Dexter

Release 1.0

Wojciech Klaudiusz Zaborowski

Apr 19, 2022

CONTENTS

Dexter is a platform for simulating several DEX variants. Dexter is implemented in Anylogic.

CHAPTER

ONE

CONTENTS

1.1 01. Introduction

1.1.1 Motivation

Decentralized exchange (or “DEX” in short) is an exchange running on a blockchain. While DEXes are a hot topic over last years, so far there is no well established “standard design” of a DEX. Rather, attempting various DEX designs happens to be a field of active research around the world.

Onomy project is one of such attempts. Dexter was build as a research-level tool for supporting internal Onomy research. The idea was to create a laboratory environment where several variants of DEX design can be tested.

1.1.2 Goals

1. Implement a general framework for DEX designs.
2. Allow pluggable implementations of “executor” (the core part of DEX, where orders are executed).
3. Implement a behaviour of traders, so that the simulation will cover both DEX itself and the surrounding trading traffic.
4. Establish rich set of metrics, so that performance of different DEX variants can be comparatively measured.
5. Visualize a running DEX with rich collection of charts and stats, so that team members can get a better insight into the internals of DEX design.
6. Provide a command-line interface and a small scripting language, so that the simulator can be also used as a testing environment (where a collection of test cases written as scripts is maintained).
7. Simulate the blockchain layer - at least up to the point of having explicit control on network and consensus delays.

1.1.3 Key design decisions

1. Use Anylogic as the main development platform.
2. Use agent-based simulation as the simulator construction paradigm.
3. Maintain a separate JVM-based library for data model and core business logic.

1.1.4 TLA+ specification

Dexter is based on a formal DEX model specification in TLA+. The specification can be found here:

https://github.com/onomyprotocol/specs/

1.2 02. Technological context

In this chapter we do a short overview of relevant technologies - either used in the simulator or related to the simulated solutions.

1.2.1 Blockchain

A peer-2-peer network of computers materializing together an idea of consensus-based **virtual computer**. The consensus protocol protect this virtual computer against hacking attacks (as opposed to physical computers that can always be hacked).

The operations of this virtual computer are called **transactions**. The history of all transactions executed on the blockchain virtual computer since its launch is what is called the "shared ledger". The ledger is typically partitioned into **blocks**, hence it can be seen as a sequence of blocks.

Examples of (public) blockchains: Bitcoin, Ethereum, Cardano.

1.2.2 Cryptocurrency

The idea of money re-established on top of a blockchain.

Examples of cryptocurrencies: BTC,, ETH, DOT, USDC.

1.2.3 Decentralized exchange (DEX)

Know also as "DEX". This is just an exchange (i.e. a market) operating on a blockchain, where cryptocurrencies can be traded.

Can be seen as blockchain-based analogy of Forex (however Forex is a relatively homogenous solution in terms of orders execution mechanics, while every DEX may be based on a completely different algorithm).

A primordial example of a DEX is Uniswap: https://uniswap.org/

1.2.4 Discrete event simulation (DES)

A method of building simulators centered around the priority queue of events. Every event has a timestamp and the queue is ordered by these timestamps. Events are processed sequentially. Processing of an event E with timestamp T can produce arbitrary number of new events with timestamps bigger or equal T. All these events are then enqueued to the central events queue.

This technique effectively materializes "virtualization of time", i.e. allows accurate simulation of timing while optimally utilizing resources of the computer running the simulation.

In lame terms, DES allows simulating months of operation of target system within couple of hours of your computer (provided your computer is fast enough).

https://en.wikipedia.org/wiki/Discrete-event_simulation

1.2.5 Agent-based simulation

A method of building DES simulators, where components of the simulated system are represented as message-passing entities. This technique can be seen as a merger of actor model and DES.

https://en.wikipedia.org/wiki/Agent-based_model

1.2.6 JVM

Java Virtual Machine. This is the cornerstone of Java Platform. Collection of programing languages compiled to JVM is quite large and includes (among others): Java, Groovy, Kotlin, Scala, Clojure.

JVM languages usually allow interoperability with Java, which is considered the “root” language of JVM.

JAR is the packaging format on JVM. A library written for JVM is distributed as JAR file.

1.2.7 Anylogic

A java-based development platform for building agent-based simulators.

https://www.anylogic.com/

1.3 03. Quick start guide

In this chapter we summarize steps required to run Dexter on your computer.

1.3.1 Step 0: Supported platforms and hardware requirements

Officially, Anylogic claims to be tested and supported on these platforms:

- Microsoft Windows 10, x64, Internet Explorer 11
- Apple macOS 12.0.1 (Monterey), Universal, Safari 9+
- Ubuntu Linux 18 and 20, x64 (with installed GTK+, libwebkitgtk-1.0-0, libudev, libssl), Firefox 24+
- Linux Mint 17, x64 (with installed GTK+, libwebkitgtk-1.0-0, libudev, libssl), Firefox 24+

According to our own testing, we had positive results on:

- Windows 10
- MacOS Bis Sur, running on Intel-based macbook

However testing on Linux revealed some severe problems (GUI crashing). Therefore we do not recommend running Dexter on Linux.

The situation with M1-based Apple computers is also not clear (as of January 2022 java platform is not yet officially supported on M1 Macs).

Hardware requirements

Dexter was developed and tested on a computer with the following capabilities:

- 32GB of RAM
- CPU computing power (as measured with Geekbench 5): single-core-score=597, multi-core-score=2346
- display with resolution 2560x1440

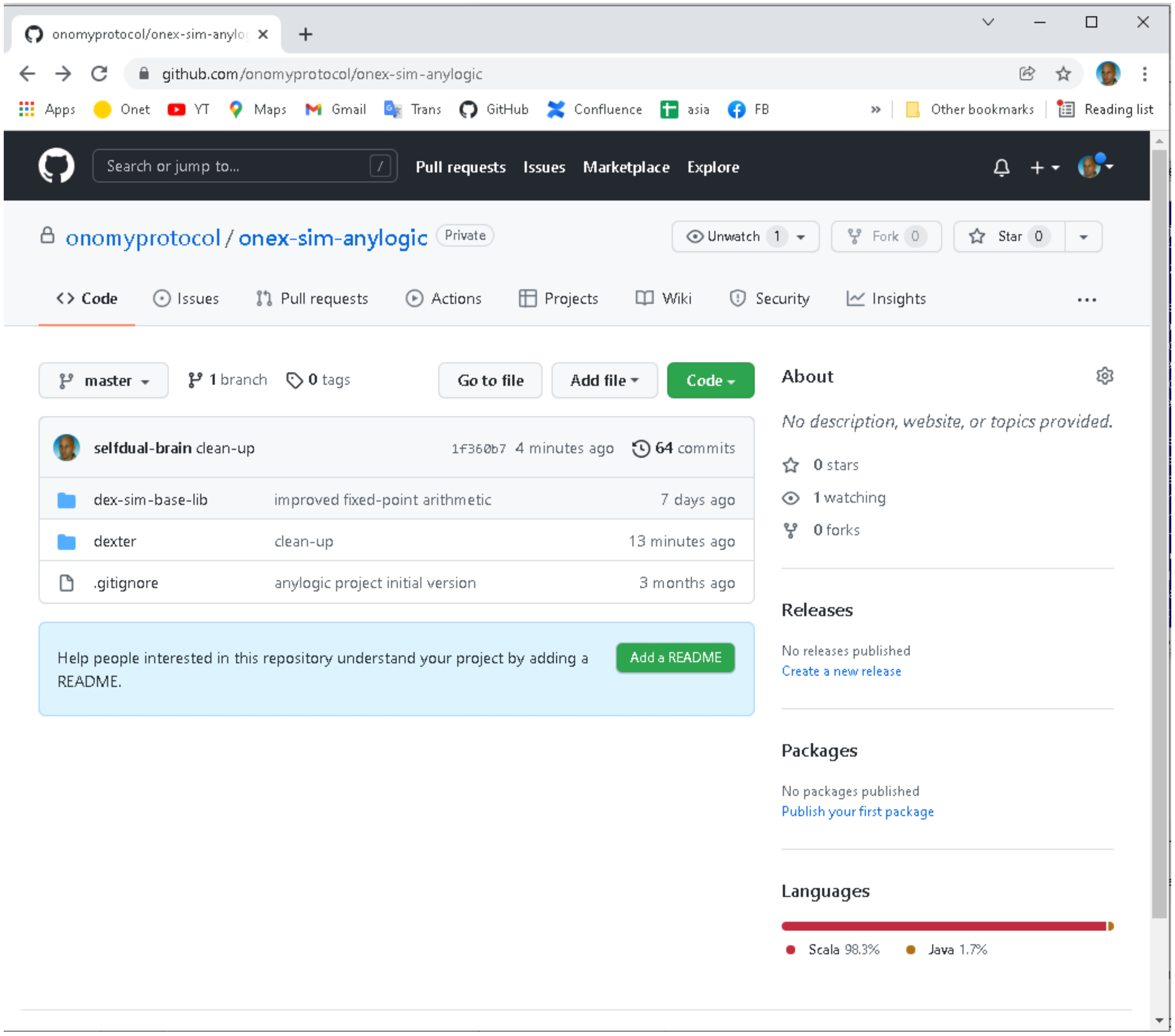
Running Dexter on a machine with lower capabilities can be uncomfortable.

Caution: running really long simulations may require way bigger RAM amounts.

1.3.2 Step 1: Download Dexter

Go to this Github repository:

https://github.com/onomyprotocol/onex-sim-anylogic



selfdual-brain clean-up 1f360b7 4 minutes ago 64 commits

dex-sim-base-lib	improved fixed-point arithmetic	7 days ago
dexter	clean-up	13 minutes ago
.gitignore	anylogic project initial version	3 months ago

Help people interested in this repository understand your project by adding a README.

Clone the repo (this step requires basic familiarity with GitHub interface). Then, find Dexter executables in **dexter** directory. There are 2 files there:

- `dexter.alp` - this is Anylogic "project" file of the simulator
- `lib/dex-sim-base-lib.jar` - this is the library used by simulator

1.3.3 Step 2: Download and install Anylogic

Anylogic can be downloaded from manufacturer's website:

https://www.anylogic.com/downloads/

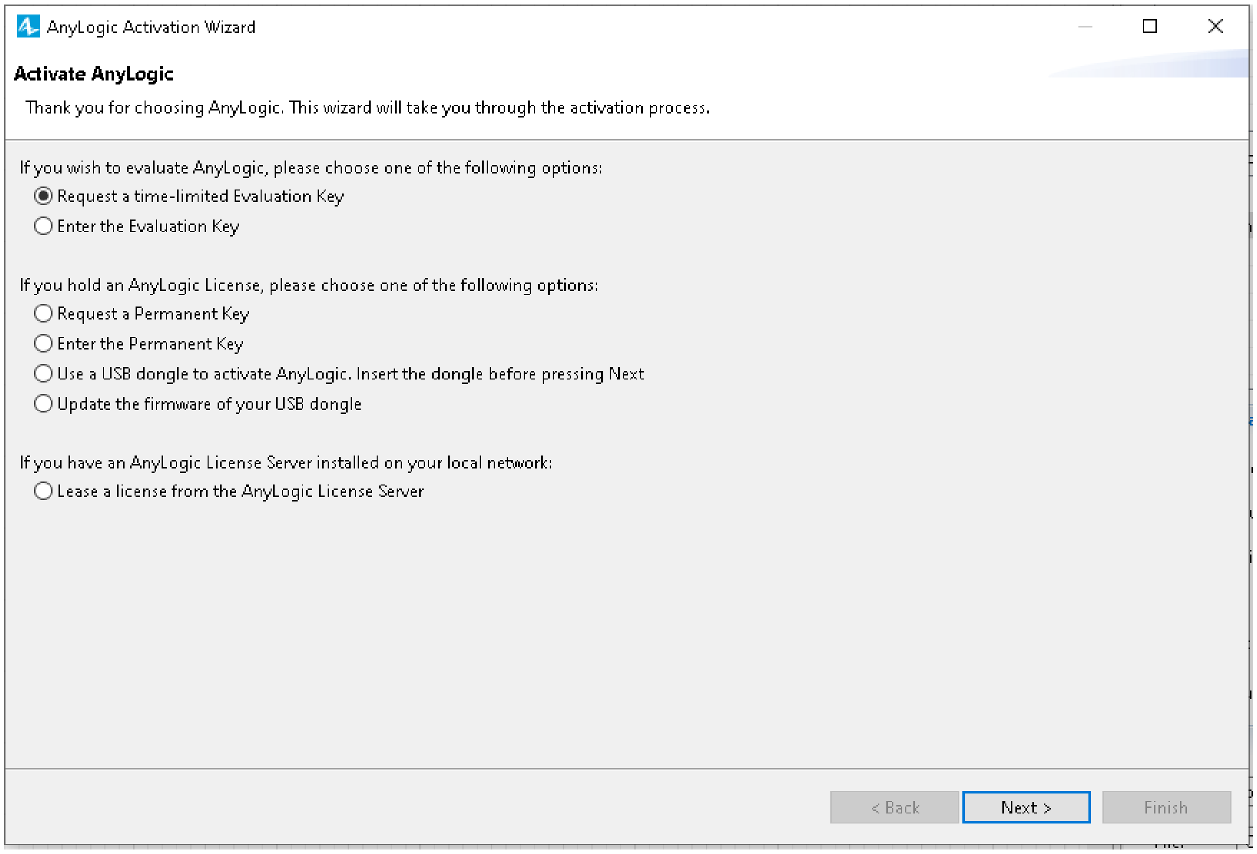
Pick the version compatible with your platform.

Dexter was tested on Anylogic version 8.7.9.

1.3.4 Step 3: Obtain Anylogic license

Anylogic is a commercial product. You must purchase a license to run it.

Alternatively it can be run on evaluation license. Connecting the paid license or requesting the evaluation license both are available inside the app - just launch Anylogic and then go to **HelpActivate product** in the main menu.


AnyLogic Activation Wizard

Activate AnyLogic

Thank you for choosing AnyLogic. This wizard will take you through the activation process.

If you wish to evaluate AnyLogic, please choose one of the following options:

- [✓ Request a time-limited Evaluation Key]
- Enter the Evaluation Key

If you hold an AnyLogic License, please choose one of the following options:

- Request a Permanent Key
- Enter the Permanent Key
- Use a USB dongle to activate AnyLogic. Insert the dongle before pressing Next
- Update the firmware of your USB dongle

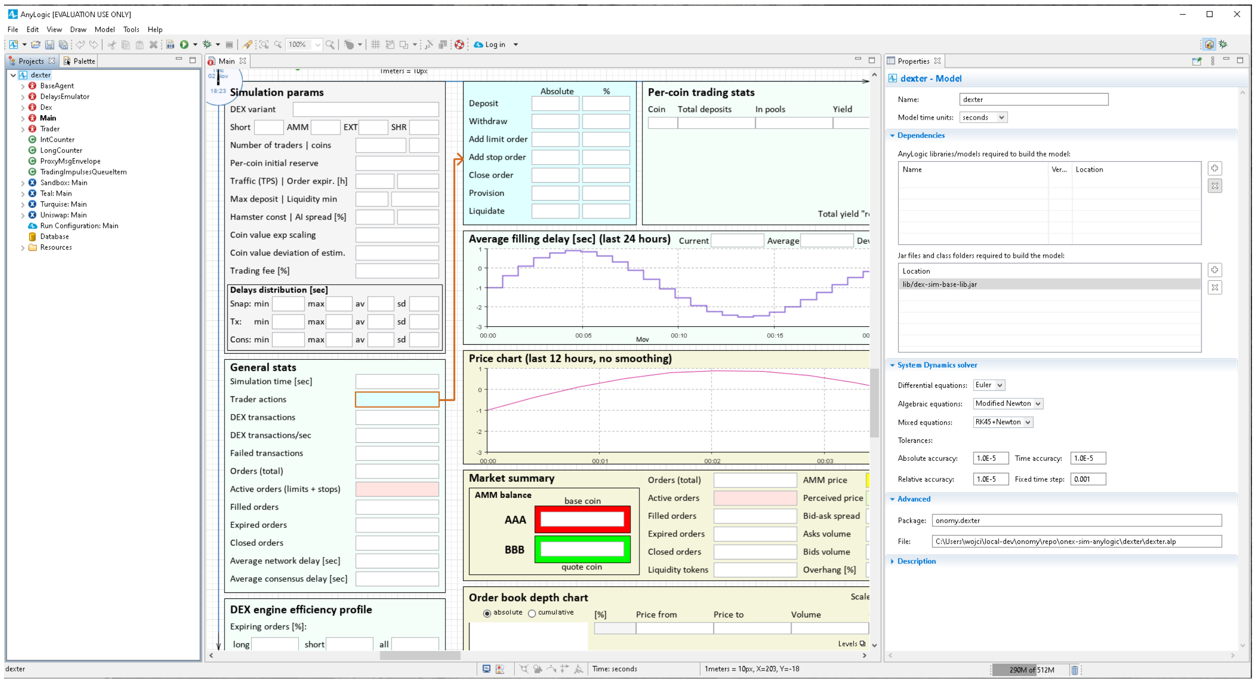
If you have an AnyLogic License Server installed on your local network:

- Lease a license from the AnyLogic License Server

1.3.5 Step 4: Open Dexter project

1. Launch Anylogic.
2. Close the blue "welcome" window (check the "do not show this window in the future" checkbox).
3. Go to **File/Open** in the main menu.
4. Point to **dexter.alp** file downloaded in step 1.

