Security Audit Report

Neutron Dex Model

Authors: Manuel Bravo, Ivan Gavran, Gabriela Moreira

Last revised 11 December, 2024

Table of Contents

Project Overview

In Q4 of 2024, as a part of security partnership with Neutron, Informal Systems set out to write a formal model of Neutron's Dex in Quint. Working closely with the Dex's development team, we captured the most important aspects of the Dex, defined the desired properties, and analyzed their validity.

While our analysis did uncover some violations of the desired properties, further investigation showed that they do not constitute a security threat to the Dex.

In this report we give the basics of the model and we detail the analysis results. The full model will be open-sourced in the near future.

1. Input

As a starting point to building the model, we used the following material:

- Public documentation of the Dex module, corresponding to the commit bfb0c31 of the docs repository
- System specification, written by the dev team specially for this modelling effort, in which the basic behavior and expected properties of the system are described. This document is dynamic and was developed simultaneously with modelling.
- The implementation of the dex, at commit `01b71f7` and, in the second phase, at release v5.0.0-rc0. While our explicit goal was to avoid relying too much on the implementation details, sometimes it was necessary in order to understand e.g, the rounding of rationals, which was influencing whether the behavior was correct or erroneous.

2. The Quint Model

Each Quint model is conceptually a state machine whose evolution is defined by its _initial state_ and the set of _transitions_ from a current to the next state.

In this model, we opted for having a single variable called `state`, that captures all the relevant aspects of the system.

Let us first see how the state of the system looks like:

State

```
type State = {
    tranches: TrancheKey -> Tranche,
    tranchesShares: TrancheShares,
    pools: PoolId -> Pool,
    poolShares: PoolShares,
    // balances of all the users in the system
    coins: Coins,
    blockNumber: BlockTime,
    // helper state variables, to be used in invariants
    bookkeeping: TrackedValue
}
```

Fields `tranches` and `tranchesShares` contain the information on the state of the tranches and users' ownership in them. The same for pools is achieved by `pools` and `poolShares`. The field `coins` keeps track of the state of users' funds, and `blockNumber` tracks the blocks. Finally, `bookkeeping` contains helper variables, that will be used for inspecting desired properties (e.g., total value deposited by a user).

Each field has a type. As an example, `pools` is a mapping from `PoolId` into `Pool` type, which are defined by

```
type PoolId = {
    tokenPair: (Token, Token),
    // Pool's tick is the token0 sell tick, that is: 1 Token0 = price(tick)* 1 Token1
    tick: TickIndex,
    fee: Fee
}

type Pool = {
    reserves: (int, int),
    shares: int
}
```

Each pool contains the information about its reserves (a pair of two integers) and total existing shares (an integer). Check the other types in types.qnt.

Model Evolution: Initial State

The initial state is very simple, with nothing in pools or tranches, and with some coins amount assigned to each user:

```
pure val initState: State = {
    tranches: Map(),
    coins: tuples(CREATORS, TOKENS).mapBy(_ => INITIAL_CREATOR_BALANCE),
    tranchesShares: Map(),
    blockNumber: 0,
    poolShares: Map(),
    pools: Map(),
    bookkeeping: emptyTrackedValue
}

action init = {
    state' = initState,
}
```

Model Evolution: Transition

Possible transitions are defined by the action predicate `step`:

```
action step: bool = any {
    placeLimitOrderAct,
    cancelLimitOrderAct,
    withdrawLimitOrderAct,
    withdrawPoolAct,
    depositAct,
    singlehopSwapAct,
    advanceTimeAct,
}
```

In short, `any` of the seven listed things may happen: a user can place a new limit order, or withdraw or cancel from an existing one; a user can deposit into a pool or withdraw from an existing one; a user can swap using available liquidity; or a time may progress.

As an illustration, let's look at the `withdrawPoolAction` (called directly by `withdrawPoolAct`):

```
action withdrawPoolAction(processResult: (Result, Message) => bool): bool = all {

    // a cross-product of creators and pools in which they have >0 share
    val relevantCreatorsAndPools = tuples(CREATORS, state.pools.keys()).filter(((c,
p)) => state.poolShares.get(c).getOrElse(p, 0) > 0)
    all {
        require(relevantCreatorsAndPools != Set()),

        nondet creatorAndPool = relevantCreatorsAndPools.oneOf()
        val creator = creatorAndPool._1
        val poolKey = creatorAndPool._2
        nondet withdrawAll = oneOf(Set(true, false))
        val userShares = state.poolShares.get(creator).get(poolKey)
        nondet partialWithdrawAmount = oneOf(1.to(userShares))
        val amountToWithdraw = if (withdrawAll) userShares else partialWithdrawAmount
        val msg = {
            creator: creator,
```

```
            tokenPair: poolKey.tokenPair,
            sharesAmount: amountToWithdraw,
            tick: poolKey.tick,
            fee: poolKey.fee
        }
        val result = withdrawPool(state, msg)
        state' = res.state
    }
}
```

The action non-deterministically defines which exact message should be sent. Let's examine it part-by-part.

