# *Comparison of Object-Oriented Programming and Data-Oriented Design for Implementing Trading Strategies Backtester*

**Timur Mironov**
*Institute of Engineering Sciences*
*Pskov State University*
Pskov, Russia
jarwis9812@gmail.com

**Lilia Motaylenko**
*Institute of Engineering Sciences*
*Pskov State University*
Pskov, Russia
lvs_@mail.ru

**Dmitry Andreev**
*Institute of Engineering Sciences*
*Pskov State University*
Pskov, Russia
dandreev60@mail.ru

**Igor Antonov**
*Institute of Engineering Sciences*
*Pskov State University*
Pskov, Russia
igorant63@yandex.ru

**Mikhail Aristov**
*Institute of Engineering Sciences*
*Pskov State University*
Pskov, Russia
maristov@list.ru

***Abstract* - This research proposes a way to accelerate backtesting of trading strategies using data-oriented design (DOD). The research discusses the differences between DOD and object-oriented approach (OOP), which is the most popular at the current moment. Then, the paper proposes efficient way to parallelize a backtesting using DOD. Finally, this research provides a performance comparison between DOD and OOP backtester implementations on the example of typical technical indicators. The comparison shows that use of DOD can speed up the process of quantitative features calculation up to 33% and allows for parallelization scheme that better utilizes resources in multiprocessor systems.**

***Keywords* - *algorithmic trading, data-oriented design (DOD), high performance computing (HPC), parallel computing***

## I. INTRODUCTION

Term backtest refers to testing of a trading strategy on historical data in order to assess its effectiveness or optimize parameters [1]. In essence, backtest is a simulation, in which the algorithm being tested is placed in conditions as close as possible to the real exchange trading that took place in the past. Even though the results of such test are hypothetical and in no case can be unambiguously considered a reliable indicator of the strategy success on the real market, they allow, at least, drawing conclusions about its adequacy and obtaining an estimation of many of its properties, such as the ratio of purchases to sales, the number of transactions per unit of time, etc. Although various studies indicate a high probability of overfitting when a strategy's model is constructed and optimized based on the results of a backtest [2], the testing of trading algorithms on historical data is still the main tool for trading strategy development and can often be seen both in practice and in academic research.

Regardless of the goals and methods of using the backtest results, the procedure itself requires a lot of computational power. Due to the development of trading platforms and the growing popularity of algorithmic trading, the amount of data that needs to be analysed and processed is constantly growing [3]. Modern trading robots are capable of performing hundreds of transactions per second. To remain competitive a trading system must be able to respond to even the smallest events. Many exchanges, in turn, provide an opportunity to receive sufficiently detailed data in the form of real-time anonymous order flow. Just for single trading instrument the number of orders, passing through the trading system per day, can run into millions. For reliable strategy testing all this amount of data must also flow through the backtesting system. Of course, one strategy can work with a large number of instruments at once, and the tested period can be as long as several years, which only increases the number of calculations.

This work is devoted to the computational side of backtesting. To increase the speed of computations it is proposed to use data-oriented design (DOD) [4]. Currently,

*Online ISSN 2256-070X*
*https://doi.org/10.17770/etr2021vol2.6629*

DOD is mainly used in the video game development and serves as a tool for optimizing resource-intensive computational tasks; it also offers techniques for parallelizing data processing efficiently. The goal of the research is to compare DOD with the most popular at the moment OOP approach and evaluate performance of backtesting software developed using both approaches.

## II. Materials and methods

To understand the context of the study, it will be useful to define some abstract architecture that any software for testing trading strategies on historical data will sufficiently implement. First of all, let's consider the input data. Most of modern markets are double auction markets, meaning that buyers and sellers simultaneously offer their prices by submitting bids to the exchange's trading system. Those bids form an order book (table of quotes), which reflects the current supply and demand at various prices and is available for all bidders. If the system receives an order, which satisfies certain conditions (usually, the condition is that the order price is not worse than the price of one or several opposite orders), an execution (deal or trade) occurs at the best price for both parties. Thus, the main sources of the data, from which a strategy generates a trading signal, are order book and a table (list) of deals. As mentioned above, the most detailed type of data is the flow of all orders, entering the exchange. However, this format is not available on certain trading platforms. Instead, an exchange might broadcast two separate data streams: a simplified representation of the order book, consisting of price levels (price and quantity pairs), and trade executions with varying degrees of detail (for example, some platforms broadcast passive participants of transactions only or vice versa).

