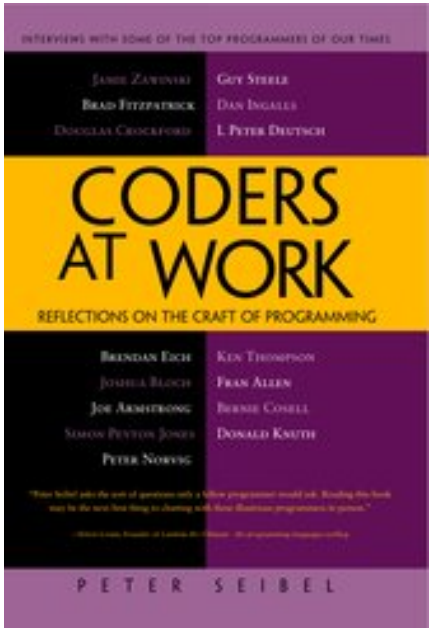
# Concurrency Analysis for Correct Concurrent Programs:

## *Fight Complexity Systematically and Efficiently*

**Moonzoo Kim**

**Computer Science, KAIST**

# Motivation for Concurrency Analysis




***Most of my subjects (interviewee) have found that the hardest bugs to track down are in concurrent code***

***…***

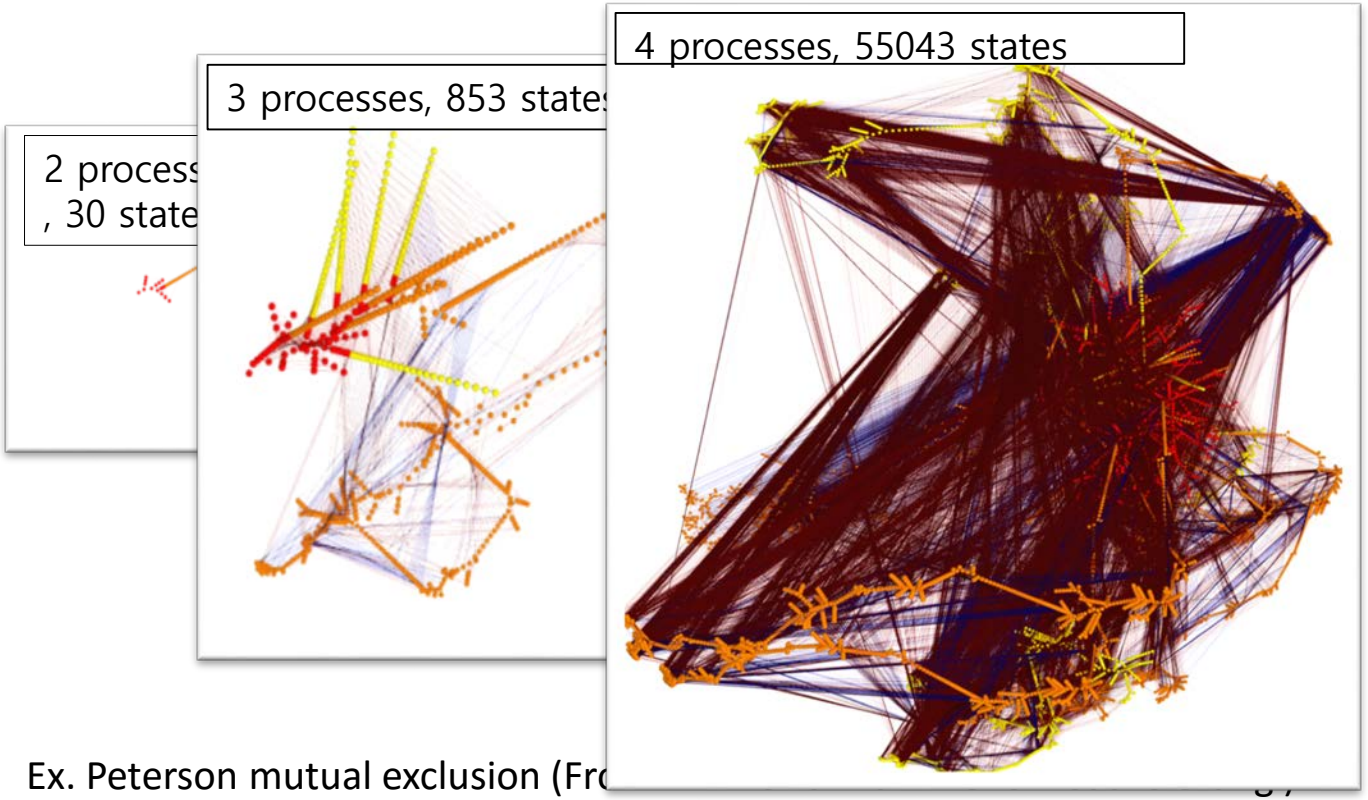
***And almost every one seem to think that ubiquitous multi-core CPUs are going to force some serious changes in the way software is written***

