Object Oriented Programming and AB-SFC modelling

Alessandro Caiani[1]

Department of Economics and Social Sciences
(Università Politecnica delle Marche)
Ancona, Italy

May 7, 2015

[1] a.caiani@univpm.it

Outline

Outline

4 purposes of AB modeling (I)

Tesfatsion and Judd [2006] differentiates ABMs by their objectives:

1)Empirical understanding

Why some empirical regularities are persistent and continuously evolving in time despite the lack of a social planner?
These models try to identify the causal links rooted in the repeated interactions of agents operating in a realistic environment.
The objective is to assess through computational experiments under which conditions agents interaction gives rise to the empirical regularities observed. (generativist approach, [Epstein and Axtell, 1996])

2)Normative understanding

AB models used as a laboratory to identify and test good social designs.
The aim is to assess whether certain economic policies, social/institutional plans will result in a socially desirable systemic performance.
In which measure the worlds emerging from these experiments are efficient, fair, orderly?

4 purposes of AB modeling (II)

3) Qualitative insight and theory generation

How can we better understand economic system through the systematic investigation of their potential behaviors under different specifications of initial conditions? Why certain regularities and not others?

4) Methodological Advancement

Every modeler must formulate theoretical propositions over the model, assess the validity of these propositions through expressively designed experiments, extrapolate and reporting the information gathered through these experiments in a clear and convincing way.
Finally he has to test the data obtained from the simulations by comparing them with empirical data.

Still a huge variety of ways to conceive, develop computer programs, validate, and present the results.

The crucial role of Stylized Facts

Roughly, the logic of macroeconomic AB modeling consists of two pillars.

i First, empirical laws at a macroeconomic level should be expressed in terms of statistical distributions, such as the distribution of people according to their income or wealth, or the distribution of firms according to their size or growth rate [Steindl, 1965].

ii Second, suitable modeling strategies should be adopted, that is explanatory methodologies capable to combine a proper analysis of the behavioral characteristics of individual agents and the aggregate properties of social and economic structures [Sunder, 2006].

Hence the role of stylized facts in thinking an ABM is twofold:

- Micro stylized facts provide a guide to define the characteristics of the environment in which economic actors operate and their behavioral rules;
- Then, an ABM model aims to explain how the disperse interaction of micro agents *generate* macro stylized facts as emergent properties of the system.

Some examples of stylized facts I

Some typical macro stylized facts:

- Since Gibrat (1931) the size distribution of firms has proven to be right skewed, with upper-tails made of few large firms, for several different countries and historical periods [De Wit, 2005]. These patterns vary significantly across industries [Bottazzi and Secchi, 2003a,b].
- The distribution of firms' growth rates appears to be approximated by a Laplace (double-exponential) distribution [Amaral et al., 1997, Bottazzi et al., 2002], that is a fat-tailed, tent-shaped distribution.
- Earnings, income and wealth are well known to be highly concentrated over households, regardless of the measure of concentration.
- Pro-cyclicality of firms' debt and bankruptcy rates (power law distributed).
- Power-law distributed firm-level 'bad debt'.

Delli Gatti et al. [2008] provides an extensive treatment of the previous SF, also analyzed conditionally to business cycles.

Some examples of stylized facts II

Dosi et al. [2005] provides several stylized facts on business cycles:

Macro SF

1. Investment is considerably more volatile than output. Consumption is less volatile than output.
2. Investment, consumption and changes in inventories tend to be pro-cyclical and coincident variables. Inflation pro-cyclical and lagging, mark-ups countercyclical.
3. Aggregate employment unemployment rates tend to be lagging variables. The former is pro-cyclical, whereas the latter is anti-cyclical.

Micro SF

1. Investment is lumpy (not smoothed, instead concentrated in certain periods). Investment is influenced by firms' financial structures.
2. Technological learning is firm-specific and local (it takes place in a neighborhood of technology already mastered). Big jumps are rare.
3. Innovation takes time to diffuse and it's 'industry specific'.
4. Productivity dispersion among firms is high and inter-firm differentials are persistent.

Constructive understanding

Tesfatsion and Judd [2006] proposes a 4 step practical procedure to think about the structure of an ABM model.

