



Formal Verification Report for Rocket Joe

Summary

This document describes the specification and verification of Rocket Joe system using the Certora Prover. We undertook the work from January 13, 2022 to March 4, 2022. The latest commit that was reviewed and ran through the Certora Prover was [ef2ef302](#).

The scope of this verification is Rocket Joe system, particularly the following contracts:

- `LaunchEvent.sol`
- `RocketJoeFactory.sol`
- `RocketJoeStaking.sol`
- `RocketJoeToken.sol`

The Certora Prover proved the implementation of the distribution system is correct with respect to formal specifications written by the Certora team. The team also performed a manual audit of these contracts.

The formal specifications focus on validating the system's correct behavior as described by the Trader Joe team. The rules verify valid states of a system, proper transitions between states and the solvency of the system. The formal specifications have been submitted as a [pull request](#) against TraderJoe's public git repository.

Main Issues Discovered

Severity: High

Issue:	User unable to receive LP tokens
---------------	---

Issue:	User unable to receive LP tokens
Description:	<code>user.hasWithdrawnPair</code> is set to true before the transfer of LP tokens. This results in a <code>pairBalance(user)</code> value of 0.
Trader Joe Response:	This issue was fixed in commit 4804a0a9 .

Severity: High

Issue:	Reentrancy attack tokens
Description:	if the event is stopped and we can call <code>emergencyWithdraw()</code> , we can drain the system and take all the eth. <code>emergencyWithdraw()</code> calls <code>_safeTransferAVAX()</code> which uses the low level <code>call</code> function to <code>msg.sender</code> with amount <code>user.allocation</code> . it sets <code>user.allocation</code> to 0 only after the transfer, but in the transfer, we can call <code>emergencyWithdraw()</code> again and take the <code>user.allocation</code> before it's set to 0 until we take all the money from the system.
Trader Joe Response:	This issue was fixed in commit 578a4d5c .

Severity: High

Issue:	Create bad event DOS
Description:	anyone can create an event but there can only be a single event for each token. An attacker can create bad event for all the tokens and prevent anyone from creating an event. (bad event can be one with <code>maxAllocation==0</code>).
Trader Joe Response:	This issue was fixed in commit c750e722 .

Severity: High

Issue:	LaunchEvent createPair DOS
---------------	-----------------------------------

Issue:	LaunchEvent createPair DOS
Description:	An attacker can call <code>factory.createPair(address(WAVAX), address(token))</code> before the LaunchEvent reaches phase three, causing <code>LaunchEvent.createPair()</code> to revert on <code>require(factory.getPair(address(WAVAX), address(token)) == address(0))</code> . This will prevent anyone from creating a pair for the given token.
Trader Joe Response:	This issue was fixed in commit 93b2fcc9 .

Severity: High

Issue:	rJoeToken not initialized after setter
Description:	<code>setRJoe(address _rJoe)</code> only sets the <code>rJoe</code> state variable to <code>_rJoe</code> but doesn't call <code>initialize()</code> on it (the factory constructor also initializes the rJoe token).
Trader Joe Response:	This issue was fixed in commit 93b2fcc9 .

Severity: High

Issue:	lastRewardTimestamp is not set in constructor
Description:	A user can transfer joe externally to the contract to make <code>accRJoePerShare</code> increase by a huge number in the first <code>updatePool()</code> .
Trader Joe Response:	This issue was fixed in commit 93b2fcc9 .

Severity: Medium

Issue:	Anyone can call RocketJoeToken.initialize()
Description:	the <code>initialize()</code> function has no modifier and thus anyone can invoke it before it is called by the RocketJoeFactory. This prevents RocketJoeFactory from being initialized or changing the rJoe state variable.

Issue:	Anyone can call <code>RocketJoeToken.initialize()</code>
Trader Joe Response:	We acknowledge this issue, but handle it by making sure the contract is initialized properly post-deployment

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful, but we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Verification

Notation

✔ indicates the rule is formally verified on the latest reviewed commit, with the listed assumptions and simplifications.

