Systems are known to be error-prone

- Capture complex aspects such as:
  - Threads and synchronization (e.g., Java locks)
  - Dynamically heap allocated structured data types (e.g., Java classes)
  - Dynamically stack allocated procedures (e.g., Java methods)
  - Non-determinism
  - Many input/output pairs
- Challenging to reason about all possible traces through the systems

Ways to get your code right

- **Verification & Validation**
  - Purpose is to uncover problems and increase confidence
  - Combination of manual and automated reasoning (e.g., model checkers, theorem provers) as well as testing
- **Debugging**
  - Purpose is finding out why a program is not functioning as intended
  - Pinpoint location + cause of problem
- **Defensive programming**
  - Programming with validation and debugging in mind

Overview of model checking

Automate reasoning about whether all traces through a system satisfy a given property specification

- If so, report “Is satisfied”
- If not, report “May be violated” and generate a counterexample trace that illustrates a potential violation of the property specification
Architecture of a model checker
[Java PathFinder or JPF]

Property Specification [FSA(s)]

Reasoning engine [Java virtual machine]

System Model [Java class file(s) as bytecode]

"Is satisfied"

[https://fl.arc.nasa.gov/tech/rse/vandv/jpf/]

"May be violated" + Counterexample trace [Java stack trace]

System translator

System [Java source file(s)]

Property [FSA(s)]

"Is satisfied"

[https://fl.arc.nasa.gov/tech/rse/vandv/jpf/]

"May be violated" + Counterexample trace [Java stack trace]
Roadmap

- Architecture of JPF model checker
  - System and translation
  - Property specification and translation
  - Reasoning engine
- Application of model checking
  - JPF demo
  - Evaluation
  - Potential benefits and disadvantages
- Upcoming assignments and guest lecture

Example: Bounded buffer (BB) system

```java
public class boundedBuffer {
  static int BUFFER_SIZE = 2;
  static int nProducers = 1;
  static int nConsumers = 1;
  static Object mutex = "Mutex";
  //---- the bounded buffer implementation
  protected final int buf[];
  protected int in = 0;
  protected int out = 0;
  protected int count = 0;
  protected final int SIZE;

  public boundedBuffer() {
    ...}
  public synchronized void put(Object obj) throws InterruptedException {}
  public synchronized void get() throws InterruptedException {}
  static class Producer extends Thread {
    // thread body calls put method
  }
  static class Consumer extends Thread {
    // thread body calls get method
  }
  public static void main(String[] args) {
    // Creates bounded buffer, producers, and consumers
    // Based on command line arguments
    // 1 producers and 1 consumers
    // Starts the producers and consumers
  }
}
```

Example: BB producer thread

```java
//---- the producer
static class Producer extends Thread {
  static int nProducers = 1;
  boundedBuffer buf;
  Producer(boundedBuffer b) {
    buf = b;
    setName("Producer\n" + nProducers);
  }
  public void run() {
    try {
      while (true) {
        // to ease state matching, we don't put different objects
        // in the buffer
        buf.put(buf, DATA);
    }
    } catch (InterruptedException e) {}}
  // From main method: new Producer(buf).start();
```
Example: BB producer thread

```java
//--- the producer
static class Producer extends Thread { 
    static int nProducers = 1;
    BoundedBuffer buf;

    Producer(BoundedBuffer b) { 
        buf = b;
        setName("p" + nProducers++);
    }

    public void run() {
        try {
            while(true) { 
                // to ease state matching, we don't put different objects
                // in the buffer
                buf.put(DATA);
            }
        } catch (InterruptedException e) {} 
    }
} 

From main method:
for (int i=0; i<10; i++) {
    new Producer(buf).start();
}
```
Example: BB constructor

```java
// --- the bounded buffer implementation
protected Object[] buf;
protected int in = 0;
protected int out = 0;
protected int count = 0;
protected int size;

public BoundedBuffer(int size) {
    this.size = size;
    buf = new Object[size];
}
```

