Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

STOCKYARD: A DISCRETE EVENT-BASED STOCK MARKET EXCHANGE SIMULATOR

Jianling Wang
Vivek George
Tucker Balch

College of Computing
Georgia Institute of Technology
801 Atlantic Dr. NW
Atlanta, GA 30332, USA

Maria Hybinette

Department of Computer Science
The University of Georgia
415 Boyd Graduate Studies Research Center
Athens, GA 30602-7404, USA

ABSTRACT

We describe an agent-based stock market simulator built using an asynchronous discrete event simulation framework. The simulator is unique in that it's driven by real-world financial algorithms and protocols; and it's open source. It utilizes an order book bid and ask matching model, and real-world exchange protocols. Our simulation is based on multiple agents interacting through an exchange agent. This method is distinct from those that utilize historical pricing data. Order book execution supports a more realistic interaction between agents. Pricing in our model arises from the dynamics of matching orders in the order book. Our simulator enables the study of market dynamics, and trading strategies using real-world exchange protocols. We present our design and implementation of a market simulator and discuss our initial results using the message protocols defined by NASDAQ: OUCH and ITCH. Our initial results demonstrate StockYard's capability and efficiency in simulating markets with realistic trade volumes.

1 INTRODUCTION

We have designed and implemented a stock market simulator: StockYard. It is an agent-based discrete event simulator that provides a virtual environment to study the dynamics of the market. StockYard is implemented using an asynchronous discrete event simulation framework. Discrete event simulation has been shown to be effective and flexible to study the effects of security prices in real-world events that includes a mix of different investor strategies such as value or momentum type of investor (Jacobs et al. 2004, and Jacobs et al. 2010).

We describe a framework that enables analysts to investigate phenomena such as market impact and co-location. Market impact is the effect on the market when an investor or trader sells or buys a stock. For example, a (large) buy order may drive up the price of a stock while a (large) sell order may drive down the price. We are interested in enabling the investigation of both the magnitude and latitude of market impact, i.e., how much does the price go up or down, and how long does the impact persist. We are not aware of another framework that provides such capabilities.

Co-located traders (those with machines located physically at the exchange) may have an advantage over other traders because they can observe and respond to activities at the exchange more rapidly than more distant agents. Co-located machines are able to access stock prices and run computer models to decide whether to buy or sell stocks more quickly (before anyone else). We are interested in providing a framework to study the value of co-location, not just with regard to proximity to a particular exchange, such as the New York Stock Exchange (NYSE), NASDAQ, or the London Stock Exchange, but also the value of having access to different exchanges simultaneously.

The StockYard platform includes: 1) A discrete event simulation kernel that supports multiple agents interacting via messages; 2) An exchange agent that supports protocols modeled on those supported by NASDAQ; 3) Trading agents that interact with the exchange by sending orders to trade stocks; 4) Data feed agents that supply timely information to the trading agents that may influence appropriate pricing.

In addition to the kernel and the exchange agent, the software distribution includes a set of random or "noise" trading agents that provide an "ecosystem" for testing market hypotheses. Noise traders are often called uninformed traders. Noise traders were introduced to incorporate traders that trade on information based on noise (they are called uninformed because these individuals may believe that they act on relevant information)(Kyle 1985, Black 1986). Noise traders are currently implemented by generating orders using a random number generator with prices related to current market prices. We discuss this in more detail in a later section. The trading agents in our simulator compete against each other by placing orders to an exchange during a simulation run.

The exchange in StockYard is emulated by an exchange agent. The exchange agent accepts orders to buy or sell assets from multiple trading agents. In order to match buy and sell orders from different trading agents, the exchange maintains an "order book" to track which agents want to buy or sell at which prices. The exchange executes trades when a price match is found between a buying and a selling agent.

An important feature of the simulator is realism regarding time delays for computational and network latency when interactions occur between trading agents and the exchange. A network delay is calibrated to emulate the distance between the trading agent and the relevant exchange. In other words the time between sending the trade order to execution of the order at the exchange. Network delay between agents is maintained in the form of latency matrix. The elements of the matrix indicate the agents which are taking part in the simulation. The latency matrix stores the delays involved in communication between agents of the simulator and also represent the topology of the connections between the agents. Modeling the network delay enables analysts to study the value of co-location.