- Select as your benchmark case an equilibrium modeling from economic literature that addresses some issue you want to study;
- Remove from this economic model every assumption that entails the external imposition of an equilibrium condition (e.g. market clearing conditions, correct expectations, etc.);
- Dynamically complete the model by the introduction of production, pricing, and trade processes driven solely by interactions among agents.
- Define an "equilibrium" for the resulting dynamically complete economic model.

Outline

Agent Based Models

ABM

Agent Based Models (ABM) in Economics conceive the economic system as a *complex adaptive system* composed of *heterogeneous*, *adaptive* economic actors intertwined through an *evolving network* of *local interactions* taking place within a *well-structured space*.
These interactions concur in shaping the *emergent properties* of the system. Continuous feedbacks between micro and macro levels.

ABM thus aim at providing an alternative way to link coherently the micro-meso and macro levels capable of overcoming the fallacy of composition implicit in the reductionist microfoundation approach, while maintaining tractability.

Economic systems reconstructed *in vitro*: in Philosophy of Science this is called a *"'generativist approach"'* to science [Epstein and Axtell, 1996].

SFC modeling

Thanks to the rigorous accounting rules underlying the construction of the accounting matrices SFC models provide a **complete, integrated, and coherent picture of the real and financial sides of an economic system** which allows to address fundamental questions such as:

- What form does personal saving take?
- Where does any excess of sectoral income over expenditure actually go to?
- Which sector provides the counterparty to every transaction in assets?
- Where does the finance for investment come from?
- How are budget deficits financed?

Avoid black boxes in describing stocks and flows dynamics, and real vs monetary variables.

Drawbacks of SFC modeling

The adoption of a pure aggregate perspective generates several problems

- Difficult to analyze heterogeneity (emerging) within sectors. Behavioral rules are specified only at sector level.
- Difficult to track intra-sectoral flows.
- Difficult to model asynchronous decision in consumption, investment and other behaviors.
- Stability/equilibrium at aggregate and sectoral level do not imply in any way that all agents are in equilibrium.
- Not possible to analyze economic networks: particularly important when network topology is likely to affect some diffusion process (e.g. of innovations, of shocks, of beliefs and financial conventions etc.).

The adoption of the bottom-up perspective of ABM would help to overcome this limits.

AB-SFC framework

- Consistent and coherent framework without blackholes
  - Both demand and supply for all markets
  - Both asset and liabilities of Balance Sheets
- Full representation of financial side of the economy
- Adds a layer of complexity by explicitly modelling the dynamic network of balance sheets
  - Direct local Interaction
  - Heterogeneity

Outline

Challenges

Still only few examples of truly AB-SFC models: Kinsella et al. [2011], Seppecher [2012], Riccetti et al. [2014], and Eurace Project. High heterogeneity of topics and solutions to ensure SFC in AB models.
This implies:

- narrow applicability of these solutions outside the specific models for which they were conceived;
- difficult to compare these models and check their actual consistency (SFC often treated implicitly).

Till now, a number of parallel paths pursuing specific solutions to common problems, rather than a common set of concepts, rules and tools, that is a new paradigm.
In particular, we suffer the lack of well-suited computational techniques to handle the heterogeneity of stocks characterizing the economy and to track their variations across time.

Limits of existing tools

Most AB macroeconomic models developed in Matlab, C, Java, R, Python using matrix algebra.

The economy modeled as a set of matrices representing either network relationship (e.g deposits network linking banks to households) or agent's characteristics (where rows are agents and columns some of their attributes).

These matrices are then update period by period, according to behavioral equations expressed in the form of matrix equations.

Pro: very efficient in terms of memory allocation and speed.
But...

Limits of existing tools

This "matrix approach" severely constrains the construction of the model regarding timing/frequencies and the heterogeneity of real and financial stocks:

- Difficult to consider stocks with different maturities, durations,;
- Difficult to model asynchronous decisions, different length of economic processes and different frequencies.

$\rightarrow$ Simplifying assumptions to avoid an exponential multiplication of matrices required to handle these features.

Our contribution aims at providing a **general, flexible, and coherent set of rules and tools to develop AB-SFC models**, avoiding these criticisms.