✘ indicates the rule was violated under one of the tested versions of the code.

👉 indicates the rule is not yet formally specified.

🕒 indicates that some functions cannot be verified because the rules timed out

In this document, verification conditions are either shown as logical formulas or Hoare triples of the form $\{p\} C \{q\}$. A verification condition given by a logical formula denotes an invariant that holds if every reachable state satisfies the condition. Hoare triples of the form $\{p\} C \{q\}$ hold if any non-reverting execution of program C that starts in a state satisfying the precondition p ends in a state satisfying the postcondition q . Preconditions and postconditions are similar to the Solidity `require` and `assert` statements.

The syntax $\{p\} (C1 \sim C2) \{q\}$ is a generalization of Hoare rules, called relational properties. $\{p\}$ is a requirement on the states before $C1$ and $C2$, and $\{q\}$ describes the states after their executions. Notice that $C1$ and $C2$ result in different states.

Formulas relate the results of method calls. In most cases, these methods are getters defined in the contracts, but in some cases they are getters we have added to our harness or definitions provided in the rules file. Undefined variables in the formulas are treated as arbitrary: the rule is checked for every possible value of the variables.

Verification of LaunchEvent.sol

Summary

The LaunchEvent contract is used to collect AVAX from investors in preparation for the creation of a new liquidity pool. The contract manages 4 kinds of tokens:

- **WAVAX** (`LaunchEvent.avax`): Wrapped AVAX.
- **Launch tokens** (`LaunchEvent.token`): The new token being offered.
- **LP tokens** (`LaunchEvent.pair`): The tokens that allow investors to withdraw liquidity from the created pool.
- **RJoe** (`LaunchEvent.rJoe`): These tokens are required to invest in the launch.

The launch proceeds in several phases:

- Pre-start: an issuer creates the LaunchEvent contract and transfers some amount of launch tokens to it.
- Phase 1: investors can deposit or withdraw AVAX, although withdrawals are penalized at a rate that increases over time.
- Phase 2: no further deposits are allowed, but investors can withdraw their AVAX at a penalty
- Phase 3: All invested AVAX and some of the launch token are used to create a liquidity pool. The remaining launch token is held in reserve
- Post-timelock: Half of the LPToken and half of the reserves are allocated to the issuer; the other half is divided among the investors.

The launch can also be cancelled by the issuer before phase 3, allowing investors to withdraw their AVAX without penalty.

Assumptions and simplifications for verification

We made the following assumptions for the verification of the system:

- We assumed that each loop is executed at most one time.
- We assume that the WAVAX contract behaves correctly.
- We assumed that all libraries used in contracts (e.g., OpenZeppelin's libraries) are not malicious.
- We assumed that `LaunchEvent` contract is always initialized.
- We harnessed `_safeTransferAVAX()` method to overcome current tool limitations. This means that some reentrancy bugs may not be caught.
- We assumed that the `JoeRouter`, `JoeFactory`, `RJoeToken`, and other contracts use the implementations from the [joe-core repository](#) that were present at the time of verification.

Verification conditions

Definitions - valid states of a Launch Event

`open` : Period after `intialize()` call until `createPair()` call (it doesn't include stopped state of a system).

```
pair() == 0 ∧ ¬stopped()
```

`closed` : Period after `createPair()` call (it doesn't include stopped state of a system).

```
pair() ≠ 0 ∧ ¬stopped()
```

`openStopped` : Period after `allowEmergencyWithdraw()` call in `open` state.

```
pair() == 0 ∧ stopped()
```

`closedStopped` : Period after `allowEmergencyWithdraw()` call in `closed` state.

```
pair() ≠ 0 ∧ stopped()
```

Properties

Launch Event - always

(✓) `al_issuerAllocationZero` : Issuer's allocation is always 0.

```
getUserAllocation(issuer) == 0
```

