Gnosis Audit Report

AbstractEvent.sol

Events (and other related contracts) are using three integer types for different purposes:
- Type **uint8** is used as an index in arrays of outcomes, implying that there could be no more than 255 possible outcomes
- Type **int** (signed 256 bit) is used for the outcome's value. A signed type is chosen to be able to represent negative as well as positive outcome values (important for ScalarEvent-s)
- Type **uint** (unsigned 256 bit) is used for representing quantities of tokens.

Events (logs) declared in the AbstractEvent.sol, appear to be consistent with the use of integer types described above.

Constructor

Creation of events with many outcomes could be costly in terms of gas. Here are results of measurements (together with measurements for other functions):

Number of outcomes	Gas cost of creation	Gas cost of buying all outcomes (1st time)	Gas cost of buying all outcomes (2nd time)	Gas cost of selling all outcomes (not last time)	Gas cost of selling all outcomes (last time)	Gas cost of querying outcome token distribution for an address
2	2.033 M	147 K	42.5 K	81.9 K	33.4 K	25.8 K
3	2.666 M	191 K	56.8 K	96.3 K	40.6 K	27.1 K
4	3.299 M	236 K	71.0 K	110 K	47.9 K	28.4 K
5	3.932 M	280 K	85.2 K	125 K	55.1 K	29.7 K
6	4.564 M	324 K	99.5 K	139 K	62.4 K	31.0 K
7	5.197 M	368 K	113 K	154 K	69.6 K	32.3 K
8	5.830 M	412 K	127 K	168 K	76.8 K	33.6 K
9	6.463 M	457 K	142 K	183 K	84.1 K	34.9 K
10	7.096 M	501 K	156 K	197 K	91.3 K	36.2 K
11	7.729 M	545 K	170 K	212 K	98.5 K	37.5 K
12	8.361 M	589 K	184 K	226 K	105 K	38.8 K
13	8.8994 M	634 K	199 K	241 K	113 K	40.1 K

buyAllOutcomes

Using this function has associated gas cost of roughly 45 K gas per event outcome (shown in the table above). This cost get reduced when an address buys all outcomes for the second time, where it costs roughly 14K per event outcome (due to the cost of occupying the storage).

Suggestion: it is important to keep the gas cost of “buyAllOutcomes” function low. Therefore, one way would be for the Event contract to store a mapping of number of complete sets purchased. This mapping will then be made accessible to the OutcomeToken contract (functions “balanceOf”, “transfer”, “transferFrom”). It might require an additional (more costly) function to “optimise” the holdings of outcomes tokens into the complete sets, with the gas cost comparable to the cost of “sellAllOutcomes” function today.

collateralToken is a potentially untrusted contract, can it execute any attacks on buyAllOutcomes from the “transferFrom” method? No, since there are no state modification before the invocation of transferFrom that are assumed to still hold after this method. Suggestion: add a comment before the “transferFrom” call that any state modifications need to be re-checked after “transferFrom”, due to possible re-entrancy.

sellAllOutcomes

Gas cost of using this function is similar to buyOutcomes. However, the gas cost is lowest when an address sells remaining outcomes, because of the gas refund for cleared storage.

collateralToken is a potentially untrusted contract, can it execute any attacks on sellAllOutcomes from the “transfer” method? No, since there are no state reading after the invocation of “transfer”. Suggestion: add a comment after the “transfer” call that any state modifications need to be re-checked after “transfer”, due to possible re-entrancy.

getOutcomeTokens
Gas costs of this function call depending on the number of outcomes are not presented in the table above, but the measurements are present in the corresponding unit test. For 13 outcomes the cost is 25.1 K gas.

getOutcomeTokenDistribution
The gas cost of calling this method for one address is presented in the table above. Obviously, this would only be relevant when this function is called from within another smart contract.

setOutcome
This function could be potentially factored out so that the event does not store the outcome and the isOutcomeSet flag, but instead delegates it to the oracle. This could simplify the code and remove an extra step in the event’s lifecycle.

