Analysis for Verification
Reachability analysis

Topics:
- Use of reachability analysis in modeling and verifying multi-threaded OOP programs
- For more information, see the book:
  J. Magee and J. Kramer
  Concurrency: State Models and Java Programs
  Wiley Publishers, 2000

Concurrent programs

Exhibit all of the potential errors of sequential programs plus:
- Non-deterministic results, race conditions
- Emergent properties: Deadlock/starvation

These additional failures difficult to reproduce using testing
Thus, another verification method is needed

Concurrent access to shared data

Problem: Multiple active objects might access the same passive object “at the same time”
  - Generally OK if the active objects are only reading data from the passive object(s)
  - But if one or more active objects is modifying the data members of the shared object, then we get anomalies

Example: Two active objects trying to pull an element off of a shared queue

To prevent these data-access anomalies requires synchronizing the active objects.

Example

class Processor {
  public:
    ...
    void activate()
      { while(1) {
          perform();
          if( !q.empty() ) {
            Event e = q.back();
            use(e);
            q.pop();
          }
        }
      }
  ...
  protected:
    queue<Event>* q;
};

int main(void)
  { queue<Event>* sharedQ;
    Processor p1(sharedQ);
    Processor p2(sharedQ);
    Thread t1(&p1);
    p2.activate();
  }

Example: Safe execution

Example: Unsafe execution
Thread synchronization

Definitions:
- Critical section: region of code in which at most one thread should be allowed to execute concurrently
- Mutex lock: OS facility used to synchronize threads
  - One and only one thread can own a lock
  - Thread comes to own a lock by acquiring it
  - A thread will block if it attempts to acquire a lock owned by another thread

Important: Whenever you write multi-threaded programs, you must identify and protect critical sections in your code

Problem: How do you know you’ve found all critical sections?

Reachability analysis can help

Not a panacea, but allows you to model specific circumstances and analyze all possible interleavings of thread accesses

Procedure:
- Model your solution in terms of FSP processes
- Identify unsafe conditions
- Use LTSA to check for reachability of these unsafe conditions

Example: Class Queue

class Queue

const QSIZE = 2
range QRANGE = 0..QSIZE
QUEUE = Q[QSIZE],
Q[i:QRANGE] = ( when (i==0) empty -> Q[0]
| when (i>0) nonempty -> Q[i]
| top -> Q[i]
| pop -> Q[i-1]
| push -> Q[i+1] ).

Example: Class Processor

class Processor

PROCESSOR = ( perform -> CHECKQ ),
CHECKQ =
( empty -> ( isEmpty -> PROCESSOR
| nonEmpty -> back -> use -> pop -> PROCESSOR ) ).

Class queue

QUEUE = Q[QSIZE],
Q[i:QRANGE] =
( empty -> ( when (i==0) isEmpty -> Q[0]
| when (i>0) nonEmpty -> Q[i] )
| back -> Q[i]
| pop -> Q[i-1] ).

Put it together...

||SYSTEM =
( p1:PROCESSOR
||
{p1,p2}:QUEUE
||
p2:PROCESSOR ).
Add lock to protect critical region

As the failure scenario demonstrates, we need to protect accesses to the shared queue.

**Solution:** Surround the code that requires atomic access to the queue with lock/unlock statements.

For example...

### Modified example

```c
int main(void)
{
    pthread_mutex_t lock;
    queue<Event> sharedQ;
    Processor p1(&sharedQ, &lock);
    Processor p2(&sharedQ, &lock);
    Thread t1(&p1);
    p2.run();
}
```

### Modified Processor specification

```
PROCESSOR = ( perform → CHECKQ ),
CHECKQ =
    ( lock →
        empty →
        ( isEmpty → unlock → PROCESSOR
        | nonEmpty → back →
        use →
        pop → unlock → PROCESSOR ) ).
```

### Mutex locks

Calls to `pthread_mutex_lock()` cause the caller to block if another thread has already acquired the lock.

We can model the mutual exclusion of locks using an FSP process:

```
LOCK = ( lock → unlock → LOCK ).
```

Then just need to introduce an instance of LOCK into the assembly

```c
||SYSTEM =
    ( p1:PROCESSOR
    ||
    {p1,p2}::QUEUE
    ||
    {p1,p2}::LOCK
    ||
    p2:PROCESSOR ).
```
What about efficiency?
Problem with this solution:
- One thread may lock out the other for a noticeable period of time
- Would it be safe to refactor the code so as to minimize time in the critical section?

Refactoring design of activate
```c
void activate()
{
  bool b;
  Event e;
  while(1) {
    perform();
    pthread_mutex_lock(l);
    if( !(b=q→empty()) ) {
      e = q→back();
      q→pop();
    } 
    pthread_mutex_unlock(l);
    if (b) use(e);
  }
}
```

Summary
Reachability analysis good for modeling tricky concurrent interactions in multi-threaded OO programs
- because it analyzes all possible execution paths
- invalid sequences lead to ERROR states
- FSP spec "structurally similar" to structure of program objects
Useful in verification and in modifying designs to fix bugs or increase concurrency