Market Microstructure Simulator: Strategy Definition Language

Anton Kolotaev

Chair of Quantitative Finance, École Centrale Paris

anton.kolotaev@gmail.com

March 3, 2014

Overview

Evolution of the simulator I

1. Initial C++ version was developed in 2009-2011 by Riadh Zaatour. In this version a user implements strategy logic in C++. Though this version was quite easy to learn and understand, it had problems with flexibility and scalability.
2. In order to improve its extensibility and performance the simulator was rewritten using C++ template metaprogramming techniques by Anton Kolotaev in 2012. Python bindings to it were implemented using Boost.Python. Unfortunately the price for providing high extensibility with no overhead was quite high: in order to use it a proficiency in C++ template metaprogramming was required.

Evolution of the simulator II

1. In order to make the simulator easy to to start work with, a Python version with a Web interface was developed in 2013. Karol Podkanski implemented number of trading strategies and indicators during his internship at summer 2013. Though this version gave a lot of insights on how a good modular design for market simulation software should be implemented, it showed that a lot of syntax noise appears in strategy description and the development becomes very error-prone because of the dynamic typing nature of Python.
2. In October 2013 a decision to introduce a strategy definition language and a compiler for it was taken.

Design goals

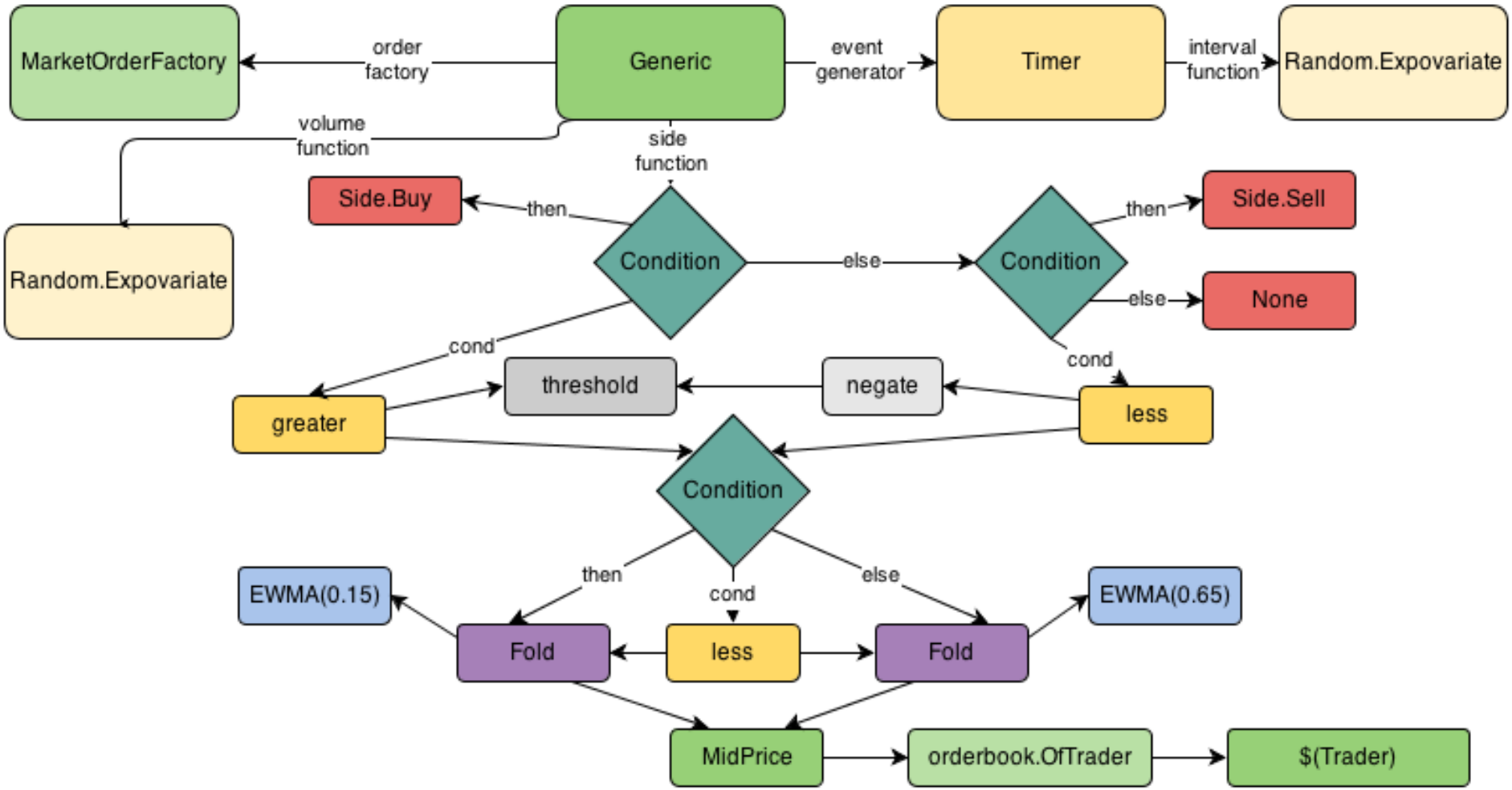
1. **Flexibility**. A simulation library must have a very modular design in order to provide a high level of flexibility to the user. This requirement comes from the original purpose of a simulation as a test bed for experiments with different models and parameters.
2. **Used-friendliness**. Since a typical simulation model is composed of many hundreds and thousands blocks, it is very important to provide a simple way for user to tell how a behaviour wanted differs from the default one. Simulator API should be friendly to modern IDEs
3. **Error checking**. A simulator should detect incorrect parameter values as soon as possible in order to accelerate simulation development cycle and facilitate refactorings.
4. **Performance**. A used should be allowed to choose between small start-up time and high simulation speed.

Modular Design

Representing a simulation model as a network of modules communicating by messages is a widely accepted practice for DES systems (e.g. Omnet++ or ns-2 for telecommunication network simulations).
Module may have **parameters** that are used to adjust their behaviour.
Modules may be organized into **hierarchy** in order to facilitate construction of large-scale simulation and it is useful to distinguish two sorts of modules:

1. **Simple modules** provide functionality which is considered elementary (and there is no reason to reuse part of it to implement other modules).
2. **Compound modules** combine together other modules in some way and define their parameters based on its own parameters. Compound module behaviour is just a composition of its constituting modules behaviours.