A Java Macroeconomic Agent Based simulator: JMAB (Caiani, Caverzasi, and Godin).

JMAB is a **general, flexible and highly modular tool-suite designed to develop AB-SFC macroeconomic models**.
It is **based on** a pre-existent platform called **JABM** [Phelps and Musial-Gabrys, 2012]
JABM was designed to run simulations on a financial market: JASA (Java Auction Simulator API) is a high-performance auction simulator that allows researchers in agent-based computational economics to write high-performance trading simulations using a number of different auction protocols. The software also provides base classes for implementing adaptive trading agents.
Implements various experiments, agents and learning algorithms described elsewhere in the computational economics literature.

Economic modeling with JABM

JABM and JASA were designed to run simulations on a single market with two counterparts.
Macro models simulations instead imply that agents interact on several markets (Good Markets, Credit Markets, Financial Markets, Labor Market, Currency Markets,...).

Furthermore, **certain classes of agents interact on more than one market** during the simulation:

Households interact with firms on the labor market, with firms on the consumption good market, with banks on the credit market, etc.
Firms may interact with other firms on good markets (more than one: suppliers, customers), with households on the labor market and financial markets, with banks on the credit market and financial markets etc.

Therefore, we modified and integrated the structure of JABM in order to make it suitable for developing/running macroeconomic models.

Outline

Thinking Agents

Agents can be: individuals, economic/social groups, organizations such as firms, banks etc., institutions, even physical and biological entities,... According to their type **they are gathered in different classes**.

Agents can be composed of other agents, thus permitting hierarchical construction (e.g. a firm composed of workers and managers).

Each agent in encapsulated in a piece of computer code which includes **data** and **methods**. Methods are procedures that operate on agents' data enabling them to take decisions/actions.

Object-Oriented Programming [Minar et al., 1996]

A natural way of programming agents is to use an *Object-Oriented Programming language* (OOP) such as *Java*, *C++*, or *Python*.
Objects are program structures that hold **data and methods** operating on these data.

Members of a class

- Objects' data are attributes/characteristics stored in **fields** and can be of any type.
- Objects' method are **blocks of operations** that act on the data. They define the behavior of each object.

In OOP languages, objects are created from *templates* called **classes** which specify the type of data stored and the methods available to act on the data.

Object-Oriented Programming [Minar et al., 1996]

All the objects *instantiated* from the same class share the same methods and data structure, although their attributes values may differ.

The classes themselves are **arranged in a hierarchy**, with subordinate classes **inheriting** the methods and data of superior classes but **adding** additional ones or **replacing** some of the superior's attributes and methods with more specialized substitutes.

Class hierarchy

Modularity

The concept of modularity in OOP refers to organizing a structure where different elements of a software are divided into separate functional units.

Modularity leads to re-usability. If the components of the software are written in abstract way to solve general problems, they can be used in various applications. This allows to build a *hierarchy* of classes where classes can be ordered from general to specialized.

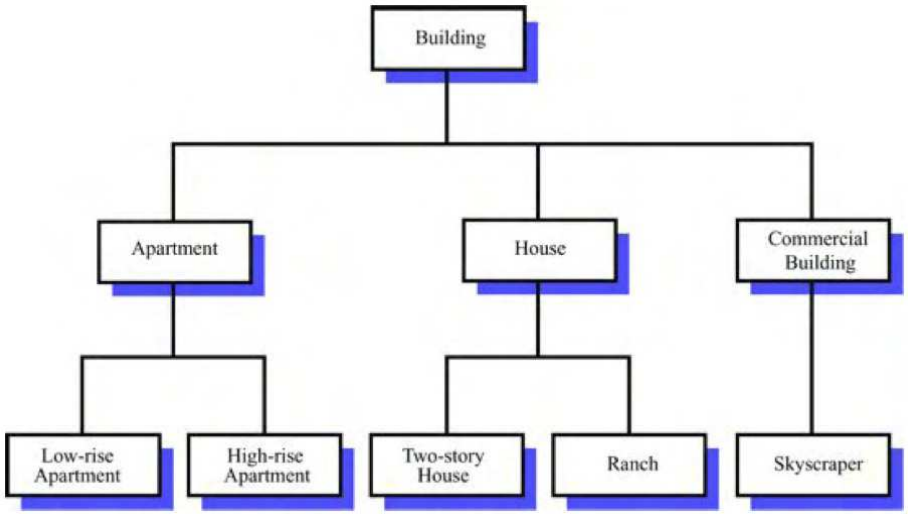

Figure : Source: Goodrich and Tamassia [2001]