In the next sections we discuss the proof of concept and some initial results of our StockYard framework. In our initial implementation, we incorporate the message protocol used by the NASDAQ exchange: ITCH and OUCH.

2 RELATED WORK

A discrete event simulation (DES) system is fast and efficient because the intervening time between state-changes is ignored. It is also amenable to parallelization of computations further speeding up execution, e.g., Time Warp (Jefferson 1985) and Sequential Simulation (Chandy and Misra 1981). DES has widely been used to address problems in health care (Karnon et al. 2012), supply chain management (Tako and Robinson 2012), manufacturing (Negahban and Smith 2014) and financial markets (Jacobs et al. 2004, Jacobs et al. 2010).

StockYard is implemented using an agent-based model. An agent-based model (ABM) is a model that is formed by a set of autonomous agents that interact with their environment (including other agents) through a set of internal rules to achieve their objectives. Agent-based modeling and simulation (ABMS) is useful, usable and already used in a variety of application domains (Macal and North 2009). ABMS helps the research and investigation in social sciences (Axelrod 1997), computation economics (Tesfatsion 2002) and marketing (Negahban and Yilmaz 2014). Many agent-based simulators have been developed (e.g., Swarm (Minar et al. 1996), Mason (Luke et al. 2005), Player/Stage (Gerkey et al. 2003)). There are also simulators that allows a large number of agents, such as SASSY (Hybinette et al. 2006). SASSY is a scalable agent simulation system for discrete event simulation that provides a middleware between an agent-based API and a parallel DES simulation kernel. We have adopted middle layer approach in StockYard to support millions of agents and to run it for a larger time span (days, weeks, months and years).

Agent-based modeling had been applied to the studies of economics activities and proved to be powerful enough (Tesfatsion and Judd 2006). For stock market, it can be represented based on the

behavior of individual investors (Levy et al. 1994). In StockYard we represent individual investors as agents. Agent-based financial market has shown to be effective for dynamics situations where agents can learn and adapt to different investment strategies (LeBaron 2011).

To give some basic background, a stock market is a place where shares of a publicly held company is issued and traded. This is mainly done through an exchange. The stock market is an essential component of the free market as it enables companies to access capital in exchange for a slice of ownership through company shares (Folger 2016). An order book is the component used to record a list of buy and sell orders in a stock exchange. A matching engine (incorporated in an exchange) uses the order book to match buy and sell bids and enables the execution of stock trade at the specified price. In our implementation we use the matching engine presented in the python library to emulate order book matching (Nguyen 2013).

An objective in stock market simulation is to create virtual stock markets emulating the live stock market, so that analysts, traders or investors can practice investment strategies or investigate factors that may or may not have an impact on the stock market. One class of stock market simulators use a process called "paper trading". Paper trading uses virtual currency for trades instead of real currency. These systems use delayed data from real market or random data to mimic price activity in the simulated process. Online stock market games, and trading training platform often uses "paper trading" or virtual currency. An example is Think of Swim game called: Paper money (Thinkorswim 2017). A disadvantage of paper trading is that in this approach stock prices are deterministic and that buy and sell behaviors cannot contribute to the change of the stock price which is not the case when using real currency. In the financial literature there are simulators that use learning behaviors with differing perspectives of past data (Levy, et al. 2004). Levy, et al. （2004） proposes a synchronous approach, but we believe that asynchronous approaches are more flexible and scalable. This view is shared by Jacobs et al. (2004), who proposed a framework called JLMSim. JLMSim is a discrete event simulator that incorporates trading rules (albeit simple strategies) and reproduces the changes in the market by executing buy and sell orders from the order book. However, JLMSim is limited in that it does not provide interface to support the implementation of complex trading strategies.

