Planning Document for ScalABM

July 1, 2016

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. It is a suite of tools which allows users to build an agent-based model of their unique economic system with minimal development using existing plug-and-play modules. The purpose of this document is to provide both an introduction to ScalABM and to coordinate, prioritize, and plan future work on the project.

2 Motivation

Failure to predict changes to economic systems is costly. According to the Dallas Federal Reserve Bank (Luttrell et al., 2013) the recent financial crisis cost the United States on the order of 15 trillion dollars, suggesting a much larger cost for the world as a whole. These costs were significantly born by the poor, who suffered directly from the downturn in housing markets, which caused many to default.

In recent history, two main techniques have emerged in attempting to predict economic behaviour: a.) Dynamic Stochastic General Equilibrium (DSGE) models, and b.) Agent-based Models (ABM). DSGE models have made significant progress in assessing the microfoundations guiding macro-level behaviour (Smets and Wouters, 2003; An and Schorfheide, 2007). Such models offer strong, quantitative economic forecasts grounded by calibration and validation against historic data. However, DSGE models have limitations. They are constructed under the presumption that human behaviour is homogeneous and that economic systems move toward an equilibrium (Farmer and Foley, 2009). This rests in stark contrast to the heterogeneous, analytically intractable human behaviour which underlies large-scale economic systems.

ABMs overcome several DSGE obstacles but possess limitations of their own. ABMs allow heterogenous agent behaviour, permitting the researcher to consider the effects of complex micro-level interactions on macro-level behaviour. As such, ABMs have had several successes (Klimek et al., 2015; Poledna et al., 2014; Deissenberg et al., 2008; Schelling, 1971). However, whilst ABMs offer heterogeneous behaviour for free, much of the research currently conducted is of a qualitative nature, lacking the calibration and validation required to ground their results. Further, progress in building 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 consequence is that, even when ABMs offer quantitative results, it is difficult to replicate or compare competing models.

In juxtaposing the two methodologies, we believe it is important to note that large-scale micro panel data-sets are becoming increasingly available. Presuming this trend continues, we believe that heterogeneous agent-based models are better placed to leverage such data. However, to exploit this recent deluge of data, agent-based models need to overcome the aforementioned weaknesses. They much strive to offer the microfounded quantitative insight of DSGE models.

3 Solution: ScalABM

ScalABM is a tool set for building agent-based economic models, which attempts to solve several of the limitations of current ABM models whilst integrating the benefits of DSGE models. The intention of ScalABM is to provide a collaborative application where users in a variety of economic disciplines can model and analyse agent-based models. ScalABM combines "Big Data" techniques, particularly machine learning methods capable of taking advantage of large-scale micro panel data sets, with structural economic modeling. Further, ScalABM offers replicable analysis by presenting a common, modular platform for using ABMs to scrutinize a variety of economic systems.

To the user, the system will be accessed through a web-application, permitting the user to a.) import data, b.) set up their economic components using ScalABMs plug and play markets and institutions, c.) use machine learning techniques to calibrate their model to fit historic data, and d.) analyse the dynamics of their model over time.

ScalABM is:

- *Data Driven, Calibrated and Validated:* ScalABM manages the flow and storage of both model generated data as well as real-world data. The user's models will be grounded by their empirical data. Calibration and validation techniques will permit the user to adjust their model to fall in line with historical data whilst analytic tools will provide the user data on the state of the model during run-time.

- *Versatile:* With multiple entry points, ScalABM offers users their choice in the granularity of control required for their research. Non-programmers may instantiate plug-and-play markets, banks, and individuals via a web interface. Analysis of agent interactions can be conducted without any programming on the end-user side. If the user requires more granular control of their system, they may implement their own types of markets, banks, etc. By developing classes which meet the ScalABM's specifications, a user can develop the system to their own specification.

- *Open-Source:* 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).

- *Modular:* 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 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.

- *Scalable:* System size is a key parameter for modeling economic systems and in order to accurately model the dynamics of some systems, users may need to build and simulate models that are as close to observed scale as possible. ScalABM outsources cluster management to third party providers. 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.

- *Reproducible:* ScalABM is not dependent on access to university or national supercomputers. This enhances the reproducibility research. The ability to "containerize" an ABM built using ScalABM's 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.

4 Deliverables

Such a large project requires the coordination of many moving parts. This document makes a preliminary attempt at diagnosing all the high-level tasks in the hopes of planning and prioritizing them. This project can be broken down into several high-level tasks, broadly classified under "Tools for Analysing Economic Systems" and "Software Architecture and Quality Infrastructure".

