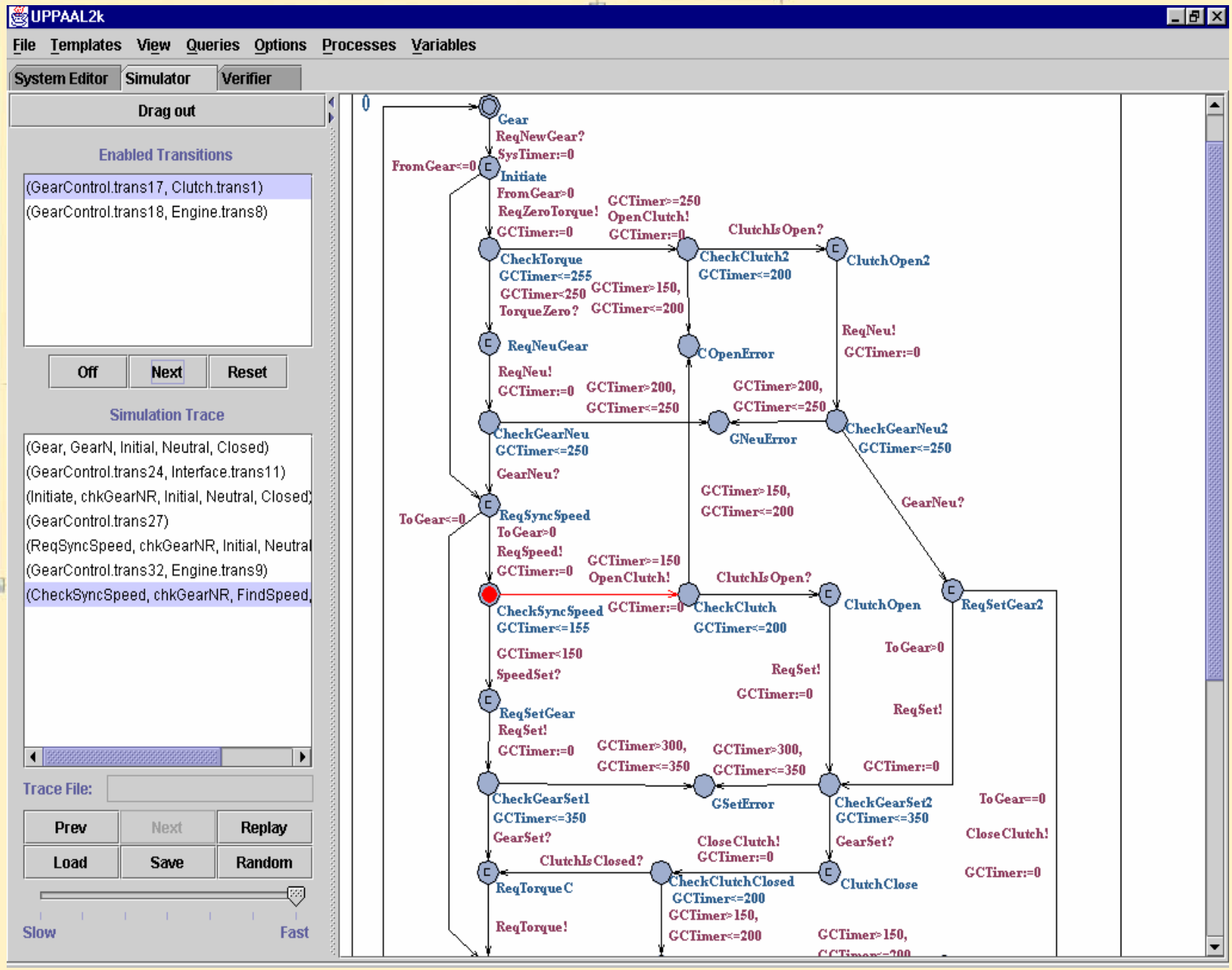


- Sequential Circuits
  - For all possible input **sequences**



Formal Verification Tools

- Protocol: UPPAAL, SGM, Kronos, …
- System Design (UML, …): visualSTATE
- Software: SPIN
- Hardware:
  - EC: Formality, Tornado
  - MC: SMV, FormalCheck, RuleBase, SGM, …
  - TP: PVS, ACL2