In our initial implementation of StockYard we focus on the protocols of the NASDAQ stock exchange. The NASDAQ Stock Exchange is the second largest exchange in the world, just behind the New York Stock Exchange (NYSE). NASDAQ enables users to connect to their exchange via the protocols OUCH (NASDAQ 2016) and ITCH (NASDAQ 2015). OUCH enables customers to enter, replace and cancel orders. ITCH allows them to receive information for different stock activities.

We believe simulating real-world message protocols and providing pluggable agents to implement different investment strategies can provide an efficient environment to test out real-world trading effectively. Initial results look promising.

3 SYSTEM DESIGN

StockYard is a dynamic and modular simulator where agents are created at run time. The diagram depicted in Figure 1 illustrates the system design of the StockYard simulator. The framework is modular for extensibility, and allow dynamic creations of agents at runtime. The modules are interconnected via a message protocol. StockYard has a number of components: Bootstrap, Trading Agents, Exchange Agents, Kernel, Order book, and Latency Matrix. These are discussed below.

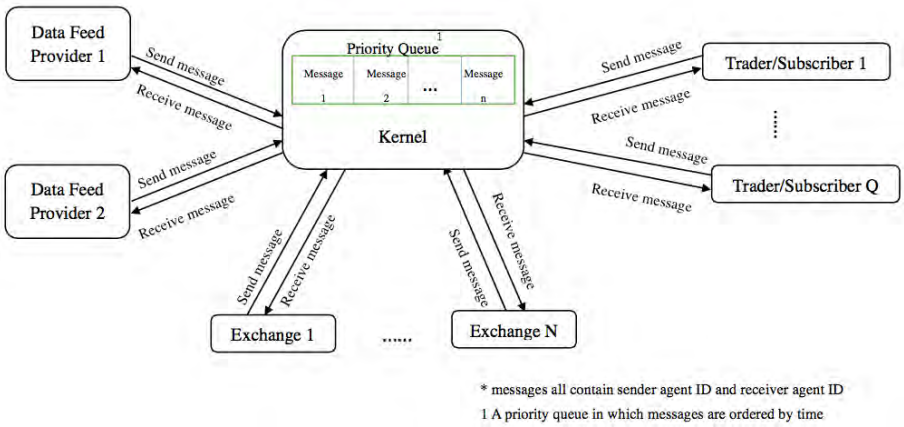

Figure 1: The diagram represents the system design of StockYard simulator.

3.1 Bootstrap

The bootstrap module initializes (agents and exchanges) and starts the simulation. It creates agents with an initial capital, a trading strategy and sets up a subscription to the agents exchange or changes of interest. Exchanges are initialized with an initial list of prices for different stocks. Finally, it initializes the kernel interface between the agents and the exchanges. The bootstrap module is not depicted in the figure above.

3.2 Trading Agents

A trading agent represents a person or stockbroker who trades on the stock market. As such, each trading agent will own a portfolio of stocks with some allocation of capital. Each trader agent follows their own trading strategy by which they place sell or buy orders on the exchange. For example, an agent may use a Bollinger Band Strategy (Schlossberg 2017), Momentum Based Strategy (Bergen 2017) or a Custom-Made Strategy based on machine learning. In our initial implementation the model is simple, and we assume that traders don't change their strategies.

3.3 Exchange Agents

The exchange agents maintain the order book for each stock showing the buy and sell prices and helps agents in interacting with each other. The order book is populated based on the orders placed by the agents. When an order is placed, the exchange updates its order book and checks whether it can fulfill the order or not. If it can, it appropriately update the order book and notify the trader about the outcome of the transaction.

3.4 Kernel

The kernel is the interface between the agents and it drives the interactions between these agents. It distributes messages according to the sender agent IDs and the receiver agent IDs. Messages are ordered by a priority queue and is prioritized by time stamps. The kernel is awaken the trading agents at the time they need to execute a trade by sending them wake-up message, and relay the order entering message from the agent to the exchange. Additionally, once the exchange performs the transaction and notifies the outcome of the transaction, the kernel will help transmit this order executed message from the exchange to the agent.