Once the Dexter project is successfully opened in Anylogic, you should see this:



Properties – dexter - Model

Name: dexter
Model time units: seconds

Dependencies

AnyLogic libraries/models required to build the model:

Name	Ver...	Location

Jar files and class folders required to build the model:

Location
lib/dex-sim-base-lib.jar

System Dynamics solver

Differential equations: Euler
Algebraic equations: Modified Newton
Mixed equations: RK45+Newton
Tolerances:
Absolute accuracy: 1.0E-5 Time accuracy: 1.0E-5
Relative accuracy: 1.0E-5 Fixed time step: 0.001

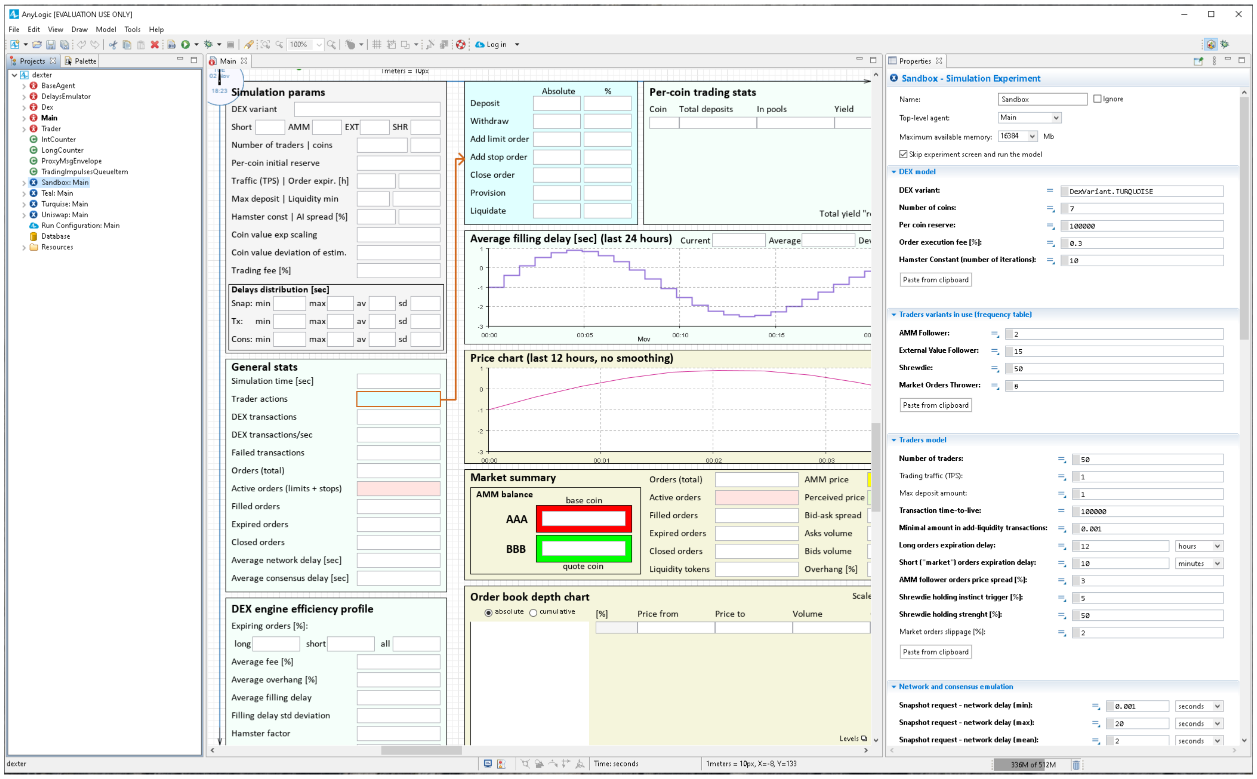
Advanced

Package: onomy.dexter
File: C:\Users\wojci\local-dev\onomy\repo\onex-sim-anylogic\dexter\dexter.alp

Description

1.3.6 Step 5: Configure a simulation experiment

In the project tree (left side of the window) find nodes with blue “X” icon. These are defined simulation experiments. Selecting one of them (mouse click) will display the configuration of given experiment (visible in the rightmost pane):



Properties – Sandbox - Simulation Experiment

Name: Sandbox Ignore [ ]
Top-level agent: Main
Maximum available memory: 16384 Mb
[✓] Skip experiment screen and run the model

DEX model

DEX variant: DexVariant.TURQUOISE
Number of coins: 7
Per coin reserve: 100000
Order execution fee [%]: 0.3
Hamster Constant (number of iterations): 10

Traders variants in use (frequency table)

AMM Follower: 2
External Value Follower: 15
Shrewdie: 50
Market Orders Thrower: 8

Traders model

Number of traders: 50
Trading traffic (TPS): 1
Max deposit amount: 1
Transaction time-to-live: 100000
Minimal amount in add-liquidity transactions: 0.001
Long orders expiration delay: 12 hours
Short ("market") orders expiration delay: 10 minutes
AMM follower orders price spread [%]: 3
Shrewdie holding instinct trigger [%]: 5
Shrewdie holding strenght [%]: 50
Market orders slippage [%]: 2

Network and consensus emulation

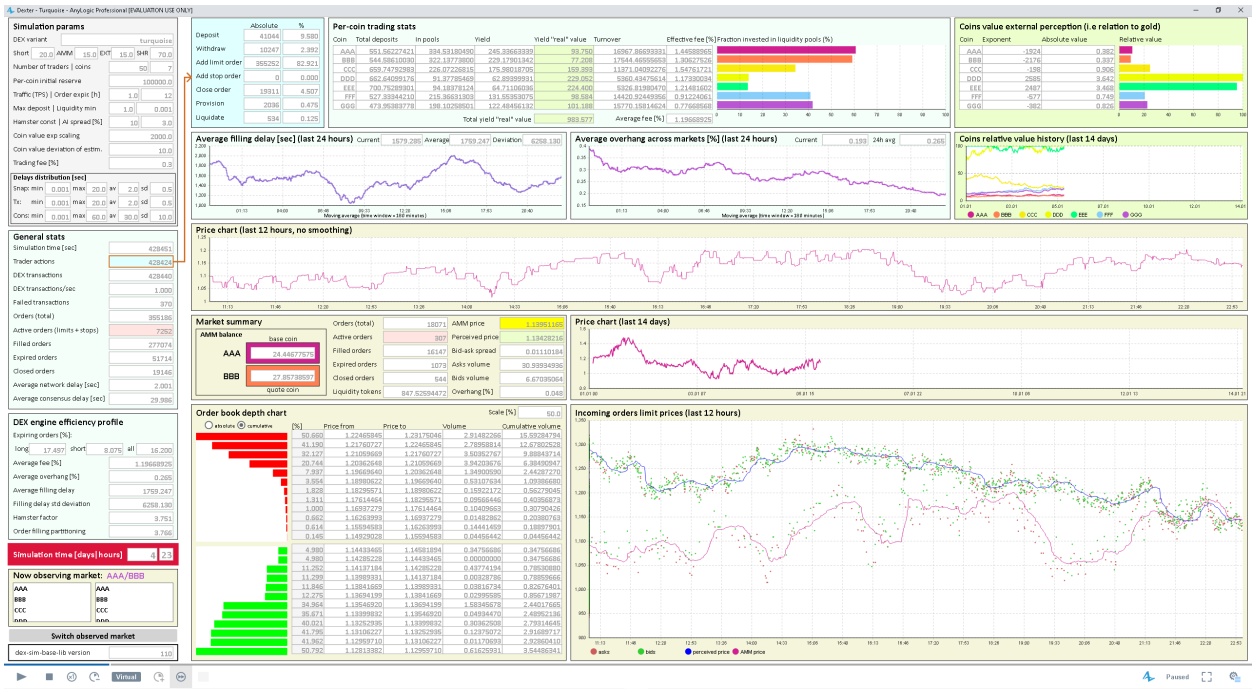
Snapshot request - network delay (min): 0.001 seconds
Snapshot request - network delay (max): 20 seconds
Snapshot request - network delay (mean): 2 seconds

1.3.7 Step 6: Run a simulation experiment

Initially, there are 4 simulation experiments defined: Sandbox, Teal, Turquoise, Uniswap. Right-clicking on an experiment brings a context menu with **Run** option (among others).

Starting an experiment will bring up a new window. This window is designed to be run full-screen (1440p or higher).

This is how a running simulation looks like:


Dexter - Turquoise - AnyLogic Professional [EVALUATION USE ONLY]
Average filling delay [sec] (last 24 hours)
Moving average (time window = 180 minutes)
Average overhang across markets [%] (last 24 hours)
Coins relative value history (last 14 days)
Price chart (last 12 hours, no smoothing)
Price chart (last 14 days)
Incoming orders limit prices (last 12 hours)
asks
bids
perceived price
AMM price
Switch observed market
Virtual
Paused

Simulation params

DEX variant	turquoise						
Short	20.0	AMM	15.0	EXT	15.0	SHR	70.0
Number of traders | coins	50	7					
Per-coin initial reserve	100000.0						
Traffic (TPS) | Order expir. [h]	1.0	12					
Max deposit | Liquidity min	1.0	0.001					
Hamster const | AI spread [%]	10	3.0					
Coin value exp scaling	2000.0						
Coin value deviation of estim.	10.0						
Trading fee [%]	0.3						

Delays distribution [sec]

Snap: min	0.001	max	20.0	av	2.0	sd	0.5
Tx: min	0.001	max	20.0	av	2.0	sd	0.5
Cons: min	0.001	max	60.0	av	30.0	sd	10.0

General stats

Simulation time [sec]	428451
Trader actions	428424
DEX transactions	428440
DEX transactions/sec	1.000
Failed transactions	370
Orders (total)	355186
Active orders (limits + stops)	7252
Filled orders	277074
Expired orders	51714
Closed orders	19146
Average network delay [sec]	2.001
Average consensus delay [sec]	29.986

DEX engine efficiency profile

Expiring orders [%]:					
long	17.497	short	8.075	all	16.200
Average fee [%]	1.19668925				
Average overhang [%]	0.265				
Average filling delay	1759.247				
Filling delay std deviation	6258.130				
Hamster factor	3.751				
Order filling partitioning	3.766				

Simulation time [days|hours] 4 23

Now observing market: AAA/BBB

AAA BBB CCC DDD | AAA BBB CCC DDD

dex-sim-base-lib version 110

	Absolute	%
Deposit	41044	9.580
Withdraw	10247	2.392
Add limit order	355252	82.921
Add stop order	0	0.000
Close order	19311	4.507
Provision	2036	0.475
Liquidate	534	0.125

Per-coin trading stats

Coin	Total deposits	In pools	Yield	Yield "real" value	Turnover	Effective fee [%]
AAA	551.56227421	334.53180490	245.33663339	93.750	16967.86693331	1.44588965
BBB	544.58610030	322.13773800	229.17901342	77.208	17544.46555653	1.30627526
CCC	659.74792983	226.07226815	175.98018705	159.393	11371.04092276	1.54761721
DDD	662.64099176	91.37785469	62.89399931	229.052	5360.43475614	1.17330034
EEE	700.75289301	94.18378124	64.71106036	224.400	5326.81980470	1.21481602
FFF	527.33344210	215.36631303	131.55353075	98.584	14420.92449356	0.91224061
GGG	473.95383778	198.10258501	122.48456132	101.188	15770.15814624	0.77668568

Fraction invested in liquidity pools (%)

Total yield "real" value 983.577 Average fee [%] 1.19668925

Coins value external perception (i.e relation to gold)

Coin	Exponent	Absolute value
AAA	-1924	0.382
BBB	-2176	0.337
CCC	-198	0.906
DDD	2585	3.642
EEE	2487	3.468
FFF	-577	0.749
GGG	-382	0.826

Relative value

● AAA ● BBB ● CCC ● DDD ● EEE ● FFF ● GGG

Average filling delay [sec] (last 24 hours): Current 1579.285 Average 1759.247 Deviation 6258.130

Average overhang across markets [%] (last 24 hours): Current 0.193 24h avg 0.265

Market summary

AMM balance

	base coin
AAA	24.44677575
BBB	27.85738597
	quote coin

Orders (total)	18071	AMM price	1.13951165
Active orders	307	Perceived price	1.13428216
Filled orders	16147	Bid-ask spread	0.01110184
Expired orders	1073	Asks volume	30.93934936
Closed orders	544	Bids volume	6.67035064
Liquidity tokens	847.52594472	Overhang [%]	0.048

Order book depth chart

absolute [✓ cumulative] Scale [%] 50.0

[%]	Price from	Price to	Volume	Cumulative volume
50.660	1.22465845	1.23175046	2.91482266	15.59284794
41.190	1.21760727	1.22465845	2.78958814	12.67802528
32.127	1.21059669	1.21760727	3.50352767	9.88843714
20.744	1.20362648	1.21059669	3.94203676	6.38490947
7.937	1.19669640	1.20362648	1.34900590	2.44287270
3.554	1.18980622	1.19669640	0.53107634	1.09386680
1.828	1.18295571	1.18980622	0.15922172	0.56279045
1.311	1.17614464	1.18295571	0.09566446	0.40356873
1.000	1.16937279	1.17614464	0.10409663	0.30790426
0.662	1.16263993	1.16937279	0.01482862	0.20380763
0.614	1.15594583	1.16263993	0.14441459	0.18897901
0.145	1.14929028	1.15594583	0.04456442	0.04456442
4.980	1.14433465	1.14581894	0.34756686	0.34756686
4.980	1.14285228	1.14433465	0.00000000	0.34756686
11.252	1.14137184	1.14285228	0.43774194	0.78530880
11.299	1.13989331	1.14137184	0.00328786	0.78859666
11.846	1.13841669	1.13989331	0.03816734	0.82676401
12.275	1.13694199	1.13841669	0.02995585	0.85671987
34.964	1.13546920	1.13694199	1.58345678	2.44017665
35.671	1.13399832	1.13546920	0.04934470	2.48952136
40.021	1.13252935	1.13399832	0.30362508	2.79314645
41.795	1.13106227	1.13252935	0.12375072	2.91689717
41.962	1.12959710	1.13106227	0.01170693	2.92860410
50.792	1.12813382	1.12959710	0.61625931	3.54486341

1.4 04. Model data types

In this chapter we do an overview of data types used in the simulator. This is pretty technical but needed to properly explain the data model underpinning Dexter. Skip this chapter (and following two chapters) if you are not interested in that many technical details.

1.4.1 Basic java types

As Anylogic is based on Java, there are several data types we take as granted and we use them throughout this documentation:

- Int: the type of 32-bit signed integers
- Long: the type of 64-bit signed integers
- Boolean: the type of boolean values (true/false)
- Double: the type of floating-point values

1.4.2 Amounts and prices

We follow the TLA+ spec in making arithmetic representation decisions:

- fixed-point numbers are used for representing amounts of tokens (see `FPNumber` class)
- fractions are used for representing prices (see `Fraction` class)

Fixed-point arithmetic

`FPNumber` implements fixed-point arithmetic. The decimal precision (i.e. number of places after decimal point) is statically configured via constant `io.onomy.commons.FIXED_POINT_ARITHMETIC_PRECISION: Int`. This class is tailored to represent "money" values, i.e. it works mostly like integers. There are some cases when rounding occurs (these places are marked below).

Internally, `FPNumber` instance is just represented as BigInt value `pips`.

