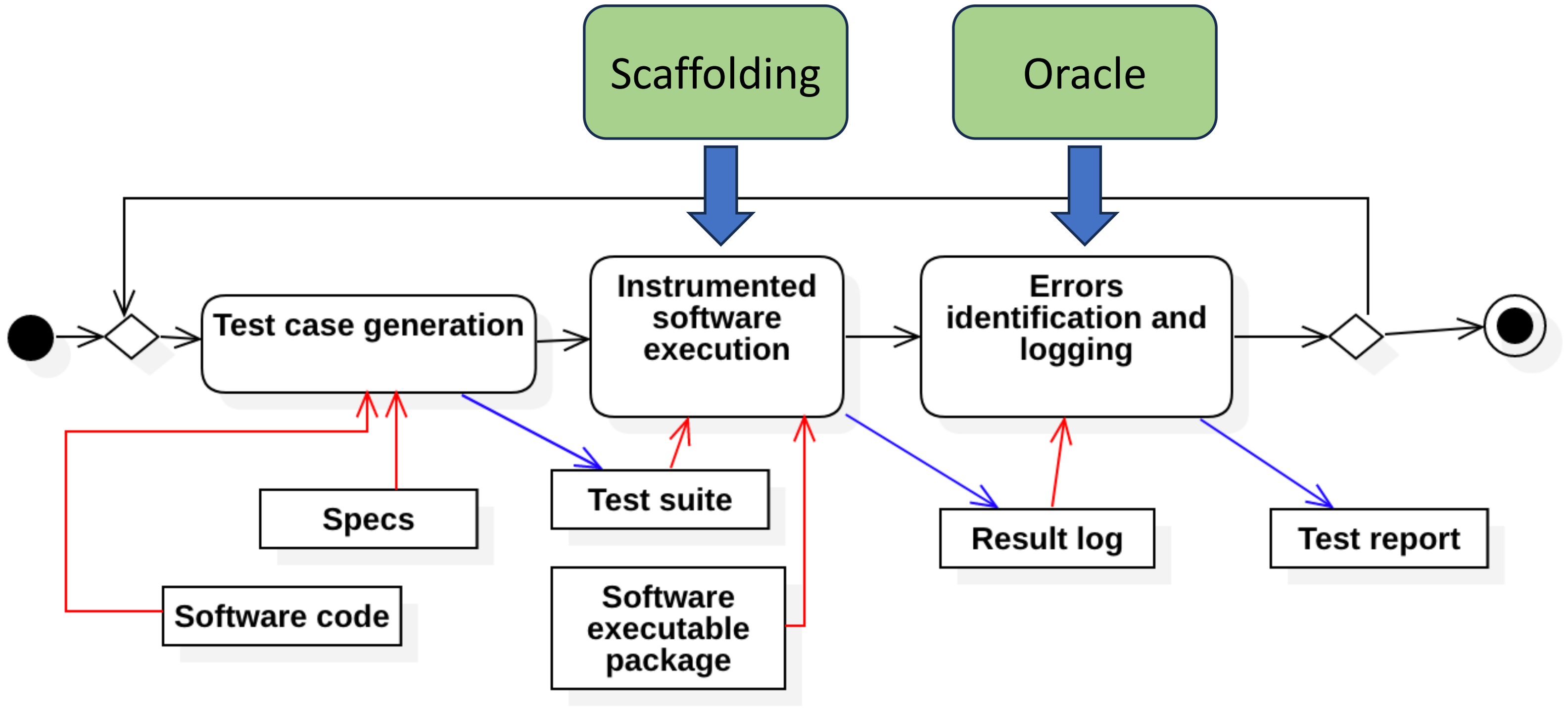
Verification & Validation

Test case generation: introduction, concolic execution, concurrent systems testing

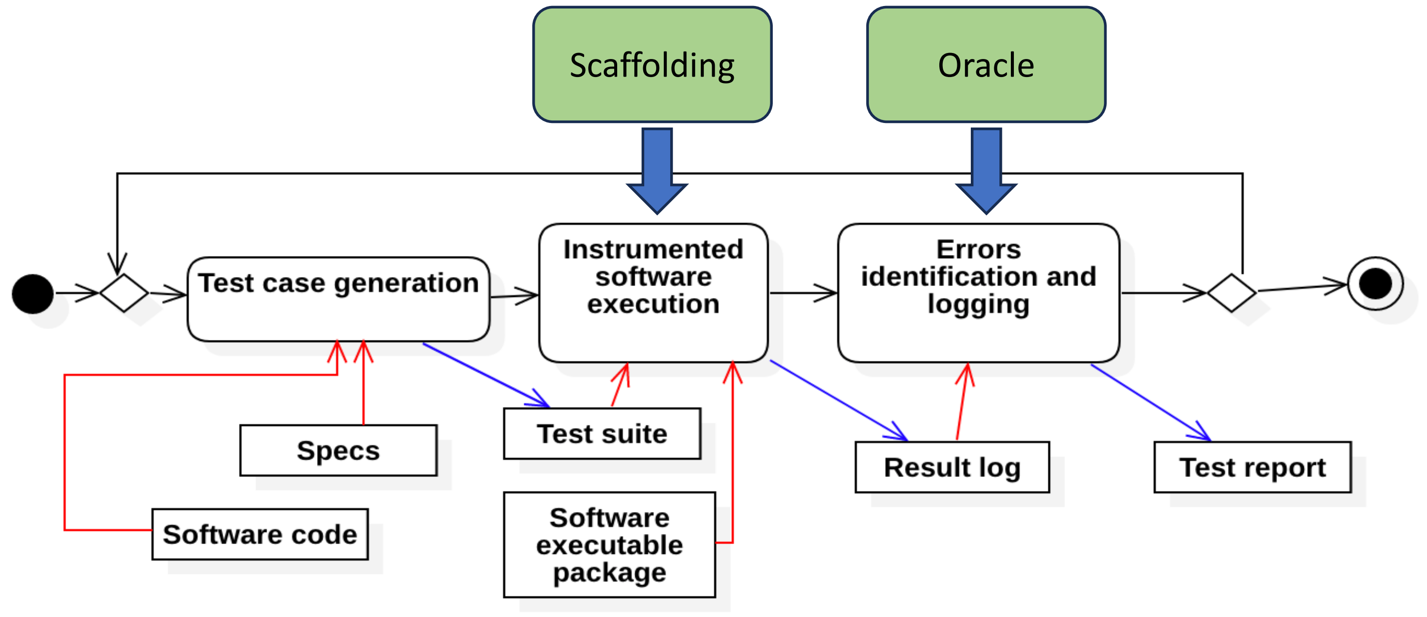
Testing frameworks: *Unit, JMeter

Testing workflow


Scaffolding
Oracle
Test case generation
Instrumented software execution
Errors identification and logging
Specs
Software code
Test suite
Software executable package
Result log
Test report