Classes and inheritance, a simple example

We create a class named "Firms".
This class will be characterized by a certain number of:

- general attributes defining the state of the firm, such as: stock of capital, workers employed, technology coefficients (capital/labor ratios and productivity of capital/labor), output, prices...etc.
- general methods: procedures to define the level of output, production function, price-setting procedures, investment functions etc. which operate on data to take decision/actions.

Each instance instantiated from this class will automatically inherit the attributes and methods of the class.

Classes and inheritance, an example

Then, suppose we want to distinguish between different types of firms.
For example we allow some firms to imitate the technology of others by creating a subclass named "Imitative Firms".
This subclass inherits all the slots/methods of the superior class, but in addition it contains:

- specific attributes (e.g. a list of other firms' technologies)
- specific methods (a method to choose which technology to imitate and how to implement it).

Similarly we can imagine to subdivide the general class "Firms" in different categories characterized by different investment/pricing/production behaviors, different innovation strategies, different financing behavior, and so on.

The advantages of OOP [Gilbert, 2008]

Once a set of classes has been defined, individual agents are generated by creating instances from them ('instantiation').
By specifying the rules at the class level, all agents instantiated from that class share the same rules/attributes, while the contents of their memories can differ between agents.
Hence OOP provides a practical way to **manage heterogeneity** among agents within and across the classes of agents populating our models.
In addition, OOP leads naturally to a useful **encapsulation**, with each agent clearly distinguishable within the program.
In other words it allows both to **look at the functioning of a model from the level of the observer (emergent dynamics) and to go inside the single objects** of our simulated world through particular **"probes"** which track and control the variables and functions of each agent during the simulation

OOP and modularity I [Minar et al., 1996]

In addition to being a natural way to implement multi-agent simulations, OOP is also a convenient technology for **building libraries of reusable software**.

Modelers can start to build models by directly instantiating useful classes from pre-existing libraries (Swarm, JABM, JAS, RePast etc.).

If there is no particular class that has the precisely needed behavior, one can take a preexisting class and specialize it, adding new variables and methods (or replacing some of them) via inheritance.

Modularity $\rightarrow$ **Easy to share** codes.

OOP and modularity

AB models may differ under several respect such as:

- the types of agents involved in the simulation;
- the types of markets (goods, credit, financial assets etc.) and stocks that are considered (deposits, cash, real capital, bonds, shares etc.);
- the sequence of events taking place within each period;
- the strategies followed by agents to take decisions, such as consumption, investment, pricing decisions;
- the mechanism through which transactions are cleared
- the scheme of amortization of capital
- ...

This raised an important issue related to the project design of our platform: in order to provide a general tool for AB-SFC modeler the platform should then allow to develop a high variety of models without having to intervene on its fundamental structure $\rightarrow$ Modularity

OOP and modularity II

We conceived our platform as a system composed of **rather independent functional blocks** communicating and interacting with each other while running the program.
Each functional block takes the **inputs to operate from some other blocks, elaborate them, and send its output to some other functional blocks** which use it as an input.
Each block does not need to know how the other blocks work in order to operate.
The only thing that matters is that functional blocks **agree to a a "contract" that spells out how the blocks interacts**. In OOP these contracts are called interfaces.
This allows to change the features of each functional block without having to intervene on the others and without compromising the functioning of the "system" and to develop each block without the need to know how the other blocks operate.

Interfaces

An interface is a reference type similar to a class, but it provides **only the signature of some methods** (i.e. their name and the types of input they uses) without their implementation, that is **without the "method's body"** defining how the method elaborates the inputs to produce its output.

To use an interface, we must write a class that **"implements"** the interface.

When a class implements an interface, it provides a method's body for each of the methods declared in the interface.