The signature of `FPNumber` class is as follows:

```
case class FPNumber(pips: BigInt) extends Ordered[FPNumber] {

  /** Sum of two numbers.*/
  def +(amount: FPNumber): FPNumber

  /** Difference of two numbers.*/
  def -(amount: FPNumber): FPNumber

  /** x ---> -x */
  def negated: FPNumber

  /** Returns -1 for negative values, 1 for positive values, 0 for zero.*/
  def signum: Int

  /**
   * Shifts the decimal point given number of places (left or right).
   * @param p number of decimal places (positive = shift right, negative = shift left)
   */
  def decimalShift(p: Int): FPNumber

  /**
   * Fixed-point multiplication.
   * Caution: rounding can occur here.
   */
  def *(other: FPNumber): FPNumber

  /**
   * Fixed-point division.
   * Caution: rounding can occur here.
   */
  def /(other: FPNumber): FPNumber

  /** x ---> 1/x */
  def reciprocal: FPNumber

  /**
```

(continues on next page)

(continued from previous page)

```
  * Multiplication by a fraction.
  * Caution: rounding can occur here.
  */
  def **(fraction: Fraction): FPNumber

  /**
   * Conversion to floating-point arithmetic.
   * Caution: this is precision-losing operation.
   */
  def toDouble: Double

  /**
   * Conversion to Fraction.
   * This is a precise conversion, as Fraction has unlimited precision.
   */
  def toFraction: Fraction

  /** Rounding towards positive infinity.*/
  def ceiling: BigInt

  /** Rounding towards negative infinity. */
  def floor: BigInt

  /** Returns fractional part of the value, ignoring sign.*/
  def unsignedFractionalPart: FPNumber

  /**
   * Rounding towards zero.
   * This is just cutting away the fractional part of this value.
   */
  def roundTowardsZero: BigInt

  /** Rounding away from zero.*/
  def roundAwayFromZero: BigInt

  /** Mathematical ordering.*/
  override def compare(that: FPNumber): Int

  /** Standard distance between numbers.*/
  def distanceTo(that: FPNumber): FPNumber
}
```

Additionally, there are some class-level functions associated with `FPNumber`:

```
object FPNumber {

  /** n to the power of k */
  private def power(n: Long, k: Int): BigInt

  /** Shortcut for FPNumber.fromLong(0).*/
  val zero: FPNumber

```

(continues on next page)

(continued from previous page)

```
  /** Shortcut for FPNumber.fromFraction(Fraction(1,2)).*/
  val half: FPNumber

  /** Shortcut for FPNumber.fromLong(1).*/
  val one: FPNumber

  /** Conversion String ---> FPNumber.*/
  def parse(s: String): FPNumber

  /**
   * Conversion Long ---> FPNumber.
   * For example, when precision is set to 5:
   * fromLong(123).toString == "123.00000"
   */
  def fromLong(n: Long): FPNumber

  /** Conversion BigDecimal ---> FPNumber.*/
  def fromBigDecimal(x: BigDecimal): FPNumber

  /**
   * Conversion Double ---> FPNumber (with mathematical rounding).
   */
  def fromDouble(a: Double): FPNumber

  /** Conversion Double ---> FPNumber (with rounding towards negative infinity).*/
  def fromDoubleRoundingDown(a: Double): FPNumber

  /** Conversion Double ---> FPNumber (with rounding towards positive infinity).*/
  def fromDoubleRoundingUp(a: Double): FPNumber

  /** Conversion Fraction ---> FPNumber (with mathematical rounding).*/
  def fromFraction(f: Fraction): FPNumber

  /** Returns smaller number from given two.*/
  def min(a: FPNumber, b: FPNumber): FPNumber

  /** Returns bigger number from given two.*/
  def max(a: FPNumber, b: FPNumber): FPNumber

  /** Cancels "minus" sign. */
  def abs(a: FPNumber): FPNumber

  /**
   * Smallest value that could be represented with FPNumber, given the configured
   * arithmetic precision.
   */
  val quantum: FPNumber = FPNumber.one.decimalShift(- FIXED_POINT_ARITHMETIC_PRECISION)

}
```

Fractions

`Fraction` implements arbitrary-precision mathematical quotient numbers. Internal representation is based on a pair of BigInteger values. We use it mostly for representing prices.

The signature of class `Fraction` is as follows:

```
class Fraction(x: BigInt, y: BigInt) extends Comparable[Fraction] {

  def numerator: BigInt

  def denominator: BigInt

  /** Conversion Fraction ---> Double (rounding can occur) */
  def toDouble: Double

  /** Conversion BigInt ---> Fraction */
  def this(x: BigInt): Fraction

  /** Mathematical comparison of fractions, coherent with 'Comparable' interface */
  override def compareTo(other: Fraction): Int

  /** Adding of fractions. */
  def +(that: Fraction): Fraction

  /** Adding of a fraction and an integer value */
  def +(that: BigInt): Fraction

  /** Subtracting of fractions. */
  def -(that: Fraction): Fraction

  /** Multiplication of fractions. */
  def *(that: Fraction): Fraction

  /** Multiplication of a fraction by a BigInt value. */
  def *(that: BigInt): Fraction

  /** Division of fractions */
  def /(that: Fraction): Fraction

  /** Division of fractions. */
  def /(that: BigInt): Fraction

  /** x ---> -x */
  def negated : Fraction

  /** Mathematical 'signum' function over fractions. */
  def sgn: Int

  /** Conversion Fraction ---> BigInt. */
  def toBigInt: BigInt

  /** Converts fraction a/b to fraction b/a. */
  def reciprocal: Fraction
```

(continues on next page)

(continued from previous page)

```
  /** Arithmetic comparison */
  def >(that: Fraction): Boolean

  /** Arithmetic comparison */
  def >=(that: Fraction): Boolean

  /** Arithmetic comparison */
  def >(that: BigInt): Boolean

  /** Arithmetic comparison */
  def >=(that: BigInt): Boolean

  /** Arithmetic comparison */
  def <(that: Fraction): Boolean

  /** Arithmetic comparison */
  def <=(that: Fraction): Boolean

  /** Arithmetic comparison */
  def <(that: BigInt): Boolean

  /** Arithmetic comparison */
  def <=(that: BigInt): Boolean

  /** Raises `this` to the power of n */
  def ^(n: Int): Fraction

}
```

1.4.3 Time

There are two notions of time in use:

- **simulation time**: this is the time simulated by Anylogic engine, following the DES model of events queue; timepoints are represented as Double values and are interpreted as seconds
- **blockchain time**: this the blockchain-implementation-specific "internal" time of a blockchain, represented as Long value

Blockchains in general do not have the idea of "real" time - this is due to the very nature of what a blockchain is. However every blockchain has some notion of "internal" time-like concept, which corresponds to the chronology of transactions execution, namely the following invariant holds:

if transaction t_1 can see transaction t_2 in its past, then $bTime(t_1) > bTime(t_2)$

Caution: when running Dexter in command-line mode (see chapter 15), there is no proper simulation of time in place, hence the simulation clock is mocked. Therefore time-related statistics are meaningless in command-line.

1.4.4 Hash

Hashes show up naturally as identifiers of transactions, coins and accounts. This is typically how identifiers of various thing appear on a blockchain.

Internally, hash is just a binary array. We use `Hash` type to represent it. We frequently use the fact that hashes have natural ordering (by lexicographic comparison).

1.4.5 Battery of counters

This is a collection of FPNumber values indexed by some index type. In other words, `BatteryOfCounters[T]` is equivalent to `Map[T,FPNumber]`, where `T` is the type of indexes.

1.5 05. Blockchain model

A DEX - as being deployed on a blockchain - has 4 conceptual "perspectives":

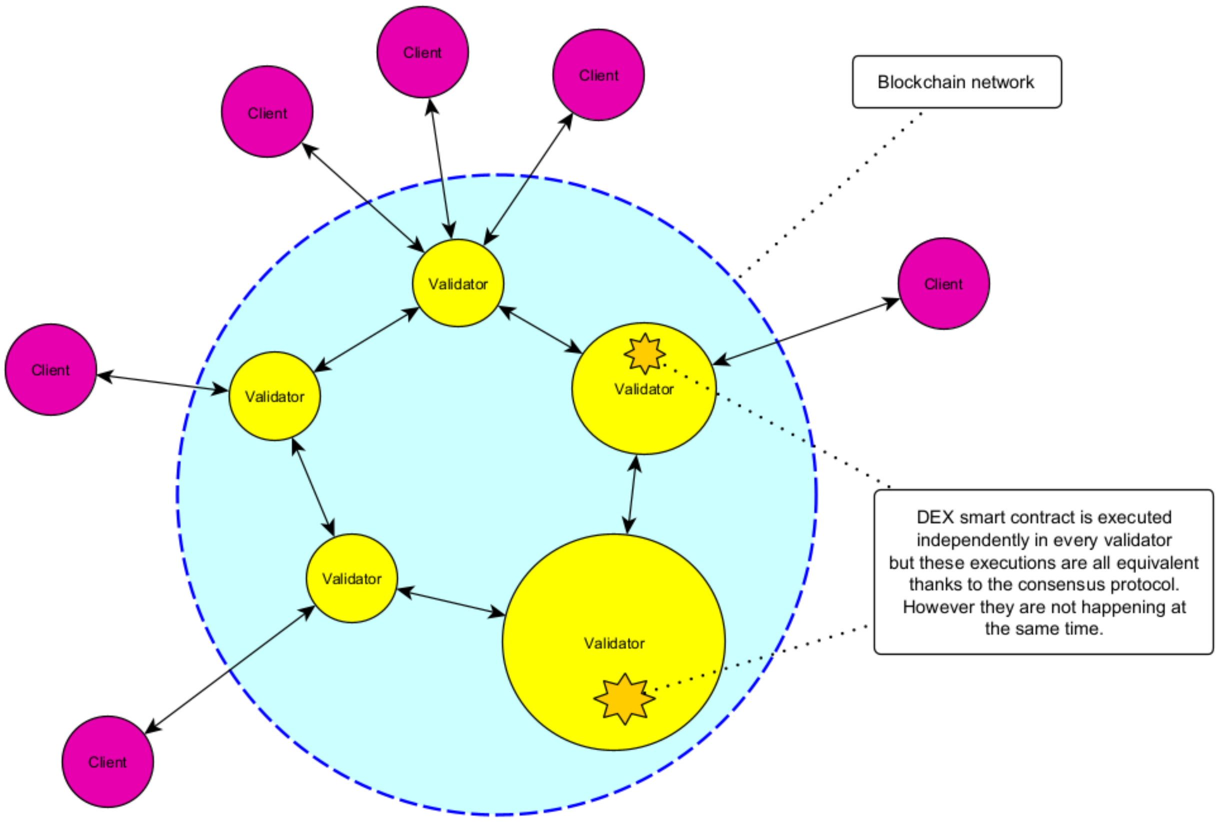
1. End-user perspective: this covers the view of an end-user, so how the usage of the DEX looks like in practice; this would normally coincide with the GUI of the DEX client.
2. Client-api perspective: this covers the view of a client-developer; is is composed of APIs available to the DEX client software; this APIs may communicate to on-chain and off-chain components
3. Blockchain perspective: this covers the view of a DEX developer and includes all the blockchain-specific solutions. used to implement the DEX.
4. Internal DEX model perspective: this covers the internal working of a DEX, abstracting away from design choices specific to a given blockchain where the DEX will be deployed; in the implementation phase this model would typically be mapped to a smart-contract.

Here we focus on the the way we mock blockchain architecture in the simulator. This covers perspectives (2) and (3).

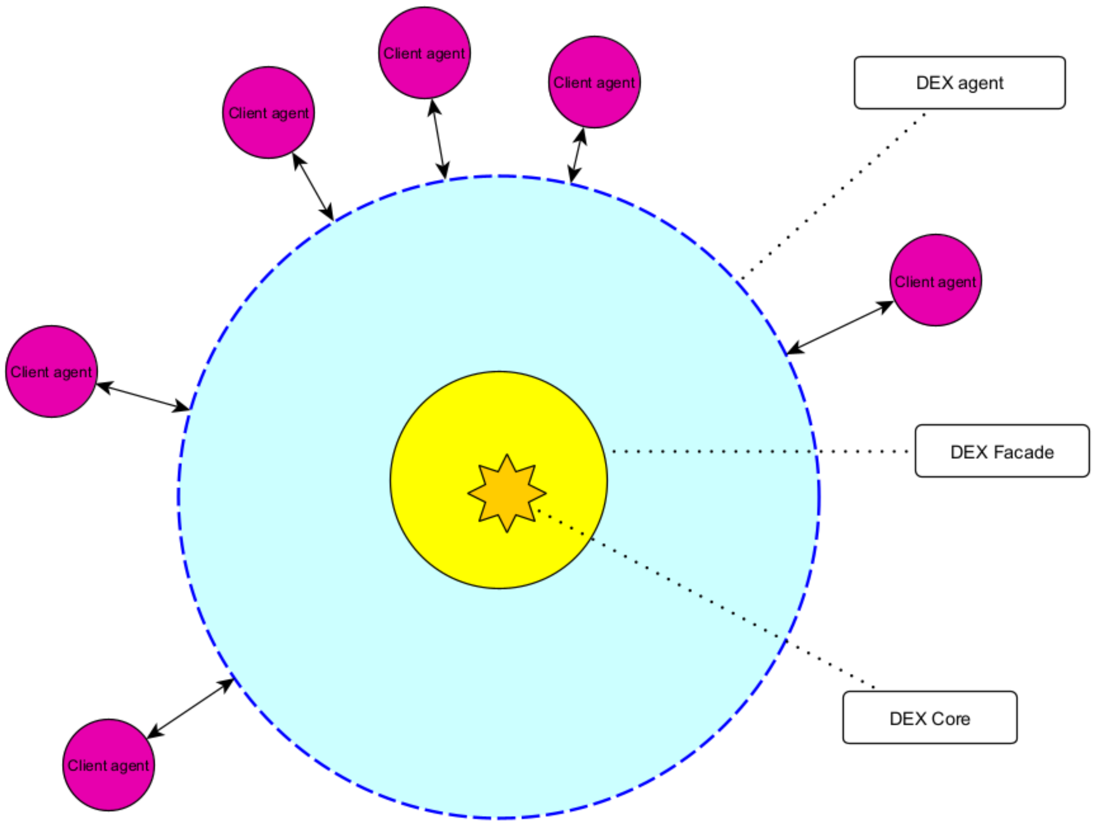
1.5.1 Real blockchain vs blockchain mock

A real blockchain is a P2P network of nodes (validators) building blocks and collectively running a consensus protocol to come up with ever-growing sequence of blocks. The goal of underlying consensus protocol is to ensure that the sequence of blocks - once established up to position N - is visible to every node as such (i.e. is "finalized") and the sequence can never be changed.

Surrounding the network of validators is the (usually much larger) network of blockchain clients. Every blockchain client picks (arbitrarily) some validator and uses it as a gateway to the blockchain. In theory this selection should not influence the operation of a client, because all validators offer semantically equivalent API for clients.


Client
Client
Client
Client
Client
Client
Client
Blockchain network
Validator
Validator
Validator
Validator
Validator
DEX smart contract is executed independently in every validator but these executions are all equivalent thanks to the consensus protocol. However they are not happening at the same time.

For the purpose of DEX simulation we do not simulate the actual network of validators. Instead we just have a single agent representing the blockchain, while clients are also represented as agents, and the communication between agents and the blockchain is materialized as agent-to-agent message passing.



Caution: there are some additional agents involved in the design so to accommodate the simulation of network communication between clients and the blockchain. Go to chapter 10 for more details on this.

1.5.2 Blockchain accounts

A blockchain account is normally identified by a cryptographic public key (of public-private key pair). For the purpose of the simulation we stripped cryptography and the account id is represented as a (randomly generated) hash value.

1.5.3 Client-Blockchain communication protocol

There are two types of messages that a client can send to the blockchain:

- transactions
- queries

There is no response for a transaction.

On the other hand, queries follow request-response pattern.

Caution: The meaning of fields in the data structures enumerated below may be not obvious from the context. In case of doubts, please refer to chapter 6, where the DEX model is explained.

1.5.4 Transactions

A transaction has the following structure:

```
creator: AccountAddress
clientTime: SimulationTime
ttl: BlockchainTime
body: DexTransactionBody
hash: Hash
```

A transaction is in fact an operation to be executed on the blockchain virtual computer, i.e this execution changes the state of the blockchain computer.