The line `require(relevantCreatorsAndPools != Set())` requires that there are some users with non-zero shares in pools, from which they could withdraw. If there are none, this action may not be executed (and is removed from a set of actions to choose from in the `step`).

The `creatorAndPool` is a non-deterministic (`nondet`) value obtained by choosing `oneOf` the elements from those users and corresponding pools.

The `nondet` variable `withdrawAll` is a slight optimization: we want to bias the action to occasionally withdraw all the shares (as we privilege this choice over choices with all possible share values). Thus, if `withdrawAll` is true, we will withdraw all existing shares, and otherwise a `nondet` value `partialWithdrawAmount`.

The message `msg` is formed with all the choices, and executed on the current `state`. Finally, the `state` variable is updated with the result of the execution. (`state'` means "the variable state after the transition".)

Transitions Business Logic

While the action defines the choices on what to execute, the whole business logic is contained in the function `withdrawPool` (and similar), which can be found in dex.qnt.

They all share the same signature, transforming the pair `(State, Message)` into (the next) `State`. In that regard, all these functions are pure: there is no non-determinism and no accessing the `state` variable: all of this is done in the transition actions described above.

Modeling Choices and Simplifications

In the model, we made the following simplifying choices:

1. We did not model a multi-hop swap. Instead, we achieve it by consecutive applications of a single-hop swap.
2. We did not add the feature of the user limiting the maximum amount to receive from the placed limit order message. This is a nice UX feature, but is not security-critical.
3. In the model, users always specify price ticks instead of direct prices (this choice has to do with technical limitations of Quint, the lack of the `logarithm` function).

3. Properties of Interest

In this section we describe properties we encoded in the model and checked their validity.

They can be divided into state properties (expressed through predicates on a single state) and transition properties (expressed through predicates on two subsequent states).

State Properties

1. [DEPOSIT-PROFITS] For a user who has deposited to the pool `p` and who has withdrawn all their shares from `p`, the following holds:

- If there were swaps through `p`, the user has withdrawn more value than deposited
- If there were no swaps through `p`, the user has withdrawn the same value as deposited
- The user withdrew no less value than deposited.

Note: In case the autoswap fee needed to be paid, the initial deposited value is considered the one after subtracting the deposit fee.

The Quint encoding of the property can be seen in `baseDexEvolution.qnt::poolProfitInv` and looks like this:

```
val poolProfitInv: bool =
    state.bookkeeping.pools.deposits.keys().forall(((creator, poolId)) =>
        // If the user has withdrawn all their shares
        state.poolShares.get(creator).get(poolId) == 0 implies
            val totalDeposited =
state.bookkeeping.pools.depositValues.getOrElse((creator, poolId), 0)
            val withdrawn = state.bookkeeping.pools.withdrawals.getOrElse((creator,
poolId), (0, 0))
            val swaps = state.bookkeeping.pools.swaps.getOrElse((creator, poolId),
Set())
            val totalWithdrawn = computeDepositValue(withdrawn,
poolId.tick).truncate()
            val tol = TOLERANCE * computeUnitTolerance(poolId.tick)
            // If there were swaps, the user withdrew more value than deposited
            if (swaps != Set()) totalWithdrawn + tol >= totalDeposited
            // Otherwise, the user withdrew no less value than deposited
            else abs(totalWithdrawn - totalDeposited) <= tol
    )
```

This predicate requires that `forall` deposits, in case the user has withdraws all their shares, it `implies` a relation between `totalWithdrawn` and `totalDeposited`, depending on whether or not there were swaps.

2. [NO-LOSS-ON-PLACED-TRANCHE] A user who has placed a limit order (the remaining maker part, post-swap) cannot lose value with respect to the specified `sellingPrice`.

Once the user has withdrawn all their shares, be it through a (sequence of) withdrawal or a cancellation-induced withdrawal, the user is at no loss (with respect to the price set by `sellTick` specified by the user). See the predicate at `baseDexEvolution.qnt::noLossOnExhaustedTranches`.