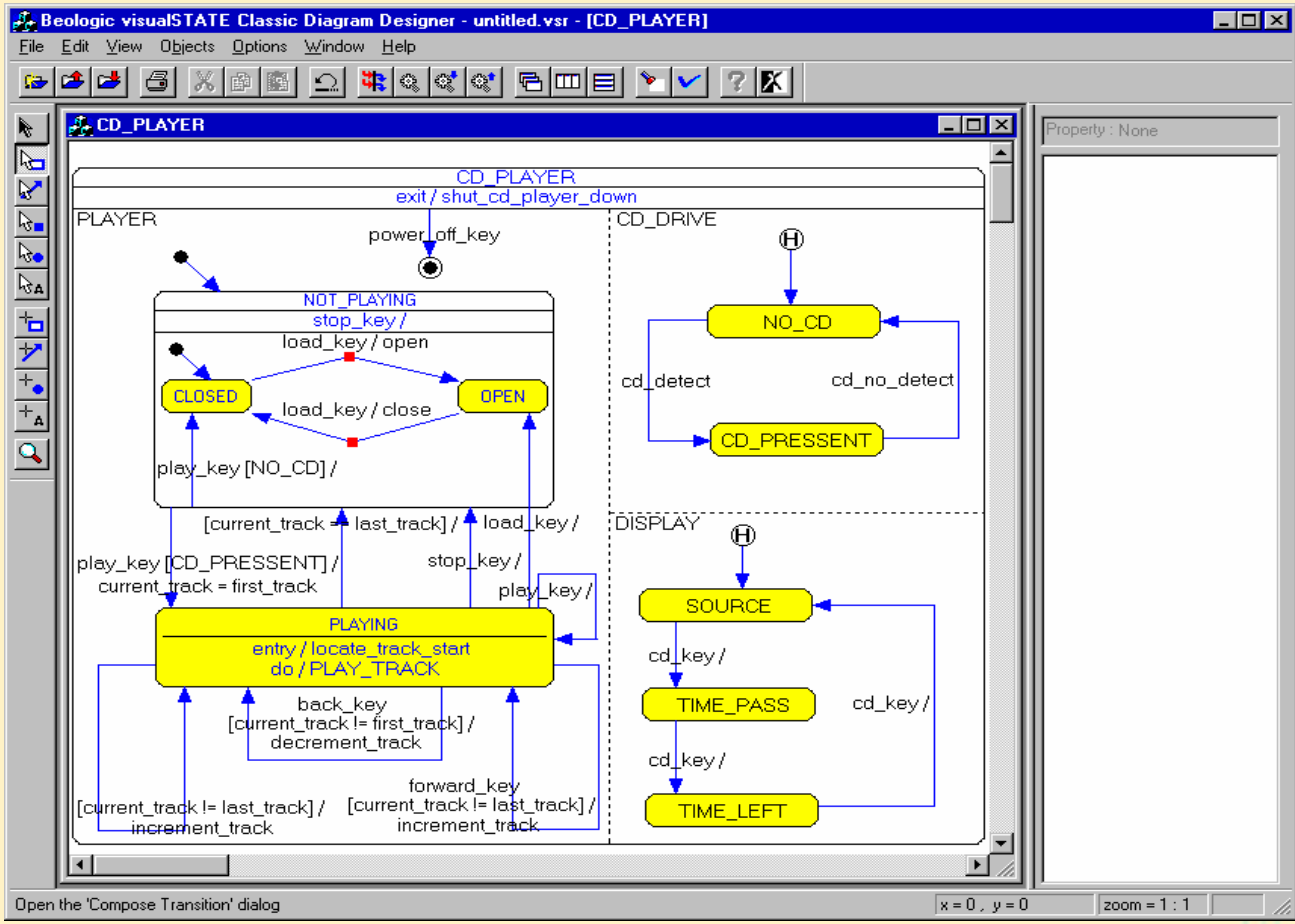

UPPAAL2k
File Templates View Queries Options Processes Variables
System Editor Simulator Verifier
Drag out
Enabled Transitions
(GearControl.trans17, Clutch.trans1)
(GearControl.trans18, Engine.trans8)
Off Next Reset
Simulation Trace
(Gear, GearN, Initial, Neutral, Closed)
(GearControl.trans24, Interface.trans11)
(Initiate, chkGearNR, Initial, Neutral, Closed)
(GearControl.trans27)
(ReqSyncSpeed, chkGearNR, Initial, Neutral
(GearControl.trans32, Engine.trans9)
(CheckSyncSpeed, chkGearNR, FindSpeed,
Trace File:
Prev Next Replay
Load Save Random
Slow Fast
0
Gear
ReqNewGear?
SysTimer:=0
FromGear<=0
Initiate
FromGear>0
ReqZeroTorque!
GCTimer:=0
GCTimer>=250
OpenClutch!
GCTimer:=0
ClutchIsOpen?
CheckTorque
GCTimer<=255
GCTimer<250
TorqueZero?
CheckClutch2
GCTimer<=200
ClutchOpen2
GCTimer>150,
GCTimer<=200
ReqNeuGear
COpenError
ReqNeu!
GCTimer:=0
ReqNeu!
GCTimer:=0
GCTimer>200,
GCTimer<=250
GCTimer>200,
GCTimer<=250
CheckGearNeu
GCTimer<=250
GNeuError
CheckGearNeu2
GCTimer<=250
GearNeu?
GearNeu?
GCTimer>150,
GCTimer<=200
ToGear<=0
ReqSyncSpeed
ToGear>0
ReqSpeed!
GCTimer:=0
GCTimer>=150
OpenClutch!
ClutchIsOpen?
CheckSyncSpeed
GCTimer<=155
GCTimer:=0
CheckClutch
GCTimer<=200
ClutchOpen
ReqSetGear2
GCTimer<150
SpeedSet?
ToGear>0
ReqSet!
GCTimer:=0
ReqSet!
ReqSetGear
ReqSet!
GCTimer:=0
GCTimer>300,
GCTimer<=350
GCTimer>300,
GCTimer<=350
GCTimer:=0
CheckGearSet1
GCTimer<=350
GSetError
CheckGearSet2
GCTimer<=350
ToGear==0
GearSet?
ClutchIsClosed?
CloseClutch!
GCTimer:=0
GearSet?
CloseClutch!
GCTimer:=0
ReqTorqueC
CheckClutchClosed
GCTimer<=200
GCTimer>150,
GCTimer<=200
ClutchClose
ReqTorque!
GCTimer>150,

