1. Introduction to Concurrent Programming

A concurrent program contains two or more threads that execute concurrently and work together to perform some task.

Java provides a Thread class for defining user threads.

```java
class simpleThread extends Thread {
    public simpleThread(int ID) { myID = ID; }
    public void run() { System.out.println("Thread "+ myID + " is running."); }
    private int myID;
}

public class javaConcurrentProgram {
    public static void main(String[] args) {
        simpleThread thread1 = new simpleThread(1);
        simpleThread thread2 = new simpleThread(2);
        thread1.start(); thread2.start(); // causes the run() methods to execute
    }
}
```

Threads in the same Java program can communicate by calling methods on a shared object.
Something similar can be done in C++/Pthreads/Win32.

```cpp
int s=0; // shared variable s

class communicatingThread: public Thread {
public:
    communicatingThread (int ID) : myID(ID) {} // constructor
    virtual void* run();
private:
    int myID;
};

void* communicatingThread::run() {
    std::cout << "Thread " << myID << " is running" << std::endl;
    for (int i=0; i<10000000; i++) // increment s ten million times
        s = s + 1;
    return 0;
}

int main() {
    std::auto_ptr<communicatingThread> thread1(new communicatingThread (1));
    std::auto_ptr<communicatingThread> thread2(new communicatingThread (2));
    thread1->start(); thread2->start();
    thread1->join(); thread2->join();
    std::cout << "s: " << s << std::endl; // expected final value of s is 20000000
    return 0;
}
```

This C++ program was executed fifty times.
- in forty-nine of the executions, the value 20000000 was displayed,
- the value displayed for one of the executions was 19215861.

Concurrent programs exhibit non-deterministic behavior - two executions of the same program with the same input can produce different results.
Example 1. Assume that \( y \) and \( z \) are initially 0.

\[
\begin{array}{ll}
\text{Thread1} & \text{Thread2} \\
\hline
x = y + z; & y = 1; \\
& z = 2;
\end{array}
\]

Assume (incorrectly) that each statement is atomic, what are the expected final values?

The machine instructions for \textit{Thread1} and \textit{Thread2}:

\[
\begin{array}{ll}
\text{Thread1} & \text{Thread2} \\
(1) \; \text{load r1, y} & (4) \; \text{assign y, 1} \\
(2) \; \text{add r1, z} & (5) \; \text{assign z, 2} \\
(3) \; \text{store r1, x} & \\
\end{array}
\]

Possible interleavings:

- (1), (2), (3), (4), (5) \( \Rightarrow \) \( x \) is 0
- (4), (1), (2), (3), (5) \( \Rightarrow \) \( x \) is 1
- (1), (4), (5), (2), (3) \( \Rightarrow \) \( x \) is 2 *
- (4), (5), (1), (2), (3) \( \Rightarrow \) \( x \) is 3

If there are \( n \) threads (\textit{Thread}_1, \textit{Thread}_2, \ldots, \textit{Thread}_n) such that \textit{Thread}_i executes \( m_i \) atomic actions, then the number of possible interleavings of the atomic actions is:

\[
(m_1 + m_2 + \ldots + m_n)! \\
\frac{(m_1! \times m_2! \times \ldots \times m_n!)}{m_1! \times m_2! \times \ldots \times m_n!}
\]
Example 2. Variable \textit{first} points to the first \textit{Node} in the list. Assume the list is \texttt{!empty}.

```java
class Node {
    public valueType value;
    public Node next;
}

Node first; // \textit{first} points to the first \textit{Node} in the list;

void deposit(valueType value) {
    Node p = new Node(); // (1)
    p.value = value; // (2)
    p.next = first; // (3)
    first = p; // (4) insert the new \textit{Node} at the front of the list
}

valueType withdraw() {
    valueType value = first.value; // (5) withdraw the first value in the list
    first = first.next; // (6) remove the first \textit{Node} from the list
    return value; // (7) return the withdrawn value
}
```

The following interleaving of statements is possible:

- `valueType value = first.value; // (5) in withdraw`
- `Node p = new Node(); // (1) in deposit`
- `p.value = value // (2) in deposit`
- `p.next = first; // (3) in deposit`
- `first = p; // (4) in deposit`
- `first = first.next; // (6) in withdraw`
- `return value; // (7) in withdraw`

At the end of this sequence:

- the withdrawn item is still pointed to by \textit{first}
- the deposited item has been lost.

To fix this problem, each of methods \textit{deposit} and \textit{withdraw} must be implemented as an atomic action.

In Java: make each method “synchronized”: public \texttt{synchronized} void deposit(...) {}
**Introduction to Testing and Debugging Multithreaded Programs**

The purpose of testing is to find program failures.

A *failure* is an observed departure of the external result of software operation from software requirements or user expectations [IEEE90]. Failures can be caused by hardware or software faults or by user errors.

A software *fault* (or defect, or bug) is a defective, missing, or extra instruction or a set of related instructions, that is the cause of one or more actual or potential failures [IEEE 88].

*Debugging* is the process of locating and correcting faults.

The conventional approach to testing and debugging a sequential program \{A&O Ch.6\}

1. Select a set of test inputs

2. Execute the program once with each input and compare the test results with the intended results.

3. If a test input finds a failure, execute the program *again* with the same input in order to collect debugging information and find the fault that caused the failure.

