Question 3: Static Verification

Invariants and function specifications for the Escrow Smart Contract.

Invariant 1:

```
/*@ invariant
@ address(this) != sender
@*/
```

Description: This invariant ensures that the address of the contract (`address(this)`) is not equal to the `sender` address. This guarantees that the contract itself cannot be one of the parties involved in the escrow.

Invariant 2:

```
/*@ invariant
@ address(this) != receiver
@*/
```

Description: Similar to the first invariant, this one ensures that the contract's address is not equal to the `receiver` address, preventing the contract from being a party to the escrow.

Invariant 3:

```
/*@ invariant
@ net(address(this)) == amountInEscrow
@*/
```

Description: This invariant checks that the net balance of the contract's address (`net(address(this))`) is equal to the `amountInEscrow`. This ensures that the funds in escrow are correctly accounted for within the contract.

Invariant 4:

```
/*@ invariant
@ net(sender) + net(receiver) == amountInEscrow
@*/
```

Description: This invariant verifies that the net balances of the `sender` and `receiver` addresses, when summed, equal `amountInEscrow`. It ensures that the escrowed funds are accurately allocated between the two parties.

Invariant 5:

```
/*@ invariant
@ releaseTime >= delayUntilRelease
@*/
```

Description: This invariant ensures that `releaseTime` is greater than or equal to `delayUntilRelease`. It guarantees that the release time is correctly set to a future time that respects the specified delay.

Invariant 6:

```
/*@ invariant
@ state == State.AwaitingDeposit => releasedBySender == false && releasedByRe-
ceiver == false
@*/
```

Description: This invariant checks that if the state is `AwaitingDeposit`, then both `releasedBySender` and `releasedByReceiver` are false. It indicates that funds can only be released when the escrow state changes from `AwaitingDeposit`.

Invariant 7:

```
/*@ invariant
@ state == State.Withdrawn => releasedBySender == true && releasedByReceiver
== true
@*/
```

Description: This invariant checks that if the state is `Withdrawn`, then both `releasedBySender` and `releasedByReceiver` are true. This indicates that funds have been successfully released by both the sender and the receiver when the escrow is in the `Withdrawn` state.

Function Specifications

`placeInEscrow` Function Specifications

```
/*@ succeeds_only_if
@ state != State.AwaitingDeposit
@ msg.sender == sender
@ msg.value > 0
@ after_success
@ amountInEscrow == msg.value
@ releaseTime == now + delayUntilRelease
@ state == State.DepositPlaced
@*/
```

Description: These specifications describe the behaviour of the `placeInEscrow` function.

- The function can only succeed if the current state is not `AwaitingDeposit`, ensuring that funds can't be placed in escrow again before the previous transaction is completed.
- It checks that the caller of the function (`msg.sender`) is the same as the `sender` (party initiating the escrow) and that a positive amount of Ether is sent with the transaction.
- After a successful execution, `amountInEscrow` should be equal to the value sent with the function call (`msg.value`), and `releaseTime` should be set to the current block's timestamp plus `delayUntilRelease`. Additionally, the state should transition to `DepositPlaced`.

`withdrawFromEscrow` Function Specifications

```
/*@ succeeds_only_if
@ state != State.DepositPlaced
@ msg.sender == receiver
@ releaseTime <= now
@ releasedByReceiver == true
@ releasedBySender == true
@ after_success
@ state == State.Withdrawn
@ net(owner) == \old(amountInEscrow)
@ amountInEscrow == 0
@*/
```

Description: These specifications describe the behaviour of the `withdrawFromEscrow` function.

- The function can only succeed if the current state is not `DepositPlaced`, ensuring that funds can't be withdrawn before the escrow conditions are met.
- It checks that the caller of the function (`msg.sender`) is the same as the `receiver` (party receiving the escrowed funds), that the `releaseTime` has passed (`releaseTime <= now`), and that both `releasedByReceiver` and `releasedBySender` are true, indicating that both parties have agreed to release the funds.
- After a successful execution, the state should transition to `Withdrawn`, the net balance of the contract owner (`owner`) should be the same as the previous `amountInEscrow` (`\old(amountInEscrow)`), and `amountInEscrow` should be set to 0.