CategoricalEvent.sol

redeemWinnings

An obvious quirk of this function is the fact that “outcome” field of the type “int" (unsigned to accommodate scalar events), is re-interpreted as an index in the array of the categorical outcomes. If the outcome is negative, it prevents everyone from ever redeeming their winnings. This scenario is added to the corresponding test case **events/ test_redeem_winnings_for_categorical_event.py**

Suggestion - add method revokeAll() in the OutcomeToken to be able to do balanceOf and revoke in the same go (saving gas).

collateralToken is a potentially untrusted contract. Suggestion: add a comment after the “transfer” call that any state modifications need to be re-checked after “transfer”, due to possible re-entrancy.

getEventHash

This particular composition of hash function (which is then used to ensure the uniqueness of the events via the event factories), as well as closely matching life cycles (see comments on the “setOutcome” function in the AbstractEvent.sol section) mean that events and oracles are

essentially in 1:1 relation to each other (because you cannot create an event on the same oracle with the same factory). If this is intentionally, it begs the question “why are they not sub-components of a single entity?”.

ScalarEvent.sol

redeemWinnings

There is a way to cause an overflow in the formulae calculating convertedWinningOutcome. This situation is demonstrated in the added test to the **events/test_redeem_winnings_for_scalar_event.py**
Parameters: lowerBound = -2^254, upperBound = 2^254, outcome = 2^254. In a correct calculation, the convertedWinningOutcome should be OUTCOME_RANGE, and the HIGH tokens should be fully rewarded. However, due to the overflow, convertedWinningOutcome turns out to be 0, and LOW tokens win instead.

Considered storing outcome as “factorLong” and “factorShort” instead of “outcome” for this type of event? To save of calculation for each redemption.

getEventHash

Same comment as for the “getEventHash” of the CategoricalEvent.sol

EventFactory.sol

createCategoricalEvent, createScalarEvent

Suggestion: provide a static method in the CategoricalEvent contract to compute hash, so that the components and their order is in one place and easier to refactor.

Suggestion: if the goal of the factory is to control the creation of the events, it might make sense to add the ability to check from within any contract that a certain event has been created via the factory. It is currently possible, but requires iterating over two mappings in the factory contract. Perhaps there should be just one mapping, but the event hash needs to include the event type.

StandardToken.sol

Noticed that transferFrom and transfer do not throw when the addition and subtraction is not safe
Noticed that instead of using solidity-generated balanceOf, allowance, and total supply, use private fields and public methods. Is this for gas saving?

EtherToken.sol

Suggestion: add a fallback function to this contract that would simply call “deposit” method.

Suggestion: potentially remove the EtherToken altogether by adding a special handling into Event, Market, and UltimateOracle contracts. This could greatly increase usability for the largest proportion of the users (assumption), who will be using ETH as collateral. It can also further reduce the gas cost of such transactions.

OutcomeToken.sol

Suggestion: see suggestion above about the gas cost of “buyAllOutcomes” of the AbstractEvent.sol

CentralisedOracle.sol

Member fields “outcome” and “isSet” do not need to be public, because there are already public getters.

DifficultyOracle.sol

It does not get the difficulty at a certain block, but a difficulty at a block not earlier than one specified in the constructor. Apart from correcting comments in the contract, it is good to think of situations when two difficulty oracles which look the same based on the constructor arguments, can nevertheless return different results. Could this be a problem anywhere?
Suggestion: perhaps this oracle needs a small incentive attached to it, which will compensate for the gas cost of calling “setOutcome”, and give a bit more? Post-Metropolis (due to EIP-86) this will likely be a very common pattern, and miners will be routinely looking for such incentives to execute them on time.

MajorityOracle.sol

Constructor

The table below gives approximate gas cost of creating a MajorityOracle depending on the number of sub-oracles

Number of sub-oracles	Gas cost of creation	Gas cost of computing the outcome (sub-oracles are voting the same way)	Gas cost of computing the outcome (sub-oracles are voting in different ways)
3	373 K	29.2 K	30.0 K
4	396 K	31.5 K	33.1 K
5	417 K	33.8 K	36.4 K
6	440 K	36.0 K	39.9 K
7	462 K	38.3 K	43.6 K
8	484 K	40.5 K	47.6 K
9	506 K	42.8 K	51.8 K
10	529 K	45.1 K	56.2 K
11	551 K	47.3 K	60.9 K
12	573 K	49.6 K	65.8 K
13	595 K	51.8 K	70.9 K

getStatusAndOutcome
One concern that has been considered was whether a majority oracle could become un-resolvable due to gas cost limitation (when number of sub-oracles is high). However, as the table above shows, the cost of creation of such an oracle would be considerably more than the cost of resolution, therefore if it was possible to create an oracle, it should be possible to resolve it.