(✓) `al_balanceLessThanAllocation` : User's balance cannot be greater than their allocation.

```
getUserBalance(user) ≤ getUserAllocation(user)
```

(✓) `al_userAllocationLessThanMaxAllocation` : User's allocation cannot exceed `maxAllocation`.

```
getUserAllocation(user) ≤ maxAllocation
```

(✓) `al_issuerTimelockNonZero` : `issuerTimelock` cannot be 0.

```
issuerTimelock ≥ 1
```

(✓) `al_userTimelockSeven` : `userTimelock` cannot exceed seven days.

```
userTimelock ≤ sevenDays()
```

(✓) `al_timelocksCorrelation` : `issuerTimelock` is greater than `userTimelock`.

```
issuerTimelock > userTimelock
```

Launch Event - open state

(✓) `op_incentivesCorrelation` : incentives correlation: `tokenIncentivesForUsers` + `tokenIncentiveIssuerRefund` == `tokenIncentivesBalance`.

```
open ⇒ tokenIncentivesForUsers + tokenIncentiveIssuerRefund ==  
tokenIncentivesBalance
```

(✓) `op_userNotWithdrawnPair` : User cannot have flag `hasWithdrawnPair` set to true.

```
open ⇒ ¬userHasWithdrawnPair(user)
```

(✓) `op_userNotWithdrawnIncentives` : User cannot have flag `hasWithdrawnIncentives` set to true.

```
open ⇒ ¬userHasWithdrawnIncentives(user)
```

(✓) `op_wavaxBalanceAndSumBalances` : `avaxReserve` is equal to the sum of all users balances: `avaxReserve == Σ getUI[user].balance`.

```
open ⇒ avaxReserve == sum_of_users_balances()
```

(✓) `op_avaxAllocZero` : `avaxAllocated` can be only 0.

```
open ⇒ avaxAllocated == 0
```

(✓) `op_lpSupplyZero` : `lpSupply` can be only 0.

```
open ⇒ lpSupply == 0
```

(✓) `op_PairBalanceIsZero` : `pair.balanceOf(address(this))` can be only 0.

```
open ⇒ getPairBalanceOfThis() == 0
```

(✓) `opPairAndTotalSupplyCorrelation` : TotalSupply of non-existing pair should be 0.

```
open ⇒ getPairTotalSupplyOfThis() == 0
```

(✓) `op_AvaxCorrelation` : `address(this).balance` is equal to the `avaxReserve`.

```
open ⇒ (getBalanceOfThis() == avaxReserve)
```

(✓) `op_tokenCorrelation` : token correlation: `token.balanceOf(currentContract) ≥ tokenReserve + tokenIncentivesBalance`.


```
open ⇒ getTokenBalanceOfThis() ≥ tokenReserve + tokenIncentivesBalance
```

(✓) `op_token_res_fixed` : `tokenReserve` remains unchanged.

```
{
  open ∧
  tokenReserveBefore = tokenReserve
}

< call to any function f >

{
  open ∧
  tokenReserveAfter = tokenReserve ∧
  tokenReserveBefore == tokenReserveAfter
}
```

Launch Event - closed state

(✓) `cl_avaxAllocSumUserBalances` : `avaxAllocated` is equal to the sum of all users balances: `avaxAllocated` is $\sum \text{getUA[user].balance}$.

```
closed ⇒ avaxAllocated == sum_of_users_balances()
```

(✓) `cl_avaxReservCheck` : `avaxReserve` has to be 0.

```
closed ⇒ avaxReserve == 0
```

(✓) `cl_incentivesCorrelation` : `tokenIncentivesBalance` less or equal than the sum of `tokenIncentivesForUsers` and `tokenIncentiveIssuerRefund()`.

```
closed ⇒ (tokenIncentivesBalance ≤ tokenIncentivesForUsers +
tokenIncentiveIssuerRefund)
```

