SCALABM

DAVID R. PUGH, DANIEL F. TANG, J. DOYNE FARMER

1. Objective

ScalABM is a *community driven*, *open-source* project to develop a *user-friendly* modeling platform and tool stack for building *scalable*, *data-driven*, and *reproducible* agent-based models (ABMs) of *economic* systems on the JVM using Scala and Akka.

1.1. **General requirements.** The high-level architecture design and the software development processes of the ScalABM project should mimic the layered architecture design and open-source development processes that have become standard "best practices" for developing Reactive "Fast Data" web applications.[1]

1.1.1. *...community driven, open source...* Progress in the building of large scale models of economic systems has been significantly hindered by...

- ...the lack of a common modeling platform as well as a common tool stack for data management and analytics. The lack of a common modeling platform and tool stack has, to a large degree, made it impossible to compare competing models.
- ...the lack of access to model source code. Lack of access to a model's source code dramatically increases the costs associated with reproducing that model's results and makes it difficult or impossible for other researchers to extend that model in the future.

The development of all ScalABM libraries will be driven by the needs of the economics ABM community. All software development will take place in public: from the start of the project all source code for the ScalABM project will be hosted on GitHub under a permissive Apache 2.0 license that allows for free use of the software libraries (even in commercial applications).

1.1.2. *...user-friendly...* The ability to interactively explore ABMs in near-real time is crucial for developing intuition about the mechanisms driving the key results of these models. In order to facilitate interactive exploration of ABMs built using the ScalABM libraries all models should have two, complementary user interfaces:

Date: May 27, 2016.

[1] See figure 1 from Wampler (2015) for a similar high-level summary of a Reactive, "Fast Data" web application architecture.

(1) a user-friendly, web-browser based user interface based around the Play web framework. The GUI should leverage existing, high-quality libraries for real-time data streaming, analysis and visualization.
(2) an intuitive and consistent command line interface (CLI). In addition to facilitating efficient batch processing of model simulations (i.e., parameter sweeps), the CLI should allow for easy replication of any particular model simulation.

In addition to making it easy to interactively explore existing ABMs we want to make sure that the ScalABM libraries are structured in a way that minimizes the amount of development time needed to reconfigure and existing model or build a completely new model. In particular...

- ...in order to maximize reuse of code, models built using our toolkit should be composed of mostly existing components. Leveraging mostly existing components reduces development time for a new model to that needed to create a few novel components together with the time needed to wire all the desired model components together.
- ...the process of wiring model components together (which includes the specification of all model parameters, agent behavioral strategies, etc), sometimes called dependency injection (DI), should be as simple and transparent as possible. The wiring process should be specified in a single, *human-readable*, model configuration file.
- ...the build process (i.e., specification of dependencies, platform specific build options, etc) for a particular model should be completely specified in a single build file.

1.1.3. *...scalable...* Aggregate behavior of many (most?) real-world economic systems fundamentally depends on system size. Put another way, system size is a key parameter for modeling economic systems and in order to accurately model the dynamics of some systems, we may need to build and simulate models that are as close to observed scale as possible.

Broadly speaking, there are two kinds of solutions to the scaling problem...

- ...software solutions: ScalABM leverages the Actor model of concurrency as implemented in the Akka library to build ABMs that are highly concurrent, distributed, and message-driven.
- ...hardware solutions: because models built using the ScalABM libraries are basically scalable web applications, we are able to leverage the massively multi-core cloud computing platforms that are quickly becoming the dominant form of large-scale computing outside of academia.

Existing solutions to the scaling problem either rely on access to university (or national) supercomputing resources or leverage the computing power of GPU clusters but then force researchers to express their ABMs in the

restrictive GPU computing environment. We believe that our approach to the scaling problem has several benefits over existing solutions.

- Outsourcing of cluster management to third party provider. ABMs built using our framework can be "containerized" using technologies such as Docker or Vagrant and then deployed on a third-party cloud computing service provider such as AWS, Google Compute Engine, Heroku, Mesosphere, etc. This third party provider then handles all of the intricacies involved with scaling up the model on the cluster to meet our needs.
- No longer dependent on access to university or national supercomputers enhances the reproducibility or our research. The ability to "containerize" an ABM built using out framework means that researchers not directly involved in developing a model can still access everything (even down to the operating system) necessary to completely reproduce that model's output. The container can be used to run the model locally on a laptop or sent to a third party provider to scale up via the cloud.