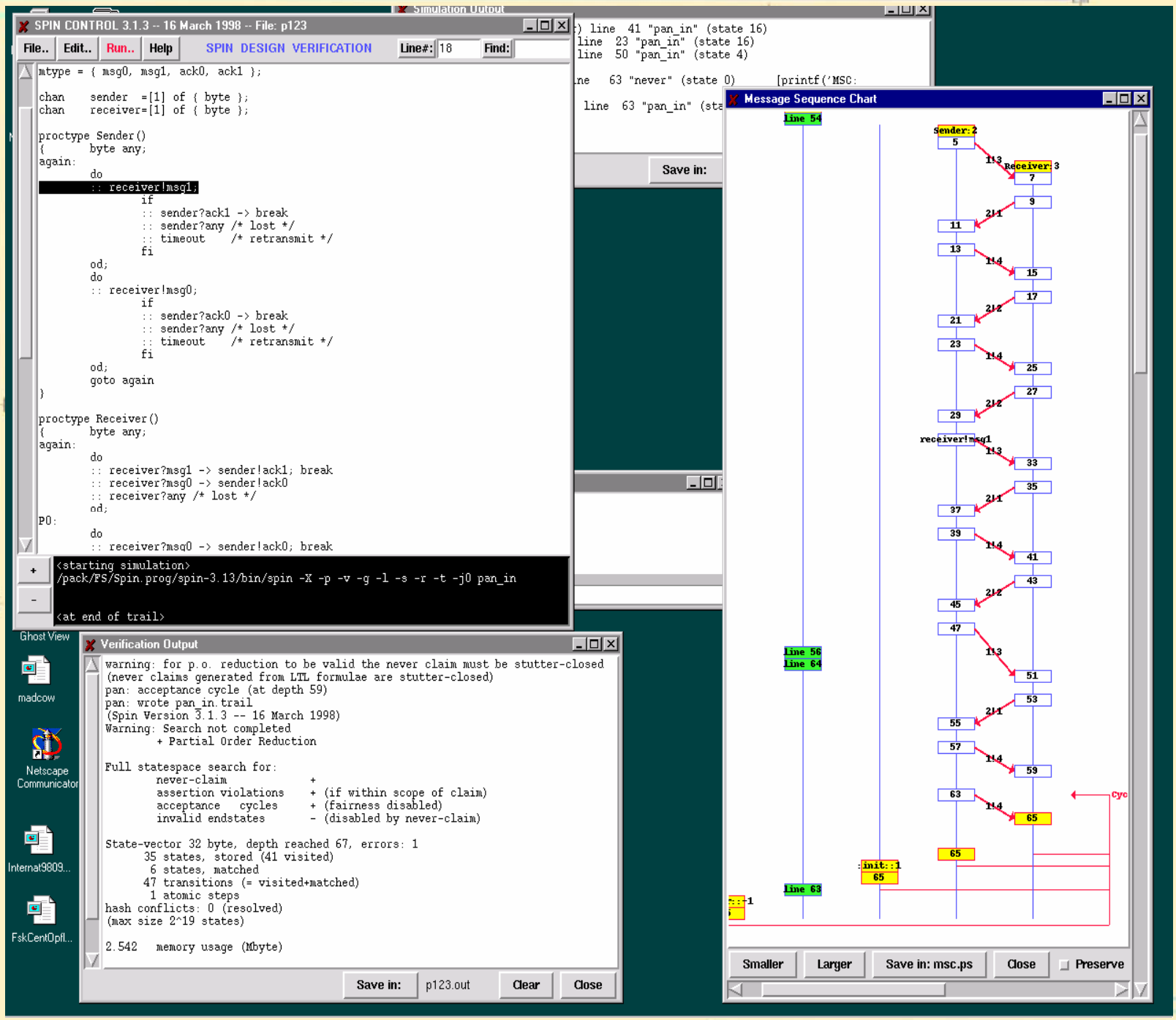
UPPAAL

visualSTATE

w Baan Visualstate, DTU (CIT project)

- Hierarchical state systems
- Flat state systems
- Multiple and inter-related state machines
- Supports UML notation
- Device driver access




SPIN CONTROL 3.1.3 -- 16 March 1998 -- File: p123
File..
Edit..
Run..
Help
SPIN DESIGN VERIFICATION
Line#: 18
Find:
mtype = { msg0, msg1, ack0, ack1 };
chan sender =[1] of { byte };
chan receiver=[1] of { byte };
proctype Sender()
{ byte any;
again:
do
:: receiver!msg1;
if
:: sender?ack1 -> break
:: sender?any /* lost */
:: timeout /* retransmit */
fi
od;
do
:: receiver!msg0;
if
:: sender?ack0 -> break
:: sender?any /* lost */
:: timeout /* retransmit */
fi
od;
goto again
}
proctype Receiver()
{ byte any;
again:
do
:: receiver?msg1 -> sender!ack1; break
:: receiver?msg0 -> sender!ack0
:: receiver?any /* lost */
od;
P0:
do
:: receiver?msg0 -> sender!ack0; break
<starting simulation>
/pack/FS/Spin.prog/spin-3.13/bin/spin -X -p -v -g -l -s -r -t -j0 pan_in
<at end of trail>
Simulation Output
) line 41 "pan_in" (state 16)
line 23 "pan_in" (state 16)
line 50 "pan_in" (state 4)
ne 63 "never" (state 0) [printf('MSC:
line 63 "pan_in" (sta
Save in:
Message Sequence Chart
line 54
Sender: 2
Receiver: 3
receiver!msg1
line 56
line 64
line 63
init::1
Cyc
Smaller
Larger
Save in: msc.ps
Close
Preserve
Verification Output
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
pan: acceptance cycle (at depth 59)
pan: wrote pan_in.trail
(Spin Version 3.1.3 -- 16 March 1998)
Warning: Search not completed
+ Partial Order Reduction
Full statespace search for:
never-claim +
assertion violations + (if within scope of claim)
acceptance cycles + (fairness disabled)
invalid endstates - (disabled by never-claim)
State-vector 32 byte, depth reached 67, errors: 1
35 states, stored (41 visited)
6 states, matched
47 transitions (= visited+matched)
1 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)
2.542 memory usage (Mbyte)
Save in:
p123.out
Clear
Close
Ghost View
madcow
Netscape Communicator
Internat9809...
FskCentOpfl...

