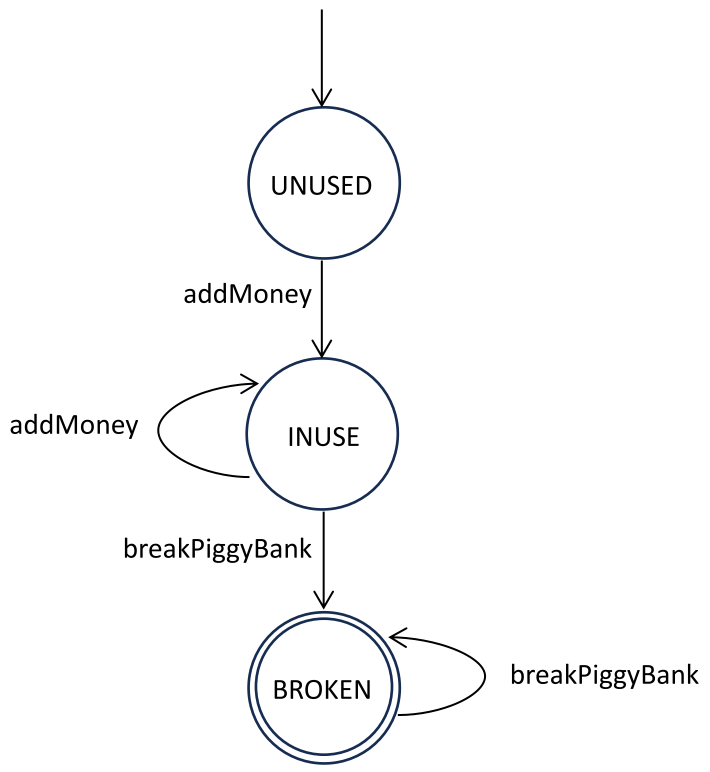
Question 0: Specification Languages

Piggy Bank Smart Contract:

Piggy Bank Smart Contract represents a PiggyBank, which allows the owner to deposit money into it and, after at least 365 days, break the PiggyBank to retrieve the deposited funds.

Finite State Automata:

To ensure that the piggy bank can only be broken once after at least one deposit has been made, we can represent this property using a finite state automaton which will illustrate the different states and transitions that occur as interactions with the piggy bank take place. Here's a finite state automaton diagram to visualise this property:



State-transition table that illustrates the possible state transitions for the PiggyBank smart contract:

Current State	Allowed Actions	New State
Unused	addMoney	InUse
InUse	addMoney	InUse
InUse	breakPiggyBank	Broken
Broken	breakPiggyBank	Broken

Legend:

- `***Unused***`: Initial state when the piggy bank contract is deployed.

- `***InUse***`: State after at least one deposit has been made.

- `***Broken***`: State after the piggy bank has been broken (funds withdrawn).

Transitions In this finite state automaton:

1. The initial state is Unused.
2. The transition addMoney from Unused to InUse signifies that the piggy bank has been used and is now in the InUse state. This transition also records the time of the first deposit.
3. From the InUse state, the breakPiggyBank transition leads to the Broken state only if the condition of at least 365 days since the first deposit has passed. Once in the Broken state, no more transitions are allowed.
4. If subsequent addMoney calls are made from either Unused or InUse, the piggy bank remains in the InUse state.
5. If the breakPiggyBank function is called again while in the InUse state and the time condition is met, the piggy bank transitions to the Broken state.
6. Once the breakPiggyBank function is successfully called and the piggy bank transitions to the Broken state, it remains in the Broken state permanently. Subsequent calls to the breakPiggyBank function will still be allowed, but they won't have any effect on the state or the balance, as the piggy bank will already be in the Broken state.
7. The breakPiggyBank function can only be called from the InUse state, so the direct transition from Unused to Broken is not possible using this function.
8. The addMoney() function can be called multiple times, allowing the owner to make multiple deposits into the piggy bank over time.

This finite state automaton enforces the desired property that the piggy bank can only be broken once after at least one deposit has been made, and any further attempts to break it are blocked.

