CS 520
Theory and Practice of Software Engineering
Spring 2020
Model Checking
March 24, 2020

Online lecture guidelines

- Connect using this Zoom link:
  https://umass-amherst.zoom.us/j/976464357 (The link will remain active for the rest of the semester.)
- Will screen share the lecture slides and present them. Please mute your microphones.
- Will try to pause more often for questions asked by either using the:
  - raise hand feature
  - chat feature
- Will post the recorded lecture afterwards

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]

System [Java source file(s)]

Property Specification translator

System translator

“Is satisfied”

[https://ftp.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 final int BUFFER_SIZE = 2;
    static final int MAX_CONTENTS = 4;

    static BufferData = "buffer"
    // The bounded buffer implementation
    protected Buffer buffer;
    protected int in = 0;
    protected int out = 0;
    protected int count = 0;
    protected int size;
    public BoundedBuffer() {
        ...}
    public synchronized void putObject(Object obj) {
        ...}
    public synchronized void getBuffer(Object obj) {
        ...}
    static class Producer extends Thread {
        // Thread exits after put
    }
    static class Consumer extends Thread {
        // Thread exits after get
    }
    // static void main(String[] args) {
    // ...}
    public static void main(String[] args) {
        ...}
```
Example: BB producer thread

```java
// The producer class.
static class Producer extends Thread {
    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 < NUM_PRODUCERS; i++) {
            new Producer(buf).start();
        }
    }
}
```

Example: BB producer thread

```java
// The producer class.
static class Producer extends Thread {
    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 < NUM_PRODUCERS; i++) {
            new Producer(buf).start();
        }
    }
}
```
Example: BB consumer thread

```java
//--- the consumer
static class Consumer extends Thread {
    static int nConsumers = 1;
    BoundedBuffer buf;
    Consumer(BoundedBuffer b) {
        buf = b;
        setName("C" + nConsumers++);
    }
    public void run() {
        try {
            Object tmp = buf.get();
        } catch(InterruptedException e) {} // do nothing
        for (int i=0; i<10; i++) { } // do nothing
    } // run
}
From main method: new Consumer(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().getName() + ++count);
    in = (in + 1) % size;
    notify();
}
```
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
class BoundedBuffer {
    public synchronized void put(Object o) throws InterruptedException {
        while (count == size) {
            wait();
        }
        buf[in] = o;
        //System.out.println("PUT from " + Thread.currentThread().getName() + " count = " + count);
        in = (in + 1) % size;
        notify();
    }
}
```

Example: BB put method

```java
class BoundedBuffer {
    public synchronized void put(Object o) throws InterruptedException {
        while (count == size) {
            wait();
        }
        buf[in] = o;
        //System.out.println("PUT from " + Thread.currentThread().getName() + " count = " + count);
        in = (in + 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
class BoundedBuffer {
    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))
- Reasoning engine [Java virtual machine]
- System model [Java class file(s) as bytecode]
- System translator

"Is satisfied" [https://ti.arc.nasa.gov/tech/re/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

```java
public class NoBufferExceptionProperty extends GeneralProperty {
    // A check for buffer under/overflows
    static CounterspaceData unsatModel;
    public NoBufferExceptionProperty(CounterspaceData unsatModel) {
        throw new IllegalAccessException("counterexample in model!");
    }
    public String getErrorString() {
        return unsatModel.toString();
    }
    public String getErrorMessage() {
        return "counterexample in model!";
    }
    public boolean check(CounterspaceData c) {
        return unsatModel.toString().equals(c.toString());
    }
}
```

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

Example: No buffer under/over flow

1

Throw ArrayIndexOutOfBoundsException

viol

Throw ArrayIndexOutOfBoundsException
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)]
"May be violated" + Counterexample trace
Property [FSA(s)]
"Is satisfied"

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)

---

**Property specification:**
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.[R]
2. **Hold and wait or resource holding:** 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 is in turn waiting for the first process to release the resource. In general, there is a set of waiting processes, \( F = (P_1, P_2, ..., P_n) \), such that \( P_1 \) is waiting for a resource held by \( P_2 \), \( P_2 \) is


Example: JPF inputs (Deadlock 1)

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

[Example: JPF inputs (Deadlock)](https://en.wikipedia.org/wiki/Deadlock)
Example: JPF results (Deadlock 1)

```java
error #1: get.name: jpf run -m Deadlock1 Property "deadlock encountered
thread Deadlock1: 01:00 - 10:00"

----------------------------------- statistics
elapsed time: 00:00:00
rate: 0.0000, visits=19, basic=0.0000, evict=0, errors=169
search: maxDepth=170, constraints hit=4
choice generators: thread=18 (signal=95, lockw=6, shared ref=2), data=100
map: max=45, released=50, max live=392, gc-cycles=233
instructions: 877
max memory: 31MB
code size: class=89, methods=176
search finished: 3/23/20 7:36 PM
```

Example: Counterexample trace (Deadlock 1)

```java
Thread 0: Class B.println("A")
Thread 1: Class B.println("B")
Thread 2: Class B.println("C")
Thread 3: Class B.println("D")
Thread 4: Class B.println("E")
Thread 5: Class B.println("F")
Thread 6: Class B.println("G")
Thread 7: Class B.println("H")
Thread 8: Class B.println("I")
Thread 9: Class B.println("J")
Thread 10: Class B.println("K")
Thread 11: Class B.println("L")
Thread 12: Class B.println("M")
Thread 13: Class B.println("N")
Thread 14: Class B.println("O")
Thread 15: Class B.println("P")
Thread 16: Class B.println("Q")
Thread 17: Class B.println("R")
Thread 18: Class B.println("S")
Thread 19: Class B.println("T")
Thread 20: Class B.println("U")
Thread 21: Class B.println("V")
Thread 22: Class B.println("W")
Thread 23: Class B.println("X")
Thread 24: Class B.println("Y")
Thread 25: Class B.println("Z")
```

Example: JPF results (Deadlock 2)

```java
Example: BB put and get methods (2)

```java
public synchronized void put(Object o) throws InterruptedException {
    while (count == size) {
        wait();
    }
    buf[count] = o;
    //System.out.println("PUT from " + Thread.currentThread().getName());
    ++count;
    notifyAll(); // if this is not a notifyAll() we might notify the wrong waiters
}
```

```java
public synchronized Object get() throws InterruptedException {
    while (count == 0) {
        wait();
    }
    Object o = buf[0];
    buf[0] = null;
    //System.out.println("GET from " + Thread.currentThread().getName());
    --count;
    notifyAll(); // if this is not a notifyAll() we might notify the wrong waiters
    return o;
}
```
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, to similar to each other)
  - Use A* search with various heuristics

Upcoming assignment deadlines

- Homework 1 (v2) is due Wednesday March 25 9:00 AM EDT
- In-class exercise 3 is due Thursday March 26 9:00 AM EDT
- Final project mid-point report is due Thursday April 2 9:00 AM EDT
Next class: Guest lecture

Title: "High-Quality Automated Program Repair"
Speaker: Manish Motwani is a PhD candidate in CICS at the UMass Amherst, where he received the MS degree in 2018. His research involves studying large software repositories to learn interesting phenomena in software development and maintenance, and to use that knowledge to design novel automation techniques for testing and program repair.

http://people.cs.umass.edu/mmotwani

NOTE) We plan to post a link on the course website to a recording of this guest lecture.