The purpose of testing is to find program failures. A successful test is a test that causes a program to fail.

Ideally, tests are designed before the program is written.

The conventional approach to testing a program:

- execute the program with each selected test input once
- compare the test results with the expected results.

The term failure is used when a program produces unexpected results.

A failure is an observed departure of the external result of software operation from software requirements or user expectations [IEE90].

Failures can be caused by hardware or software faults.

Ways in which concurrent programs can fail: deadlock, livelock, starvation, and data races.

A software fault (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.

Example: an error in writing an if-else statement may result in a fault that causes an execution to take a wrong branch:

- If this execution produces the wrong result, then it is said to fail;
- otherwise, the result is “coincidentally correct”, and the internal state and path of the execution must be examined to detect the error.
If a test input causes a program to fail, the program is executed again, with the same input, in order to collect debugging information.

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

Since it is not possible to anticipate the information that will be needed to pinpoint the location of a fault, debugging information is collected and refined over the course of many executions until the problem is understood.

**Regression testing:** After the fault has been located and corrected, the program is executed again with each of the previously tested inputs to verify that the fault has been corrected and that in doing so no new faults have been introduced.

This cyclical process of testing, followed by debugging, followed by more testing, breaks down when it is applied to 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 fail.

- **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 does not imply that the detected fault has been corrected.
7.1 Synchronization Sequences of Concurrent Programs

An execution of CP is characterized by CP’s inputs and the sequence of synchronization events that CP exercises, referred to as the *synchronization sequence* (or *SYN-sequence*) of the execution.

The definition of a SYN-sequence can be *language-based* or *implementation-based*

- A language-based definition is based on the concurrent programming constructs available in a given programming language.
- An implementation-based definition is based on the implementation of these constructs, including the interface with the run-time system, virtual machine, and operating system.

Threads in a concurrent program synchronize by performing synchronization operations (e.g., P, V, send, receive) on synchronization objects (e.g., semaphores, channels).

A synchronization event, or “SYN-event”, refers to the execution of one of these operations.

The order in which SYN-events are executed is non-deterministic.
7.1.1 Complete Events vs. Simple Events

For a concurrent programming language or construct, its *complete SYN-event set* is the set of all types of SYN-events.

In general, the complete SYN-event format contains the following information:

(thread name(s), event type, object name, additional information)

This indicates that a specific thread, or a pair of synchronizing threads, executes a SYN-event of a specific type on a specific SYN-object.

Some additional information may be recorded to capture important details about the event, such as the event’s vector timestamp (Section 6.3.6).

The information recorded about a SYN-event may not include the values of the messages that are received or the values of the shared variables that are read or written:
- These values are not needed for program replay, since they will be (re)computed during the normal course of execution.
- They may be needed, say, to assess the correctness of an execution.

The result of an execution of CP is determined by the text of CP and the input and complete SYN-sequence of this execution.
Example 1.

mailbox C1, C2; // synchronous mailboxes

<table>
<thead>
<tr>
<th>Thread1</th>
<th>Thread2</th>
<th>Thread3</th>
<th>Thread4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1.send(msg1); msg = C1.receive(); msg = C1.receive(); C1.send(msg1);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2.send(msg2); msg = C2.receive(); msg = C2.receive(); C2.send(msg2);</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 7.1 Message passing using synchronous mailboxes

SR-events: sending and receiving messages through *mailboxes*.

The complete format of an SR-event is:

(sending thread, receiving thread, mailbox name, event type).

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).

End Example 1.
Some of the information recorded in a complete SYN-event is not needed for program replay. We can use “simple SYN-sequences” for replay.

Example 2:

For the program in Listing 7.1, the format of a simple SR-event is: (Sender, Receiver).

The simple SR-sequence corresponding to the complete SR-sequence in Example 1 is:

(Thread1, Thread2),
(Thread4, Thread3),
(Thread1, Thread2),
(Thread4, Thread3).

end Example 2.

The result of an execution of CP is determined by CP and the input and simple SYN-sequence of this execution (same as for complete SYN-sequences).
Why does the channel name appear in a complete SYN-sequence but not a simple SYN-sequence?

In Example 2 the first channel that Thread1 and Thread2 access is guaranteed to be C1 and does not need to be controlled or checked during replay; thus, the channel name “C1” does not appear in the first simple SR-event.

During replay: there is a need to control which pair of threads, (Thread1 and Thread2) or (Thread4 and Thread3), access channel C1 first:
- Thus, thread names appear in the simple SR-events recorded during an execution.
- During replay, the threads are forced to access the channels in the order given in the recorded sequence.

During testing: need to determine whether the threads are synchronizing on the correct channels, in the correct order. In this case, whether Thread1 and Thread2 can and should synchronize on channel C1 is a question that we need to answer.
- Use complete SR-events, which specify the threads and the channel name for each synchronization event, and a complete SR-sequence, which specifies a synchronization order.
- During execution, try to force the threads to synchronize in the specified order, on the specified channels. If the results are not as expected, then either there is a problem with the program or we made a mistake when we generated the test sequence.

In general, it takes less information to replay an execution than to determine whether an execution is allowed by a program.

Define different types of SYN-sequences for the different activities that occur during testing and debugging.
7.1.2 Total Ordering vs. Partial Ordering

A SYN-sequence can be a \textit{total} or \textit{partial} ordering of SYN-events. The SYN-sequences in Examples 1 and 2 were totally-ordered.

When a concurrent program is tested or debugged using totally-ordered SYN-sequences, events must be executed one-by-one, in sequential order, which can have a significant impact on performance.

A partially-ordered SYN-sequence is actually a collection of sequences – there is one sequence for each thread or SYN-object in the program.

In a partially-ordered sequence, SYN-events that are concurrent are unordered, so that concurrent events can be executed at the same time, which speeds up execution.

Assume that a concurrent program CP consists of threads $T_1, T_2, ..., T_m$, $m>0$, and SYN-objects $O_1, O_2, ..., O_n$, $n>0$. A partially-ordered SYN-sequence of CP may be thread-based or object-based:

\textit{Thread-based sequences:} The general format of a thread-based, partially-ordered SYN-sequence of CP is $(S_1, S_2, ..., S_m)$, where sequence $S_i$, $1 \leq i \leq m$, denotes a totally-ordered sequence of SYN-events of thread $T_i$.

- Each event in $S_i$ has thread $T_i$ as the executing thread and has the general format:
  
  (object name, object order number, event type, other necessary information).

- An object’s order number is used to indicate the relative order of this event among all the events executed on the object (i.e., an event with object order number $i$ is the $i^{th}$ event executed on the object).
Object-based sequences: The general format of an object-based, partially-ordered SYN-sequence of CP is \((S_1, S_2, \ldots, S_n)\), where sequence \(S_j\), \(1 \leq j \leq n\), denotes a totally-ordered sequence of SYN-events on object \(O_j\).

- Each event in \(S_j\) has object \(O_j\) as the SYN-object and has the general format:
  
  \((\text{thread name(s)}, \text{thread order number(s)}, \text{event type}, \text{other necessary information})\).

- A thread’s order number is used to indicate the relative order of this event among all the events executed by the thread.

Thread-based sequences are a natural way to visualize message passing programs. The space-time diagrams in Chapter 6 are partially-ordered, thread-based sequences.

Object-based sequences are helpful for understanding the behavior of an individual SYN-object. For example, object-based M-sequences were defined in Chapter 4 for monitors.

Program replay can be implemented using either thread-based or object-based sequences. Implementation details and the desired user interface may favor one approach over the other.
Example 3.

mailbox C1, C2; // synchronous mailboxes

<table>
<thead>
<tr>
<th></th>
<th>Thread1</th>
<th>Thread2</th>
<th>Thread3</th>
<th>Thread4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1.send</td>
<td>C1.send(msg1); msg = C1.receive();</td>
<td>C1.send(msg1); msg = C1.receive();</td>
<td>C1.send(msg1);</td>
<td></td>
</tr>
<tr>
<td>C2.send</td>
<td>C2.send(msg2); msg = C2.receive();</td>
<td>C2.send(msg2); msg = C2.receive();</td>
<td>C2.send(msg2);</td>
<td></td>
</tr>
</tbody>
</table>

Listing 7.1 Message passing using synchronous mailboxes

The synchronization objects are synchronous mailboxes.

The format of an SR-event in an object-based sequence is:

(sending thread, receiving thread, sender’s order number, receiver’s order number,
eventType)

where:

- the *sending thread* executed the send operation
- the *receiving thread* executed the receive operation
- the *sender’s order number* gives the relative order of this event among all of the
  sending thread’s events
- the *receiver’s order number* gives the relative order of this event among all of the
  receiving thread’s events
- *eventType* is the type of this send-receive event

One feasible object-based SR-sequence of this program is:

Sequence of mailbox C1: (Thread1, Thread2, 1, 1, SendReceive-synchronization),
(Thread4, Thread3, 1, 1, SendReceive-synchronization).

Sequence of mailbox C2: (Thread1, Thread2, 2, 2, SendReceive-synchronization),
(Thread4, Thread3, 2, 2, SendReceive-synchronization).
A thread-based SR-event for thread $T$ is denoted by:

$\text{(channel name, channel’s order number, eventType)}$

where:

- the *channel name* is the name of the channel
- the *channel order number* gives the relative order of this event among all of the channel’s events
- *eventType* is the type of this send-receive event

The thread-based SR-sequence corresponding to the object-based sequence above is:

- Sequence of Thread1: (C1, 1, SendReceive-synchronization), (C2, 1, SendReceive-synchronization).
- Sequence of Thread2: (C1, 1, SendReceive-synchronization), (C2, 1, SendReceive-synchronization).
- Sequence of Thread3: (C1, 2, SendReceive-synchronization), (C1, 2, SendReceive-synchronization).
- Sequence of Thread4: (C2, 2, SendReceive-synchronization), (C2, 2, SendReceive-synchronization).

end Example 3.

Totally-ordered SYN-sequences can be converted into object- and thread-based, partially-ordered SYN-sequences. Object- and thread-based sequences can be converted into each other.

Note that the totally-ordered and partially-ordered SYN-sequences of an execution should have the same "happened before" relation.

Chapter 6 described how to use integer timestamps to translate a partially-ordered sequence into a totally-ordered sequence.
7.2 Paths of Concurrent Programs

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

7.2.1 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

Thread1   Thread2          Thread3
(1) M.send(A);    (2) M.send(B);    (3) X = M.receive();
                      (4) Y = M.receive();
                      (5) output the difference (X – Y) of X and Y
```

Listing 7.2 Program CP using synchronous communication.

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

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

Events $(3, \text{Thread}1)$ and $(4, \text{Thread}2)$ 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 $T_1, T_2, ..., T_n$. A partially-ordered path of CP is $(P_1, P_2, ..., P_n)$, where $P_i$, $1 \leq i \leq n$, 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) If a partially-ordered SYN-sequence $S$ is feasible for CP with input X, there exists only one partially-ordered, feasible path of CP with input X such that the partially-ordered SYN-sequence of this path is $S$. Thus, there exists a one-to-one mapping between partially-ordered, feasible paths of CP with input X and partially-ordered, feasible SYN-sequences of CP with input X.

(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) If two or more totally-ordered, feasible paths of CP with input X have the same totally- or partially-ordered SYN-sequence, these paths produce the same result and thus are considered to be equivalent.

(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) If two or more different partially-ordered, feasible paths of CP have the same partially-ordered SYN-sequence, then their input domains are mutually disjoint. However, this statement is not true for totally-ordered, feasible paths of CP.
We will illustrate relationship (e) with an example. Consider the following program:

<table>
<thead>
<tr>
<th>Thread1</th>
<th>Thread2</th>
<th>Thread3</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>(6) p2.receive();</td>
<td>(7) or</td>
<td>(8) p1.receive();</td>
</tr>
<tr>
<td>(7) or</td>
<td>(9) p2.receive();</td>
<td></td>
</tr>
<tr>
<td>(9) p1.receive();</td>
<td>(10) end select;</td>
<td></td>
</tr>
<tr>
<td>(10) end select;</td>
<td></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.
7.2.2 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.

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.

- The predicates in if-else and loop statements divide the input domain into partitions and define the paths of a program.
- Simple predicates contain a single condition which is either a single Boolean variable (e.g., if (B)) or a relational expression (e.g., if (e1 < e2)), possibly with one or more negation operators (!).
- Compound predicates contain two or more conditions connected by the logical operators AND (\(\land\)) and OR (\(\lor\)), (e.g., if ((e1 < e2) \(\land\) (e2 < e3))).
Predicate coverage criteria require certain types of tests for each predicate:

- **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.

  For example, decision coverage for the predicate \((A \land B)\) is satisfied by two tests, the first being \((A=true, B=true)\) and the second being \((A=true, B=false)\). But neither of these tests causes \(A\) to be false. Condition coverage requires \(A\) to be false at least once.

- **decision/condition coverage** requires both decision coverage and condition coverage to be satisfied. Note that condition coverage can be satisfied without satisfying decision coverage.

  For example, for the predicate \((A \lor B)\), condition coverage is satisfied by two tests: \((A=true, B=false)\) and \((A=false, B=true)\). But neither of these tests causes the predicate to be false. Decision/condition coverage requires the predicate to be false at least once.

- **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 $C_2$.

Example: decision coverage subsumes statement coverage since covering all decisions necessarily covers all statements.

A coverage criterion that subsumes another is not always more effective at detecting failures. Whether or not a failure occurs may also depend on the particular input values that are chosen for a test.

Fig. 7.3 shows a hierarchy of criteria based on the subsumes relation. A path from criterion X to Y indicates that X subsumes Y.

![Hierarchy of sequential, structural coverage criteria based on the subsumes relation.](image)

Figure 7.3 Hierarchy of sequential, structural coverage criteria based on the subsumes relation.
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.

The *all-du-paths* criterion requires tests for "definition-use (du)" pairs: if a variable is defined in one statement and used in another, there should be at least one test path that passes through both statements.

Uses may occur in predicates or in computations.

Under certain assumptions, all-du-paths subsumes decision coverage.
Structural coverage criteria are often defined with respect to a flowgraph model of a program.

In a flowgraph:
- Each circular node represents a statement or a collection of sequential statements that will be executed as a block. That is, if the first statement in the block is executed, then all the statements in the block will be executed.
- The edges between the nodes represent the flow of control from one block of code to the next.

Fig. 7.4 shows 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.