A class might implement numerous interfaces.

Interfaces: an example in JMAB I

Let's consider how a transaction between two agents is realized in JMAB:

- a **strategy of supply-agents** to determine their supply and a strategy to determine the price;
- a **strategy of demand-agents** to choose the counterpart and a strategy to set their demand;
- a **transaction mechanism** which compares desired demand and available supply, check that agents have enough resources to perform the transaction, and realizes the transfer.

To operate, **the transaction mechanism must** know what is the demand and supply, the prices, and the means of payment that will be used to clear the transaction and it have to **"ask to the agents involved in the transaction" these information**.

Interfaces: an example in JMAB II

But many types of agents may be involved in a certain type of transaction:

E.g. Households and consumption firms buying goods, capital and consumption firms selling their output, consumption and capital firms asking for loans, households and firms buying a deposit.
These **different agents usually have also different strategies** to set their demand/supply, the prices, and so on.

Therefore, the **transaction mechanism should be designed independently from the specific strategies followed by agents**, so that it can operate every time a certain type of transaction is realized (e.g. purchase of a good, emission of a new loan, purchase of a financial asset such as deposit, or a bond etc.) regardless the types of agents involved in the transaction.

Interfaces: an example in JMAB III

The only thing that should matter for the transaction mechanism to operate is that there is an agent acting as a buyer with a certain demand and certain means of payment and an agent acting as a supplier, with a given supply and a given price.
We define several interfaces representing the different roles agents can perform on the different markets: *Deposit Demander (Supplier), Good Demander (Supplier), Labor Demander (Supplier), Bond Demander (Supplier)* etc.
Each interface contains the signature of the methods related to the role they represent: for example *"setGoodsDemand"* for a *Good Demander*, *"setLoanRequirement"*, *"decideLoanLength"*, *"decideLoanAmortizationScheme"* for a *Credit Demander*.

Interfaces: an example in JMAB IV

These methods are then implemented by the classes representing the different types of agents in our models: *Households*, *ConsumptionFirms*, *Capital Firms*, *Government*, *Banks*, *CentralBank* etc. For example *Households* are usually *LaborSupplier* and *GoodDemander*, they might also be *BondDemander*, *CreditDemander* etc.; *Firms* are usually *CreditDemander*, *LaborDemander*, *GoodSuppliers* etc.
Instead of developing as many transaction mechanisms as the different types of agents possibly involved in a certain type of transaction, **we can develop a unique transaction mechanism, which calls the methods defined by the demand/supplier interfaces** without having to know how they are implemented by the classes representing agents.

The structure of agents in JMAB I

- We defined a **taxonomy** of agents based on the their functional role in the economic system. We can thus draw a **hierarchy based on 4 main branches**, each one having at its root an abstract class representing a *Household*, a *Firm*, a *Bank* or a *Government*.
- These can be seen as basic prototypes containing fields and methods shared by all agents belonging of a certain type. In turn, these abstract classes are an extension of an even more fundamental class called *SimpleAbstractAgent* which contains fields and methods shared by all types of agent (for example, the *StockMatrix*).
- The concrete classes representing agents implements some of the interfaces seen above according to the roles they perform on the different markets.

The structure of agents in JMAB II

- As we are **generally interested in analyzing different agents' strategies** (different pricing, investment, purchasing strategies for example) we keep the definition of these strategies separated from the definition of the classes. That is, **agents and their strategies are two independent blocks**.
  For this sake, we use again interfaces: e.g. a *InvestmentStrategy* defining a general method "*computeDesiredGrowth*" , a *FinanceStrategy* defining a method "*computeCreditDemand*" , *WageStrategy* defining a methods "*setAskedWage*" , a *TaxPayerStrategy*, a *SupplyCreditStrategy*, a *SelectWorkersStrategy*, a *SelectSellerStrategy* etc.
- This interfaces are then implemented by specific classes of strategies, representing concrete behaviors.

The structure of agents in JMAB III

- If one wants to analyze a new behavior by an agent, **he only have to develop the concrete class representing the new strategy without having to intervene in the class representing the agents that use that strategy**. The only thing to do is to inject the new strategy in the agents through the configuration file. This is called: **"Dependency Injection"**.