(✓) `cl_userAllocUnchanging` : `getUA[user].allocation` remains unchanged.

```
{
  closed ∧
  userAllocationBefore = getUserAllocation(user)
}
```

```
< call to any function f >

{
  closed  $\wedge$ 
  userAllocationAfter = getUserAllocation(user)  $\wedge$ 
  userAllocationBefore == userAllocationAfter
}
```

(✓) `c1_userBalanceFixed` : `getUserInfo[user].balance` remains unchanged in closed state.

```
{
  closed  $\wedge$ 
  balanceBefore = getUserBalance(user)
}

< call to any function f >

{
  closed  $\wedge$ 
  balanceAfter = getUserBalance(user)  $\wedge$ 
  balanceBefore == balanceAfter
}
```

(✓) `c1_avaxAllocUnchanging` : `getUA[user].allocation` remains unchanged.

```
{
  closed  $\wedge$ 
  avaxAllocatedBefore = avaxAllocated
}

< call to any function f >

{
  closed  $\wedge$ 
  avaxAllocatedAfter = avaxAllocated  $\wedge$ 
  avaxAllocatedBefore == avaxAllocatedAfter
}
```

(✓) `c1_lpSupplyFixed` : `lpSupply` remains unchanged.

```
{
  closed  $\wedge$ 
```

```

    lpSupplyBefore = lpSupply
  }

  < call to any function f >

  {
    closed  $\wedge$ 
    lpSupplyAfter = lpSupply  $\wedge$ 
    lpSupplyBefore == lpSupplyAfter
  }

```

(✓) `cl_AvaxCorrelation` : `address(this).balance` and `avaxReserve` are equal to 0.

```

closed() => (getBalanceOfThis() == avaxReserve() && avaxReserve() == 0)

```

Launch Event - openStopped state

(✓) `os_tokenCorrelation` : Token correlation: `token.balanceOf(currentContract) ≥ tokenReserve + tokenIncentivesBalance`.

```

openStopped  $\Rightarrow$  getTokenBalanceOfThis()  $\geq$  tokenReserve +
tokenIncentivesBalance

```

(✓) `os_avaxAllocSumUserBalances` : `avaxAllocated` is equal to the sum of all users balances: `avaxAllocated` is \sum `getUA[user].balance`.

```

openStopped  $\Rightarrow$  avaxReserve == sum_of_users_balances()

```

(✓) `os_avaxReserveDecrease` : `avaxReserve` cannot increase

```

{
  openStopped  $\wedge$ 
  avaxBefore = avaxReserve
}
< call to any function f >
{
  openStopped  $\wedge$ 
  avaxAfter = avaxReserve  $\wedge$ 
  avaxBefore  $\geq$  avaxAfter
}

```

(✓) `os_userBalanceNonIncreasing` : `user.balance` cannot increase.

```
{
  openStopped  $\wedge$ 
  balanceBefore = getUserBalance(msg.sender)
}

< call to any function f >

{
  openStopped  $\wedge$ 
  balanceAfter = getUserBalance(msg.sender)  $\wedge$ 
  balanceBefore  $\geq$  balanceAfter
}
```

Launch Event - closedStopped state

(✓) `cs_lpSupplyFixed` : `lpSupply` remains unchanged.

```
{
  closedStopped  $\wedge$ 
  lpSupplyBefore = lpSupply
}

< call to any function f >

{
  closedStopped  $\wedge$ 
  lpSupplyAfter = lpSupply  $\wedge$ 
  lpSupplyBefore == lpSupplyAfter
}
```

(✓) `cs_userBalanceFixed` : `user.balance` remains unchanged.

```
{
  closedStopped  $\wedge$ 
  balanceBefore = getUserBalance(msg.sender)
}

< call to any function f >

{
  closedStopped  $\wedge$ 
  balanceAfter = getUserBalance(msg.sender)  $\wedge$ 
  balanceBefore == balanceAfter
}
```

```
    balanceBefore == balanceAfter
  }
```