1.1.4. *...data-driven...* We want to build ABMs that can be initialized and validated against real-world data. In order for our ABMs to be data-driven, we need to think carefully about how ScalABM we will manage the flow and storage of data (both model generated data as well as real-world data). There are (at least!) three components to data management: access, analytics, and storage.

- Access: A running ABM will generate a large volume of data. Model generated data might be stored, sent to a data analytics engine, or logged out to a file(s). Additionally, data will likely flow in the reverse direction. In particular, agents in a running ABM may need to read data from a data store (for example, we might wish to initialize economic agents using real-world data).
- Analytics: A running ABM is a continuous source of data whose volume is not predetermined. Put another way: ABMs generate reactive data streams. Our data analytics components should therefore include tooling for processing and analyzing streaming data. In addition to processing and analyzing streaming data, our data analytics tool stack should include tooling for dealing with "batch" or "mini-batch" computations. Such batch processing jobs would be performed either relatively infrequently on streaming data or upon completion of a model simulation. The Apache Spark project is a Scala based project for large-scale data analytics that supports batch, mini-batch and stream processing.
- Storage: Our ABMs should have read/write access to a scalable data store. Additionally, the data analytics components will need a source of input data. In order to avoid simulations being I/O bound, our data store should have extremely fast write access. We

should design our tool stack to integrate cleanly with leading NoSQL database architectures such as Apache Cassandra and Neo4j.

1.1.5. *...economic...* ScalABM is not intended to be a general purpose platform for ABM. Rather we want to build a toolkit that is designed to facilitate the construction of agent-based models of economic systems. An economy is populated with many seemingly disparate types of agents (i.e., consumers, producers, financiers, government, some markets, etc). We need to distill the core essence (in terms of data and behaviors) of these different agents into a multi-layered Application Programming Interface (API) defining a generic *economic agent* that can then be specialized to the various types of economic agents needed for any particular model. Our hope is that by doing this we can reduce the effort needed for the parts of agent-based modeling that consume a great deal of software development time. We also hope that by identifying the key components we can build standard, highly modular interfaces that make it easy to interchange components of models.

1.1.6. *...reproducible...* The results of many (most?) ABMs are not easily reproducible. Often this is due to some combination of lack of availability of source code, poor documentation, and insufficiently detailed research papers. Reproducibility is further hindered by the lack of use of "best practices" for software development.

In order to facilitate the reproducibility of results generated by models built using the ScalABM libraries, the source code for all ScalABM libraries will strive to adhere to the following "best practices"...

- ...be under version control using Git, and publicly available via GitHub under a permissive Apache 2.0 license.
- ...will be subject to continuous unit testing with code coverage data analytics will produced by Coveralls, etc.
- ...all source code will be subject to continuous static analysis using Codacy, Code Climate, etc.
- ...all source code will be extensively documented. Documentation will be subject to automatically re-built, and integrated on commit, and hosted with the source code on GitHub.
- ...all source code will be subject to continuous integration using either Travis CI, Jenkins, etc.

2. High-level design

2.1. **Platform architecture.** The high-level design of our platform's architecture should mimic the layered architecture of a reactive, "Fast Data" web application. Figure 2.1 summarizes the high-level design of our platform architecture.[2]

[2] See figure 1 from Wampler (2015) for a similar high-level summary of a Reactive, "Fast Data" web application architecture.

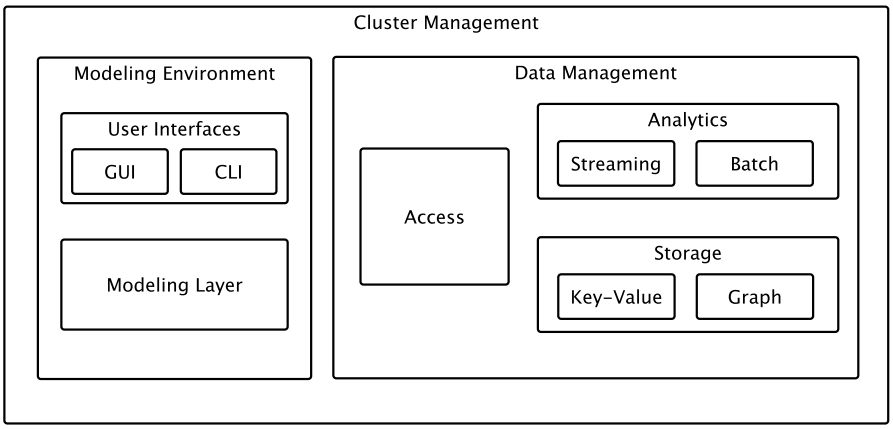

FIGURE 1. High-level architecture design for ScalABM.