These comprehensive specifications provide a clear and detailed description of the expected behaviour of the `placeInEscrow` and `withdrawFromEscrow` functions. They ensure that the functions can only be called under specific conditions and define the state changes and variable updates that should occur after successful execution. These specifications are essential for verifying the correctness and security of the Escrow Smart Contract.

PiggyBank Smart Contract:

Invariant 1: Owner Address

```
/*@ invariant
  @ address(this) != owner;
  @*/
```

Explanation: This invariant states that the address of the PiggyBank contract (`address(this)`) should not be equal to the owner's address (`owner`). In other words, the contract's address should not be the same as the owner's address. This ensures that the owner cannot accidentally send funds directly to the contract's address.

Invariant 2: Owner's Net Balance Equals Piggy Bank Balance

```
/*@ invariant
  @ net(owner) == balance;
  @*/
```

Explanation: This invariant specifies that the net balance owned by the owner should be equal to the balance stored in the PiggyBank contract. The `net(owner)` expression represents the net balance of the owner, which takes into account any gas costs. This ensures that the contract accurately reflects the owner's actual balance.

`addMoney` Function Specification

```
/*@ succeeds_only_if
  @ state != PiggyBankState.Broken;
  @ msg.sender == owner;
  @
  @ after_success
  @ balance >= \old(balance);
  @ timeOfFirstDeposit >= now;
  @ state == PiggyBankState.InUse;
  @*/
```

Conditions for Successful Execution (`succeeds_only_if`):

1. `state != PiggyBankState.Broken`: The `addMoney` function should only succeed if the piggy bank is not in the "Broken" state. This ensures that funds cannot be added to a broken piggy bank.
2. `msg.sender == owner`: The function should only succeed if the caller (`msg.sender`) is the owner of the piggy bank. This ensures that only the owner can add money to the piggy bank.

Post-conditions (`after_success`):

1. `balance >= \old(balance)`: After a successful execution of the `addMoney` function, the new balance (`balance`) should be greater than or equal to the previous balance (`\old(balance)`). This condition ensures that the balance is increased or remains unchanged after adding money.
2. `timeOfFirstDeposit >= now`: After a successful deposit, the `timeOfFirstDeposit` should be greater than or equal to the current timestamp (`now`). This condition indicates that the `timeOfFirstDeposit` is correctly updated when the first deposit is made.
3. `state == PiggyBankState.InUse`: After a successful deposit, the state of the piggy bank should transition to "InUse." This condition signifies that the piggy bank is actively in use after a deposit.

`breakPiggyBank` Function Specification:

```
/*@ succeeds_only_if
  @ state == PiggyBankState.InUse;
  @ msg.sender == owner;
  @ now >= timeOfFirstDeposit + 365 days;
  @
  @ after_success
  @ net(owner) == \old(balance);
  @ state == PiggyBankState.Broken;
  @*/
```

Conditions for Successful Execution (`succeeds_only_if`):

1. `state == PiggyBankState.InUse`: The `breakPiggyBank` function should only succeed if the piggy bank is in the "InUse" state. This condition ensures that the piggy bank is actively in use before it can be broken.
2. `msg.sender == owner`: The `breakPiggyBank` function should only succeed if the caller (`msg.sender`) is the owner of the piggy bank. This security check ensures that only the owner has the authority to break the piggy bank.
3. `now >= timeOfFirstDeposit + 365 days`: The function should only succeed if the current timestamp (`now`) is greater than or equal to 365 days (1 year) after the `timeOfFirstDeposit`. This condition enforces a waiting period, ensuring that the piggy bank can only be broken after a year has passed since the first deposit.

Post-conditions (`after_success`):

1. `net(owner) == \old(balance)`: After a successful execution of the `breakPiggyBank` function, the net balance owned by the owner should remain unchanged. In other words, the owner's net balance (accounting for any gas costs) should be the same as the previous balance (`\old(balance)`).
2. `state == PiggyBankState.Broken`: After a successful execution, the state of the piggy bank should transition to "Broken." This condition indicates that the piggy bank has been successfully broken, and its state has changed accordingly.

These specifications provide a clear understanding of the expected behavior and conditions for the `addMoney` and `breakPiggyBank` functions in the PiggyBank smart contract, along with invariants that ensure the integrity of the contract's state.