3.5 Order Book

Each exchange maintains an order book that ultimately determines the stock prices. The order book is utilized by the matching engine that scans the sell and buy orders to find matches. The order book has a list of all pending trades from buyers and sellers of stocks. A trade order by an agent has the following components:

- Type of order: BUY or SELL (also referred to as BID and ASK respectively)
- Stock symbol
- Number of shares to be traded
- Limit / Market order
- Price (if limit order)
- Time Stamp

The order book is sorted whether it is a buy or sell order.

Table 1 illustrates an example of the order book. In Table 1, focusing at buy orders: 20 shares of IBM are ready to be purchased at a price point of $118 (or less), and 10 shares is ready to be purchased at $120 (or less). Now focusing at the sell orders: 25 shares of IBM is ready to be sold at $118 (or higher) and 10 shares are ready to be sold at $130 (or higher). The matching engine scans through the order book and in this example it matches the 10 shares "buy" order (ID 2) of IBM stock to be sold at $120 and immediately after that 15 share of IBM will be sold at $118. The new update order book would is depicted in Table 2.

Table 1: A sample order book table which consists of Buy and Sell limit orders.

Limit Order – Buy Table				
ID	Stock	Quantity	Price	Time Stamp
1	IBM	20	118	3
2	IBM	10	120	2
Limit Order – Sell Table				
ID	Stock	Quantity	Price	Time Stamp
3	IBM	25	118	5
4	IBM	10	130	7

Table 2: The updated order book derived from Table 1.

Limit Order – Buy Table				
ID	Stock	Quantity	Price	Time Stamp
1	IBM	5	118	3
Limit Order – Sell Table				
ID	Stock	Quantity	Price	Time Stamp
4	IBM	10	130	7

3.6 Latency Matrix

Low communication delays with the exchange is of importance to traders as it enables them to respond to changes in stock markets at a faster pace. High Frequency Hedge Funds often co-locate servers on an exchange floor to shave off milliseconds worth of communication delays. In order to accurately simulate

this mechanics in financial markets, it is important to accurately simulate these communication latencies between agents and their distances from the actual exchange. In StockYard we consider the network topology as input and generate a latency matrix. The software architecture implemented in StockYard that accounts for network delay is depicted in Figure 2.

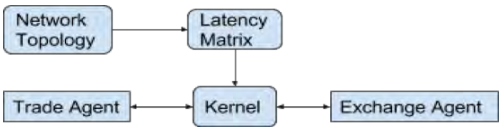

Figure 2: The design we have used to add communication latency when agents communicate with each other.

The Network Topology input is represented in the Pajek Net Format (NetWiki 2011). The agent vertices comprise of trade agents and exchange agents. The Latency Matrix stores the minimum latency required to communicate between each pair of agents. The kernel has direct access to the latency matrix. When an agent requests a message to be sent to another agent e.g., to an exchange agent, the kernel reads the appropriate latency value from the latency matrix and incorporates its value in the message send time (schedule time).

4 SIMULATION FLOW

We will now discuss the simulation flow of events in StockYard. A flow chart of these events are depicted in Figure 3.

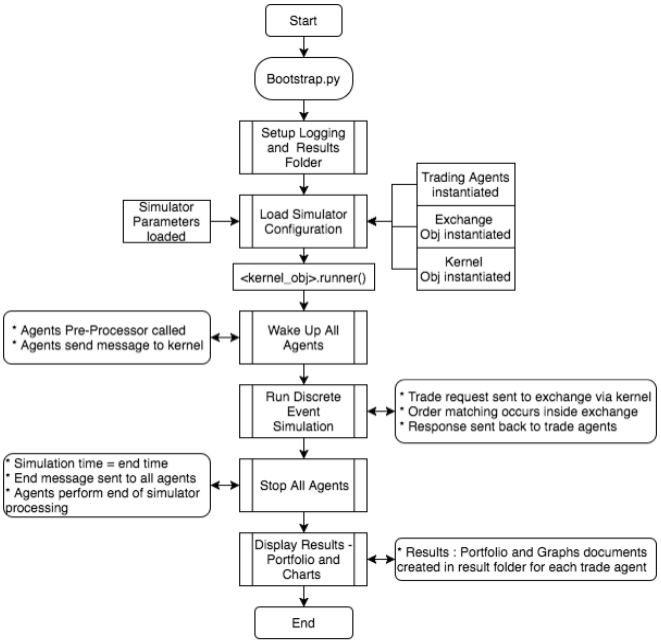

Figure 3: Flowchart of the stock market simulator.

