Software is ...

...one of the most complex man made artifacts

"I believe the [spreadsheet product] I’m working on now is far more complex than a 747 (jumbo jet airliner)"
-- Chris Peters (Microsoft, 1992)

"It’s different [from other engineering disciplines] in that we take on novel tasks every time. The number of times [civil engineers] make mistakes is very small. And at first you think, what’s wrong with us? It’s because it’s like we’re building the first skyscraper every time.”
-- Bill Gates (Microsoft, 1992)
Software is ...

...one of the most complex man made artifacts

Microsoft Word is ... 1 million lines of code
Microsoft NT ... 16 million lines of code
Even pacemakers have 100 thousand lines of code ...

... but perhaps software complexity shouldn't even be measured in terms of lines of code, but instead, in terms of number of states

States >> SLOC

- The size of a system is sometimes more accurately expressed using semantic point of view
  - the number of different states a system can reach
  - ... an integer has 4.2 billion possible values
  - ... an object with 2 ints and a boolean field has 40 thousand quadrillion values
- How about Windows NT?
Software is...

...critical to the conduct of modern life.

- Process Control (oil, gas, water, ...)
- Transportation (air traffic control, ...)
- Health Care (patient monitoring, device control ...)
- Finance (automatic trading, bank security ...)
- Defense (intelligence, weapons control, ...)
- Manufacturing (precision milling, assembly, ...)

Failing software costs money and lives!

Failing Software Costs Money

- Thousands of dollars for each minute of factory down-time
- Huge losses of monetary and intellectual investment
  - Rocket boost failure (e.g., Arianne 5)
- Business failures associated with buggy software (Ashton-Tate dBase)
Failing Software Costs Lives

- Potential problems are obvious:
  - Software used to control nuclear power plants
  - Air-traffic control systems
  - Spacecraft launch vehicle control
  - ....
- A well-known and tragic example
  - Therac-25 radiation machine failures

Software is ...

...becoming the dominant component of society’s infrastructure.

In the future...

- Everything will be monitored/controlled
  - networked watches, clothes, ...
  - autonomous vehicles, intelligent highways, ...
  - virtual X rather than physical X
- These systems may not have manual backup
  - no workarounds for y2k-like problems
- Failures will be very costly and dangerous
Software is ...

... what you'll be building after graduation.

- You’ll be developing systems in 2020+
  - in the context we just mentioned
- Given the importance of software
  - you may be regulated, licensed
  - you may be liable for errors
  - your job may depend on your ability to produce reliable systems

Current Software Development Methods Are Insufficient

- Testing
  - samples execution behavior, misses some
- Systematic Inspections
  - don’t scale very well, although they are thorough
- Rigorous development processes
  - helping but most organizations don’t apply them

**Formal methods are becoming more popular**
Software Failures

NASA Mars Pathfinder Mission

- Rover's watchdog timer observed tasks missing their deadlines due to delays caused by a priority inversion problem.
- Each such missed deadline led to a system reset and a one-day delay in retransmission of data which wasted valuable mission time.
- This type of error could have been detected using model-checking.

Software Failures

Ariane 5 Rocket

- Ariane 5 software reused old code from Ariane 4 that was not respecified and retested in new environment
- Code in question performed floating point calculations
- Ariane 5 (being more powerful than Ariane 4) caused unanticipated floating-point exception (which would have never occurred on Ariane 4), causing an exception to be thrown which was not caught
- Triggered automatic destruction: $500 million loss
Software Failures

Reusing a shared variables for different purposes and a race condition on that variable led to a situation where lethal doses of radiation were given.

Several deaths resulted from this software error.

Reasoning About Concurrent Systems Is Hard

Very hard to predict all possible ways in which thread execution steps can be interleaved.

Often hard to determine/predict what sequences of actions the environment of a system may generate.

