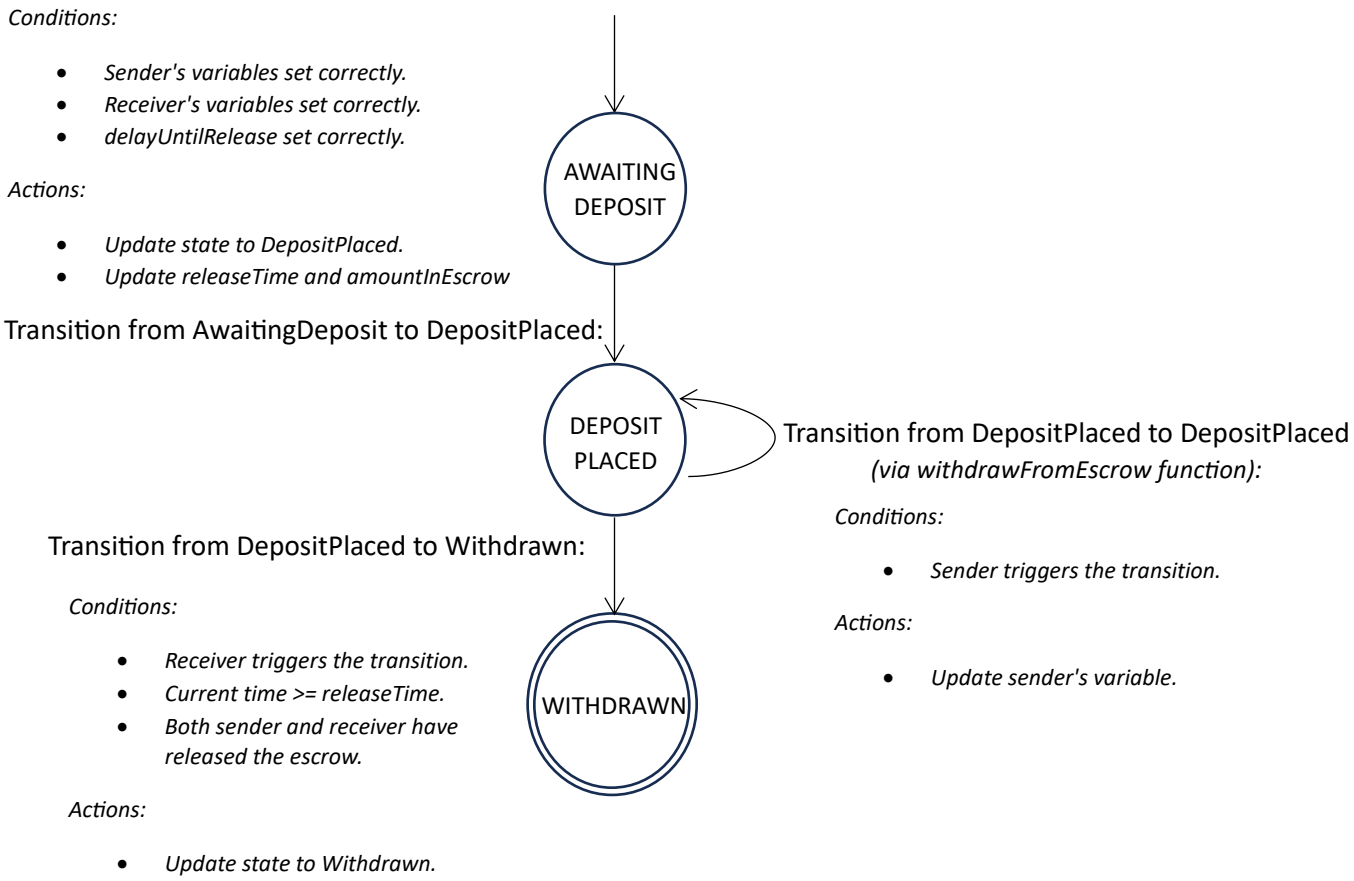
Question 1: Testing

Escrow Smart Contract:

The escrow smart contract employs a state-based framework consisting of three distinct states. The initial state, "AwaitingDeposit," involves parameter setup, including defining sender and receiver addresses, and specifying a delay until release. Upon successful fund placement by the sender, the contract transitions to the "DepositPlaced" state. Here, the funds are held until a predetermined release time. Finally, the contract advances to the "Withdrawn" state when the receiver withdraws the funds after the release time has elapsed. This state management ensures controlled and secure progression throughout the escrow process, providing a reliable mechanism for conditional fund transfers while protecting the interests of both parties.