Another things that was considered: “Could the algorithm be re-written into using a hash map of outcomes and counting the out that got the maximum votes?” Probably not, because mappings cannot be created dynamically. But we can try to clear up the mapping after the execution. Compared the gas costs with the existing function - existing method is still cheaper.

Coding style: For loop without curly braces - potential trap for someone adding the code.

isOutcomeSet, getOutcome
Both of these call getStatusAndOutcome as a subroutines, and they assume that “getOutcome” function of the sub-oracles does not change its value. If one of the sub-oracles contains an error which causes it to provide different values for getOutcome when called multiple times, it might appear that the MajorityOracle is buggy. Suggestion - make sure that the getStatusAndOutcome function does not call sub-oracles once their outcomes are set.

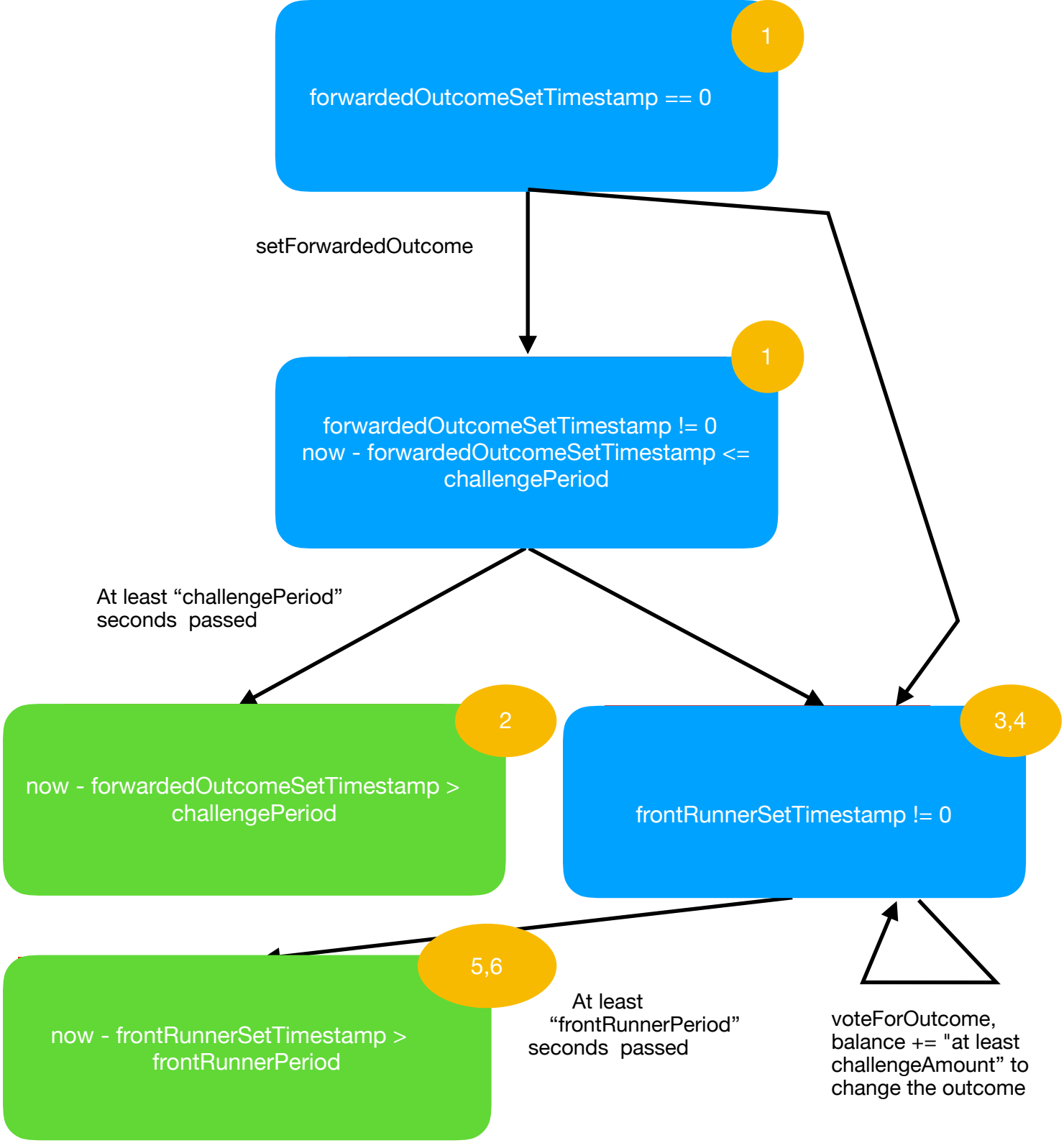
As pointed out in the documentation, it is possible that neither outcome ever gets a majority, and therefore, the event never resolves. The proposed solution in the document is always use something like UltimateOracle on top of any MajorityOracle.
Suggestion: Introduce penalties for the sub-oracles (for example, for CentralisedOracle-s) that cause the oracle to escalate to the UltimateOracle. In a properly working system, the UltimateOracle should not be invoked at all, but there should be a threat of punishment.