2.1.1. *Cluster management.* Current approaches to running large-scale ABMs leverage either...

- University/national super-computers: Use FLAME or Repast to build large-scale, complicated models. Both FLAME and Repast use some kind of message passing (but NOT "peer-to-peer" message passing) under the hood.
- GPU computing: Use CUDA, FLAME GPU (or similar) to build large-scale, simple models.

Our strategy for running ABMs at scale will instead leverage massively multi-core cloud computing clusters that are quickly becoming the dominant form of large-scale computing outside of academia.

Benefits to our approach:

- Outsourcing of cluster management to third party provider. ABMs built using our framework can be "containerized" and sent off to a third-party cloud computing service provider such as AWS, Google Compute Engine, Heroku, Mesosphere, etc. This third party provider then handles all of the intricacies involved with scaling up the model on the cluster to meet our needs.
- No longer dependent on access to university/national super-computers enhances the reproducibility or our research. The ability to "containerize" an ABM built using out framework means that researchers not directly involved in developing a model can still access all material necessary to completely reproduce that model's output. The container can be used to run the model locally on a laptop or sent to a third party provider to scale up via the cloud.

2.1.2. *Modeling environment.* The modeling environment consists of user interfaces and the modeling layer. The "front end" of our modeling environment should consist of two, complementary user interfaces.

(1) A user-friendly, web-browser based graphical user interface (GUI). The GUI should facilitate interactive exploration of an existing model in near real-time. The GUI should support real-time data streaming, analysis, and visualization.
(2) An intuitive and consistent command line interface (CLI). In addition to facilitating efficient batch processing of model simulations (i.e., parameter sweeps), the CLI should allow for easy replication of any particular model simulation.

The "back end" of our modeling environment is the modeling layer which consists of the actual source code libraries used to implement our ABMs. Important characteristics of our modeling layer:

- The modeling layer should facilitate the construction of new ABMs out of pre-existing, modular components. Novel model components should be able to easily extend pre-existing components.
- Model configuration, including specification of all model parameters as well as the "wiring" of model components, should be specified in configuration files that are separate from the actual source code.

The modeling layer itself is organized into a number of sub-layers: a behavioral layer, an information layer, and a data analytics layer. See figure 2.1.2.

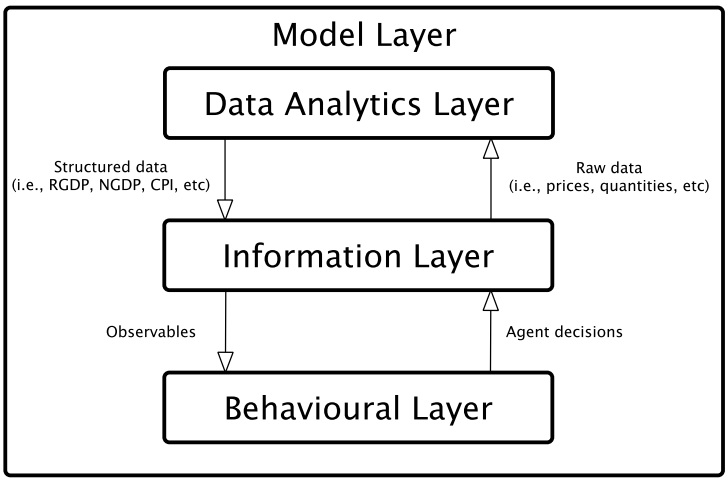

FIGURE 2. High-level organization of the modeling layer.

Behavioral layer. An economy is populated with many seemingly disparate types of agents (i.e., consumers, producers, financiers, government, etc). Our goal with the behavioral layer is to distill the core essence (in terms of data and behaviors) of these different agents into a multi-layered API defining a generic *economic agent* that can then be specialized to the various types of economic agents needed for any particular model.