If you’re not convinced, let’s consider a few very small examples...
**Reasoning About Concurrent Systems is Hard**

```java
public class Job extends Thread {
    Container objref;
    Object x;

    public Job incr () {
        synchronized (objref) {
            objref.counter = objref.counter + 1;
        }
        return this;
    }

    public void setref(Container o) {
        objref = o;
    }

    public void run() {
        for (int i=0; i++; i<3) {
            incr();
            incr();
        }
    }
}
```

```java
public class Container {
    public int counter;
}
```

```java
public class Apprentice {
    public static void main(String[] args) {
        Container c1 = new Container();
        Container c2 = new Container();
        Job j1 = new Job();
        Job j2 = new Job();
        j1.setref(c2); j2.setref(c1);
        j1.start(); j2.start();
        j1.setref(c1);
    }
}
```

**Does the value of counter ever decrease?**

Source: J.S. Moore, George Porter "Proving Properties of Java Threads".

**Reasoning About Concurrent Systems is Hard**

```plaintext
1 Boolean array b(0;1) integer k, i, j;
2 comment process i, with i either 0 or 1 and j = 1 - i;
3 C0: b(i) := false;
4 C1: if k = i then begin
5 C2: if not (b(j)) then go to C2;
6 else k := i; go to C1 end;
7 else critical section
8 b(i) := true;
9 remainder of program
10 go to C0;
11 end
```

**Consider two processes 0 and 1 each running the pseudo-code above. Does the code guarantee mutual exclusion of the critical section (i.e., does the code ensure that the two processes can never be in the respective critical sections at the same time)?**

Source: Comm. of the ACM, Vol. 9, No. 1, p. 45
For You To Do...

- Pause the lecture.
- Study the examples on the previous two slides, answer the question given at the bottom of each slide and be able to justify your answer.

Classes of Requirements

- Safety Requirement
  - The system never reaches a “bad state”

- Liveness Requirement
  - The system eventually arrives at a “good state”

Requirements for concurrent systems often take one of these two forms.
Safety Requirements

The system never reaches a “bad state”

- The system never reaches a state where the current value of the counter is less its value in a previous state.
- The system never reaches a state where both process are in their critical regions at the same time.
- Any invariant...
  - Variable x is always greater than 5
    - In other words, the system never reaches a state where x is not greater than 5
- The system never reaches a deadlocked state

Liveness Requirements

The system eventually reaches a “good state”

- The system eventually reaches a point where it terminates (no further step is possible).
- If a button is pushed, the system eventually reaches a state where an acknowledgement is sent.
- A enabled process is not infinitely delayed in execution (freedom from livelock and starvation).
- Any process that wishes to enter its critical section will eventually succeed.
- A process that has a resource will eventually release it.

Identify the good states in each of the above requirements.
**Goal: Increase Software Reliability**

*Trends:*
- Size, complexity, concurrency, distributed
- Cost of software engineer
- Cost of CPU cycle

*Future: Automated Fault Detection*

---

**The Dream**

Program:
```java
void add(Object o) {
    buffer[head] = o;
    head = (head+1)%size;
}

Object take() {
    if (size > tail + 1) {
        return buffer[tail];
    }
    tail = (tail+1)%size;
    return buffer[tail];
}
```

Property 1: ...
Property 2: ...

Requirement:

Checker

OK
or
Error trace
Model Checking

Finite-state model → Model Checker → OK or Error trace

Temporal logic formula: $\Box (\Phi \rightarrow \diamond \Omega)$

A Success In Hardware

- Every major hardware manufacturer uses model-checking in their quality assurance process
  - AT&T, Cadence, Fujitsu, HP, IBM, Intel, Motorola, NEC, SGI, Siemens, Sun
- Example successes
  - Gigamax distributed multi-processor
  - IEEE Futurebus+ standard
  - Cray SV1 Supercomputer memory arbiter
Why Try to Use Model Checking for Software?

- Automatically check, e.g.,
  - invariants, simple safety & liveness properties
  - absence of dead-lock and live-lock
  - complex event-sequencing properties

"Between the window open and the window close, button X can be pushed at most twice."

- In contrast to testing, gives complete coverage by exhaustively exploring all paths in system,
- It’s been used for years with good success in hardware and protocol design

This suggests that model-checking can complement existing software quality assurance techniques.