SignedMessageOracle.sol
Member fields “outcome” and “isSet” do not need to be public, because there are already public getters.
replacedSigner
Introduction of nonce to prevent replay attacks is superfluous. Because this function can only be called by the current signer (isSigner modifier), and signers have in-built nonces to prevent replays, in-contract nonce is not needed. It could also be that introducing the “isSigner” modifier was a mistake, in which case the nonce is required.

SignedMessageOracleFactory.sol
In the creation function, one can read “signer” from the newly created oracle instead of doing ecrecover another time.

UltimateOracle.sol

Please see on the next page


forwardedOutcomeSetTimestamp == 0
1
setForwardedOutcome
forwardedOutcomeSetTimestamp != 0
now - forwardedOutcomeSetTimestamp <=
challengePeriod
1
At least “challengePeriod”
seconds passed
now - forwardedOutcomeSetTimestamp >
challengePeriod
2
frontRunnerSetTimestamp != 0
3,4
now - frontRunnerSetTimestamp >
frontRunnerPeriod
5,6
At least
“frontRunnerPeriod”
seconds passed
voteForOutcome,
balance += "at least
challengeAmount” to
change the outcome

Reasoning about the UltimateOracle

Lemma 1: Once "isOutcomeSet()" starts returning "true", it never returns "false" anymore.
Proof:
Here is the formula for "isOutcomeSet":

```
isChallengePeriodOver() && !isChallenged()

    || isFrontRunnerPeriodOver();
```

First thing to observe is that "isFrontRunnerPeriodOver() implies "isChallenged()". Therefore, there are three cases to consider:

	isChallengePeriodOver	isChallenged	isFrontRunnerPeriodOver	isOutcomeSet
1	FALSCH	FALSCH	FALSCH	FALSCH
2	WAHR	FALSCH	FALSCH	WAHR
3	FALSCH	WAHR	FALSCH	FALSCH
4	WAHR	WAHR	FALSCH	FALSCH
5	FALSCH	WAHR	WAHR	WAHR
6	WAHR	WAHR	WAHR	WAHR

In the case 2, we have:

```
forwardedOutcomeSetTimestamp != 0 &&
now.sub(forwardedOutcomeSetTimestamp) > challengePeriod

!(frontRunnerSetTimestamp != 0);

!(frontRunnerSetTimestamp != 0 && now.sub(frontRunnerSetTimestamp)
> frontRunnerPeriod);
```

Eliminating negation and applying De-Morgan law we get:

```
forwardedOutcomeSetTimestamp != 0
now.sub(forwardedOutcomeSetTimestamp) > challengePeriod
frontRunnerSetTimestamp == 0
frontRunnerSetTimestamp == 0 || now.sub(frontRunnerSetTimestamp)
<= frontRunnerPeriod
```

The last expression is superseded by the one before the last:

```
forwardedOutcomeSetTimestamp != 0
now.sub(forwardedOutcomeSetTimestamp) > challengePeriod
frontRunnerSetTimestamp == 0
```

Now we look at all the functions that could potentially change these variables above:

"setForwardedOutcome" will fail at the precondition "forwardedOutcomeSetTimestamp == 0"
"challengeOutcome" will fail at the precondition "!isChallengePeriodOver"
"voteForOutcome" will fail at the precondition "isChallenged"

In the cases 5 and 6, we have

```
frontRunnerSetTimestamp != 0 && now.sub(frontRunnerSetTimestamp) >
frontRunnerPeriod
```

We look again at the functions that can change these variables:
“setForwardedOutcome” does not change the value of “frontRunnerSetTimestamp”, therefore, the expression “isFrontRunnerPeriodOver()” will still hold.
“challengeOutcome” will fail at the precondition “!isChallenged”.
“voteForOutcome” will fail at the precondition “!isFrontRunnerPeriodOver”.