Unit test descriptions for each of the states and transitions of the smart contract:

Unit 1: Transition into 'AwaitingDeposit'

a. Test that the 'sender' variable is set correctly.
b. Test that the 'receiver' variable is set correctly.
c. Test that the 'delayUntilRelease' variable is set correctly.
d. Test that the contract state is updated to 'AwaitingDeposit'.
e. Test that no transition to the 'Withdrawn' state is possible when the contract is in the 'AwaitingDeposit' state.

```
const Escrow1 = artifacts.require('Escrow1');
contract('TestEscrow1', function (accounts) {
    let escrowInstance;
    const sender = accounts[0];
    const receiver = accounts[1];
    const delay = 3600; // 1 hour
    beforeEach(async function () {
        escrowInstance = await Escrow1.new(sender, receiver, delay);});
    it('should transition to AwaitingDeposit state', async function () {
        // Test that the 'sender' variable is set correctly
        const contractSender = await escrowInstance.sender();
        assert.equal(contractSender, sender, "Sender should be set
correctly");
        // Test that the 'receiver' variable is set correctly
        const contractReceiver = await escrowInstance.receiver();
        assert.equal(contractReceiver, receiver, "Receiver should be set
correctly");
        // Test that the 'delayUntilRelease' variable is set correctly
        const contractDelay = await escrowInstance.delayUntilRelease();
        assert.equal(contractDelay, delay, "Delay should be set correctly");
        // Perform transition to AwaitingDeposit state
        await escrowInstance.transitionToAwaitingDeposit();
        // Test that the contract state is updated to AwaitingDeposit
        const currentState = await escrowInstance.state();
        assert.equal(currentState, 0, "Should be in AwaitingDeposit state");
        // Ensure no transition to the Withdrawn state from AwaitingDeposit
        try {
            await escrowInstance.transitionToWithdrawn();
            assert.fail("Transition to Withdrawn state should fail from
AwaitingDeposit state");
        } catch (error) {
            assert(error.message.includes("revert"), "Expected revert error");
        }
    });
});
```

Unit 2.1: Transition into 'DepositPlaced' via 'placeInEscrow'

a. Test that only the sender can transition the contract to the 'DepositPlaced' state.
b. Test that the transaction with a value greater than 0 triggers the transition.
c. Test that the contract state is updated to 'DepositPlaced'.
d. Test that the 'releaseTime' variable is updated accordingly.
e. Test that the 'amountInEscrow' variable is updated accordingly.
f. Test that no transition to 'DepositPlaced' is possible via the 'placeInEscrow' transition when the contract is already in the 'DepositPlaced' state.

```
const Escrow1 = artifacts.require('Escrow1');
contract('TestEscrow1', function (accounts) {
    let escrowInstance;
    const sender = accounts[0];
    const receiver = accounts[1];
    const delay = 3600; // 1 hour
    beforeEach(async function () {
        escrowInstance = await Escrow1.new(sender, receiver, delay);});
    it('should transition to DepositPlaced state via placeInEscrow', async
function () {
        // Set up sender
        await escrowInstance.setSender(sender);
        // Perform transition to DepositPlaced state via placeInEscrow
        await escrowInstance.transitionToDepositPlaced(1); // Simulate value
of 1 ether
        // Test that only the sender can transition to the DepositPlaced state
        const currentState = await escrowInstance.state();
        assert.equal(currentState, 1, "Should be in DepositPlaced state");
        // Test that the 'releaseTime' variable is updated accordingly
        const releaseTime = await escrowInstance.releaseTime();
        const currentTime = Math.floor(Date.now() / 1000);
        assert.isAtLeast(releaseTime.toNumber(), currentTime + delay,
"releaseTime should be updated accordingly");
        // Test that the 'amountInEscrow' variable is updated accordingly
        const amountInEscrow = await escrowInstance.amountInEscrow();
        assert.equal(amountInEscrow.toNumber(), web3.utils.toWei("1",
"ether"), "amountInEscrow should be updated accordingly");
        // Ensure no transition to DepositPlaced via placeInEscrow when
already in DepositPlaced state
        try {
            await escrowInstance.transitionToDepositPlaced(1); // Simulate
value of 1 ether
            assert.fail("Transition to DepositPlaced state should fail when
already in DepositPlaced state");
        } catch (error) {
            assert(error.message.includes("revert"), "Expected revert error");
        }
    });
});
```