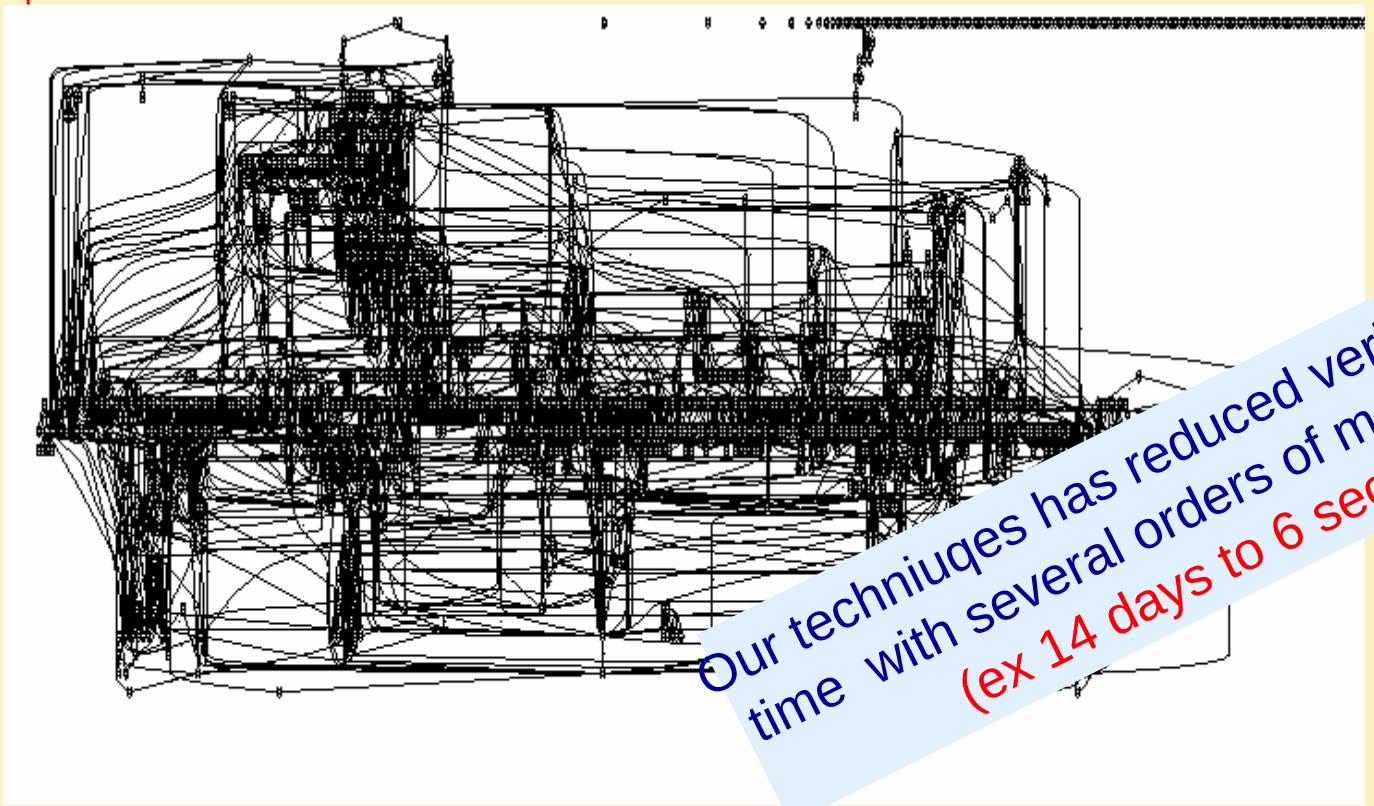
SPIN

VVS
visualSTATE

Train Simulator

1421 machines
11102 transitions
2981 inputs
2667 outputs
3204 local states
Declare state sp.: 10^476

BUGS ?



Our techniuqes has reduced verification time with several orders of magnitude (ex 14 days to 6 sec)

HW Verification Tools

Supplier	Tool Name	Class of Tool	HDL	Design Level
COMMERCIAL TOOLS				
Chrystalis	Design Verifier	Equiv. Checking	VHDL/Verilog	RTL/Gate
Synopsys	Formality	Equiv. Checking	VHDL/Verilog	RTL/Gate
Cadence	Affirma	Equiv. Checking	VHDL/Verilog	RTL/Gate
Compass	VFormal	Equiv. Checking	VHDL/Verilog	RTL/Gate
Verysys	Tornado	Equiv. Checking	VHDL/Verilog	RTL/Gate
Abstract Hardware Ltd.	Checkoff-E	Equiv. Checking	VHDL/Verilog	RTL/Gate
IBM	BoolsEye	Equiv. Checking	VHDL/Verilog	RTL/Gate
Cadence	FormalCheck	Model Checking	VHDL/Verilog	RTL
Abstract Hardware Ltd.	Checkoff-M	Model Checking	VHDL/Verilog	RTL/Gate
IBM	RuleBase	Model Checking	VHDL	RTL
Abstract Hardware Ltd.	Lambda	Theorem Proving	VHDL/Verilog	RTL/Gate
PUBLIC DOMAIN TOOLS				
CMU	SMV	Model Checking	own language	RTL
Cadence	Cadence SMV	Model Checking	Verilog	RTL
UC Berkeley	VIS	Model/Equ. Check.	Verilog	RTL/Gate
Stanford U.	Murphy	Model Checking	own language	RTL
Cambridge U.	HOL	Theorem Proving	(SML)	universal
SRI	PVS	Theorem Proving	(LISP)	universal
UT Austin/CLI	ACL2	Theorem Proving	(LISP)	universal

Hardware Verification

- Fits well in design flow
  - Designs in VHDL, Verilog
  - Simulation, synthesis, and verification
  - Used as a debugging tool
- Who is using it?
  - Design teams: Lucent, Intel, IBM, ...
  - CAD tool vendors: Cadence, Synopsis
  - Commercial model checkers: FormalCheck

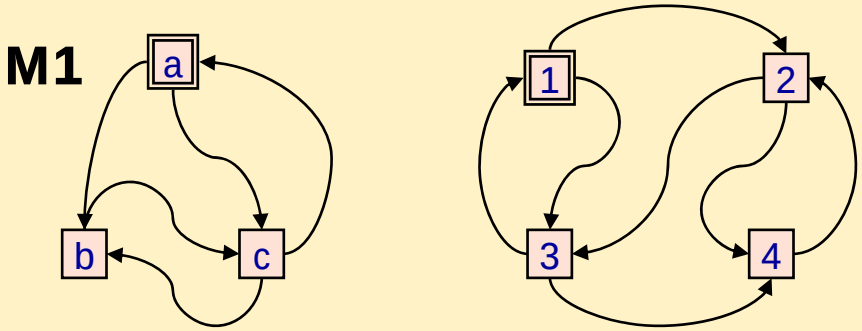
Software Verification

- Software
  - High-level modeling not common
  - Applications: protocols, telecommunications
  - Languages: ESTEREL, UML
- Recent trend: integrate model checking in programming analysis tools
  - Applied directly to source code
  - Main challenge: extracting model from code
  - Sample projects: SLAM (Microsoft), Feaver (Bell Labs)