Dependency injection in JABM and JMAB [Phelps, 2012]

As already seen, OOP provides a very general and flexible framework to deal with different classes of agents, which allows AB models developed using OOP to be easily analyzed under a wide variety of assumptions concerning agents' behaviors, and the models' specification.

This features are enhanced by the adoption of a particular design pattern called *Dependency Injection* [Fowler, 2004].

Dependency injection is one of the most interesting features of JMAB.

The logic of Dependency Injection

Roughly speaking, the **construction of the general attributes of classes and instances of agents is maintained separated from their configuration**, so that we can easily change agents' strategies, parameters value, simulations' initial conditions, without having to 'hard-code' them in the code representing the structure of the underlying simulation model.

That is, the general attributes of each agent (fields) are defined in the code representing the model, but the contents of these fields is defined outside the model and then injected into it through a separate XML file.
These feature enhance the modularity and flexibility of JMAB.

Dependency Injection and MC simulations

AB models are stochastic and executed as Monte Carlo simulations.

Therefore, we run the simulations many times to **check for across run volatility induced by stochastic terms and to perform sensitivity analysis.** Different draws of free parameters from a certain distribution, different distributions, or systematic increase in a certain parameter region ('parameter sweep').

One solution is to directly write pieces of code, for example in each class' constructor, which initialize each free parameter from a certain probability distributions, or alternatively set explicitly its value.
But **in this way we are forced to modify the code representing the objects every time we want to check for alternative configurations**
$\rightarrow$ Time demanding, potential source of defects and bugs, less general.

Dependency Injection and MC simulations

Dependency Injection offers an alternative to overcome this drawback by **separating the definition of the objects (i.e. the implementation of the model) from their configuration**. The configuration information are stored in a separated file and then *injected* in the model when starting the simulation.

In this way we can write the code representing the agents **without having to make any assumption regarding how the properties** representing agents' attributes **are initialized** and we can re-execute the model under different configurations without having to change the code.

Dependency Injection and models' behavioral specification

Beside being a useful tool for running AB models as Monte Carlo simulations, the combination of the advantages brought about by OOP principles and Dependency Injection allows to **easily change also the behavioral equations of the models, without having to modify the structure of a model**.

That is, we can easily change the configuration of: agents' routines, agents' learning algorithms, agents expectation formation mechanisms, pricing, investment, production rules, market protocols, market matching mechanisms, without having to hard-code anything in the model.

We only have to focus on the development of the new specific behavior we want to model (creating a new class) and then we can inject it through the XML configuration file $\rightarrow$ Modularity, Flexibility, Accessibility.

Sequence of Events

Traditional AB models

As already seen, most macro AB models are written as a sequence of matrix operations which defines the sequence of events in every tick of the simulation. In most cases, the modeller has a specific order in mind, reflecting a particular vision (theory) of how the economy works.

A crucial feature for a platform aspiring to provide a **general, flexible and non theoretically-biased tool** should then be the neutrality regarding the definition of the sequence of events. Hence, in JMAB we adopted a different approach to define the sequence of actions during each period of the simulation, based on the use of specific objects called **events**.

Tick events

Events (General)

An event is an object sent from an object called *EventScheduler* to a number of objects called *EventListener*. The event may contain data used by listeners' to perform actions through their methods. It can thus be seen as a **message** being sent to all **listeners** that **subscribed** to the broadcasting channel of events.

In the case of JMAB, we created a class symbolizing tick events. Each one of these ticks represent a sub-period of one period of the simulation.
There are two types of tick event: **market** and **agent** and each tick event contains a **unique id**.
These tick events are then sent to agents by the MacroSimulation ("*OrderedEventsSimulation*").

Here is how it works

The simulation has a list of ticks events stored in one of its fields ("events"). For each of these:

1. the simulation sends the tick event
2. the agents listening to the broadcasting channel (one for each type of tick event) receive the message
3. depending on the type of event and on value of the tick id
   - each agent uses zero, one or more methods to update its data
   - each market invokes agent interactions or not

Agent tick event

The agents listening to these events will react to these messages through their methods, taking decisions and performing actions based on the information they possess and/or the information passed to them by the message contained in the event.
Every event is characterized by a particular index identifying the event and its position in the ordering of events, decided ex ante by the modeler.