StockYard first instantiates the bootstrap module. The bootstrap module instantiates and initializes both the trading agents and the stock market exchange agent based on properties from a bootstrap file. Trading agents are initiated at their initial capital (currency) and the volume of stock of their initial portfolio, and the strategies that they employ. In our current implementation the strategy is static and does not change, e.g., they may employ a strategy that leverages Bollinger Band. Next, the Exchange agents

are initialized with the stocks at the exchange and the appropriate exchange matching engine. The kernel is instantiated and initialized by coupling it with the trading and exchange agents. The kernel controls and drives agents via time-stamped messages. The kernel gets control via a kernel.runner() method. The stock market simulation is run until the market closes (end of day). StockYard provides a monitoring graphical interface that depicts agents running, and plots price variations of different stocks or a portfolio value as time evolves. For more detailed information of the buy and sell orders of traders and the orders of different symbols, users can check the generated csv files. This can be replayed off-line. Subscriber agents can subscribe to statistics while online and off-line. Even more detailed monitoring can be provided by console logging and by turning on a debugging mode, however debugging may slow down the execution.

5 TRADING AGENTS

Currently, in our design, StockYard provides two types traders, random and strategy-based traders. They are described below.

5.1 Random Traders

Random traders provides a sense of how the market would behave and provide a specific kind of environment for the interaction of strategy-based traders. Random traders have configurable parameters determined by a configuration file. The parameters primarily define the bounds on how these traders place their orders. Below are the types of random traders supported by StockYard:

- **Random Positive Traders**: These traders make Buy/Sell orders at higher price as they believe that the stock price would increase.
- **Random Negative Traders**: These traders order Buy/Sells at lower price as they believe that the stock price would decrease.
- **High Volatility Trader**: These traders trade a price which is highly further away from the current traded price.
- **Low Volatility Trader**: These traders trade a price which is highly further away from the current traded price.

5.2 Strategy-Based Traders

Strategy-based Traders work on a specific kind of strategy. They analyze the system market based on a strategy to make Buy/Sell calls:

- **Momentum-Based Trader:** Traders place buy and sell orders based on the volume weighted momentum of stock prices in an particular direction over specific time periods (Bergen 2017).
- **Bollinger Band-based traders:** Traders try to gauge overbought and oversold price levels for stock based on moving averages and their standard deviations for different time periods. This strategy is very popular in range bound markets where the stock price keeps fluctuating within a specific price range (Schlossberg 2017).
- **Machine Learning-Based Traders**: Traders try to predict stock prices by using machine learning algorithms such as Regression, Decision Trees and Neural Networks to identify patterns from historical stock data.

We are in process of implementing these strategies.

6 EXCHANGE MESSAGE SYSTEM

In StockYard, the communication between trading agents, the data feed agents and exchange agents is modeled using real-world message protocols. In our current implementation we have implemented the message protocol of NASDAQ. To implement the NASDAQ protocol we deploy data feed provider and transaction price data feed agents. The data feed agents interact with the trader agents and the exchange agents. This framework is depicted Figure 4.

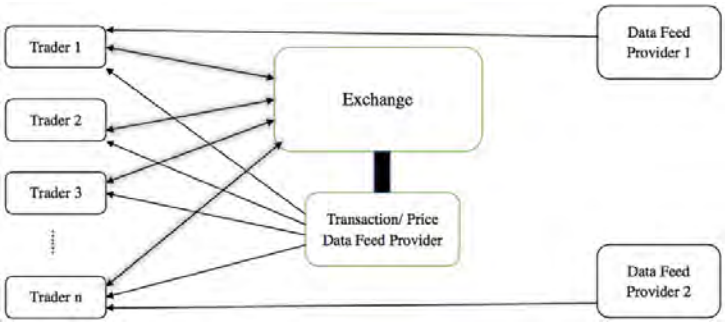

Figure 4: Message flow between different types of agents in StockYard.