(✓) `cs_avaxAllocatedFixed` : `avaxAllocated` remains unchanged.

```
{
  closedStopped  $\wedge$ 
  avaxBefore = avaxAllocated
}

< call to any function f >

{
  closedStopped  $\wedge$ 
  avaxAfter = avaxAllocated  $\wedge$ 
  avaxBefore == avaxAfter
}
```

Launch Event - high-level properties

(✓) `hl_EqualityOfToken` : Tokens correlation: token balance of this == `tokenReserve` + `tokenIncentivesBalance` .

```
getTokenBalanceOfThis() == tokenReserve + tokenIncentivesBalance
```

(✓) `hl_depositAdditivity` : Additivity of deposit: `deposit(a)`; `deposit(b)` has same effect as `deposit(a+b)`.

```
e.msg.sender == e2.msg.sender  $\wedge$ 
e.msg.value == 2 * e2.msg.value  $\wedge$ 
depositAVAX(e) ~ depositAVAX(e2); depositAVAX(e2)
```

Equivalent with respect to the `getUserBalance(msg.sender)`

(✓) `hl_withdrawAdditivity` : Additivity of withdraw: `withdraw(a)`; `withdraw(b)` has same effect as `withdraw(a+b)`.

```
x = y + z  $\wedge$ 
withdrawAVAX(x) ~ withdrawAVAX(y); withdrawAVAX(z)
```

Equivalent with respect to the `getUserBalance(msg.sender)`

(✓) `h1_noDepositFrontRun` : No front-running for deposit: effect of deposit unchanged by an intervening operation by another user.

```
< call to any function f by another user(e2.msg.sender)>; depositAVAX(e) ~  
    depositAVAX(e)
```

Equivalent with respect to the `getUserBalance(e.msg.sender)`

Also, the second part of this rule checks the correctness of withdraw in both cases.

```
{  
  open  $\Lambda$   
  e.msg.sender  $\neq$  e2.msg.sender  $\wedge$   
  userBalanceBefore = getUserBalance(e.msg.sender)  
}  
  
depositAVAX(e); userBalanceAfter1 := getUserBalance(e.msg.sender) ~  
  < call to any function f by another user(e2.msg.sender)>;  
  depositAVAX(e); userBalanceAfter2 :=  
  getUserBalance(e.msg.sender)  
  
{  
  userBalanceBefore - amount == userBalanceAfter1  $\wedge$   
  userBalanceBefore - amount == userBalanceAfter2  
}
```

(✓) `h1_stoppedOnlySwitch` : If stopped then only `allowEmergencyWithdraw` was called.

```
{  
  stopped == false  
}  
  
< call to any function f >  
  
{  
  stopped == true  $\Rightarrow$  f == allowEmergencyWithdraw()  
}
```

(✓) `h1_onlyOwnerSwitch` : Only owner can call `allowEmergencyWithdraw()`.

```
{  
  stopped == false  
}
```

```

allowEmergencyWithdraw()

{
  stopped == true ⇒ msg.sender == getOwner()
}

```

(✓) `h1_whatShouldRevert` : If event is stopped, appropriate functions will revert.

```

{
  stopped == true
}

< call to any function f >

{
  f == depositAVAX() || withdrawAVAX() ||
  createPair() || withdrawLiquidity() reverts
}

```

(✓) `h1_twoSideInverse` : Deposit and withdraw are two-sided inverses on the state (if successful).

```

depositAVAX(); withdrawAVAX() ~ withdrawAVAX(); depositAVAX()

Equivalent with respect to the getUserBalance(msg.sender)

```

(✓) `h1_noWithdrawFrontRun` : No front-running for withdrawAVAX().

```

withdrawAVAX(e, amount) ~
  < call to any function f by another user(e2.msg.sender)>;
withdrawAVAX(e, amount)

Equivalent with respect to the getUserBalance(e.msg.sender)

```

Also, the second part of this rule checks the correctness of withdraw in both cases.

```

{
  open ∧
  e.msg.sender ≠ e2.msg.sender ∧
  userBalanceBefore = getUserBalance(e.msg.sender)
}