Tools for Analysing Economic Systems:

Tasks under this classification represent the agent-based economic system and the tools and components used to manipulate the economic system. These tasks represent the components the user will need to access whilst conducting research.

- **Economic System Interface**: What are the requirements (i.e. interfaces) for markets, contracts, and individual agents? What are the possible actions for each type of agent?

- **Agent Behaviour**: How does an individual agent learn, process information, and act?
- **Communication Layer**: How do the above systems interact and communicate?
- **Calibration**: What tools are needed to calibrate a model to historical data?
- **Validation**: What tools are required to validate predictions? Can null-hypotheses be instantiated in a plug and play manner?

Software Architecture and Quality Infrastructure:

These tasks represent the underlying infrastructure of ScalABM, as well as the controls put in place to maintain quality assurance.

- **High-level Software Architecture**: How is the UI separated from the model? What languages and databases are used?
- **Data processing**: How is data imported, processed, and accessed?
- **User Interface**: How will an end-user interact with the application?
- **Open-Source Contributions**: How are open-source contributions managed and requested?
- **Website**: What is the best way to advertise ScalABM? How should open-source contributions be linked to the site?
- **Quality Management**: How are internal and open-source contributions reviewed and quality maintained?

Below, each task is reviewed in turn. Particular focus is made toward enumerating the requirements and deliverables of each sub-task. This document will be extended as requirements are fleshed out.

5 Tools for Analysing Economic Systems

5.1 Economic Systems Interface

The intention of ScalABM is to provide multiple and flexible entry points for analysing a variety of economic systems. To do this, the user must have flexibility in the creation of the components of their economic system. For example, the nuances of a market must be altered to meet each user's needs. However, where possible, ScalABM intends to permit generic, plug-and-play components for users who do not require additional complexity.

To strike this balance, ScalABM offers an Interface for components of economic systems. These define the minimum requirements for each component. Once the APIs are finalized, ScalABM will offer simple concrete instantiations as plug and play components. Additional components will also be available via open-source contributions.

The interface for economic components consists in the following atomic components. The specifications of each of these components are currently in their design phase and will be extended shortly.

- *Actor*: Each `Actor` has a `Balance Sheet` and the ability to act on `Promises`. `Actors` are banks, corporations, and individuals. They are the atomic acting agent underlying ScalABM.

- *Balance sheets*: A list of the assets and liabilities of each `Actor`. Assets and liabilities consist of `Goods` and `Promises`.

- *Goods*: `Goods` represent physical objects which possess some value.

- *Promises*: A `Promise` represents a commitment between two (or more) `Actors` to undertake certain actions, potentially involving both `Goods` and additional `Promises`, specified by some `Sentence` when a certain `StateofAffairs` has occurred. For a more complete definition of the `Promise` API, please see Appendix A.

- *Markets*: A `Market` provides some infrastructure for `Actors` to exchange `Goods` and `Promises`. The working low-level version of the `Market` API can be found in Appendix B.

- *Sentences*: A `Sentence` is the logical language of promises. They not only define what assets to transfer, but also explain under what circumstances the trade will go forward. No API currently exists, but the preliminary thought is to use a contract language as described in Bahr et al. (2015), Bahr et al. (2014), and Andersen et al. (2006).

5.1.1 Plug and Play Components

Once the design of the interface has stabilized, ScalABM will generate a group of concrete instantiations of the interfaces which can be used as plug-and-play markets, promises, etc. A list, however, should be constructed prior to the design of the interfaces so that the developers can keep use-cases in mind when designing the interface. Additionally, open-source contributions of additional extensions to the interface will be made available online (see Section 6.6)

5.2 Agent Behaviour

An economy is populated with a variety of agents (i.e., consumers, producers, financiers, government, etc). ScalABM's 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 and 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