Fields explained:

creator account address; the transaction will be executed on behalf of this account; in other words this is the trader's blockchain identity

clientTime real time of the client at the moment of sending this transaction

ttl latest blockchain time for this transaction; this transaction should not get executed at later blockchain time

body contains the "business-level" information, specific to transaction type (see below)

hash transaction id; in real life it would be the real hash of transaction binary representation, but we just mock this with randomly-generates hashes

Execution of a transaction may fail. Such failure is signaled in the diagnostic log.

1.5.5 Transaction body

Transaction body can have one of the following structures:

Deposit

Transfers tokens from the reserve to trader's account. At the business level it corresponds to minting tokens.

```
Deposit(
  accountName: String,
  amount: FPNumber,
  coin: Coin
)
```

Withdraw

Transfers tokens from trader's account back to reserve. At the business level it corresponds to burning tokens.

```
Withdraw(
  amount: FPNumber,
  coin: Coin
)
```

Init AMM

Initializes a market. This will succeed only for an uninitialized market. Specified amounts of both coins will be transferred from trader's account.

```
InitAMM(
  aCoin: Coin,
  bCoin: Coin,
  aCoinAmount: FPNumber,
  bCoinAmount: FPNumber
)
```

Add liquidity

Adds liquidity to the AMM at the selected market. This operation creates new liquidity coins and transfers them to the trader's account. This is how a trader becomes a liquidity provider.

```
AddLiquidity(
  marketId: CoinPair, //must be a normalized pair
  amountCoin: Coin, //points to one of coins in the market it
  amount: FPNumber //amount of selected coin (the other amount will be automatically␣
↪calculated)
)
```

Withdraw liquidity

Burns liquidity coins and decreases balances of the AMM at the selected market. This is how a liquidity provider consumes his profits.

```
WithdrawLiquidity(
  marketId: CoinPair,
  amountOfLiquidityCoinsToBurn: FPNumber
)
```

Add order

Registers new order on the exchange (i.e adds it to the relevant order book). Upon registering the new order becomes ready for execution.

```
AddOrder(
  orderDirectionFrom: Coin, //coin to be sold
  orderDirectionTo: Coin, //coin to be bought
  orderType: OrderType, //LIMIT or STOP
  price: Fraction,
  amount: FPNumber, amount of sell coin
  expirationTimepoint: SimulationTime,
  isShort: Boolean
)
```

Close order

Terminates an order. The order will be removed from order book.

```
CloseOrder(
  askCoin: Coin,
  bidCoin: Coin,
  positionId: Hash
)
```

1.5.6 Queries

Every query has separate request and response structures.

Currently only one type of query is utilized in the simulator design - **GetAccountSnapshot**. With this query we simulate a typical activity of a trader, namely - checking current state of his account. A trader makes his trading decision with a fresh account snapshot at hand.

Request

```
GetAccountSnapshotRequest(
  creator: AccountAddress,
  clientTime: SimulationTime,
  requestId: Long
)
```

Response

```
GetAccountSnapshotResponse(
  requestId: Long,
  dexTimepoint: SimulationTime,
  accountSnapshot: AccountSnapshot
)
```

AccountSnapshot is the following structure:

```
 AccountSnapshot(
   blockchainTime: BlockchainTime,
   coinsWithNonZeroBalance: Array[Coin],
   coin2FreeBalance: Map[Coin, FPNumber],
   coin2LockedBalance: Map[Coin, FPNumber],
   activePositions: Array[Position],
   liquidityParticipation: Map[CoinPair, FPNumber],
   initializedMarkets: Map[CoinPair, (FPNumber, FPNumber)]
)
```

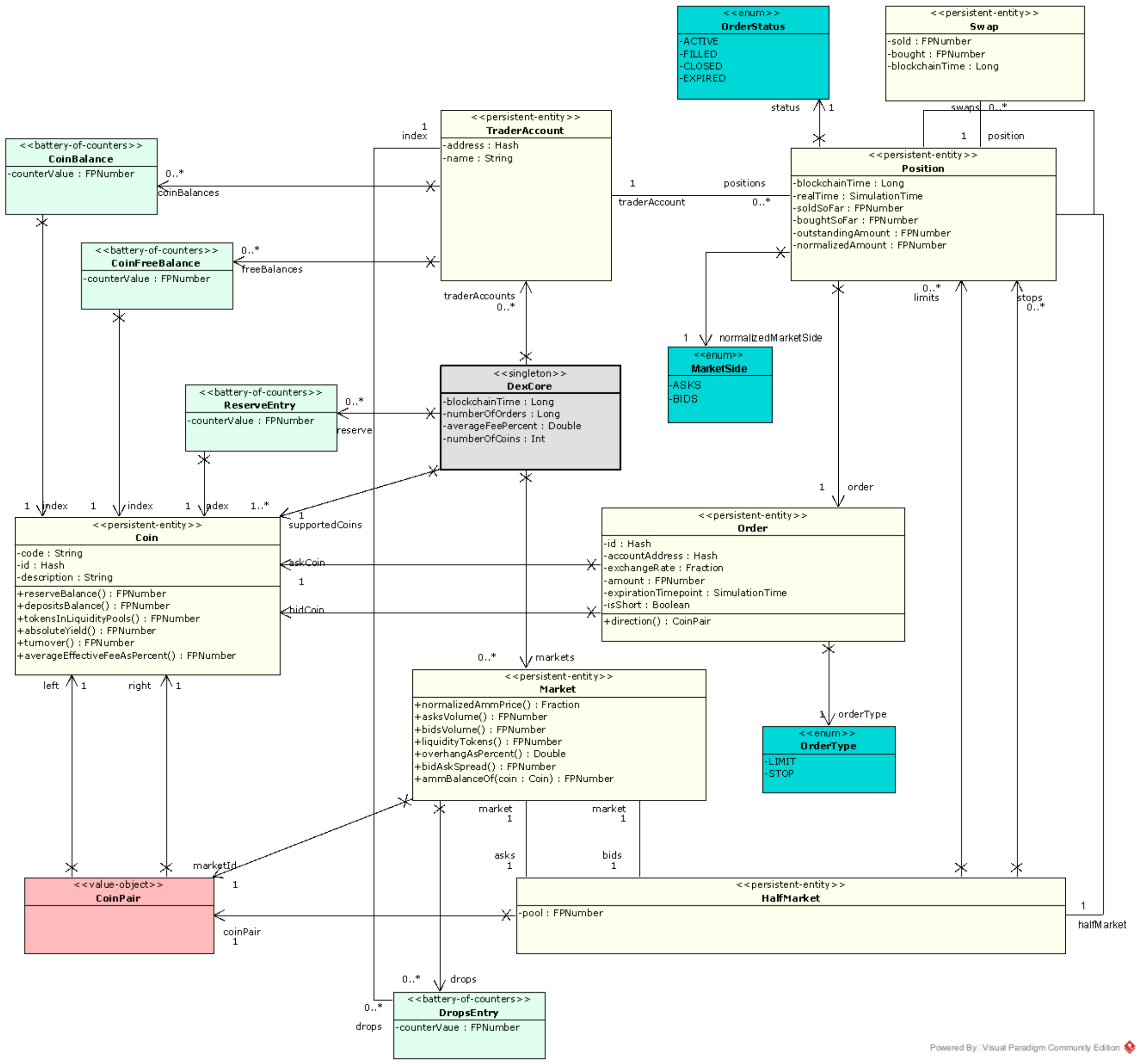
1.6 06. DEX model

There is a general model of a DEX (derived from the TLA+ spec we mentioned before) which sits at the conceptual center of Dexter design. Only one part of this model - the executor - is pluggable, so that comparative simulation of various executors ca nbe achieved.

In this chapter we describe this common base model, while the next chapter is devoted to the various executors pre-installed in the current version of Dexter.

This chapter covers perspective (4) according to the list of perspectives explained in chapter 5.

This UML diagram covers the whole model:



To illustrate the behaviour of DEX we use several examples written in Dexter scripting language. See chapter 15 for the details on the syntax, semantics and how to run these scripts in Dexter command-line mode.

1.6.1 Coins and tokens

We use the term **coin** meaning "type of cryptocurrency". For example, in our lingo, BTC and ETH are coins. On the other hand, we use the term **token** when we talk about amounts of coins. In practice: when I sell 1.305 bitcoins, we say that the the coin of that transaction is "bitcoin" and the number of tokens transferred is 1.305.

Coins are represented with `Coin` type, while token amounts are represented with `FPNumber` type.

We frequently need to talk about pairs of coins. When `AAA` and `BBB` are some coins, we want to be able to form the pairs `(AAA,BBB)` and `(BBB,AAA)`. This concept is represented with `CoinPair` type.

We also sometimes need 2-element coin sets. This is different than a coin pair, because a pair is ordered, while a set is not. However, to keep things simpler, we represent a 2-element coin set as a **normalized coin pair**. This normalization works as follows: because every coin has an id (hash), we consider a CoinPair to be **normalized** if coins in this pair are ordered along their hashes.

Coins collection is automatically generated on simulation start.

Example

With number of coins configured to 7, at the beginning of the simulation the following information will show up in the console:

```
coins in use:
  code=AAA id=ee93-992a-dbdd-6168 description=Sample coin AAA
  code=BBB id=9853-0e3b-6c5e-fd60 description=Sample coin BBB
  code=CCC id=aea4-0d22-8e88-7e5b description=Sample coin CCC
  code=DDD id=ba32-7373-3682-7484 description=Sample coin DDD
  code=EEE id=e999-5a44-0ea6-4d46 description=Sample coin EEE
  code=FFF id=1531-f159-e87b-6776 description=Sample coin FFF
  code=GGG id=a8f4-8174-5e0a-3c25 description=Sample coin GGG
```

These are automatically generated coins. For coins AAA and BBB, two coin pairs are possible: `CoinPair(AAA,BBB)` and `CoinPair(BBB,AAA)`. Now let us look at the hashes. A hash is printed using hex encoding of corresponding byte array and the comparison of hashes is lexicographic-per-byte. First byte of coin AAA identifier is `ee` in hex, which is number 238. First byte of coin BBB identifier is `98` in hex, which is number 152. Hence, BBB has smaller hash than AAA so we can conclude that `CoinPair(BBB,AAA)` is normalized and `CoinPair(AAA, BBB)` is not normalized.

1.6.2 DEX core

`DexCore` keeps all the information needed to process transactions generated by traders. It stands as a facade for decentralized exchange operations (see *Execution of orders*). On top of this, `DexCore` calculates exchange statistics (see chapter 12).

We also have `DexFacade`, which plays the role of a more "high level" API to the DEX, while `DexCore` offers API structure closer to the TLA+ spec.

1.6.3 Trader account

Trader account is just the same as blockchain account. DEX becomes aware of a trader account while executing the first **deposit** operation for this account.

An account stores the following information:

- current balance of tokens (per each coin)
- current free balance of tokens (per each coin)
- current balance of liquidity tokens (per market)
- open positions

When a trader adds an order, the sell-coin amount gets locked so that it cannot be re-used in another order.

1.6.4 Reserve

Reserve is the way we represent total tokens supply (per coin):

- transferring tokens from reserve to trader account corresponds to "minting" tokens
- transferring tokens from trader account to reserve corresponds to "burning" tokens

Reserve is represented as a battery-of-counters, indexed by coin.

The following DexFacade operations deal with reserve:

```
class DexFacade {

  //Take a number of tokens (of specific coin) from the reserve and place it into an␣
↪exchange account.
  def deposit(accountAddress: AccountAddress, accountName: String, amount: FPNumber,␣
↪coin: Coin): Unit

  //Take a number of tokens (of specific coin) from specified trader account and return␣
↪it to the reserve.
  def withdraw(accountAddress: AccountAddress, amount: FPNumber, coin: Coin): Unit

}.
```

Example

The following scripts initializes trader accounts for 3 traders.

```
trader 00: deposit  11.234 AAA
trader 01: deposit  5.01 AAA
trader 01: deposit  1.203 BBB
trader 02: deposit  0.099 CCC
trader 00: withdraw 0.1 AAA
trader 02: deposit  0.099 CCC
```

Caution: In this example (and all subsequent examples) we use per-coin initial reserve set to 1000.

After executing this script, the final state of the DEX will be:

```
-------------------------------- final state dump --------------------------------------
↪--------
per-coin stats
  AAA: reserve=983.8560000000000000 deposits=16.1440000000000000 in-pools=0.
↪0000000000000000 yield=0.0000000000000000 turnover=0.0000000000000000 fee=0.
↪0000000000000000
  BBB: reserve=998.7970000000000000 deposits=1.2030000000000000 in-pools=0.
↪0000000000000000 yield=0.0000000000000000 turnover=0.0000000000000000 fee=0.
↪0000000000000000
  CCC: reserve=999.8020000000000000 deposits=0.1980000000000000 in-pools=0.
↪0000000000000000 yield=0.0000000000000000 turnover=0.0000000000000000 fee=0.
↪0000000000000000
accounts
  trader-0 (f03e-152a-9bcc-84a5)
    AAA: 11.1340000000000000 (free=11.1340000000000000 locked=0.0000000000000000)
  trader-1 (964c-867b-98d2-f9a5)
    BBB: 1.2030000000000000 (free=1.2030000000000000 locked=0.0000000000000000)
    AAA: 5.0100000000000000 (free=5.0100000000000000 locked=0.0000000000000000)
  trader-2 (7daa-7a83-0bc9-b98c)
    CCC: 0.1980000000000000 (free=0.1980000000000000 locked=0.0000000000000000)
markets
-------------------------------------- end ---------------------------------------------
↪--------
```

1.6.5 Markets

DEX contains the collection of markets. For every normalized pair of coins we generate one market. This follows the tradition of Forex: a market AAA/BBB is where coin AAA can be traded for BBB and coin BBB can be traded for AAA.

Hence, if we configure Dexter to simulate n coins, $\frac{n*(n-1)}{2}$ markets will be generated.

We also borrow from Forex another naming convention: base/quote. Base is the coin we think of as being the "asset" and quote is the coin we think of as being the "money". The selection of base is arbitrary - we just again use the normalization of coin pairs here and in a normalized coin pair `CoinPair(AAA,BBB)`, left-side coin (AAA in this example) is base, while right-side coin (BBB in this example) is quote.

Base-quote convention is basically a way to pick some orientation of the market (which is otherwise fully symmetric), hence talking about buyers, sellers and prices becomes unambiguous without further explanation of which coin we are selling. With base/quote setup in place, whoever is selling the base coin is the seller, and whoever is buying the base coin is the buyer. The price is then the amount of quote tokens to be paid for 1 base token.

A market maintains 4 ordered collections of orders:

- limit orders (buyers collection)
- limit orders (sellers collection)
- stop orders (buyers collection)
- stop orders (sellers collection)

Caution: stop orders are considered "experimental" feature and are not fully covered by this version of Dexter manual.

A market also contains a liquidity pool (also called "automated market maker" or AMM), which is a "locked" amount of base tokens and quote tokens.

1.6.6 Liquidity pools

DEX-es we investigate are all based on so called “liquidity pools”. A liquidity pool is just certain amount of tokens kept as a “operational buffer” by the exchange. Thanks to the existence of such buffer, orders execution may be achieved as trader-vs-pool transfers. This is in contrary to traditional exchanges (like Forex), where execution of orders is achieved via matching sell-vs-buy orders, which leads to trader-vs-trader transfers.

A separate liquidity pool is attached to every market and technically has a form of two variables tracing the pool balance - one variable for each coin on the market. See the variable `HalfMarket.pool` on DEX model above.

The accumulation of tokens in the liquidity pool is based on active participation of investors. Any trader can become an investor in any liquidity pool by issuing “addLiquidity” transaction. The effect of this transaction will be a donation of proportional amounts of both coins to the pool.

`Market.drops` collection is the way DEX tracks participation of traders in given liquidity pool. Participation tracking is based on a fictional “liquidity” coin (separate for every market), and drops is a collection keeping the balance of liquidity coin per trader.

The following DexFacade operations deal with liquidity management:

```
class DexFacade {

  //Initialize liquidity pool.
  def initAMM(account: AccountAddress, aCoin: Coin, bCoin: Coin, aCoinAmount: FPNumber,␣
↪bCoinAmount: FPNumber): Boolean

  //Add liquidity to an already initialized pool.
  //Only one coin and its amount is provided as argument, the other side is␣
↪automatically calculated
  def addLiquidity(accountAddress: AccountAddress, marketId: CoinPair, amountCoin: Coin,␣
↪amount: FPNumber): Boolean

  //Burns specified amount of liquidity coins owned by specified trader account.
  //The trader will get proportional share of both coins of the liquidity pool.
  def withdrawLiquidity(account: AccountAddress, marketId: CoinPair,␣
↪amountOfLiquidityCoinsToBurn: FPNumber): Boolean
}
```

`InitAMM` just allocates fixed amount of liquidity coins (100.0) as drops entry for the issuing trader.