Software Model-checking Efforts

- Lucent/Bell Labs (Spin)
- NASA (Java PathFinder)
- Microsoft Research (SLAM)
- IBM (CANVAS)
- UC Berkeley (BLAST)
- Kansas State (Bandera)
- ...several other projects from academia
Spin

- Developed at Lucent/Bell Labs by Gerard Holzmann
- Most widely-used model-checking tool
- Designed primarily for protocol verification
- Been used in major industrial telephony applications (e.g. PathStar)
- Winner of the 2001 ACM Software Award
  - for software that has had a lasting influence, reflected in contributions to concepts, in commercial acceptance, or both.
  - Unix, TCP/IP, Apache were previous recipients.

Java PathFinder (J PF)

- Developed at NASA Ames Automated Software Engineering Lab
  - http://ase.arc.nasa.gov/visser/jpf/
- Works directly on Java byte-code
- Includes several interesting heuristic search strategies
- Being in several case-studies involving safety and mission-critical software
SLAM

- Developed at Microsoft Research
- Works on sequential C code
- Targeted toward verification of Windows device drivers
- Includes interesting automated abstraction and refinement facilities

Canvas

- Developed at IBM’s T.J. Watson Research Center
- Targeted toward verification of Java components
- Includes interesting automated abstraction techniques
Bandera

- Developed at Kansas State University
  - [http://www.cis.ksu.edu/bandera](http://www.cis.ksu.edu/bandera)
- Translates concurrent Java programs to model descriptions that can be processed by several different model-checkers (including Spin and JPF)
- Includes a number of interesting model-reduction techniques including slicing and data abstraction, as well as other facilities for property specification and counter-example visualization

How does one describe systems to a model-checker?

- Hand-coded model built using the high-level system description language of a model-checking tool
  - We will use this approach with Spin
- Source Code
  - We will use this approach with Bandera
- Other possibilities
  - formal requirements, partial behavioral descriptions (UML collaboration diagrams & sequence charts), etc.
  - we will not consider these in detail in this course
SPIN Example

```
#define N 5 /* nr of processes (use 5 for demos) */
#define I 3 /* node given the smallest number */
#define L 10 /* size of buffer (>= 2*N*/

mtype = {one, two, winner};
chan q[N] = [L] of {mtype, byte};
byte nr_leaders = 0;

proctype node (chan in, out; byte mynumber)
{
  bit Active = 1, know_winner = 0;
  byte nr, maximum = mynumber, neighbourR;
  xr in;
  xs out;

  printf("MSC: %d\n", mynumber);
  out!one(mynumber);
end:

  do
  :: in?one(nr) -> if :: Active ->
    if :: nr != maximum ->
      out!two(nr);
      neighbourR = nr
    :: else ->
      assert(nr == N);
      know_winner = 1;
      out!winner, nr;
    fi
  :: else -> out!one(nr)
  fi
```

Fragment of Promela specification of Leader Election protocol

How does one describe systems to a model-checker?

- Temporal Logic
  - Linear Temporal Logic (LTL)
  - Computation Tree Logic (CTL)
- Temporal Specification Patterns
  - pattern-based approach to constructing specifications
- Automata ("never claims")
  - graphical
  - textual
- Other graphical notations
  - Timeline Editor (from Lucent)
  - Graphical Interval Logic
LTL Examples

Requirement: For any state, a request (for some resource) will eventually be acknowledged

Formal LTL Specification

\[ (\text{requested} \rightarrow <> \text{acknowledged}) \]

Requirement: An upwards travelling elevator at the second floor does not change its direction when it has passengers waiting to go to the fifth floor

Formal LTL Specification

\[ (\text{floor}=2 \&\& \text{direction}=\text{up} \&\& \text{button5pressed}) \rightarrow (\text{direction}=\text{up} \lor \text{floor}=5) \]

Specification Pattern Examples

Requirement: If a buffer becomes full, it will eventually become non-full.