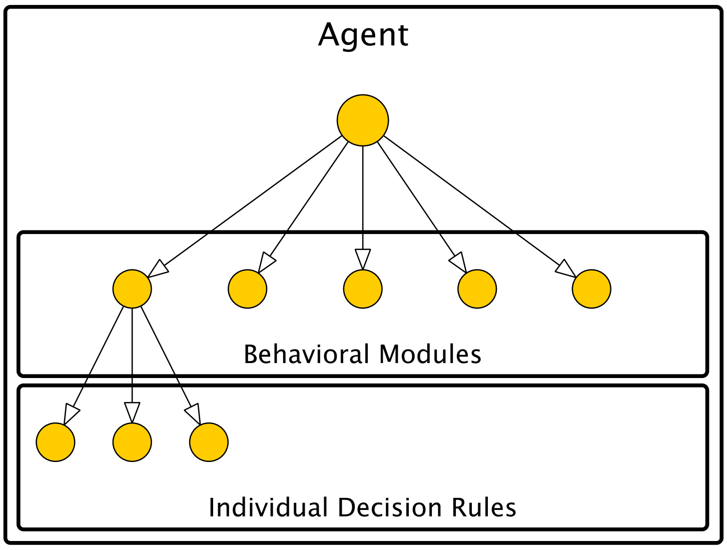

Figure 1: An agent in our framework is a layered collection of behaviors and decision rules.

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 (e.g., reinforcement learning) and learning through observation (e.g., 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).

Figure 1 depicts a high-level description of an agent as a layered collection of behavior modules and decision rules. An example of this layered collection would be an agent who has a utility maximization behavior module and, consequently, several decision-rules comprising how to act in order to maximize its utility.

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.

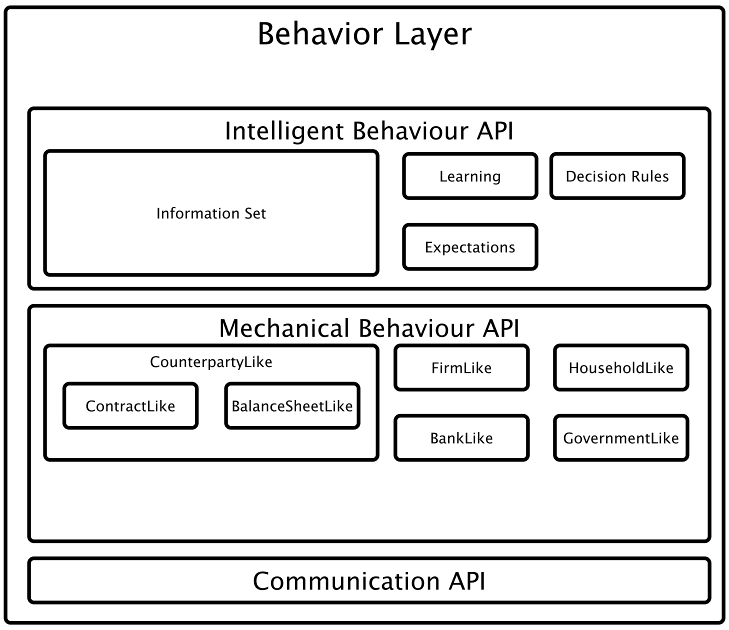

Figure 2: Behavioural layer: Incorporates all Actor behaviour with inter Actor communication.

5.3 Communication Layer

Sections 5.1 and 5.2 define the components of an economic system, but not how these components interact. 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), send more messages, determine how to respond to the next message received, etc.

Figure 2 illustrates how this communication layer resides in the behavioural 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).

Finally, 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.

5.4 Calibration

If a user is attempting to understand an existing economic system, they will need to calibrate the interaction of their economic components such that the entire system fits data as well as possible and generalizes out of sample. Example calibration questions include:

- Historically, how do individuals process information to make decisions?
- How does the scarcity of resources affect market dynamics?
- What variables (or non-linear combination of variables) provide the most predictive force when compared to historic data?

The intention of the calibration tools is to allow 1.) linking dataset variables to model parameters, 2.) feature selection, and 3.) predictive analysis of how economic components interact.

5.4.1 Associating historic variables to model parameters:

At the simplest level, calibration involves syncing some variable in the imported, historic data with a variable in the model. For example, the temperature in a dataset may need to be linked to a temperature model parameter. ScalABM should provide some user-friendly mechanism for associating values in the dataset to parameters in the model. Though the most simple 1:1 example is given, thought should go into a user-friendly design for more complicated cases. As mentioned in Section 5.3, this will hopefully be accomplished via a configuration file — though the details still need to be discussed.

5.4.2 Feature Selection

In an attempt to simplify their model, users may seek to diagnose the parameters which offer the most explanatory historic effect. Discussion needs to occur, regarding to what level ScalABM will offer tools to support this operation.

There are a variety of techniques available for feature selection, and often these solutions will simultaneously perform predictive analysis:

- Genetic Algorithms
- Random Forests
- etc...