For example, the "COMPUTEEXPECTATIONS" event, identified by index 1, is the first one to be sent to agent in each simulation round telling them to compute expectations (on demand, wages etc.), the " CAPITALPRICE" "CONSUMPTIONPRICE" events (index 2 and 3) then tell capital and consumption firms to set the price of their output, the "INVESTMENTDEMAND" event, asks consumption firms to compute their desired rate of growth and then the amount/type of real capital units to buy, etc...

Market tick event

Market tick events trigger a set of operation representing market interactions. In JMAB, a market is composed of:

- two populations (buyers and sellers) ("*MarketPopulation*"),
- a mixing mechanism ("*MarketMixer*") determining which agent is being activated (a specific buyer or seller) and specifying who are the counterparts with whom the activated agent is allowed to interact (the potential sellers or buyers),
- a matching mechanism which selects one of the counterparts according to the specific strategy followed by agents (e.g. *SelectSellerStratetegy*, *SelectLenderStrategy*, *SelectDepositStrategy*, *SelectWorkerStrategy*),
- and, finally, a transaction mechanism (e.g. *CreditMechanism*, *GoodMechanism*, *BondMechanism*, *DepositMechanism*, etc.) which supervises the actual exchange between the activated agent and its counterpart.

Benefits

- the sequence of events is not imposed by the platform, but it is determined by the modeller
- the definition of the order of events is kept totally separated from the definition of the agents' methods. Hence we can change the ordering without having to change the blocks representing agents.
- it allows for the possibility of modelling different frequencies of economic processes (e.g. more than one market interaction per period).
- it gives the possibility of dealing with asynchronous decisions by agent (agents can decide whether to react or not to a specific tick event).

Conclusion

The platform does not superimpose any structure to the model, leaving the modeller full flexibility in defining the sequence and timing of events

Fundamental representation of an agent

Mother class

Each agent, may it be of the type household, firm, bank or government, is endowed with a stock matrix, representing its real and financial stocks, subdivided into assets and liabilities. This *StockMatrix* is inherited from the superior *SimpleAbstractAgent* class.

The structure of the stock matrix:

- two lists representing assets and liabilities containing
  - as many slots as there are types of stocks (each one identified by a specific index), containing
    - Lists of **Items** representing stocks of a certain type (e.g. capital, durable consumption goods, cash, loans, deposits, bonds, shares etc. held by an agent

The Stock Matrix and agents' Balance Sheet

Item

All stocks are object of the abstract class *Item*, which is then specialized by a number of concrete classes representing specific types of stocks. All **real goods** contain generic and specific information regarding, for example, its owner, its producer, its price, its age, or its productivity. All **financial assets** contain generic and specific information regarding, for example, the liability holder and asset owner, its value or its interest rate.

The **stock matrix is then deeply related to the representation of agents' balance sheet** (though being a broader concept), as it contains all the information required to compute it.

Indeed, we can easily access the accounting value of the various stocks, aggregating them for each type of asset/liability, and then summing them up in order to derive the total value of assets and liabilities, with the net wealth as balancing item.

Benefits

- flexible and efficient tool to store information and manage heterogeneity
- deeply related to the representation of agents' balance sheet
- network representation of stock-matrix interdependencies via asset-liability duality of each financial item
- helps in ensuring the stock-stock consistency. Every time a stock is updated (e.g. a share of a loan is repaid), also the balance sheets of agents holding it as an asset/liability are automatically updated.

Flow-flow and Stock-flow consistency

Both consistencies are guaranteed by the **transaction mechanisms** such as *AtOnceMechanism* (for purchases of real/financial goods), ConstrainedCreditMechanism (for loans), DepositMechanism, ReserveMechanism (for banks' cash advances asked to the CB).