Bandera Specification:

\[ \text{FullToNonFull} : \]
\[
\forall b : \text{BoundedBuffer}. \]
\[
\{ ! \text{Full}(b) \} \text{ responds to } \{ \text{Full}(b) \} \text{ globally;}
\]

Requirement: Empty buffers must be added to before being taken from

Bandera Specification:

\[ \text{NoTakeWhileEmpty} : \]
\[
\forall b : \text{BoundedBuffer}. \]
\[
\{ \text{take.\ Return}(b) \} \text{ is absent after } \{ \text{Empty}(b) \} \]
\[
\{ \text{add.\ Call}(b) \};
\]
**SPIN Never-claim Example**

**Requirement:** Eventually, the system will arrive at a state where there is always exactly one leader.

**LTL formalization:** $<>[] \text{oneLeader}$

**Never-claim formalization:**
```
define oneLeader (nr_leaders == 1) never { /* !(<>[] oneLeader) */
T0_init:
    if :: (1) -> goto T0_init
    :: (! (<>[] oneLeader)) -> goto accept_S5
    fi;
accept_S5:
    if :: (1) -> goto T0_init
    fi;
accept_all:
    skip }
```

**Timeline Edit w/ Automaton**

**Trigger event**

**Required event**

...with condition

"Monitoring" automaton
When is Model-checking Applied?

- Many opportunities
  - Requirements analysis
    - e.g., checking properties of message sequence charts
  - Design
    - e.g., checking properties of design notations such as UML statecharts
    - crafting design models in the input notations of model-checking tools such as Spin
  - Code unit testing
    - check code directly using Bandera, JPF, etc.
    - create a hand-crafted model by inspecting the code (less common these days)
  - Code integration testing
    - again, check code directly using Bandera, etc. (less common because model-checking is expensive and is best applied to code units)
    - checking that code conforms to interface specifications (hot research area)
Course Objectives

- Understand the challenges involved with designing concurrent/reactive software
- Be able to formalize typical requirements of concurrent systems in the various specification formalisms processed by model-checking tools
- Be able to effectively apply a modeling tool like Spin in the process of designing concurrent systems
- Be able to effectively apply a tool like Bandera or JPF to check properties of concurrent Java implementations
- Know the basic algorithms and implementation strategies for explicit-state model-checking to the point where you could code your own simple model-checker

In this course ...

- You will study various tools and techniques for debugging and verifying properties of concurrent systems (software, in particular).
  - BIR Checker: a model-checker dedicated to the Bandera intermediate representation (BIR)
  - Spin: system designed for verifying protocols based on communicating FSA
  - Bandera: tool set designed for model-checking concurrent Java software
  - JPF: model-checker that works directly on Java bytecodes
In this course ...

- You will learn the basic algorithms and data structures used in a model-checker
  - You will program several versions of a model-checker for BIR
  - Small programming assignments using Java
  - You will study the formal semantics of various abstraction and slicing techniques used for software model-checking

In this course ...

- In mini-project components, you will apply Bandera and other model-checking engines (e.g., Spin, dSpin, JPF) to check properties of medium-size Java systems.
  - Formalize system requirements in e.g., Bandera’s specification language
  - Identify appropriate code units and test harnesses for to be used in checking
  - Perform abstractions and other model-reduction techniques required for obtaining a tractable model
  - Write multiple documents describing each phase of the project
Summary

- Software is becoming pervasive and very complex
- Model-checking is an effective technique for modeling, debugging, and verifying properties of concurrent systems
- Multiple projects are attempting to apply model-checking throughout the development process
- We will learn the basic principles of explicit state model-checking and methods for applying it effectively to real-world concurrent software
- Explore current research topics that may impact the future of software model-checking

Acknowledgements

- See course web page for interesting short articles about the famous software failures mentioned in this lecture.
- See The Temporal Logic of Reactive and Concurrent Systems: Specification by Manna and Pnueli (Springer-Verlag) for an excellent and extensive discussion on safety and liveness properties (pp. 302-337).