Future discussion should prioritize algorithms which are better at uncovering non-linear interactions.

5.4.3 Prediction Analysis

Prior to attempting to uncover experimental effects, the model should be calibrated to historic data. If, at some starting point t_0, the model is able to predict the historic flow of the economic system, then there is a greater likelihood that the model can accurately predict the movement of the system in novel state space (i.e. the future).

Again, discussion is required to define to what extent ScalABM will provide predictive analysis tools. Potential tools include, but are not limited to (for a nice comparison of several techniques, see Bajari et al., 2015):

- Neural Networks
- Random Forests
- Linear Regression
- Support Vector Machines
- Bagging
- Kalman Filters (Ward et al., 2016)

Further, research at INET is currently attempting to scaffold from calibration techniques in meteorology. Depending on the success of this project, such tools may be incorporated into ScalABM.

5.5 Validation

Validation tools will likely be integrated into the calibration tool set. However, conceptually it is worth discussing them separately. Validation tools work in coordination with calibration tools to:

- Generalize the predictive capacity of the model.
- Increase the likelihood that the model will accurately predict market behaviour in novel state space.

Validation aids in avoiding overfitting, whereby the model is overly calibrated to historic data. Since most real-world behaviour involves a noisy process, perfectly mapping one's model to historic data attempts to capture the concept of noise as a predictive variable. This leads to poor predictions in novel state

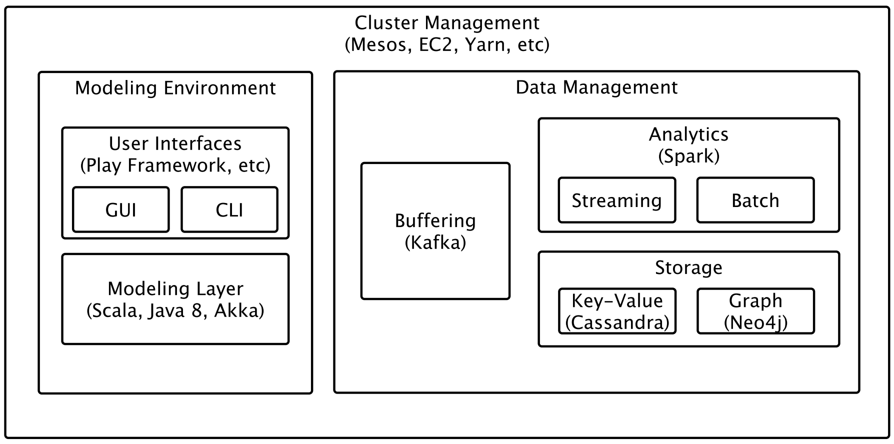

Figure 3: High-level architecture design for ScalABM.

spaces. Validation techniques provide a mechanism for generalizing the predictive function, (hopefully) uncovering the more fundamental causative dynamics of the system. This will likely come in the form of cross-validation. By testing one's model on data the model have never consumed, the predictive capacity of the model in novel state spaces is increased.

As in the case of calibration techniques, future discussions should focus on whether ScalABM will integrate validation tools in their system.

6 Software Architecture and Quality Infrastructure

6.1 High-Level Software Architecture

The high-level design of ScalABM's platform architecture should mimic the layered architecture of a reactive, "Fast Data" web application. Figure 3 summarizes the high-level design of the platform architecture.[1]

6.1.1 Cluster management

ScalABM attempts to provide increases in processing speed through parallelism and 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.

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

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 this approach include:

- Outsourcing of cluster management to third party providers. ABMs built using ScalABM 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 can then handle all of the intricacies involved with scaling up the model on the cluster to meet the users needs.
- Not being dependent on access to university/national super-computers. This enhances the reproducibility or our research. The ability to "containerize" an ABM built using the ScalABM 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.

6.1.2 Modeling environment

The modeling environment consists of user interfaces and the modeling layer. The "back end" of the modeling environment is the modeling layer which consists of the actual source code libraries used to implement our ABMs (see Sections 5.1 and 5.2 for more detail). Important characteristics of the modeling layer are that:

- 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 4). The Data Analytics layer represents the methods for storing and processing data, whether imported into the system or created by the system (see Section 6.3). The information layer acts as the middle-man between the Data Analytics layer and the Behavioral layer. Data which has been processed in the Data Analytics layer can be passed down as observables to the agents. On the other hand, agent and system behavior is passed through the Information layer to be stored and processed in the Data Analytics layer. The Behavior layer represents agent and system behavior. As agents in the system are fed observables by the Information layer, their behaviors are fed back to the Information layer, which can act as observables for future behaviors.