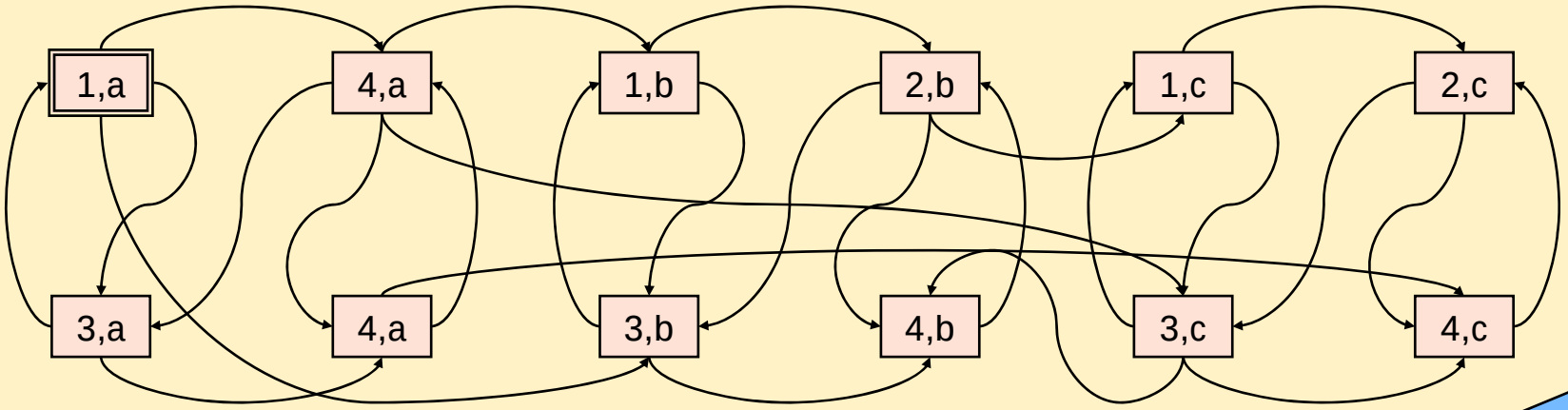
Limitations

- Appropriate for control-intensive applications
- Decidability and complexity remains an obstacle
- Falsification rather than verification
  - Model, and not system, is verified
  - Only stated requirements are checked
- Finding suitable abstraction requires expertise

'State Explosion' Problem



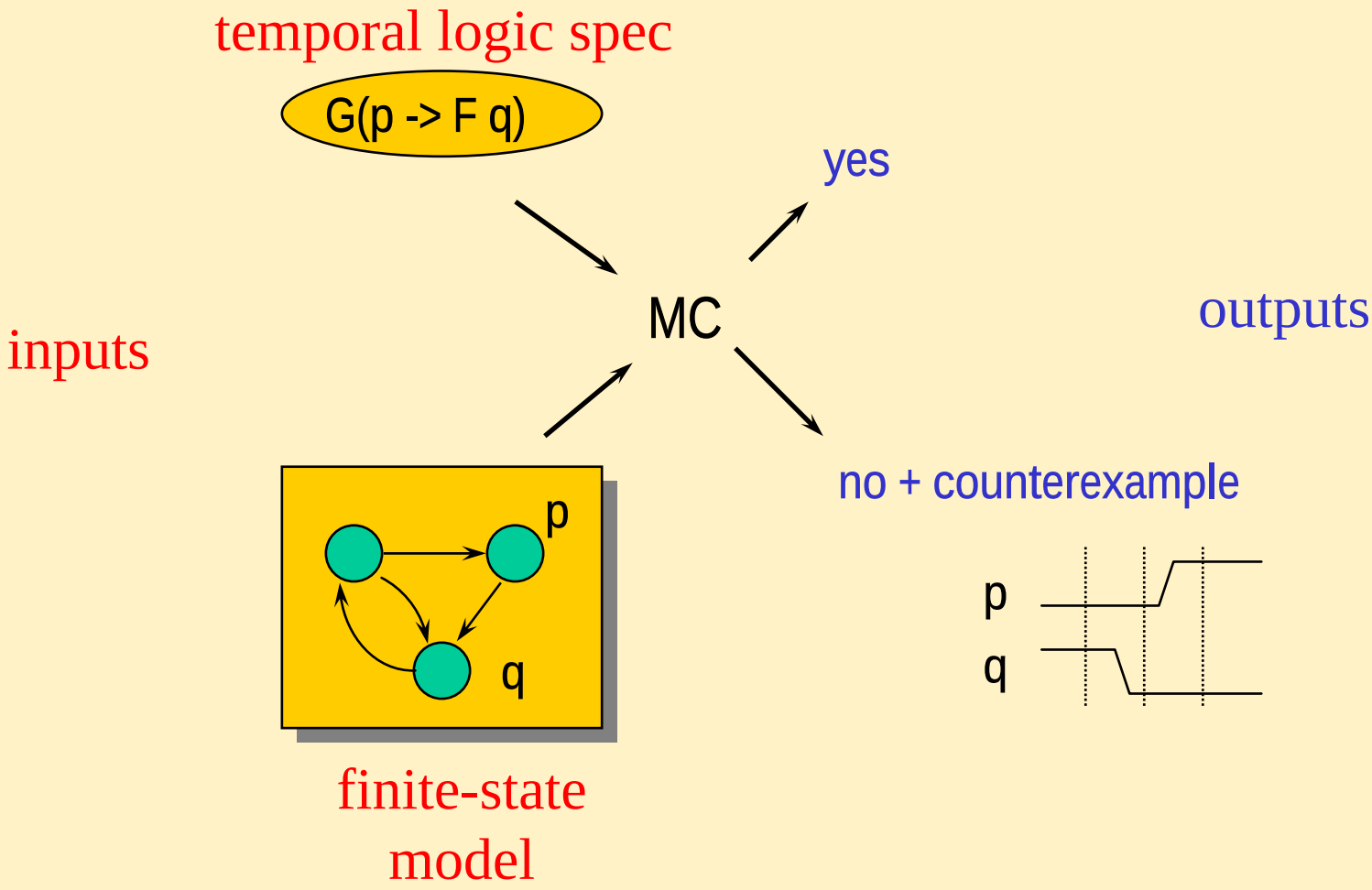
M1 x M2



All combinations = exponential in no. of compo

Provably theoretical intractable

Model Checking



Linear temporal logic (LTL)

