UTA

Today's Agenda

- Quick Review
- Finish Monitor
- Replay Monitor

UTA

Quick Review

- What is the fundamental difference between SC and SU?

Tracing/Replay/Testing for Monitors

- Simple M-sequences
- M-sequences
- Correctness and Faults

Entry-based Execution

We assume that (1) all shared variables are accessed inside a monitor; and (2) the only source of non-determinism in a program is due to uncertainty with thread scheduling.

The executions of such a program is referred to as entry-based executions, as their behaviors are determined by the order in which the threads enter/ reentry the monitor.

UTA

Entry-based Execution (cont'd)

An entry-based execution can be replayed if the order in which threads (re)enter the monitor is repeated.

UTA

Simple M-sequence

A simple M-sequence can be used to replay a program's execution during debugging.

The synchronization events in a simple M-sequence depend on the type of monitors being used.

UTA

Simple M-sequence (cont'd)

A simple M-sequence for an SC monitor consists of the following two types of synchronization events: (1) entry into the monitor by a new thread; and (2) reentry into the monitor by a signaled thread.

A simple M-sequence for an SU monitor consists of the only type of synchronization event: entry into the monitor by a new thread.

An event in a simple M-sequence is denoted by the identifier (ID) of the thread that executed the event.

UTA

Example

Consider the bounded buffer monitor. Assume there is a single Producer thread (with ID 1) and a single Consumer thread (with ID 2).

With SC monitor, a possible simple M-sequence is (2,1,2,1,1,2,2).

The equivalent M-sequence with SU monitor is (2,1,1,1,2,2).

UTA

Tracing/Replay (SU)

In MonitorSU, mutex controls entry into the monitor.

During execution, enterMonitor is modified to send entry event to a control monitor, which records the event into a trace file.

To replay a simple M-sequence, a thread requests an entry permit from the control monitor to enter the monitor, and releases the permit to allow the next one to enter.

UTA

enterMonitor

```
public void enterMonitor () {
   if (replayMode)
      control.requestEntryPermit (ID);

   mutex.P ();

   if (replayMode)
      control.releasentryPermit ();
   else
      control.traceMonitorEntry (ID);
}
```

UTA

Control

```
class Control extends MonitorSC () {
    Control () {
    }
    public void requestEntryPermit (int ID) {
        enterMonitor ();
        if (ID != ((Integer) simpleMSequence.elementAt(index)).intValue()) {
            threads[ID].waitC ();
        }
        exitMonitor ();
    }
    public void releaseEntryPermit () {
        enterMonitor ();
        if (index < simpleMSequence.size() – 1) {
            ++ index;
            threads[((Integer) simpleMSequence.elementAt(index)).intValue()].signalC();
        }
        exitMonitor ();
    }
    public void traceMonitorEntry (int ID) { // record ID in trace file }
    public void traceMonitorReEntry (int ID) { … }
    private vector simpleMSequences;
    private ConditionVariable[] threads;
    private int index = 1;
}
```


UTA

Tracing/Replay (SC)

Tracing/Replay for SC monitor needs to record the reentry event, in addition to the entry event.

Besides enterMonitor, waitC method also needs to be modified.

UTA

waitC

```
public void waitC () {
   numWaitingThreads ++;
   threadQue.VP (mutex);

   if (replayMode)
      control.requestEntryPermit (ID);

   mutex.P ();

   if (replayMode)
      control.releaseEntryPermit ();
   else
      control.traceMonitorReEntry (ID);
}
```


UTA

Tracing/Replay/Testing for Monitors

- Simple M-sequences
- M-sequences
- Correctness and Faults

UTA

Regression Testing

When a failure is detected during an execution, we want to replay the execution, perhaps for many times, to locate the fault that has caused the failure.

After the fault is located, the program is then modified to correct the fault.

Regression testing must be performed to ensure that the fault has been corrected and no new faults were introduced.

UTA

Regression Testing (cont'd)

Regression testing requires that we determine whether or not a particular SYN-sequence is feasible. (This is different from the replay problem. Why?)

A SYN-sequence may represent an illegal behavior that was observed when the program failed. In this case, the sequence is expected to be infeasible.

If the SYN-sequence represents a legal behavior, then the sequence is expected to remain feasible during RT.

UTA

A simple M-sequence is insufficient…

Consider a program that contains an SU monitor for a two-slot buffer. Assume that there is a single producer (Thread 1) and a single consumer (Thread 2). A possible simple M-sequence is (1,1,1,2,2,2).

During replay, a thread always execute the same method as it did in the original execution. However, during RT, this is no longer true (why?).