6.2 User Interface

The "front end" of the modeling environment consists of two, complementary user interfaces.

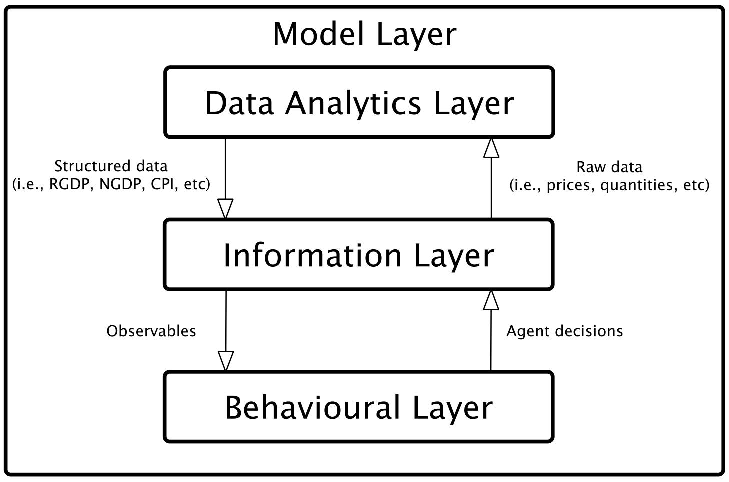

Figure 4: High-level organization of the modeling layer.

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 (e.g., parameter sweeps), the CLI should allow for easy replication of any particular model simulation.

The designs of both UIs require extensive discussion. Even before design discussions can occur, preliminary questions include:

- Who is the intended audience for the GUI and CLI? i.e. How simplistic must the UI be?
- What is the priority of the UI compared to the back-end modelling layer?
- Will the primary contributors of the UI be from INET or from the open source community?

6.3 Data Processing

In order for our ABMs to be data-driven, we need to think carefully about how our framework 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, one 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. The data analytics components should therefore include tools for processing and analyzing streaming data. In addition to processing and analyzing streaming data, there is also a 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. The 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.

6.3.1 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.

6.4 Website

The website will act as the front-face of the application. As such its role will change as ScalABM advances.

6.4.1 First Phase

At the planning stage of the product, the website will advertise ScalABM to potential:

- *collaborators/investors*: The website should contain enough information to offer individuals a moderate understanding of the details of the project. Contact information should be available for deeper inquiries. Information should be presented with a bias toward potential collaboration.

- *open-source contributors*: Whilst the details of open-source contribution are being worked out, the website should advertise the intention of open-source contributions. If a potential contributor wishes to subscribe to a mailing list, this should be made available.

6.4.2 Mid-Life

During the mid-life of ScalABM the website will focus on:

- offering details and documentation of the product.
- requesting open-source contributions.
- where possible, offering prototypes for early adopters.
- continuing to support collaborators/investors.

6.4.3 Project Maturity

When ScalABM is mature, the website will:

- advertise the use of the tool as an out-of-the-box solution for researching economic systems through ABMs.
- describe how to use the product
- offer online forums and discussions
- continue to advertise how to contribute to extending the tool via open-source contributions.

6.4.4 Planning

From a planning perspective, the initial version of the website should be prioritized. Connecting with potential investors, collaborators, and contributors is vital for the long-term success of the project. Currently design is an open question which should be considered.

6.5 Quality Management

Quality control will need to consider internal management, as well as controlling external, open-source contributions.

6.5.1 Internal Quality Management

Development within INET will be quality controlled via a mixture of unit test cases, code reviews, and the encapsulation garnered from feature branch workflow and pull requests.

- *Unit Test Cases*: Code will be subject to continuous unit testing. Validation of test case code coverage will be automated (e.g. Coveralls). Further, during code reviews, peers will validate that outlying use cases are considered within the test case suite.

- *Code Reviews*: Part of a developer's responsibility is to review the code of their peers. These code review sessions will be scheduled at regular intervals and following pull requests.
- *Feature Branch Workflow and Peer Requests*: In an attempt at encapsulation and decoupling, development will be compartmentalized into feature branches. Each feature of the application will be assigned a lead developer. This developer will act as a gatekeeper in maintaining the integrity and quality of that feature. Prior to integrating a feature into the main application (i.e. master branch), a pull request will be required. A peer review will take place prior to integration.