```

```

    withdrawAVAX(e, amount); userBalanceAfter1 := getUserBalance(e.msg.sender) ~
      < call to any function f by another user(e2.msg.sender)>;
    withdrawAVAX(e, amount); userBalanceAfter2 := getUserBalance(e.msg.sender)

    {
      userBalanceBefore - amount == userBalanceAfter1 ^
      userBalanceBefore - amount == userBalanceAfter2
    }

```

Verification of RocketJoeStaking.sol


Summary

The RocketJoeStaking contract is responsible for the distribution of the RocketJoeToken (rJoe), which is one of the two tokens required of a user when investing into a launch event. Users will stake JOE, and in return when the user deposits or withdraws JOE they will also receive rJoe based on the duration of time they staked.

Important Variables

- JOE balance of RJStaking
- JOE balance of individual users
 - userInfo[user].amount
- Reward debt of users
 - userInfo[user].rewardDebt
 - essentially a marker for the last time the user received rewards
- rJoePerSec
 - emission rate of rJoe
- accRJoePerShare
- pendingReward(user)
 - Calculated owed rJoe
- initialized
- totalJoeStaked

Invariants

()¹ `staking_joe_bal_sums_user_balance` : Joe balance of the staking contract is greater than or equal to the sum of all staked JOE

```
ERC20(JOE).balanceOf(currentContract) ≥ ∑userInfo.amount
```


(✓)¹ `totalJoeStaked_sums_user_balance` : Similar to the above invariant but done a different way. The stored balance of the contract in the variable `totalJoeStaked` is greater than or equal to the sum of all staked Joe

```
totalJoeStaked ≥ ∑userInfo.amount
```

(✓) `balanceOf_Joe_eq_totalJoeStaked` : `totalJoeStaked` is accurate to the ERC20 Joe balance of staking

```
ERC20(Joe).balanceOf(currentContract) == totalJoeStaked
```

(✓) `user_balances_less_than_totalJoeStaked` : The sum of user balances for any two users is less than or equal to the total sum of Joe staked

```
totalJoeStaked ≥ userInfo[userA].amount + userInfo[userB].amount
```

Rules

(✓) `userInfo_amount_safe_mutate` : Only functions called by the user may alter the staked balance of that user, and only through the main functions of deposit, withdraw, and emergency withdraw

```
{
  userBalance_pre = userJoeStaked(user)
}

< call any function f >

{
  userBalance_post = userJoeStaked(user) ∧
  (userBalance_pre ≠ userBalance_post =>
    (f == deposit() ∨ withdraw() ∨ emergencywithdraw()) ∧
    msg.sender == user)
}
```

(✓)² `RJPS_only_owner_and_function` : Only the owner may change the `rJoePerSec`, and only through the function `updateEmissionRate`

```

{
  RJPS_pre = rJoePerSec()
}

< call any function f >

{
  RJPS_post = rJoePerSec() ∧
  (userBalance_pre ≠ userBalance_post =>
    f == updateEmissionRate() ∧
    msg.sender == getOwner())
}

```

(👉) ²³ `pending_reward_decreased_only_user` : Only functions called by the user may change the pending reward received by the user

```

{
  totalJoeStaked() < 1000000000000000 ∧
  rjoe_pre = pendingRJoe(user)
}

< call any function f >

{
  rjoe_post = pendingRJoe(user) ∧
  rjoe_post < rjoe_pre => e.msg.sender == user
}

```

(✅) ⁴ `staking_non_trivial_rJoe` : If a user stakes non-zero joe they will eventually receive rJoe

```

{
  PRECISION() > 0 ∧
  rJoePerSec() > 0 ∧
  rJoePerSec() < 1000000
}

deposit()

{
  block.timestamp > lastRewardTimestamp() ∧
  userRewardDebt(msg.sender) < max_uint256 ∧
  dt = e1.block.timestamp - lastRewardTimestamp() ∧
  rewards = pendingRJoe(msg.sender) ∧
}