Usually, data is transmitted in incremental manner - instead of entire current state, each data packet contains only a change of a state since previous packet (or information about particular event in exchange's trading system). Typically, these features are reflected both in the market data storage system (information is stored as incremental updates) and in the architecture of the backtest software, which is implemented as an event-driven system. Market trading, as a rule, isn't going on continuously. Instead, the trading day is separated into several sessions with short breaks between them. Similarly, backtesting is often applied to one or more sessions. One day is often chosen as the minimum time unit for backtesting.

Thus, the primary function of the backtester is to read the raw data of a trading session from some source (file, database, TCP connection, etc.) and restore the order book and deals table - i.e., restore a structured data.

The next function of the backtester is strategies calculation. In this paper it is understood as the computation of a set of simple predicates that compare certain computable features, with some threshold values. The fixed thresholds, as well as the weights and coefficients needed to extract the features, are called strategy parameters. From this point of view, a strategy is, in fact, a decision tree, the output of which is a trading signal that tells to buy or sell certain volume of a given trading instrument.

The next function of the backtester is to extract some quantitative features from structured market data. By analogy with indicators in technical analysis, in this study, a module that calculates a certain feature, is called an indicator. Thus, an indicator is a module that makes any calculations based on market data of a specific financial instrument. The main structural function of such modules is to minimize the computation graph by reusing already calculated features. As a rule, the indicator algorithm depends not only on the market data, but also on additional parameters, i.e., it implements the calculation of a whole class of features instead of just one. Despite the apparent simplicity, it is the effectiveness of the organization of indicators that determines the performance of the entire system. Since different strategies within a single backtest session can use the same indicators, the backtester must provide a mechanism for reusing the results of calculations. This is important, because in the problem of parameter optimization, in the vast majority of cases, heuristic search techniques are used, such as a particle swarm [5, 6], genetic algorithms [7, 8], and machine learning methods [9] - [12]. The use of such techniques implies computation of a large number of combinations of parameters, many of which would partially repeat each other, which would inevitably lead to repetitive calculations.

The last and the main function of the backtester is the simulation of executions, and the calculation of various trading metrics, starting with simple numerical characteristics of the strategy, such as the number of orders sent, the maximum position size (the number of units of a trading instrument in the portfolio) or the ratio of purchases to sales, and ending with financial metrics for assessing efficiency of investment portfolio (e.g., Sharpe ratio).

In addition, the backtester must provide an adequate simulation of the conditions in which the algorithm being tested is supposed to operate. Thus, it is necessary to control the number of transactions and comply with the restrictions on lot sizes or the minimum price step set on the specific trading platform. The number of different rules that must be checked during simulation can vary greatly depending on the exchange, regulatory rules, or the needs of a trader.

So, to summarize: strategy testing software performs the following operations:

a) Reading data from a certain source and restoring structured data necessary for calculating trading signals;

b) Extracting quantitative features from structured data;

c) Generation of trading signals;

d) Simulation of executions and computation of effectiveness metrics of the trading algorithms.

This study is mainly devoted to extraction of features and generation of trading signals. Since simulation of executions may greatly vary depending on many factors, the process of trading simulation is not considered in this paper.

The classic implementation of the backtester performs all the processing steps sequentially for each individual event. In the most common OOP paradigm different types of indicators are represented by classes. Each instance of such class encapsulates the thresholds, coefficients and intermediate values required for calculations. Strategies are implemented in a similar way. Together the instances of these classes represent the computing core of the trading system, for which an example of a UML class diagram is shown in “Fig. 1”. The indicator instances implement the calculate method in which, based on the incoming data packet and the internal state, some features are computed. Instances of strategies, in turn, generate trading signals, based on values calculated by indicators. Later the signals can go through several more stages of processing, but this is not essential in this study.