6.5.2 Open Source Management

The vetting and implementation of open-source contributions will play a significant part in quality management. This document discusses the planning of open-source contributions, including their quality management, as a whole in the next section.

6.6 Open-Source Contributions

Several decisions must be made in order to manage the quality of open-source contributions. At a high-level, the contribution workflow needs to be defined. This flow will need to answer several open questions, including:

- How are potential contributors informed of ScalABM's needs?
- How are contributions vetted?
- When and how often will contributions be assimilated into a new build?
- Are different types of contributions vetted differently? For instance, if a contributor extends the Market interface to meet their own needs, does INET provide a repository where individuals can share their work with the community?

More fundamentally a discussion needs to address:

- When should open source contributions be requested?
- Are contributions going to be (at least initially) focused in a particular area of the project (e.g. the GUI)?

7 Prioritization

This section should contain the prioritization of the aforementioned tasks. What should happen when and what can happen in tandem? In the current, preliminary stage, perhaps the website and the design of the back-end Economic Systems Interface should be prioritized? A lot of the UI design will relate to how the back-end is implemented, and since the back-end design will invariably require some iteration, it may be a good idea to hold off on the front-end.

8 Project Roles and Responsibilities

This section will contain the points of contact for each of the high-level tasks once they are assigned.

9 Schedule

This section will contain a schedule for the high-level tasks once it is made available.

A Promises API

A.1 Definition of a `Promise`

A `Promise` represents a commitment between a `promisor` and some `promisee` to undertake certain `actions`, potentially involving both `goods` and additional `promises`, specified by some `Sentence` `when` a certain `StateofAffairs` has occurred.

These considerations lead to the following low-level interface for the `Promise` trait.

```
trait Promise {

  def promisor: ActorRef

  def promisee: ActorRef

  def actions: Sentence

  def when: StateOfAffairs

}
```

Want to allow for the possibility that some promises can not be given (and therefore not transferred or exchanged) from one `Actor` to another `Actor`. To capture this feature we define a separate `GiveablePromise` trait.

```
trait GiveablePromise extends Promise {

  val isGiveable: Boolean

}
```

A.2 `Actions` over `Promises`

A.2.1 Uni-lateral `Actions`

An `Actor` i can *unilaterally* decide to perform any of the following `Actions` with a `Promise`.

- `create` a new `Promise`: When an `Actor` creates a new `Promise`, the `Actor` becomes its `promisor` and the new `Promise` becomes a liability for that `Actor`. Similarly, the new `Promise` becomes an asset for which ever `Actor` is the `promisee`.
- `accept` a `Promise`: If an `Actor` i chooses to `accept` a `Promise` from another `Actor` j, then the `Promise` is added as an asset to the balance sheet of `Actor` i and as a liability to the balance sheet of `Actor` j.
- `reject` a `Promise`: An `Actor` i can always choose to `reject` (i.e., *not* `accept`) a `Promise` from another `Actor` j. Rejected promises are not added to balance sheets.
- `fulfill` a `Promise`: An `Actor` i who is the `promisor` of a `Promise` may choose to perform `actions` specified in the `Sentence` `when` a certain `StateOfAffairs` has occurred. Once a `Promise` is successfully fulfilled, it is removed from both the balance sheet of its `promisor` and the balance sheet of its `promisee`.
- `break` a `Promise`: An `Actor` i who is the `promisor` of a `Promise` may choose *not* to perform `actions` specified in the `Sentence` `when` a certain `StateOfAffairs` has occurred. A decision to `break` a `Promise` is the same as a decision *not* to `fulfill` a `Promise`.[2]
- `destroy` a `Promise`: An `Actor` i who is the `promisee` of a `Promise` may choose to `destroy` that `Promise` prior to a certain `StateOfAffairs` occurring. Once destroyed a `Promise` is removed from both the balance sheet of the `promisee` *and* the balance sheet of the `promisor`.
- `give` a `Promise`: An `Actor` i who is the `promisee` of a `Promise` may choose to `give` that `Promise` to another `Actor` j.
- `redeem` a `Promise`: An `Actor` i who is the `promisee` of a `Promise` may choose to `redeem` that `Promise` after a certain `StateOfAffairs` has occurred. Redemption of a `Promise` can be thought of as a request that the `promisor` of that `Promise` `fulfill` the `Promise`.