Unit 2.2: Transition into 'DepositPlaced' via 'withdrawFromEscrow'

a. Test that the contract state remains unchanged when the sender calls this transition.
b. Test that if the sender calls this transition, the 'sender' variable is updated to true.
c. Test that if the sender calls this transition again, the value of the 'sender' variable remains true.

```
const Escrow1 = artifacts.require('Escrow1');
contract('TestEscrow1', function (accounts) {
    let escrowInstance;
    const sender = accounts[0];
    const receiver = accounts[1];
    const delay = 3600; // 1 hour
    beforeEach(async function () {
        escrowInstance = await Escrow1.new(sender, receiver, delay);
    });
    it('should transition to DepositPlaced state via withdrawFromEscrow',
async function () {
        // Set up sender
        await escrowInstance.setSender(sender);
        // Perform transition to DepositPlaced state via withdrawFromEscrow
        await escrowInstance.transitionToDepositPlacedViaWithdraw();
        // Test that the contract state remains unchanged
        const currentState = await escrowInstance.state();
        assert.equal(currentState, 0, "State should remain unchanged");
        // Test that if the sender calls this transition, the 'sender'
variable is updated to true
        const senderVariable = await escrowInstance.sender();
        assert.equal(senderVariable, true, "'sender' variable should be
updated to true");
        // Test that if the sender calls this transition again, the value of
the 'sender' variable remains true
        await escrowInstance.transitionToDepositPlacedViaWithdraw(); // Call
again
        const updatedSenderVariable = await escrowInstance.sender();
        assert.equal(updatedSenderVariable, true, "'sender' variable should
remain true");
    });
});
```

Unit 3: Transition into 'Withdrawn'

a. Test that only the receiver can transition the contract to the 'Withdrawn' state.
b. Test that the current time needs to be greater than or equal to the release time for the transition to occur.
c. Test that both the sender and the receiver need to have released the escrow for the transition to occur.
d. Test that the contract state is updated to 'Withdrawn'.
e. Test that no transition to 'DepositPlaced' is possible via the 'placeInEscrow' transition when the contract is in the 'Withdrawn' state.
f. Test that no transition to 'DepositPlaced' is possible via the 'withdrawFromEscrow' transition when the contract is in the 'Withdrawn' state.
g. Test that no transition to 'Withdrawn' is possible once the contract is already in the 'Withdrawn' state.