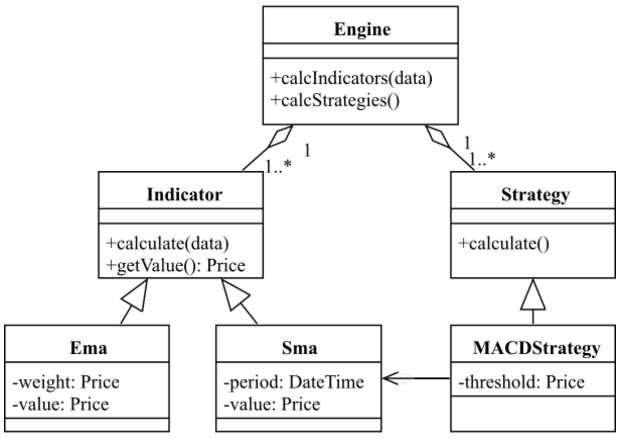

Fig. 1. UML class diagram of backtester’s core.

It can be seen from the diagram that the main cycle of the system looks as follows: iterate over all indicator instances and call their calculate method, and then repeat the same for all strategy instances. Data-Oriented Design offers an alternative approach to computing. One of the main principles of DOD is: "structure of arrays versus array of structures" (SoA versus AoS). Its essence lies in the reorganization of data in such a way that instead of sequential placement of records corresponding to a certain entity, individual components inherent in this entity are sequentially located in memory. For example, if the task is to process an array of N pixels in RGB format, the traditional representation as a vector of N tuples of the form (R, G, B) is replaced with a 3xN matrix, where each row corresponds to a colour component, and the column represents the entire pixel. This layout can significantly improve performance due to greater data locality. Of course, the speed gain depends on the pattern of accessing the fields of such records. The greatest increase of performance is achieved when the computations is done on the individual rows of the matrix. It should be noted that hereinafter it is assumed that matrices are implemented as a set of separate one-dimensional arrays that makes it easy to add or remove rows.

In the backtester, the principle of "structure of arrays versus array of structures" finds several uses. The first relates to the backtester's computational core. In this paper, the following organization of computing units is proposed. To calculate K indicators of the same type, which require C coefficients for calculations, a matrix of parameters IP: CxK is allocated, where a row represents separate indicator coefficient. The module that performs calculations receives IP and market data as an input. The calculation results are placed in the output matrix I: LxT, where L is the number of output values of the indicator, T is the total number of all indicators involved in the current backtesting. Thus, in proposed architecture each OOP instance of the indicator class is represented by a column in the parameters matrix, and instead of calculating each feature separately all indicators of the same type are processed at once. At a minimum, this allows reducing the cost of a function call (which can be very significant in the case of using polymorphism and virtual functions), at best, this approach increases the locality of the instruction and data cache.

In a similar way, the instances of strategy classes are replaced by the matrix of parameters PS. The input data for the strategy calculation module are matrices I, PS and PM. The essence of the PM matrix is as follows: the number of strategies can be greater than the number of indicators, since the calculated values of the latter can be used in the signals generation by several different strategies. Therefore, it is necessary to provide a way to map the index of the strategy to the corresponding value in the matrix I, what is accomplished with PM.

Such organization is, in fact, an implementation of the ECS (entity-component-system) pattern, in which each object (called an entity, according to the pattern) is represented by a set of components that only store data or state. All logic of a program is executed by so-called systems that perform calculations with specific components [13, 14]. In terms of this pattern, indicators and strategies are systems, and their parameters are components. In this case, the trading system itself can be considered an ECS-entity. “Fig. 2” shows the OOP architecture after applying proposed transformations.

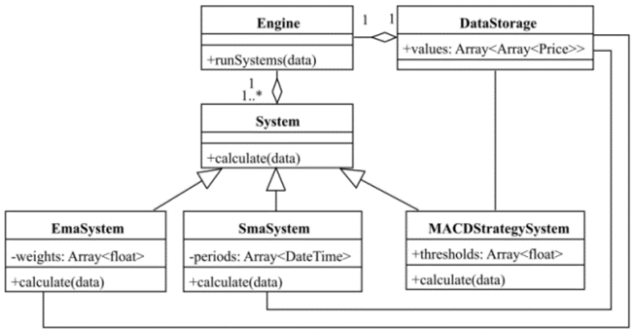

Fig. 2. UML diagram of DOD backtester’s core.