The above actions can be combined into a low-level, `PromiseMaker` trait that extends the `Actor` base trait:

```
trait PromiseMaker extends Actor {

  def create(promisee: ActorRef,
             actions: Sentence,
             when: StateOfAffairs): Promise

  def destroy(promise: Promise): Unit

  def accept(promise: Promise): Unit

  def reject(promise: Promise): Unit
```

[2] Breaking a `Promise` may or may not have consequences. Should consequences be specified in the original `Promise`? I think so.

```
  def fulfill(promise: Promise): Unit

  def break(promise: Promise): Unit

  def give(promise: Promise, other: ActorRef): Unit

  def redeem(promise: Promise): Unit

}
```

A.2.2 Cooperative `Actions`

Two `Actors` i and j can cooperate *bi-laterally* to perform additional actions over `Promises`:

- `transfer` a `Promise`: An `Actor` i who is the `promisee` of the `Promise` can `transfer` a `Promise` to `Actor` j as follows:
  1. `Actor` i `give` `Promise` to `Actor` j
  2. `Actor` j `accept` `Promise` from `Actor` i
- `exchange` of `Promises`: Two `Actors` i and j who are the `promisees` of different `Promises` can `exchange` these `Promises` with one another as follows:
  1. `Actor` j `create` new `Promise` {`give` existing `Promise` to `Actor` i}.
  2. `Actor` j `transfer` new `Promise` to `Actor` i.
  3. `Actor` i `transfer` existing `Promise` to `Actor` j
  4. `Actor` j `fulfill` `Promise` to `Actor` i

A few things are worth noting about a bi-lateral `exchange`. First, by choosing to `fulfill` the new `Promise` in step 4, `Actor` j gives his existing `Promise` to `Actor` i (which completes the `exchange`). Second, the new `Promise` issued by `Actor` j in step 1 involved a promise to `give` and *not* a promise to `transfer` an existing `Promise` to `Actor` i.[3] Finally, note that an `exchange` could take place with `Actor` i creating the new `Promise` in step 1. The important point is that either `Actor` i or `Actor` j (not necessarily both) must be able to *credibly* commit to `give` its `Promise` upon receipt of the other's `Promise`.[4]

It is interesting to compare the above bi-lateral exchange mechanism with a multi-lateral exchange mechanism involving cooperation between three `Actors` i, j, and k. Two `Actors` i, j who are the `promisees` of different `Promises` can `exchange` these `Promises` using `Actor` k as an intermediary as follows:

1. `Actor` k `create` new `Promise` {`give` `Actor` j `Promise` to `Actor` i}.
2. `Actor` k `create` new `Promise` {`give` `Actor` i `Promise` to `Actor` j}.

[3] Should it be possible to for an `Actor` to create a `Promise` that commits *other* `Actors` to perform actions? Do we have any real world examples?.

[4] The credibility of any particular `Promise` should be endogenously determined within the model and *not* imposed by us *a priori*

3. `Actor` k `transfer` new `Promise` to `Actor` i.
4. `Actor` k `transfer` new `Promise` to `Actor` j.
5. `Actor` i `transfer` existing `Promise` to `Actor` k
6. `Actor` j `transfer` existing `Promise` to `Actor` k
7. `Actor` k `fulfill` `Promise` to `Actor` i
8. `Actor` k `fulfill` `Promise` to `Actor` j

An important feature of this multi-lateral process is that, so long as `Actor` k can *credibly* commit to *both* `Actors` i *and* j, then the exchange between i and j can take place even if *neither* `Actor` i *nor* `Actor` j can bi-laterally commit to `give` its `Promise` upon receipt of the other's `Promise`.[5] There are several interpretations of `Actor` k's role in the above process. One interpretation is that `Actor` k is functioning as a central clearing party (CCP) for transactions between other `Actors`; another more institutional interpretation is that `Actor` k is an actual `Market`.

One final cooperative action needs to be specified: `transfer` of a `Promise` that is liability for one `Actor` to some other `Actor`. An `Actor` i who is the `promisor` on a `Promise` can only `transfer` that `Promise` to another `Actor` j with permission from the `promisee` of that `Promise`, `Actor` k.

1. `Actor` i `create` new `Promise` {`give` `Actor` j existing `Promise` }.
2. `Actor` k `accept` new `Promise`.
3. `Actor` i `fulfill` `Promise` to `Actor` k.
4. `Actor` j `accept` `Promise` from `Actor` i.