Regular Expressions

Positive Regex:

```
 ( @addMoney .
@addMoney * .
 @addMoney * @breakPiggyBank .
)
```

Explanation:

1. **`( @addMoney .`** - This part ensures that the `addMoney` function is called at least once. The `(` symbol denotes the beginning of a sequence, and `@addMoney .` represents the occurrence of the `addMoney` function.
2. **`@addMoney * .`** - After the first call to `addMoney`, the `*` symbol indicates that the function can be called any number of times (`0` or more occurrences), allowing for multiple deposits.
3. **`@addMoney * @breakPiggyBank .`** - This segment specifies that after any number of calls to `addMoney`, the `breakPiggyBank` function can be called. This ensures that there is a sequence of depositing money followed by the ability to break the piggy bank.
4. **`)`** - The closing parentheses denote the end of the sequence.

In summary, this positive regex captures the idea that money can be deposited an arbitrary number of times before eventually breaking the piggy bank.

Negative Regex:

```
1.      ( ! @modify_timeOfFirstDeposit ) * .
2.      @begin_addMoney . ( ! @end_addMoney ) * .
3.      @end_addMoney . ( ! @begin_addMoney ) *
4.      ) . @timeOfFirstDeposit
```

Explanation:

1. ***`( ! @modify_timeOfFirstDeposit ) * .`*** - This part addresses the constraint that the `timeOfFirstDeposit` variable can only be modified during the initial execution of the `addMoney` function. The `( ! @modify_timeOfFirstDeposit ) * .` segment ensures that there can be any number of events before the `modify_timeOfFirstDeposit` event, which captures attempts to modify the variable.
2. ***`@begin_addMoney . ( ! @end_addMoney ) * .`*** - After the `begin_addMoney` event (indicating the start of a deposit), there can be any number of events (excluding `end_addMoney`) before a period. This enforces the constraint that the `timeOfFirstDeposit` variable can only be modified during the initial execution of the `addMoney` function.
3. ***`@end_addMoney . ( ! @begin_addMoney ) *`*** - After the `end_addMoney` event (indicating the end of a deposit), there can be any number of events (excluding `begin_addMoney`). This segment further enforces the constraint that modifications to `timeOfFirstDeposit` are allowed only within the context of the initial deposit.
4. ***`) . @timeOfFirstDeposit`*** - The closing parentheses denote the end of the sequence, followed by the `timeOfFirstDeposit` event.

In summary, this negative regex ensures that the `timeOfFirstDeposit` variable can only be modified during the initial execution of the `addMoney` function and prevents subsequent modifications.

Linear Time Logic (LTL)

```
1.      G ( @end_addMoney => 2. G !@*transfer )
```

Explanation of the LTL specification:

1. ***`G` (Globally):*** This operator specifies that the property within the parentheses should hold globally, meaning it should be true at all stages of the execution of the smart contract.

2. ***`@end_addMoney`:*** This represents a point in time immediately after the successful execution of the `addMoney` function. It indicates that money has been deposited into the smart contract.

3. ***`G !@*transfer`:*** This part specifies that there should not be any outgoing transfer events (`@*transfer`) from the smart contract at any point in time after the successful execution of the `addMoney` function.

Explanation:

The LTL specification focuses on ensuring that once money has been successfully deposited (`@end_addMoney`), no outgoing transfer events (`@*transfer`) should occur from the smart contract at any subsequent point in time (`G`). This specification aims to guarantee that funds deposited into the PiggyBank remain within the contract and are not transferred out under any circumstances.

In summary, the LTL specification strengthens the property that funds deposited into the PiggyBank contract cannot be withdrawn or transferred out, reinforcing the security and integrity of the deposited funds.