`AddLiquidity` increases investment of issuing trader into AMM of selected market. The algorithm here ensures that the amounts of base and quote coin transferred from trader account will not change the current price. If and $ammQuote$ are the initial balances of the AMM, and the trader adds liquidity by transferring x tokens of base coin and y tokens of quote coin, the DEX ensures that the following condition holds:

$$\frac{ammBase}{ammQuote} = \frac{ammBase + x}{ammQuote + y}$$

Let:

- td denote the total amount of liquidity tokens minted for the market under consideration
- d denote the amount of liquidity tokens DEX will mint in effect of `AddLiquidity` and add to the total balance of drops for the issuing trader account

`AddLiquidity` mints liquidity tokens so that the following equation is satisfied:

$$\frac{x}{ammBase} = \frac{y}{ammQuote} = \frac{d}{td}$$

Caution: because of integer rounding, this equation usually cannot be satisfied exactly. DEX attempts to adhere to this equation as much as the fixed-point arithmetic allows to.

Liquidity pool is the way we establish the concept of "current price" on the market: it is just the value `ammQuote / ammBase`.

`WithdrawLiquidity` burns specified amount of liquidity tokens. Let $ammBase$ and $ammQuote$ denote current balance of the AMM for the relevant market. Let td be the total number of liquidity tokens for this market. Let d be the amount of liquidity tokens that a trader wants to burn. This will end up executing the following token transfers from AMM to trader's account:

- base coin: $\frac{d}{td} * ammBase$
- quote coin: $\frac{d}{td} * ammQuote$

Example

In this example there are 3 traders, but 2 of them become liquidity providers.

```
trader 00: deposit  11.234 AAA
trader 00: deposit  5.01 BBB
trader 01: deposit  5.01 AAA
trader 01: deposit  7.901 BBB
trader 02: deposit  0.099 CCC
trader 00: amm-init AAA=1.2 BBB=3.1
trader 01: +amm AAA/BBB AAA=0.23
trader 01: deposit  3.3 CCC
trader 01: amm-init BBB=2 CCC=1.9
```

After executing this script, the final state of the DEX will be:

```
-------------------------------- final state dump ---------------------------------------
↪--------
per-coin stats
  AAA: reserve=983.7560000000000000 deposits=16.2440000000000000 in-pools=1.
↪4300000000000000 yield=0.0000000000000000 turnover=0.0000000000000000 fee=0.
↪0000000000000000
  BBB: reserve=987.0890000000000000 deposits=12.9110000000000000 in-pools=5.
↪6941666666666666 yield=0.0000000000000000 turnover=0.0000000000000000 fee=0.
↪0000000000000000
  CCC: reserve=996.6010000000000000 deposits=3.3990000000000000 in-pools=1.
↪9000000000000000 yield=0.0000000000000000 turnover=0.0000000000000000 fee=0.
↪0000000000000000
accounts
  trader-0 (6520-118d-99d3-651f)
    BBB: 1.9100000000000000 (free=1.9100000000000000 locked=0.0000000000000000)
    AAA: 10.0340000000000000 (free=10.0340000000000000 locked=0.0000000000000000)
  trader-1 (dd9d-dd30-e1fb-ebf6)
    BBB: 5.3068333333333334 (free=5.3068333333333334 locked=0.0000000000000000)
    AAA: 4.7800000000000000 (free=4.7800000000000000 locked=0.0000000000000000)
    CCC: 1.4000000000000000 (free=1.4000000000000000 locked=0.0000000000000000)
  trader-2 (aa15-a7ce-b653-fbb1)
    CCC: 0.0990000000000000 (free=0.0990000000000000 locked=0.0000000000000000)
markets
  BBB/AAA amm-price: 0.3870967741935483 amm-balance: BBB=3.6941666666666666 AAA=1.
↪4300000000000000
```

(continues on next page)

(continued from previous page)

```
    stats
      active orders: 0
      asks volume (limit orders only): 0.0000000000000000
      bids volume (limit orders only): 0.0000000000000000
      liquidity tokens: 119.1666666666666666
      overhang [%]: 0.0
      bid-ask spread: 0.0000000000000000
    liquidity providers participation (aka 'drops')
      trader-1 = 19.1666666666666666
      trader-0 = 100.0000000000000000
    order book - asks
      limits
      stops
    order book - bids
      limits
      stops
  BBB/CCC amm-price: 0.9500000000000000 amm-balance: BBB=2.0000000000000000 CCC=1.
↪9000000000000000
    stats
      active orders: 0
      asks volume (limit orders only): 0.0000000000000000
      bids volume (limit orders only): 0.0000000000000000
      liquidity tokens: 100.0000000000000000
      overhang [%]: 0.0
      bid-ask spread: 0.0000000000000000
    liquidity providers participation (aka 'drops')
      trader-1 = 100.0000000000000000
    order book - asks
      limits
      stops
    order book - bids
      limits
      stops
-------------------------------------- end ----------------------------------------------
↪--------
```

1.6.7 Yield

By **yield** we mean the profit made by a liquidity provider - as seen at some specified state of DEX. We formalize yield as a value calculated separately for every <coin, market,trader> combination:

Let:

- A and B be fixed coins
- t be fixed trader account
- $prov$ be the sum of A tokens added to the liquidity pool on the market $< A, B >$ by trader t via **add-liquidity** operations (in the whole lifetime of DEX, up to the currently considered state)
- liq be the sum of A tokens withdrawn from the liquidity pool on the market $< A, B >$ by trader t via **remove-liquidity** operations (in the whole lifetime of DEX, up to the currently considered state)
- a and b be balances of respectively A and B on the market $< A, B >$

- d be the amount of liquidity tokens owned by trader t on the market $< A, B >$
- td be the total amount of liquidity tokens on the market $< A, B >$

If at this moment trader t decided to capitalize its share of the liquidity pool, it would be:

- this amount of coin A: $\frac{d}{td} * a$
- this amount of coin B: $\frac{d}{td} * b$

Let us focus on the overall coin A balance as seen by the trader t from the perspective of calculating the profit/loss:

- so far I invested $prov$ tokens into the pool
- so far I withdrew liq tokens from the pool
- as of now $\frac{d}{td} * a$ tokens in the pool belongs to me

We define **yield** (per trader-market-coin combination) as: $owned + withdrew - invested$, i.e:

$$yield_{tmc} = \frac{d}{td} * a + liq - prov$$

Yield can be summed over all markets, leading to the "yield per trader-coin combination", which has the form of a function:

$$yield_{tc} : T \times C \to \mathcal{FP}$$

... where T is the set of all trader accounts, C is the set of all coins and $\mathcal{FP}$ are fixed-point numbers.

Yield can be further summed over all traders, leading the "yield per coin" function:

$$yield_c : C \to \mathcal{FP}$$

This is exactly the yield listed in the **per-coin-stats** section (see the final state dump in examples).

Caution: please notice that yield can be negative.

1.6.8 Orders and positions

An order represents a willing to trade. An order is prepared by a DEX client and is represented as an immutable structure:

```
case class Order(
    id: Hash,
    orderType: OrderType,
    accountAddress: AccountAddress,
    askCoin: Coin,
    bidCoin: Coin,
    exchangeRate: Fraction,
    amount: FPNumber,
    expirationTimepoint: SimulationTime,
    isShort: Boolean
)
```

Fields explained:

id hash code of an order

orderType `OrderType.Limit` or `OrderType.Stop`

accountAddress trader id

askCoin coin which the trader wants to buy

bidCoin coin which the trader wants to sell

exchangeRate worst price the trader is going to accept for the execution of this order; if `x` tokens is sold and `y` tokens is bought, the price is calculated as `y/x` and the invariant is `exchangeRate >= y/x`

amount amount of sell bid coin that the trader wants to sell

expirationTimepoint timepoint when this order expires; we use real time here which is a quick-and-dirty hack, because there is no precise notion of real time on a blockchain

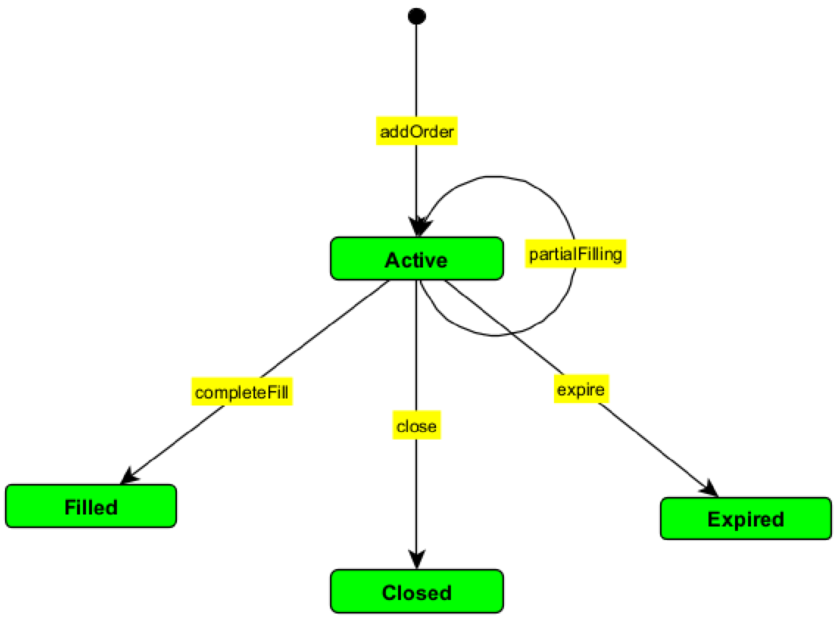
isShort a flag we use for simulation of pseudo-market-orders (see chapter 9)

During `addOrder` operation `DexCore` wraps an order within a `Position` instance. Position contains transient/mutable processing information about an order.

```
class Position(
                   order: Order,
                   market: Market,
                   account: Account,
                   blockchainTime: BlockchainTime, //blockchain time at the moment of␣
↪adding the order
                   realTime: SimulationTime
                 ) extends Comparable[Position] {

  private var amountSold: FPNumber = FPNumber.zero
  private var amountBought: FPNumber = FPNumber.zero
  val fillingReceipts = new ArrayBuffer[OrderFillingReceipt]
  private var xStatus: OrderStatus = OrderStatus.Active
)
```

Execution of an order is called **filling**. When `AAA` is the ask coin and `BBB` is the bid coin, we say that the direction of this order is `BBB -> AAA`, i.e. the trader wants to sell some amount of `BBB` coin and buy some amount of `AAA` coin. This conversion may happen in one or more steps. Every such step is called a **swap**. For a position `P` with direction `BBB -> AAA`, the amount of tokens `BBB` sold so far is what we call `amountFilled`. In general, an order will become completely filled when `amountFilled = amount`. It is the executor who runs the lifecycle of a position. Lifecycle of a position conforms to the following state machine:



As long as the position is **Active**, it is listed in the order book and is subject to further filling attempts. When a position leaves **Active** state, no more swaps for this position will be performed.

Example

Caution: in command-line mode **expirationTimepoint** is mocked and **isShort** is always set to false.

In the following script there are 3 traders, and two orders are placed on the exchange - one LIMIT order in `AAA/BBB` market and one STOP order in `AAA/CCC` market.

```
trader 01: deposit  11.120 AAA
trader 01: deposit  8.001 BBB
trader 01: deposit  20.005 CCC
trader 01: amm-init AAA=4.01 BBB=4.23
trader 01: amm-init AAA=3.5 CCC=9.12
trader 02: deposit  5.0 AAA
trader 02: deposit  5.0 BBB
trader 02: deposit  10.0 CCC
trader 02: +amm     AAA/CCC AAA=2.2
trader 01: open     #a01 AAA->BBB limit 1.0 [0.9]
trader 02: open     #a02 AAA->CCC stop  1.52 [000.010]
trader 01: close    #a01
trader 01: +amm     AAA/BBB AAA=1.112
trader 02: -amm     AAA/CCC 0.5
```

When executed with TEAL executor (see next chapter for the overview of supported executors), the execution log of this script looks like this:

```
---------------------------------- execution log -----------------------------------------
↪--------
line 00001|trader 001: deposit  11.1200000000000000 AAA
  introduced trader 1: auto-generated account address=578d-8c40-a64f-6dec
  [btime 0] deposit: accountAddress=578d-8c40-a64f-6dec amount=11.1200000000000000 AAA
line 00002|trader 001: deposit  8.0010000000000000 BBB
```

(continues on next page)

(continued from previous page)

```
  [btime 1] deposit: accountAddress=578d-8c40-a64f-6dec amount=8.0010000000000000 BBB
line 00003|trader 001: deposit  20.0050000000000000 CCC
  [btime 2] deposit: accountAddress=578d-8c40-a64f-6dec amount=20.0050000000000000 CCC
line 00004|trader 001: amm-init AAA=4.0100000000000000 BBB=4.2300000000000000
  [btime 3] provision: accountAddress=578d-8c40-a64f-6dec aAmount=4.0100000000000000 AAA␣
↪bAmount=4.2300000000000000 BBB drop=100.0000000000000000
line 00005|trader 001: amm-init AAA=3.5000000000000000 CCC=9.1200000000000000
  [btime 4] provision: accountAddress=578d-8c40-a64f-6dec aAmount=3.5000000000000000 AAA␣
↪bAmount=9.1200000000000000 CCC drop=100.0000000000000000
line 00006|trader 002: deposit  5.0000000000000000 AAA
  introduced trader 2: auto-generated account address=bb8b-4f43-cd02-af10
  [btime 5] deposit: accountAddress=bb8b-4f43-cd02-af10 amount=5.0000000000000000 AAA
line 00007|trader 002: deposit  5.0000000000000000 BBB
  [btime 6] deposit: accountAddress=bb8b-4f43-cd02-af10 amount=5.0000000000000000 BBB
line 00008|trader 002: deposit  10.0000000000000000 CCC
  [btime 7] deposit: accountAddress=bb8b-4f43-cd02-af10 amount=10.0000000000000000 CCC
line 00009|trader 002: +amm     AAA/CCC AAA=2.2000000000000000
  [btime 8] provision: accountAddress=bb8b-4f43-cd02-af10 aAmount=2.2000000000000000 AAA␣
↪bAmount=5.7325714285714285 CCC drop=62.8571428571428571
line 00010|trader 001: open     #a01 AAA->BBB limit 1.0000000000000000 [0.
↪9000000000000000]
  [btime 9] [rtime 2.0] open order: id=5bf7-082e-6fce-6801 type=Limit account=578d-8c40-
↪a64f-6dec askCoin=BBB bidCoin=AAA price=9/10 amount=1.0000000000000000 exptime=259201.0
  [btime 9] [rtime 4.0] created new position for order 5bf7-082e-6fce-6801 normalized-
↪amount=1.0000000000000000
  limit-execute: askCoin=BBB bidCoin=AAA
  limit-execute decision: partial filling of Position(Order(5bf7-082e-6fce-6801,Limit,
↪578d-8c40-a64f-6dec,BBB,AAA,9/10,1.0000000000000000,259201.0,false),Market[AAA/BBB],
↪Account-578d-8c40-a64f-6dec,9,3.0)
  old-normalized-amount=1.0000000000000000 new-normalized-amount=0.6731578947368422␣
↪delta=0.3268421052631578
  partial-filling [position 5bf7-082e-6fce-6801] (0.3268421052631578 AAA) ~~~~> (0.
↪2941578947368420 BBB)
line 00011|trader 002: open     #a02 AAA->CCC stop 1.5200000000000000 [0.
↪0100000000000000]
  [btime 10] [rtime 8.0] open order: id=51d9-58c4-4514-157a type=Stop account=bb8b-4f43-
↪cd02-af10 askCoin=AAA bidCoin=CCC price=1/100 amount=1.5200000000000000 exptime=259207.
↪0
  [btime 10] [rtime 10.0] created new position for order 51d9-58c4-4514-157a normalized-
↪amount=0.0152000000000000
  stop-execute: askCoin=AAA bidCoin=CCC
line 00012|trader 001: close    #a01
  [btime 11] close: askCoin=BBB bidCoin=AAA position=5bf7-082e-6fce-6801
line 00013|trader 001: +amm     AAA/BBB AAA=1.1120000000000000
  [btime 12] provision: accountAddress=578d-8c40-a64f-6dec aAmount=1.1120000000000000␣
↪AAA bAmount=1.0091804854368932 BBB drop=25.6407766990291267
line 00014|trader 002: -amm     AAA/CCC 0.5000000000000000
  [btime 13] liquidate: accountAddress=bb8b-4f43-cd02-af10 aAmount=0.0175000000000000␣
↪AAA bAmount=0.0455999999999999 CCC drop=0.5000000000000000
```