4. After the fault has been located and corrected, execute the program *again* with each of the test inputs to verify that the fault has been corrected and that, in doing so, no new faults have been introduced (a.k.a “regression testing”).

This process breaks down when it is applied to concurrent programs.
Problems and Issues for concurrent programs

Let CP be a concurrent program. Multiple executions of CP with the same input may produce different results. This non-deterministic execution behavior creates the following problems during the testing and debugging cycle of CP:

Problem 1. When testing CP with input X, a single execution is insufficient to determine the correctness of CP with X. Even if CP with input X has been executed successfully many times, it is possible that a future execution of CP with X will produce an incorrect result.

Problem 2. When debugging a failed execution of CP with input X, there is no guarantee that this execution will be repeated by executing CP with X.

Problem 3. After CP has been modified to correct a fault detected during a failed execution of CP with input X, one or more successful executions of CP with X during regression testing do not imply that the detected fault has been corrected or that no new faults have been introduced.

There are many issues that must be dealt with in order to solve these problems:

Program Replay: Programmers rely on debugging techniques that assume program failures can be reproduced. Repeating an execution of a concurrent program is called “program replay”.

Program Tracing: Before an execution can be replayed it must be traced. But what exactly does it mean to replay an execution?

- If a sequential C++ program is executed twice and the inputs and outputs are the same for both executions, are these executions identical? Are the context switches always on the same places? Does this matter?
- Now consider a concurrent program. Are the context switches among the threads in a program important? Must we somehow trace the points at which the context switches occur and then repeat these switch points during replay?

Sequence Feasibility: A sequence of actions that is allowed by a program is said to be a feasible sequence. Testing involves determining whether or not a given sequence is feasible or infeasible. “Good” sequences are expected to be feasible while “bad” sequences are expected to be infeasible. How are sequences selected?

Sequence Validity: Sequences allowed by the specification, i.e. “good” sequences, are called valid sequences; other sequences are called invalid sequences. A goal of testing is to find valid sequences that are infeasible and invalid sequences that are feasible.

The Probe Effect: Modifying a concurrent program to capture a trace of its execution may interfere with the normal execution of the program.

Three different problems:
- Observability problem: the difficulty of accurately tracing a given execution,
- Probe effect: the ability to perform a given execution at all.
- Replay problem: repeating an execution that has already been observed.

Real-Time: The probe effect is a major issue for real-time concurrent programs. The correctness of a real-time program depends not only on its logical behavior, but also on the time at which its results are produced.
For an example of program tracing and replay, we’ve modified the C++ program shown earlier so that it can trace and replay its own executions.

sharedVariable<int> s(0); // shared variable s

class communicatingThread: public TDThread {
public:
    communicatingThread(int ID) : myID(ID) {}
    virtual void* run();
private:
    int myID;
};

void* communicatingThread::run() {
    std::cout << "Thread " << myID << " is running" << std::endl;
    for (int i=0; i<2; i++) // increment s two times (not 10 million times)
        s = s + 1;
    return 0;
}

int main() {
    std::auto_ptr<communicatingThread> thread1(new communicatingThread (1));
    std::auto_ptr<communicatingThread> thread2(new communicatingThread (2));
    thread1->start(); thread2->start();
    thread1->join(); thread2->join();
    std::cout << "s: " << s << std::endl; // the expected final value of s is 4
    return 0;
}

A possible trace:

Read(thread1,s) // thread1 reads s; s is 0
Read(thread2,s) // thread2 reads s; s is 0
Write(thread2,s) // thread2 writes s; s is now 1
Write(thread1,s) // thread1 writes s; s is now 1
Read(thread1,s) // thread1 reads s; s is 1
Write(thread1,s) // thread1 writes s; s is now 2
Read(thread2,s) // thread2 reads s; s is 2
Write(thread2,s) // thread2 writes s; s is now 3
Consider the following programming tasks:

1. Implement and test a Java Barrier class:

   Worker Thread i {
     Barrier b;
     while (true) {
       perform thread i’s task;
       wait for all n threads to complete; // b.waitB();
     }
   }

   class Barrier {
     private int count = 0; // count of waiting threads
     private int n = 0; // number of threads
     public Barrier (int n) {this.n = n;}
     public synchronized void waitB() {
       // threads call b.waitB() to wait on Barrier b
     }
   }

2. Implement and test a distributed mutual exclusion algorithm.

   When a process wishes to access a shared resource (e.g., printer), it requests and waits for permission from all the other processes. When a process receives a request:

   - if the process is not interested in entering its critical section, the process gives its permission by sending a reply as soon as it receives the request.
   - if the process does want to access the resource, then it may defer its reply, depending on the relative order of its request among the requests made by other processes.
First, let’s more carefully specify what our Barrier should do (ignoring how to make this happen):

```java
public void waitB(int ID) {
    enterMonitor("manEnter");
    exerciseEvent("Thread "+ID+" beginWaitB");
    /*
     * count++;
     * if (count != n)
     *     wait for other threads;       // pseudocode
     * else {
     *     count = 0;
     *     wakeup waiting threads;
     * }
    */
    exerciseEvent("Thread "+ID+" endWaitB");
    exitMonitor();
}
```

Scenario 1: a valid sequence: threads wait in the order: 1 2 3 1 2 3.