Test case generation

- Test cases can be generated in a black box or white box manner
  - **White box**: generation is based on code characteristics
  - **Black box**: generation is based on specs characteristics


Scaffolding
Oracle
Test case generation
Instrumented software execution
Errors identification and logging
Specs
Software code
Test suite
Software executable package
Result log
Test report

Test case generation

- Test cases can be defined manually or
- automatically
  - Combinatorial testing = enumerate all possible inputs following some policy (e.g., smaller to larger)
  - Concolic execution = pseudo-random generation of inputs guided by symbolic path properties
  - Fuzz testing (fuzzing) = pseudo-random generation of inputs including invalid, unexpected inputs
  - Search-based testing = explores the space of valid inputs looking for those that improve some metrics (e.g., coverage, diversity, failure inducing capability)
  - Metamorphic testing = generates new test cases based on some metamorphic relations and other previously defined test cases

Concrete-symbolic (concolic) execution

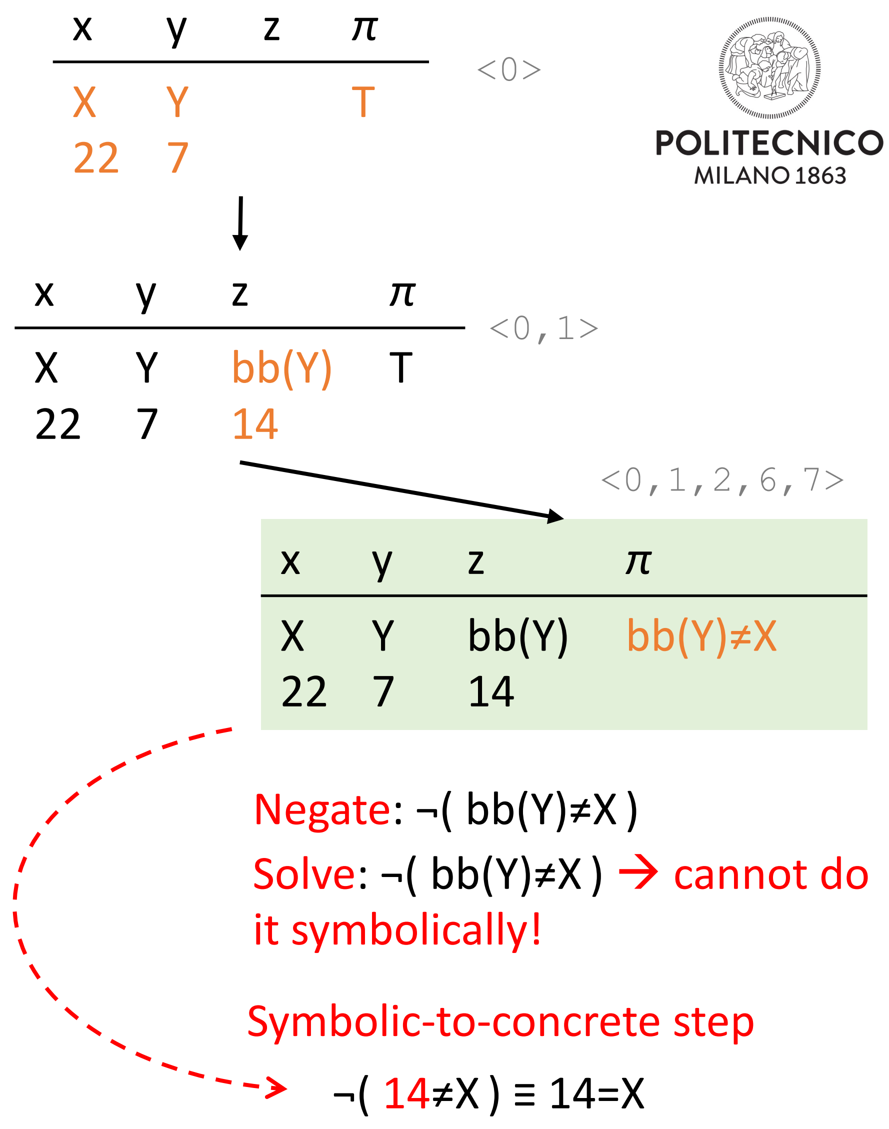
- The very idea
  - Perform symbolic execution **alongside** a concrete one (concrete inputs)
  - The **state** of concolic execution combines a **symbolic** part and a **concrete** part, used as needed to make progress in the exploration
- Steps
  - Concrete to symbolic: derive conditions to explore new paths
  - Symbolic to concrete: simplify conditions to generate concrete inputs

Concolic execution: example

- **Example**: Let's explore all paths of procedure `m2`

```
0: void m2(int x, int y) {
1:   int z := bb(y) //black-box function
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

- We start from a (random) concrete input, at the same time, we build the symbolic condition of the explored path
- In some cases, we cannot solve the symbolic condition…
- Behavior of `bb` is unknown. We execute it with the identified input cases
  - Example: run bb(7) returns 14
- Now the condition can be solved



<0>

x	y	z	π
X	Y		T
22	7		

<0,1>

x	y	z	π
X	Y	bb(Y)	T
22	7	14	

<0,1,2,6,7>

x	y	z	π
X	Y	bb(Y)	bb(Y)≠X
22	7	14	

Negate: ¬(bb(Y)≠X)

Solve: ¬(bb(Y)≠X) → cannot do it symbolically!

Symbolic-to-concrete step

¬(14≠X) ≡ 14=X

x	y	z	π
X	Y		T
14	7		

<0>

Concolic execution: example

- Now the constraint can be solved and we can start a new exploration

```
void m2(int x, int y) {
  int z := bb(y) //black-box function
  if (z == x) {
    z := y + 10
    if (x <= z)
      print("Log message.")
  }
}
```

Solve: ¬(14≠X) ≡ 14=X

Concolic execution: example

- New explorations follow the same approach

```
0: void m2(int x, int y) {
1:   int z := bb(y) //black-box function
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```

<0>

x	y	z	π
X	Y		T
14	7		

<0,1>

x	y	z	π
X	Y	bb(Y)	T
14	7	14	

<0,1,2>

x	y	z	π
X	Y	bb(Y)	bb(Y)=X
14	7	14	

<0,1,2,3>

x	y	z	π
X	Y	Y+10	bb(Y)=X
14	7	17	

<0,1,2,3,4,5,6,7>

x	y	z	π
X	Y	Y+10	bb(Y)=X
14	7	17	X≤Y+10

Negate last condition:
bb(Y)=X ∧ ¬(X≤Y+10) ≡ bb(Y)=X ∧ X>Y+10

Solve: bb(Y)=X ∧ X>Y+10 → cannot!