UTA

Still insufficient …

Now we get a new M-sequence:

((1, deposit), (1, deposit), (1, deposit), (2, withdraw), (2, withdraw))

If the third item was actually deposited, then the first item was lost. However, if the third deposit operation was actually blocked in full.wait, then the program still behaves correctly.

M-sequence

A complete M-sequence consists of the following types of events:

- the entry of a monitor method and, for SC monitors, the reentry of a monitor
- the exit of a monitor method
- the start of execution of a wait operation
- the start of execution of a signal, or signalAndExit, or signalAll operation.

M-sequence (cont'd)

Each event in a M-sequence is encoded in the following format: (Vi, Ti, Mi, Ci), where

- Vi: the type of this event
- Ti: the ID of the thread executing this event
- Mi: the monitor method of this event
- Ci: the name of the condition variable if applicable

UTA

Example

```
( (enter,    consumer, withdraw, NA),
  (wait,     consumer, withdraw, notEmpty),
  (enter,    producer, deposit, NA),
  (signal,   producer, deposit, notEmtpy),
  (exit,     producer, deposit, NA),
  (reenter, consumer, withdraw, NA),
  (signal,   consumer, withdraw, notFull),
  (exit,     consumer, withdraw, NA) )
```


UTA

Feasibility Determination

Before a thread can perform a monitor operation, it requests permission from a control monitor.

The control monitor is responsible for reading a recorded M-sequence and forcing the execution to proceed according to the sequence.

If the M-sequence is found to be infeasible, the control module displays a message and terminates the program.

UTA

enterMonitor

```
public void enterMonitor (String methodName) {
   if (testMode) {
      control.requestMPermit (ENTRY, threadID, methodName, "NA");
   }
   mutex.P ();
   if (testMode) {
      control.releaseMPermit ();
   }
   else {
      control.trace (ENTRY, threadID, methodName, "NA");
   }
}
```


UTA

waitC

```
public void waitC () {
   if (testMode) {
      control.requestMPermit (WAIT, threadID,  methodName, conditionName);
   }
   else {
      control.trace (WAIT, threadID, methodName, conditionName);
   }
   numWaitingThreads ++;
   threadQue.VP (mutex);
   if (testMode) {
      control.requestMPermit (REENTRY, threadID, methodName, "NA");
   }
   mutex.P ();
   if (testMode) {
      control.releaseMPermit ();
   }
   else {
      control.trace (REENTRY, threadID, methodName, "NA");
   }
}
```

UTA

signalC & exitMonitor

```
public void signalC () {
   if (testMode) {
      control.requestMPermit (SIGNAL, threadID, methodName, conditionName);
   } else {
      control.trace (SIGNAL, threadID, methodName, conditionName);
   }
   if (numWaitingThreads > 0) {
      numWaitingThreads --;
      threadQue.V ();
   }
}

public void exitMonitor () {
   if (testMode) {
      control.requestMPermit (EXIT, threadID, methodName, "NA");
   }
   else {
      control.trace (EXIT, threadID, methodName, "NA");
   }
   mutex.V();
}
```


UTA

Control

```
public void requestEntryPermit (EventType et, int ID, String methodName,
          String conditionName) {
   monitorEvent nextEvent = (MonitorEvent) Msequence.elementAt(index);
   if (ID != nextEvent.getThreadID ()) {
      threads[ID].waitC ();
      nextEvent = (MonitorEvent) Msequence.elementAt (index);
   }
   if (!(et.equals(nextEvent.getEventType()))) { // issue diagnostic and terminate }
   if (!(methodName.equals(nextEvent.getMethodName()))) { // issue diagnostic and terminate}
   if (et.equals(WAIT) || et.equals(SIGNAL)) {
      if (!(conditionName.equals(nextEvent.getConditionName()))) { // issue diagnostic and terminate}
      ++ index;
      if (index < Msequence.size ())
         threads [((MonitorEvent) Msequence.elementAt(index)).getThreadID()].signalC();
   }
   else if (et.equals(EXIT)) {
      ++ index;
      if (index < Msequence.size ()) {
         threads[((MonitorEvent) Msequence.elementAt(index)).getThreadID()].signalC();
      }
   }
}
public void releaseEntryPermit () {
   if (index < simpleMSequence.size() – 1) {
      ++ index;
      threads[((MonitorEvent) MSequence.elementAt(index)).getThreadID()].signalC();
   }
}
```

releaseMPermit

This method is called when the current event is of type ENTRY or REENTRY. It will signal the thread waiting for the next event.