Example: BB put method

```java
public synchronized void put(Object o) throws InterruptedException {
    while (count == size) {
        wait();
    }
    buf[in] = o;
    // System.out.println("PUT from " + Thread.currentThread().getId());
    ++count;
    in = (in + 1) % size;
    notify();
}
```

Example: BB put method

```
Each BoundedBuffer object is associated with a reentrant lock:
- Acquire that lock at the beginning of a synchronized method
- Release the lock at the end of that method
```
Example: BB put method

```java
public synchronized void put(Object o) throws InterruptedException {
    while (count == size) {
        wait();
    }
    buf[n] = o;
    //System.out.println("PUT from " + Thread.currentThread().getName()
    ++count;
    ia = (ia + 1) % size;
    notify();
}
```

Each BoundedBuffer object is also associated with a wait set:
- Add the current thread to that set for wait method
- Non-deterministically remove a thread from the set for notify method

Example: BB get method

```java
public synchronized Object get() throws InterruptedException {
    while (count == 0) {
        wait();
    }
    Object o = buf[out];
    buf[out] = null;
    //System.out.println("GET from " + Thread.currentThread().getName());
    --count;
    out = (out + 1) % size;
    notify();
    return (o);
}
```

System translation

Steps:
1) Translate Java source file(s) to Java class file(s) represented as bytecode
2) Optimize Java class file(s)

Architecture of a model checker
[Java PathFinder or JPF]

- Property Specification [FSA(s)]
- System [Java source file(s)]
- Property Specification translator
- System translator
- Reasoning engine [Java virtual machine]
- Property [FSA(s)]
- System model [Java class file(s) as bytecode]

"Is satisfied"
[https://tr.arc.nasa.gov/tech/rse/vandv/jpf/]

"May be violated" + Counterexample trace [Java stack trace]
Property specifications

- System requirements are represented as a set of property specifications

- Each property specification formally defines an intended (or unintended) behavior of the system [Represented conceptually as an FSA]

Example: No buffer under/over flow

```
public class NoBufferOverflowErrorProperty extends GenericProperty {
    // Code for handling under/over flow cases
    static void checkOverflow
    System.checkOverflow(buf)
    // Code for handling under/over flow cases

    public NoBufferOverflowErrorProperty(String error) {
        // Constructor code
    }

    public static void main(String[] args) {
        // Main method code
    }
```

In our case, BB buf field accesses (e.g., buf[0], buf[1]) don’t throw ArrayIndexOutOfBoundsException exceptions

Architecture of a model checker [Java PathFinder or JPF]

```
Property Specification (FSA(s))

Property Specification Translator

System Translator

System model
[Java class file(s)]

Reasoning engine
[Java virtual machine]

"Is satisfied"

"May be violated" + Counterexample trace
```

[http://arc.nasa.gov/tech/notes/vandv/jpf/]

Reasoning engine: Reachability graph (1)

- Generate the reachability graph for the given system model and property specification (PS)

- Each node captures a system model execution state and property state, e.g., initial node is:
  - P1: Not started, ... C4: Not started,
  - BB reentrant lock is free, BB wait set is empty,
  - BB buf is empty, BB count, in, and out are zero,
  - PS: Start state

Reasoning engine: Reachability graph (2)

- Each edge captures a current node, e.g., initial node, executing an “instruction”, e.g., P1 start, to generate the next node, e.g.,:
  - P1: Started, ... C4: Not started,
  - BB reentrant lock is free, BB wait set is empty,
  - BB buf is empty, BB count, in, and out are zero,
  - PS: Start state

Reasoning engine: Determining results

Report:
- “May be violated” if a node is encountered that illustrates a potential violation of the property (and generate the counterexample trace)
- “Is satisfied” if no such nodes are encountered

Counterexample trace

Represented as a sequence of reachability graph nodes where:
1. Start at the initial node
2. For each current node at index i, be able to generate its next node at index i + 1
3. End at a final node illustrating how to potentially reach the violation state of the property
Search-based counterexample trace generation

- Want to support:
  - Breadth first search: Generally slow but short counterexample traces that are different
  - (Bounded) depth first search: Generally fast but long counterexample traces that are similar