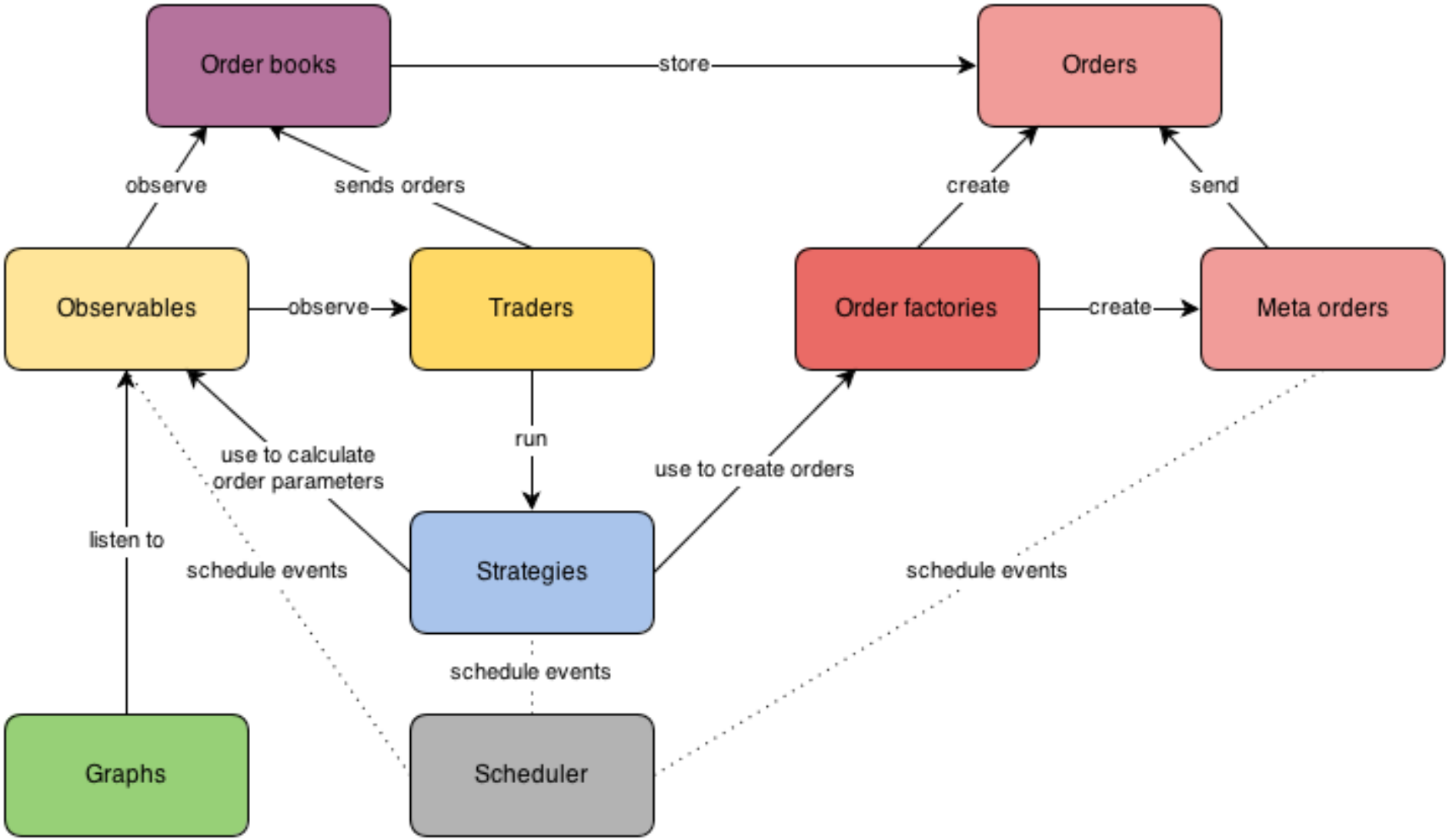
Modular design sample



Installation

- OS supported: Linux, Mac OS X, Windows
- Browsers supported: Chrome, Firefox, Safari, Opera
- Python 2.7
- Python packages can be installed using `pip` or `easyinstall`:
  - Veusz (for graph plotting)
  - Flask (to run a Web-server)
  - Blist (sorted collections used by ArbitrageTrader)
- Source code downloadable from SourceForge

Simulator components



Scheduler

- Main class for every discrete event simulation system.
- Maintains a set of actions to fulfill in future and launches them according their action times: from older ones to newer.

Interface:

- Event scheduling:
  - `schedule(actionTime, handler)`
  - `scheduleAfter(dt, handler)`
- Simulation control:
  - `workTill(limitTime)`
  - `advance(dt)`
  - `reset()`

Order book

- Represents a single asset traded in some market (Same asset traded in different markets would be represented by different order books)
- Matches incoming orders
- Stores unfulfilled limit orders in two order queues (`Asks` for sell orders and `Bids` for buy orders)
- Corrects limit order price with respect to tick size
- Imposes order processing fee
- Supports queries about order book structure
- Notifies listeners about trades and price changes

Order book for a remote trader

- Models a trader connected to a market by a communication channel with non-negligible latency
- Introduces delay in information propagation from a trader to an order book and vice versa (so a trader has outdated information about market and orders are sent to the market with a certain delay)
- Assures correct order of messages: older messages always come earlier than newer ones

Basic orders

Orders supported internally by an order book:

- `Market(side, volume)`
- `Limit(side, price, volume)`
- `Cancel(limitOrder)`

Limit and market orders notifies their listeners about all trades they take part in. Factory functions are usually used in order to create orders.

Meta orders

Follow order interface from trader's perspective (so they can be used instead of basic orders) but behave like a sequence of base orders from an order book point of view.

- `Iceberg(volumeLimit, orderToSplit)` splits `orderToSplit` to pieces with volume less than `volumeLimit` and sends them one by one to an order book ensuring that only one order at time is processed there
- `AlwaysBest(volume, limitOrderFactory)` creates a limit-like order with given volume and the most attractive price, sends it to an order book and if the order book best price changes, cancels it and resends with a better price
- `WithExpiry(lifetime, limitOrderFactory)` sends a limit-like order and after `lifetime` cancels it
- `LimitMarket(limitOrderFactory)` is like `WithExpiry` but with `lifetime` equal to 0

Traders

Single asset traders

- send orders to order books
- bookkeep their position and balance
- run a number of trading strategies
- notify listeners about trades done and orders sent

Single asset traders operate on a single or multiple markets. Multiple asset traders are about to be added.

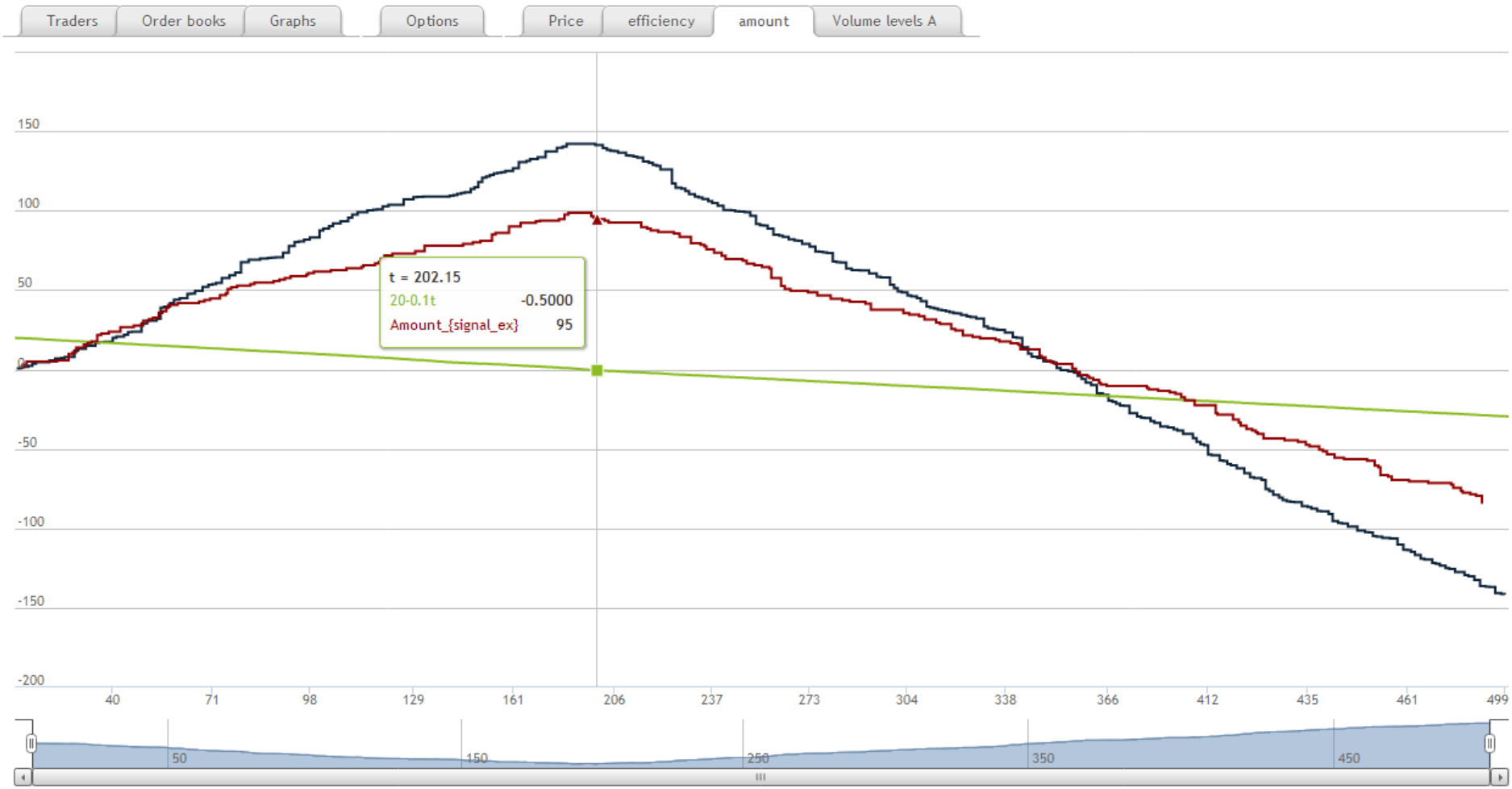
Generic strategy

Generic strategy that wakes up on events given by `eventGen`, chooses side of order to create using `sideFunc` and its volume by `volumeFunc`, creates an order via `orderFactory` and sends the order to the market using its trader

```
class Generic(Strategy):
    def __init__(self):
        event.subscribe(self.eventGen, self._wakeUp, self)

    def _wakeUp(self, _):
        if not self._suspended:
          # determine side and parameters of an order to create
          side = self.sideFunc()
          if side <> None:
            volume = int(self.volumeFunc())
            if volume > 0:
              # create order given side and parameters
              order = self.orderFactory(side)(volume)
              # send order to the order book
              self._trader.send(order)
```