```plaintext
port M; // synchronous port

Thread1      Thread2      Thread3
(1) M.send(A); (2) M.send(B); (3) X = M.receive();
(4) Y = M.receive();
(5) output the difference (X – Y) of X and Y
```

Listing 7.2 Program CP using synchronous communication.

In Listing 7.2, 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 the definition of a path in Section 7.2.1, 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.

Fig. 7.5 shows the flowgraphs for the threads in Listing 7.2 and the concurrency graph for the program. Note that the concurrency graph captures both paths in the program.

![Concurrency Graph Diagram]

Figure 7.5 Concurrency graph for the program in listing 7.2.
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:

- \textit{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.

- \textit{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 in Fig. 7.6.

![Subsumes hierarchy diagram](image)

Figure 7.6 Subsumes hierarchy of structural coverage criteria for concurrent programs.

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

The deterministic testing process, which was illustrated in Section 5.5 for message passing programs and is described in this chapter in Section 7.4, can be applied with the selected test sequences.
7.3 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?

Outline:
- definitions of correctness, failure, and fault for concurrent programs
- formally define three common types of failures, which are known as deadlock, livelock, and starvation.
7.3.1 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 \textit{Feasible} and \textit{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) \( \text{Feasible}(CP, X) = \text{Valid}(CP, X) \), 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.

For some concurrent programs, it is possible that the valid/feasible SYN-sequences are independent of the values of the program inputs. This was true for all of the programs in the previous chapters.

Several possible modifications of condition (a) are given below:

(a1) \( \text{Feasible}(CP, X) \) is a proper subset of \( \text{Valid}(CP, X) \). This condition is used when the specification of CP uses non-determinism to model design decisions that are to be made later. That is, a choice that is to be made during the design process is modeled in the specification as a non-deterministic selection between design alternatives.

- Making design decisions is a reduction of the non-determinism in the specification.
- In this context, specifications and implementations are relative notions in a series of system descriptions, where one description is viewed as an implementation of another description, the specification.
(a2) Valid(CP,X) is a proper subset of Feasible(CP,X). This condition is used when the specification of CP is incomplete and thus is extended by the implementation. In this case, the implementation adds information that is consistent with the specification.

(a3) Valid(CP,X) = Feasible(CP,X)\S, where S is a set of implementation events that are not mentioned in the specification of CP. Feasible(CP,X)\S is obtained from Feasible(CP,X) by deleting from each sequence in Feasible(CP,X) events in S.

- This condition is used when the events in CP's specification are a proper subset of those in CP's implementation.
- Example: In Section 4.10.3 specification-based communication events and sequences were defined for monitor-based programs. A specification may contain communication events such as “deposit” and “withdraw”, while abstracting away implementation events such as entering or executing a monitor or executing a wait or signal operation.

These alternative equivalence relations can be used by other verification techniques and may be more appropriate at certain phases of the life-cycle.

When checking the correctness of an execution it may be convenient to annotate each event in a SYN-sequence with a label that provides an abstract representation of the event, e.g., “receive_acknowledgement”. Labels are useful for

- mapping between (abstract) specifications and their implementations
- determining whether or not a feasible sequence of the implementation is also a valid sequence of the specification
7.3.2 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) Feasible(CP,X) is not equal to Valid(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.

The existence of condition (a) is referred to as a synchronization failure (or "timing error" or "data race")

The existence of condition (b) is referred to as a computation failure.

Note that a sequential program may have computation failures but not synchronization failures.
Assume that an execution exercises a SYN-sequence $S$, and produces a result $R$. Then one of the following conditions holds: (i) $S$ is valid and $R$ is correct
(ii) $S$ is valid and $R$ is incorrect
(iii) $S$ is invalid and $R$ is incorrect
(iv) $S$ is invalid and $R$ is correct.

Condition (ii) implies that a computation failure has occurred,

Condition (iii) and condition (iv) imply that a synchronization failure has occurred.

Note that in condition (iv) a correct result $R$ is produced from an incorrect SYN-sequence $S$, a condition also known as “coincidental correctness”.

If the programmer checks only the correctness of $R$, the invalidity of $S$ will go undetected and might cause condition (iii) to occur in the future for the same input or a different input.

$\Rightarrow$ when CP is executed, collect the SYN-sequences that are exercised and determine the validity of each collected SYN-sequence. (The collected SYN-sequences are also needed for regression testing and debugging.)
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:

```java
deposit.guard (fullSlots <= capacity);
```

should be

```java
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:

```
(producer, boundedBuffer, deposit, rendezvous),
(producer, boundedBuffer, deposit, rendezvous),
(producer, boundedBuffer, deposit, rendezvous), // deposit into a full buffer
(consumer, boundedBuffer, withdraw, rendezvous),
(consumer, boundedBuffer, withdraw, rendezvous),
(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.
7.3.3 Deadlock, Livelock, and Starvation

The absence of deadlock and livelock is usually an implicit requirement of all programs.

Deadlock, livelock, and starvation can be formally defined in terms of the reachability graph (Section 7.2.2.) of a concurrent program.

The reachability graph of program CP, denoted by RG_{CP}, contains all the reachable states of CP.

- a state of RG_{CP} contains the number of the next statement to be executed by each of the threads in CP, and the values of the variables in CP.
- a path of RG_{CP} corresponds to a sequence of statements and SYN-events in CP.

We assume that the reachability graph of CP contains a finite number of states.

Let CP be a concurrent program containing threads T_1, T_2, …, T_r, where r > 1.

A state of CP is denoted as (S_1, S_2, …, S_r, other information):

- S_i, 1 ≤ i ≤ r, is the atomic action or the set of atomic actions that can possibly be executed next by thread T_i
- “other information” in a state may include the values of local and global variables, the contents of message queues, etc.
Let $S = (S_1, S_2, \ldots, S_n, \text{other information})$ be a state of CP:

- If action $S_i$, $1 \leq i \leq r$, is a blocking synchronization statement, and executing $S_i$ blocks thread $T_i$ in state $S$, then thread $T_i$ is said to be blocked in state $S$.
- If thread $T_i$, $1 \leq i \leq r$, is neither terminated nor blocked in state $S$, then $S$ has one or more outgoing transitions for thread $T_i$ resulting from the execution of $S_i$.
- If $S_i$ is an if, while, or assignment statement, then $S$ has exactly one outgoing transition for $T_i$.
- If $S_i$ is a non-deterministic statement, such as a selective wait statement, then $S$ may have multiple outgoing transitions for $T_i$.
- If $S_i$ is an input statement for one or more variables, then $S$ may have one outgoing transition for each possible combination of values of the input variables.

A strong component $G'$ of a directed graph $G$ is a maximal sub-graph of $G$ in which there is a path from each node of $G'$ to any other node of $G'$ [Baase 1988].

Let the condensation of graph $G$, denoted as $\text{Condensed}(G)$, be $G$ modified by considering each strong component as a node.

- $\text{Condensed}(G)$ is cycle-free and has a tree structure.
- A leaf node in $\text{Condensed}(G)$ is a node without child nodes.

Fig. 7.8 shows a directed graph and the condensation of the graph. Nodes B and C are leaf nodes.
7.3.3.1 Deadlock.

Assume that at the end of some execution of program CP there exists a thread T that satisfies these conditions:

- T is blocked due to the execution of a synchronization statement (e.g., waiting for a message to be received)
- T will remain blocked forever, regardless of what the other threads do

Thread T is said to be deadlocked and CP is said to have a deadlock.

Example:
- Thread1 is blocked waiting to receive a message from Thread2
- Thread2 is blocked waiting to receive a message from Thread1

Both Thread1 and Thread2 will remain blocked forever since neither thread is able to send the message for which the other thread is waiting.

Let CP be a concurrent program and S be a state in RG_{CP}:

- If a thread T in CP is blocked in S and all states reachable from S, then T is deadlocked in S and S is a deadlock state of T.
- S is a deadlock state if at least one thread of CP is deadlocked in S. A deadlock state S is a global deadlock state if every thread is either blocked or terminated in S; otherwise, S is a local deadlock state.
- CP has a deadlock if RG_{CP} contains at least one deadlock state.
7.3.3.2 An algorithm for detecting deadlock. Let CP be a concurrent program containing threads $T_1, T_2, \ldots, T_r$.

For each node N in the condensed reachability graph, algorithm DeadlockTest computes two sets of threads:

- Blocked(N) is the set of threads that are blocked in every state in node N. (Remember that all the states in N are in the same strong component of the reachability graph.)
- Deadlock(N) contains $i$, $1 \leq i \leq r$, if and only if thread $T_i$ is deadlocked in every state in node N.

Program CP contains a deadlock if Deadlock(N) is not empty for some node N. Algorithm DeadlockTest is as follows:

- Construct the condensation of RG$_{CP}$, denoted as Condensed(RG$_{CP}$). A node in Condensed(RG$_{CP}$) is a set of states in RG$_{CP}$.
- Perform a depth-first traversal of the nodes in Condensed(RG$_{CP}$). For each node N in Condensed(RG$_{CP}$), after having visited the child nodes of N:
  - Blocked(N) = \{i | thread $T_i$ is blocked in every state in N\}
  - if N is a leaf node of Condensed(RG$_{CP}$) then Deadlock(N) = Blocked(N), else Deadlock(N) = the intersection of Blocked(N) and the Deadlock sets of N’s child nodes.

Let n be the number of transitions in RG$_{CP}$. Since RG$_{CP}$ has only one initial state, the number of states in RG$_{CP}$ is less than or equal to n+1.

- step (a) is at most O(n)
- step (b) at most O(n*r).

So the time complexity of algorithm DeadlockTest is at most O(n*r).
Fig. 7.9 shows four threads and the reachability graph for these threads. (The reachability graph and the condensed graph are the same.) The state labels show the next statement to be executed by each thread.

Algorithm DeadlockTest proceeds as follows:

- Node (2,2,2,end) is a leaf node. Since Thread1, Thread2, and Thread3 are blocked, Deadlock(2,2,2,end) = {Thread1, Thread2, Thread3}.
- In node (2,2,1,1), Thread1 and Thread2 are blocked. The only child node of (2,2,1,1) is node (2,2,2,end), where Deadlock(2,2,2,end) was just computed to be {Thread1, Thread2, Thread3}. Thus, Deadlock(2,2,1,1) = {Thread1,Thread2} ∩ {Thread1,Thread2,Thread3} = {Thread1, Thread2}.
- In node (1,1,2,end), thread Thread3 is blocked. The only child node of (1,1,2,end) is node (2,2,2,end). Thus, Deadlock(1,1,2,end) = {Thread3} ∩ {Thread1,Thread2,Thread3} = {Thread3}.
- In node (1,1,1,1), there are no blocked threads. Thus, Deadlock(1,1,1,1) = { }.

Hence, DeadlockTest has detected a deadlock in the program. States (2,2,1,1) and (1,1,2,2) are both local deadlock states, while state (2,2,2,end) is a global deadlock state.
3.3.3.3 Livelock.

Assume that some statements in CP are labeled as "progress statements”, indicating that
the threads are expected to eventually execute these statements. Examples:

• the last statement of a thread,
• the first statement of a critical section,
• the statement immediately following a loop or a synchronization statement.

If a thread executes a progress statement, it is considered to be “making progress”.

Assume there is an execution of CP that exercises an execution sequence S, and at the
end of S there exists a thread T that satisfies the following conditions, regardless of what
the other threads will do:

• T will not terminate or deadlock
• T will never make progress

Thread T is said to be livelocked at the end of S, and CP is said to have a livelock.

Livelock is the busy-waiting analog of deadlock. A livelocked thread is running (or ready
to run), not blocked, but it can never make any progress.

Example: Incorrect solution 2 in Section 2.1.2 (and reproduced below) has an execution
sequence that results in a violation of the progress requirement for solutions to the critical
section problem (not to be confused with the more general requirement to “make
progress” that we introduced in this section.)

The first statement of the critical section is designated as a progress statement.
int turn = 0;
Thread0 while (true) {
    while (turn != 0) { ; } critical section (2)
    turn = 1; (3)
    non-critical section (4)
} 
Thread1 while (true) {
    while (true) { (1)
        while (turn != 1) { ; } (1)
        critical section (2)
        turn = 0; (3)
        non-critical section (4)
    } (1)
}

Below is a prefix of the execution sequence that violates the progress requirement of the critical section problem.

- Thread0 executes (1), (2), and (3). Now turn is 1.
- Thread1 executes (1), (2), and (3) and then terminates in its non-critical section. Now turn is 0.
- Thread0 executes (4), (1), (2), (3), (4), and (1). Now turn is 1.
- Thread0 is stuck in its busy-waiting loop at (1) waiting for turn to become 0. Thread0 will never exit this busy-waiting loop and enter its critical section, i.e., make any progress. Thus, Thread0 is livelocked.

Let CP be a concurrent program and S a state of RG_CP:

- A thread in CP is said to make progress in S if S contains a progress statement for this thread.
- If a thread T in CP is not deadlocked, terminated, or making progress in S or any state reachable from S, then T is livelocked in S and S is a livelock state for T.
- S is a livelock state of RG_CP if at least one thread is livelocked in S. A livelock state S is a global livelock state if every thread in S is either livelocked or terminated; otherwise, S is a local livelock state.
- CP has a livelock if RG_CP contains at least one livelock state.
7.3.3.4 An algorithm for detecting livelock.

Let CP be a concurrent program containing threads $T_1, T_2, \ldots, T_r$. For each node $N$ in the condensed reachability graph, algorithm $LivelockTest$ computes two sets of threads:

- NoProgress($N$) is the set of threads that are not deadlocked, terminated, or executing a progress statement in any state in $N$.
- Livelock($N$) contains $i, 1 \leq i \leq r$, if and only if thread $T_i$ is livelocked in every state in $N$.

Program CP contains a livelock if Livelock($N$) is not empty for some node $N$. Algorithm $LivelockTest$ is as follows:

(a) Construct Condensed($RG_{CP}$).

(b) Perform a depth-first traversal of the nodes in Condensed($RG_{CP}$). For each node $N$ in Condensed($RG_{CP}$), after having visited the child nodes of $N$:

- NoProgress($N$) = $\{i \mid$ thread $T_i$ is not deadlocked, terminated, or executing a progress statement in any state in $N\}$,
- if $N$ is a leaf node of Condensed($RG_{CP}$) then Livelock($N$) = NoProgress($N$), else Livelock ($N$) = the intersection of NoProgress($N$) and the NoProgress sets of $N$’s child nodes.

Algorithm $LivelockTest$ has the same time complexity as algorithm $DeadlockTest$. 

7.3.3.5 Starvation.

Assume that the scheduling policy used in executing a concurrent program CP is fair, i.e., a thread ready for execution will eventually be scheduled to run.

A cycle in $RG_{CP}$ is said to be a fair cycle if for every thread $T$ in CP, either this cycle contains at least one transition for $T$, or $T$ is blocked or terminated in every state on this cycle.

Informally, CP is said to have a starvation if CP can reach a state on a fair cycle such that some thread in CP is not deadlocked, livelocked, or terminated in this state, but this thread may not make any progress in any state on this cycle.
Incorrect solution 3 in Section 2.1.3 (and reproduced below) has a starvation.