- Iteratively generate the reachability graph
  - Store a worklist of current nodes (e.g., BFS queue)
  - Store a visited set of nodes (e.g., BFS hash set of nodes)

Example: JPF inputs
(No buffer under/over flow)

**System:** BoundedBuffer(2,2,4)

**Property specification:**
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty

Example: JPF results
(No buffer under/over flow)

Example: JPF inputs (Data race)

**Common definition of a data race:**
- Two or more threads concurrently access a memory location
- One or more access is a write
- One or more access is unsynchronized

**System:** BoundedBuffer(2,2,4)

**Property specification:**
gov.nasa.jpf.listener.PreciseRaceDetector
Example: JPF results (Data race)

```
search started: 2020/09/29 2:14 PM
no errors detected
search: maxDepth=12, constraints hit=0
error generators: thread=79549 (signal=95179, lock=9790, shared ref=0, data=0), data=0
search: maxDepth=12, constraints hit=0
search finished: 2020/09/29 2:14 PM
```

Property specification: Deadlock

1. **Mutual exclusion**: At least one resource must be held in a non-shareable mode. Otherwise, the processes would not be prevented from using the resource when necessary. Only one process can use the resource at any given instant of time.[9]
2. **Hold and wait**: a process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. **No preemption**: a resource can be released only voluntarily by the process holding it.
4. **Circular wait**: each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, \( P_1, P_2, \ldots, P_n \), such that \( P_i \) is waiting for a resource held by \( P_{i+1} \), \( P_n \) is waiting for a resource held by \( P_1 \).


Example: JPF inputs (Deadlock 1)

- **System**: BoundedBuffer(2,2,4)
- **Property specification**: `gov.nasa.jpf.jvm.NotDeadlockedProperty`

Example: JPF results (Deadlock 1)

```
search started: 2020/09/29 2:14 PM
no errors detected
search: maxDepth=12, constraints hit=0
error generators: thread=79549 (signal=95179, lock=9790, shared ref=0, data=0), data=0
search: maxDepth=12, constraints hit=0
search finished: 2020/09/29 2:14 PM
```
Example: Counterexample trace (Deadlock 1)

Example: BB put and get methods (2)

```java
public synchronized void put(Object o) throws InterruptedException {
    while (count == size) {
        wait();
    }
    buf[in] = o;
    //System.out.println("PUT from " + Thread.currentThread().getName());
    count++;
}
```

```java
public synchronized Object get() throws InterruptedException {
    while (count == 0) {
        wait();
    }
    Object o = buf[out];
    buf[out] = null;
    //System.out.println("GET from " + Thread.currentThread().getName());
    count--;
    notf(out); // if this is not a notifyAll() we might notify the wrong wait
    return o;
}
```

Example: JPF results (Deadlock 2)

Evaluation of model checking

- Applied to benchmarks and actual systems
  - Have found actual bugs

- Compared in terms of:
  - performance: space and time
  - counterexample traces generated: usually by their length
Potential benefits of model checking

- Automatically checks that all traces through a given system model satisfies its property specifications
  - Can be re-checked after any changes
- Generates counterexample traces that can be used for debugging
- Generally requires less expertise than theorem provers

Disadvantages of model checking

- Writing property specifications can be error-prone
  - Use property patterns (or PROPEL)
- May not scale well because of the state space explosion problem
  - Use optimizations for system model translation and/or reasoning engine
- May not generate counterexample traces that are useful for debugging (e.g., too long, too similar to each other)
  - Use A* search with various heuristics

Mid-point presentations

- Each final project group needs to put together a presentation that briefly describes the following:
  1. Problem
  2. Design
  3. Evaluation
  4. Plan
- Each presentation will be given a time slot of 8 minutes: 7 minutes to talk and 1 minute for Q&A.
- The time slots will be randomly assigned during next week’s lectures on Tuesday 10/27 and Thursday 10/29
  - If your group cannot attend either lecture, you should contact me to schedule another day/time