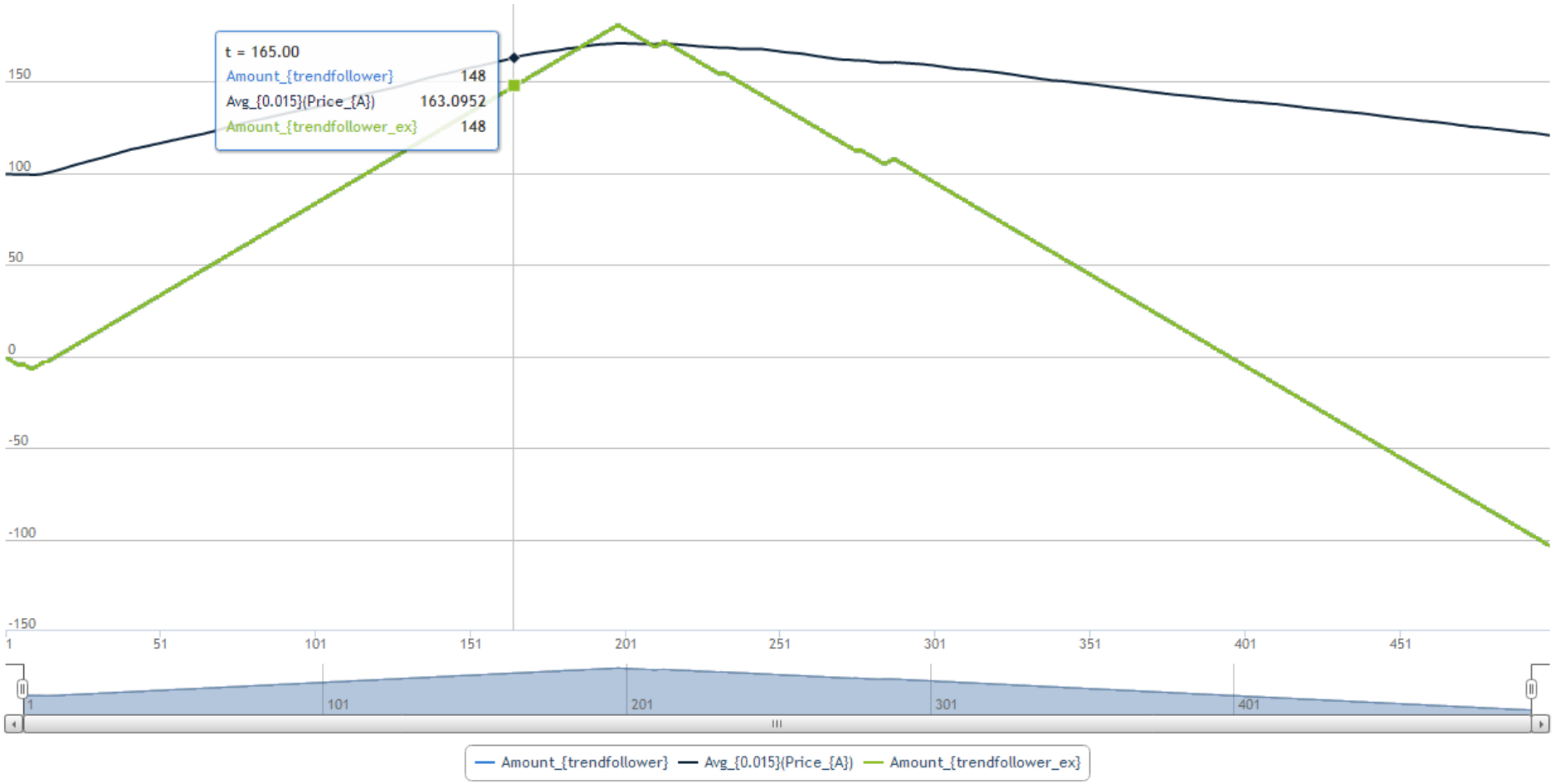
Signal strategy

Signal strategy listens to some discrete signal and when the signal becomes more than some threshold it starts to buy. When the signal gets lower than -threshold the strategy starts to sell.



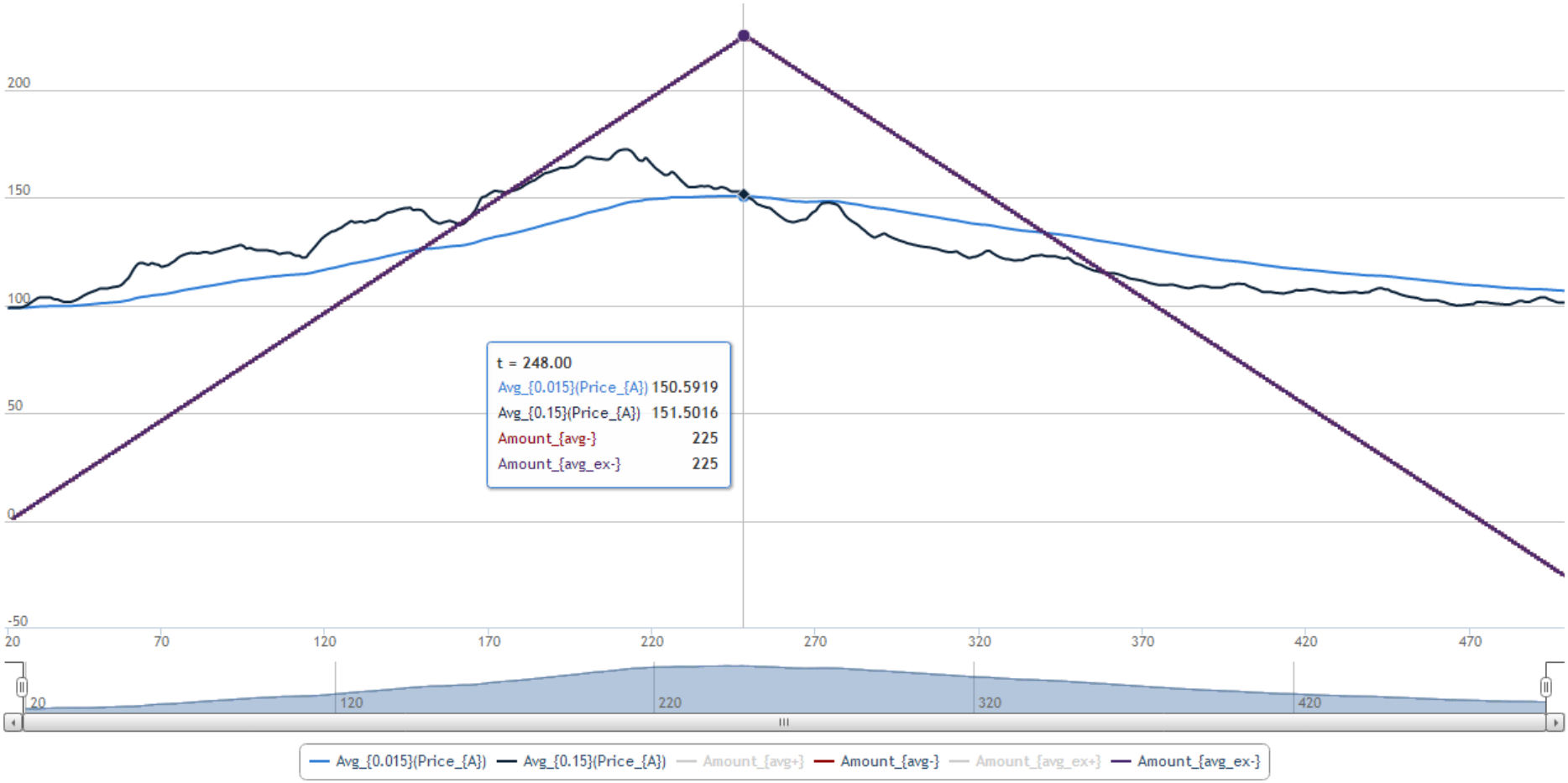
Trend follower strategy

Trend follower is an instance of a signal strategy with signal equal to the first derivative of a moving average of the asset's price (i.e trend).



Two averages strategy

Two averages is an instance of a signal strategy with signal equal to the difference between two moving averages of the asset's price (i.e trend).