The principle of "structure of arrays versus array of structures" is well suited not only for storing intermediate calculations during backtesting, but also for representing input data. One of the main types of input data for a strategy testing system is a table of deals. Minimum set of fields that are required to represent a deal record are the following:

a. Instrument id - trading instrument identifier;

b. Timestamp - time of the transaction;

c. Price - the price of the deal;

d. Quantity - the volume of the deal;

e. Flags - a set of bits used to encode the direction of the deal (buy or sell) and utility information.

In addition to the specified fields, for example, the identifier of the exchange or market may be added (in case if a backtest involves instruments on different exchanges). Also, some trading platforms may broadcast the identifiers of orders involved in the transaction. In general, the set of fields can vary in different implementations depending on the needs of the users, but one way or another, the fields specified above should be present.

It is important to understand that, as a rule, not every field is required for calculating indicators (basically, the price, volume and direction of the transaction are needed). When reading transaction records using the classical approach of the "array of structures" a significant part of the processor's cache lines is wasted on utility fields that are not involved in the calculations. Organizing records as a SoA solves this problem. In addition, such a layout makes it possible to use SIMD instructions more efficiently due to a more optimal "vertical" arrangement of data in memory [15].

One of the advantages of a more vertical data arrangement as a "structure of arrays" is the ease of parallelization. As it was said, data processing in the strategy testing system can be logically divided into 4 stages. These stages, in the case of an OOP system, are sequentially executed for each packet of incoming data. With such approach, parallelism is possible only at the level of trading sessions and only if the strategies being tested are designed for short-term trading and remain position-neutral most of the time, i.e., always have empty portfolio at the end a trading session.

To ensure efficient parallel execution, this paper proposes to combine the stage of generating trading signals and trading simulation in one stage. Instead of sequential processing of data packets, it is proposed to move to sequential processing of stages. Thus, the first stage (responsible for reading data) prepares not just single piece of data for the subsequent calculation of features, but processes all incoming data at once. The results are saved in SoA format, namely in the form of a matrix D: PxN, where P is the number of values calculated at this stage, N is the number of data packets. The amount of calculations required is determined by a set of indicators. However, there is a mandatory row which must always be present in D - an array of time stamps of data packet arrival. Without this array time averaging, which is often found in financial calculations, is impossible. Time also plays an important role in the trading simulator module for emulating delays in sending and receiving orders and other messages. This stage differs little from the OOP approach, because I/O operations are performed most of the time and use parallelism is limited by storage devices.

The next stage is the calculation of indicators. It takes D and a matrix of parameters as an input and performs computation of the required features. A new matrix I is formed as the result of this stage. It should be noted that strategies do not always generate a signal upon the arrival of each individual packet. It is a fairly common practice in trading systems to split a continuous data stream into intervals within which the features are calculated. Thus, the number of columns of I is limited by the number of packets, and the number of rows can be infinitely large, since the indicator can calculate more than one feature. Such organization of computations ensures the independence of indicators from each other, which allows them to be processed in parallel. At this stage, almost any number of processors can be involved, of course, within the limits of the number of indicators.

The third stage is the calculation of strategies and trading simulation. The strategies receive parameters matrix and matrix I, on the basis of which trading signals are calculated. Ultimately, the signals become trade orders for placing or cancelling bids with certain parameters (price, volume, etc.). Trade orders together with the data D prepared at the first stage are sent to the trading simulation module, where the placing of orders on the exchange and their execution are simulated. Information about executed orders is fed back to the strategy to update information about the current position. Also, based on the executions, various financial metrics of the strategies are calculated, which are the final result of the entire system. Similarly to indicators, each individual strategy is completely independent and its financial metrics calculation and trading simulation can be done in parallel.