We also checked for a sequence of other properties that can be viewed as sanity checks, whose violation would imply that there are potential problems to explore:

3. [RESERVES-IMPLY-SHARES] If a pool has some positive amount of reserves, it also has a positive amount of shares.
4. [COINS-CONSTANT] The total amount of coins in the system (tranches, pools, users' coins) is invariant.
5. [NONEGATIVE-AMOUNTS] All amounts of pool shares and reserves, tranches shares and reserves, and user coins are non-negative.

Transition Properties

Transition properties describe the relation between a state `before` and a state `after` a transition happened, and their signature is `(State, Message, State): bool`.

All transitions are inspected for the state changes as described in the specification. Furthermore, there are specific properties to be expected after transitions.

7. [SWAP-TRANSITION] When a user swaps using a `SinglehopSwapMsg` message:

- Their limit price is honored.
- The value of existing pools either increases corresponding to the fee, or remains the same (if the pool did not take part in the swap).
- The value of all tranches remains the same.

8. [LIMIT-ORDER-TRANSITION] When a user swaps using a `PlaceLimitOrderMsg` message:

- Their limit price is honored.
- The value of existing pools either increases or remains the same.
- The value of all tranches remains the same, except for the tranches corresponding to the tick, which may increase in value due to the maker part of the limit order message.

9. [DEPOSIT-TRANSITION] When a user deposits to a pool:

- The value of existing pools either increases (for the pool to which the user deposits) or remains the same.
- If the deposit is provided in a different ratio than the existing ratio of the pool, a fee for the swap (to reach the existing ratio) is paid.
- [POOL-RATIO-CONSTANT] If the autoswap option is set to false, the ratio in the pool does not change when performing (non-initial) deposits.
- [SHARES-FOR-DEPOSITS] If a user successfully deposits positive amount of coins (at least one in the pair), they will get in return a positive amount of shares.

10. [CANCEL-TRANSITION] When a user cancels a tranche:

- No shares of other users are affected.
- The user received the pro-rata shares of any unused maker denom and of all the swap proceeds.

11. [TRANCHE-WITHDRAW-TRANSITION] When a user withdraws from a tranche:

- The user gets the pro-rata portion of all the swap proceeds not yet claimed (since the last withdrawal)

12. [POOL-WITHDRAW-TRANSITION] When a user withdraws from a pool:

- The user gets the specified portion of the value they provided, increased by their part of the profits from all the swaps occurring since the last withdrawal.

4. Analysis Results

After developing the model, we have used it to analyze the properties of the system. Any violation to the property from the model called for closer inspection. Some violations pointed to discrepancies between the model and the input, some to discrepancies between the model&input and the implementation, and some to actual violations of the properties in the implementation.

Violated Properties

Below, we list the properties for which the simulation found violations and discuss the relevant context and consequences.

1. [DEPOSIT-PROFITS]:

The reason for why this property was violated has to do with rounding when withdrawing. Repeated rounding may make it happen so that the loss grows with the number of steps (transactions).

An example run is given in `violationRuns.qnt::noLossViolationRoundingRun`.

Note that here we are talking about differences in dozens of **micro**-tokens. Thus, this violation is considered acceptable by the dev team. In order to disregard rounding errors, we have phrased a similar property that included tolerance for rounding errors:

1.a [NO-LOSS-TOLERANCE] Assuming that we allow for the tolerance proportional to the number of swaps and the value of a pair of unit tokens (to counter off-by-one errors), the user withdraws no less value from the pool than deposited.

However, even this property does not hold.

The problem occurs when the total value of reserves becomes much larger than the total number of shares. When withdrawing all their shares from the pool, the user receives `userShares * (totalPoolValue / totalPoolShares)`. Assuming an off-by-one error in `userShares`, the max possible error is `0.99 * (totalPoolValue / totalPoolShares)`.

One way in which `totalPoolValue` and `totalPoolShares` can substantially diverge in a few steps is the following:

a. Bob deposits `(smallAmount, 0)` in the pool with a very large fee.

b. Alice deposits `(0, largeAmount)` with (default) autoswap enabled to the same pool.

This translates into calculating shares as if Alice has first converted the whole `largeAmount`. Thus, Alice receives very few shares, and the total amount of shares is much smaller than the pool value (because the majority of the value is provided by the autoswap fee).

c. Alice withdraws all of her shares, and receives a much smaller value than what she deposited (accounting for the fee paid for the swap).

d. Bob withdraws all of his shares, and receives, beside his deposited part and the Alice's fee paid, also the rest of Alice's funds.

An example run is given in `violationRuns.qnt::noLossViolationAutoswapRun`.

A potential attack is a user front-running any extremely large deposit (bigger than the pool's size) by changing the pool's ratio so as to be able to take advantage of the described behavior. Alternatively, and much more lucrative, is a user front-running any initial deposit to a new pool.