Concolic execution: example

x	y	z	π
X	Y		T
34	17		

<0>

- New explorations follows the same approach

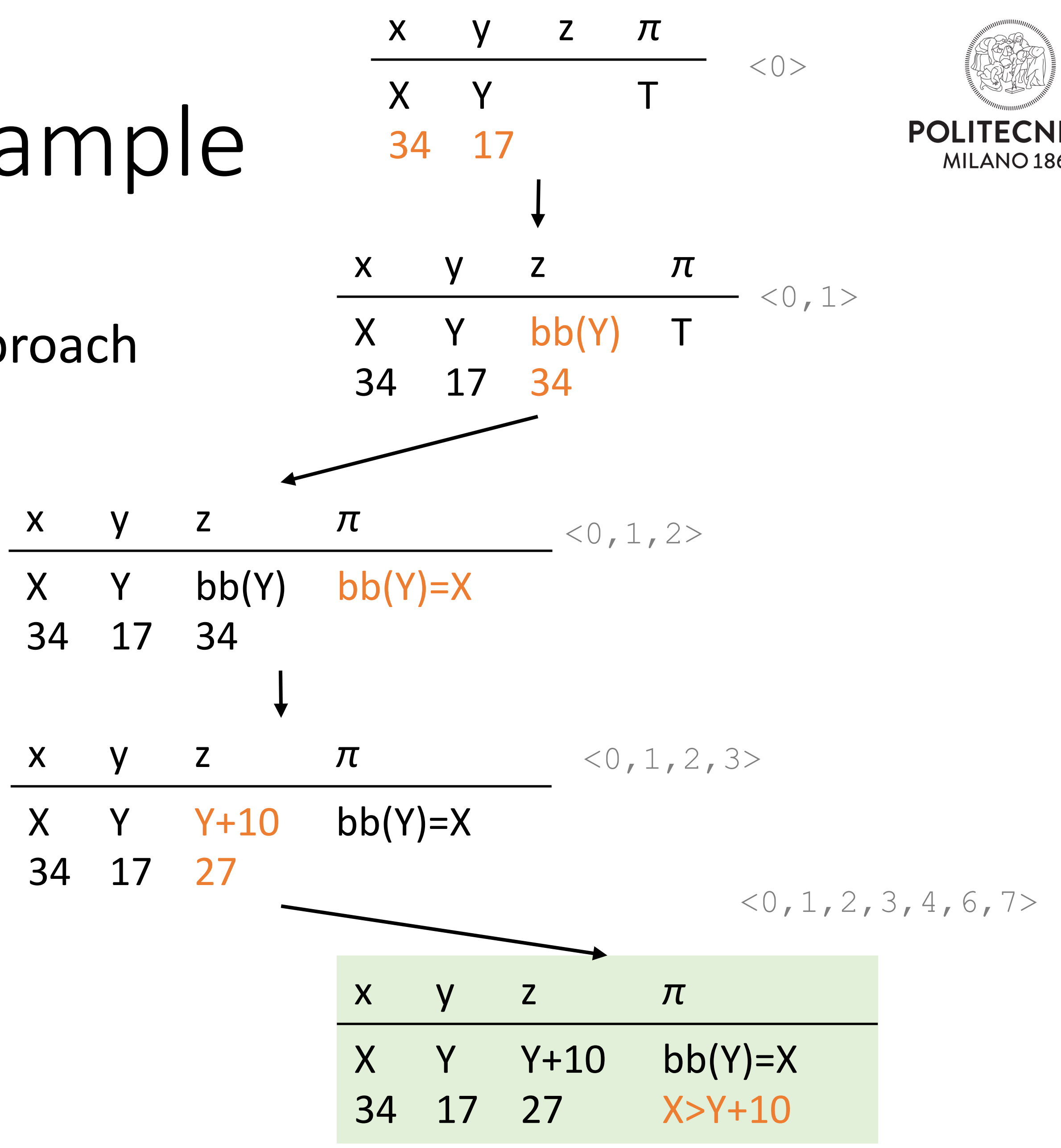
```
0: void m2(int x, int y) {
1:   int z := bb(y) //black-box function
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print(“Log message.”)
6:   }
7: }
```

Concretize: $\boxed{bb(Y)}$=X ∧ X>Y+10
We select Y randomly, execute bb(Y) and check if the formula holds
Example: Y=17, bb(Y)=34
Solve: Y=17 ∧ bb(Y)=34 ∧ bb(Y) =X ∧ X>Y+10

Concolic execution: example

- New explorations follows the same approach

```
0: void m2(int x, int y) {
1:   int z := bb(y) //black-box function
2:   if (z == x) {
3:     z := y + 10
4:     if (x <= z)
5:       print("Log message.")
6:   }
7: }
```



<0>

x	y	z	π
X	Y		T
34	17		

<0,1>

x	y	z	π
X	Y	bb(Y)	T
34	17	34	

<0,1,2>

x	y	z	π
X	Y	bb(Y)	bb(Y)=X
34	17	34	

<0,1,2,3>

x	y	z	π
X	Y	Y+10	bb(Y)=X
34	17	27	

<0,1,2,3,4,6,7>

x	y	z	π
X	Y	Y+10	bb(Y)=X
34	17	27	X>Y+10

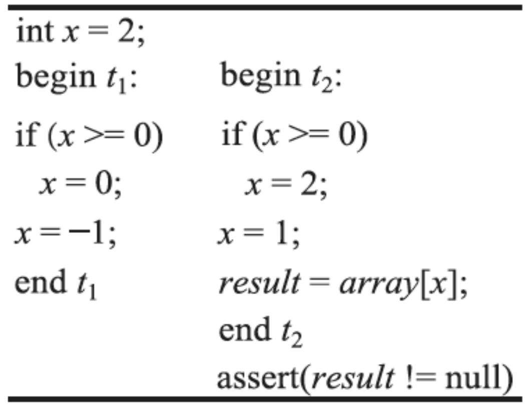
Concolic execution: pros and cons

- Advantages
  - Can deal with black-box functions in path conditions (not possible with symbolic exec)
  - Can generate concrete test cases automatically, according to some code coverage criterion
- Limitations
  - Finds just **one input example** per path, however…
    - Faults typically occur with certain inputs only
    - If faults are rare events, it's unlikely to spot them with concolic exec
  - #paths explode due to complex nested conditions → **large search space**
  - **Does not guide the exploration**, it just explores possible paths one by one as long as we have budget (e.g., time, #runs)