For efficient use of resources a work-stealing task scheduler is used in the system [16]. Also, it is necessary to take into account the priority. It means that strategy processing and trading simulation is of the highest priority, since the user is interested in getting the results as early as possible. The read data step only needs to be performed if no other tasks are available. In addition, dynamic load balancing is implemented. Since the number of threads, as a rule, is less than the number of indicators and strategies, it is necessary to group their calculations. In this way, the grouping that provides the greatest locality of data and instruction cache is more preferable, i.e., grouping in which the number of ECS-systems processed by one processor is minimal. Since the backtesting procedure assumes repeated execution of the same algorithms, it is proposed to store information about the time spent on a certain type of computation at each stage with subsequent averaging over all observations. The resulting value can be used as an estimate of the execution time. Based on the estimate, ECS-systems are distributed among threads, so that all of them spend approximately the same amount of time.

## III. Results and Discussion

In order to demonstrate the effect of the proposed transformation, for this study a backtesting system that implements OOP and DOD approaches has been developed. Several performance tests were conducted. The first test was to calculate a simple technical indicator SMA on simulated dataset consisting of 1 million records. In different launches of the program, a different number of indicators are calculated, with each launch repeated 10 times. The average time of all launches is considered to be the result. Subsequent performance tests in this work are carried out in a similar fashion. The results of the program runs are shown in "Fig. 3". The vertical axis shows the time in milliseconds, i.e., the smaller the result, the better. The horizontal axis shows the number of calculated indicators. The program in this and subsequent tests is built with g++ compiler version 9.3 with O3 optimization level and is executed on a multiprocessor server system with 120 Intel Xeon CPU E7-8880 v2 @ 2.50GHz processors under the Gentoo Linux operating system.

As it can be seen from the graph, in the case of a small amount of calculations, the architecture proposed by DOD shows the result no worse than OOP, but significantly exceeds the performance of object-oriented approach with an increase in the number of indicators. So for 8 indicators the speed increase is 23%, and for 64 the acceleration reaches 36%.

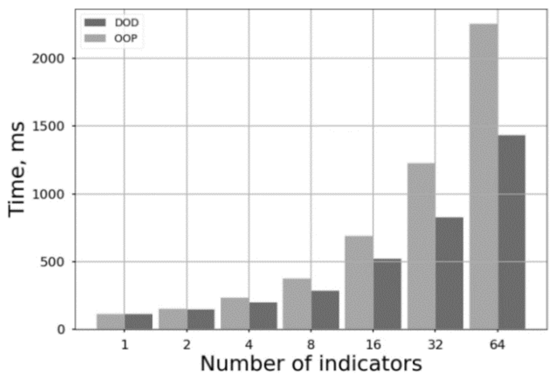

Fig. 3. Plot of execution time versus number of indicators.

The effectiveness of the proposed data organization is demonstrated in "Fig. 4". The graph shows the results of calculations of the weighted average price VWAP for an array of 10 million transactions, represented by structures of different sizes. OOP stands for the classic AoS approach, and DOD stands for the SoA approach. The horizontal axis marks the size of the deal data structure.

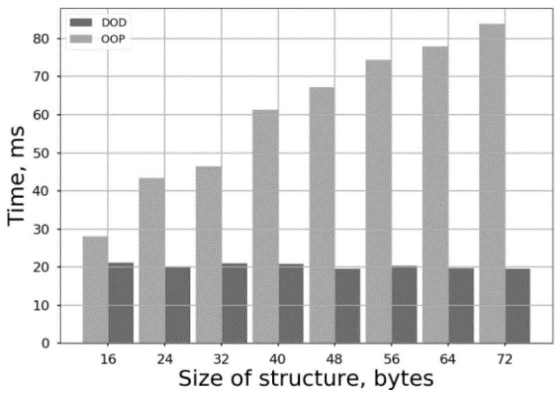

Fig. 4. Plot of performance versus data structure size.

Assuming that 64 bits are allocated for each field of the structure, at least 40 bytes are required for single deal record. As it can be seen from the graph, with the DOD approach the processing speed does not depend on the size of the structure, and the performance gain is observed even for 16 bytes (as if only two fields were allocated to represent the transaction - price and volume). Speedup only grows with and increases in memory size and for 40 bytes the performance gain is 66%.