After execution, the final state of the DEX will be:

```
-------------------------------- final state dump ---------------------------------------
↪--------
per-coin stats
  AAA: reserve=983.8800000000000000 deposits=16.1200000000000000 in-pools=11.
↪1313421052631578 yield=0.3268421052631578 turnover=0.3268421052631578 fee=100.
↪0000000000000000
  BBB: reserve=986.9990000000000000 deposits=13.0010000000000000 in-pools=4.
↪9450225907000512 yield=-0.2941578947368420 turnover=0.2941578947368420 fee=-100.
↪0000000000000000
  CCC: reserve=969.9950000000000000 deposits=30.0050000000000000 in-pools=14.
↪8069714285714286 yield=0.0000000000000000 turnover=0.0000000000000000 fee=0.
↪0000000000000000
accounts
  trader-1 (578d-8c40-a64f-6dec)
    AAA: 2.1711578947368422 (free=2.1711578947368422 locked=0.0000000000000000)
    CCC: 10.8850000000000000 (free=10.8850000000000000 locked=0.0000000000000000)
    BBB: 3.0559774092999488 (free=3.0559774092999488 locked=0.0000000000000000)
  trader-2 (bb8b-4f43-cd02-af10)
    AAA: 2.8175000000000000 (free=2.8175000000000000 locked=0.0000000000000000)
    CCC: 4.3130285714285714 (free=2.7930285714285714 locked=1.5200000000000000)
    BBB: 5.0000000000000000 (free=5.0000000000000000 locked=0.0000000000000000)
    [>] a02 [btime 10 position 51d9-58c4-4514-157a] CCC->AAA Stop initial-amount 1.
↪5200000000000000 limit-price 0.0100000000000000 outstanding-amount 1.5200000000000000␣
↪bought/sold 0.0 sold/bought 0.0
markets
  AAA/BBB amm-price: 0.9075364077669903 amm-balance: AAA=5.4488421052631578 BBB=4.
↪9450225907000512
    stats
      active orders: 0
      asks volume (limit orders only): 0.0000000000000000
      bids volume (limit orders only): 0.0000000000000000
      liquidity tokens: 125.6407766990291267
      overhang [%]: 0.0
      bid-ask spread: 0.0000000000000000
    liquidity providers participation (aka 'drops')
      trader-1 = 125.6407766990291267
    order book - asks
      limits
      stops
    order book - bids
      limits
      stops
  AAA/CCC amm-price: 2.6057142857142857 amm-balance: AAA=5.6825000000000000 CCC=14.
↪8069714285714286
    stats
      active orders: 1
      asks volume (limit orders only): 0.0000000000000000
      bids volume (limit orders only): 0.0000000000000000
      liquidity tokens: 162.3571428571428571
      overhang [%]: 0.0
      bid-ask spread: 0.0000000000000000
    liquidity providers participation (aka 'drops')
      trader-2 = 62.3571428571428571
```

(continues on next page)

(continued from previous page)

```
      trader-1 = 100.0000000000000000
    order book - asks
      limits
      stops
    order book - bids
      limits
      stops
        [ ] 100.0000000000000000 btime=10 amount=1.5200000000000000 order-id=a02␣
↪account=trader-2 [position 51d9-58c4-4514-157a]
-------------------------------------- end ----------------------------------------------
↪--------
```

1.6.9 Representation of an order book

An order book is represented as a sorted sequence of positions. Sorting is based on compound key: `(order.exchangeRate, position.blockchainTime)`. In a normalized view of the market, the sorting is different for sellers and b

Example

This is a dump of a market state obtained by running Dexter in command-line mode (see chapter 15 for more details on command-line mode).

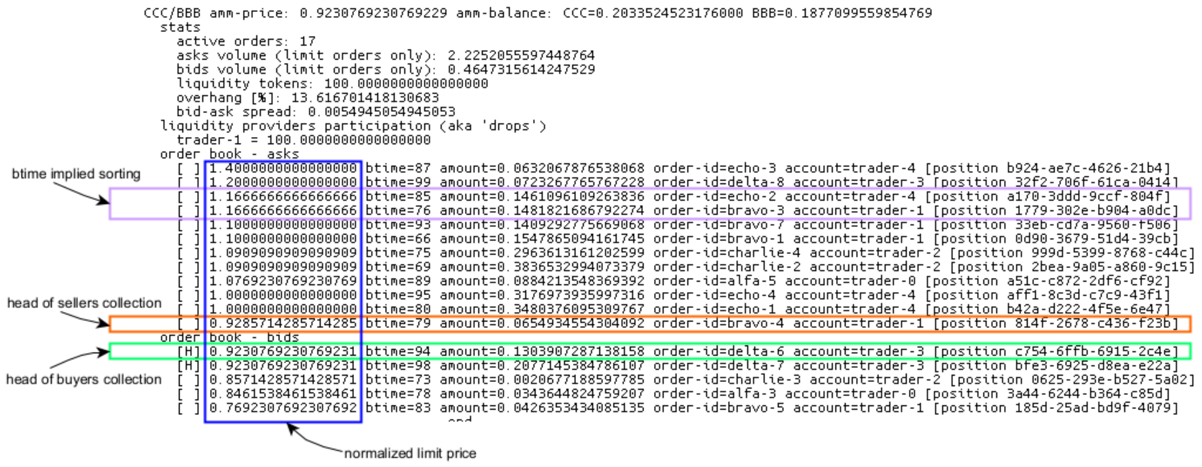
```
CCC/BBB amm-price: 0.9230769230769229 amm-balance: CCC=0.2033524523176000 BBB=0.
↪1877099559854769
  stats
    active orders: 17
    asks volume (limit orders only): 2.2252055597448764
    bids volume (limit orders only): 0.4647315614247529
    liquidity tokens: 100.0000000000000000
    overhang [%]: 13.616701418130683
    bid-ask spread: 0.0054945054945053
  liquidity providers participation (aka 'drops')
    trader-1 = 100.0000000000000000
  order book - asks
    [ ] 1.4000000000000000 btime=87 amount=0.0632067876538068 order-id=echo-3␣
↪account=trader-4 [position b924-ae7c-4626-21b4]
    [ ] 1.2000000000000000 btime=99 amount=0.0723267765767228 order-id=delta-8␣
↪account=trader-3 [position 32f2-706f-61ca-0414]
    [ ] 1.1666666666666666 btime=85 amount=0.1461096109263836 order-id=echo-2␣
↪account=trader-4 [position a170-3ddd-9ccf-804f]
    [ ] 1.1666666666666666 btime=76 amount=0.1481821686792274 order-id=bravo-3␣
↪account=trader-1 [position 1779-302e-b904-a0dc]
    [ ] 1.1000000000000000 btime=93 amount=0.1409292775669068 order-id=bravo-7␣
↪account=trader-1 [position 33eb-cd7a-9560-f506]
    [ ] 1.1000000000000000 btime=66 amount=0.1547865094161745 order-id=bravo-1␣
↪account=trader-1 [position 0d90-3679-51d4-39cb]
    [ ] 1.0909090909090909 btime=75 amount=0.2963613161202599 order-id=charlie-4␣
↪account=trader-2 [position 999d-5399-8768-c44c]
    [ ] 1.0909090909090909 btime=69 amount=0.3836532994073379 order-id=charlie-2␣
↪account=trader-2 [position 2bea-9a05-a860-9c15]
    [ ] 1.0769230769230769 btime=89 amount=0.0884213548369392 order-id=alfa-5␣
↪account=trader-0 [position a51c-c872-2df6-cf92]
```

(continues on next page)

(continued from previous page)

```
    [ ] 1.0000000000000000 btime=95 amount=0.3176973935997316 order-id=echo-4␣
↪account=trader-4 [position aff1-8c3d-c7c9-43f1]
    [ ] 1.0000000000000000 btime=80 amount=0.3480376095309767 order-id=echo-1␣
↪account=trader-4 [position b42a-d222-4f5e-6e47]
    [ ] 0.9285714285714285 btime=79 amount=0.0654934554304092 order-id=bravo-4␣
↪account=trader-1 [position 814f-2678-c436-f23b]
  order book - bids
    [H] 0.9230769230769231 btime=94 amount=0.1303907287138158 order-id=delta-6␣
↪account=trader-3 [position c754-6ffb-6915-2c4e]
    [H] 0.9230769230769231 btime=98 amount=0.2077145384786107 order-id=delta-7␣
↪account=trader-3 [position bfe3-6925-d8ea-e22a]
    [ ] 0.8571428571428571 btime=73 amount=0.0020677188597785 order-id=charlie-3␣
↪account=trader-2 [position 0625-293e-b527-5a02]
    [ ] 0.8461538461538461 btime=78 amount=0.0343644824759207 order-id=alfa-3␣
↪account=trader-0 [position 3a44-6244-b364-c85d]
    [ ] 0.7692307692307692 btime=83 amount=0.0426353434085135 order-id=bravo-5␣
↪account=trader-1 [position 185d-25ad-bd9f-4079]
```

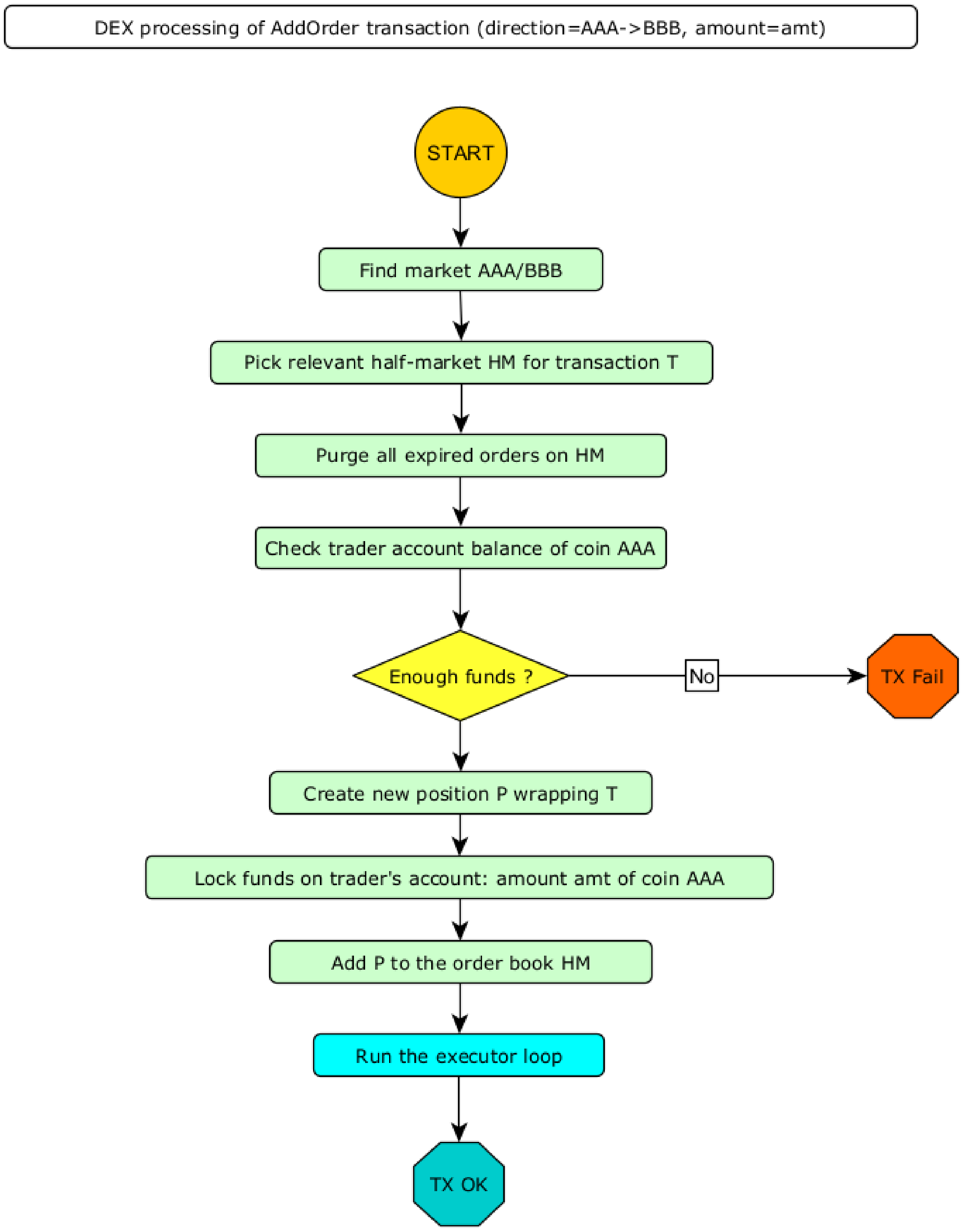
The market is presented in the normalized view. Order book is presented in a way similar to depth chart. Ordering of positions sequence corresponds to execution priority.Head of positions sequence for ask (sellers) is marked with red border. Head of positions sequence for bid (buyers) is marked with green border.Purple border highlights a situation where the limit price was the same for two positions and therefore the blockchain time decided about the priority. Smaller btime value means older position (because this is the blockchain time at registering that order).



```
CCC/BBB amm-price: 0.9230769230769229 amm-balance: CCC=0.2033524523176000 BBB=0.1877099559854769
  stats
    active orders: 17
    asks volume (limit orders only): 2.2252055597448764
    bids volume (limit orders only): 0.4647315614247529
    liquidity tokens: 100.0000000000000000
    overhang [%]: 13.616701418130683
    bid-ask spread: 0.0054945054945053
  liquidity providers participation (aka 'drops')
    trader-1 = 100.0000000000000000
  order book - asks
    [ ] 1.4000000000000000 btime=87 amount=0.0632067876538068 order-id=echo-3 account=trader-4 [position b924-ae7c-4626-21b4]
    [ ] 1.2000000000000000 btime=99 amount=0.0723267765767228 order-id=delta-8 account=trader-3 [position 32f2-706f-61ca-0414]
    [ ] 1.1666666666666666 btime=85 amount=0.1461096109263836 order-id=echo-2 account=trader-4 [position a170-3ddd-9ccf-804f]
    [ ] 1.1666666666666666 btime=76 amount=0.1481821686792274 order-id=bravo-3 account=trader-1 [position 1779-302e-b904-a0dc]
    [ ] 1.1000000000000000 btime=93 amount=0.1409292775669068 order-id=bravo-7 account=trader-1 [position 33eb-cd7a-9560-f506]
    [ ] 1.1000000000000000 btime=66 amount=0.1547865094161745 order-id=bravo-1 account=trader-1 [position 0d90-3679-51d4-39cb]
    [ ] 1.0909090909090909 btime=75 amount=0.2963613161202599 order-id=charlie-4 account=trader-2 [position 999d-5399-8768-c44c]
    [ ] 1.0909090909090909 btime=69 amount=0.3836532994073379 order-id=charlie-2 account=trader-2 [position 2bea-9a05-a860-9c15]
    [ ] 1.0769230769230769 btime=89 amount=0.0884213548369392 order-id=alfa-5 account=trader-0 [position a51c-c872-2df6-cf92]
    [ ] 1.0000000000000000 btime=95 amount=0.3176973935997316 order-id=echo-4 account=trader-4 [position aff1-8c3d-c7c9-43f1]
    [ ] 1.0000000000000000 btime=80 amount=0.3480376095309767 order-id=echo-1 account=trader-4 [position b42a-d222-4f5e-6e47]
    [ ] 0.9285714285714285 btime=79 amount=0.0654934554304092 order-id=bravo-4 account=trader-1 [position 814f-2678-c436-f23b]
  order book - bids
    [H] 0.9230769230769231 btime=94 amount=0.1303907287138158 order-id=delta-6 account=trader-3 [position c754-6ffb-6915-2c4e]
    [H] 0.9230769230769231 btime=98 amount=0.2077145384786107 order-id=delta-7 account=trader-3 [position bfe3-6925-d8ea-e22a]
    [ ] 0.8571428571428571 btime=73 amount=0.0020677188597785 order-id=charlie-3 account=trader-2 [position 0625-293e-b527-5a02]
    [ ] 0.8461538461538461 btime=78 amount=0.0343644824759207 order-id=alfa-3 account=trader-0 [position 3a44-6244-b364-c85d]
    [ ] 0.7692307692307692 btime=83 amount=0.0426353434085135 order-id=bravo-5 account=trader-1 [position 185d-25ad-bd9f-4079]
```

1.6.10 Execution of orders

The following sequence diagram illustrates the general algorithm of processing new orders on a DEX. Several technical details (such as updating various statistics) are stripped.