Computation Tree Logic (CTL)

1. AG (@start_addMoney => AF @end_breakPiggyBank)

Explanation of the CTL specification:

1. ***AG` (Universally Globally):*** This operator specifies that the property within the parentheses should hold true for all possible paths and states in the execution of the smart contract.
2. ***`@start_addMoney`:*** This represents a point in time immediately after the successful execution of the `addMoney` function starts. It indicates the initiation of a deposit.
3. ***`AF` (Universally Finally):*** This part specifies that for all possible paths, the event within the parentheses (`@end_breakPiggyBank`) will eventually happen.
4. ***`@end_breakPiggyBank`:*** This represents the point in time when the `breakPiggyBank` function is called, indicating that the user has chosen to break the piggy bank and withdraw the funds.

Explanation:

The CTL specification focuses on ensuring that for all possible paths of execution, if a deposit is initiated (`@start_addMoney`), then eventually on every path the `breakPiggyBank` function will be called (`@end_breakPiggyBank`).

In summary, the CTL specification strengthens the property that once a user initiates a deposit, on every possible path of execution within the smart contract, there will always be a point in time when the user can successfully break the piggy bank and withdraw the funds. This specification emphasises the universality of the property across all potential execution paths.

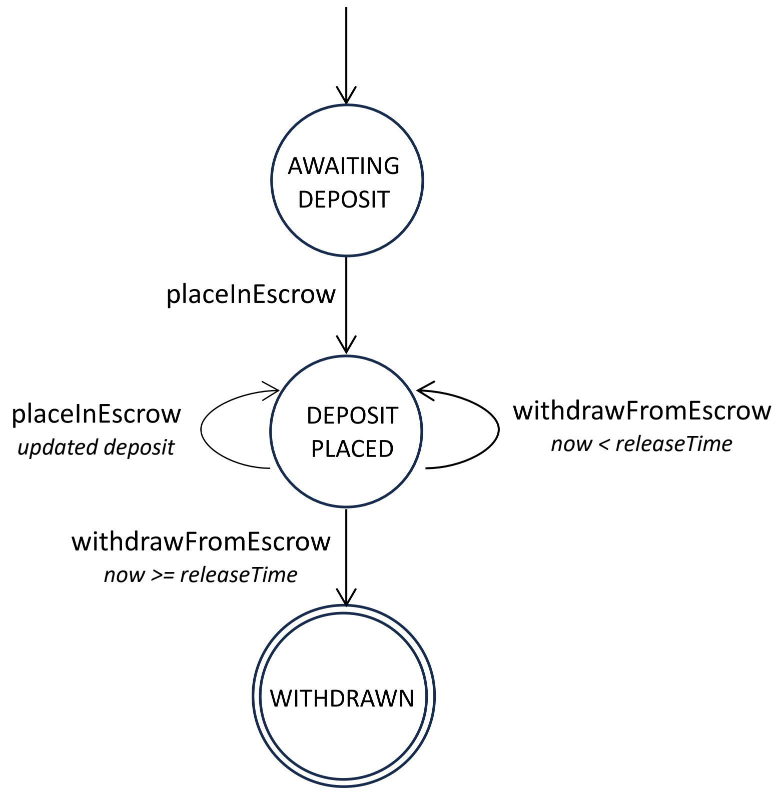
Escrow Smart Contract

Escrow1 is an implementation of a time-locked and unconditional escrow contract. An escrow contract acts as a trusted intermediary that holds funds until certain conditions are met, at which point the funds are released to a designated party. In this case, the contract has three main states: AwaitingDeposit, DepositPlaced, and Withdrawn.

Finite State Automata:

To ensure that the amount placed in escrow can only be withdrawn once by the receiver is an appropriate property to consider for an escrow contract. This property ensures that the receiver can only withdraw the funds from escrow once, preventing any possibility of double-spending or repeated withdrawals of the same funds. It adds an important layer of security and prevents potential misuse of the escrow contract.

Here's a finite state automaton diagram to visualise this property:



State Transitions and Events:

1. The contract starts in the AwaitingDeposit state.
2. The sender can place funds into escrow by calling placeInEscrow(), transitioning the contract to the DepositPlaced state.
3. After the releaseTime is reached, the receiver can call withdrawFromEscrow() to withdraw the funds, transitioning the contract to the Withdrawn state.
4. Once in the Withdrawn state, the contract remains there and no further actions can be performed.