P. Siebel, *Coders at work* (2009) -- interview with 15 top programmers of our times:
Jamie Zawinski, Brad Fitzpatrick, Douglas Crockford, Brendan Eich, Joshua Bloch, Joe Armstrong, Simon Peyton Jones, Peter Norvig, Guy Steele, Dan Ingalls, L Peter Deutsch, Ken Thompson, Fran Allen, Bernie Cosell, Donald Knuth

Unintended/unexpected thread scheduling (a.k.a., interleaving scenarios) raises hard to detect concurrency errors

# Concurrent Programming is Error-prone

- Correctness of concurrent programs is hard to achieve
  - Interactions between threads should be carefully performed
  - A large # of thread executions due to non-deterministic thread scheduling
  - Testing technique for sequential programs do not properly work




Ex. Peterson mutual exclusion (Fr

# Concurrency

- Concurrent programs have very high complexity due to non-deterministic scheduling
- Ex. int x=0, y=0, z =0;

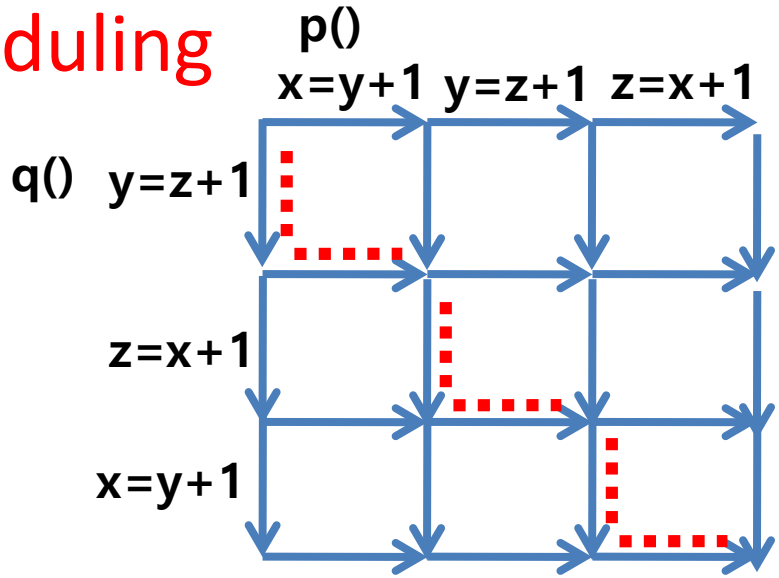
  void p() {x=y+1; y=z+1; z= x+1;}

  void q() {y=z+1; z=x+1; x=y+1;}

  – Total 20 interleaving scenarios

  = (3+3)!/(3!x3!)