NOTE: This particular violation seems to be stemming from the assumption that every pool's value will be significantly larger than the deposit value. Indeed, the calculation for the autoswap fee starts with the goal of

bringing deposit amounts to match the ratio of the pool, where sometimes it would be easier to do it the other way round.

Discussing this issue with the dev team, they informed us they were aware of the issue and were planning to address it by runtime monitoring, especially in the initial phase of pool creation.

2. [POOL-RATIO-CONSTANT] Repeated rounding can change the ratio of the pool significantly (even without autoswap disabled).

In the `violationRuns.qnt::autoswapIssueRun`, we give an example in which the ratio starts at 0.0136 and is then changed into 0.0055 within 4 steps.

We discussed the issue with the development team and concluded that this violation does not constitute an attack surface, since no free-swap is possible with it.

3. [NO-LOSS-ON-PLACED-TRANCHE] The violation again has to do with the rounding problem when calculating the value to withdraw.

This leads to situations in which it is possible for a tranche to have some funds in it, without anybody having shares in the tranche to ever withdraw from it. While the kinds of errors are off-by-one in terms of a token, depending on the price of the maker and the taker, they can be large in terms of the value of the pool.

For an example, see the `violationRuns.qnt::noLossOnTranchesViolationRun` run.

For an equivalent property [NO-LOSS-ON-PLACED-TRANCHE], which tolerates rounding errors multiplied by number of steps, we found no violation.

Moreover, many other properties we checked had violations on their respective versions without tolerance, and we experimented and reasoned about rounding in order to be able to set up tolerances that were enough to make properties hold under different numbers of steps. This was the main struggle of property checking in this project, as many of our long-running experiments would result in violations which, upon debugging, turned out to be yet another not-accounted-for rounding problem. However, the fact that simulation always found violations on the properties that didn't have tolerance is a strong indication that it's coverage is sufficient for these properties.

In the next section, we discuss what kind of confidence we can get about the overall functioning of the system based on these simulation results.

Model Simulation: Size of the Input Space

Once a model is developed, there are two main ways in going about checking the properties of it. The first one is by doing bounded model checking---a way to inspect *all possible behaviors up to a certain length*.

With Quint, model checking can be done either using a symbolic model checker Apalache, which encodes the whole model as a logical formula and uses a solver to solve it; or using an enumeration-based model checker TLC, which explores all possible states of the model evolution in a breadth-first manner.

Unfortunately, the model of the DEX, which captures different actions and exact computations in them, could not be meaningfully checked either with Apalache or with TLC. On top of that exponential growth of the state space, the computation involves exponentiation of rationals. All that combined made model checking of this model intractable.

To give the idea of the complexity of the input space.

- At every step, we choose between 7 actions. For each of them, we also have to non-deterministically choose parameters. In model checking, non-deterministically means that all possibilities are accounted for.
- If, for every parameter, (ticks, fees, amounts, token, users, etc) we pick from a set of only 2 values (severely constraining the model), it is still too big of a state space.
- Considering how many parameters we have for each action of the 7 actions (some of them have more parameters than others):
    - 2^2 + 2^7 + 2^7 + 2^4 + 2 + 2 + 2 = 298
    - This means, for each step in this scenario, we'd have 298 possible transitions to make

- If we want to model check the model of executions of up to 4 steps, this would have to check 298^4 = 7886150416 possible states

Our intuition is that restricting the model so much in order to get a barely manageable model-checking is not justified, since

- the chances a bug can be reproduced only using the 2 possible values we pick for each parameter
- the chances a bug can be reproduced with 4 steps

are fairly low.

The second option to inspect the model is by performing many random simulations of the model evolution. Simulation is a depth-focused search of the state space, compared to the breadth-first approach of model checking. While random simulations do not provide guarantees about covering the whole (bounded) state space, they enable inspecting much longer traces. Furthermore, with some small interventions into the model, we can guide the simulator to explore the behaviors of interest. Thus, simulation was our method of choice for checking the model of the Dex.

While it is more productive to explore larger depths (thus: more interactions) than insist on covering the whole breadth (thus, sacrificing some number of choices), it still makes sense to reduce the breadth as much as possible to hit the most interesting scenarios.

Model Simulation: Shrinking the State Space

To achieve the goal of reducing the state space breadth without sacrificing the bugs our model can find, we tweaked the following aspects of the model:

- There is a tension between choosing from large number of ticks (to cover all possible numerical edge cases) and focusing generated scenarios towards dense interplay between tranches and pools. (Otherwise, we may just get many almost independent actions.)