- A logical notation that allows to:
  - specify relations in time
  - conveniently express finite control properties
- Temporal operators
  - G p "henceforth p"
  - F p "eventually p"
  - X p "p at the next time"
  - p U q "p until q"

Types of Temporal Properties

- **Safety** (nothing bad happens)

  G ~(ack1 & ack2) "mutual exclusion"

  G (req → (req W ack)) "req must hold until ack"

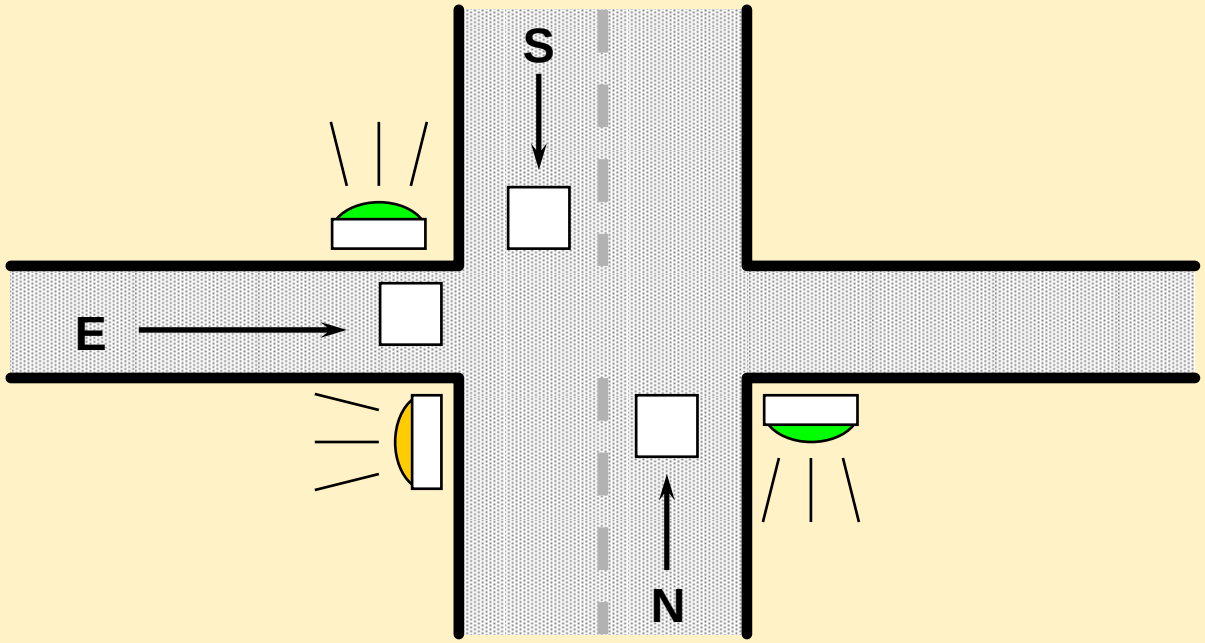
- **Liveness** (something good happens)

  G (req → F ack) "if req, eventually ack"

- **Fairness** (something good keeps happening)

  GF req → GF ack "if infinitely often req, infinitely often ack"

Example: Traffic Light Controller



- Guarantee no collisions
- Guarantee eventual service

Controller Program

```
module main(N_SENSE,S_SENSE,E_SENSE,
                    N_GO,S_GO,E_GO);
  input  N_SENSE, S_SENSE, E_SENSE;
  output N_GO, S_GO, E_GO;
  reg    NS_LOCK, EW_LOCK, N_REQ, S_REQ, E_REQ;

  /* set request bits when sense is high */

  always begin if (!N_REQ & N_SENSE) N_REQ = 1; end
  always begin if (!S_REQ & S_SENSE) S_REQ = 1; end
  always begin if (!E_REQ & E_SENSE) E_REQ = 1; end
```

Example continued...

```
/* controller for North light */
 always begin
    if (N_REQ)
      begin
        wait (!EW_LOCK);
        NS_LOCK = 1; N_GO = 1;
         wait (!N_SENSE);
        if (!S_GO) NS_LOCK = 0;
         N_GO = 0; N_REQ = 0;
      end
  end

/* South light is similar . . . */
```

Example code, cont…

```
/* Controller for East light */
 always begin
   if (E_REQ)
     begin
       EW_LOCK = 1;
       wait (!NS_LOCK);
       E_GO = 1;
       wait (!E_SENSE);
       EW_LOCK = 0; E_GO = 0; E_REQ = 0;
     end
 end
```

Specifications in temporal logic

- Safety (no collisions)

  G ~(E_Go & (N_Go | S_Go));

- Liveness

  G (~N_Go & N_Sense -> F N_Go);
  G (~S_Go & S_Sense -> F S_Go);
  G (~E_Go & E_Sense -> F E_Go);