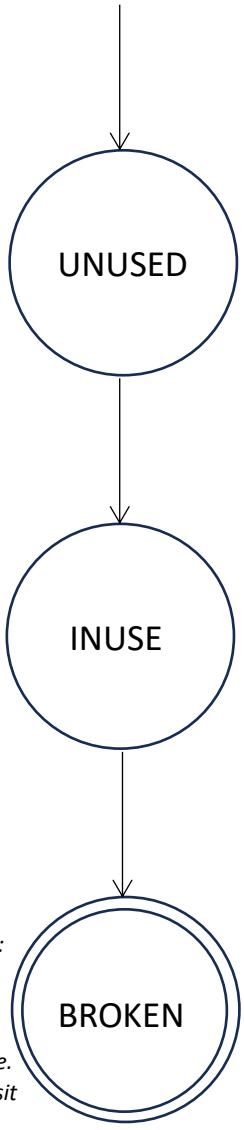
```
const Escrow1 = artifacts.require('Escrow1');
contract('TestEscrow1', function (accounts) {
    let escrowInstance;
    const sender = accounts[0];
    const receiver = accounts[1];
    const delay = 3600; // 1 hour
    beforeEach(async function () {
        escrowInstance = await Escrow1.new(sender, receiver, delay); });
    it('should transition to Withdrawn state', async function () {
        // Set up sender and receiver
        await escrowInstance.setSender(sender);
        await escrowInstance.setReceiver(receiver);
        await escrowInstance.setDelayUntilRelease(delay);
        // Place escrow
        await escrowInstance.transitionToDepositPlaced(1);
        // Simulate value of 1 ether
        // Set release time to a past time
        await escrowInstance.setReleaseTime(Math.floor(Date.now()/1000)-3600);
        // Perform transition to Withdrawn state
        await escrowInstance.transitionToWithdrawn();
        // Test that only the receiver can transition to the Withdrawn state
        try {
            await escrowInstance.transitionToWithdrawn({ from: sender });
            assert.fail("Transition to Withdrawn state should fail when called
by sender");
        } catch (error) {
            assert(error.message.includes("revert"), "Expected revert error");
        }
```

```
        // Test that the current time needs to be greater than or equal to the
release time
        try {
            await escrowInstance.transitionToWithdrawn();
            assert.fail("Transition to Withdrawn state should fail when
release time not reached");
        } catch (error) {
            assert(error.message.includes("revert"), "Expected revert error");
        }
        // Test that both sender and receiver need to have released the escrow
        try {
            await escrowInstance.transitionToWithdrawn({ from: receiver });
            assert.fail("Transition to Withdrawn state should fail when sender
has not released");
        } catch (error) {
            assert(error.message.includes("revert"), "Expected revert error");
        }
        // Release escrow from sender
        await escrowInstance.transitionToDepositPlacedViaWithdraw();
        // Perform transition to Withdrawn state
        await escrowInstance.transitionToWithdrawn({ from: receiver });
        // Test that the contract state is updated to Withdrawn
        const currentState = await escrowInstance.state();
        assert.equal(currentState, 2, "Should be in Withdrawn state");
        // Test that no transition to DepositPlaced is possible when in
Withdrawn state
        try {
            await escrowInstance.transitionToDepositPlaced(1); // Simulate
value of 1 ether
            assert.fail("Transition to DepositPlaced state should fail when in
Withdrawn state");
        } catch (error) {
            assert(error.message.includes("revert"), "Expected revert error");
        }
        // Test that no transition to DepositPlaced via withdrawFromEscrow is
possible when in Withdrawn state
        try {
            await escrowInstance.transitionToDepositPlacedViaWithdraw();
            assert.fail("Transition to DepositPlaced state should fail when in
Withdrawn state");
        } catch (error) {
            assert(error.message.includes("revert"), "Expected revert error");
        }
```

```
        // Test that no transition to Withdrawn is possible once already in
Withdrawn state
        try {
            await escrowInstance.transitionToWithdrawn({ from: receiver });
            assert.fail("Transition to Withdrawn state should fail when
already in Withdrawn state");
        } catch (error) {
            assert(error.message.includes("revert"), "Expected revert error");
        }
    });
});
```

Piggy Bank Smart Contract:

The PiggyBank smart contract operates in three distinct states: 'Unused', 'InUse', and 'Broken'. In the 'Unused' state, the contract is initialized with the owner's address and a balance of 0. It transitions to the 'InUse' state when the owner adds money, updating the balance and marking the time of the first deposit if transitioning from 'Unused'. After a year of use, the contract can transition to the 'Broken' state, where the owner can retrieve the balance. Once in the 'Broken' state, no further actions or transitions are allowed. Each state enforces specific conditions and actions, ensuring controlled usage and withdrawal of funds in the PiggyBank.