- ***function placeInEscrow():*** This function is used by the sender to place funds into escrow. It can only be called by the sender (msg.sender == sender) and only when the contract is in the AwaitingDeposit state. It requires a positive amount of value to be sent along with the transaction. Upon successful execution, the contract transitions to the DepositPlaced state, sets the amountInEscrow, and calculates the releaseTime.

- ***function withdrawFromEscrow():*** This function is used by the receiver to withdraw funds from escrow. It can only be called by the receiver (msg.sender == receiver) and only when the contract is in the DepositPlaced state and the releaseTime has been reached (the current time is greater than or equal to releaseTime). Upon successful execution, the contract transitions to the Withdrawn state, transfers the funds to the receiver, and sets amountInEscrow to zero.

The table outlines the possible state transitions and their associated triggers:

Current State	Function Call	Next State
AwaitingDeposit	placeInEscrow	DepositPlaced
DepositPlaced	withdrawFromEscrow	Withdrawn
DepositPlaced	placeInEscrow	DepositPlaced
Withdrawn	withdrawFromEscrow/ placeInEscrow	Withdrawn
AwaitingDeposit	withdrawFromEscrow	AwaitingDeposit

Explanation:

1. If the contract is in the `AwaitingDeposit` state and the sender calls the `placeInEscrow` function with a positive value, the contract transitions to the `DepositPlaced` state.

2. If the contract is in the `DepositPlaced` state and the receiver calls the `withdrawFromEscrow` function after the `releaseTime` has been reached, the contract transitions to the `Withdrawn` state.

3. If the contract is in the `DepositPlaced` state and any other trigger occurs (such as an attempt to call `withdrawFromEscrow` before `releaseTime`), there is no transition; the contract remains in the `DepositPlaced` state.

4. If the contract is in the `Withdrawn` state, no triggers can cause a transition; the contract remains in the `Withdrawn` state.

5. If the contract is in the `AwaitingDeposit` state, no triggers can cause a transition; the contract cannot progress to other states until a deposit `placeInEscrow` is placed.

   - The Withdrawn state, both withdrawFromEscrow and placeInEscrow functions will revert when called. The state checks and modifiers prevent these functions from being executed in inappropriate states.

   - It possible to call the placeInEscrow function in the "DepositPlaced" state. This would allow the sender to update the escrowed amount and reset the release time. This is because the stateIs(State _state) modifier is used to check the state before executing but the modifier is not applied to the placeInEscrow function.

   - In the AwaitingDeposit state the withdrawFromEscrow function will revert because of the state modifier stateIs(State.DepositPlaced) inside the withdrawFromEscrow function. This modifier checks that the contract is in the DepositPlaced state before allowing the function to execute.

This finite state automaton captures the key states and transitions related to the property where the amount placed in escrow can only be withdrawn once by the receiver. It enforces the behaviour that once funds are successfully withdrawn, any further attempts to withdraw are considered invalid.

Regular Expressions

Positive Regular Expression

This positive regular expression verifies that the sender can place funds in escrow, release them through `releaseEscrow`, and then withdraw the funds through `withdrawFromEscrow`.

```
1.      @placeInEscrow . @releaseEscrow * . @withdrawFromEscrow .
```

1. ***@placeInEscrow:*** This part of the expression represents the event or action where the sender places funds into escrow.
2. ***.*:*** The dot (.) represents any character, and the asterisk (*) means "zero or more occurrences." So, .* matches any sequence of events that might occur between placing funds in escrow and releasing them.
3. ***@releaseEscrow***: This part represents the event or action where the funds are released from escrow.
4. ***.*:*** Similar to before, this part captures any events that might happen between releasing the funds and withdrawing them.
5. ***@withdrawFromEscrow:*** This part represents the event or action where the funds are actually withdrawn from the escrow contract.

The regular expression, as a whole, is trying to ensure that these three events (@placeInEscrow, @releaseEscrow, and @withdrawFromEscrow) happen in the correct sequence.