Testing concurrent software – main difficulties

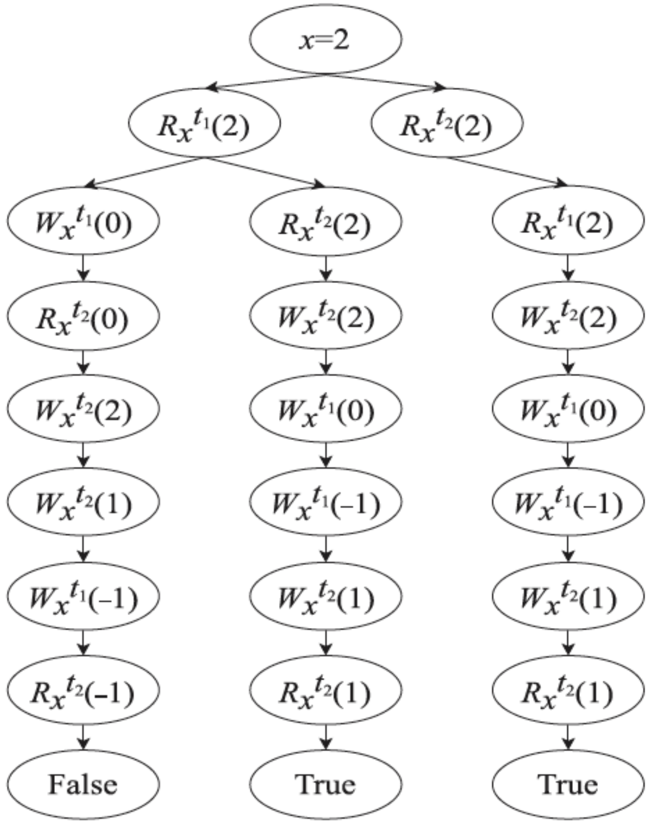
- Concurrency faults are non-deterministic
- They can manifest only within specific *interleavings*
- *Interleavings* depend on execution conditions not under the direct control of the program

Different interleavings



```
int x = 2;
begin t1:          begin t2:
if (x >= 0)        if (x >= 0)
   x = 0;             x = 2;
x = −1;            x = 1;
end t1             result = array[x];
                   end t2
                   assert(result != null)
```

(a) A concurrent program p_{array}



$$\begin{tikzcd}
 & x=2 \arrow[dl] \arrow[dr] & \\
R_x^{t_1}(2) \arrow[d] \arrow[dr] & & R_x^{t_2}(2) \arrow[d] \\
W_x^{t_1}(0) \arrow[d] & R_x^{t_2}(2) \arrow[d] & R_x^{t_1}(2) \arrow[d] \\
R_x^{t_2}(0) \arrow[d] & W_x^{t_2}(2) \arrow[d] & W_x^{t_2}(2) \arrow[d] \\
W_x^{t_2}(2) \arrow[d] & W_x^{t_1}(0) \arrow[d] & W_x^{t_1}(0) \arrow[d] \\
W_x^{t_2}(1) \arrow[d] & W_x^{t_1}(-1) \arrow[d] & W_x^{t_1}(-1) \arrow[d] \\
W_x^{t_1}(-1) \arrow[d] & W_x^{t_2}(1) \arrow[d] & W_x^{t_2}(1) \arrow[d] \\
R_x^{t_2}(-1) \arrow[d] & R_x^{t_2}(1) \arrow[d] & R_x^{t_2}(1) \arrow[d] \\
\text{False} & \text{True} & \text{True}
\end{tikzcd}$$

(b) Part of possible execution behaviors

Sun et al. An Interleaving Guided Metamorphic Testing Approach for Concurrent Programs. ACM Transactions on Software Engineering and Methodology, Vol. 33, No. 1, Article 8. Pub. date: November 2023

Testing concurrent software – focus

- Discover problems related to
  - Data races

Listing 2. An example of data race

```
begin f_1                begin f_2
x='AAAAAAAA'             x='BBBBBBBB'
end f_1                  end f_2
```

  - Atomicity violations

Listing 3. An example of atomicity violation

```
begin f_1           begin f_2
if (x>0)            x = 0
 x = x-1            end f_2
end f_1
```

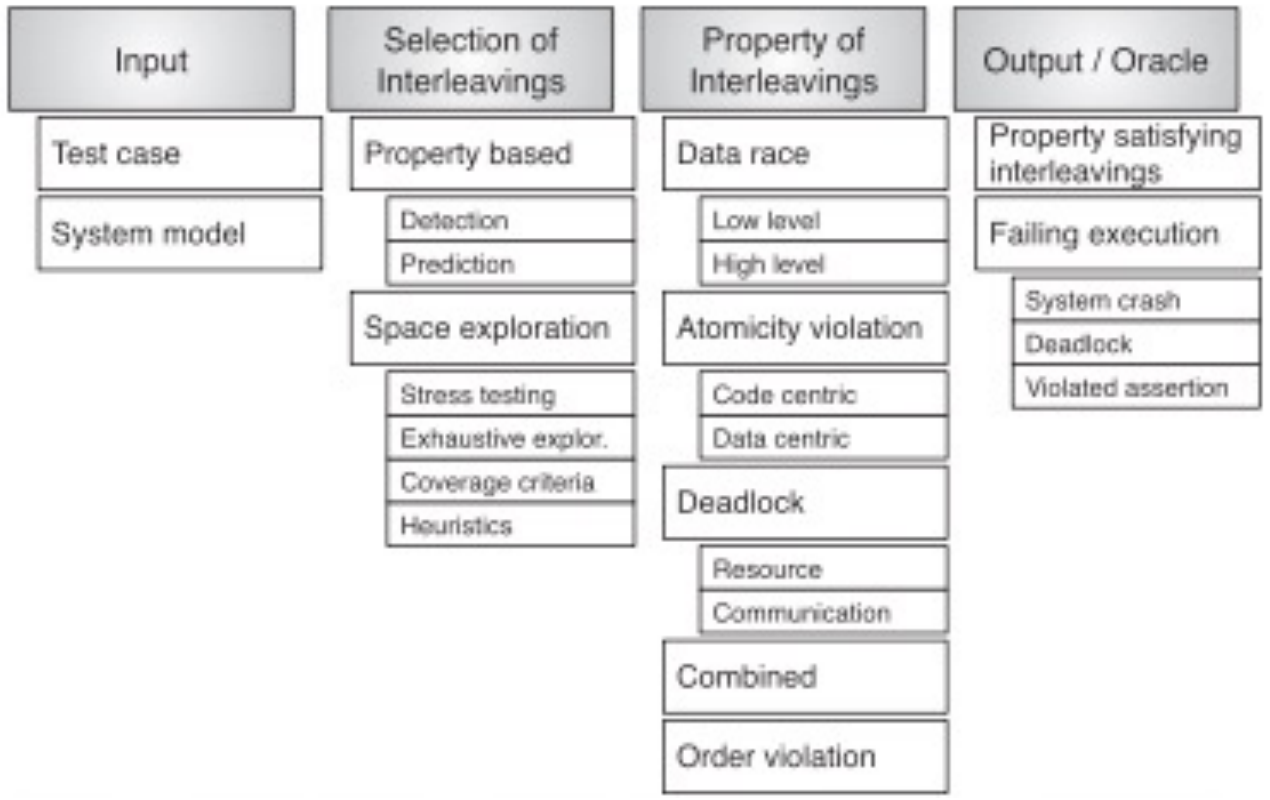
  - Deadlocks

Listing 4. An example of deadlock

```
begin f_1             begin f_2
acquire(L1)           acquire(L2)
  acquire(L2)           acquire(L1)
  ...                   ...
  release(L2)           release(L1)
release(L1)           release(L2)
end f_1               end f_2
```