Transition into Unused

Conditions:

- *owner variable should be set correctly.*
- *balance variable should be initialized with a value of 0.*

Actions:

- *Set the owner variable to the provided address.*
- *Initialize the balance variable with 0.*
- *Update the contract state to 'Unused'.*

Transition into InUse

Conditions:

Only the owner can transition the smart contract into the 'InUse' state.

Actions:

If the contract is in the 'Unused' state:

- *Update the contract state to 'InUse'.*
- *Increment the balance by the message value.*
- *Update timeOfFirstDeposit if transitioning from 'Unused' to 'InUse'.*

If the contract is already in the 'InUse' state:

- *If the 'AddMoney' transition is reiterated:*
  - *Do not update the contract state.*
  - *Do not update timeOfFirstDeposit*

Transition into Broken

Conditions:

- *Only the owner can transition the smart contract into the 'Broken' state.*
- *A year must have passed since the time of the first deposit.*

Actions:

If the contract is in the 'InUse' state:

- *Check if a year has passed since timeOfFirstDeposit.*

If the condition is met:

- *Update the contract state to 'Broken'.*
- *Transfer the balance to the owner's address.*

Unit Test1: Transition into 'Unused' State:

Conditions Tested:

- The owner variable should be set correctly.
- The balance variable should be initialized with a value of 0.
- The contract state should be in the 'Unused' state.

Actions Tested:

- No specific actions are tested in this case, as this transition involves initialization.

```
const PiggyBank = artifacts.require("PiggyBank");

contract("PiggyBank", (accounts) => {
  let piggyBank;
  const owner = accounts[0];

  beforeEach(async () => {
    piggyBank = await PiggyBank.new(owner);
  });

  it("should transition into 'Unused'", async () => {
    // Conditions
    const contractOwner = await piggyBank.owner();
    const balance = await piggyBank.balance();
    const state = await piggyBank.state();

    assert.equal(contractOwner, owner, "Owner should be set correctly");
    assert.equal(balance.toNumber(), 0, "Balance should be initialized to 0");
    assert.equal(state.toNumber(), 0, "State should be Unused");

    // Actions
    // No actions required for this test
  });
```

Unit Test2: Transition into 'InUse' State

Conditions Tested:

- The initial state should be 'Unused'.
- The contract owner can transition the smart contract to 'InUse' state.

Actions Tested:

- The owner successfully transitions the contract to 'InUse' state by adding money.
- The contract state updates correctly to 'InUse'.
- The balance is incremented by the value added.
- If transitioning from 'Unused', the timeOfFirstDeposit variable updates.

```
  it("should transition into 'InUse'", async () => {
    // Conditions
    const initialState = await piggyBank.state();
    assert.equal(initialState.toNumber(), 0, "Initial state should be
Unused");

    // Actions
    // Only the owner can transition to 'InUse'
    await piggyBank.addMoney({ value: 10 });
    const newState = await piggyBank.state();
    const newBalance = await piggyBank.balance();

    assert.equal(newState.toNumber(), 1, "State should be InUse");
    assert.equal(newBalance.toNumber(), 10, "Balance should be incremented");
  });
```

Unit Test 3. Transition into 'Broken' State:

Conditions Tested:

- The initial state should be 'Unused'.
- A year has passed since the time of the first deposit.

Actions Tested:

- The owner transitions the contract to 'Broken' state.
- The contract state updates correctly to 'Broken'.
- If transitioning from 'InUse', the contract's balance is transferred to the owner.

```
  it("should transition into 'Broken'", async () => {
    // Conditions
    const initialState = await piggyBank.state();
    assert.equal(initialState.toNumber(), 0, "Initial state should be
Unused");

    // Actions
    await piggyBank.addMoney({ value: 10 });
    await piggyBank.breakPiggyBank();
    const newState = await piggyBank.state();

    assert.equal(newState.toNumber(), 2, "State should be Broken");
  });
});
```