  – However, only 11 unique outcomes




| | |
|---|---|
| Trail1: 2,2,3 | Trail7: 2,1,3 |
| Trail2: 3,2,4 | Trail8: 2,3,3 |
| Trail3: 3,2,3 | Trail9: 4,3,5 |
| Trail4: 2,4,3 | Trail10: 4,3,2 |
| Trail5: 5,4,6 | Trail11: 2,1,2 |
| Trail6: 5,4,3 | |



KAIST

## Read-Write Locks

KAIST

```
typedef struct {
    int readers;
    int writer;
    pthread_cond_t readers_proceed;
    pthread_cond_t writer_proceed;
    int pending_writers;
    pthread_mutex_t read_write_lock;
} mylib_rwlock_t;

void mylib_rwlock_init (mylib_rwlock_t *l) {
    l -> readers = l -> writer = l -> pending_writers = 0;
    pthread_mutex_init(&(l -> read_write_lock), NULL);
    pthread_cond_init(&(l -> readers_proceed), NULL);
    pthread_cond_init(&(l -> w
}
```

## Read-Write Locks

KAIST

```
void mylib_rwlock_rlock(mylib_rwlock_t *l) {
    /* if there is a write lock or pending writers, perform
       condition wait.. else increment count of readers and
       grant read lock */
    pthread_mutex_lock(&(l -> read_write_lock));
    while ((l -> pending_writers > 0) || (l -> writer > 0))
        pthread_cond_wait(&(l -> readers_proceed),
                          &(l -> read_write_lock));
    l -> readers ++;
    pthread_mutex_unlock(&(l -> read_write_lock));
}
```

Very difficult to find concurrency bugs !!!

## Read-Wri

```
void mylib_rwlock_wlock(mylib_rwlock_t *l) {
    /* if there are readers or writers, increment pending
       writers count and wait. On being woken, decrement
       pending writers count and increment writer count */

    pthread_mutex_lock(&(l -> read_write_lock));
    while ((l -> writer > 0) || (l -> readers > 0)) {
        l -> pending_writers ++;
        pthread_cond_wait(&(l -> writer_proceed),
                          &(l -> read_write_lock));
    }
    l -> pending_writers --;
    l -> writer ++;
    pthread_mutex_unlock(&(l -> read_write_lock));
}
```