(\begin{itemize}
  \item Thread 1 beginWaitB
  \item Thread 2 beginWaitB
  \item Thread 3 beginWaitB
  \item Thread 3 endWaitB
  \item Thread 1 endWaitB
  \item Thread 2 endWaitB
  \item Thread 1 beginWaitB
  \item Thread 2 beginWaitB
  \item Thread 3 beginWaitB
  \item Thread 3 endWaitB
  \item Thread 1 endWaitB
  \item Thread 2 endWaitB
\end{itemize})

Scenario 2: an invalid sequence.
Purpose: threads 1 and 2 call waitB, but thread 2 exits before thread 3 calls waitB

(\begin{itemize}
  \item Thread 1 beginWaitB
  \item Thread 2 beginWaitB
  \item Thread 2 endWaitB
\end{itemize})

Later, we can use these scenarios as (specification-based) test sequences \{A&O sect. 5.4\}. Alternately, we can write and verify an FSM specification, and use it to generate test sequences.
The preceding sequences were specification-level sequences – they described what the program is expected to do.

Implementation level sequences are called Synchronization-sequences, or SYN-sequences – they describe how threads are synchronized to achieve a specified behavior.

A synchronization event, or “SYN-event”, refers to the execution of a synchronization operation (e.g., P, V, send, receive) on a synchronization object (semaphore, channel).

Example:

```plaintext
mailbox C1, C2; // synchronous mailboxes

Thread1          Thread2          Thread3          Thread4
C1.send(msg1);   msg = C1.receive(); msg = C1.receive(); C1.send(msg1);
C2.send(msg2);   msg = C2.receive(); msg = C2.receive(); C2.send(msg2);
```

One possible complete SR-sequence of this program is:

- (Thread1, Thread2, C1, SendReceive-synchronization),
- (Thread4, Thread3, C1, SendReceive-synchronization),
- (Thread1, Thread2, C2, SendReceive-synchronization),
- (Thread4, Thread3, C2, SendReceive-synchronization).

Example: For a Barrier: (1, enterMonitor), (1, wait), (2, enterMonitor), (2, wait), (3, enterMonitor), (3, notifyAll), (3, exitMonitor), (1, exitMonitor), (2, exitMonitor).

The result of an execution is determined by the program text, the input data, and the SYN-sequence exercised by the execution.

SYN-sequences can be traced and replayed. Their validity and feasibility can be checked. All the SYN-sequences of a program can be enumerated systematically and automatically (if you have the time 😃).
Paths of Concurrent Programs {A&O Ch. 2}

What is the relationship between the paths and SYN-sequences of a concurrent program?

Defining a Path

An execution of a sequential program exercises a sequence of statements, referred to as a path of the program.

The result of an execution of a sequential program is determined by the input and the sequence of statements executed during the execution. However, this is not true for a concurrent program.

port M; // synchronous port

<table>
<thead>
<tr>
<th>Thread1</th>
<th>Thread2</th>
<th>Thread3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) M.send(A);</td>
<td>(2) M.send(B);</td>
<td>(3) X = M.receive();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(4) Y = M.receive();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(5) output the difference (X – Y) of X and Y</td>
</tr>
</tbody>
</table>

Assume that an execution of CP with input A=1 and B=2 exercises the totally-ordered sequence of statements (1), (2), (3), (4), (5).

This is not information to determine the output of the execution.
A totally-ordered path of a concurrent program is a totally-ordered sequence of statements plus additional information about any synchronization events that are generated by these statements.

For example, a totally-ordered path of program CP in Listing 7.2 is

$$((1), (2), (3, \text{Thread1}), (4, \text{Thread2}), (5)).$$

Events (3, Thread1) and (4, Thread2) denote that the receive statements in (3) and (4) receive messages from Thread1 and Thread2, respectively.

Information about the synchronization events of a path can also be specified separately in the form of a SYN-sequence. Thus, a totally-ordered path of CP is associated with a SYN-sequence of CP, referred to as the SYN-sequence of this path.

Assume that CP contains threads T_1, T_2, ..., and T_n. A partially-ordered path of CP is (P_1, P_2, ..., P_n), where P_i, 1 ≤ i ≤ n, is a totally-ordered path of thread T_i. A partially-ordered path of CP is associated with the partially-ordered SYN-sequence of this path.

- A path (SYN-sequence) of CP is said to be feasible for CP with input X if this path (SYN-sequence) can be exercised by some execution of CP with input X.

- A path (SYN-sequence) of CP is said to be feasible for CP if this path (SYN-sequence) can be exercised by some execution of CP.

- The domain of a path or SYN-sequence S of CP is a set of input values. Input X is in the domain of a path or SYN-sequence S if S is feasible for CP with input X. The domain of an infeasible path or SYN-sequence is empty.
The following relationships exist between the paths and SYN-sequences of CP:

(a) If a path is feasible for CP with input X, the SYN-sequence of this path is feasible for CP with input X.