Our solution was to non-deterministically pick 3 ticks **at the beginning of each simulation**. This allowed for density of interactions within a single simulation, while exploring the whole space of ticks among many simulations.

- There are some privileged parameter-choices of the actions. For example, a liquidity provider may choose to withdraw any number of their shares from a pool (expressed by `oneOf(1.to(userShares)`). However, the most interesting scenarios happen when the user withdraws all their shares. In a default setup, this would happen too rarely. Thus, we add a Boolean `nondet withdrawAll = oneOf(Set(true, false))` which flips a coin to decide whether to withdraw all shares, or some amount of shares.
- We also ran some tests that were not completely free ranging. Phrased as runs, they were given a *blueprint* of the simulation: what sequence of actions needs to happen first, followed by some random choice of actions, and then again followed by predetermined action.

With each violation we found, we inspected it, documented if it were a novel one, and modified the model to avoid the paths leading to the violation. Once we reached a stable state (violations not being found in our typical 100000 run simulation), we ran a final experiment consisting of 300000 simulations of 50 steps each. This run found no novel violation.

To assess the quality of the generated traces, we have created predicates describing interesting scenarios and then monitored the executions to track how often these interesting predicates were true.

- A tranche was exhausted through cancellation in almost all 300k traces.
- A tranche was exhausted through a series of withdrawals in ~45k traces.
- A single tranche was shared by multiple users in ~36k traces.
- A single swap could potentially use liquidity from both a tranche and a pool in ~100k traces.

Descriptively, we saw among the inspected scenarios interactions of swaps with both pools and tranches, as well as multiple users placing their orders, withdrawing proceeds repeatedly and cancelling their placements.

Another modification that enabled us to explore more breadth was projecting the model onto only a part of the available actions. This modification comes from the insight that for a certain kinds of bugs the exact numbers do not matter. Rather, what matters are relations between numerical choices.

For instance:

- for a swap to succeed, the selling price must be lower than the buying price (and concrete number choices do not matter),
- for the interaction of a tranche placement, withdrawal, and cancellation, the interactions with different tranches does not matter, only actions relevant to the single tranche matters.
- for the interaction of pool deposits, withdraws, and swaps; again, only a particular pool matters.

Thus, some of the interesting properties may be examined by factoring out only parts of the model.

In such setup, the simulation may provide good coverage:

When checking only for pools' properties, we choose among three actions: `SinglehopSwap`, `Deposit`, `WithdrawPool`, of a fixed pool key. With the previous optimizations, `WithdrawPool` can be seen as two actions, `WithdrawPoolFull` and `WithdrawPoolPartial`, and similarly for the `SinglehopSwap`. Our hypothesis is that these 5 actions cover all interesting behaviors with respect to pools. An equivalent decomposition holds for tranches' properties.

In such a restricted setup, where all reorderings of actions can be achieved with ~6000 different choices of actions (a size of all permutations of a set of size 6), it becomes more likely that the simulation produces an interesting behavior. The complexity of parameter choice and calculations remains, and makes the challenge difficult for model checking. Running simulations in such a factorized model did not find novel violations.

5. Conclusion

We have created a Quint model of the Neutron DEX. The model captures the essential properties of the DEX, including numerical considerations.

We have used the model to analyze the properties of the system. In the process, we have uncovered violations of the desired properties, as described in section 4. Analysis Results . We discussed the violation with the development team and concluded that all but one of the uncovered violations do not present a security concern. The remaining violation that could present a danger is handled by runtime monitoring.

We did our analysis by running a suite of simulations, that inspected many interesting behaviors of the system. Due to the complexity of the system (and hence the model), we were not able to check the whole model and obtain a guarantee of every possible behavior being inspected.

The simulations run can be compared to the integration tests from the codebase, with two major differences:

1. We ran the order-of-magnitude of 100000 simulations, with different parameters and action orderings, compared to a couple of dozens of integration tests.
2. The simulations do not test the actual code, that integration tests do.

Another important benefit of this model is that it brings clarity with respect to the expected behavior of the system. In the process of developing the model, we have discovered many misleading things in the documentation and specification, which can stem from staleness, typos, or attempting to simplify the explanation for the audience to follow more easily. With the formal model, it gets much less likely to introduce a mistake due to the model's internal consistency checks.

Thus, one direction of future work may be to create a new, dynamic documentation: it could use executable Quint in the explanations and examples, enabling users to try out their own examples (rather than relying on the provided, constant examples). Another exciting direction is using the developed model to write integration tests for the implementation, establishing a firmer connection between the model and the code, and increasing the test coverage with as many generated tests as needed.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.