Bianchi et al. A Survey of Recent Trends in Testing Concurrent Software Systems. IEEE Transactions on Software Engineering, vol 44, no. 8, August 2018

Some of the dimensions for classifying the available techniques



Bianchi et al. A Survey of Recent Trends in Testing Concurrent Software Systems. IEEE Transactions on Software Engineering, vol 44, no. 8, August 2018

Metamorphic testing

- Technique applicable to any kind of software and recently adapted to discover data race issues in concurrent software
- Basic idea of general metamorphic testing: rely on the knowledge of *Metamorphic Relations* (MRs) holding between multiple program inputs and corresponding outputs to derive
  - New test cases from existing ones
  - An oracle for the program

Metamorphic testing - example

- Consider a program P computing the shortest path in an undirected graph G
- We may express the following properties (MRs)
  - MR1: The length of the shortest path between two points in the graph does not vary if we swap the two points |P(G, a, b)| = |P(G, b, a)|
  - MR2: If c belongs to the shortest path between a and b: |P(G, a, c)| + |P(G, c, b)| = |P(G, a, b)|
- Now we can
  - Pick a *source test case* <a1, b1> selected with any technique
  - Run the software and obtain the result, that is, the shortest path and its length
  - Based on MR1, generate a *follow up test case* <b1, a1>
  - Run the software with the new test case
  - Check if MR holds: |P(G, a1, b1)| = |P(G, b1, a1)|. If not, then P is failing
- So, we have got a technique for generating test cases and an oracle!

Chen et al. Metamorphic Testing: A Review of Challenges and Opportunities. ACM Computing Surveys, Vol. 51, No. 1, Article 4. Publication date: January 2018

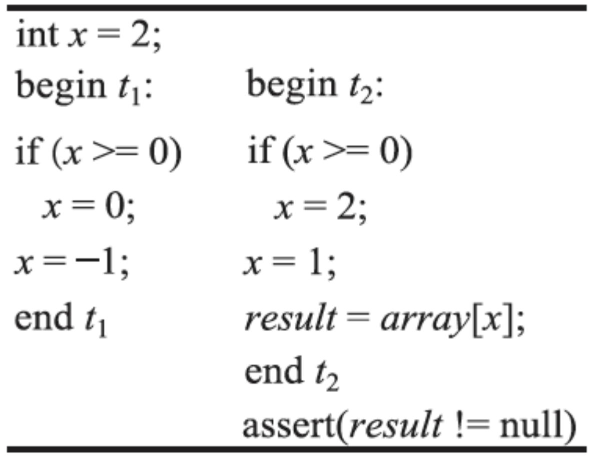
Metamorphic testing for concurrent programs

- Test case for concurrent programs: *C = (X, S)*
  - X set of values of the input parameters
  - S a series of read and write events concerning the variable shared between different threads/processes
- MRs are relations between multiple program inputs+series of read/write events and corresponding outputs

Sun et al. An Interleaving Guided Metamorphic Testing Approach for Concurrent Programs. ACM Transactions on Software Engineering and Methodology, Vol. 33, No. 1, Article 8. Pub. date: November 2023

Metamorphic testing for concurrent programs

- Example of a test case for p_{array}: $C = (\{2\}, <R_x^{t1}(2), W_x^{t1}(0), R_x^{t2}(0), W_x^{t2}(2), W_x^{t1}(-1), W_x^{t2}(1), R_x^{t2}(1)>\}$
- Let's assume this is the source test case
- Intuitively, our MR is that the output remains the same for the same inputs and different R/W sequences
- We define follow up test cases based on MR
- We execute the concurrent program making sure the selected R/W sequences occur
- If the output changes, the test is unsuccessful



```
int x = 2;
begin t1:          begin t2:
if (x >= 0)        if (x >= 0)
   x = 0;             x = 2;
x = −1;            x = 1;
end t1             result = array[x];
                   end t2
                   assert(result != null)
```

(a) A concurrent program p_{array}

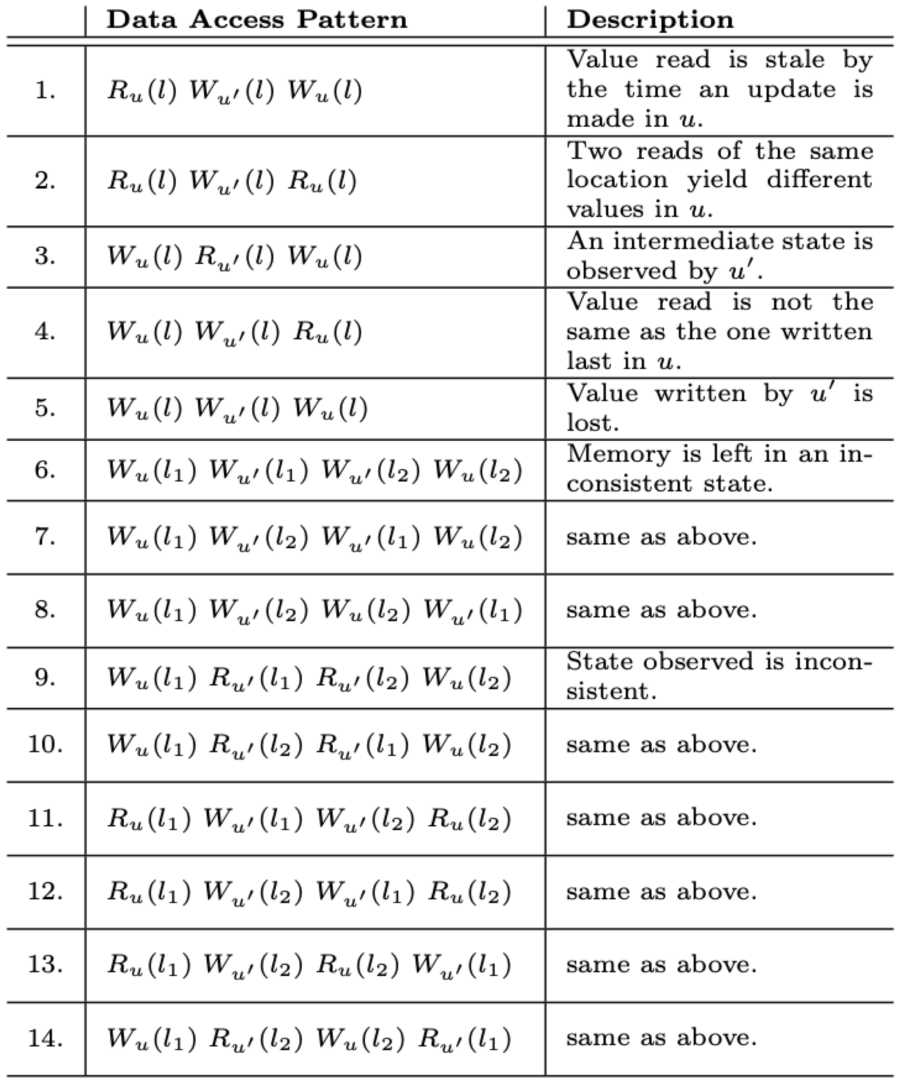
Sun et al. An Interleaving Guided Metamorphic Testing Approach for Concurrent Programs. ACM Transactions on Software Engineering and Methodology, Vol. 33, No. 1, Article 8. Pub. date: November 2023

Questions

- How to select the R/W sequences for follow up test cases?
- How to control that the program executes following the selected interleavings?