Several key features distinguish *economic* agents from more generic types of agents. At a minimum these features include: purpose driven (or goal oriented) behaviour, an ability to learn, and the ability to anticipate future events. This suggests that our behavioral layer will need APIs for...

- Goals or objectives (and their associated behavioral rules): Our goals/objectives API needs to be as un-opinionated as possible as prospective users of ScalABM are likely to have strong opinions on how to define appropriate goals and decision rules for their agents. At the same time, we will need to have some type of underlying null model of agent goals/objectives. Utility maximization, Belief-Desire-Intent (BDI), probabilistic discrete choice (AKA, random utility maximization) are possibilities.
- Learning rules: A large number of various learning rules/mechanisms have been proposed in the academic literature. Roughly, learning rules seem to fall into two camps: learning through previous experience (i.e., reinforcement learning) and learning through observation (i.e., belief learning). Useful references for learning rules for ABMs are the two handbook chapters Brenner (2006) and Duffy (2006).
- Expectations formation rules: A large number of various expectation formation rules have been proposed in the literature. Useful references for expectation formation rules are Hommes (2006), Anufriev and Hommes (2012), Woodford (2013), Assenza et al (2014).

High-level description of an agent in our framework is a layered collection of behaviors and decision rules...

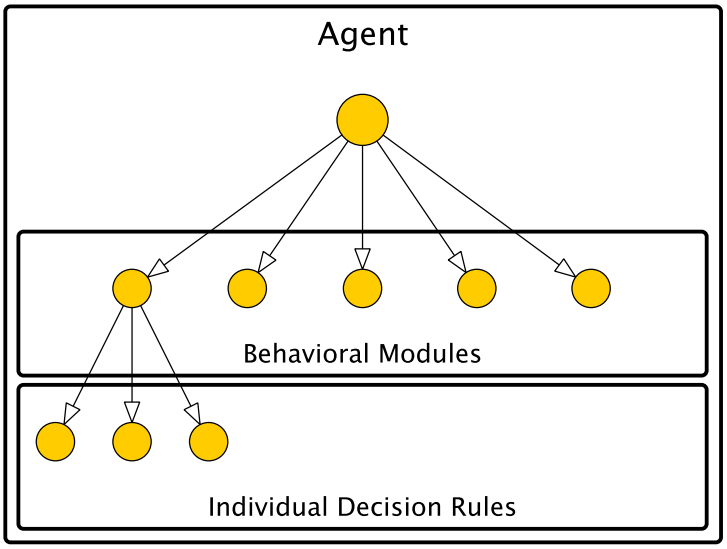

FIGURE 3. An agent in our framework is a layered collection of behaviors and decision rules.

Concurrent communication between real-world economic agents is a fundamental fact of economic life. Inter-agent communication can be either direct (i.e., "peer-to-peer") or indirect (i.e., via market institutions). In order to model both direct and indirect communication between agents in our framework we leverage the Actor model of concurrency. The Actor model treats "actors" as the universal primitives of concurrent computation: in response to a message that it receives, an actor can make local decisions (given its information set), possible create more actors, send more messages, determine how to respond to the next message received, etc.

Stock flow consistency is an important property of large-scale macro models, but is not a property that makes sense to impose in general for our framework. Need to have a behavioral module that implements stock flow consistent accounting rules.

Our economic agents will need to condition their decisions on an information set. In contrast to (most) DSGE models, information sets in our framework will be highly heterogeneous. The information set for any particular economic agent should consist of three components:

- Private information: information that is idiosyncratic to a particular agent.
- Public information: information that a particular agent shares with one or more agents.
- Global information: information that is shared between all agents.