A.3 A language for `Promises`

Having defined the concepts of a `Good` and a `Promise` as well as sets of `actions` over `Goods` and `Promises` that can be performed by an `Actor` or groups of `Actors` to complete the API we need to define a language (grammar?) for building `Sentences` that describe valid `Promises`.

A.3.1 Examples

Need to build a catalogue of examples demonstrating how to build common contracts using our language.

B Markets API

The Markets API explicitly defines various disequilibrium dynamic processes by which market prices and quantities are determined.[6]

[5] The difference between multi-lateral and bi-lateral commitment has been stressed by many monetary theorists, in particular Kiyotaki and Moore in a series of papers.

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

B.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.

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.

B.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.

B.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.

B.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.[7]

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.[8]

- 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.

- 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 MatchingEngineLike behavior.

B.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

[7] 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.

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

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...

B.4 Settlement mechanisms

Fundamental objective of a SettlementMechanismLike 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.

B.5 Specific use cases for MarketLike actors

In this section I sketch out some specific use cases for the Markets API.

B.5.1 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).

B.5.2 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).

B.5.3 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.

B.5.4 Housing market

HousingMarketLike behavior would extend MarketLike behavior with:

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

Note similarity of HousingMarketLike to RetailMarketLike

B.5.5 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.

B.5.6 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.

B.5.7 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.

References

An, S., Schorfheide, F., 2007. Bayesian analysis of dsge models. Econometric Reviews 26 (2-4), 113–172.

Andersen, J., Elsborg, E., Henglein, F., Simonsen, J. G., Stefansen, C., 2006. Compositional specification of commercial contracts. International Journal on Software Tools for Technology Transfer 8 (6), 485–516.
URL http://dx.doi.org/10.1007/s10009-006-0010-1

Bahr, P., Berthold, J., Elsman, M., 2014. Towards certified management of financial contracts. In: In Proc. of 26th Nordic Workshop on Programming Theory (NWPT).

Bahr, P., Berthold, J., Elsman, M., Aug. 2015. Certified symbolic management of financial multi-party contracts. SIGPLAN Not. 50 (9), 315–327.

Bajari, P., Nekipelov, D., Ryan, S. P., Yang, M., 2015. Machine learning methods for demand estimation. The American Economic Review 105 (5), 481–485.

Deissenberg, C., van der Hoog, S., Dawid, H., 2008. Eurace: A massively parallel agent-based model of the european economy. Applied Mathematics and Computation 204 (2), 541 – 552, special Issue on New Approaches in Dynamic Optimization to Assessment of Economic and Environmental Systems.
URL http://www.sciencedirect.com/science/article/pii/S0096300308003019

Farmer, J. D., Foley, D., 2009. The economy needs agent-based modelling. Nature 460, 685–686.
URL http://dx.doi.org/10.1038/460685a

Klimek, P., Poledna, S., Farmer, J. D., Thurner, S., 2015. To bail-out or to bail-in? answers from an agent-based model. Journal of Economic Dynamics and Control 50, 144 – 154, crises and ComplexityComplexity Research Initiative for Systemic InstabilitieS (CRISIS) Workshop 2013.
URL http://www.sciencedirect.com/science/article/pii/S0165188914002097

Luttrell, D., Atkinson, T., Rosenblum, H., 2013. Assessing the costs and consequences of the 2007?09 financial crisis and its aftermath. Economic Letter 8 (sep).
URL https://ideas.repec.org/a/fip/feddel/y2013isepnv.8no.7.html

Poledna, S., Thurner, S., Farmer, J. D., Geanakoplos, J., 2014. Leverage-induced systemic risk under basle ii and other credit risk policies. Journal of Banking & Finance 42, 199 – 212.
URL http://www.sciencedirect.com/science/article/pii/S0378426614000521

Schelling, T. C., 1971. Dynamic models of segregation. The Journal of Mathematical Sociology 1 (2), 143–186.

Smets, F., Wouters, R., 2003. An estimated dynamic stochastic general equilibrium model of the euro area. Journal of the European Economic Association 1 (5), 1123 – 1175.
URL http://search.ebscohost.com.ezproxy1.bath.ac.uk/login.aspx?direct=true&db=bth&AN=12285012&site=ehost-live

Ward, J. A., Evans, A. J., Malleson, N. S., 2016. Dynamic calibration of agent-based models using data assimilation. Open Science 3 (4).
URL http://rsos.royalsocietypublishing.org/content/3/4/150703