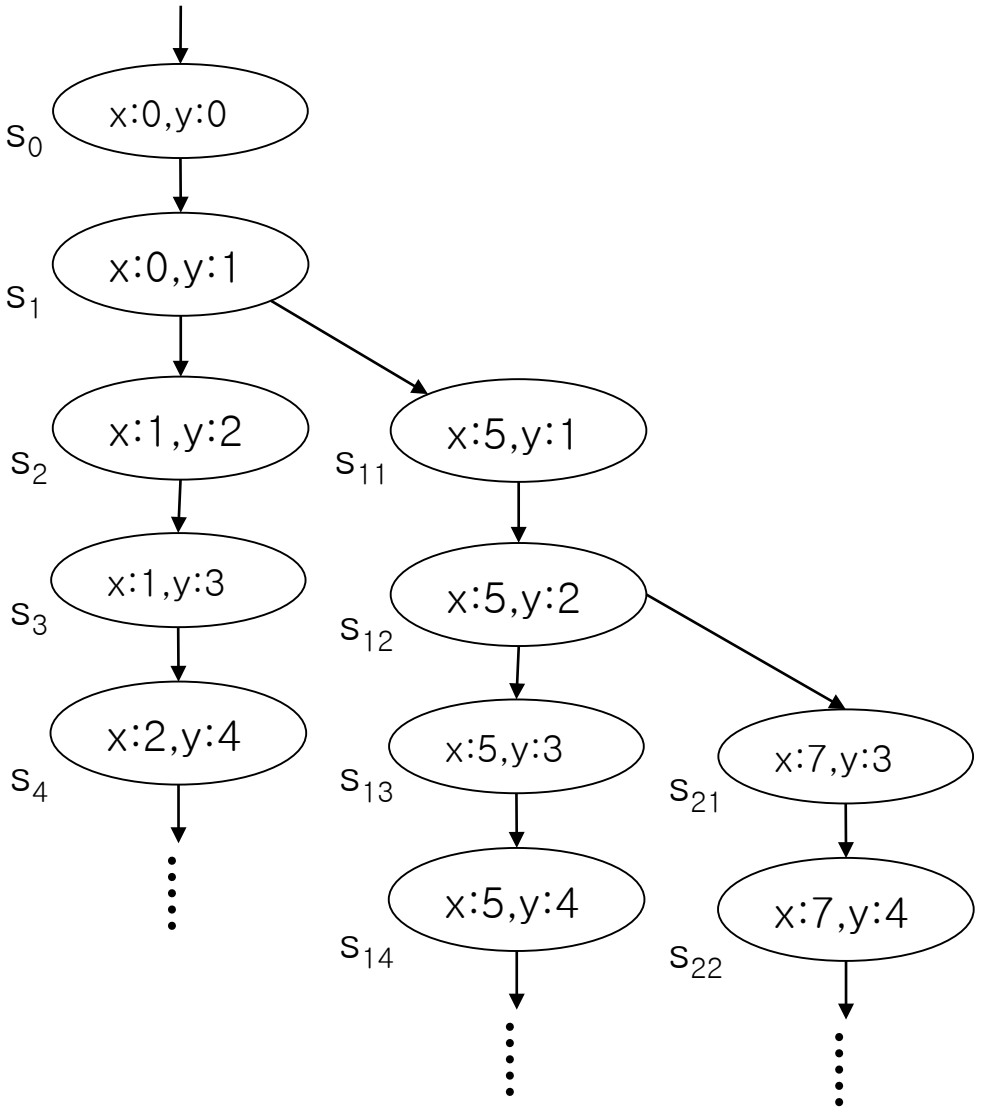
## e Locks

KAIST

```
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
/* if there is a write lock then unlock, else if there are
   read locks, decrement count of read locks. If the count
   is 0 and there is a pending writer, let it through, else
   if there are pending readers, let them all go through */
   pthread_mutex_lock(&(l -> read_write_lock));
   if (l -> writer > 0)
      l -> writer = 0;
   else if (l -> readers > 0)
      l -> readers --;
   pthread_mutex_unlock(&(l -> read_write_lock));
   if ((l -> readers == 0) && (l -> pending_writers > 0))
      pthread_cond_signal(&(l -> writer_proceed));
   else if (l -> readers > 0)
      pthread_cond_broadcast(&(l -> readers_proceed));
}
```

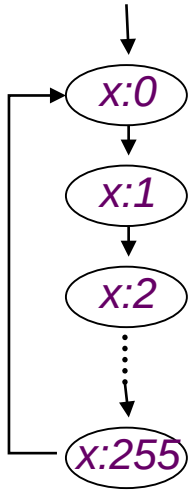
# Operational Semantics of Software

- A system execution $\sigma$ is a sequence of states $s_0 s_1 \ldots$
  - A state has an environment $\rho_s$: *Var-> Val*
- A system has its semantics as a set of system executions



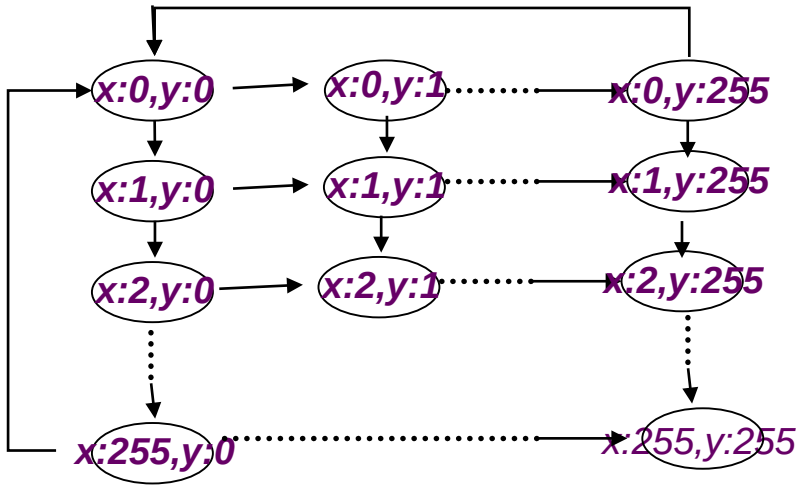
# Model Checker Analyzes All Possible Scheduling

```
active type A() {
byte x;
again:
     x++;
    goto again;
}
```