DEX processing of AddOrder transaction (direction=AAA->BBB, amount=amt)
START
Find market AAA/BBB
Pick relevant half-market HM for transaction T
Purge all expired orders on HM
Check trader account balance of coin AAA
Enough funds ?
No
TX Fail
Create new position P wrapping T
Lock funds on trader's account: amount amt of coin AAA
Add P to the order book HM
Run the executor loop
TX OK

Executor loop is the pluggable part of this algorithm. Current version of Dexter supports 3 executors (more can be added in the future). The next chapter is devoted to specify internals of all currently supported executors.

1.6.11 Summary of "price" concepts across Dexter

"Exchange rate" aka "price" seems to be a well known concept: if the price of bitcoin is 40 kUSD then I will need to pay 120 kUSD for 3 bitcoins. However, because of the symmetry, one could be interested in the dual point of view - the price of 1 dollar is 0.000025 bitcoin. Therefore we need to introduce **price direction**: for coins AAA and BBB, we say that price in direction $AAA \rightarrow BBB$ is the following fraction:

$$\frac{tokens\ AAA\ paid}{tokens\ BBB\ received}$$

… where tokens paid and received corresponds to an imaginary, infinitely small swap to be executed.

When we apply this intuition to our DEX model and more generally to the inner working of Dexter - things get a bit more complicated. It turns our that there are 5 different concepts of "price" in Dexter. We enumerate all these concepts below. For keeping the notation coherent, we assume that we are talking about a market with two coins `AAA` and `BBB` and the price direction considered is `AAA -> BBB`.

AMM price This is a market-level value. It reflects the official price as displayed for given market. It is derived from the current balance of the liquidity pool attached to this market. The exact formula is: `market.ammBalanceOf(AAA) / market.ammBalanceOf(BBB)`.

Limit price This is an order-level value. On creating an order, a trader defines the worst price he is willing to accept for executing given order. During the whole lifetime of a position p based on an order , the following condition must hold: $AP(P) <= LP(P)$, where $AP(P)$ is the achieved price for P calculated in the same direction as the direction of P, and $LP(P)$ is same-direction limit price for P.

Perceived price This is a market-level value. It reflects the price derived from the "external value" of coins. See chapter 8 for a detailed explanation of the external value model.

Swap price This is a swap-level value. Every swap represents an "atomic" exchange of tokens. As a swap means an already executed conversion of tokens, we know the exact amounts of tokens sold and bought in this swap. The way swap price is calculated of course depends on the direction of the order, which the swap belongs to. For example for an order with direction `AAA -> BBB`, the price in direction `AAA -> BBB` is calculated as `swap.sold / swap.bought`.

Achieved price This is an position-level value. It is the average price achieved so far given all the swaps executed in the context of given position. The way achieved price is calculated depends on the direction of the position vs the direction of the price. As we want to calculate the price in direction *AAA -> BBB`*, this works as follows:

- for a position `p` with direction `AAA -> BBB` achieved price defined as `p.soldSoFar / p.boughtSoFar`.
- for a position `p` with direction `BBB -> AAA` achieved price defined as `p.boughtSoFar / p.soldSoFar`.

1.7 07. Executors

Because a DEX operates on top of a blockchain, it follows the basic principle of blockchains: there is a decentralized ledger of transactions which validators are building. So, one can see a DEX as an evolving state of a sequential computer.

As was explained in chapter 6, we fixed a certain set of transaction types and we fixed their semantics for all but one. The single exceptional case being `add-order` transaction.

On the technical level, a single `add-order` execution must be resolved to a sequence of swaps. This is the job of an **executor**. We consider an executor to be pluggable part of DEX implementation.

In this chapter we specify the algorithms of all executors currently supported in Dexter. The dynamic behaviour of these algorithms and especially the comparative measurements of their statistics is the primary motivation behind creating Dexter.

1.7.1 Space of executors research

The space of all possible executor is vast. Dexter was created as a proof of concept tool for investigating the contents of this space.

However, in the current version of Dexter, we limit our attention to rather "simplistic" class of executors, specifically, we require executors to be compliant with the following rules:

Rule #1: Equality of traders Swaps chronology can depend only on the state of markets (i.e. it is not allowed that it depends on the state of traders).

This rule means that the exchange in not "biased" on trader's identity or other trader properties (such as wealth of a trader for example). In lame terms it means that all traders are considered "equally important" by the executor - i.e. from the perspective of an executor, any trader is seen as "just an account number" (similar concept to "all citizens are considered equal by law in a democracy"). In particular - current balance of trader's account, past history of his trading activity and trader's name - all this cannot influence the path of DEX evolution that an executor is creating.

Rule #2: Isolation of markets Arrival of a new order generates swaps only on the market where this order belongs to.

Rule #3: Isolation of traders Every swap involves only one order.

Rule #4: Isolation of swaps An executor makes next-swap decision based only on the current state of the market and number of swaps executed in the current executor loop. The intention here is that the executor is not allowed to decide based on the history of executes swaps.

Rule #5: Natural trading priority Execution prioritizes traders happy to accept less attractive exchange rate (i.e. an exchange rate that gives him less profit). In case of a tie, a trader waiting longer is served first.

Rule #6: Funds locking Trader can place an order on the DEX only if the amount of tokens to be sold is covered by the balance of his account. Upon placing an order, the executor will issue a "lock" on the amount of tokens declared in the order as sell amount. Locked tokens cannot be reused in another order. Locked tokens get unlock on order expiration or cancellation.

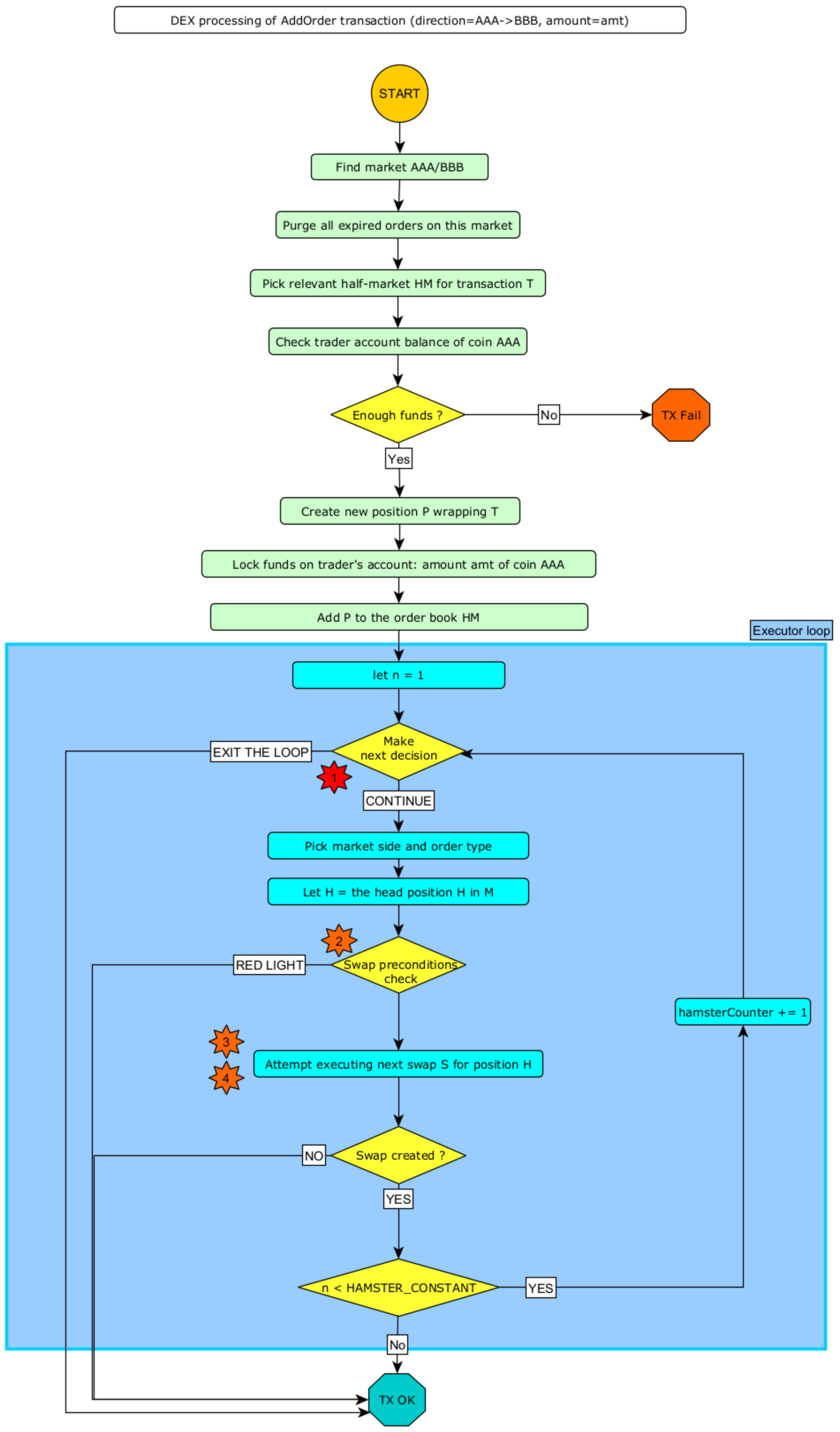
Motivation for above rules comes from various sources:

- **Equality of traders** and **Natural trading priority** seem to be very natual and go along the general expectations of the wide public; any DEX violating these rules would be really a weird one.
- **Funds locking** is a business-level decision; DEXes violating this rule are probably quite useful and interesting, although it must be said that their trading mechanics would be more distant from Forex tradition. Therefore the motivation behind funds locking can be seen as "let's keep things mentally close to Forex community for now" (please notice that Dexter is all about simulating hybrid exchanges, so the analogy to Forex is going to be always around).
- All 3 "isolation" rules delineate the boundary of "simplest executors one can consider"; anything beyond this region immediately becomes complex and sophisticated. From this perspective the mission of Dexter can be understood as "let's see how far we can get with those simplistic executors before we delve into sophisticated ones". A good motivation for dropping **isolation of markets** would be building a DEX where arbitrage is impossible. A good motivation for dropping **isolation of traders** would be borrowing from Forex the idea of direct trader-to-trader swaps, i.e. bypassing the AMM interaction when possible.

1.7.2 Executor loop overview

Because we assume **Rule #2: Isolation of markets**, once a new order arrives to the DEX, we are going to process the order book only for the market where the new order belongs. Let us fix the coins to be AAA and BBB, so the market is AAA/BBB and orders have direction $AAA \rightarrow BBB$ or $BBB \rightarrow AAA$. We will further refer to this market as `market` (see the scripts below).

Thanks to the rules enumerated in previous chapter, the job of an executor loop can be largely simplified. Let us outline the template of such loop by extending the activity diagram in from previous chapter:


DEX processing of AddOrder transaction (direction=AAA->BBB, amount=amt)
START
Find market AAA/BBB
Purge all expired orders on this market
Pick relevant half-market HM for transaction T
Check trader account balance of coin AAA
Enough funds ?
No
TX Fail
Yes
Create new position P wrapping T
Lock funds on trader's account: amount amt of coin AAA
Add P to the order book HM
Executor loop
let n = 1
Make next decision
EXIT THE LOOP
1
CONTINUE
Pick market side and order type
Let H = the head position H in M
2
Swap preconditions check
RED LIGHT
hamsterCounter += 1
3
4
Attempt executing next swap S for position H
Swap created ?
NO
YES
n < HAMSTER_CONSTANT
YES
No
TX OK

Places marked above with red asterisk are where non-trivial logic must be plugged-in:

1: Making next executor decision At this point, one of two half-markets must be picked. This is equivalent to selecting a direction: $AAA \to BBB$ or $BBB \to AAA$. In the context of "oriented" market, this means selecting "asks" or "bids" side of the order book. Additionally, the order type (`Limit` vs `Stop`) must be selected here. An executor is free to give up at this point, if the state of the market does not allow for executing of any order.

2: Checking swap preconditions At this point swap preconditions are checked. Red light will abort swap creation and terminate the executor loop.

3: Calculating swap amount and executing the swap Deciding on amounts of the next swap to be executed. Because of **Rule #5: Natural trading priority**, the next swap to be executed must be for the head position on the positions collection (because the positions list is ordered by price-then-order-time).

4: Post-swap assertions Extension point for plugging-in diagnostic assertions to be checked after every swap.

1.7.3 Arithmetic precision problems and their solution

Internal working of the executor is vulnerable to "strange" effects caused by imperfectness of computer arithmetic. This effects generally disrupt the operation of mathematical definitions of the executor. Two particular problems are:

- **When (at least one side of) the AMM balance becomes small enough, integer rounding effects can cause significant** errors in calculated swap amounts.
- When calculated swap amounts are small enough, integer rounding may cause limit-price invariant to fail.

To avoid such anomalies we generally apply a simple approach:

1. Enforce that AMM balances are always above certain AMM_MIN_BALANCE (which is a parameter).
2. Enforce that order amount is always above certain TRADING_MIN_AMOUNT (which is a parameter).
3. Enforce that swap amounts are always above certain SWAP_MIN_AMOUNT (which is a parameter).

To work as expected, above parameters must be set accordingly to the arithmetic precision used by the implementation of DEX. Please notice that arithmetic precision is also limited in the fixed-point arithmetic - because of the necessary rounding in operations such as multiplication.

For example if the arithmetic precision is at the order 1e-18, then "reasonable" values for above params could be:

- AMM_MIN_BALANCE = 1e-14
- TRADING_MIN_AMOUNT = 1e-8
- SWAP_MIN_AMOUNT = 1e-10

1.7.4 Executor loop details

The general pattern of open-order processing is implemented in class `ExecutorTemplate`. The executor loop is part of this:

```
def open(order: Order): Unit = {

  //log diagnostic info "new order arrived"
  if (coreCallsDump.isDefined)
    coreCallsDump.get.print(s"[btime $blockchainTime] [rtime ${clock.apply()}] open␣
↪order: id=${order.id} type=${order.orderType} account=${order.accountAddress}" +
      s" askCoin=${order.askCoin} bidCoin=${order.bidCoin} price=${order.exchangeRate.
↪toDouble} amount=${order.amount} exptime=${order.expirationTimepoint}")
```

(continues on next page)

(continued from previous page)

```
  //check if the account address if valid
  precondition(hash2account.contains(order.accountAddress), s"unknown account: ${order.
↪accountAddress}")

  //decode account address to account instance
  val account: Account = hash2account(order.accountAddress)

  //find relevant market and half-market
  val coinPairAB: CoinPair = CoinPair(order.askCoin, order.bidCoin)
  val halfMarketAB: HalfMarket = coinPair2HalfMarket(coinPairAB)
  val market: Market = coinPair2Market(coinPairAB.normalized)

  //purge expired orders on this market
  purgeExpiredPositionsOnMarket(market)

  //check if the trader has enough funds to place this order
  precondition (order.amount <= account.getFreeBalanceFor(order.sellCoin), s"order␣
↪amount exceeds free balance for this account and coin: ${account.
↪getFreeBalanceFor(order.sellCoin)}")

  //create new position instance (wrapping the order)
  val position = Position(order, market, account, blockchainTime, clock.apply())
  hash2position += order.id -> position

  //lock amount of tokens on trader's account corresponding to sell amount of this order
  account.addPosition(position)

  //add position to the order book
  halfMarketAB.addPosition(position)

  //update statistics
  market.onPositionOpened(position)

  //log diagnostic info "new position added"
  if (coreCallsDump.isDefined)
    coreCallsDump.get.print(s"[btime $blockchainTime] [rtime ${clock.apply()}] created␣
↪new position for order ${position.id} normalized-amount=${position.normalizedAmount}␣
↪normalized-price=${position.normalizedLimitPrice.toDouble}")

  //executor loop
  var n = 1
  var lastDecision: Option[(MarketSide, OrderType)] = executorNextDecision(market,␣
↪position)
  var lastExecutionWasOk: Boolean = true
  while (n <= hamsterConstant & lastDecision.nonEmpty && lastExecutionWasOk && market.
↪isAmmBalanceAboveTradingMinimum) {
    lastExecutionWasOk = lastDecision match {
      case Some((MarketSide.Asks, OrderType.Limit)) => attemptCreatingNextSwap(n, market,
↪ market.halfMarketAsks, OrderType.Limit)
      case Some((MarketSide.Bids, OrderType.Limit)) => attemptCreatingNextSwap(n, market,
↪ market.halfMarketBids, OrderType.Limit)
```