```java
boolean intendToEnter0 = false, intendToEnter1 = false;

Thread0
while (true) {
    intendToEnter0 = true; (1) intendToEnter0 = false; (3)
    while (intendToEnter0) { (2) intendToEnter0 = true; (5)
        critical section (6) critical section (6)
    } (7) intendToEnter1 = false; (7)
    non-critical section (8) non-critical section (8)
}

Thread1
while (true) {
    intendToEnter1 = true; (1) intendToEnter1 = false; (3)
    while (intendToEnter1) { (2) intendToEnter1 = true; (5)
        critical section (6) critical section (6)
    } (7) intendToEnter0 = false; (7)
}
```

State (4,2) is not a livelock state for Thread0 since the following sequence allows Thread0 to enter its critical section (4,2) are the next statements to be executed by Thread0 and Thread1, respectively:

(a) Thread1 executes (2) and (6), enters and exits it critical section, and executes (7)
(b) Thread0 executes (4), (5), and (2), and then enters its critical section at (6)

State (4,2) has a cycle to itself that contains one transition for Thread0 (representing an iteration of the busy-waiting loop in (4)) and no other transitions. This cycle is not a fair cycle since it does not contain a transition for Thread1.

State (4,2) has another cycle to itself that represents the following execution sequence:

(c) Thread1 executes (2) and (6), enters and exits it critical section, and then executes (7), (8), and (1)
(d) Thread0 executes (4) and stays in its busy-waiting loop

This cycle is fair. After state (4,2) is entered, if this cycle is repeated forever, Thread0 never enters its critical section. State (4,2) is called a starvation state for Thread0.
Let CP be a concurrent program and S a state of RG$_{CP}$:

- A cycle in RG$_{CP}$ is said to be a no-progress cycle for a thread T in CP if T does not make progress in any state on this cycle. (Assume some statements are labeled as “progress statements”).

- A cycle in RG$_{CP}$ is said to be a starvation cycle for a thread T in CP if (1) this cycle is fair, (2) this cycle is a no-progress cycle for T, and (3) each state on this cycle is not a deadlock, livelock, or termination state for T.

- A starvation cycle for thread T is said to be a busy-starvation cycle for T if this cycle contains at least one transition for T, and is said to be a blocking-starvation cycle for T otherwise (i.e., T is blocked in each state on this cycle).

- If state S is on a starvation cycle for thread T then S is a starvation state for T. A starvation state is a global starvation state if every thread in S is either starved or terminated; otherwise, it is a local starvation state.

- CP is said to have a starvation if RG$_{CP}$ contains at least one starvation state.
7.3.3.6 An algorithm for detecting starvation. Let CP be a concurrent program containing threads T₁, T₂, …, Tᵣ.

For each node N in the condensed reachability graph, algorithm StarvationTest computes two sets of threads:

- **NoProgress(N)** is the set of threads that do not terminate in N, and for which N contains a fair, no-progress cycle.
- **Starvation(N)** contains i, 1 ≤ i ≤ r if and only if a starvation cycle for thread Tᵢ exists in N.

Program CP contains a starvation if Starvation(N) is not empty for some node N.

Algorithm StarvationTest is as follows:

(a) Construct Condensed(RG<sub>CP</sub>).

(b) Perform a depth-first traversal of the nodes in Condensed(RG<sub>CP</sub>). For each node N in Condensed(RG<sub>CP</sub>), after having visited the child nodes of N:

- if N does not contain any fair cycles, then Starvation(N) = empty,
- else NoProgress(N) = {i | thread Tᵢ does not terminate in N, and N contains a fair, no-progress cycle for Tᵢ} and Starvation(N) = NoProgress(N) – Deadlock(N) – Livelock(N).

To compute NoProgress(N), we need to search for fair cycles in N.

- We must consider cycles of length at most (1 + #Transitions) where #Transitions is the number of transitions in N.
- The number of cycles with length less than or equal to (1 + #Transitions) is at most O(2<sup>#Transitions</sup>).

Let n be the number of transitions in RG<sub>CP</sub>. The time complexity of algorithm StarvationTest is at most O(r*2<sup>n</sup>).
7.3.3.7 Other Definitions

Local deadlock has been referred to as a deadness error and permanent blocking. Global deadlock has been referred to as infinite wait, global blocking, deadlock, and system-wide deadlock.

Circular deadlock: a circular list of two or more threads such that each thread is waiting to synchronize with the next thread in the list:

- Similar to a circular wait condition that arises during resource allocation (see Section 3.10.4).
- A circular wait condition is a necessary condition for deadlock during resource allocation.
- According to our definition, a deadlock in a concurrent program is different from a deadlock during resource allocation, since the former is not necessarily a circular deadlock.

Alternate definitions of livelock:

- A thread that is spinning (i.e., executing a loop) while waiting for a condition that will never become true.
- The existence of an execution sequence that can be repeated infinitely often without ever making effective progress.

Alternate definitions of starvation:

- A process, even though not deadlocked, waits for an event that may never occur.
- A situation in which processes wait indefinitely,
- A situation in which processes continue to run indefinitely, but fail to make any progress.
Definitions of deadlock, livelock, and starvation based on reachability graphs:

- are independent from the programming language and constructs used to write the program
- are formally defined in terms of the reachability graph of a program
- cover all undesirable situations involving blocking or not making progress
- define deadlock, livelock, and starvation as distinct properties of concurrent programs
- provide a basis for developing detection algorithms.

The mutual exclusion, progress, and bounded waiting requirements for solutions to the critical section problem can be defined in terms of deadlock, livelock, and starvation and the correctness of a solution to the critical section problem can be verified automatically.
7.4 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.

*Limited white-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 in Section 7.2.2. 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.

The selection of IN-SYN tests for CP can be done in different ways:

- Select inputs and then select a set of SYN-sequences for each input
- Select SYN-sequences and then select a set of inputs for each SYN-sequence
- Select inputs and SYN-sequences separately and then combine them
- Select pairs of inputs and SYN-sequences together
Chapters 1 through 6 dealt with various issues that arise during deterministic testing and debugging. These issues are summarized below:

**Program Replay**: Repeating an execution of a concurrent program is called “program replay”.

The SYN-sequence of an execution must be traced so that the execution can be replayed.

- Program replay uses simple SYN-sequences, which have a simpler format than the complete sequences used for testing.
- Definitions of simple SYN-sequences for semaphores, monitors, and message passing were given in Chapters 3 – 6.

The synchronization library developed in the text supports replay, but it does not have the benefit of being closely integrated with a source-level debugger.

**Program Tracing**:

- Chapters 2 – 6 showed how to trace simple and complete SYN-sequences for shared variables, semaphores, monitors and various types of message channels.

- Observability problem: When tracing a distributed program it is difficult to accurately determine the order in which actions occur during an execution. Vector timestamps (Chapter 6) can be used to ensure that an execution trace of a distributed program is consistent with the actual execution.

- For long-running programs, storing all the SYN-events requires too much space. “Adaptive tracing” techniques minimize the number of SYN-events required to exactly replay an execution.
Sequence Feasibility: A sequence of events that is allowed by a program is said to be a feasible sequence.

- Program replay always involves repeating a feasible sequence of events.

- Testing, on the other hand, involves determining whether or not a given sequence is feasible or infeasible. Valid sequences are expected to be feasible while invalid sequences are expected to be infeasible.

- The information and the technique used to determine the feasibility of a SYN-sequence are different from those used to replay a SYN-sequence. The techniques illustrated in Chapters 4 – 6 check the feasibility of complete SYN-sequences of monitors and message channels.

Approaches for selecting valid and invalid SYN-sequences for program testing:

- Collect the feasible SYN-sequences that are randomly exercised during non-deterministic testing. These SYN-sequences can be used for regression testing when changes are made to the program.

- Generate sequences that satisfy a coverage criteria (Section 7.2.2) or that are adequate for the mutation-based testing (Section 7.4.3.2). Mutation testing has the advantage that it requires both valid and invalid sequences to be generated.
**Sequence Validity**: A sequence of actions captured in a trace is definitely feasible, but the sequence may or may not be valid.

The goal of testing is to find valid sequences that are infeasible and invalid sequences that are feasible; such sequences are evidence of a program failure.

A major issue then is how to check the validity of a sequence.
- If a formal specification of valid program behavior is available, then checking the validity of a SYN-sequence can be partially automated.
- Without such a “test oracle”, manually checking validity becomes time-consuming, error prone, and tedious.
**The Probe Effect**: Modifying a concurrent program to capture a trace of its execution may interfere with the normal execution of the program:

- Working programs may fail when instrumentation is removed
- Failures may disappear when debugging code is added.

On the other hand, executions can be purposely disturbed during non-deterministic testing in order to capture as many different SYN-sequences as possible - instrumentation at least offers the prospect of being able to capture and replay the failures that are observed.

One approach to circumventing the probe effect is to systematically generate all the possible SYN-sequences. This approach can be realized through reachability testing if the number of sequences is not too large (Sections 3.10.5, 4.11.4, 5.5.5, and 7.5).

Three different problems:

- The observability problem is concerned with the difficulty of accurately tracing an execution of a distributed program. In Section 6.3.6, we saw how to use vector timestamps to address the observability problem.

- The probe effect is concerned with the ability to perform a given execution at all:
  - Deterministic testing partially addresses the probe effect by allowing us to choose a particular SYN-sequence that we want to exercise.
  - Reachability testing goes one step further and attempts to exercise all possible SYN-sequences.

- The observability problem and the probe effect are different from the replay problem, which deals with 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. [Tsai et al. 1996].

A real-time program may have execution deadlines that will be missed if the program is modified for tracing.
- tracing is performed by using special hardware to remove the probe effect, or by trying to account for or minimize the probe effect.
- Real-time programs may also receive sensor inputs that must be captured for replay.

The text does not consider the special issues associated with timing correctness.

**Tools**: The synchronization library presented in Chapters 1 – 6 is a simple but useful programming tool; however, it is no substitute for an integrated development environment that supports traditional source level debugging as well as the special needs of concurrent programmers.

**Life-Cycle Issues**: Deterministic testing is better suited for the types of testing that occur early in the software life-cycle. Feasibility checking and program replay require information about the internal behavior of a system. Thus, deterministic testing is a form of white-box or limited white-box testing.

Deterministic testing can be applied during early stages of development allowing concurrency bugs to be found as early as possible, when powerful debugging tools are available and bugs are less costly to fix.
Deterministic testing has advantages over non-deterministic testing but it requires considerable effort for selecting SYN-sequences and determining their feasibility.

This effort can be reduced by combining deterministic and non-deterministic testing. Below are four possible strategies for combining these approaches:

(a) Apply non-deterministic testing first with random delays to collect random SYN-sequences and detect failures. Then apply deterministic regression testing with the collected sequences. No extra effort is required for generating SYN-sequences since they are all randomly selected during non-deterministic executions.

(b) Apply non-deterministic testing until test coverage reaches a certain level. Then apply deterministic testing to achieve a higher level of coverage. This strategy is similar to the combination of random and special value testing for sequential programs.