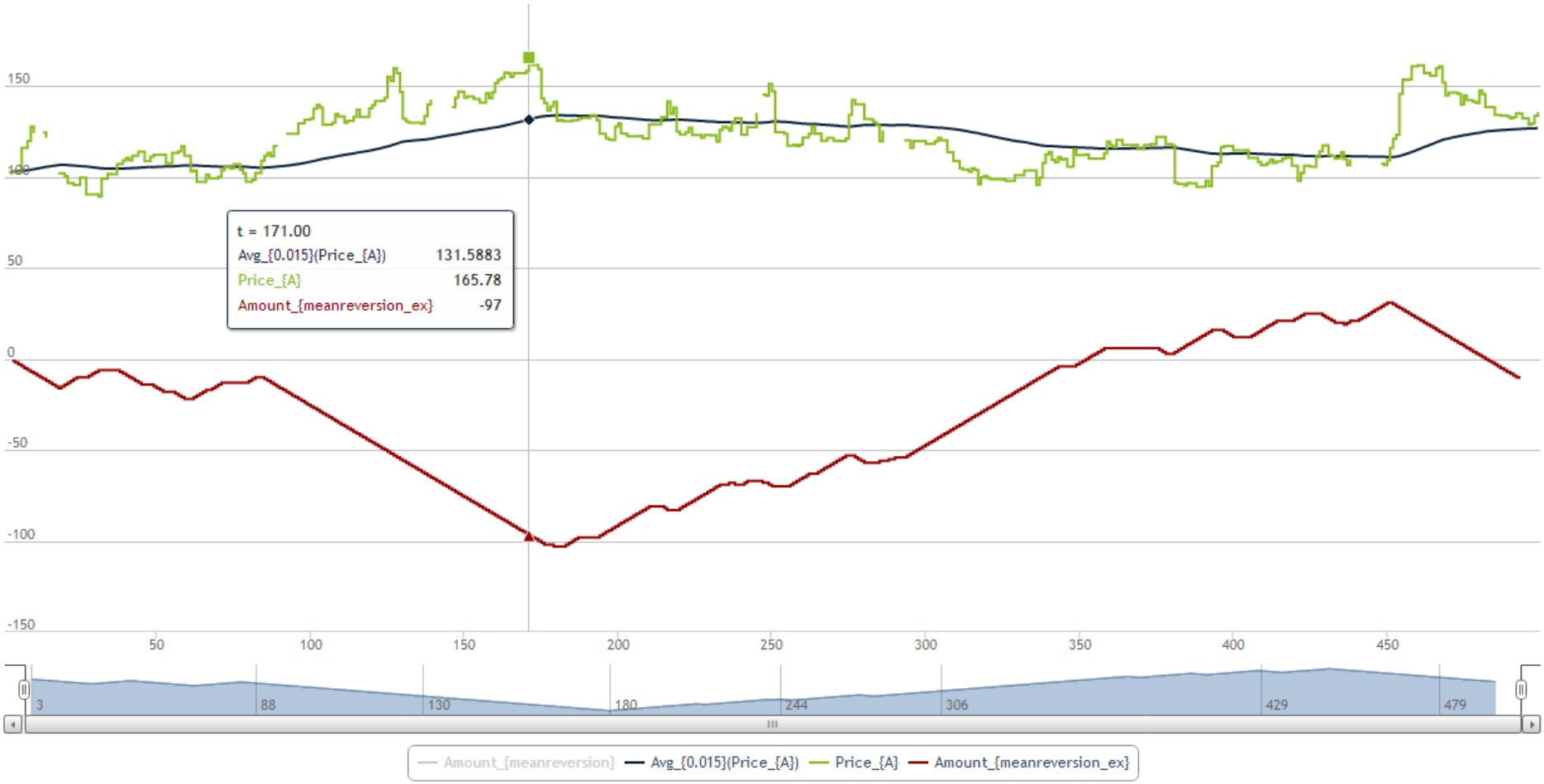


Fundamental value strategy

Fundamental value strategy is an instance of a signal strategy with signal equal to the difference between the asset's price and some fundamental value.

Mean reversion strategy

Mean reversion strategy is an instance of a fundamental value strategy with fundamental value equal to some moving average of the asset's price.



Dependency strategy

Dependency strategy is an instance of a fundamental value strategy with fundamental value equal to another asset's price multiplied by given factor.

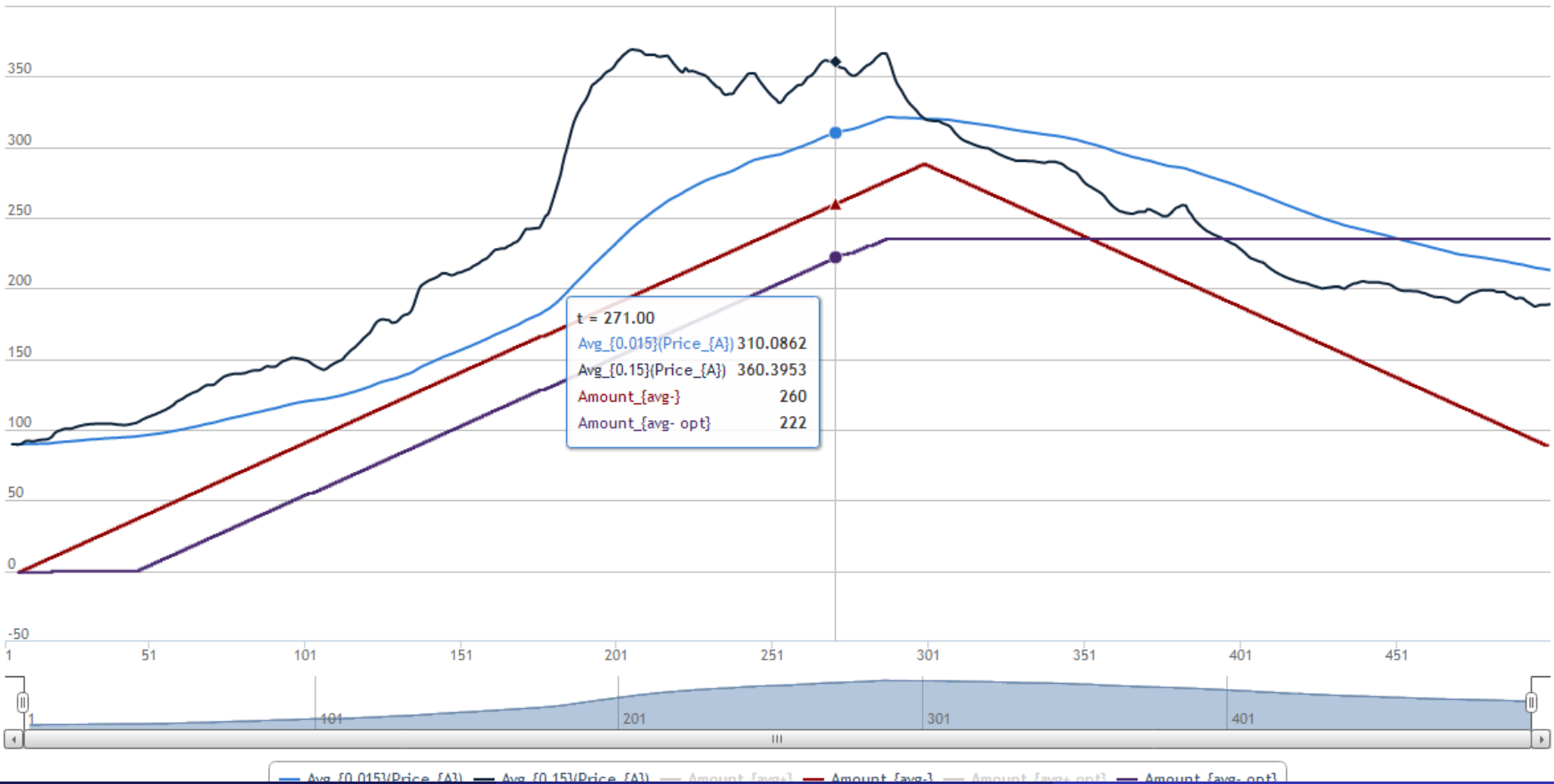
Liquidity provider

Liquidity provider sends limit-like orders with a price equal to the current asset's price multiplied by some randomly chosen factor

```
@registry.expose(["Generic", 'LiquidityProviderSide'], args = ())
def LiquidityProviderSideEx(side                    = Side.Sell,
                            orderFactory            = order.LimitFactory,
                            defaultValue            = 100.,
                            creationIntervalDistr   = mathutils.rnd.expovariate(1.),
                            priceDistr              = mathutils.rnd.lognormvariate(0., .1),
                            volumeDistr             = mathutils.rnd.expovariate(1.)):

    orderBook = orderbook.OfTrader()
    r = Generic(eventGen    = scheduler.Timer(creationIntervalDistr),
                volumeFunc  = volumeDistr,
                sideFunc    = ConstantSide(side),
                orderFactory= order.AdaptLimit(orderFactory,
                                               mathutils.product(
                                                  SafeSidePrice(orderBook, side, defaultValue),
                                                  priceDistr)))

    return r
```