```

```
exists uint256 t. (t == dt) => rewards > 0
}
```

(✓) `staking_trivial_on_zero_time` : After staking, a user will not be able to immediately increase their pending rJoe

```
{ }
deposit()
{
  pendingRJoe(msg.sender) == 0
}
```

(✓) `longer_stake_greater_return` : Staking longer eventually yields greater return. If user A and user B stake the same amount of Joe, if user A stakes longer than user B they eventually will receive a greater return.

```
{
  e2.block.timestamp > e1.block.timestamp
}
deposit(e1); rJoe1 := pendingRJoe(e1, msg.sender) ~
  deposit(e2); rJoe2 := pendingRJoe(e2, msg.sender)
{
  exists uint256 dt. e2.block.timestamp - e1.block.timestamp >= dt =>
    rJoe2 > rJoe1
}
```

(✓) `deposit_no_frontrunning` : Calling a function before deposit will not change the effects of deposit. Measures the balance added to the staking account

```
< call any function f >; deposit(x) ~ deposit(x)
```

Equivalent with respect to the `userJoeStaked(msg.sender)` and `joe.balanceOf(msg.sender)`

(✓) `withdraw_no_frontrunning` : Calling a function before withdraw will not change the effects of calling withdraw. Measures the balance removed from the staking account

```
< call any function f >; withdraw(x) ~ withdraw(x)
```

Equivalent with respect to the `userJoeStaked(msg.sender)` and `joe.balanceOf(msg.sender)`

(✓) `additivity_withdraw` : successive calls to `withdraw` of `x` and `y` are the same as withdrawing the sum of `x` and `y`

```
withdraw(x); withdraw(y) ~ withdraw(x + y)
```

Equivalent with respect to the `userJoeStaked(msg.sender)`

(✓) `additivity_deposit` : successive calls to `deposit` of `x` and `y` are the same as depositing the sum of `x` and `y`

```
deposit(x); deposit(y) ~ deposit(x + y)
```

Equivalent with respect to the `userJoeStaked(msg.sender)`

Unit Test Rules

(✓) `verify_deposit` : when calling `deposit` the right amount of Joe is transferred, `totalJoeStaked` is updated properly, reward debt is increased, user receives all of their pending `rJoe`

(✓) `verify_withdraw` : when calling `withdraw` the right amount of JOE is transferred, `totalJoeStaked` is updated properly, reward debt is increased, user receives all of their pending `rJoe`

(✓) `verify_updateEmissionRate` : accurately changes `EmissionRate`

(✓) `updatePool_contained` : `updatePool` does not affect users staked joe or reward debt

(✓) `verify_emergencyWithdraw` : `emergencyWithdraw` does not fail, updates `totalJoeStaked`, and removes all of the user's balance

* Was not able to successfully verify it does not revert due to issues with the tool,
Assuming it not to revert allowed the others to pass

Rules not fully Implemented

`updatePool_increases_accRJoePerShare` : guarantees that `accRJoePerShare` is consistently increasing.

- dropped due to other constraints and little additional coverage

`rJoe_solvency` : Staking always contains enough `rJoe` to payout a users pending `rJoe` rewards `rJoe.balanceOf(currentContract) ≥ pendingRewards(user)` Assumes `msg.sender` is not the current Contract

`stake_duration_correlates_return` : If one user stakes the same amount of Joe as another, but for a longer duration, they will receive greater than or equal the amount of `rJoe`

1. This rule does not currently pass on the `initialize` method, but we believe this is a setup problem and does not reflect an error in the contract. ↩ ↩²
 2. We assume the state is initialized. ↩ ↩²
 3. This rule is passing on everything but the emergency withdraw function. We believe that the rule failures on emergency are caused by setup issues and do not reflect bugs in the code. ↩
 4. This rule was originally written to calculate the exact interval within which `rJoe` would be distributed. However, due to timeouts, it was changed to only guarantee that at *some* time `t` it would be distributed. ↩
-