Six Pascal programs were randomly tested against the same specification. Random testing rapidly reached steady-state values for several test coverage criteria: 60% for decision (or branch) coverage, 65% for block (or statement) coverage, and 75% for definition-use coverage, showing that special values (including boundary values) are needed to improve coverage.

(c) SYN-sequences collected during non-deterministic testing can be modified to produce new SYN-sequences for deterministic testing (easier than starting from scratch).

(d) Apply deterministic testing during module and integration testing and non-deterministic testing during system and acceptance testing.
7.4.3.1 Prefix-Based Testing.

The purpose of prefix-based testing is to allow non-deterministic testing to start from a specific program state other than the initial one.

Prefix-based testing uses a “prefix sequence”, which contains events from the beginning part of an execution, not a complete execution.

Prefix-based testing of CP with input X and prefix sequence S proceeds as follows:

1. Force a deterministic execution of CP with input X according to S. If this forced execution succeeds, (i.e., it reaches the end of S), then go to step (2); otherwise S is infeasible.
2. Continue the execution of CP with input X by performing non-deterministic testing of CP.

If S is feasible for CP with input X, then prefix-based testing replays S in step (1).

The purpose of step (1) is to force CP to enter a particular state, e.g., a state in which the system is under a heavy load, so that we can see what happens after that in step (2).

Prefix-based testing is an important part of reachability testing (Section 7.5).
7.4.3.2 Mutation-Based Testing.

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 in Fig. 7.3. That is, if mutation coverage is satisfied, then the criteria in Fig. 7.3 are also satisfied.

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

Figure 7.3 Hierarchy of sequential, structural coverage criteria based on the subsumes relation.

Mutation-based testing also provides some guidance for the generation of invalid SYN-sequences, unlike the criteria in Fig. 7.3.

Mutation-based testing constructs of a set of mutants of the program under test:

- Each mutant differs from the program under test by one mutation.
- A mutation is a single syntactic change made to a program statement, generally inducing a typical programming fault, e.g., changing \( <= \) to \( < \).
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.
Fig. 7.10 shows a mutation-based testing procedure for a sequential program $P$.

Non-deterministic execution behavior creates the following problem:

In line (10), the condition $Actual_p <> Actual_{mi}$ is not sufficient to mark mutant $m_i$ 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.

(1) Generate mutants ($m_1, m_2, ..., m_n$) from $P$;
(2) repeat {
(3) Execute $P$ with test input $X$ producing actual result $Actual_p$;
(4) Compare the actual result $Actual_p$ with the expected result $Expected_p$;
(5) if ($Expected_p != Actual_p$)
(6) Locate and correct the error in $P$ and restart at (1);
(7) else
(8) for (mutant $m_i$, $i <= i <= n$) {
(9) Execute $m_i$ with test input $X$ producing actual result $Actual_{mi}$;
(10) if ($Actual_p <> Actual_{mi}$)
(11) mark mutant $m_i$ as distinguished;
(12) }
(13) }
(14) until (the mutation score is adequate);

Figure 7.10 A mutation-based testing procedure for a sequential program $P$. 
A two-phase procedure for deterministic mutation testing (DMT).

- **phase one:** SYN-sequences are randomly generated using non-deterministic testing, until the mutation score has reached a steady value.
- **phase two:** select IN_SYN test cases and apply deterministic testing until an adequate mutation score is achieved.

Fig. 7.11 shows a phase one procedure using non-deterministic testing to randomly select SYN-sequences for mutation-based testing:

- **line (4):** if SYN-sequence $S_{CP}$ and actual result $Actual_{CP}$ were produced by an earlier execution of CP with input X, then we should execute CP again until a new SYN-sequence or actual result is produced.
- **line (16),** deterministic testing is 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.
repeat 
  Generate mutants \((m_1,m_2,...,m_n)\) from CP;
  Apply non-deterministic testing to randomly execute CP with test input X;
  Assume execution exercises new SYN-sequence \(S_{CP}\), or produces a new actual result \(Actual_{CP}\).

Check which of the following conditions holds:

(a) \(S_{CP}\) is valid and \(Actual_{CP}\) is correct
(b) \(S_{CP}\) is valid and \(Actual_{CP}\) is incorrect
(c) \(S_{CP}\) is invalid and \(Actual_{CP}\) is correct
(d) \(S_{CP}\) is invalid and \(Actual_{CP}\) is incorrect;

if (condition (b), (c), or (d) holds) {
  Locate and correct the error in CP using program replay; apply
  deterministic testing to validate the correction by forcing an execution of CP with IN_SYN test case \((X, S_{CP})\); and restart at (1);
} else
  for (mutant \(m_i\), \(i\leq i\leq n\)) {
    Apply deterministic testing to \(m_i\) with IN_SYN test case \((X, S_{CP})\)
    producing actual result \(Actual_{m_i}\);
    if ((\(S_{CP}\) is infeasible for \(m_i\)) or (
      \(S_{CP}\) is feasible and \(Actual_{CP} \not\equiv Actual_{m_i}\)))
      mark mutant \(m_i\) as distinguished;
  }
} until (the mutation score reaches a steady value);

Figure 7-11 Deterministic Mutation Testing (DMT) using non-deterministic testing to generate SYN-sequences.
It may not be possible to distinguish some of the mutants if non-deterministic testing alone is applied to CP in line (3):

- To distinguish a mutant \( m_i \), we may need to exercise SYN-sequences that are feasible for mutant \( m_i \) but \textit{infeasible} for CP;
- however, in line (3) only \textit{feasible} SYN-sequences of CP can be exercised using non-deterministic testing.

Example 1. Assume that the program under test is an incorrect version of the bounded buffer that allows at most \textit{one} (instead of two) consecutive \textit{deposits} into the buffer. (In other words, the program under test has a fault.) Call this program \texttt{boundedBuffer1}.

A possible mutant of this program is the correct version in Listing 5-10. Call this correct version \texttt{boundedBuffer2}.

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

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

A possible mutant of this program is \texttt{boundedBuffer3} (the mutation shown in Listing 7.7).

Mutant \texttt{boundedBuffer3} is distinguished by an SR-sequence that exercises \textit{three} consecutive \textit{deposits}. But this SR-sequence is an \textit{invalid, infeasible} SYN-sequence of \texttt{boundedBuffer2} that cannot be exercised when non-deterministic testing is applied to \texttt{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.

A phase two test procedure using selected IN_SYN test cases in line (3) is shown in Fig. 7.12.

(1) repeat {
(2) Generate mutants (m_1,m_2,...,m_n) from CP;
(3) Apply DT to deterministically execute CP with a selected IN_SYN test case (X,S);
(4) Compare the actual and expected results of this forced execution:
(5) (a) The results are identical. Then no error is detected by the test (X,S).
(6) (b) The results differ in the feasibility of S.
(7) (c) The results agree on the feasibility of S, but not on the termination condition of CP.
(8) (d) The results agree on the feasibility of S and the termination condition, but not on the output of CP.
(9) if (condition (b), (c), or (d) holds) {
(10) Locate and correct the error in CP using program replay;
(11) Apply DT to validate the correction by forcing an execution of CP with IN_SYN test case (X,S); and restart at (1);
(12) } else
(13) for (mutant m_i, i<=i<=n) {
(14) Apply DT to m_i with IN_SYN test case (X,S);
(15) Compare the actual results of the forced executions of CP and mutant m_i;
(16) if (the results differ in the feasibility of S, the termination condition, or the output)
(17) mark mutant m_i as distinguished;
(18) }
(19) }
(20) until (the mutation score is adequate);

Figure 7.12 Deterministic mutation testing (DMT) using deterministic testing (DT) with selected IN_SYN test cases.
Example: Deterministic mutation testing was applied to the correct version of the bounded buffer program, denoted as `boundedBuffer2`.

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

In phase one, we used non-deterministic testing to generate SR-sequences of `boundedBuffer2`.

- Random delays were inserted into `boundedBuffer2` to increase the chances of exercising different 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), (D,W,D,D,W,W), (D,W,D,W,D,W), (D,D,W,D,W,W).

- It was not possible to distinguish any more mutants using non-deterministic testing to select SR-sequences of `boundedBuffer2`.

Two of the SR-sequences exercised using non-deterministic testing were modified to produce two new invalid SR-sequences for phase 2:
- (D,D,D,W,W,W)  // invalid: three consecutive deposits into a 2-slot buffer
- (W,D,D,W,D,W)  // invalid: the first withdrawal is from an empty buffer

Both of these invalid SR-sequences were shown to be infeasible for `boundedBuffer2`, but feasible for the remaining mutants. Thus, all of the remaining mutants were distinguished.
7.5 Reachability Testing

Non-deterministic testing is easy to carry out, but it can be very inefficient. It is possible that some behaviors of a program are exercised many times while others are never exercised at all.

Deterministic testing allows a program to be tested with carefully selected valid and invalid test sequences.

- Test sequences are usually selected from a static model of the program or of the program’s design.
- Several coverage criteria for reachability graph models were defined in Section 7.2.2.
- However, accurate static models are difficult to build for dynamic program behaviors.

*Reachability testing* is an approach that combines non-deterministic and deterministic testing.

Reachability Testing is based on prefix-based testing, which was described in Section 7.4.3.1:

- prefix-based testing controls a test run up to a certain point, and then lets the run continue non-deterministically.
- The controlled portion of the test run is used to force the execution of a prefix SYN-sequence, which is the beginning part of one or more feasible SYN-sequences of the program.
- The non-deterministic portion of the execution randomly exercises one of these feasible sequences.
Reachability testing uses prefix-based testing to generate test sequences automatically and on-the-fly as the testing process progresses.

- the SYN-sequence traced during a test run is analyzed to derive prefix SYN-sequences that are “race variants” of the trace.
- A race variant represents the beginning part of a SYN-sequence that definitely could have happened but didn’t, due to the way race conditions were arbitrarily resolved during execution.
- The race variants are used to conduct more test runs, which are traced and then analyzed to derive more race variants, and so on.

If every execution of a program with a given input terminates, and the total number of possible SYN-sequences is finite, then reachability testing will terminate and every partially-ordered SYN-sequence of the program with the given input will be exercised.
7.5.1 The Reachability Testing Process

Assume that an execution of some program CP with input X exercises SYN-sequence Q represented by the space-time diagram in Fig. 7.13.

Send events s1 and s2 in Q have a race to see which message will be received first by Thread2.

We can see that there exists at least one execution of CP with input X in which the message sent at s2 is received by r1.

⇒ message sent by s2 is in the race set for r1.

An analysis of sequence Q in Fig. 7.13 allows us to guarantee that s2 can be received at r1. It does not, however, allow us to guarantee that s1 can be received at r2 since we cannot guarantee that Thread2 will always execute two receive statements.

```
Thread2
x = port.receive(); // generates event r1 in Q
if (x>0)
    y = port.receive(); // generates event r2 in Q
```

If r1 receives the message sent by s2 instead of s1, the condition (x>0) may be false, depending on the value of s2’s message.

But if the condition (x>0) is false, the second receive statement will not be executed, and since we do not examine CP’s code during race analysis, it is not safe to put s1 in the race set of r2.
A race variant represents the beginning part of one or more alternative program paths, i.e., paths that could have been executed if the message races had been resolved differently.

Fig. 7.14 shows the race variant produced for sequence Q in Fig. 7.13.

When this variant is used for prefix-based testing, Thread2 will be forced to receive its first message from Thread3, not Thread1.

What Thread2 will do after that is unknown:
- Perhaps Thread2 will receive the message sent at s1, or perhaps Thread2 will send a message to Thread1 or Thread3.
- The dashed arrow from s1 indicates that s1 is not received as part of the variant, though it may be received later.
- In any event, whatever happens after the variant is exercised will be traced, so that new variants can be generated from the trace and new paths can be explored.
Next, we illustrate the reachability testing process by applying it to a solution for the bounded buffer program.