QED (Lemma 1)

Next, we will analyse non-terminal states and verify the state transition diagram.

If we negate the formulae for “isOutcomeSet()”:

```
isChallengePeriodOver() && !isChallenged()

                || isFrontRunnerPeriodOver()
```

we get:

```
(!isChallengePeriodOver() || isChallenged())

                && !isFrontRunnerPeriodOver()
```

	isChallengePeriodOver	isChallenged	isFrontRunnerPeriodOver	isOutcomeSet
1	FALSCH	FALSCH	FALSCH	FALSCH
2	WAHR	FALSCH	FALSCH	WAHR
3	FALSCH	WAHR	FALSCH	FALSCH
4	WAHR	WAHR	FALSCH	FALSCH
5	FALSCH	WAHR	WAHR	WAHR
6	WAHR	WAHR	WAHR	WAHR

We need to consider three cases again.
In the case 1, we get

```
!(forwardedOutcomeSetTimestamp != 0 &&
now.sub(forwardedOutcomeSetTimestamp) > challengePeriod)
frontRunnerSetTimestamp == 0
!(frontRunnerSetTimestamp != 0 && now.sub(frontRunnerSetTimestamp)
> frontRunnerPeriod)
```

After simplifications, we get

```
forwardedOutcomeSetTimestamp == 0 ||
now.sub(forwardedOutcomeSetTimestamp) <= challengePeriod
frontRunnerSetTimestamp == 0
frontRunnerSetTimestamp == 0 || now.sub(frontRunnerSetTimestamp)
<= frontRunnerPeriod
```

The last equation is superseded by the one before last:

```
forwardedOutcomeSetTimestamp == 0 ||
now.sub(forwardedOutcomeSetTimestamp) <= challengePeriod
frontRunnerSetTimestamp == 0
```

This situation can be interpreted as follows. Since “forwardedOutcomeSetTimestamp” can only be set in the “setForwardedOutcome” function, it means that this function has not been successfully called yet (forwardedOutcomeSetTimestamp == 0), or it has been been successfully called no earlier than “challengePeriod” seconds ago. No other function can manipulate “forwardedOutcomeSetTimestamp. Since “frontRunnerSetTimestamp==0”, none of the functions “challengeOutcome” or “voteForOutcome” have been called yet.
Therefore, in this situation, the balance of the UltimateOracle is 0.

In the case 3, we get:

```
!(forwardedOutcomeSetTimestamp != 0 &&
now.sub(forwardedOutcomeSetTimestamp) > challengePeriod)
frontRunnerSetTimestamp != 0
!(frontRunnerSetTimestamp != 0 && now.sub(frontRunnerSetTimestamp)
> frontRunnerPeriod)
```

Having simplified:

```
forwardedOutcomeSetTimestamp == 0 ||
now.sub(forwardedOutcomeSetTimestamp) <= challengePeriod
frontRunnerSetTimestamp != 0
frontRunnerSetTimestamp == 0 || now.sub(frontRunnerSetTimestamp)
<= frontRunnerPeriod
```

The first term in the last equation is incompatible with the one before the last, so we eliminate it:

```
forwardedOutcomeSetTimestamp == 0 ||
now.sub(forwardedOutcomeSetTimestamp) <= challengePeriod
frontRunnerSetTimestamp != 0
now.sub(frontRunnerSetTimestamp) <= frontRunnerPeriod
```

This situation can be interpreted as follows. Since “forwardedOutcomeSetTimestamp” can only be set in the “setForwardedOutcome” function, it means that this function has not been successfully called yet (forwardedOutcomeSetTimestamp == 0), or it has been been successfully called no earlier than “challengePeriod” seconds ago. No other function can manipulate “forwardedOutcomeSetTimestamp. However, since “frontRunnerSetTimestamp != 0”, the “challengeOutcome” and/or “voteForOutcome” functions have been successfully called no earlier than “frontRunnerPeriod” ago.