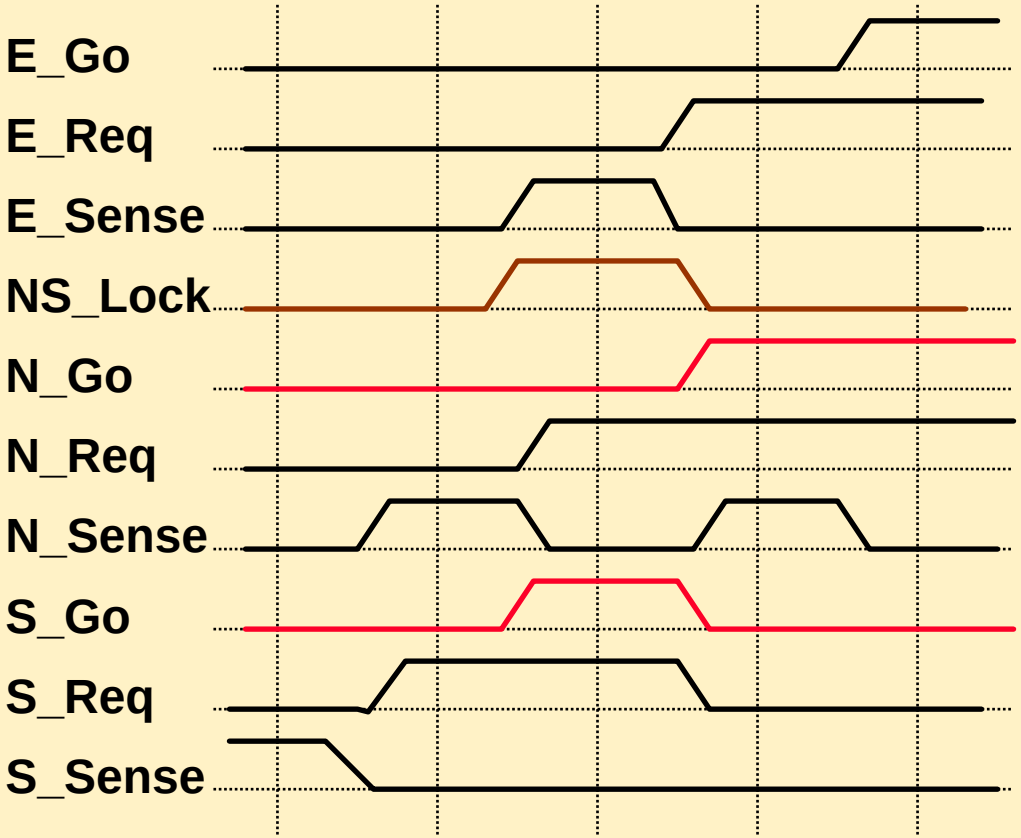
- Fairness constraints

  GF ~(N_Go & N_Sense);
  GF ~(S_Go & S_Sense);
  GF ~(E_Go & E_Sense);

/* assume each sensor off infinitely often */

Counterexample

- East and North lights on at same time...



N light goes on at same time S light goes off.

S takes priority and resets NS_Lock

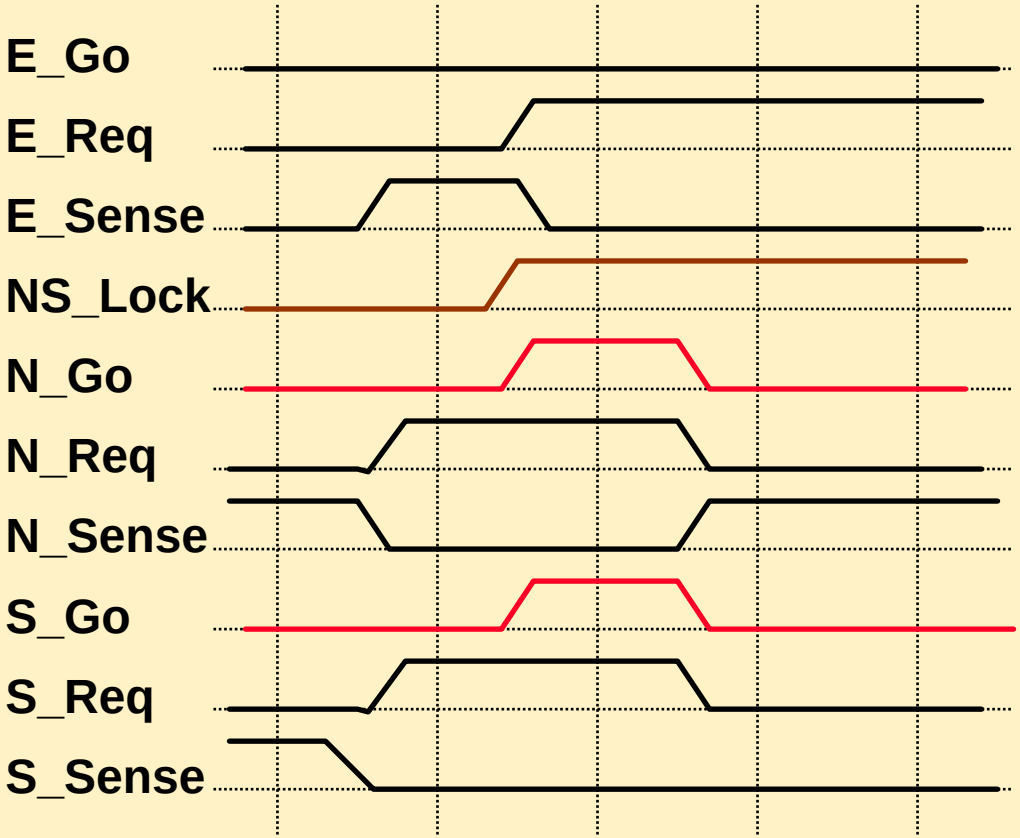
Fixing the error

- Don’t allow N light to go on while south light is going off.

```
always begin
   if (N_REQ)
     begin
       wait (!EW_LOCK & !(S_GO & !S_SENSE));
       NS_LOCK = 1; N_GO = 1;
       wait (!N_SENSE);
       if (!S_GO) NS_LOCK = 0;
                  N_GO = 0; N_REQ = 0;
     end
 end
```

Another counterexample

- North traffic is never served...



N and S lights go off at same time

Neither resets lock

Last state repeats forever

Fixing the liveness error

- When N light goes off, test whether S light is also going off, and if so reset lock.