```
Producer                          Consumer                          Buffer
(s1) deposit.call(x1);          (s4) item = withdraw.call();     loop
(s2) deposit.call(x2);          (s5) item = withdraw.call();     select
(s3) deposit.call(x2);          (s6) item = withdraw.call();     when (buffer is not full) =>
                                                                               item = deposit.acceptAndReply();
                                                                               /* insert item into buffer */
                                                                               or
                                                                               when (buffer is not empty) =>
                                                                               withdraw.accept();
                                                                               /* remove item from buffer */
                                                                               withdraw.reply(item);
                                                                               end select;
                                                                               end loop;
```

Assume sequence Q0 is recorded during a non-deterministic execution. Sequence Q0 and the three variants derived from Q0 are shown in Fig 7.15.

The variants are derived by changing the order of deposit (D) and withdraw (W) events whenever there is a message race.

If the message for a receive event r is changed, then all the events that happened after r are removed from the variant (since we cannot guarantee these events can still occur).

Notice that there is no variant in which the first receiving event is for a withdraw. Runtime information collected about the guards will show that the guard for withdraw was false when the first deposit was accepted in Q0. Thus, we do not generate a variant to cover this case.
To create variant $V1$ in Fig. 7.15, the outcome of the race between $s3$ and $s5$ in $Q0$ is reversed. During the next execution of CP, variant $V1$ is used for prefix-based testing.

Sequence $Q1$ in Fig. 7.16 is the only sequence that can be exercised when $V1$ is used as a prefix. No new variants can be derived from $Q1$.

To create variant $V2$ in Fig. 7.15, the outcome of the race between $s3$ and $s4$ in $Q0$ is reversed. When variant $V2$ is used for prefix-based testing, sequence $Q2$ in Fig. 7.16 is the only sequence that can be exercised. No new variants can be derived from $Q2$.

To create variant $V3$ in Fig. 7.15, the outcome of the race between $s2$ and $s4$ in $Q0$ is reversed. During the next execution of CP, variant $V3$ is used for prefix-based testing.
Assume that sequence $Q_3$ in Fig. 7.17 is exercised. Variant $V_4$ can be derived from $Q_3$ by changing the outcome of the race between $s_3$ and $s_5$. Notice that there is no need to change the outcome of the race between $s_2$ and $s_5$ in $Q_3$ since the information collected about the guard conditions will show that a withdraw for $s_5$ cannot be accepted in place of the deposit for $s_2$.

During the next execution of CP, variant $V_4$ is used for prefix-based testing and sequence $Q_4$ in Fig. 7.17 is the only sequence that can be exercised. Reachability testing stops at this point since $Q_0$, $Q_1$, $Q_2$, $Q_3$, and $Q_4$ are all the possible SYN-sequences that can be exercised by this program.
7.5.2 SYN-sequences for Reachability Testing

In order to perform reachability testing, we need to find the race conditions in a SYN-sequence. The SYN-sequences defined for replay and testing were defined without any concern with identifying races.

For reachability testing, an execution is characterized as a sequence of event pairs:

- For asynchronous and synchronous message-passing programs, an execution is characterized as a sequence of send and receive events. (For the execution of a synchronous send statement, the send event represents the start of the send, which happens before the message is received.)
- For programs that use semaphores or locks, an execution is characterized as a sequence of call and completion events for P, V, lock, and unlock operations.
- For programs that use monitors, an execution is characterized as a sequence of monitor call and monitor entry events.

We refer to a send or call event as a sending event, and a receive, completion, or entry event as a receiving event.

We refer to a pair <s,r> of sending and receiving events as a synchronization pair. In the pair <s,r>, s is said to be the sending partner of r, and r is said to be the receiving partner of s.

An arrow in a space-time diagram connects a sending event to a receiving event if the two events form a synchronization pair.
An event descriptor is used to encode certain information about each event:

A descriptor for a sending event \( s \) is denoted by \((\text{SendingThread}, \text{Destination}, \text{op}, i)\), where

- \(\text{SendingThread} \) is the thread executing the sending event
- \(\text{Destination} \) is the destination thread or object (semaphore, monitor, etc)
- \(\text{op} \) is the operation performed (P, V, send, receive, etc)
- \(i \) is the event index indicating that \( s \) is the \(i^{th}\) event of the \(\text{SendingThread}\).

A descriptor for a receiving event \( r \) is denoted by \((\text{Destination}, \text{OpenList}, i)\), where

- \(\text{Destination} \) is the destination thread or object and \(i \) is the event index indicating that \( r \) is the \(i^{th}\) event of the \(\text{Destination}\) thread or object.
- The \(\text{OpenList} \) contains program information that is used to compute the events that could have occurred besides \( r \). Several \(\text{OpenList} \) examples are given below.

The individual fields of an event descriptor are referenced using dot notation. For example, operation \(\text{op} \) of sending event \( s \) is referred to as \(s.\text{op} \).
Tables 7.1 and 7.2 summarize the specific information that is contained in the event descriptors for the various synchronization constructs.

<table>
<thead>
<tr>
<th>Synchronization construct</th>
<th>SendingThread</th>
<th>Destination</th>
<th>Operation</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>asynchronous message passing</td>
<td>sending thread</td>
<td>port ID</td>
<td>send</td>
<td>event index</td>
</tr>
<tr>
<td>synchronous message passing</td>
<td>sending thread</td>
<td>port ID</td>
<td>send</td>
<td>event index</td>
</tr>
<tr>
<td>semaphores</td>
<td>calling thread</td>
<td>semaphore ID</td>
<td>P or V</td>
<td>event index</td>
</tr>
<tr>
<td>locks</td>
<td>calling thread</td>
<td>lock ID</td>
<td>lock or unlock</td>
<td>event index</td>
</tr>
<tr>
<td>monitors</td>
<td>calling thread</td>
<td>monitor ID</td>
<td>method name</td>
<td>event index</td>
</tr>
</tbody>
</table>

Table 7.1 Event descriptors for a sending event $s$.

<table>
<thead>
<tr>
<th>Synchronization construct</th>
<th>Destination</th>
<th>OpenList</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>asynchronous message passing</td>
<td>receiving thread</td>
<td>the port of $r$</td>
<td>event index</td>
</tr>
<tr>
<td>synchronous message passing</td>
<td>receiving thread</td>
<td>list of open ports (including the port of $r$)</td>
<td>event index</td>
</tr>
<tr>
<td>semaphores</td>
<td>semaphore ID</td>
<td>list of open operations (P and/or V)</td>
<td>event index</td>
</tr>
<tr>
<td>locks</td>
<td>lock ID</td>
<td>list of open operations (lock and/or unlock)</td>
<td>event index</td>
</tr>
<tr>
<td>monitors</td>
<td>monitor ID</td>
<td>list of the monitor’s methods</td>
<td>event index</td>
</tr>
</tbody>
</table>

Table 7.2 Event descriptors for a receiving event $r$.
7.5.2.1 Descriptors for asynchronous message passing events.

For asynchronous message-passing, the OpenList of a receive event $r$ contains a single port, which is the source port of $r$.

A send event $s$ is said to be open at a receive event $r$ if port $s.Destination$ is in the OpenList of $r$, which means that the ports of $s$ and $r$ match.

In order for a sending event $s$ to be in the race set of receive event $r$ it is necessary (but not sufficient) for $s$ to be open at $r$.

Fig. 7.18 shows a space-time diagram representing an execution with three threads.

![Space-time diagram](image)

7.5.2.2 Descriptors for synchronous message passing events.

Synchronous message passing may involve the use of selective waits.

The OpenList of a receive event $r$ is a list of ports that had open receive-alternatives when $r$ was selected. Note that this list always includes the source port of $r$.

For a simple receive statement that is not in a selective wait, the OpenList contains a single port, which is the source port of the receive statement.

Event $s$ is said to be open at $r$ if port $s.Destination$ is in the OpenList of $r$. 
Fig. 7.19 shows a space-time diagram representing an execution with three threads.

Assume that whenever \( p2 \) is selected, the alternative for \( p1 \) is open, and whenever \( p1 \) is selected, the alternative for \( p2 \) is closed. This is reflected in the \( \text{OpenLists} \) for the receive events, which are shown between braces \{\ldots\} in the event descriptors.

Note that each solid arrow is followed by a dashed arrow in the opposite direction. The dashed arrows represent the updating of timestamps when the synchronous communication completes. Timestamp schemes are described in Section 7.5.4.

7.5.2.3 Descriptors for semaphore events.

Fig. 7.20 shows an execution involving threads \( T1 \) and \( T2 \) and semaphore \( s \), where \( s \) is a binary semaphore initialized to 1.

There is one timeline for each thread and each semaphore.

A solid arrow represents the completion of a \( P() \) or \( V() \) operation.
The open-lists for the completion events model the fact that P and V operations on a binary semaphore must alternate. This means that the *OpenList* of a completion event for a binary semaphore always contains one of P or V but not both.

A call event *c* for a P or V operation is open at a completion event *e* if *c* and *e* are operations on the same semaphore, i.e., *c.Destination = e.Destination*, and operation *c.op* of *c* is in the *OpenList* of *e*.

### 7.5.2.4 Descriptors for lock events.

If a lock is owned by some thread *T* when a completion event *e* occurs, then each operation in the *OpenList* of *e* is prefixed with *T* to indicate that only *T* can perform the operation. (Recall that if a thread *T* owns lock *L*, then only *T* can complete a *lock()* or *unlock()* operation on *L*.)

For example, if the *OpenList* of a completion event *e* on a lock *L* contains two operations *lock()* and *unlock()* and if *L* is owned by thread *T* when *e* occurs, then the *OpenList* of *e* is \{*T:*lock, *T:*unlock\}.

A call event *c* on lock *L* that is executed by thread *T* is open at a completion event *e* if (i) *c.Destination = e.Destination*; (ii) operation *c.op* is in the *OpenList* of *e*, and (iii) if *L* is already owned when *e* occurs then *T* is the owner.

Fig. 7.21 shows a space-time diagram representing an execution with two threads and a mutex lock *k*. 
The OpenList for e2 reflects the fact that only thread T2 can complete a lock() or unlock() operation on k since T2 owns k when e occurs.

7.5.2.5 Descriptors for monitor events.

The invocation of a monitor method is modeled as a pair of monitor-call and monitor-entry events:

- **SU monitors**: When a thread T calls a method of monitor M, a monitor-call event c occurs on T. When T eventually enters M, a monitor-entry event e occurs on M, and then T starts to execute inside M.

- **SC monitors**: When a thread T calls a method of monitor M, a monitor-call event c occurs on T. A call event also occurs when T tries to reenter a monitor M after being signaled. When T eventually (re)enters M, a monitor-entry event e occurs on M, and T starts to execute inside M.

In these scenarios, we say that T is the calling thread of c and e, and M is the destination monitor of c as well as the owning monitor of e.
A call event \( c \) is open at an entry event \( e \) if the destination monitor of \( c \) is the owning monitor of \( e \), i.e., \( c.Destination = e.Destination \).

The OpenList of an entry event always contains all the methods of the monitor since threads are never prevented from entering any monitor method (though they must enter sequentially and they may be blocked after they enter).

Fig. 7.22 shows a space-time diagram representing an execution involving three threads \( T1 \), \( T2 \), and \( T3 \), an SC monitor \( m1 \) with methods \( a() \) and \( b() \), and an SC monitor \( m2 \) with a single method \( c() \).

Note that if \( m1 \) were an SU monitor, there would be no \( c3 \) event representing reentry.
7.5.3 Race Analysis of SYN-sequences

To illustrate race analysis, we will first consider a program CP that uses asynchronous ports.

- We assume that the messages sent from one thread to another may be received out of order.
- To simplify our discussion, we also assume that each thread has a single port from which it receives messages.

Let Q be an SR-sequence recorded during an execution of CP with input X.

Assume that $a \square b$ is a synchronization pair in Q, $c$ is a send event in Q that is not $a$, and $c$’s message is sent to the same thread that executed $b$. We need to determine whether sending events $a$ and $c$ have a race, i.e., whether $c \square b$ can happen instead of $a \square b$ during an execution of CP with input X.

Furthermore, we need to identify races by analyzing Q, not CP.

In order to accurately determine all the races in an execution, the program’s semantics must be analyzed. Fortunately, for the purpose of reachability testing, we need only consider a special type of race, called a lead race.

Lead races can be identified by analyzing the SYN-sequence of an execution, i.e., without analyzing the source code.
**Definition 6.1:** Let \( Q \) be the SYN-sequence exercised by an execution of a concurrent program \( CP \) with input \( X \). Let \( a \rightarrow b \) be a synchronization pair in \( Q \) and let \( c \) be another sending event in \( Q \). There exists a lead race between \( c \) and \( <a, b> \) if \( c \rightarrow b \) can form a synchronization pair during some other execution of \( CP \) with input \( X \), provided that all the events that happened before \( c \) or \( b \) in \( Q \) are replayed in this other execution.

Note that Definition 6.1 requires all events that can potentially affect \( c \) or \( b \) in \( Q \) to be replayed in the other execution.