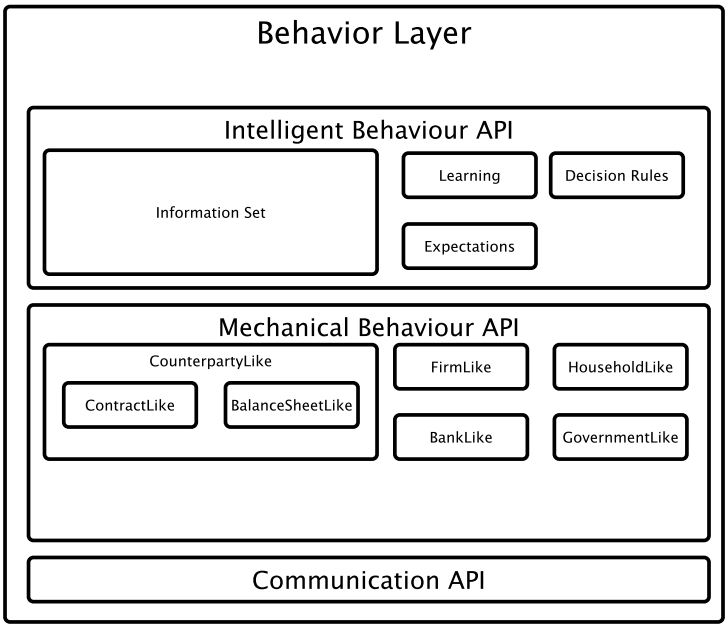

FIGURE 4. ???

Information Layer. Markets are institutions that aggregate data: markets take agent decisions/choices as raw data which they aggregate into prices and quantities. These prices and quantities are, typically, but not always observable by others. Most market prices are public information.

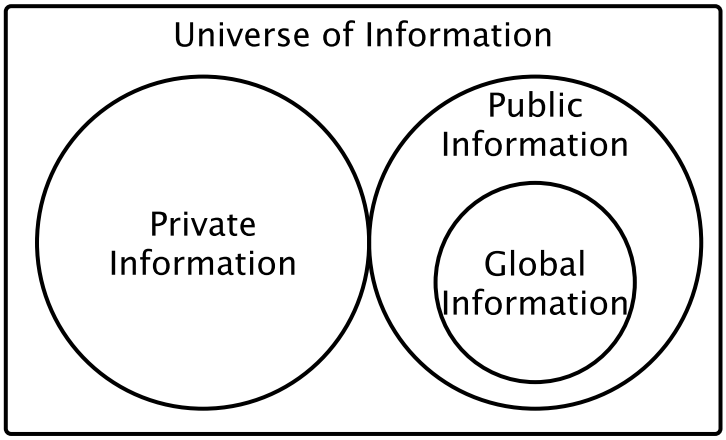

FIGURE 5. Information set for a particular economic agent.

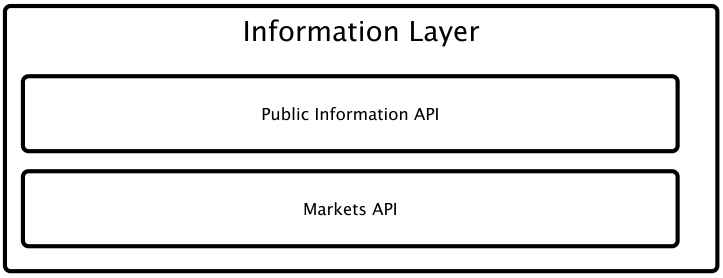

FIGURE 6. Information layer consists of the markets API and a public information API.

Data Analytics Layer. The data analytics layer encapsulates the functionality for aggregating raw data into additional structured data that agents can use for making decisions. Specifically, the data analytics layer would contain classes and methods for computing standard macro economic indicators (i.e., real and nominal GDP, stock market index, house price index, consumer price index, unemployment rates, etc). Users should be able to extend the data analytics capabilities as needed for a particular model.

2.1.3. *Data Management.* In order for our ABMs to be data-driven, we need to think carefully about how our framework we will manage the flow and

storage of data (both model generated data as well as real-world data). There are (at least!) three components to data management: access, analytics, and storage.

- Access: A running ABM will generate a large volume of data. Model generated data might be stored, sent to a data analytics engine, or logged out to a file(s). Additionally, data will likely flow in the reverse direction. In particular, agents in a running ABM may need to read data from a data store (for example, we might wish to initialize economic agents using real-world data).
- Analytics: A running ABM is a continuous source of data whose volume is not predetermined. Put another way: ABMs generate reactive data streams. Our data analytics components should therefore include tooling for processing and analyzing streaming data. In addition to processing and analyzing streaming data, we will also need to preform various "batch" or "mini-batch" computations. Such batch processing jobs would be performed either relatively infrequently on streaming data or upon completion of a model simulation. Our data analytics should include tooling for dealing with batch computations.
- Storage: The modeling layer should have read/write access to a scalable data store. Additionally, the data analytics components will need a source of input data. In order to avoid simulations being I/O bound, our data store should have extremely fast write access.