(b) …

(c) If a totally-ordered SYN-sequence S is feasible for CP with input X, there exists at least one totally-ordered, feasible path of CP with input X such that the totally-ordered SYN-sequence of this path is S.

(d) …

(e) The domains of two or more different partially-ordered, feasible paths of CP are not necessarily mutually disjoint. This statement is also true for two or more totally-ordered, feasible paths of CP. The reason is that CP with a given input may have two or more different partially- or totally-ordered, feasible SYN-sequences.

(f) …
We will illustrate relationship (e) with an example. Consider the following program:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) p1.send();</td>
<td>(1) p2.send();</td>
<td>(1) input(x);</td>
</tr>
<tr>
<td>(2) if (x)</td>
<td>(3) output(x);</td>
<td></td>
</tr>
<tr>
<td>(4) select</td>
<td></td>
<td>(5) p1.receive();</td>
</tr>
<tr>
<td>(5) p1.receive();</td>
<td>(6) p2.receive();</td>
<td></td>
</tr>
<tr>
<td>(7) or</td>
<td>(8) p2.receive();</td>
<td></td>
</tr>
<tr>
<td>(9) p1.receive();</td>
<td>(10) end select;</td>
<td></td>
</tr>
</tbody>
</table>

One partially-ordered path of this program is

- Thread1: (1)
- Thread2: (1)
- Thread3: (1), (2), (3), (5,Thread1), (6,Thread2)

and another path is

- Thread1: (1)
- Thread2: (1)
- Thread3: (1), (2), (3), (8,Thread2), (9,Thread1)

These paths are different, but the value of input $x$ is *true* in both paths so their input domains are not disjoint.

In sequential programs, paths that are different have disjoint domains.
Path-based Testing and Coverage Criteria

Coverage criteria are used to determine when testing can stop and to guide the generation of input values for test cases.

Structural coverage criteria focus on the paths in a program.

The all-paths criterion requires every path to be executed at least once. Since the number of paths in a program may be very large or even infinite, it may be impractical to cover them all.

How many paths:
- *Barrier* with 3 threads: 144. Q: How many threads should be used for testing? Compare to Object-Oriented Testing {A&O Section 7.1}

The minimum structural coverage criterion is statement coverage, which requires every statement in a program to be executed at least once.

Some stronger criteria focus on the predicates in a program.

- *decision coverage* requires every (simple or compound) predicate to evaluate to true at least once and to false at least once. Decision coverage is also known as branch coverage.

- *condition coverage* requires each condition in each predicate to evaluate to true at least once and to false at least once. Note that decision coverage can be satisfied without testing both outcomes of each condition in the predicate.
- **multiple-condition coverage** requires all possible combinations of condition outcomes in each predicate to occur at least once. Note that for a predicate with \( N \) conditions, there are \( 2^N \) possible combinations of outcomes for the conditions.

These criteria can be compared based on the *subsumes* relation.

A coverage criterion \( C_1 \) is said to *subsume* another criterion \( C_2 \) if and only if any set of paths that satisfies criterion \( C_1 \) also satisfies criterion \( C_2 \).

```
  multiple condition coverage
    ↓
  decision/condition coverage
    ↓   ↓
  decision coverage  condition coverage
    ↓   ↓
  statement coverage
```

Instead of focusing on the control characteristics of a program, other structural coverage criteria focus on the patterns in which data is defined and used, e.g., *all-du-paths*.

Under certain assumptions, all-du-paths subsumes decision coverage.

(We’ll keep these in mind when we test our Barrier class, but consider instead mutation testing.)
Structural coverage criteria are often defined with respect to a flowgraph model of a program.

The following is an example flowgraph for a thread that contains an if-else statement and a do-while loop.

![Flowgraph Diagram]

Figure 7.4 A thread and its control-flow graph.

Note that some paths through a flowgraph may represent program paths that are not executable. The predicates in the if-else and loop statements must be examined to determine which paths are executable.

In a flowgraph model, statement coverage is achieved by covering *all-nodes*. Note that when a node is executed, each statement in the block represented by that node is guaranteed to be executed.

Decision coverage is achieved by covering *all-edges* of the flowgraph.
port M; // synchronous port

<table>
<thead>
<tr>
<th>Thread1</th>
<th>Thread2</th>
<th>Thread3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) M.send(A);</td>
<td>(2) M.send(B);</td>
<td>(3) X = M.receive();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(4) Y = M.receive();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(5) output the difference (X – Y) of X and Y</td>
</tr>
</tbody>
</table>

In the program above none of the threads in CP contain any branches. Thus, any single execution of the program will cover all the statements in CP and all the paths in each of the threads. (Each thread has one path.)

However, based on our earlier definition of a path, there are two partially-ordered paths of CP:
- one path in which T3 receives T1’s message first
- one path in which T3 receives T2’s message first.

Path-based coverage criteria for concurrent programs should consider the statements exercised within threads and also the synchronization between threads, since both are used to define the paths of a concurrent program.

The paths of a concurrent program can be presented by a graph structure called a reachability graph.

The flowgraphs of individual threads can be used to build a reachability graph of a program.
The flowgraph constructed for a thread contains only the nodes and edges necessary to capture the thread’s synchronization activity, e.g., sending and receiving messages, selecting alternatives in a selective wait, and ignore thread activities unrelated to synchronization.