If the events that happened before \( b \) are replayed, and the events that happened before \( c \) are replayed, then we can be sure that \( b \) and \( c \) will also occur, without analyzing the code.

**Definition 6.2:** The race set of \( a \rightarrow b \) in \( Q \) is defined as the set of sending events \( c \) such that \( c \) has a (lead) race with \( a \rightarrow b \) in \( Q \).

We will refer to the receive event in \( Q \) that receives the message from \( c \) as receive event \( d \), denoted by \( c \rightarrow d \). (If the message from \( c \) was not received in \( Q \), then \( d \) does not exist.)

To determine whether \( a \rightarrow b \) and \( c \) in \( Q \) have a message race, consider the eleven possible relationships that can hold between \( a, b, c, \) and \( d \) in \( Q \):

1. \( c \rightarrow d \) and \( d \rightarrow b \)
2. \( c \rightarrow d, b \rightarrow d, \) and \( b \rightarrow c \)
3. \( c \) is send event that is never received and \( b \rightarrow c \)
4. \( c \rightarrow d, b \rightarrow d, c \parallel b, \) and \( a \) and \( c \) are send events of the same thread
5. \( c \rightarrow d, b \rightarrow d, c \parallel b, \) and \( a \) and \( c \) are send events of different threads
6. \( c \rightarrow d, b \rightarrow d, c \rightarrow b, \) and \( a \) and \( c \) are send events of the same threads
7. \( c \rightarrow d, b \rightarrow d, c \rightarrow b, \) and \( a \) and \( c \) are send events of different threads
8. \( c \) is a send event that is not received, \( c \parallel b, \) and \( a \) and \( c \) are send events of the same thread
(9) $c$ is a send event that is not received, $c \parallel b$, and $a$ and $c$ are send events of different threads

(10) $c$ is a send event that is not received, $c \Rightarrow b$, and $a$ and $c$ are send events of the same thread

(11) $c$ is a send event that is not received, $c \Rightarrow b$, and $a$ and $c$ are send events of different threads

The happened before relation $e \Rightarrow f$ was defined in Section 6.3.4.

Recall that it is easy to visually examine a space-time diagram and determine the causal relations. For two events $e$ and $f$ in a space-time diagram, $e \Rightarrow f$ if and only if there is no message $e \leftrightarrow f$ or $f \leftrightarrow e$ and there exists a path from $e$ to $f$ that follows the vertical lines and arrows in the diagram.

Fig. 7.23 shows eleven space-time diagrams that illustrate these eleven relations.

Each of the diagrams contains a curve, called the frontier. Only the events happening before $b$ or $c$ are above the frontier. (A send event before the frontier may have its corresponding receive event below the frontier, but not vice versa.)

For each of diagrams (4) through (11), if the send and receive events above the frontier are repeated, then events $b$ and $c$ will also be repeated and the message sent by $c$ could be received by $b$. This is not true for diagrams (1), (2), and (3).
(1) $c \Rightarrow d$ and $d \Rightarrow b$
(2) $c \Rightarrow d, b \Rightarrow d$, and $b \Rightarrow c$
(3) $c$ is a send event that is never received and $b \Rightarrow c$
(4) $c \Rightarrow d, b \Rightarrow d, c \parallel b$, and $a$ and $c$ are send events of the same thread
(5) $c \Rightarrow d, b \Rightarrow d, c \parallel b$, and $a$ and $c$ are send events of different threads
(6) $c \Rightarrow d, b \Rightarrow d, c \Rightarrow b$, and $a$ and $c$ are send events of the same threads
(7) $c \Rightarrow d, b \Rightarrow d, c \Rightarrow b$, and $a$ and $c$ are send events of different threads
(8) $c$ is a send event that is not received, $c \parallel b$, and $a$ and $c$ are send events of the same thread
(9) $c$ is a send event that is not received, $c \parallel b$, and $a$ and $c$ are send events of different threads
(10) $c$ is a send event that is not received, $c \Rightarrow b$, and $a$ and $c$ are send events of the same thread
(11) $c$ is a send event that is not received, $c \Rightarrow b$, and $a$ and $c$ are send events of different threads

Based on these diagrams, we can define the race set of $a \Rightarrow b$ in $Q$ as follows:

**Definition 6.3:** Let $Q$ be an SR-sequence of a program using asynchronous communication and let $a \Rightarrow b$ be a synchronization pair in $Q$. The race set of $a \Rightarrow b$ in $Q$ is

\[ \{ c \mid c \text{ is a send event in } Q; c \text{ has } b \text{'s thread as the receiver; not } b \Rightarrow c; \text{ and if } c \Rightarrow d \text{ then } b \Rightarrow d \}. \]
Fig. 7.24a shows an SR-sequence and the race set for each receive event in this SR-sequence.

Consider send event $s_8$ in Fig. 7.24a.

- Send event $s_8$ is received by Thread2 and is in the race sets for receive events $r_1$ and $r_2$ of Thread2.

- Send event $s_8$ is not in the race set for receive event $r_6$ since $r_6$ happens before $s_8$.

- Send event $s_8$ is not in the race set for receive event $r_7$ since $s_8 \nleq r_8$ but $r_8 \Rightarrow r_7$.

Thus, $s_8$ is in the race sets for receive events of Thread2 that happen before $r_8$ but do not happen before $s_8$. 
The asynchronous ports and mailboxes used in Chapters 5 and 6 are FIFO ports, which means that messages sent from one thread to another thread are received in the order that they are sent.

With FIFO ordering, some of relations (1) through (11) above must be modified:

- Relations (4) and (8) no longer have a race between message \( a \square b \) and \( c \)
- Relations (6) and (10) are not possible
- Relations (5), (7), (9), and (11) have a race between \( a \square b \) and \( c \) if and only if all the messages that are sent from \( c \)'s thread to \( b \)'s thread before \( c \) is sent are received before \( b \) occurs

Thus, the definition of race set must also be modified for FIFO asynchronous SR-sequences.

**Definition 6.4:** Let \( Q \) be an SR-sequence of a program using FIFO asynchronous communication, and let \( a \rightarrow b \) be a message in \( Q \). The race set of \( a \rightarrow b \) in \( Q \) is \( \{ c \mid c \) is a send event in \( Q \); \( c \) has \( b \)'s thread as the receiver; not \( b \Rightarrow c \); if \( c \Rightarrow d \) then \( b \Rightarrow d \); and all the messages that are sent from \( c \)'s thread to \( b \)'s thread before \( c \) is sent are received before \( b \) occurs \}.}
Fig. 7.24b shows a FIFO asynchronous SR-sequence and the race set for each receive event in this SR-sequence. (Since the asynchronous SR-sequence in Fig. 7.24a satisfies FIFO ordering, it is also used in Fig. 7.24b.)

Consider the non-received send event $s_3$ in Fig. 7.24b.

- Send event $s_3$ has Thread2 as the receiver and is in the race sets for receive events $r_7$ and $r_8$ in Thread2.
- Thread2 executes $r_2$ immediately before executing $r_8$.
- Since $r_2$ has the same sender as $s_3$ and $s_2$ is sent to Thread2 before $s_3$ is sent, $s_2$ has to be received by Thread2 before $s_3$ is received.

$\Rightarrow s_3$ is not in the race set for receive event $r_2$. 
In general, sending and receiving events may involve constructs such as semaphores, locks, and monitors, not just message passing.

The following definition describes how to compute the race set of a receiving event assuming all the constructs use FIFO semantics.

**Definition 6.5:** Let $Q$ be a SYN-sequence exercised by program CP. A sending event $s$ is in the race set of a receiving event $r$ if (1) $s$ is open at $r$; (2) $r$ does not happen before $s$; (3) if $<s, r'>$ is a synchronization pair, then $r$ happens before $r'$; and (4) $s$ and $r$ are consistent with FIFO semantics (i.e., all the messages that were sent to the same destination as $s$, and were sent before $s$, are received before $r$).
Below are some examples of race sets:

**Asynchronous message passing.** The race sets for the receive events in Fig. 7.18 are as follows: \( \text{race}(r1) = \{s2\} \) and \( \text{race}(r2) = \text{race}(r3) = \text{race}(r4) = \{\} \).

- Note that \( s3 \) is not in the race set of \( r1 \) because \( s3 \) is sent to a different port and thus \( s3 \) is not open at \( r1 \).
- For the same reason, \( s4 \) is not in the race set of \( r3 \).
- Note also that \( s4 \) is not in the race set of \( r1 \), because FIFO semantics ensures that \( s2 \) is received before \( s4 \).
Synchronous message passing. The race sets of the receive events in Fig. 7.19 are as follows: \( \text{race}(r1) = \{ s2 \}, \text{race}(r2) = \{ \}, \text{race}(r3) = \{ s4 \}, \text{and} \text{race}(r4) = \{ \}. \)

- Since the receive-alternative for port \( p2 \) was open whenever thread \( T2 \) selected the receive-alternative for port \( p1 \), the race set for \( r1 \) contains \( s2 \) and the race set for \( r3 \) contains \( s4 \).
- Since the receive-alternative for \( p1 \) was closed whenever thread \( T2 \) selected the receive-alternative for \( p2 \), the race set for \( r2 \) does not contain \( s3 \).

Semaphores. The race sets of the completion events in Fig. 7.20 are as follows: \( \text{race}(e1) = \{ p2 \} \) and \( \text{race}(e2) = \text{race}(e3) = \text{race}(e4) = \{ \}. \)

- Note that since \( P() \) is not in the \textit{OpenList} of \( e2 \), the race set for \( e2 \) does not contain \( p2 \). This captures the fact that the \( P() \) operation by \( T1 \) could start but not complete before the \( V() \) operation by \( T2 \) and hence that these operations do not race.
**Locks.** The race sets of the completion events in Fig. 7.21 are as follows: \( \text{race}(e_1) = \{l_3\} \) and \( \text{race}(e_2) = \text{race}(e_3) = \text{race}(e_4) = \text{race}(e_5) = \text{race}(e_6) = \{\} \).

- Note that since \( T_2 \) owned lock \( k \) when the operations for events \( e_2, e_3, \) and \( e_4 \) were started, the race sets for \( e_2, e_3, \) and \( e_4 \) are empty. This represents the fact that no other thread can complete a \textit{lock()} operation on \( k \) while it is owned by \( T_2 \).
Monitors. The race sets of the entry events in Fig. 7.22 are as follows: \( \text{race}(e1) = \{ c2 \} \), \( \text{race}(e2) = \text{race}(e3) = \{ \} \), \( \text{race}(e4) = \{ c5 \} \), and \( \text{race}(e5) = \text{race}(e6) = \{ \} \).

- Sending event \( c3 \) is not in the race set of \( e2 \) since \( c3 \) happened after \( e2 \). (Thread T2 entered monitor \( m \) at \( e2 \) and executed a signal operation that caused \( T1 \) to issue the call at \( c3 \).)
7.5.4 Timestamp Assignment

As we just saw, the definition of a race between sending events is based on the happened-before relation, which was defined in Section 6.3.3. and can be computed using vector timestamps.

A thread-centric timestamp has a dimension equal to the number of threads involved in an execution.

An object-centric timestamp has a dimension equal to the number of synchronization objects involved.

A thread-centric scheme is preferred when the number of threads is smaller than the number of synchronization objects, and an object-centric scheme is preferred otherwise.

7.5.4.1 A Thread-Centric Scheme.

The vector timestamp scheme described in Section 6.3.5 is thread-centric and can be used for race analysis:

- Each thread maintains a vector clock. A vector clock is a vector of integers used to keep track of the integer clock of each thread.
- The integer clock of a thread is initially zero, and is incremented each time the thread executes a send or receive event.
- Each send and receive event is also assigned a copy of the vector clock as its timestamp.
Let $T.v$ be the vector clock maintained by a thread $T$. The vector clock of a thread is initially a vector of zeros.

Let $f.ts$ be the vector timestamp of event $f$.

The following rules are used to update vector clocks and assign timestamps to the send and receive events in asynchronous message passing programs:

1. When a thread $T_i$ executes a non-blocking send event $s$, it performs the following operations: (a) $T_i.v[i] = T_i.v[i] + 1$; (b) $s.ts = T_i.v$. The message sent by $s$ also carries the timestamp $s.ts$.

2. When a thread $T_j$ executes a receive event $r$ with synchronization partner $s$, it performs the following operations: (a) $T_j.v[j] = T_j.v[j] + 1$; (b) $T_j.v = max(T_j.v, s.ts)$; (c) $r.ts = T_j.v$.

Fig. 7.25a shows the timestamps for the asynchronous message passing program in Fig. 7.18.
A timestamp scheme for synchronous message passing was also described in Section 6.3.5, but this scheme must be extended before it can be used for race analysis.

The scheme in Section 6.3.5 assigns the same timestamp to send and receive events that are synchronization partners:

- When a thread $T_i$ executes a blocking send event $s$, it performs the operation $T_i.v[i] = T_i.v[i] + 1$. The message sent by $s$ also carries the value of vector clock $T_i.v$.

- When a thread $T_j$ executes a receiving event $r$ that receives the message sent by $s$, it performs the following operations: (a) $T_j.v[j] = T_j.v[j] + 1$; (b) $T_j.v = \max(T_j.v, T_i.v)$; (c) $r.ts = T_j.v$. Thread $T_j$ also sends $T_j.v$ back to thread $T_i$.

- Thread $T_i$ receives $T_j.v$ and performs the following operations (a) $T_i.v = \max(T_i.v, T_j.v)$; (b) $s.ts = T_i.v$.

The exchange of vector clock values between threads $T_i$ and $T_j$ represents the synchronization that occurs between them -- their send and receive operations are considered to be completed at the same time.

Fig. 7.25b shows the timestamps for the synchronous message passing program in Fig. 7.19.
In our execution model for synchronous message passing,

- a send event models the start of a send operation, not its completion.

- For send and receive events that are synchronization partners, the start of the send is considered to happen before the completion of the receive

Thus, when a synchronization completes:

- we use the timestamp of the receive event to update the vector clock of the sending thread, which models the synchronization that occurs between the threads.

- we do not use the timestamp of the receive event to update the timestamp of the send event, since the start of the send is considered to happen before the completion of the receive.