How to select the follow up interleavings?

- We can select those that may raise specific problems, for instance

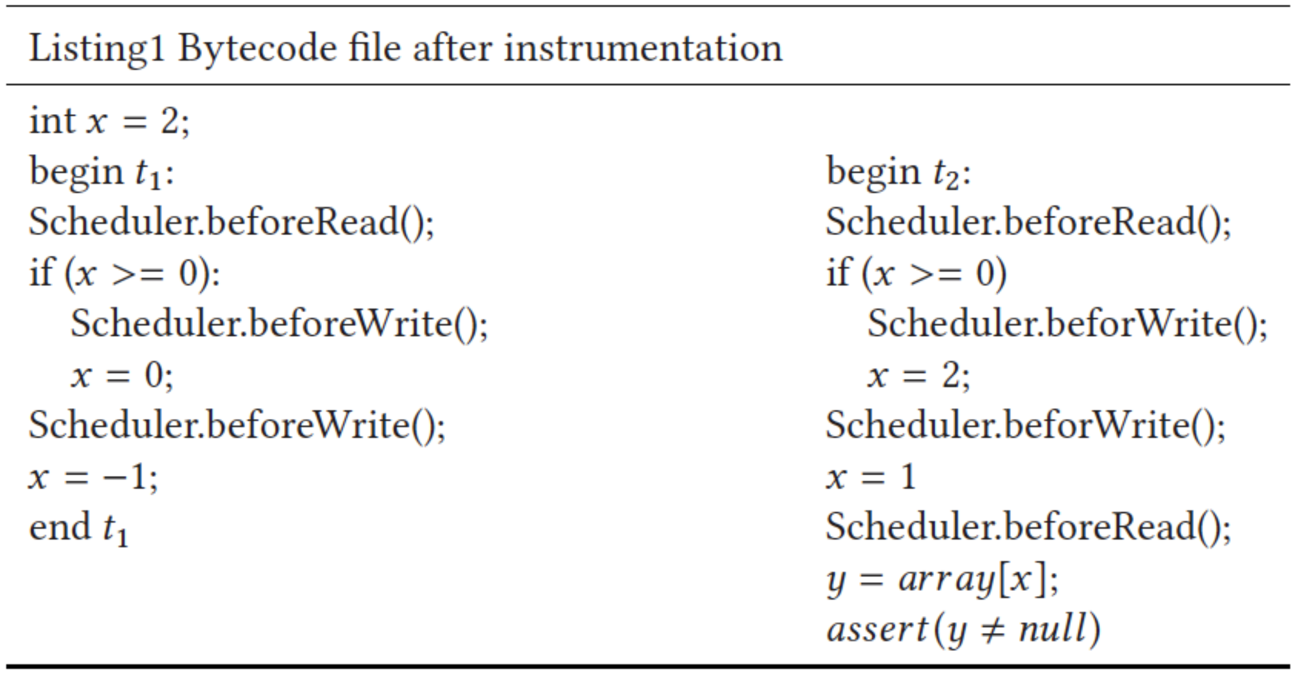


	Data Access Pattern	Description
1.	$R_u(l)\ W_{u'}(l)\ W_u(l)$	Value read is stale by the time an update is made in u.
2.	$R_u(l)\ W_{u'}(l)\ R_u(l)$	Two reads of the same location yield different values in u.
3.	$W_u(l)\ R_{u'}(l)\ W_u(l)$	An intermediate state is observed by u'.
4.	$W_u(l)\ W_{u'}(l)\ R_u(l)$	Value read is not the same as the one written last in u.
5.	$W_u(l)\ W_{u'}(l)\ W_u(l)$	Value written by u' is lost.
6.	$W_u(l_1)\ W_{u'}(l_1)\ W_{u'}(l_2)\ W_u(l_2)$	Memory is left in an inconsistent state.
7.	$W_u(l_1)\ W_{u'}(l_2)\ W_{u'}(l_1)\ W_u(l_2)$	same as above.
8.	$W_u(l_1)\ W_{u'}(l_2)\ W_u(l_2)\ W_{u'}(l_1)$	same as above.
9.	$W_u(l_1)\ R_{u'}(l_1)\ R_{u'}(l_2)\ W_u(l_2)$	State observed is inconsistent.
10.	$W_u(l_1)\ R_{u'}(l_2)\ R_{u'}(l_1)\ W_u(l_2)$	same as above.
11.	$R_u(l_1)\ W_{u'}(l_1)\ W_{u'}(l_2)\ R_u(l_2)$	same as above.
12.	$R_u(l_1)\ W_{u'}(l_2)\ W_{u'}(l_1)\ R_u(l_2)$	same as above.
13.	$R_u(l_1)\ W_{u'}(l_2)\ R_u(l_2)\ W_{u'}(l_1)$	same as above.
14.	$W_u(l_1)\ R_{u'}(l_2)\ W_u(l_2)\ R_{u'}(l_1)$	same as above.

Hammer et al. 2008. Dynamic detection of atomic-set-serializability violations. In Proceedings of the 30th International Conference on Software Engineering (ICSE'08). 231–240.

How to control that the program follows the selected interleavings?