Flowgraphs of the individual threads are analyzed to derive a “concurrency graph” of the program:

- A concurrency graph contains nodes that represent the concurrency states of the program and edges representing transitions between the states.
- A concurrency state specifies the next synchronization activity to occur in each of the program’s threads.

The Fig. below shows the flowgraphs for the threads in the previous program and the concurrency graph for the program. Note that the concurrency graph captures both paths in the program.

Since concurrency-graphs ignore statements that are unrelated to synchronization, a path through a concurrency-graph corresponds to a SYN-sequence of a program, not a path of the program.

This is because two or more different program paths may exercise the same SYN-sequence.
In general, the concurrency-graph of program CP may not be an accurate representation of the feasible SYN-sequences of CP.

For example, some paths in the graph might not be allowed if the predicates in selective wait and if-else statements were taken into consideration.

Building accurate graph models of programs is hard to do.

Also, reachability graph models are limited by the state explosion problem, which refers to the rapid increase in the number of states as the number of threads increases.

Structural coverage criteria for synchronous message-passing programs can be defined based on the concurrency graph model:

- *all-concurrency-paths* requires all paths through the concurrency graph (i.e., all SYN-sequences) to be exercised at least once. This criterion is impossible to satisfy if cycles exist in the concurrency graph.

- *all-proper-concurrency-paths* requires all proper paths through the concurrency graph to be exercised at least once. A proper path is a path that does not contain any duplicate states, except that the last state of the path may be duplicated once. Thus, proper paths have a finite length.

- *all-edges-between-concurrency-states* requires that for each edge E in the concurrency graph there is at least one path along which E occurs.

- *all-concurrency-states* requires that for each state S in the concurrency graph there is at least one path along which S occurs.
- **all-possible-rendezvous** requires that for each state $S$ in the concurrency graph that involves a rendezvous between threads there is at least one path along which $S$ occurs.

The subsumes hierarchy for these criteria is shown below:

```
  all-concurrency-paths
    ↓
all-proper-concurrency-paths
    ↓
all-edges-between-concurrency-states
    ↓
all-concurrency-states
    ↓
all-possible-rendezvous
```

Once a coverage criterion is chosen, a set of SYN-sequences can be selected from the concurrency graph to satisfy the selected criterion.

**Definitions of Correctness and Faults for Concurrent Programs**

The purpose of testing is to find failures, i.e., show that a program is incorrect.

How to define the correctness of a concurrent program?

What types of failures and faults can be found when testing concurrent programs?
**Defining Correctness for Concurrent Programs**

Let CP be a concurrent program.

A SYN-sequence is said to be feasible for CP with input X if this SYN-sequence can be exercised during an execution of CP with input X.

\[
\text{Feasible}(CP,X) = \text{the set of feasible SYN-sequences of CP with input } X.
\]

A SYN-sequence is said to be valid for CP with input X if, according to the specification of CP, this SYN-sequence is expected to be exercised during an execution of CP with input X.

\[
\text{Valid}(CP,X) = \text{the set of valid SYN-sequences of CP with input } X.
\]

Sets *Feasible* and *Valid* are, in general, impossible to determine.

But they are still useful for defining the correctness of concurrent programs, classifying the types of failures in concurrent programs, and comparing various validation techniques for concurrent programs.
CP is said to be correct for input X (with respect to the specification of CP) if:

(a) $\text{Feasible}(\text{CP}, X) = \text{Valid}(\text{CP}, X)$, and
(b) every possible execution of CP with input X produces the correct (or expected) result.

The result of an execution includes the output and termination condition of the execution. The possible types of abnormal termination include divide-by-zero errors, deadlock, expiration of allocated CPU-time, etc.

CP is said to be correct (with respect to the specification of CP) if and only if CP is correct for every possible input.

**Failures and Faults in Concurrent Programs**

Based on our earlier definition of correctness, CP is incorrect for input X if and only if one or more of the following conditions hold:

(a) $\text{Feasible}(\text{CP}, X)$ is not equal to $\text{Valid}(\text{CP}, X)$. Thus, one or both of the following conditions hold:
   (a1) there exists at least one SYN-sequence that is feasible but invalid for CP with input X
   (a2) there exists at least one SYN-sequence that is valid but infeasible for CP with input X
(b) There exists an execution of CP with input X that exercises a valid (and feasible) SYN-sequence, but computes an incorrect result.

Note that a sequential program may have computation failures but not synchronization failures.
Consider the faulty bounded buffer solution in Listing 7.7.

- Assume that the capacity of the buffer is two
- Assume that a single producer and a single consumer execute deposit.call() and withdraw.call() three times, respectively.

Note that the set of feasible SR-sequences and the set of valid SR-sequences of this program are independent of the program’s inputs.

Thread boundedBuffer contains a fault. The guard for the deposit alternative:

\[
\text{deposit.guard (fullSlots} \leq \text{capacity)};
\]

should be

\[
\text{deposit.guard (fullSlots < capacity)};
\]

This fault can cause a synchronization failure since it allows an item to be deposited when the buffer is full.