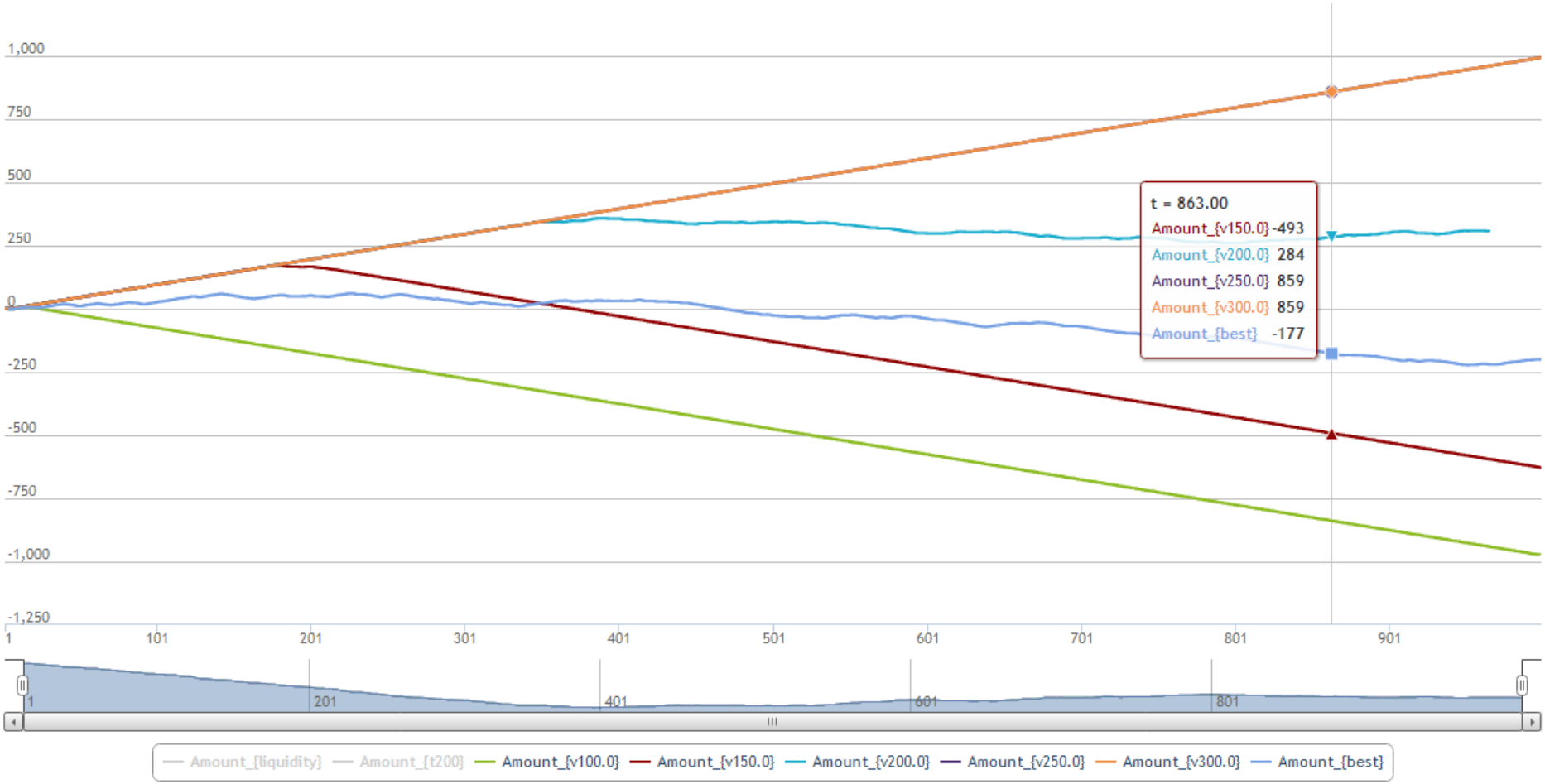
Trade-if-profitable strategy

Suspends or resumes an underlying strategy basing on its performance backtesting. By default, first derivative of a moving average of 'cleared' trader's balance (trader's balance if its position was cleared) is used to evaluate the efficiency.



Choose-the-best strategy

Backtests aggregated strategies and allows to run only to that one who has the best performance. By default, first derivative of a moving average of 'cleared' trader's balance is used to evaluate the efficiency.



Observable

Traders and order books provide basic accessors to their current state but don't collect any statistics. In order to do it in an interoperable way a notion of observable value was introduced: it allows to read its current value and notifies listeners about its change.

- Primitive observables on
  - traders: position, balance, market value of the portfolio, 'cleared' balance etc.
  - order books: ask/mid/bid price, last trade price, price at volume, volume of orders with price better than given one etc.
- `OnEveryDt(dt, dataSource)` evaluates `dataSource` every `dt` moments of time. Often used with `Fold(observable, accumulator)` where `accumulator` may be a moving average or another statistics collector.

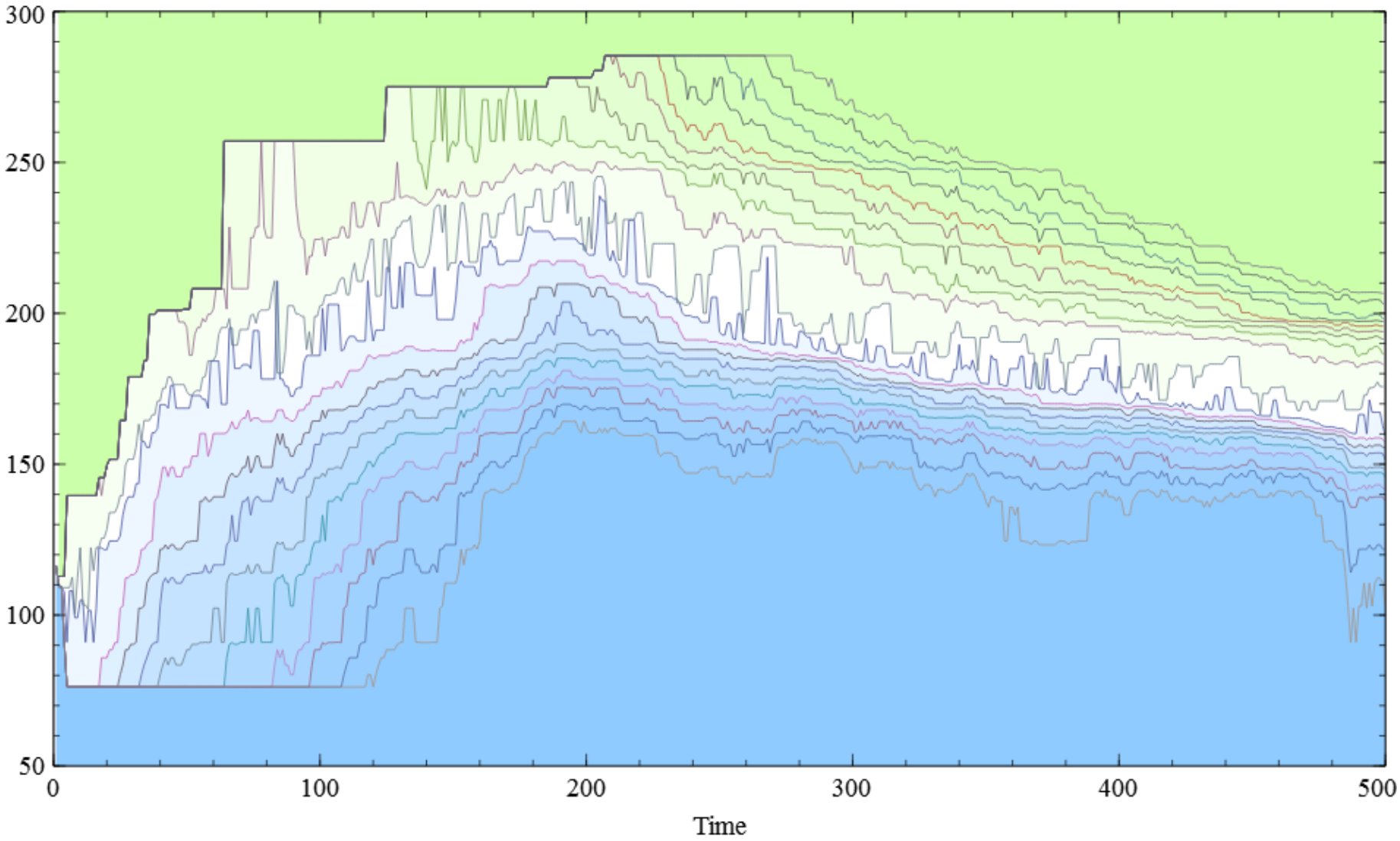
History of an observable can be stored in a `TimeSerie` and rendered later on a graph.