- Instrumenting the bytecode



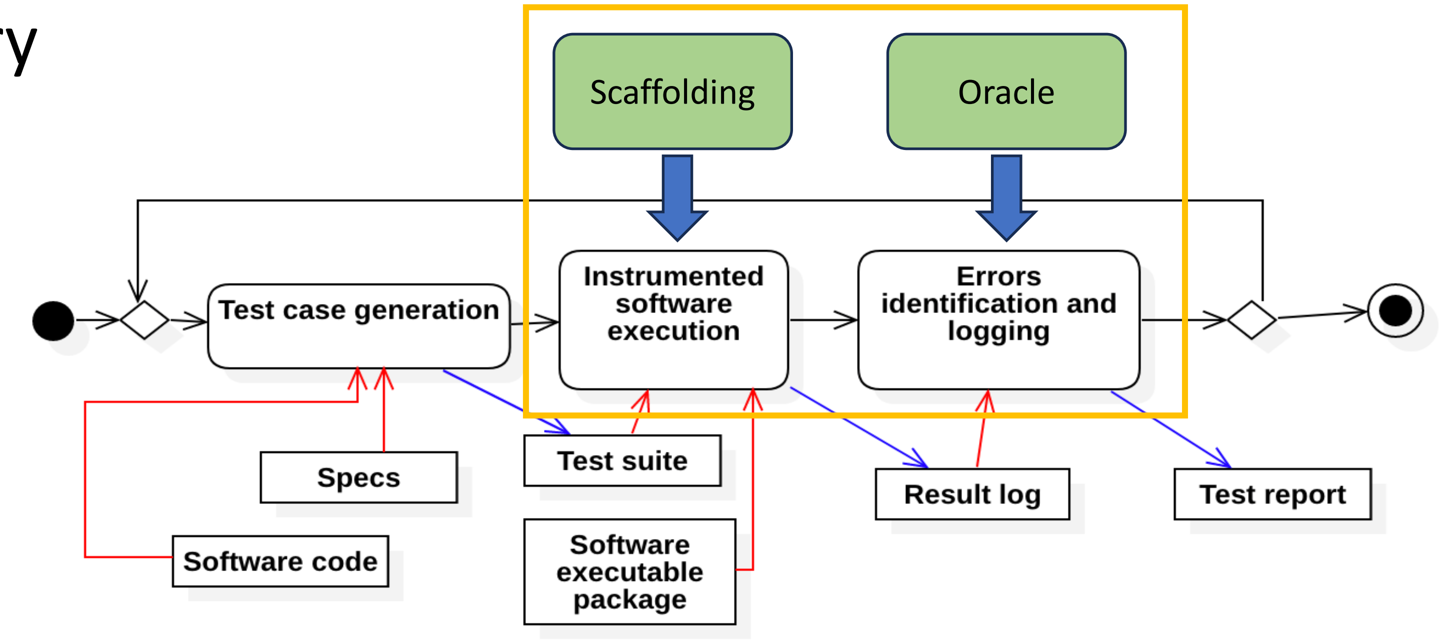
Listing1 Bytecode file after instrumentation

```
int $x = 2$;
begin $t_1$:                          begin $t_2$:
Scheduler.beforeRead();               Scheduler.beforeRead();
if ($x >= 0$):                        if ($x >= 0$)
    Scheduler.beforeWrite();              Scheduler.beforWrite();
    $x = 0$;                              $x = 2$;
Scheduler.beforeWrite();              Scheduler.beforWrite();
$x = -1$;                             $x = 1$
end $t_1$                             Scheduler.beforeRead();
                                      $y = array[x]$;
                                      $assert(y \neq null)$
```

Sun et al. An Interleaving Guided Metamorphic Testing Approach for Concurrent Programs. ACM Transactions on Software Engineering and Methodology, Vol. 33, No. 1, Article 8. Pub. date: November 2023

Unit testing frameworks

- Language-specific
- Offer a way to develop drivers, stubs and oracles
- Come in the form of a library


Scaffolding
Oracle
Test case generation
Instrumented software execution
Errors identification and logging
Specs
Software code
Test suite
Software executable package
Result log
Test report

Elements of unit testing frameworks

- **Test runner:**
  - This is a component which orchestrates the execution of tests and provides the outcome to the user.
  - The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.
- **Test case:**
  - In most cases it is a class from which our application-specific code inherits
- **Test fixture:**
  - This represents the preparation needed to
    - set the initial state needed for a test case before the test
    - return back to the original state after the test

Elements of unit testing frameworks

- **Test suite:**
  - It is a collection of test cases that share the same fixture.
- **Assertions**:
  - Functions/macros that check the state or the output of the system under test (oracles)

JUnit: and example of class to test

```
package moneyExample;
public class Money {
          private int fAmount;
          private String fCurrency;
          public Money(int amount, String currency) {
                  fAmount = amount;
                  fCurrency = currency;  }
          public int amount() { … }

          public String currency() { … }

          public Money add(Money m) {… }

          public boolean equals(Object anObject) { … }
```

MoneyTest (1)

```
package junit;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import moneyExample.Money;
public class MoneyTest {
 private Money money1EUR;
 private Money money2EUR;
 private Money money1$;
 @Before
 public void setUp() throws Exception {
  money1EUR = new Money(100, "EUR");
  money2EUR = new Money(1000, "EUR");
  money1$ = new Money(111, "$");
 }
```

MoneyTest (2)

```
@Test
public void testAmount() {
 assertEquals(100, money1EUR.amount());
 assertEquals(1000, money2EUR.amount());
 assertEquals(111, money1$.amount());
}
@Test
public void testCurrency() {
 assertEquals("EUR", money1EUR.currency());
 assertEquals("EUR", money2EUR.currency());
 assertEquals("$", money1$.currency());
}
```

MoneyTest (3)

```
@Test
public void testEquals() {
 assertFalse("money1EUR != money2EUR",
                  money1EUR.equals(money2EUR));
 assertFalse("money1EUR != money1$", money1EUR.equals(money1$));
 assertFalse("money1$ != money2EUR", money1$.equals(money2EUR));
 assertTrue("money1EUR is the expected obj",
                  money1EUR.equals(new Money(100, "EUR")));
 assertTrue("money1$ is the expected obj",
                  money1$.equals(new Money(111, "$")));
}
```

MoneyTest (4)

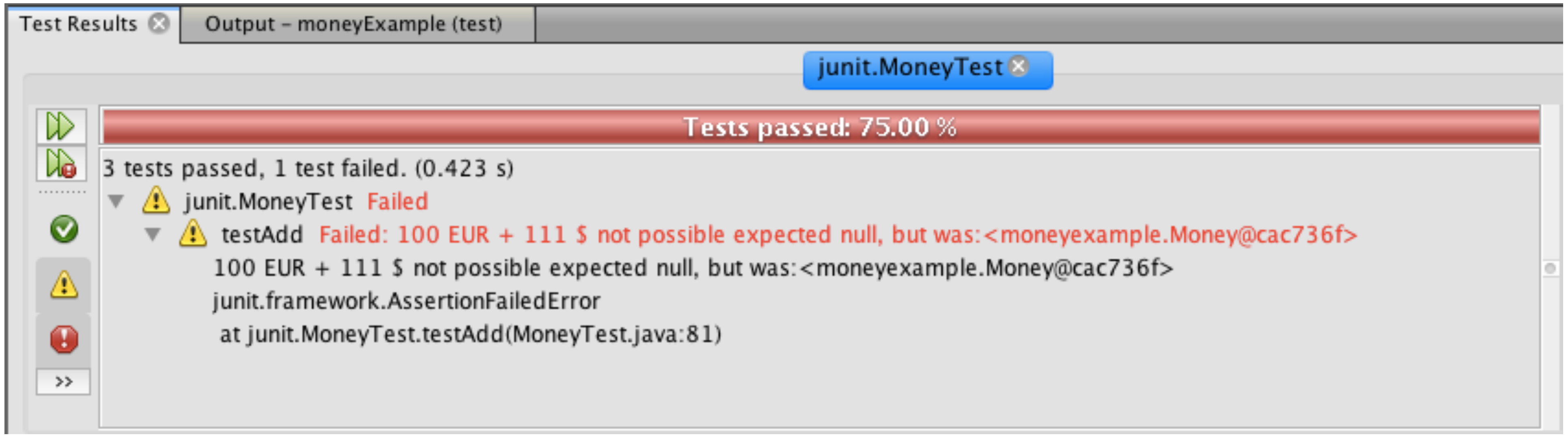
```
 @Test
 public void testAdd() {
  assertTrue("100 EUR + 1000 EUR = 1100 EUR",
   money1EUR.add(money2EUR).equals(new Money(1100, "EUR")));
  assertNull("100 EUR + 111 $ not possible",
   money1EUR.add(money1$));
 }
}
```