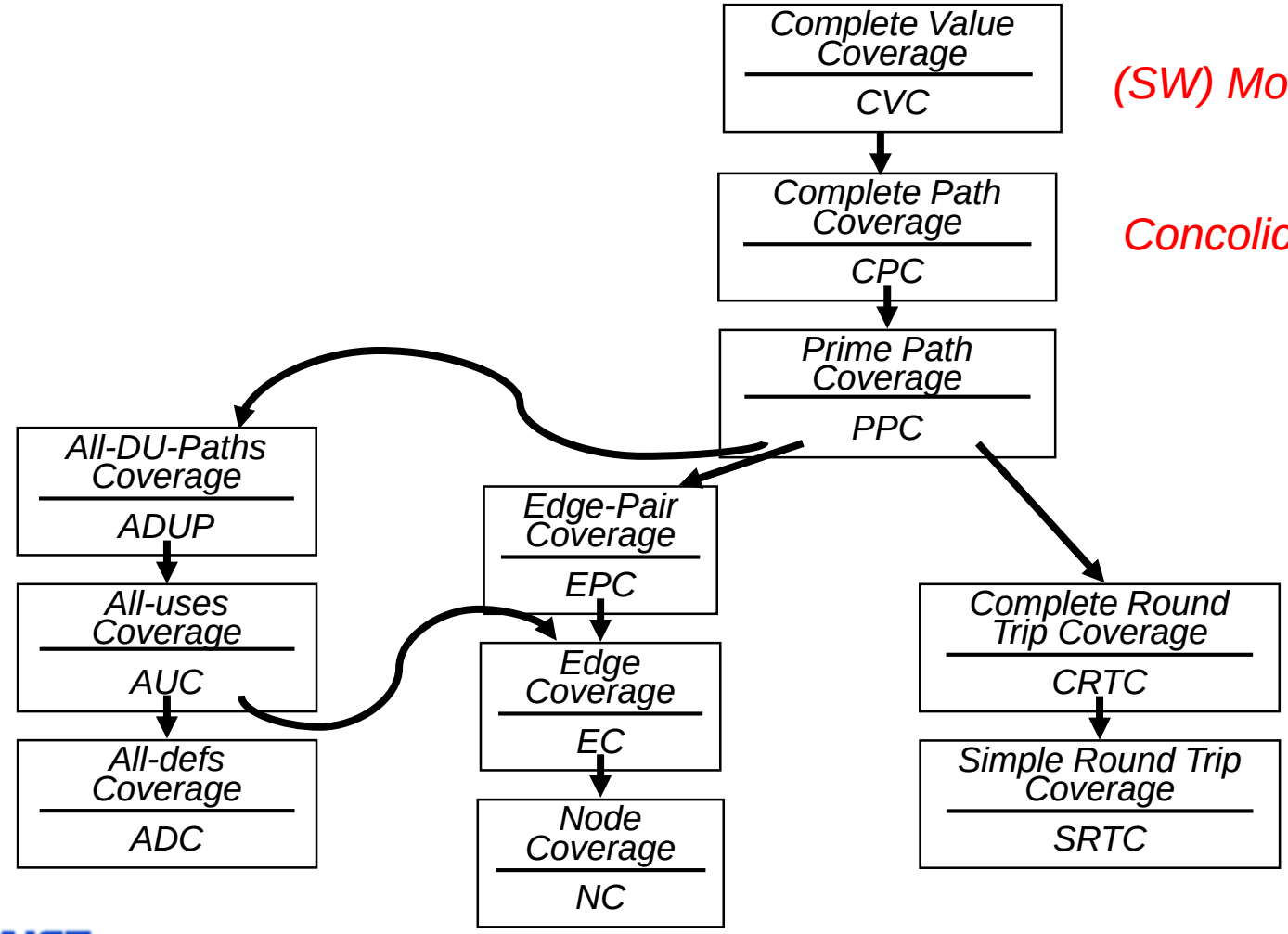



```
active type A() {
byte x;
again:
     x++;
    goto again;
}

active type B() {
byte y;
again:
     y++;
    goto again;
}
```



# Hierarchy of SW Coverage Criteria




*(SW) Model checking*

*Concolic testing*