Negative Regular Expression

This negative regular expression ensures that the `releaseTime` variable can only be modified during the execution of the `placeInEscrow` function.

```
1.      ( !@begin_placeInEscrow )* . @releaseTime . ( !@end_placeInEscrow )*
```

This negative regular expression is intended to ensure that any assignment to the releaseTime variable should only occur between corresponding @begin_placeInEscrow and @end_placeInEscrow events.

- ***(!@begin_placeInEscrow)*:*** This part captures zero or more occurrences of a negated @begin_placeInEscrow event. The negation symbol (!) is used to indicate the absence of an event.
- ***@releaseTime:*** This is the event where the releaseTime variable is modified.
- ***(!@end_placeInEscrow)*:*** Similar to the first part, this captures zero or more occurrences of a negated @end_placeInEscrow event.

The intention of this regular expression is to make sure that any assignment to releaseTime is surrounded by matching pairs of @begin_placeInEscrow and @end_placeInEscrow events.

These regular expressions provide variations of positive and negative scenarios for the escrow smart contract, testing different properties and behaviors.

Linear Time Logic (LTL)

```
G ( (@start_withdrawFromEscrow => X @*transfer) U (@end_placeInEscrow && X
@start_withdrawFromEscrow) )
```

Explanation of the LTL specification:

1. ***`G`:*** The globally quantified operator that asserts the specified condition holds at all points in time.
2. ***`@start_withdrawFromEscrow`:*** The temporal event representing the moment when the `withdrawFromEscrow` function starts its execution.
3. ***`X`:*** The "next" operator, asserting that the condition following it holds at the next point in time.
4. ***`@*transfer`:*** The event of any outgoing transfer from the smart contract.
5. ***`U`:*** The "until" operator, specifying that the condition on the left holds until the condition on the right holds.
6. ***`@end_placeInEscrow`:*** The temporal event representing the moment when the `placeInEscrow` function completes its execution.

Explanation:

- The specification checks a different temporal relationship between events in the contract's execution timeline.
- The specification asserts that from the moment the `withdrawFromEscrow` function starts (`@start_withdrawFromEscrow`), there should be an outgoing fund transfer (`@*transfer`) occurring at the next point in time (`X`). This captures the idea that after initiating the withdrawal, funds are indeed transferred.
- The specification continues to assert that this condition should hold until the `placeInEscrow` function completes (`@end_placeInEscrow`) and then again until the `withdrawFromEscrow` function starts (`@start_withdrawFromEscrow`).

This specification provides a different perspective on the temporal relationships between the key events in the contract's execution, focusing on the occurrence of fund transfers after the initiation of withdrawal and before and after the completion of placing funds in escrow.

Computation Tree Logic (CTL)

1. AG (@start_placeInEscrow => AX @end_withdrawFromEscrow)

1. ***`AG`:*** The "For All Globally" operator, which asserts that the specified condition holds for all paths through the system.
2. ***`@start_placeInEscrow`:*** The temporal event representing the moment when the `placeInEscrow` function starts its execution.
3. ***`=>`:*** The implication operator, indicating that the condition on the right side implies the condition on the left side.
4. ***`AX`:*** The "Existence Next" operator, which asserts that the specified condition holds at the next point in time along a path.
5. ***`@end_withdrawFromEscrow`:*** The temporal event representing the moment when the `withdrawFromEscrow` function finishes its execution.

Explanation:

- The CTL specification aims to verify a different property of the smart contract: Whenever an escrow is placed (`@start_placeInEscrow`), it guarantees that in the next state (`AX`), the receiver will have successfully withdrawn the funds (`@end_withdrawFromEscrow`).
- The `AG` operator asserts that this property holds for all possible paths through the system.
- The implication (`=>`) captures the relationship between placing an escrow and the successful withdrawal of funds in the next state.
- The `AX` operator ensures that the withdrawal event occurs immediately after the placement event along the same path.

The CTL specification provided here asserts that whenever an escrow is placed in the smart contract, it is guaranteed that the receiver will have successfully withdrawn the funds in the very next state along all possible paths. This specification emphasises the immediate relationship between placing an escrow and the subsequent withdrawal event.