NASDAQ supports the OUCH and ITCH message protocols. These protocols are used by traders while interacting with the NASDAQ stock exchange. The OUCH protocol is used to enter orders and receive order execution information. Agents can also subscribe to other data feed agents (not included in the "enter" orders), by sending out a subscription request. These messages are implemented using the ITCH protocol. The interaction of OUCH and ITCH is depicted in the Figure 5.

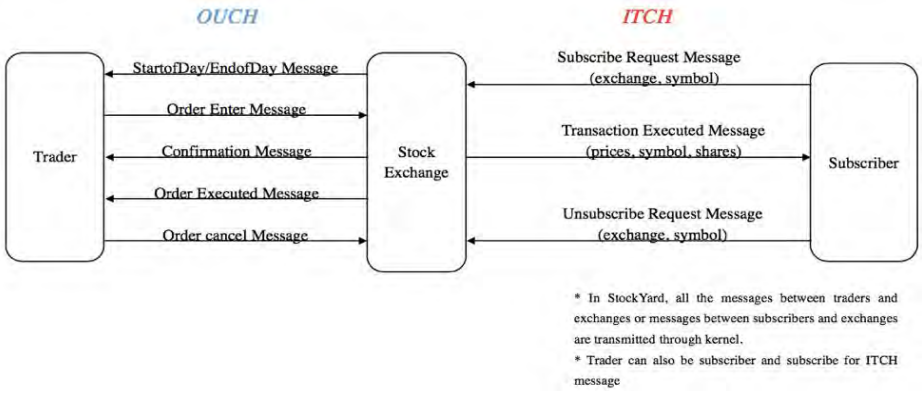

Figure 5: Message protocol in StockYard.

7 EVALUTION

At this stage, we mainly focus on the extendibility and functionality of StockYard. We want it to be more realistic so that it can be used to investigate phenomena such as market impact and co-location. Currently, there is no other market simulator which can support pluggable agents, real-world message protocols and the execution of order book at the same time. Thus we can't find an appropriate simulator that can run the similar scenarios as StockYard.

In order to test the feasibility of StockYard we examined the trades in the Trade and Quote (TAQ) data set from the Wharton Research Data Services (WRDS 2011). From this data set one can estimate that there are on average 25,000 transactions per trading day on a stock at NASDAQ.

To test whether StockYard supports a volume of trades similar to NASDAQ, we tested StockYard on a platform of a 1.7 GHz Intel Core i7 CPU and 8 GB DDR3 main memory. In our experiment we varied the number of trading agents from 100 to 1,000 and measured the execution time of the simulator program. In our first experiment StockYard is in monitoring mode and the result is depicted in Figure 6. The performance demonstrates that as traders increase, the execution time of the program increases linearly. This is reasonable because exporting and plotting data takes more time than running the simulating process. It takes more time to plot and export the results for 1000 agents than 100 agents because the program needs to create and write in more files.

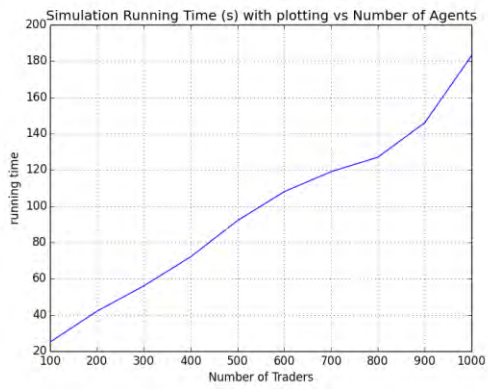

Figure 6: With plotting turning on, to execute ~25000 transactions, the running time of StockYard is proportional to the number of traders in the simulation.