The timestamp scheme synchronous message passing is as follows:

1. When a thread $T_i$ executes a blocking send event $s$, it performs the following operations: (a) $T_i.v[i] = T_i.v[i] + 1$. (b) $s.ts = T_i.v$. The message sent by $s$ also carries the value of vector clock $T_i.v$.

2. When a thread $T_j$ executes a receiving event $r$ that receives the message sent by $s$, it performs the following operations: (a) $T_j.v[j] = T_j.v[j] + 1$; (b) $T_j.v = max(T_j.v, T_i.v)$; (b) $r.ts = T_j.v$. Thread $T_j$ also sends $T_j.v$ back to thread $T_i$.

3. Thread $T_i$ receives $T_j.v$ and performs the operation: $T_i.v = max(T_i.v, T_j.v)$. 
Fig. 7.26 shows the timestamps that are assigned so that race analysis can be performed on the synchronous message passing program in Fig. 7.19.

Note that the dashed arrows represent the application of rule (3).

The timestamps for $s_1$ and $s_2$ indicate that these send events were concurrent even though the synchronization between $T_1$ and $T_2$ happened after the synchronization between $T_3$ and $T_2$.

A thread-centric timestamp scheme for semaphores, locks, and monitors.

We refer to semaphores, locks, and monitors generally as “synchronization objects”.

Each thread and synchronization object maintains a vector clock.

- Position $i$ in a vector clock refers to the integer clock of thread $T_i$
- Synchronization objects do not have integer clocks and thus there are no positions in a vector clock for the synchronization objects.
Let $T.v$ (or $O.v$) be the vector clock maintained by a thread $T$ (or a synchronization object $O$).

The following rules are used to update vector clocks and assign timestamps to events:

1. When a thread $T_i$ executes a sending event $s$, it performs the following operations: (a) $T_i.v[i] = T_i.v[i] + 1$; (b) $s.ts = T_i.v$;

2. When a receiving event $r$ occurs on a synchronization object $O$, the following operations are performed: (a) $O.v = \max(O.v, s.ts)$; (b) $r.ts = O.v$, where $s$ is the sending partner of $r$.

3. **Semaphore/Lock**: When a thread $T_i$ finishes executing an operation on a semaphore or lock $O$, it updates its vector clock using the component-wise maximum of $T_i.v$ and $O.v$, i.e., $T_i.v = \max(T_i.v, O.v)$.

   **SU monitor**: When a thread $T_i$ finishes executing a method on monitor $O$, it updates its vector clock using the component-wise maximum of $T_i.v$ and $O.v$, i.e., $T_i.v = \max(T_i.v, O.v)$.

   **SC monitor**: When a thread $T_i$ finishes executing a method on a monitor $O$, or when a thread $T_i$ is signaled from a condition queue of $O$, it updates its vector clock using the component-wise maximum of $T_i.v$ and $O.v$, i.e., $T_i.v = \max(T_i.v, O.v)$.
Figs. 7.27a and 7.27b show the thread-centric timestamps assigned for the executions in Figs. 7.20 and 7.22, respectively. Again, dashed arrows represent the application of the third rule.

Thread-centric timestamps can be used to determine the happened-before relation between two arbitrary events, as the following Proposition shows:

**Proposition 6.1:** Let CP be a program with threads $T_1$, $T_2$, ..., $T_n$ and one or more semaphores, locks, or monitors. Let $Q$ be a SYN-sequence exercised by CP. Assume that every event in $Q$ is assigned a thread-centric timestamp. Let $f.tid$ be the (integer) thread ID of the thread that executed event $f$, and let $f_1$ and $f_2$ be two events in $Q$. Then, $f_1 \rightarrow f_2$ if and only if (1) $\langle f_1, f_2 \rangle$ is a synchronization pair; or (2) $f_1.ts[f_1.tid] \leq f_2.ts[f_1.tid]$ and $f_1.ts[f_2.tid] < f_2.ts[f_2.tid]$. 
7.5.4.2 An object-centric scheme.

Each thread and synchronization object (port, semaphore, lock, or monitor) maintains a version vector.
- A version vector is a vector of integers used to keep track of the version number of each synchronization object.
- The version number of a synchronization object is initially zero, and is incremented each time a thread performs a sending or receiving event.
- Each sending and receiving event is also assigned a version vector as its timestamp.

Let \( T.v \) (or \( O.v \)) be the version vector maintained by a thread \( T \) (or a synchronization object \( O \)). Initially, the version vector of each thread or synchronization object is a vector of zeros.

The following rules are used to update version vectors and assign timestamps to events:

1. When a thread \( T \) executes a sending event \( s \), \( T \) assigns its version vector as the timestamp of \( s \), i.e., \( s.ts = T.v \).

2. When a receiving event \( r \) occurs on a synchronization object \( O \), letting \( s \) be the sending partner of \( r \), the following operations are performed: (a) \( O.v = \max(O.v, s.ts) \); (b) \( r.ts = O.v \).
3. Semaphore/Lock: When a thread $T$ finishes an operation on a semaphore or lock $O$, $T$ updates its version vector using the component-wise maximum of $T.v$ and $O.v$, i.e., $T.v = \max(T.v, O.v)$.

- SU monitor: When a thread $T$ finishes executing a method on a monitor $O$, $T$ updates its version vector using the component-wise maximum of $T.v$ and $O.v$, i.e., $T.v = \max(T.v, O.v)$.

- SC monitor: When a thread $T$ finishes executing a method on a monitor $O$, or when a thread $T$ is signaled from a condition queue of $O$, $T$ updates its version vector using the component-wise maximum of $T.v$ and $O.v$, i.e., $T.v = \max(T.v, O.v)$.

Timestamps assigned using the above rules are called object-centric timestamps. Note that this scheme is preferred only if the number of synchronization objects is smaller than the number of threads.

Fig. 7.28 shows object-centric timestamps assigned for the executions in Fig 7.27.
Object-centric timestamps cannot be used to determine the happened-before relation between two arbitrary events.

Object-centric timestamps can be used to determine the happened-before relation between two events if at least one of the events is a receiving event, which is sufficient for our purposes.

**Proposition 6.2**: Let CP be a program that uses synchronization objects $O_1$, $O_2$, ..., $O_m$, and let $Q$ be a SYN-sequence exercised by CP. Assume that every event in $Q$ is assigned an object-centric timestamp. Let $e$ be a receiving event on $O_i$, and $f$ be a receiving event on $O_j$, where $1 \leq i, j \leq m$. Then, $e \rightarrow f$ if and only if $e.ts[i] \leq f.ts[i]$ and $e.ts[j] < f.ts[j]$

**Proposition 6.3**: Let CP be a program that uses synchronization objects $O_1$, $O_2$, ..., $O_m$, and let $Q$ be the SYN-sequence exercised by CP. Assume that every event in $Q$ is assigned an object-centric timestamp. Let $r$ be a receiving event on $O_i$, and $s$ be a sending event on $O_j$, where $1 \leq i, j \leq m$. Then, $r \rightarrow s$ if and only if $r.ts[i] \leq s.ts[i]$.


![Diagram](image-url)
7.5.5 Computing Race Variants

The race variants of a SYN-sequence Q are computed by constructing a “race table”, where every row in the race table represents a race variant of Q.

Each race variant V of Q is required to satisfy the following three conditions:

1. if we create V by changing the sending partner of receiving event r, the new sending partner of r must be a sending event in the race set of r;

2. if we create V by changing the sending partner of receiving event r, then any event e that happens after r must be removed from V if e’s execution can no longer be guaranteed

3. there must be at least one difference between Q and V.
As an example, consider the race table for sequence Q0 of the bounded buffer program in Section 7.5.1.

Sequence Q0 and its variants are reproduced in Fig. 7.29. The receiving events in Q0 are numbered and shown with their race sets.

![Figure 7-29.](image)

Table 7.3 is the race table for sequence Q0.

<table>
<thead>
<tr>
<th></th>
<th>r₂</th>
<th>r₃</th>
<th>r₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 7.3 Race table for Q0.

The three columns represent the three receiving events whose race sets are non-empty. Each row represents a race variant of Q0.

Consider the second row, which is (0, 1, -1). Each value indicates how the sending partner of the corresponding receiving event in Q0 is changed to create variant V2:

1. The value 0 indicates that the sending partner of receiving event $r_2$ will be left unchanged.
2. The value 1 indicates that the sending partner of receiving event $r_3$ will be changed to $s_3$, which is the first (and only) send event in $race(r_3)$.
3. The value -1 indicates that receiving event $r_4$ will be removed from V2.
In general, let \( r \) be the receiving event corresponding to column \( j \), \( V \) the race variant derived from row \( i \), and \( v \) the value in row \( i \) column \( j \). Value \( v \) indicates how receiving event \( r \) is changed to derive variant \( V \):

- \( v = -1 \) indicates that \( r \) is removed from \( V \)
- \( v = 0 \) indicates that the sending partner of \( r \) is left unchanged in \( V \)
- \( v > 0 \) indicates that the sending partner of \( r \) in \( V \) is changed to the \( v^{th} \) (sending) event in \( race(r) \), where the sending events in \( race(r) \) are arranged in an arbitrary order and the index of the first sending event in \( race(r) \) is 1.

The receiving events with non-empty race sets are arranged across the columns in left-to-right order with respect to the happened-before relation. (If receiving event \( a \) happens before receiving event \( b \) then the column for \( a \) appears to the left of the column for \( b \).)

Conceptually, a race table is a number system, where each row is a number in the system and each column is a digit in a number. In Table 7.3:

- each receiving event has a race set of size 1. Thus, the base of the number system is 2 and each digit (i.e., column value) has the value 0 or 1. The significance of the digits increases from right to left.

<table>
<thead>
<tr>
<th></th>
<th>( r_2 )</th>
<th>( r_3 )</th>
<th>( r_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 7.3 Race table for Q0.
The rows in the race table are computed iteratively.

Starting with the number 0, all the numbers in the number system are enumerated by adding 1 at each iteration.

Each new number (not including 0) becomes the next row in the table.

Example: For the binary (base 2) system in Table 7.3:
- the first row of the race table is 001.
- adding 1 to this row generates the second row 010
- adding one to the second row generates the third row 011, etc.

The value -1 is used to ensure that each row represents a variant that is a feasible prefix of the program being tested.

To compute the next row in the race table for a SYN-sequence Q, we increment the least significant digit whose value is less than the value of its base minus 1 and whose value is not -1.

Let $t[/]$ be an array representing the next row in the race table.
We use the following rules to ensure that $t[]$ represents a valid race variant V of Q:

1. Whenever we change digit $t[i]$ from 0 to 1, which means that the sending partner of receiving event $r_i$ will be changed to create V, we set $t[j] = -1$, $i < j \leq n$, if $r_i$ happens before $r_j$ and $r_j$ is no longer guaranteed to occur. This removes receiving event(s) $r_j$ from V and ensures that V represents a feasible prefix of one or more executions.

2. Let $b_i$ be the base of digit $t[i]$. Whenever we change digit $t[i]$ from $b_i$ to 0, which means that the sending partner of $r_i$ will be changed back to $r_i$’s original sending partner in Q, we set $t[j] = 0$, $i < j \leq n$, if the current value of $t[j]$ is -1 and there no longer exists an index $k$, $1 \leq k < j$, such that $t[k] > 0$ and $r_k$ happens before $r_j$.

In other words, if $r_i$ is the only event causing $t[j]$ to be set to -1 (due to the application of rule (1)) and we change $r_i$’s sending partner back to its original sending partner in Q, then we need to change $t[j]$ to 0 so that $r_j$ is no longer removed from V.

3. Whenever we increment $t[i]$, we need to determine whether there exists an index $j$ such that $t[j] = m$, $m>0$, and $r_i \rightarrow s$, where $s$ is the $m^{th}$ send event in $race(r_j)$. Array $t[]$ is added to the race table as the next row if and only if such an index $j$ does not exist.

If such an index $j$ does exist, then the sending partner of receiving event $r_j$ was previously changed to $s$ but since $r_i \rightarrow s$ and the sending partner of $r_i$ has just been changed, we can no longer guarantee that send event $s$ will occur.
Example: consider how to add 1 to the number represented by row one in Table 7.3:

- First, we increment the value in the second column (i.e., the column for \( r_3 \)) which is the right-most column whose value is less than its base minus 1 and is not -1. (The base of the third column is 2, which is one more than the number of send events in the race set of \( r_4 \). The value 1 in the third column is not less than 2 minus 1, hence we do not increment the value in the third column.)

- We then apply rule (1), changing the value in the third column to -1 since \( r_3 \) happens before \( r_4 \) in sequence Q0.

- Rule (2) is not applicable since we did not change the second column from 1 to 0.

- For rule (3), observe that no other column has a value greater than 0, hence changing the sending partner of \( r_3 \) does not affect the sending partners of any other receiving events.

Notice that when we change \( t[i] \) from 0 to 1 or from \( b_i \) to 0, we need only check the values of \( t[k] \), \( i < k \leq n \), which are the values in the columns to the right of \( t[i] \). This is because receiving events are ordered from left-to-right based on the happened before relation.

This ordering also ensures that the value represented by \( t[J] \) increases at each iteration. Therefore, this iterative process of computing rows will eventually terminate.
Race analysis never produces two sequences that differ only in the order of concurrent events, i.e., two different totally-ordered sequences that have the same partial ordering.

Example: consider a race table that has columns for two concurrent receiving events $r_1$ and $r_2$. Three variants will be generated:

- one in which the sending partner of $r_1$ is changed
- one in which the sending partner of $r_2$ is changed