3. Implementation

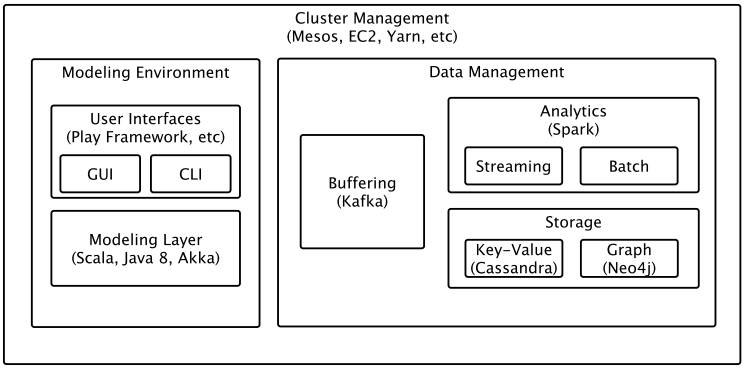

Figure 7. Macro-grain architecture for ScalABM.

3.1. **Platform architecture.**

3.1.1. *Modeling environment.* Browser based GUI interfaces should be built using the Play web application framework. The Play framework cleanly integrates Javascript libraries for real-time data analysis and visualization with backend model libraries.

3.1.2. *Data Management.* Broadly speaking, we will need to store two types of data: real-world data and model generated data. The way in which we store our data should be heavily influenced by the types of questions we intend to ask of our data. There are two kinds of questions that we will want to ask of our model generated data:

(1) Questions about cross-sectional and time series properties of model generated data. For these types of questions NoSQL Key-Value databases, such as Cassandra, are ideal data stores. Key-Value databases are designed for storing data in a schema-less way. In a key-value store, each datum consists of an indexed key and a value, hence the name.
(2) Questions about the network structures between model agents. For questions about network structure, graph databases are ideal. Graph databases, such as Neo4j, are designed for data whose relations are well represented as a graph and has elements which are interconnected, with an undetermined number of relations between them.

3.2. **Modeling Layer.** ScalABM leverages the Actor model of concurrency as implemented in the Akka library to model both direct and indirect communication between economic agents using concurrent, asynchronous message passing.[3] All agents in ScalaABM are implemented as hierarchical trees of Akka Actors.

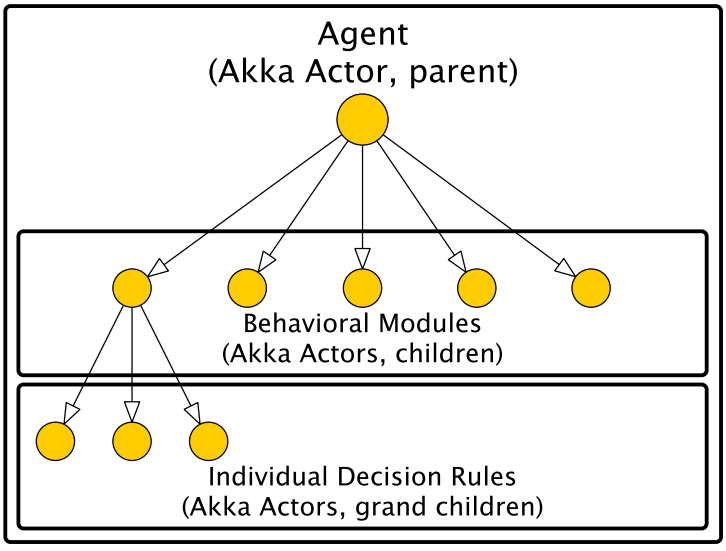

FIGURE 8. ???

[3] Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.

This design allows an agent to itself be distributed across a cluster efficiently.

3.2.1. *Communication Layer.* In order to impose structure on inter and intra agent communications we have developed an API for a scalable agent communication language for economic agents tentatively called ScalACL. The ScalACL API specifies...

- a set of abstract message types that impose structure on the messages passed between a group of economic agents,
- a set of abstract protocols that impose structure on conversations (i.e., sequences of messages) between a group of economic agents,
- a behavioral trait that allows an agent to communicate using the language.