Suppose the producer deposits items 'A', 'B', and 'C' and the following invalid SR-sequence is exercised:

\[
\text{(producer, boundedBuffer, deposit, rendezvous),}
\]
\[
\text{(producer, boundedBuffer, deposit, rendezvous),}
\]
\[
\text{(producer, boundedBuffer, deposit, rendezvous), // deposit into a full buffer}
\]
\[
\text{(consumer, boundedBuffer, withdraw, rendezvous),}
\]
\[
\text{(consumer, boundedBuffer, withdraw, rendezvous),}
\]
\[
\text{(consumer, boundedBuffer, withdraw, rendezvous).}
\]

The above SR-sequence starts with three consecutive rendezvous at deposit, followed by three consecutive rendezvous at withdraw:

- The output of this execution is ('C','B','C'), not the expected output ('A','B','C').
- This is an example of failure condition (iii), since this SR-sequence is invalid and the output ('C','B','C') is incorrect.
If an execution of *boundedBuffer* with input ('C','B','C') exercises the above invalid SR-sequence, then the output of this execution is ('C','B','C')”

- This is an example of failure condition (iv) above, since this SR-sequence is invalid but the output ('C','B','C') is correct.
- An execution of *boundedBuffer* that does not exercise the above SR-sequence will not produce an invalid SR-sequence nor will it produce an incorrect result.

Finally, assume that the incorrect guard for *deposit* is modified to:

```
deposit.guard (fullSlots+1 < capacity);
```

Now thread *boundedBuffer* allows at most one character in the buffer.

In this case, the set of feasible SR-sequences of *boundedBuffer* is a proper subset of the set of valid SR-sequences of *boundedBuffer*, i.e., *boundedBuffer* has a missing path:

- *boundedBuffer* still has a possible synchronization failure.
- This failure cannot be detected by a non-deterministic execution of *boundedBuffer* since such an execution will always exercise an SR-sequence that is feasible and valid, and will always produce a correct result.
final class boundedBuffer extends TDThread {
    private selectableEntry deposit, withdraw;
    private int fullSlots=0; private int capacity = 0;
    private Object[] buffer = null; private int in = 0, out = 0;
    public boundedBuffer(selectableEntry deposit, selectableEntry withdraw, int capacity) {
        this.deposit = deposit; this.withdraw = withdraw; this.capacity = capacity;
        buffer = new Object[capacity];
    }
    public void run() {
        try {
            selectiveWait select = new selectiveWait();
            select.add(deposit); // alternative 1
            select.add(withdraw); // alternative 2
            while(true) {
                deposit.guard (fullSlots <= capacity); // *** (fullSlots < capacity)
                withdraw.guard(fullSlots > 0);
                switch (select.choose()) {
                    case 1: Object o = deposit.acceptAndReply();
                        buffer[in] = o; in = (in + 1) % capacity; ++fullSlots;
                        break;
                    case 2: withdraw.accept();
                        Object value = buffer[out]; withdraw.reply(value);
                        out = (out + 1) % capacity; --fullSlots;
                        break;
                }
            }
        } catch (InterruptedException e) {} 
        catch (SelectException e) {
            System.out.println("deadlock detected"); System.exit(1);
        }
    }
}

Listing 7.7 A faulty bounded buffer.
Approaches to Testing Concurrent Programs

Two types of testing:

- **black-box testing**: Access to CP's implementation is not allowed during black-box testing. Thus, only the specification of CP can be used for test generation, and only the result (including the output and termination condition) of each execution of CP can be collected.

- **white-box testing**: Access to CP's implementation is allowed during white-box testing. In this case, both the specification and implementation of CP can be used for test generation. Also, any desired information about each execution of CP can be collected.

White-box testing may not be practical during system or acceptance testing, due to the size and complexity of the code or the inability to access the code.

**Grey-box testing** is a third type of testing that lies somewhere between the first two approaches: During an execution of CP, only the result and SYN-sequence can be collected:

- only the specification and the SYN-sequences of CP can be used for test generation
- an input and a SYN-sequence can be used to deterministically control (see below) the execution of CP.
7.4.1 Non-Deterministic Testing

Non-deterministic testing of a concurrent program CP involves the following steps:
1. Select a set of inputs for CP
2. For each selected input X, execute CP with X many times and examine the result of each execution

Multiple, non-deterministic executions of CP with input X may exercise different SYN-sequences of CP and thus may detect more failures than a single execution.

This approach can be used during both (limited) white-box and black-box testing.

Non-deterministic testing tries to exercise as many distinct SYN-sequences as possible:
- repeated executions do not always execute different SYN-sequences.
- the “probe effect”, which occurs when programs are instrumented with testing and debugging code, may make it impossible for some failures to be observed.

Techniques for exercising different SYN-sequences during non-deterministic testing:
- change the scheduling algorithm used by the operating system, e.g., change the value of the time quantum
- insert Sleep statements into the program with the sleep time randomly chosen to ensure a non-zero probability for exercising an arbitrary SYN-sequence,

Still:
- some sequences are likely to be exercised many times, which is inefficient, and some may never be exercised at all.
- the result of the execution must be checked, which is difficult and tedious if done manually.
7.4.2 Deterministic Testing

Deterministic testing of a concurrent program CP involves the following steps:

1. Select a set of tests, each of the form \((X, S)\), where \(X\) and \(S\) are an input and a complete SYN-sequence of CP, respectively.