What happens to events of type WAIT, SIGNAL, or EXIT?

It is undecidable ...

An M-sequence is feasible if and only if the M-sequence is completely exercised.

However, the problem of determining whether a concurrent program terminates for a given input and SYN-sequence is in general undecidable!

That is, there does not exist an algorithm f such that $f: P \times I \times Q \rightarrow \{ \text{feasible, infeasible} \}$.

UTA

Timeout

In practice, we can specify a maximum time interval that is allowed between two consecutive events.

This "timeout" value represents the maximum amount of time that the controller is willing to wait for the next event to occur.

UTA

watchDog

```
final class watchDog extends Thread {
   public void run () {
      while (index < Msequence.size ()) {
         int saveIndex = index;
         try { Thread.sleep (2000); }
         catch (InterruptedException e) {}
         if (saveIndex == index) {
            // issue diagnostic and exit program
         }
      }
   }
}
```

Tracing/Replay/Testing for Monitors

- Simple M-sequences
- M-sequences
- Correctness and Faults

Feasible Sequences

Let P be a concurrent program. A SYN-sequence is said to be feasible for P with input X if this SYN-sequence can possibly be exercised by one execution of P with input X.

Feasible(P, X) = the set of feasible SYN-sequences of P with input X.

UTA

Valid Sequences

A SYN-sequence is said to be valid for P with input X if this SYN-sequence is allowed to be exercised during an execution of P with input X by the specification of P.

Valid(P, X) = the set of valid SYN-sequences of P with input X.

UTA

Correctness

P is said to be correct for input X (with respect to the specification of P) if

- Feasible(P, X) = Valid(P, X) and
- every possible execution of P with input X produces the correct (or expected) results.

UTA

Synchronization Fault

A program P has a synchronization fault if Feasible(P, X) ≠ Valid(P, X), which consists of two possible conditions:

- There exists at least one SYN-sequence that is feasible but invalid for P with input X.
- There exists at least one SYN-sequence that is valid but infeasible for P with input X.

UTA

Computational Fault

A program P has a computational fault if there exists an execution of P with input X that exercises a valid (and feasible) SYN-sequence, but computes an incorrect results.

UTA

Synchronization or Computation Fault?

Let S be a SYN-sequence exercised by an execution of P with input X. Let R be the results of the execution.

- S is valid and R is correct.
- S is valid and R is incorrect.
- S is invalid and R is incorrect.
- S is invalid and R is correct.

UTA

Example

```
monitor class BoundedBuffer {
   private int fullSlots = 0, capacity = 0;
   private char[] buffer = null;
   private int in = 0, out = 0;
   private ConditionVariable notFull = new ConditionVariable ();
   private ConditionVariable notEmpty = new ConditionVariable ();
   public BoundedBuffer (int capacity) {
      this.capacity = capacity;
      buffer = new int [capacity];
   }
   public void deposit (char value) {
      if (fullSlots > capacity) notFull.wait ();
      buffer[in] = value;
      in = (in + 1) % capacity;
      ++ fullSlots;
      notEmpty.signal ();
   }
   public char withdraw () {
      char value;
      if (fullSlots == 0) notEmpty.wait ();
      value = buffer[out];
      out = (out + 1) % capacity;
      -- fullSlots;
      notFull.signal ();
      return value;
   }
}
```

UTA

Example (cont'd)

```
(  (enter,  producer, deposit,  NA        ),
   (signal, producer, deposit,  notEmpty  ),
   (exit,   producer, deposit,  NA        ),
   (enter,  producer, deposit,  NA        ),
   (signal, producer, deposit,  notEmpty  ),
   (exit,   producer, deposit,  NA        ),
   (enter,  producer, deposit,  NA        ),
   (signal, producer, deposit,  notEmpty  ),
   (exit,   producer, deposit,  NA        ),
   (enter,  consumer, withdraw, NA        ),
   (signal, consumer, withdraw, notFull   ),
   (exit,   consumer, withdraw, NA        ),
   (enter,  consumer, withdraw, NA        ),
   (signal, consumer, withdraw, notFull   ),
   (exit,   consumer, withdraw, NA        ),
   (enter,  consumer, withdraw, NA        ),
   (signal, consumer, withdraw, notFull   ),
   (exit,   consumer, withdraw, NA        ) )
```


UTA

Example (cont'd)

Consider an execution in which the producer deposits items 'A', 'B', 'C'.

What is the output?

UTA

Example (cont'd)

Consider an execution in which the producer deposits items 'C', 'B', 'C'.

What is the output?