Each abstract message type can be thought of as defining an "envelope" containing the actual content of the message that is to be exchanged between a group of economic agents. Defining envelopes containing messages is useful because it allows agents to react based on the type message received. Each abstract protocol defines a particular subset of message types that can be sent by an agent in response to a particular type of message that it has received.

Our agent communication API is influenced by, but not slave to, the Foundation for Intelligent Physical Agents (FIPA) compliant Agent Communication Language (ACL).

4. Markets API

The Markets API explicitly defines various disequilibrium dynamic processes by which market prices and quantities are determined.[4]

4.1. **Requirements.** The Markets API needs to be sufficiently flexible in order to handle markets for relatively homogeneous goods (firm non-labor inputs, firm outputs, final consumption goods, standard financial products etc.) as well as markets for relative heterogeneous goods (i.e., labor, housing, non-standard financial products, etc).

Here is my (likely incomplete) list..

- Receive buy and sell orders from other actors.
- Accept (reject) only valid (invalid) buy and sell orders.
- Handle queuing of accepted buy and sell orders as necessary.
- Order execution including price formation and, if necessary, quantity determination.
- Processing and settlement of executed orders once those orders have been filled.
- Record keeping of orders received, orders executed, transactions processed, etc.

[4] Connection to Sims' "wilderness of disequilibrium economics" quote.

Problem: too many requirements for a single market actor to satisfy. Solution: model the market actor as a collection of actors. Specifically, suppose that each MarketLike actor is composed of two additional actors: a ClearingMechanismLike actor that models the clearing process of buy and sell orders, and then a SettlementMechanismLike mechanism that processes the resulting filled orders.

4.2. **MarketLike actor.** The MarketLike actor should directly receive buy and sell orders for a particular Tradable, filter out any invalid orders, and then forward along all valid orders to a ClearingMechanismLike actor for further processing.

4.3. **ClearingMechanismLike actor.** . A ClearingMechanismLike actor should handle order execution (including price formation and quantity determination as well as any necessary queuing of buy and sell orders), generate filled orders, and send the filled orders to some SettlementMechanismLike actor for further processing. Note that each MarketLike actor should have a unique clearing mechanism.

4.3.1. *Order execution.* Order execution entails price formation and quantity determination. Market price formation requires clearing the market. It is important to be clear about the definition of the term "market clearing" [?] defines "market clearing" as follows:

(1) The process of moving to a position where the quantity supplied is equal to the quantity demanded.
(2) The assumption that economic forces always ensure the equality of supply and demand.

In most all mainstream macroeconomic models (i.e., RBC, DSGE, etc) it is assumed that economic forces instantaneously adjust to ensure the equality of supply and demand in all markets.[5]

In our API, however, a key component of a ClearingMechanismLike actor is a MatchingEngineLike behavioral trait which explicitly defines a dynamic process by which orders are executed, prices are formed, and quantities are determined. Note that a MatchingEngineLike behavioral trait is similar to an auction mechanism in many respects. Friedman (2007) lists four major types of two-sided auction mechanisms commonly implemented in real world markets.[6]

- Posted offer (PO): PO allows one side (say sellers) to commit to particular prices that are publicly posted and then allows the other side to choose quantities. PO is the dominant clearing mechanism used in the modern retail sector.

[5] I am sure that there are important examples in the mainstream economics literature where the process of market clearing is explicitly modeled and we should cite these.

[6] TODO: similarly classify the various types of single-sided auction mechanisms commonly implemented in real world markets.

- Bilateral negotiation (BLN): BLN requires each buyer to search for a seller (and vice versa); the pair then tries to negotiate a price and (if unsuccessful) resumes search. BLN clearing mechanisms were prevalent in preindustrial retail trade, and continue to be widely used in modern business-to-business (B2B) contracting. Some retail Internet sites also use BLN clearing mechanisms.
- Continuous double auction (CDA): CDA allows traders to make offers to buy and to sell and allows traders to accept offers at any time during a trading period. Variants of CDA markets prevail in modern financial exchanges such as the New York Stock Exchange (NYSE), NASDAQ, and the Chicago Board of Trade and are featured options on many B2B Internet sites.
- Call auction (CA): The CA requires participants to make simultaneous offers to buy or sell, and the offers are cleared once each trading period at a uniform price. Each of these auction mechanisms would correspond to a particular implementation of an MatchingEngine-Like behavior.