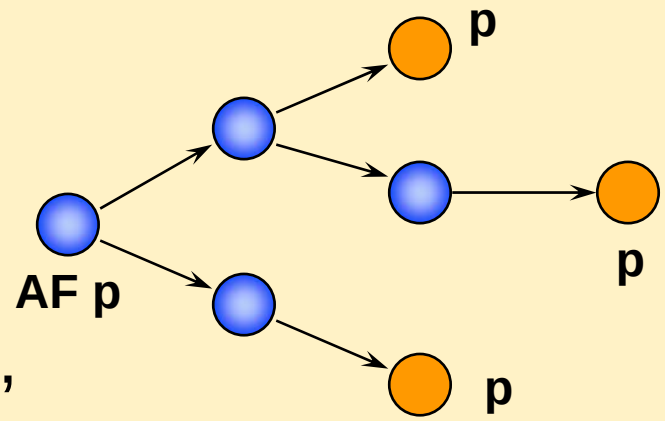
```
always begin
   if (N_REQ)
     begin
       wait (!EW_LOCK & !(S_GO & !S_SENSE));
       NS_LOCK = 1; N_GO = 1;
       wait (!N_SENSE);
       if (!S_GO | !S_SENSE) NS_LOCK = 0;
                  N_GO = 0; N_REQ = 0;
     end
 end
```

All properties verified

- Guarantee no collisions
- Guarantee service assuming fairness
- Computational resources used:
  - 57 states searched
  - 0.1 CPU seconds

Computation tree logic (CTL)

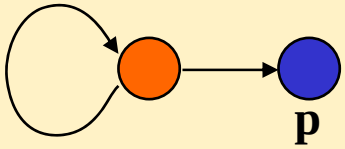
- Branching time model
- Path quantifiers
  - A = “for all future paths”
  - E = “for some future path”
- Example: AF p = “inevitably p”



- Every operator has a path quantifier
  - AG AF p instead of GF p

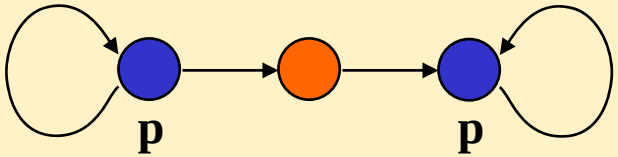
Difference between CTL and LTL

- Think of CTL formulas as approximations to LTL
  - AG EF p is weaker than G F p



Good for finding bugs...

  - AF AG p is stronger than F G p



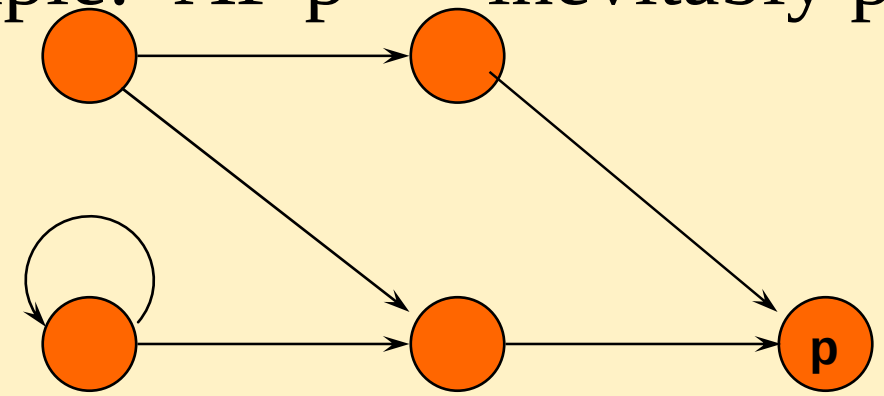
Good for verifying...

- CTL formulas easier to verify

So, use CTL when it applies...

CTL model checking algorithm

- Example: AF p = “inevitably p”

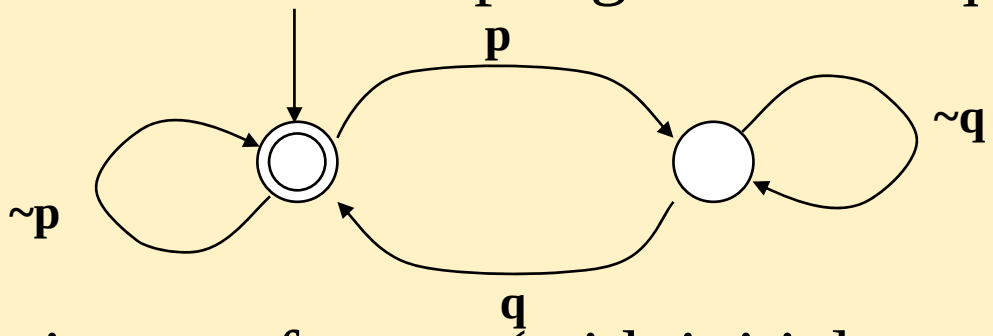


  - Complexity
    - linear in size of model (FSM)
    - linear in size of specification formula

Note: general LTL problem is exponential in formula size

Specifying using ω–automata

- An automaton accepting infinite sequences **G (p -> F q)**



  - Finite set of states (with initial state)
  - Transitions labeled with Boolean conditions
  - Set of accepting states

Interpretation:

- **A run is accepting if it visits an accepting state infinitely often**
- **Language = set of sequences with accepting runs**

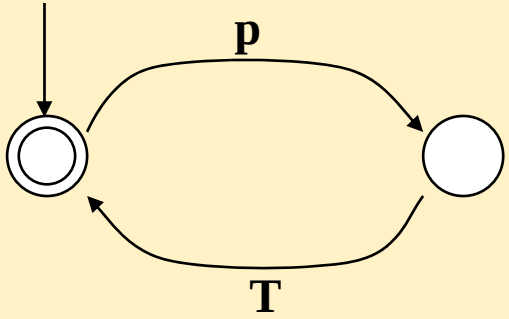
Verifying using ω–automata

- Construct parallel product of model and automaton
- Search for “bad cycles”
  - Very similar algorithm to temporal logic model checking
- Complexity (deterministic automaton)
  - Linear in model size
  - Linear in number of automaton states
  - Complexity in number of acceptance conditions varies

Automata vs. Temporal Logic

- Tableau procedure
  - LTL formulas can be translated into equivalent automata
  - Translation is exponential
- ω-automata are strictly more expressive than LTL

Example:



“p at even times”

- LTL with “auxiliary” variables = ω-automata

Example:

G (even -> p)

where:

init(even) := 1;

next(even) := ~even;

Overview of Topics

- Introduction to model checking
- System modeling and logic specification
  - Automata, reactive modules, temporal logics
- Analysis techniques
  - Explicit/Symbolic model checking
  - Bounded model checking (BMC) using Boolean satisfiability (SAT)
- Model checker implementation techniques
  - State-space reduction techniques
  - Compositional, assume-guarantee reasoning
- Advanced issues
  - Prioritized and urgent systems
  - Coverage analysis for model checking
  - Control/Program synthesis (optional)