(continues on next page)

(continued from previous page)

```
      case Some((MarketSide.Asks, OrderType.Stop)) => attemptCreatingNextSwap(n, market,␣
↪market.halfMarketAsks, OrderType.Stop)
      case Some((MarketSide.Bids, OrderType.Stop)) => attemptCreatingNextSwap(n, market,␣
↪market.halfMarketBids, OrderType.Stop)
    }
    n += 1
    if (lastExecutionWasOk)
      lastDecision = executorNextDecision(market, position)
  }
}
```

Extension points - marked previously as red asterisks on the diagram - are encoded as corresponding abstract methods. For implementing a specific executor, one needs to provide implementation of these methods only:

```
def executorNextDecision(market: Market, newPositionThatTriggeredTheLoop: Position):␣
↪Option[(MarketSide, OrderType)]

def swapPreconditionsCheck(
                            market: Market,
                            halfMarketAB: HalfMarket,
                            askCoin: Coin, //buy coin
                            bidCoin: Coin, //sell coin
                            position: Position, //position for which the swap is to be␣
↪executed
                            a: FPNumber, //AMM balance of ask coin
                            b: FPNumber, //AMM balance of bid coin
                            r: Fraction, //limit price as declared in the order
                            ammPrice: Fraction): Decision

//returns (sellAmount, buyAmount) - where sell/buy meaning is from the perspective of␣
↪the trader
def calculateSwapAmounts(
                          market: Market,
                          halfMarketAB: HalfMarket,
                          askCoin: Coin, //buy coin
                          bidCoin: Coin, //sell coin
                          position: Position, //position for which the swap is to be␣
↪executed
                          a: FPNumber, //AMM balance of ask coin
                          b: FPNumber, //AMM balance of bid coin
                          r: Fraction, //limit price as declared in the order
                          ammPrice: Fraction): (FPNumber, FPNumber)

def postSwapAssertions(
                        market: Market,
                        halfMarket: HalfMarket,
                        position: Position,
                        sellAmount: FPNumber,
                        sellCoin: Coin,
                        buyAmount: FPNumber,
                        buyCoin: Coin,
                        ammPriceBeforeSwap: Fraction): Unit
```

Most complex part is the “attempt to create next swap”. This was not covered in detail on the diagram above:

```
def attemptCreatingNextSwap(hamsterLoopIteration: Int, market: Market, halfMarketAB:␣
↪HalfMarket, orderType: OrderType): Boolean = {
  val askCoin: Coin = halfMarketAB.coinPair.left
  val bidCoin: Coin = halfMarketAB.coinPair.right

  lazy val headPosition: Position = orderType match {
    case OrderType.Limit => halfMarketAB.limits.head
    case OrderType.Stop => halfMarketAB.stops.head
  }

  val a: FPNumber = market.ammBalanceOf(askCoin)
  val b: FPNumber = market.ammBalanceOf(bidCoin)
  val r: Fraction = headPosition.exchangeRate
  val ammPrice: Fraction = market.currentPriceDirected(askCoin, bidCoin)

  if (coreCallsDump.isDefined)
    coreCallsDump.get.print(s"limit-execute: (iteration $hamsterLoopIteration) askCoin=
↪$askCoin [balance $a] bidCoin=$bidCoin [balance $b] directed-amm-price=${ammPrice.
↪toDouble}" +
      s" head-position=${headPosition.id} outstanding-amount=${headPosition.
↪outstandingAmount} limit-price=${FPNumber.fromFraction(r)}")

  swapPreconditionsCheck(market, halfMarketAB, askCoin, bidCoin, headPosition, a, b, r,␣
↪ammPrice) match {
    case Decision.GREEN =>
      val (sellAmount, buyAmount): (FPNumber, FPNumber) = calculateSwapAmounts(market,␣
↪halfMarketAB, askCoin, bidCoin, headPosition, a, b, r, ammPrice)

      //negative amounts are considered a bug in 'calculateSwapAmounts'
      assert(sellAmount >= FPNumber.zero && buyAmount >= FPNumber.zero, s
↪"calculateSwapAmounts() returned negative value: sellAmount=$sellAmount buyAmount=
↪$buyAmount")

      //avoid "nano" swaps (below SWAP_MIN_AMOUNT), unless this is a complete filling␣
↪case
      if (sellAmount <= swapMinAmount || buyAmount <= swapMinAmount)
        if (headPosition.amount > swapMinAmount)
          return false

      //zero amount is not considered a bug in 'calculateSwapAmounts', but we will not␣
↪proceed with swap execution
      if (sellAmount == FPNumber.zero || buyAmount == FPNumber.zero)
        return false

      //we need to distinguish between partial filling and complete filling
      if (headPosition.amount == sellAmount) {
        //case 1: complete filling
        if (coreCallsDump.isDefined)
          coreCallsDump.get.print(s"executor decision: complete filling of $headPosition
↪")
        val normalizedAmount: FPNumber = headPosition.normalizedAmount
```

(continues on next page)

(continued from previous page)

```
        headPosition.registerCompleteFilling(sold = sellAmount, bought = buyAmount,␣
↪blockchainTime)
        orderType match {
          case OrderType.Limit => halfMarketAB.limits.removeElementAtIndex(0)
          case OrderType.Stop => halfMarketAB.stops.removeElementAtIndex(0)
        }
        market.onPositionFilled(headPosition, normalizedAmount)
        headPosition.account.removePosition(headPosition)
        dumpFillingInfo(headPosition, sold = sellAmount, bought = buyAmount,␣
↪isCompleteFilling = true)
      } else {
        //case 2: partial filling
        if (coreCallsDump.isDefined)
          coreCallsDump.get.print(s"executor decision: partial filling of $headPosition")
        val oldNormalizedAmount: FPNumber = headPosition.normalizedAmount
        headPosition.registerIncompleteFilling(sold = sellAmount, bought = buyAmount,␣
↪blockchainTime)
        val delta: FPNumber = oldNormalizedAmount - headPosition.normalizedAmount
        market.onPositionPartiallyFilled(headPosition, delta)
        dumpFillingInfo(headPosition, sold = sellAmount, bought = buyAmount,␣
↪isCompleteFilling = false)
      }

      //update statistics
      swapExecutionsCounter += 1

      //update account balances (to reflect the swap execution)
      headPosition.account.updateBalance(bidCoin, sellAmount.negated)
      headPosition.account.updateBalance(askCoin, buyAmount)

      //update the liquidity pool (to reflect the swap execution)
      market.updateLiquidityPool(bidCoin, sellAmount)
      market.updateLiquidityPool(askCoin, buyAmount.negated)

      //accumulation of some statistics for the AMM
      tokensExchangedIn.increment(bidCoin, sellAmount)
      tokensExchangedOut.increment(askCoin, buyAmount)

      //extension point for more assertions
      postSwapAssertions(market, halfMarketAB, headPosition, sellAmount, bidCoin,␣
↪buyAmount, askCoin, ammPrice)

      //we completed the swap with success
      return true

    case Decision.RED =>
      //abort the swap
      return false
  }
}
```

1.7.5 Variant 1: TEAL executor

This executor is based on a proprietary algorithm created in Onomy Protocol. This executor follows this TLA+ specification:

https://github.com/onomyprotocol/specs/

On top of the specification we apply the "minimal trading balance" check on the AMM level. We just do not allow either size of the liquidity pool to

1. Executor next decision

We select the same half-market where the position belongs.

In this variant, the value of HAMSTER_CONSTANT is hardcoded to 1, so the executor loop has always at most 1 iteration.

```
override def executorNextDecision(market: Market, newPositionThatTriggeredTheLoop:␣
↪Position): Option[(MarketSide, OrderType)] =
  Some(newPositionThatTriggeredTheLoop.normalizedMarketSide,␣
↪newPositionThatTriggeredTheLoop.orderType)
```

2: Swap preconditions check

The head of order book is ready for next swap if the current value of ammPrice exceeds the limit price of the order. Please notice that we compare prices calculated in the same direction.

```
override def swapPreconditionsCheck(
                                     market: Market,
                                     halfMarketAB: HalfMarket,
                                     askCoin: Coin,
                                     bidCoin: Coin,
                                     position: Position,
                                     a: FPNumber,
                                     b: FPNumber,
                                     r: Fraction,
                                     ammPrice: Fraction): Decision = {
  position.orderType match {
    case OrderType.Limit => if (ammPrice > r) Decision.GREEN else Decision.RED
    case OrderType.Stop => if (ammPrice < r) Decision.GREEN else Decision.RED
  }
}
```

3: Calculate swap amounts

The "magic" formula of swap creation is defined here.

```
override def calculateSwapAmounts(
                                   market: Market,
                                   halfMarketAB: HalfMarket,
                                   askCoin: Coin,
                                   bidCoin: Coin,
                                   position: Position,
```

(continues on next page)

(continued from previous page)

```
                                    a: FPNumber,
                                    b: FPNumber,
                                    r: Fraction,
                                    ammPrice: Fraction): (FPNumber, FPNumber) =
  position.orderType match {
    case OrderType.Limit =>
      val maxBidAmt: FPNumber = (a - b ** r) ** ((r + 1).reciprocal)
      val strikeBidAmt: FPNumber = FPNumber.min(position.outstandingAmount, maxBidAmt)
      val strikeAskAmt: FPNumber = strikeBidAmt ** r
      (strikeBidAmt, strikeAskAmt)
    case OrderType.Stop =>
      val minMemberABal: FPNumber = a / FPNumber.fromLong(2)
      val maxMemberBBal: FPNumber = a + b - minMemberABal
      val maxMemberBAmt: FPNumber = maxMemberBBal - b
      val strikeBidAmt: FPNumber = FPNumber.min(position.outstandingAmount,␣
↪maxMemberBAmt)
      val strikeAskAmt: FPNumber = (strikeBidAmt * (a - strikeBidAmt)) / (b +␣
↪strikeBidAmt)
      (strikeBidAmt, strikeAskAmt)
  }
```

1.7.6 Variant 2: TURQUOISE executor

This executor is based on the idea that we execute orders always using the limit price as declared in the order itself - as long as the AMM-price allows to do so without introducing loss on DEX side. This approach is rather “unusual” when compared to FOREX-style exchanges, where the swap price is always the current market price.

Here we allow the HAMSTER_CONSTANT to be arbitrary number bigger than zero.

Caution: STOP ORDERS are not supported.

Math derivation

We consider an execution of some limit order $BBB \to AAA$, i.e. where BBB is the bid coin and AAA is the ask coin. In effect of the execution, $x : AAA$ will be received from AMM and $y : BBB$ will be given to AMM. After the execution, the new state of the AMM will be:

$$a - x : AAA, b + y : BBB$$

An order contains a declared limit price r. The execution of an order is only allowed when $ammprice \geq r$.

Additionally, we want to keep the constant conversion rate for every order and we want it to be equal to the declared limit price. In other words we want the following condition to hold:

$$\frac{x}{y} = r$$

Let’s assume that we have an order for which the condition $ammprice \geq r$ is true. We want to find the maximal amount of swap which is possible.

For the maximal swap, the inequality will turn into equality, hence we will have:

$$ammprice = r$$

The ammprice after successful execution of the order will be:

$$ammprice = \frac{a - x}{b + y}$$

Effectively, we arrive to the following system of equations (where x and y are unknown):

$$\begin{cases} \frac{a - x}{b + y} = r \\ \frac{x}{y} = r \end{cases}$$

Solving this leads to:

$$\begin{cases} x = \frac{a - br}{2} \\ y = \frac{a - br}{2r} \end{cases}$$

1. Executor next decision

We check head positions of bids and asks half-markets to understand if they are possibly ready to execute next swap, given the current AMM price value. Only-ask, only-bid and none-of-them are easy cases - we select the only side which is possible. The only tricky case is when both head bid and head ask could be picked for execution in the next step. In such case we pick the one with bigger overhang.

In case of a tie, we use a boolean variable `flipper`, to pick one side, and then we negate `flipper` so that in the case of next tie, the decision will be in favour of the other side.

```
private var flipper: Boolean = false

override def executorNextDecision(market: Market, newPositionThatTriggeredTheLoop:␣
↪Position): Option[(MarketSide, OrderType)] = {
  if (market.limitOrderBookAsks.isEmpty && market.limitOrderBookBids.isEmpty)
    return None

  if (market.limitOrderBookBids.isEmpty)
    return Some((MarketSide.Asks, OrderType.Limit))

  if (market.limitOrderBookAsks.isEmpty)
    return Some((MarketSide.Bids, OrderType.Limit))

  val topBid: Fraction = market.limitOrderBookBids.head.normalizedLimitPrice
  val bottomAsk: Fraction = market.limitOrderBookAsks.head.normalizedLimitPrice
  val ammPrice: Fraction = market.currentPriceNormalized

  if (topBid <= ammPrice && ammPrice <= bottomAsk)
    return None

  if (bottomAsk < ammPrice && topBid <= ammPrice)
    return Some((MarketSide.Asks, OrderType.Limit))

  if (topBid > ammPrice && bottomAsk >= ammPrice)
    return Some((MarketSide.Bids, OrderType.Limit))

  val bidOverhang = topBid - ammPrice
```

(continues on next page)

(continued from previous page)

```
  val askOverHang = ammPrice - bottomAsk

  if (bidOverhang > askOverHang)
    return Some((MarketSide.Bids, OrderType.Limit))

  if (bidOverhang < askOverHang)
    return Some((MarketSide.Asks, OrderType.Limit))

  //they are equal, so we pick one pointed by the flipper
  flipper = ! flipper
  flipper match {
    case true => return Some((MarketSide.Bids, OrderType.Limit))
    case false => return Some((MarketSide.Asks, OrderType.Limit))
  }
}
```

2: Swap preconditions check

Same as in TEAL variant, but without stop orders support.

```
override def swapPreconditionsCheck(
                                     market: Market,
                                     halfMarketAB: HalfMarket,
                                     askCoin: Coin,
                                     bidCoin: Coin,
                                     position: Position,
                                     a: FPNumber,
                                     b: FPNumber,
                                     r: Fraction,
                                     ammPrice: Fraction): Decision =
  if (ammPrice > r)
    Decision.GREEN
  else
    Decision.RED
```

3: Calculate swap amounts

Following the math derivation above.

```
override def calculateSwapAmounts(
                                   market: Market,
                                   halfMarketAB: HalfMarket,
                                   askCoin: Coin,
                                   bidCoin: Coin,
                                   position: Position,
                                   a: FPNumber,
                                   b: FPNumber,
                                   r: Fraction,
                                   ammPrice: Fraction): (FPNumber, FPNumber) = {
```

(continues on next page)

(continued from previous page)

```
  val x: BigInt = r.numerator
  val y: BigInt = r.denominator
  val maxBidAmt: FPNumber = FPNumber((a.pips * y - b.pips * x) / (2 * x))
  val maxAmountOfBidCoinThatWillNotDrainAmmBelowMargin: FPNumber = (market.
↪ammBalanceOf(askCoin) -  ammMinBalance) ** r.reciprocal
  val strikeBidAmt: FPNumber = FPNumber.min(FPNumber.min(position.outstandingAmount,␣
↪maxBidAmt), maxAmountOfBidCoinThatWillNotDrainAmmBelowMargin)
  val strikeAskAmt: FPNumber = strikeBidAmt ** r
  return (strikeBidAmt, strikeAskAmt)
}
```

1.7.7 Variant 3: UNISWAP_HYBRID executor

1.8 08. External value of coins

UNDER CONSTRUCTION

1.9 09. Trader models

UNDER CONSTRUCTION

1.10 10. Simulation structure

UNDER CONSTRUCTION

1.11 11. Configuration of experiments

UNDER CONSTRUCTION

1.12 12. Dashboard

UNDER CONSTRUCTION

1.13 13. Experiment params reference

UNDER CONSTRUCTION

1.14 14. Known problems

UNDER CONSTRUCTION

1.15 15. Command-line mode

UNDER CONSTRUCTION