In the case 4, we get:

```
forwardedOutcomeSetTimestamp != 0 &&
now.sub(forwardedOutcomeSetTimestamp) > challengePeriod
frontRunnerSetTimestamp != 0
!(frontRunnerSetTimestamp != 0 && now.sub(frontRunnerSetTimestamp)
> frontRunnerPeriod)
```

After simplifying, we get:

```
forwardedOutcomeSetTimestamp != 0 &&
now.sub(forwardedOutcomeSetTimestamp) > challengePeriod
frontRunnerSetTimestamp != 0
frontRunnerSetTimestamp == 0 && now.sub(frontRunnerSetTimestamp)
<= frontRunnerPeriod
```

Again, the first term in the last equation is incompatible with the one before the last, so we eliminate it:

```
forwardedOutcomeSetTimestamp != 0 &&
now.sub(forwardedOutcomeSetTimestamp) > challengePeriod
frontRunnerSetTimestamp != 0
now.sub(frontRunnerSetTimestamp) <= frontRunnerPeriod
```

This situation can be interpreted as follows. Since “forwardedOutcomeSetTimestamp != 0”, the function “setForwardedOutcome” has been successfully called. However, it has been called more than “challengePeriod” ago. since “frontRunnerSetTimestamp != 0”, the “challengeOutcome” and/or “voteForOutcome” functions have been successfully called no earlier than “frontRunnerPeriod” ago.

Lemma 2: Once “isChallenged()” starts returning “true”, it never returns “false” anymore.

The condition “isChallenged” is equivalent to “frontRunnerSetTimestamp != 0”. The contracts starts its lifecycle with “frontRunnerSetTimestamp == 0”, and the only two assignments that exist are setting “frontRunnerSetTimestamp = now”. Assuming that “now” is never going to be 0, “isChallenged” is not going to switch back to “false”, once it became “true”.
QED (Lemma 2)

From Lemma 2, and the fact that the functions “setForwardedOutcome” and “challengeOutcome” have negation of “isChallenged” as precondition, and “voteForOutcome” has “isChallenged” as precondition, it is easy to see that it is not possible to transition from states 3, 4 back to state 1. It is important, because otherwise it would have allowed someone to change “frontRunner” back to any value by simply calling “challegeOutcome” again.

Resolution of the UltimateOracle

According to the diagram above, the two scenarios in which the UltimateOracle can stay unresolved are:

1) State (1) - first part, until someone successfully calls “setForwardedOutcome” or “challengeOutcome”
2) States (3,4) - but in order to extend this state, someone needs to call “voteForOutcome” with msg.value at least “challengeAmount + n*epsilon", where “n” - number of rounds of extension, and “epsilon” is the smallest amount possible.

Concerns about UltimateOracle

- It is possible to start challenging the outcome without waiting for the outcome to become known. While this is a useful feature for dealing with un-resolvable MajorityOracle-s, it might be a problem with other oracles. If the event has not occurred yet, the UltimateOracle attached to it can already have the outcome resolved ahead of time. It might be risky to content such resolution, because there is no anchor in the reality.
- If someone controls large amount of collateral tokens for a given UltimateOracle, there is a risk of winning the outcome simply by always putting slightly more collateralToken than currently winning outcome. This could deter small participants from using events with such Oracles, especially if a precent is created (this could be done for the purpose of a griefing attack on the Gnosis platform).
- In addition to the previous concern, it is possible to delay the resolution of the UltimateOracle for a very long time by playing the both parts of the "voting game", and placing a very small amount (1 wei) above the current frontRunner to extend the period. If not many other participants join in either side (out of fear of uncertainty of the outcome), it is relatively cheap to delay the resolution. Not sure of the consequences of this on the price dynamics for tokens, because some people will try to obtain complete sets to exchange them for collateral. Others will try to get rid of any outcome tokens.
- When creating an UltimateOracle, one needs to specify the collateralToken, which may be different from the collateralToken used by the corresponding event. Because they are named the same, it could be possible to trick people into believing that these two match when the event is

created. However, an UltimateOracle with a collateralToken completely controlled by an adversary means that the outcome is also controlled by the same adversary.

- SpreadMultiplier and maxAmount in the “voteForOutcome” function looks superfluous, because the restriction it brings can always be easily circumvented by calling the function multiple times. Note that maxAmount for an outcome does not change if someone calls “voteForOutcome” for that outcome, because in the expression for the maxAmount, “(totalAmount - totalOutcomeAmounts[_outcome]).mul(spreadMultiplier)”, both sides around the minus increase by the same amount
- It is possible to construct a collateralToken, with the amounts in the order of 2**128, that would cause the Math.mul() overflow in the “withdraw” function, therefore causing all the withdrawals to fail. It could be quite insidious because other functions would unlikely to be causing the same overflows.
- In the function “voteForOutcome”, there is a possible re-entrancy via “transferFrom” call of the collateralToken. It might not be directly via the collateralToken contracts, but by some more complicated implementation of token standard, like ERC-23. Not sure if it can be exploited in any way. The only thing I can think of is to keep “maxAmount” constant in the re-entrant call.