4.3.2. *Order queuing.* Order queuing involves storing and possibly sorting received buy and sell orders according to some OrderQueuingStrategy. Different order queuing strategies will be distinguished from one another by...

- type of mutable collection used for storing buy and sell orders,
- the sorting algorithm applied to the mutable collections.

For example, some OrderQueuingStrategy behaviors might only require that unfilled buy and sell orders are stored in some mutable collection (the sorting of buy and sell orders within their respective collections being irrelevant). Other OrderQueuingStrategy behaviors might have complicated OrderBookLike rules for sorting the stored buy and sell orders. Here is a quick sketch of what the code for generic OrderQueuingStrategy would look like...

4.4. **Settlement mechanisms.** Fundamental objective of a Settlement-MechanismLike actor is to convert filled orders into settled transactions. Rough sketch of a process by which filled orders are converted into settled transaction is as follows.

- Receive filled orders from some ClearingMechanismLike actor(s).
- Send request for the desired quantity of the specified Tradable to the seller.
- Send request for some desired quantity of the specified means of payment (which will be some other Tradable) to the buyer.
- Handle response from the seller (requires handling the case in which seller has insufficient quantity of the specified Tradable).
- Handle response from the buyer (requires handling the case in which buyer has insufficient quantity of the specified means of payment).
- Generate a settled transaction.

The following two types of settlement mechanisms should cover most all possible use cases.

- Bilateral settlement: with bilateral settlement, buy and sell counterparties settle directly with one another.
- Central counterparty (CCP) settlement: With CCP settlement, a central counterparty (CCP) actor inserts itself as a both a buy and sell counterparty to all filled orders that it receives from some clearing mechanism. After inserting itself as a counterparty, the CCP actor then settles the filled orders using bilaterally. Unlike clearing mechanisms, which are unique to a particular market, settlement mechanisms could be shared across markets.

5. Use cases for MarketLike actors

5.1. **Specific use cases for MarketLike actors.** In this section I sketch out some specific use cases for the Markets API.

5.1.1. *Retail goods market.* . RetailMarketLike behavior would extend MarketLike behavior with:

- Clearing mechanism with PostedOfferLike matching engine,
- BilateralSettlement settlement mechanism.

Retail goods markets are markets for final consumption goods (typically purchased by households).

5.1.2. *Wholesale goods market.* WholesaleMarketLike behavior would extend MarketLike behavior with:

- Clearing mechanism with BilateralNegotiationLike matching engine,
- BilateralSettlement settlement mechanism.

Wholesale goods markets are markets for intermediate goods (typically purchased by firms and then used in the production of retail goods).

5.1.3. *Labor market.* LaborMarketLike behavior would extend MarketLike behavior with:

- Clearing mechanism with either BilateralNegotiationLike or PostedOffer matching engines,
- BilateralSettlement settlement mechanism.

Labor markets are tricky. If we use BilateralNegotiationLike clearing mechanism then we can link into the massive search and match literature.

5.1.4. *Housing market.* HousingMarketLike behavior would extend MarketLike behavior with:

- Clearing mechanism with PostedOfferLike matching engine,
- BilateralSettlement settlement mechanism.

Note similarity of HousingMarketLike to RetailMarketLike

5.1.5. *Securities market.* . SecuritiesMarketLike behavior would extend MarketLike behavior with:

- Clearing mechanism with ContinuousDoubleAuctionLike matching engine and OrderBookLike order queuing strategy,
- CentralCounterpartySettlement settlement mechanism.

Securities markets would include markets for stocks, bonds, and currencies. Could even create a SecuritiesExchange actor which would route orders for various securities to the appropriate SecuritiesMarketLike actor.

5.1.6. *Unsecured interbank lending market.* InterbankMarketLike behavior would extend MarketLike behavior with:

- Clearing mechanism with BilateralNegotiationLike matching engine,
- BilateralSettlement settlement mechanism.

See Perry Mehrling for more details on unsecured interbank lending markets.

5.1.7. *Secured interbank lending (repo) market.* RepoMarketLike behavior would extend MarketLike behavior with:

- Clearing mechanism with BilateralNegotiationLike matching engine,
- BilateralSettlement settlement mechanism.

See Perry Mehrling for more details on secured interbank lending (repo) markets.