Using Veusz

When developing a new strategy it is reasonable to test it using scripts and visualize results by Veusz

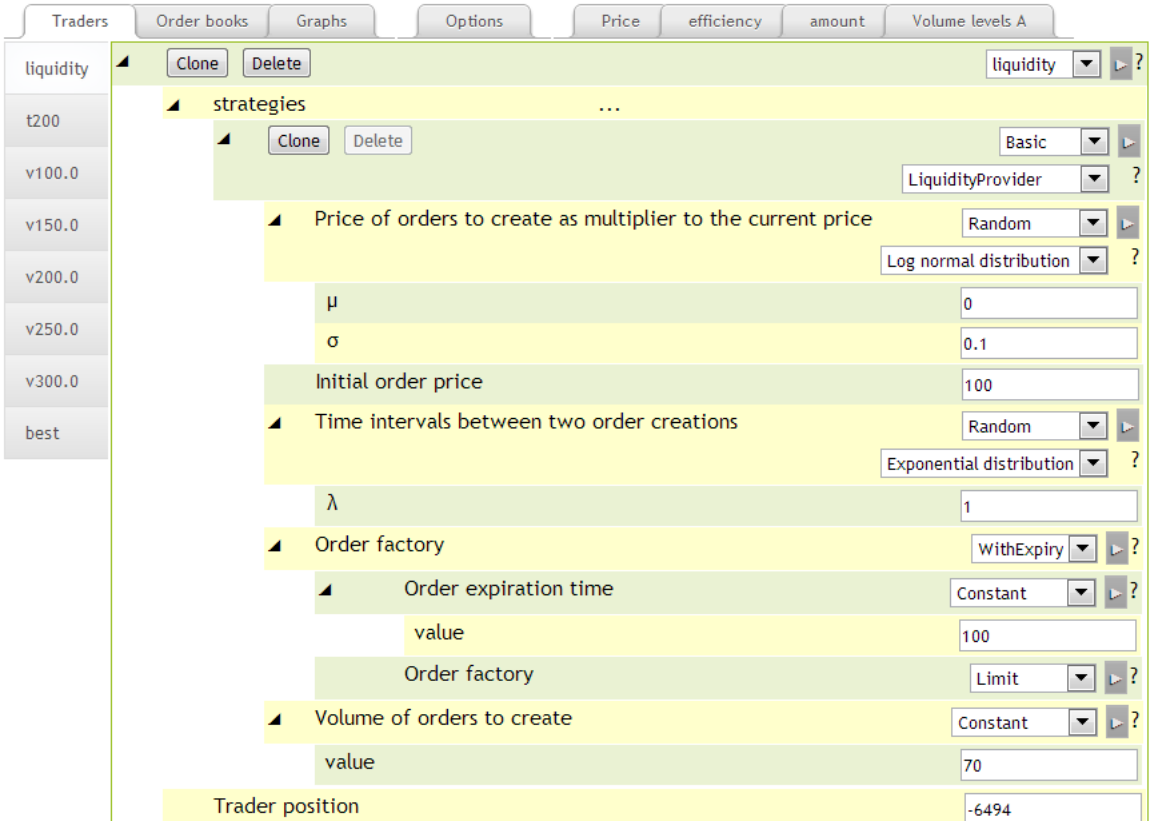
```
from marketsim import (signal, strategy, observable, mathutils)
from common import run

def Signal(ctx):

    const = mathutils.constant
    linear_signal = signal.RandomWalk(initialValue=20,
                                      deltaDistr=const(-.1),
                                      label="20-0.1t")

    return [
        ctx.makeTrader_A(strategy.LiquidityProvider(volumeDistr=const(4)), "liquidity"),

        ctx.makeTrader_A(strategy.Signal(linear_signal), "signal",
                         [(linear_signal, ctx.amount_graph)]),

        ctx.makeTrader_A(strategy.SignalEx(linear_signal), "signal_ex")
    ]

if __name__ == '__main__':
    run("signal_trader", Signal)
```

Rendering graphs by Veusz



Web interface

Web interface allows to compose a market to simulate from existing objects and set up their parameters



Basic,LiquidityProvider

Liquidity provider is a combination of two LiquidityProviderSide traders with the same parameters but different trading sides.

It has followng parameters:

Order factory

order factory function (default: order.Limit.T)

Initial value

initial price which is taken if orderBook is empty (default: 100)

Time intervals between two order creations

defines intervals of time between order creation (default: exponential distribution with $\lambda = 1$)

Price of orders to create as multiplier to the current price

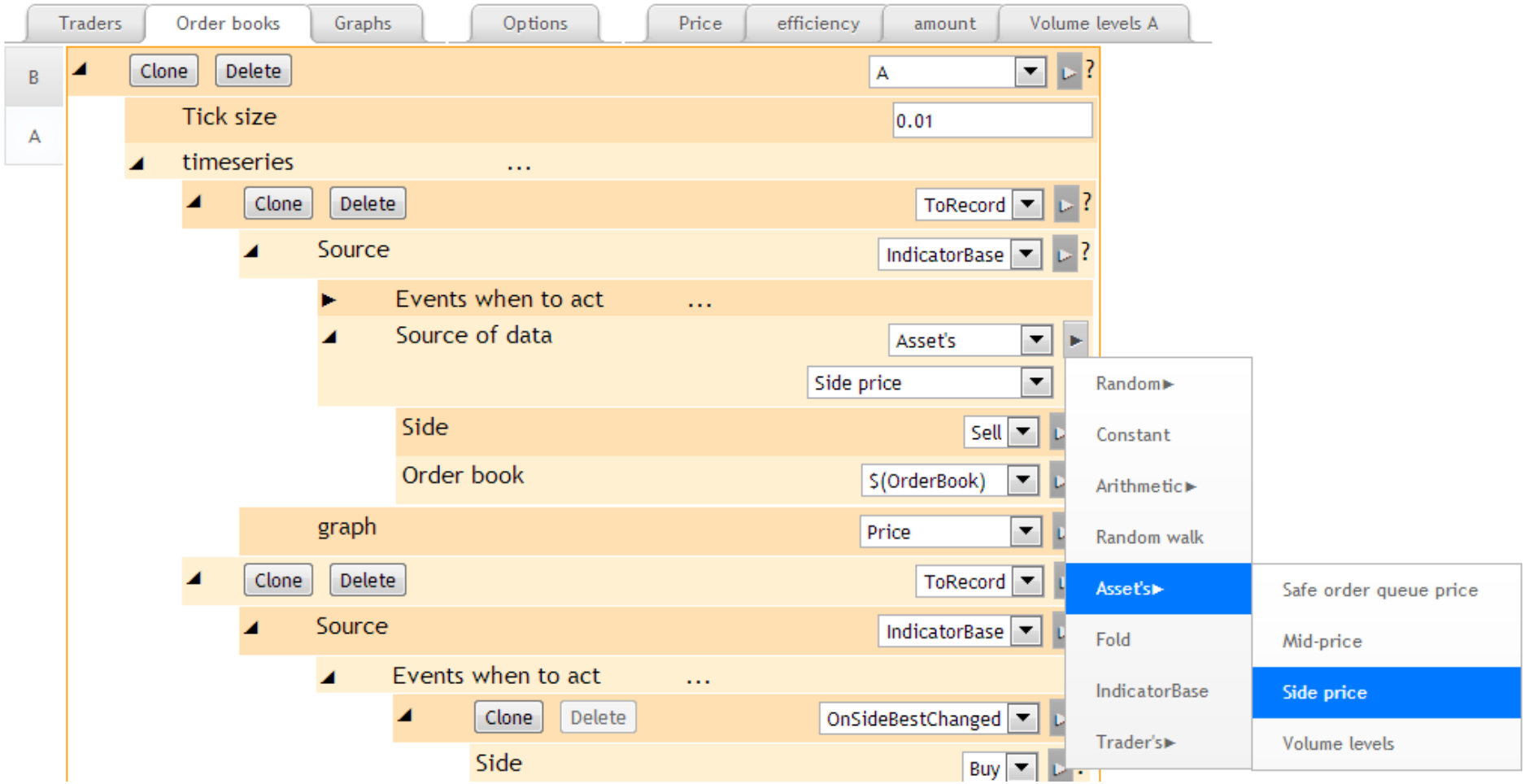
defines multipliers for current asset price when price of order to create is calculated (default: log normal distribution with $\mu = 0$ and $\sigma = 0.1$)