- one in which the sending partners of both $r_1$ and $r_2$ are changed

No variants are generated to cover the two orders in which $r_1$ and $r_2$ themselves can be executed ($r_1$ followed by $r_2$, and $r_2$ followed by $r_1$). The order in which $r_1$ and $r_2$ are executed is not specified by the variants.
7.5.6 A Reachability Testing Algorithm

The objective of reachability testing is to exercise every (partially-ordered) SYN-sequence exactly once during reachability testing.

However, if a newly derived race variant $V$ is a prefix of a SYN-sequence $Q$ that was exercised earlier, then prefix-based testing with $V$ could exercise $Q$ again.

Reachability testing algorithms must deal with the potential for duplicate sequences.

One approach to preventing duplicates:
save all the SYN-sequences that are exercised:
a newly derived variant can be used for prefix-based testing only if it is not a prefix of a SYN-sequence that has already been exercised.

For large programs, the cost of saving all of the sequences can be prohibitive, both in terms of the space to store the sequences and the time to search through them.

An alternative approach: identify variants that may cause duplicates and prevent them from being generated.

Some of these variants, however, cannot be prevented, since the best we can say before we execute these variants is that they might produce duplicates. In such cases, we allow the suspect variants to be executed, but we prevent duplicate sequences from being collected from them.
A graph-theoretic perspective: All the possible SYN-sequences that could be exercised by program CP with input X can be organized into a directed graph $G$, which we refer to as the Sequence/Variant graph of CP, or simply the S/V-graph.

Fig. 7.30a is the S/V-graph for the bounded buffer example in section 7.5.1.

Each node $n$ in S/V-graph $G$ represents a SYN-sequence that could be exercised by CP with input X.

Each edge represents a race variant. An edge labeled $V$ from node $n$ to node $n'$ indicates that sequence $n'$ could be exercised by prefix-based testing with the variant $V$ derived from sequence $n$.

For example, in Fig. 7.30a node $Q0$ has two outgoing edges that are both labeled $V3$ since prefix-based testing with variant $V3$ may exercise $Q3$ or $Q4$. 
Note also that an S/V-graph is strongly connected, which means that there is a path in the graph from each node to every other node.

From a graph-theoretic perspective, the goal of reachability testing is to construct a spanning tree of the S/V-graph.

A spanning tree of S/V-graph $G$ is a subgraph of $G$ that is a tree (i.e., a graph with no cycles) and that connects the $n$ nodes of $G$ with $n-1$ edges (i.e., each node, except the root, has one and only one incoming edge.).

Since S/V-graphs are strongly connected, reachability testing can start from an arbitrary node, which explains why the reachability testing process begins by collecting a sequence during a non-deterministic execution.

Also note that each variant is used to conduct a single test run. Therefore, in a spanning tree that represents the reachability testing process, no two edges should be labeled with the same variant.

Fig. 7.30b shows the spanning tree representing the reachability testing process that was illustrated in Section 7.5.1.
A reachability testing algorithm must constrain the way variants are generated so that every sequence is exercised exactly once, i.e., so that the reachability testing process represents a spanning tree of the SV-graph.

The SV-graph is not known when reachability testing begins!

SV-graphs and spanning trees are devices to guide the implementation of, and demonstrate the correctness of, the reachability testing algorithm.

Let $G$ be the S/V graph of program $CP$ with input $X$. If we can find some constraints on the paths through $G$ such that given two arbitrary nodes $n$ and $n'$ in $G$ there is exactly one acyclic path from $n$ to $n'$ that satisfies these constraints, then we can construct a spanning tree of $G$ by enforcing these constraints.

If the reachability testing algorithm implements these constraints, then the reachability testing process will exercise every sequence once.

Node $n$ and $n'$ are the source node and target node, or the source sequence and target sequence, respectively.

Constraints C1 and C2 below constrain the path between $n$ and $n'$ such that there is exactly one path $H$ that satisfies the constraints.
**Constraint C1:** The sending partner of a receiving event can be changed only if the receiving event exists in the target sequence and can be changed at most once along a path.

This constraint ensures that path $H$ between n and n’ is acyclic.

Example: Consider the S/V-graph in Fig. 7.30a.

![Figure 7.30](image)

A reachability testing process involving the cyclic path $Q0Q1Q0$ would not represent a spanning tree of the S/V-graph since trees cannot have cycles.

Such a path would represent a reachability testing process in which sequence Q0 was exercised twice.

Note that receiving event $r4$ has a different sending partner in $Q0$ and $Q1$.

- Variant V1 changes $r4$ so that its sending partner is $s3$ instead of $s5$. Therefore, the edge from $Q1$ to $Q0$ must change the sending partner of $r4$ back to $s5$.
- This is, however, impossible due to Constraint C1, since the sending partner of $r4$ was already changed once in V1 and is not allowed to be changed again.

Therefore, the cyclic path $Q0Q1Q0$ cannot occur during reachability testing.
Constraint C1 can be implemented during reachability testing:

- Associate each receiving event $r$ in variant $V$ with a color that is either black or white.
- If the sending partner of $r$ is changed to derive variant $V$, then $r$’s color is set to black, and this color is inherited by $r$ in any sequences collected from $V$.
- Black receiving events are excluded from race tables (even though they may have non-empty race sets), which prevents the sending partners of black receiving events from being changed again.

Example: In Fig. 7.30a, variant $V1$ was derived by changing the sending partner of $r4$ (see Fig. 7.29).

- the color of $r4$ in $V1$ will be black, and this color will be inherited by $r4$ in $Q1$.
- $r4$ will be excluded from the heading of $Q1$’s race table, preventing the sending partner of $r4$ from being changed again when deriving race variants from $Q1$. 

![Figure 7.29](image)

![Figure 7.30](image)
Constraint C2: Each edge along a path must reconcile as many differences as possible.

A difference between source sequence \( n \) and target sequence \( n' \) refers to a receiving event \( r \) that exists in both sequences but has different sending partners in each sequence.

In terms of these differences, reachability testing can be viewed as the process of transforming, through one or more variants, sequence \( n \) into sequence \( n' \).

Each variant resolves one or more differences between \( n \) and \( n' \).

Constraint C2 says that if there are differences that can be reconciled by an edge, e.g., the sending partner of \( r \) in \( n' \) is in the race set of \( r \) in \( n \), then these differences should be reconciled by this edge.

\[ \Rightarrow \text{ when deriving a variant } V, \text{ if there are receiving events whose sending partners can be changed, but are not changed, then these unchanged receiving events cannot be changed afterwards in any sequences derived from } V. \]

Recall that it is common for a variant to leave some receiving events unchanged since all combinations of changed and unchanged receiving events are enumerated in the race table. Constraint C2 ensures that a particular set of changes occurs in only one variant.
Example: consider sequence Q0 and its three variants in Fig. 7.31.

![Diagram of Q0 and its variants](image)

Notice that the SV-graph contains paths $Q_0Q_2Q_3$ and $Q_0Q_3$, both of which are paths from $Q_0$ to $Q_3$. Constraint C2 excludes path $Q_0Q_2Q_3$ from the spanning tree:

- receiving events $r_2$ and $r_4$ exist in $Q_0$ and also in $Q_3$, but the messages they receive in $Q_0$ are different from the messages they receive in $Q_3$.
- edge $V_2$ along the path $Q_0Q_2Q_3$ only changes the sending partner of $r_4$, leaving the sending partner of $r_2$ unchanged.
- The sending partner of $r_2$ is changed afterwards by the edge from $Q_2$ to $Q_3$.

$\Rightarrow$ path $Q_0Q_2Q_3$ violates Constraint C2, which prohibits $r_2$ from being changed in any sequences derived from $V_2$ since $r_2$ could have been changed in $V_2$ but wasn’t.

Note that edge $V_3$ of path $Q_0Q_3$ can be included in the spanning tree since it changes the sending partners of both $r_2$ and $r_4$. 

Figure 7.31
Constraint C2 can be implemented during reachability testing by removing old sending events from the race sets of old receiving events before variants are derived.

A sending or receiving event in a SYN-sequence $VQ$ is an old event if it also appears in the variant $V$ that was used to collect $VQ$.

Example: consider SYN-sequence $Q2$ in Fig. 7.31.

- Events $r1$ and $s2$ are old events because they appear in both $V2$ (the variant that was used to collect $Q2$) and $Q2$.
- Therefore, $s2$ will be removed from the race set of $r1$ in $Q2$, which means that the sending partner of $r1$ cannot be changed to $s2$ when the race variants of $Q2$ are derived.

$\Rightarrow$ path $Q0Q2Q3$ cannot be generated during reachability testing, as in order to reach $Q3$ from $Q2$, the sending partner of $r1$ must be changed from $s1$ to $s2$.

Figure 7.31
Implementing Constraints C1 and C2 is complicated by the possibility that a receiving event may be removed from a variant and then recollected when the variant is used for prefix-based testing.

Fig. 7.32 shows a variant V containing a receiving event \( r_1 \) that happens before receiving event \( r_2 \).

![Diagram](image)

**Figure 7.32**

Suppose variant \( V \) is used to collect sequence \( VQ \) and some thread executes a sending event \( s \) that is received by Thread2 and is in the race set of \( r_1 \) in \( VQ \).

When the sending partner of \( r_1 \) is changed from \( s_1 \) to \( s \) in order to derive variant \( VQV \) of \( VQ \):

- \( r_2 \) will be removed from \( VQV \) since \( r_1 \) happens before \( r_2 \) in \( VQ \).
- \( r_2 \) will be *recollected* when \( VQV \) is used for prefix-based testing since Thread3 will definitely execute \( r_2 \) again.

In this case, changing the sending partner of \( r_1 \) to \( s \) does not affect the flow of control in Thread3 *before* the point where Thread3 executes \( r_2 \) (though possibly after that point).

So we can guarantee that Thread3 will execute \( r_2 \) in the sequence collected from variant \( VQV \).
Recollected events like \( r2 \) must be handled carefully. There are two cases to consider:

(1) Event \( r2 \) in \( V \) is a black receiving event, indicating that the sending partner of \( r2 \) was changed earlier in the reachability testing process:

When \( r2 \) is recollected during prefix-based testing with \( VQV \), it will be recollected as a new, i.e., white, event.

The send partners of white receiving events can be changed. However, Constraint C1 would be violated if we allowed \( r2 \)’s sending partner to be changed when deriving variants of \( VQV \) since it was already changed earlier.

To prevent a violation of Constraint C1, when \( r2 \)’s color is set to black in \( V \), receiving event \( e \)’s color in \( V \) is also set to black for any receiving event \( e \) that happened before \( r2 \), such as \( r1 \).

This prevents \( r1 \)’s sending partner from being changed when deriving variants from \( VQ \), which in turn prevents \( r2 \) from being removed from any variant derived from \( VQ \) or from any sequence collected afterwards. (Recall that if event \( e \) is colored black in a variant then \( e \) inherits that color in any sequence(s) collected from that variant.)

(2) Event \( r2 \) in \( V \) is a white receiving event, indicating that the sending partner of \( r2 \) has not been changed yet.

When \( r2 \) is recollected during prefix-based testing with \( VQV \), it will be recollected as a white receiving event, but \( r2 \) will also be an old receiving event.
This means that old sending events must be pruned from $r_2$’s race set in the sequences collected from variant $VQV$; otherwise, Constraint C2 would be violated when we changed the sending partner of $r_2$ to an old sending event.

Recollected white receiving events like $r_2$ are handled as follows:

- When the race table of a sequence like VQ is built, $r_2$ should not be removed (i.e., set to -1) when variants like $VQV$ are created, since $r_2$ will definitely be recollected.
- Furthermore, in a variant like VQV, which has recollected event $r_2$ in it, we allow $r_2$’s sending partner to be changed just like the other receiving events in the race table.

If, $r_2$’s sending partner is changed, then nothing special must be done when the variant is used for prefix-based testing.

If the sending partner of $r_2$ is not changed, then the sending partner of $r_2$ must be left unspecified in the variant, since the original sending partner of $r_2$ must be removed.

In this case, $r_2$ must be prevented from receiving a message from any old sending events when the variant is used for prefix-based testing. This prevents Constraint C2 from being violated.

For example, the sending partner $s_2$ of $r_2$ in VQ happens after $r_1$ in VQ, so $s_2$ must be removed when the sending partner of $r_1$ is changed to derive $VQV$.

![Figure 7.32](image-url)
ALGORITHM Reachability-Testing (CP: a concurrent program; I: an input of CP) {

let variants be an empty set;

collect a SYN-sequence $Q_0$ by executing $CP$ with input $X$ non-deterministically;
compute the race variants of $Q_0$, $\text{variants} (Q_0)$, by constructing the race table of $Q_0$
and enforcing Constraints C1 and C2;

$\text{variants} = \text{variants}(Q_0)$;

while (variants is not empty) {

withdraw a variant $V$ from variants;

collect a SYN-sequence $Q$ by performing prefix-based testing with $V$;
compute the race variants of $Q$, $\text{variants} (Q)$, by constructing the race table of $Q$ and
enforcing Constraints C1 and C2;

$\text{variants} = \text{variants} \cup \text{variants}(Q)$;

}

}

Figure 7.33 A reachability testing algorithm.