Also, in this study similar testing of the data structure for order book processing was carried out. The paper [17] is taken as a basis, in which it is proposed to use an array of price levels in the form of pairs (p, q) where p is the price and q is the volume. The order of such array is maintained when inserting and removing elements. According to the principle of "structure of arrays", this scheme is converted into two one-dimensional arrays - separately for all prices and all volumes. Various metrics were measured for the new data structure:

a. The time of processing all incoming data packets within one session;

b. The time during which the following features are repeatedly calculated on the basis of a fixed number of the first price levels: average price; average volume; weighted average price; the price of a level at which the volume is not less than a certain fixed number. The testing was carried out on both simulated and real data for the most active trading instruments on the MOEX exchange. None of the designated tests found a significant difference in execution speed between SoA and AoS.

The performance of proposed parallelization scheme was tested using the classic MACD strategy as an example. In the test system within one trading session 2000 different combinations of MACD parameters are processed, for which it is required to calculate 1000 different moving price averages, in this test SMA was used. "Fig. 5" shows a graph of the performance of the sequential OOP version and the two types of proposed DOD system. In first case DOD system is tested when the calculations are perfectly balanced between the threads (Balanced). In second case one of the threads performs 1.5 times more work than any other (Unbalanced). It is worth noting that OOP, as well as DOD, implements all the optimizations proposed. Because the OOP version cannot be executed in parallel for one trading session, its performance does not change on the chart. DOD version, in turn, expectedly demonstrates a speedup of almost N times for N threads, in the case when computations are perfectly balanced.

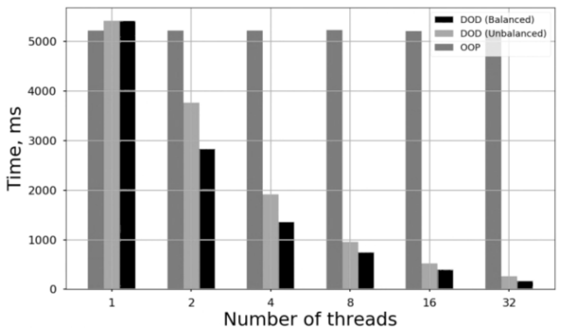

Fig. 5. Plot of performance versus number of threads.

The proposed computation scheme scales better for a large number of processors, compared to sequential computations in the classical OOP approach. Trading instruments also introduce an additional degree of parallelism. In the case when the strategy operates with several securities at once, the calculation of features for a separate security does not depend on the others and can also be performed in a separate thread.

Nevertheless, the proposed method for parallelizing computations has its drawbacks. It is easy to see that the proposed system allocates a large amount of memory. Because the calculated values are stored for each data packet, the upper limit of the total number of data cells is $N_{strategies} * N_{indicators} * N_{packets}$. Considering that the number of packets is running into millions, and the number of strategies and indicators is in thousands, the system consumes a significant amount of RAM even for simple backtest scenarios. Of course, this problem can be solved by artificially reducing the size of the trading session, but such a solution inevitably leads to additional overhead.

Another disadvantage is the bandwidth. Despite the fact that the proposed architecture is highly scalable, under certain conditions its performance does not exceed the one of sequential OOP system. This situation occurs when the number of sessions being tested equals or greater than the number of available threads. Because a sequential system can process individual sessions in parallel, it is capable of performing the same amount of computation at the same time. Considering that sequential execution does not require a large amount of memory for intermediate calculations, the classical system would be much more efficient in terms of resource consumption. However, the backtest results for each individual session would be available much later. Thus, in the proposed and sequential systems, the same backtesting of N sessions will be performed in the same time T. But the time for complete processing of single session in a sequential system will be T, and $T/N$ in parallel one.

## IV. Conclusions

This work shows how DOD optimization techniques can be applied to the automated trading and, in particular, to the testing of trading strategies on historical data. The empirical performance tests carried out show that in such a computationally expensive task like backtesting, escaping the traditional object-oriented approach towards organizing data in a more vertical layout of "structure of arrays" can give a significant increase in performance. Empirical testing shows that use of DOD can speed up the process of features calculation up to 33%, and that organizing data in AoS format may additionally increases performance up to 66%. It is worth noting that the optimizations proposed, except for parallel execution, can also be implemented in real-time trading systems.