A possible implementation of Money

```java
package moneyExample;
public class Money {
 private int fAmount;
 private String fCurrency;
 public Money(int amount, String currency) {
  fAmount = amount;
  fCurrency = currency;  }
 public int amount() { return fAmount; }
 public String currency() { return fCurrency; }
```

```java
public Money add(Money m) {
 return new Money(amount()+m.amount(), currency());
}
public boolean equals(Object anObject) {
 if (anObject instanceof Money) {
  Money aMoney= (Money)anObject;
  return aMoney.currency().equals(currency()) &&
           amount() == aMoney.amount();
}
 return false;
}
}
```

Test execution result



```
Tests passed: 75.00 %
3 tests passed, 1 test failed. (0.423 s)
▼ junit.MoneyTest Failed
  ▼ testAdd Failed: 100 EUR + 111 $ not possible expected null, but was:<moneyexample.Money@cac736f>
      100 EUR + 111 $ not possible expected null, but was:<moneyexample.Money@cac736f>
      junit.framework.AssertionFailedError
       at junit.MoneyTest.testAdd(MoneyTest.java:81)
```

```
@Test
public void testAdd() {
 assertTrue("100 EUR + 1000 EUR = 1100 EUR",
        money1EUR.add(money2EUR).equals(new Money(1100, "EUR")));
 assertNull("100 EUR + 111 $ not possible", money1EUR.add(money1$));
}
```

Unit testing frameworks for C++

- Google Test
- CppUnit
- CxxTest
- Microsoft Unit Testing Framework for C++

Example from GoogleTest Primer

https://google.github.io/googletest/primer.html

```
template <typename E> // E is the element type.
class Queue {
 public:
  Queue();
  void Enqueue(const E& element);
  E* Dequeue(); // Returns NULL if the queue is empty.
  size_t size() const;
  ...
};
```

Fixture class

```
class QueueTest : public ::testing::Test {
 protected:
  void SetUp() override {
   // q0_ remains empty
   q1_.Enqueue(1);
   q2_.Enqueue(2);
   q2_.Enqueue(3);
  }
  // void TearDown() override {}
  Queue<int> q0_;
  Queue<int> q1_;
  Queue<int> q2_;
 };
```

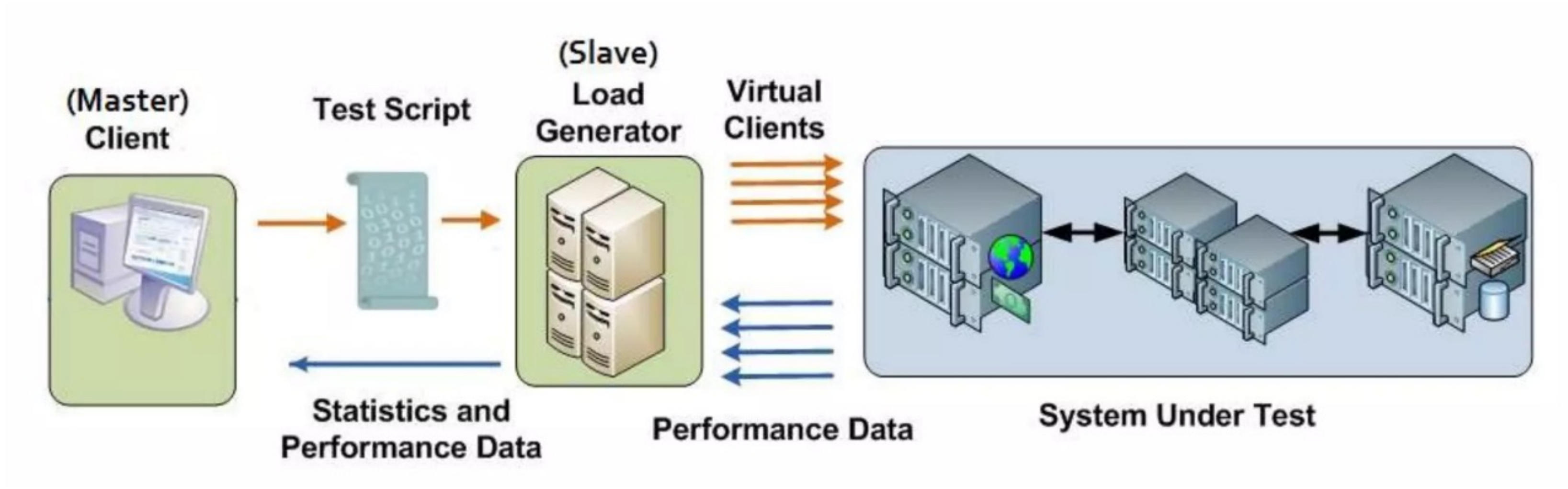
Tests

```
TEST_F(QueueTest, IsEmptyInitially) {
  EXPECT_EQ(q0_.size(), 0);
}
TEST_F(QueueTest, DequeueWorks) {
  int* n = q0_.Dequeue();
  EXPECT_EQ(n, nullptr);

  n = q1_.Dequeue();
  ASSERT_NE(n, nullptr);
  EXPECT_EQ(*n, 1);
  EXPECT_EQ(q1_.size(), 0);
  delete n;

  n = q2_.Dequeue();
  ASSERT_NE(n, nullptr);
  EXPECT_EQ(*n, 2);
  EXPECT_EQ(q2_.size(), 1);
  delete n;
}
```

Instrumented software execution for load and performance testing - JMeter

- Open source Apache Foundation project https://jmeter.apache.org/
- Supports load and performance testing for various protocols
  - Http, https, FTP, JDBC, JMS, LDAP, SOAP
- Extensible through plugins

Jmeter test architecture


(Master) Client
Test Script
(Slave) Load Generator
Virtual Clients
Statistics and Performance Data
Performance Data
System Under Test