Volume of orders to create

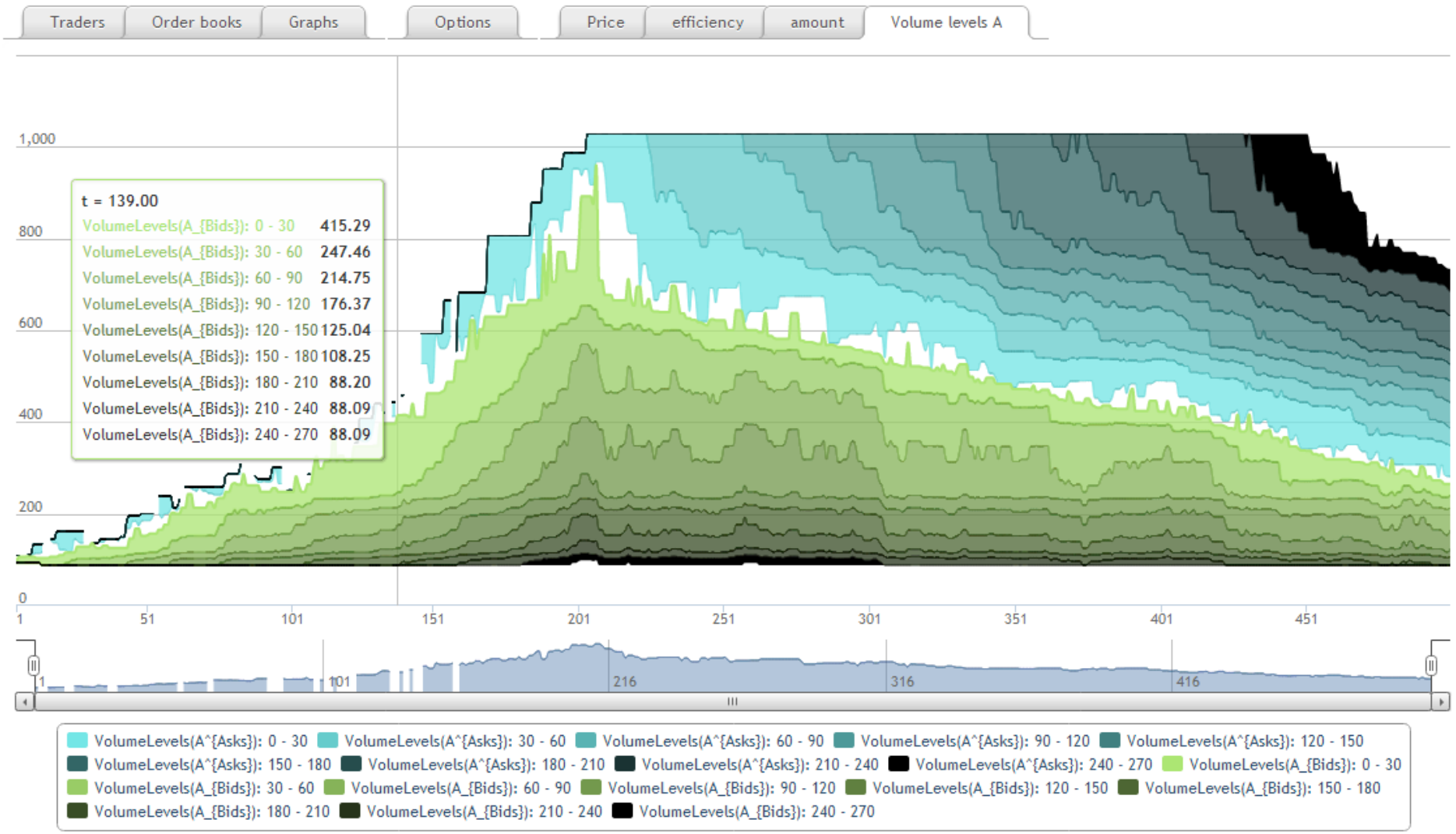
defines volumes of orders to create (default: exponential distribution with $\lambda = 1$)

Time series

Timeseries field of a trader or an order book instructs what data should be collected and rendered on graphs

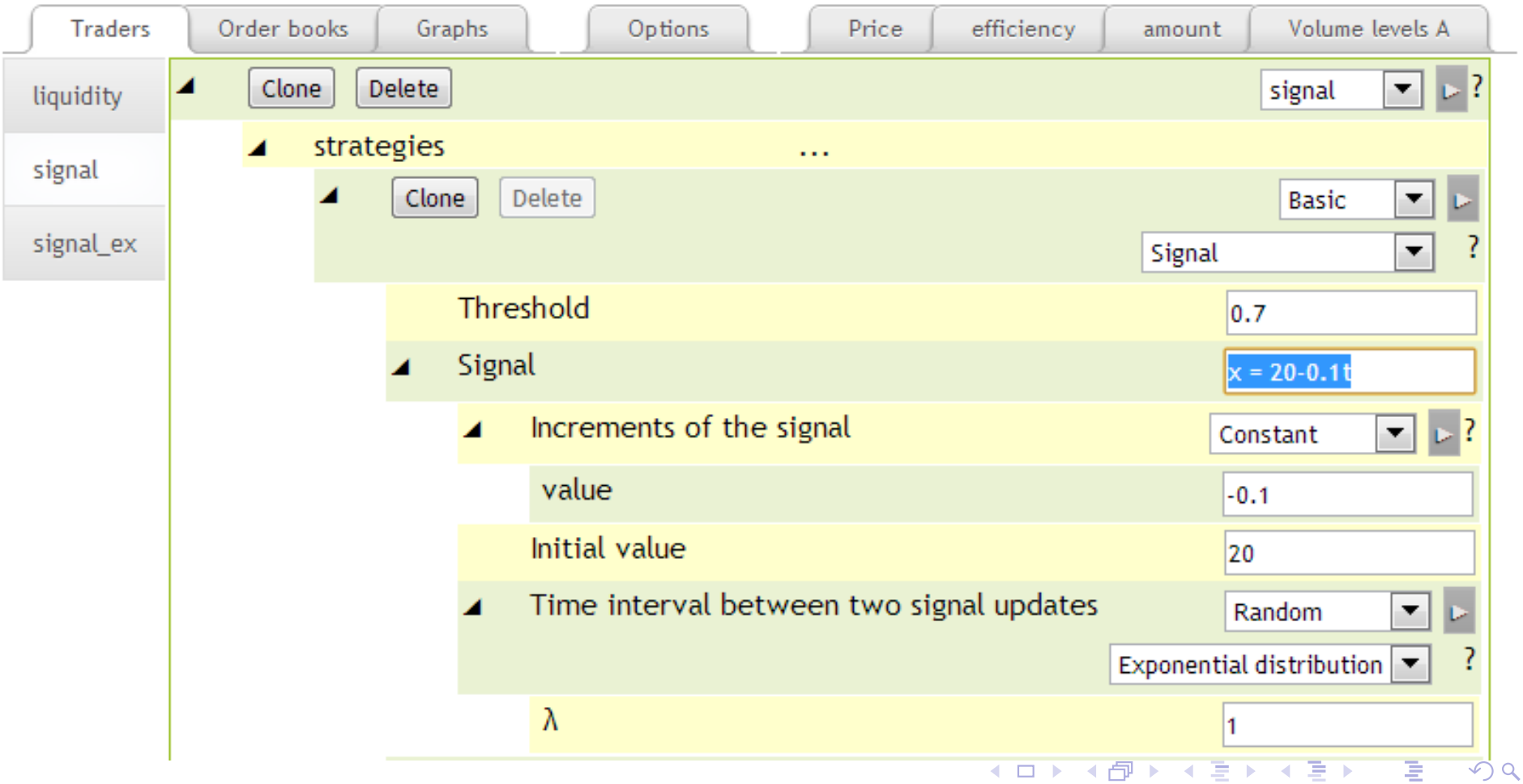


Rendering results



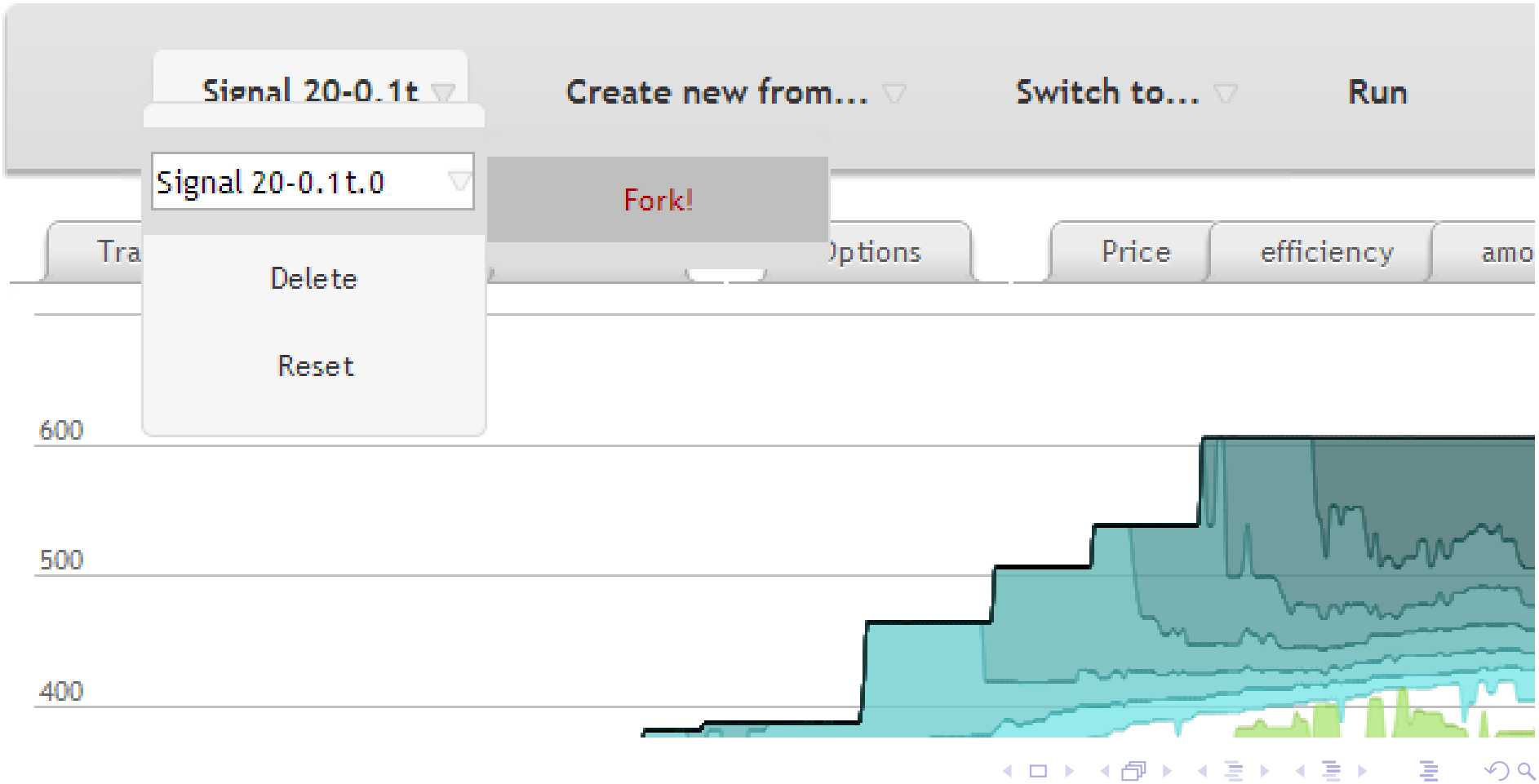
Node aliases

Object tree nodes can be assigned aliases that can be used later to refer to the sub-tree (explicit by-value or by-reference cloning semantics is to be implemented)