AbstractMarket.sol

The member fields are declared in the Market contract, not in the sub-contracts, presumably for the use in the market maker contracts. This is OK, but differs from the approach taken with the oracles, where the abstract oracle declares getter functions rather than public member fields.

StandardMarket.sol

fund

Noted that funding can only happen once, no top-ups. But failing to see the reason for such restriction.

close

Number of outcomes	Gas cost of creation of a market	Gas cost of closing a market	Gas cost of buy	Gas cost of sell	Gas cost of short sell
2	169 K	63.6 K	198 K	139 K	324 K
3	169 K	81.2 K	224 K	165 K	396 K
4	170 K	98.8 K	250 K	192 K	467 K
5	170 K	116 K	277 K	218 K	538 K
6	171 K	134 K	303 K	245 K	610 K
7	171 K	151 K	329 K	271 K	681 K
8	172 K	169 K	355 K	297 K	753 K
9	172 K	186 K	382 K	324 K	824 K
10	173 K	204 K	408 K	350 K	896 K
11	173 K	222 K	434 K	377 K	967 K
12	174 K	239 K	460 K	403 K	1.03 M
13	174 K	257 K	487 K	430 K	1.11 M

The table above shows approximate gas costs of creation and closing of markets. Because the gas cost of closing escalates much quicker than the cost of creation, it is theoretically possible (though it will require lots of outcomes) to create a market which cannot be closed due to gas cost limitation.

buy, sell

Both functions call into the “marketMaker” contract (“calcCost” and “calcProfits” functions), which in turn (based on the example of LMSRMarketMaker) call back into the market (though only into the constant functions). This circular referencing is usually considered a “code smell” and can be resolved by extracting the common part (member fields of the AbstractMarket) into a separate components. For example, the fields in the AbstractMarket can be moved into a MarketInfo contract, and StandardMarket essentially becomes a library, where MarketInfo instance is passed into corresponding functions as an argument.
It might also make sense to aggregate the “netOutcomeTokensSold” over all markets for the same event, so that LMSRMarketMaker is more accurate in its pricing.

Gas cost of both operation could be a concern. It appears that most of the cost comes from the functions “buyAllOutcomes” and “sellOutcomes” of the Event contract. Please see suggestions for reducing the gas costs for these functions above.

shortSell
This function takes “outcomeTokenCount” and “minProfit” parameters, then proceeds to “buyAllOutcomes” for the quantity of “outcomeTokenCount”. After “buyAllOutcomes”, the function proceeds to sell one of the outcomes, and within, it uses “sellAllOutcomes” function for the amount of “profit”. Effectively, the “profit” amount of collateralToken gets refunded. Suggestion: refactor this function to not have to buy more outcome tokens than needed, and then sell them again - this can lower the point at which the short selling can be done. Currently, this point is “collateralToken.balanceOf(msg.sender) >= outcomeTokenCount”, but it could be “collateralToken.balanceOf(msg.sender) >= outcomeTokenCount - profit)”. This should also reduce the gas cost of this method by avoiding a loop inside the “sellAllOutcomes” function.

LMSRMarketMaker.sol

As mentioned in the section on the StandardMarket, an improvement on the LMSRMarketMaker functionality would be access to the aggregated “netOutcomeTokensSold” for all the markets related to the same event. However, this could open up a possibility of markets affecting each other.

In the table below there are approximate gas costs for “calcCost” and “calcProfit” functions

Number of outcomes	Gas cost of calcCost/calcProfit
2	76.8 K
3	88.8 K
4	100 K
5	112 K
6	124 K
7	136 K
8	148 K
9	160 K
10	172 K
11	184 K
12	196 K
13	208 K

calcCost
Line 33: Math.ln of the number of the outcomes in the event - could be tabulated for most common number of outcomes, either in the code, or in the storage mapping.

getNetOucomeTokensSold
Outdated comment.

Math.sol

Cost of Math.ln function is approximately 28 K gas per call, and cost of Math.exp function is approximately 26 K per call.