2. For each selected test \((X, S)\), force a deterministic execution of CP with input \(X\) according to \(S\). This forced execution determines whether \(S\) is feasible for CP with input \(X\). (Since \(S\) is a complete SYN-sequence of CP, the result of such an execution is deterministic.)

3. Compare the expected and actual results of the forced execution (including the output, the feasibility of \(S\), and the termination condition). If the expected and actual results are different, a failure is detected in the program (or an error was made when the test sequence was generated). A replay tool can be used to locate the fault that caused the failure. After the fault is located and CP is corrected, CP can be executed with each test \((X, S)\) to verify that the fault has been removed and that in doing so, no new faults were introduced.

Note that for deterministic testing, a test for CP is not just an input of CP. A test consists of an input and a SYN-sequence, and is referred to as an IN-SYN test.
Deterministic testing provides several advantages over non-deterministic testing:

- Non-deterministic testing may leave certain paths of CP uncovered. Several path-based test coverage criteria were described earlier. Deterministic testing allows carefully selected SYN-sequences to be used to test specific paths of CP.

- Non-deterministic testing exercises feasible SYN-sequences only; thus, it can detect the existence of invalid, feasible SYN-sequences of CP, but not the existence of valid, infeasible SYN-sequences of CP. Deterministic testing can detect both types of failures.

- After CP has been modified to correct an error or add some functionality, deterministic regression testing with the inputs and SYN-sequences of previous executions of CP provides more confidence about the correctness of CP than non-deterministic testing of CP with the inputs of previous executions.
Mutation-Based Testing {A&O Ch. 5}

Mutation-based testing helps the tester create test cases and then interacts with the tester to improve the quality of the tests.

Mutation-based testing subsumes the coverage criteria defined earlier. That is, if mutation coverage is satisfied, then the criteria in Fig. 7.3 are also satisfied.

If a test case causes a mutant program to produce output different from the output of the program under test:
- that test case is strong enough to detect the faults represented by that mutant,
- the mutant is considered to be distinguished from the program under test.

Each set of test cases is used to compute a mutation score.
- A score of 100% indicates that the test cases distinguish all mutants of the program under test and are adequate with respect to the mutation criterion.
- Some mutants are functionally equivalent to the program under test and can never be distinguished. This is factored into the mutation score.
Non-deterministic execution behavior creates the following problem when mutation testing is applied to concurrent programs:

A mutant and the program under test may produce different results, but this not sufficient to mark the mutant as distinguished. Different actual results may be a product of non-determinism and not the mutation.

This problem can be solved by using a combination of deterministic testing and non-deterministic mutation-based testing.

Deterministic testing can be used to distinguish mutant programs by differentiating the output and the feasible SYN-sequences of the mutants from those of the program under test.

=> If the SYN-sequence randomly exercised by CP during non-deterministic testing is infeasible for the mutant program, or this sequence is feasible but the mutant program produces results that are different from CP’s, then the mutant is marked as distinguished.
Example 1. Assume that the program under test is an incorrect version of the bounded buffer that allows at most one (instead of two) consecutive deposits into the buffer. (In other words, the program under test has a fault.) Call this program boundedBuffer1.

A possible mutant of this program is the correct version shown earlier. Call this correct version boundedBuffer2.

Mutant boundedBuffer2 is distinguished by an SR-sequence that exercises two consecutive deposits, as this sequence differentiates the behaviors of these two versions. But this SR-sequence is a valid, infeasible SR-sequence of boundedBuffer1 that cannot be exercised when non-deterministic testing is applied to boundedBuffer1 in line (3).

Example 2. Assume that the program under test is boundedBuffer2, which correctly allows at most two consecutive deposit operations.

A possible mutant of this program is boundedBuffer3, which allows three consecutive deposits. But this SR-sequence is an invalid, infeasible SYN-sequence of boundedBuffer2 that cannot be exercised when non-deterministic testing is applied to boundedBuffer2 in line (3).

⇒ Upon reaching a steady mutation score, select IN_SYN test cases and apply deterministic testing (DT) to CP in line (3) in order to distinguish more mutants.

• The SYN-sequences selected for deterministic testing may need to be infeasible for CP.
• both valid and invalid SYN-sequences should be selected.
Example: Deterministic mutation testing was applied to the correct version of the bounded buffer program, denoted as \textit{boundedBuffer2}.

The result was a set of 95 mutants. Since 14 of the mutations resulted in mutants that were equivalent to \textit{boundedBuffer2}, this left 81 live mutants.

In phase one, we used non-deterministic testing to generate \textit{SR}-sequences of \textit{boundedBuffer2}.

- Random delays were inserted into \textit{boundedBuffer2} to increase the chances of exercising different \textit{SR}-sequences during non-deterministic testing.
- The mutation score leveled off at 71%.
- All four valid and feasible sequences of Deposit (D) and Withdraw (W) events had been exercised:
  \[
  (D,D,W,W,D,W), \quad (D,W,D,D,W,W), \quad (D,W,D,W,D,W), \quad (D,D,W,D,W,W).
  \]
- It was not possible to distinguish any more mutants using non-deterministic testing to select \textit{SR}-sequences of \textit{boundedBuffer2}.