Workspaces

Every user (identified by browser cookies) may switch between multiple workspaces. Workspaces can be forked, removed or created from a set of predefined ones.



Exposing Python classes to Web-interface

- displayable label for the class ('Random Walk')
- docstring in rst format
- property names and static constraints on types of their values

```
@registry.expose(['Random walk'])
class RandomWalk(types.IObservable):
    """ A discrete signal with user-defined increments.

        Parameters:

        **initialValue**
            initial value of the signal (default: 0)

        **deltaDistr**
            increment function (default: normal distribution with |mu| = 0, |sigma| = 1)

        **intervalDistr**
            defines intervals between signal updates
            (default: exponential distribution with |lambda| = 1)
    """
    _properties = { 'initialValue' : float,
                    'deltaDistr'   : meta.function((), float),
                    'intervalDistr': meta.function((), float) }
```

Type system

- Primitive types: int, float, string
- Numeric constraints: `less_than(2*math.pi, non_negative)`
- User-defined classes. If a property constraint is type B then any object of type D can be used as its value provided that D derives from B.
- Array types: `meta.listOf(types.IStrategy)`
- Functional types: `meta.function((Side, Price, Volume), IOrder)`

Possible improvements:

- `meta.function((a1, ..., aN), rettype)` could be used where `meta.function((a1, ..., aN, b1, ..., bM), rettype)` is expected
- `meta.function((..., B, ...), rettype)` could be used where `meta.function((..., D, ...), rettype)` is expected if D casts to B
- `meta.function(args, D)` could be used where `meta.function(args, B)` is expected if D casts to B

Future developments

C++ version:

1. Implement core functionality (scheduler, order books, basic orders and traders) in C++ (already done) and provide extension points to allow to a user create strategies and meta orders in Python (or use existing ones)
2. Given object tree describing a simulation model, generate on the fly C++ code as instantiations of template classes corresponding to classes in Python version

C++ version

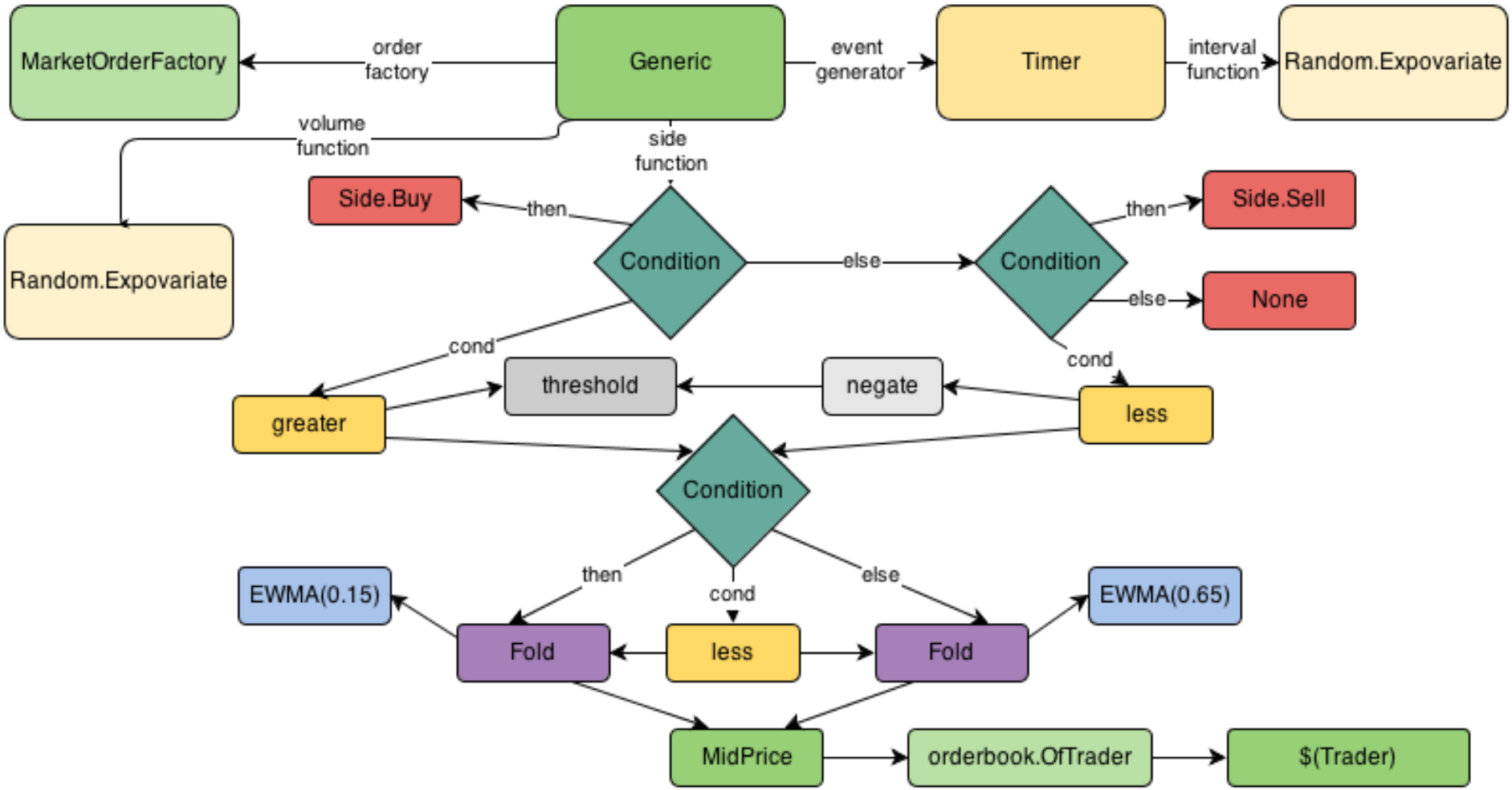
Flexible as Python version and has performance comparable to a C hand-written version. The main problem: simulation configuring is not intuitive, so let's do the configuration automatically by a code generator.

```
template <class Base>
    struct GenericStrategy : Base
    {   using Base::self; // 'this' casted to the most derived class

        GenericStrategy() {
            self().eventGen().subscribe(boost::bind(&GenericStrategy::wakeUp, this));
        }

        void wakeUp() {
            if (boost::optional<Side> side = self().sideFunc()) {
                volume_t volume = self().volumeFunc();
                if (volume > 0) {
                    auto order = self().orderFactory()(*side, volume);
                    self().trader().send(order);
                }
            }
        }
    };
```

Python version/Web interface

- Automatic dependency tracking in Python code (observables/computed observables from KnockoutJs)
- Notion of variables in the Web interface to label common object graph subtrees
- Model graph representation in Web interface (???)

Model graph representation in Web interface



Simulation components (by Karol Podkanski)

- Strategies
  - **Relative strength index (RSI).** Buy/sell when stock is oversold/overbought according to the index
  - **Stop-loss strategy.** Applied to any strategy in order to limit losses if they reach a certain threshold
  - **Multi-armed bandit.** Evaluate an array of strategies and assign them scores based on their efficiency. A strategy is then chosen randomly, with a distribution based on the scores.
  - Other meta-strategies (???)
  - **Pairs trading.** A dependence between two assets is assumed (for example, correlation). A trade is initiated when a function of the two assets (for example: weighted average) deviates from it's mean value.
- Volume management
- Enter/exit time management (currently random or constant)

Simulation components (by Karol Podkanski)

- Observables (Indicators):
  - Volatility
  - Volume
  - Performance
  - Relative strength index
  - Technical analysis
    - Trendline: support and resistance
    - New High/Low
    - Channels
    - Double Top/Bottom
    - Head and Shoulders
- Add position constraints to traders:
  - traders have to allocate their limited resources
  - certain assets cannot be shorted

Thank you!