Plotting, and monitoring adds overhead so we ran another set of experiments with monitoring turned off or disabled. The result is depicted in Figure 7. As depicted StockYard only took 10 seconds to run an entire day of trading and supporting 25,000 trades. The number of agents (traders) will not influence the running time of the simulating process.

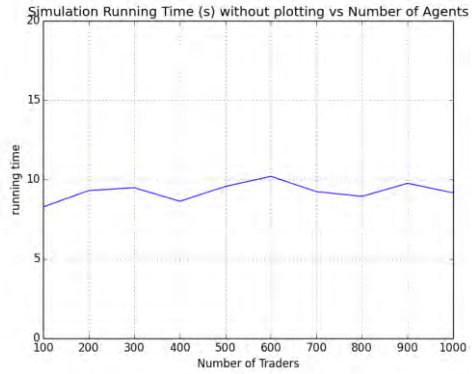

Figure 7: With plotting turning off, StockYard can support 25,000 transactions and 1,000 traders in 10 seconds.

8 CONCLUSION AND FUTURE WORK

We have presented the design and results of an initial implementation of StockYard mimicking real-world stock market environment with order-book execution using discrete event simulation. StockYard is available open source and is implemented in Python. StockYard enables users to customize trader agents,

market agents and subscriber agents with an agent-based APIs. Initial results is promising in providing an effective and efficient simulator. We hope that StockYard will help researchers exploring various market scenarios and testing trading strategies.

We are currently working on providing a richer set of trading agents (including agents that incorporates learning strategies). We are also in the process of implementing a Hidden-Markov-Model to calibrate our simulator with real-world data, and extend support to protocols of other exchanges, e.g., the New York Stock Exchange (NYSE).

REFERENCES

Axelrod, R. 1997. "Advancing the Art of Simulation in the Social Sciences". In *Simulating Social Phenomena*, 21-40. Berlin Heidelberg: Springer.

Black, F. 1986. "Noise". *The Journal of Finance* 41:529-543.

Bergen, J. V. 2017. "Introduction to Momentum Trading". http://www.investopedia.com/articles/trading/02/090302.asp.

Chandy, K. M., and J. Misra. 1981. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations". *Communications of the ACM* 24:198-206.

Folger, J. 2016. "Introduction to Order Types: Limit Orders Book". http://www.investopedia.com/university/intro-to-order-types/limit-orders.asp.

Gerkey, B. P., R. T. Vaughan, and A. Howard. 2003. "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems". In *the 11th International Conference on Advanced Robotics* 1:317-323.

Hybinette, M., E. Kraemer, Y. Xiong, G. Matthews and J. Ahmed. 2006. "SASSY: A Design for A Scalable Agent-Based Simulation System Using A Distributed Discrete Event Infrastructure". In *Proceedings of the 2006 Winter Simulation Conference*, edited by L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 926-933. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Jacobs, B. I., K. N. Levy, and H. M. Markowitz. 2004. "Financial Market Simulation". *The Journal of Portfolio Management* 30:142-152.

Jacobs, B. I., K. N. Levy, and H. M. Markowitz. 2010. "Simulating Security Markets in Dynamic and Equilibrium Modes". *Financial Analysts Journal* 66:42-53.

Jefferson, D. and H. Sowizral. 1982. "Fast Concurrent Simulation Using the Time Warp Mechanism. Part I. Local Control" (No. RAND/N-1906-AF). Rand Corp Santa Monica CA.

Karnon, J., J. Stahl, A. Brennan, J. J. Caro, J. Mar, and J. Möller. 2012. "Modeling Using Discrete Event Simulation A Report of the ISPOR-SMDM Modeling Good Research Practices Task Force–4". *Medical Decision Making* 32:701-711.

Kyle, A. S. 1985. "Continuous Auctions and Insider Trading". *Econometrica: Journal of the Econometric Society* 53:1315-1336.

LeBaron, B. 2001. "A Builder's Guide to Agent-Based Financial Markets". *Quantitative Finance* 1:254-261.