Two of the \textit{SR}-sequences exercised using non-deterministic testing were modified to produce two new invalid \textit{SR}-sequences for phase 2:

- \texttt{(D,D,D,W,W,W)} // invalid: three consecutive deposits into a 2-slot buffer
- \texttt{(W,D,D,W,D,W)} // invalid: the first withdrawal is from an empty buffer

Both of these invalid \textit{SR}-sequences were shown to be infeasible for \textit{boundedBuffer2}, but feasible for the remaining mutants. Thus, all of the remaining mutants were distinguished.
**Reachability Testing**

Combines non-deterministic and deterministic testing

- Generate test sequences dynamically, without constructing any static model
- no redundant interleavings

Can also be considered as a state exploration technique

- Exercise every possible synchronization sequence of a program with a given input
- Some states visited more than once

Important concepts:

*Race*: a receiving event can be synchronized with two or more sending events (in different executions) e.g., the messages sent by two send events can be received by the same receive event

*Race set* of a receive event r: consists of all the send events that r could possibly be synchronized with (in different executions).

*Race Variant*:

- Represents the beginning portion of one or more SYN-sequences that could have happened but didn’t
- Derived from a SYN-sequence by changing one or more race outcomes in the sequence
All the possible SYN-sequence of a program with a given input can be organized into a S/V graph

- Each node \( n \) is labeled by a SYN-sequence, denoted by \( \text{seq}(n) \).
- An edge from node \( n \) to node \( n' \) is labeled by a variant, \( \text{var}(e) \), indicating that \( \text{seq}(n') \) can be collected from a prefix-based test run with \( \text{var}(e) \).

The goal of RT is to generate variants that represent a spanning tree of the S/V-graph of a concurrent program with a given input.

\[ \begin{align*}
\text{p2.send(a);} & \quad \text{\( T_1 \)} & \quad \text{\( T_2 \)} & \quad \text{\( T_3 \)} & \quad \text{\( T_4 \)} \\
\text{x = p2.recv();} & \quad \text{\( \text{\( T_2 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_4 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} \\
\text{u = p3.recv();} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} \\
\text{v = p3.recv();} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} \\
\text{p3.send(c);} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} \\
\end{align*} \]

\[ \begin{align*}
\text{p2.send(b);} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_4 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} \\
\end{align*} \]

\[ \begin{align*}
\text{p3.send(d);} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} \\
\end{align*} \]

\[ \begin{align*}
\text{p3.send(c);} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} & \quad \text{\( \text{\( T_3 \)} \)} \\
\end{align*} \]
Empirical Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Seqs</th>
<th>Program</th>
<th>Seqs</th>
<th>Program</th>
<th>Seqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB-select</td>
<td>144</td>
<td>RW-select</td>
<td>768</td>
<td>DP-monitorSU (3P)</td>
<td>30</td>
</tr>
<tr>
<td>BB-semaphore</td>
<td>324</td>
<td>RW-semaphore</td>
<td>21744</td>
<td>DP-monitorSU (4P)</td>
<td>624</td>
</tr>
<tr>
<td>BB-monitorSU</td>
<td>720</td>
<td>RW-monitorSU</td>
<td>13320</td>
<td>DP-monitorSU (5P)</td>
<td>19330</td>
</tr>
<tr>
<td>BB-monitorSC</td>
<td>12096</td>
<td>RW-monitorSC</td>
<td>61716</td>
<td>DME-3</td>
<td>4032</td>
</tr>
</tbody>
</table>

BB: Bounded Buffer /w 3P + 3C; RW: Reader/Writer /w 3R + 2W;
DP: Dining Philosophers
DME-3: Distributed Mutual Exclusion with 3 processes. (Recall that DME-4 has 1,455,667,200 sequences).

Parallel Reachability Testing: For DME-4, running RT on 175 of the cores available in a cluster of 53 multi-core workstations requires ≈ 130 hours to complete. (Linear speedup.)

T-way reachability testing {A&O Section 4.2}: Exercise a subset of the possible sequences, based on a combinatorial testing strategy called t-way testing. Many fewer sequences, but good at killing mutants.

Modular Reachability Testing:
- Perform RT on an FSM specification to derive specification-based test sequences. (Ten times faster than RT on Java programs.)
- Use the test sequences to test the Java implementation. Only 7,126 sequences required to exhaustively test all the paths of each thread. (Compare to 1.45 billion.)

Stateful Reachability Testing:
- Guarantee each state is visited at least once, instead of exercising each sequence once.
- Benefit: Large reduction possible; Cost: states must be saved; Trick: state pruning
Conclusion

Testing Distributed Mutual Exclusion with 4 processes:

- Develop and verify an FSM specification (Ughh); generate 7,126 modular tests (in 13.5 hours on a cluster), which can be used to verify (in one minute) that the specification and implementation allow the same sequences.
- Non-deterministic (random) testing. Easy to do, but inefficient and coverage is suspect.
- 2-way RT, exercises ≈ 5 million sequences; kills all the mutants
- Exhaustive RT on a cluster: 1.45 billion sequences in 130 hours using 175 cores
- Exhaustive stateful RT on a single computer? Time? Space? TBD.