The proposed parallelization method, despite having certain drawbacks, has greater scalability, which is very important for multiprocessor systems, especially considering how rapidly the number of cores in modern processors is growing. It should also be said that the proposed data organization scheme is very well compatible with the modern machine learning tools. This is quite useful, because artificial intelligence methods are increasingly being used for the financial information analysis and the development of trading strategies. Also, a more vertical representation of data contributes to better integration with columnar databases, which are often used to process large amounts of data [18].

## References

[1] D. H. Bailey, J. M. Borwein, M. Lopez de Prado, and Q. J. Zhu, “The Probability of Backtest Overfitting,” Journal of Computational Finance, 2015. [Online serial]. Available: http://dx.doi.org/10.2139/ssrn.2326253 [Accessed: Mar. 10, 2021].

[2] D. H. Bailey, J. M. Borwein, M. Lopez de Prado, and Q. J. Zhu, “Pseudo-Mathematics and Financial Charlatanism: The Effects of Backtest Overfitting on Out-of-Sample Performance,” Notices of the American Mathematical Society, vol. 61 (5), pp. 458-471, 2014.

[3] D. Andreev, S. Lyokhin, L. Motaylenko, and S. Verteshev, “Models and algorithms for constructing a formalized description of production technologies,” in Environment. Technology. Resources: Proceedings of the 12th International Scientific and Practical Conference on Information Technologies, Rezekne, 2019, vol. II, pp. 21-27.

[4] R. Fabian, Data-oriented design: software engineering for limited resources and short schedules. Printed by the author, 2016.

[5] F. Wang, P. L. Yu, and D. W. Cheung, “Combining technical trading rules using particle swarm optimization,” Expert Systems with Applications, vol. 41, pp. 3016-3026, 2014.

[6] J. Nenortaite and R. Simutis, “Stocks’ Trading System Based on the Particle Swarm Optimization Algorithm,” Lecture Notes in Computer Science, vol. 3039, pp. 843-850, 2004.

[7] P. Kroha and M. Friedrich, “Comparison of Genetic Algorithms for Trading Strategies,” Lecture Notes in Computer Science, vol. 8327, pp. 383-394, 2014.

[8] J. Ni and C. Zhang, “An Efficient Implementation of the Backtesting of Trading Strategies,” Lecture Notes in Computer Science, vol. 3758, pp. 126-131, 2005.

[9] M. Lopez de Prado, Advances in Financial Machine Learning. Wiley, 2018.

[10] M. Lopez de Prado, Machine Learning for Asset Managers (Elements in Quantitative Finance). Cambridge University Press, 2020.

[11] M. F. Dixon, I. Halperin, and P. Bilokon. Machine Learning in Finance: From Theory to Practice. Springer, 2018.

[12] Antonov, I. Bruttan, D. Andreev, and L. Motaylenko, “The method of automated building of domain ontology,” in Environment. Technology. Resources: Proceedings of the 12th International Scientific and Practical Conference on Information Technologies, Rezekne, 2019, vol. II, pp. 34-37.

[13] D. Andreev, S. Lyokhin, V. Nikolaev, and O. Poletaeva, “Development of software for design ontological representations of production technologies,” in Environment. Technology. Resources: Proceedings of the 12th International Scientific and Practical Conference on Information Technologies, Rezekne, 2019, vol. II, pp. 28-33.

[14] D. Wiebusch and M. E. Latoschik, “Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems,” 2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), Arles, France, 2015, pp. 25-32.

[15] “Intel® 64 and IA-32 Architectures Optimization Reference Manual,” 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf [Accessed: Mar. 10, 2021].

[16] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” Journal of the ACM, vol. 46 (5), pp. 720–748, 1999.

[17] T. Mironov, L. Motaylenko, and D. Andreev, “Investigation of the performance of data structures for order book processing,” in proceedings of the international scientific and practical conference on modern innovations in engineering and manufacturing, Pskov, 2021, pp. 144-148. (in Russian)

[18] E. V. Ivanova and L. B. Sokolinsky, “Parallel processing of very large databases using distributed column indexes,” Programming and Computing Software, vol. 43, pp. 131–144, 2017.