Levy, M., Levy, H. and S. Solomon. 1994. "A Microscopic Model of the Stock Market: Cycles, Booms, and Crashes". *Economics Letters* 45:103–111.

Luke, S., C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. 2005. "MASON: A Multi-Agent Simulation Environment". *Simulation* 81:517–527.

Macal, C., and M. J. North. 2009. "Agent-Based Modeling and Simulation". In *Proceedings of the 2009 Winter Simulation Conference*, edited by, edited by M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, 86-98. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Minar, N., R. Burkhart, C. Langton, and M. Askenazi. 1996. "The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations". Santa Fe Institute.

NASDAQ OMX Group. 2016. "O*U*C*H Version 4.2 Updated July 26, 2016". http://www.nasdaqtrader.com/content/technicalsupport/specifications/tradingproducts/ouch4.2.pdf.

NASDAQ OMX Group. 2015. "NASDAQ TotalView-ITCH 5.0 Interface Specification". http://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/NQTVITCHspecification.pdf.

Negahban, A. and J. S. Smith. 2014. "Simulation for Manufacturing System Design and Operation: Literature Review and Analysis". *Journal of Manufacturing Systems* 33:241-261.

Negahban, A. and L. Yilmaz. 2014. "Agent-Based Simulation Applications in Marketing Research: An Integrated Review". *Journal of Simulation* 8: 129-142.

NetWiki. 2011. "Pajek Net And Paj Format". http://netwiki.amath.unc.edu/DataFormats/PajekNetAndPajFormats.

Nguyen, M. 2013. "Order Book Python Package 0.1.2". https://pypi.python.org/pypi/OrderBook/0.1.2.

Onggo, B. S. 2010. "Running Agent-Based Models on A Discrete-Event Simulator". In *Proceedings of the 24th European Simulation and Modelling Conference*, 51-55.

Schlossberg, B. 2017. "Using Bollinger Band® 'Bands' to Gauge Trends". http://www.investopedia.com/articles/trading/05/022205.asp.

Tako, A. A., and S. Robinson. 2012. "The Application of Discrete Event Simulation and System Dynamics in the Logistics and Supply Chain Context". *Decision Support Systems* 52:802-815.

Thinkorswim. 2017. "Paper Money". https://www.thinkorswim.com/t/trading.html?webpage=paperMoney.

Tesfatsion, L. 2002. "Agent-Based Computational Economics: Growing Economies from the Bottom Up". *Artificial Life* 8:55-82.

Tesfatsion, L. and K.L. Judd eds. 2006. *Handbook of Computational Economics: Agent-Based Computational Economics*. Elsevier.

Wharton Research Data Services. 2011. "NYSE Trade and Quote (TAQ)". http://www.whartonwrds.com/datasets/nyse-taq/.

AUTHOR BIOGRAPHIES

JIANLING WANG is currently a Ph.D. student of Department of Computer Science and Engineering at Texas A&M University. She received her M.S. degree in Computer Science from Georgia Tech in 2017. While at Georgia Tech, she was a member of the QuantSoftware Research Group led by Prof. Tucker Balch. Her email address is jwang713@gatech.edu.

VIVEK GEORGE is a software engineer at RetailMeNot. He received his M.S Degree in Computational Science & Engineering from Georgia Tech. A data science enthusiasts with and visually creative mindset, he is currently pursuing a career at the intersection of ML and Interactive Design. His e-mail address is vivek.george.gatech@gmail.com.

TUCKER BALCH is a Professor of Interactive Computing at Georgia Tech and managing partner of Lucena Research (fresh startup). He develops and uses statistical machine learning and data mining algorithms to augment and inform trading decisions. He teaches a course at Georgia Tech on this topic, and he has also built an open source software system to support this effort. His email address is tucker@cc.gatech.edu.

MARIA HYBINETTE is an Associate Professor of Computer Science at the University of Georgia, performing research in high performance simulation systems. Her e-mail address is maria@cs.uga.edu.