- The transaction mechanism first of all assesses whether the chosen supplier has enough supply to satisfy agent's demand. If not we have a supply-constrained transaction.
- In the case of a purchase of a good or financial assets:
  - the mean of payment accepted by the supplier is identified (e.g. cash or deposit transfer)
  - the transaction mechanism assesses whether the buyer has enough liquidity in form asked by the seller. If not, an internal transfer is made (e.g. from agent's deposit to cash or vice-versa). If still not enough the transaction is financially-constrained.
- The transaction mechanism updates the Stock Matrices of agents' (e.g. seller's inventories and buyers stock of that good, seller's and buyer's cash/deposits) ensuring that the flows arising out of the transaction respect *Copeland's quadruple entry system*.

From micro to meso to macro

JMAB...

1. provides a **comprehensive representation of the complex dynamic network linking agents' balance sheets** and its evolution through time as transactions between different agents occur. This allows:
   - network analysis
   - fragility analysis
2. provides an **overall balance sheet and flows of funds representation** of the economy allowing for
   - more traditional analysis at the macroeconomic level
   - synthetic measures to assess the state of the economy and its future evolution

Conclusion

JMAB **allows the researcher to browse through all the possible levels of aggregation according to his needs**, thus providing a fully scalable vision of the functioning of the simulated economy

References I

L Amaral, S. Buldyrev, S. Havlin, H. Leschhorn, P. Maas, M. Salinger, E. Stanley, and M. Stanley. Scaling behavior in economics: Empirical results for company growth. *Journal de Physique*, 7:621–633, 1997.

G. Bottazzi and A. Secchi. Why are distribution of firms growth rates tent-shaped? *Economic Letters*, 80:415–420, 2003a.

G. Bottazzi and A. Secchi. Common properties and sectoral specificities in the dynamics of u.s. manufacturing companies. *Review Of Industrial Organization*, 23:217–232, 2003b.

G. Bottazzi, E. Cefis, and G. Dosi. Corporate growth and industrial structures: Some evidence from the italian manufacturing industry. *Industrial and Corporate Change*, 11-4:705–723, 2002.

H. De Wit. Firm size distributions. an overview of steady-state distributions resulting from firm dynamics models. *International Journal of Industrial Organization*, 23:423–450, 2005.

References II

D. Delli Gatti, E. Gaffeo, and M. Gallegati. *Emergent Mcroeconomics. An AB Approach to Business Fluctutations*. Springer, 2008.

G. Dosi, G. Fagiolo, and A. Roventini. An Evolutionary Model of Endogenous Business Cycles. *Computational Economics*, 27(1):3–34, 2005.

J. Epstein and R. Axtell. *Growing Artificial Societies: Social Science from the Bottom-Up*. MIT Press and Brooking Press, Washington D.C., 1996.

M. Fowler. Inversion of control containers and the dependency injection pattern, 2004. URL http://martinfowler.com/articles/injection.html.

N. Gilbert. *Simulation for the Social Scientist*. McGraw-Hill, 2008.

M. T. Goodrich and R. Tamassia. *Data structures and algorithms in Java*. John Wiley & Sons, 2001.

References III

S. Kinsella, M. Greiff, and E. J. Nell. Income Distribution in a Stock-Flow Consistent Model with Education and Technological Change. *Eastern Economic Journal*, 37:134–149, 2011.

N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. *Santa Fe Institute Working Paper Series*, 96-06-042:–, 1996.

S. Phelps. Applying dependency injection to agent-based modeling: the jabm toolkit. *CCFEA Working Paper*, 56-12:33, 2012.

S. Phelps and K. Musial-Gabrys. Network motifs for microeconomic analysis. *CCFEA Working Paper*, 63:8, 2012.

L. Riccetti, A. Russo, and M. Gallegati. An agent-based decentralized matching macroeconomic model. *Journal of Economic Interaction and Coordination*, Forthcoming, 2014.

Pascal Seppecher. Monnaie Endogène et Agents Hétérogènes dans un Modèle Stock-Flux Cohérent. In *Political economy and the Outlook for Capitalism*, Paris (France), 2012.

References IV

J. Steindl. *Random Processes and the Growth of Firms*. Hafner, New York, 1965.

S. Sunder. Determinants of economic interaction: Behavior and structure. *Journal of Economic Interaction and Coordination*, 1:21–32, 2006.

L. Tesfatsion and K. Judd. *Handbook of Computational Economics. Volume 2: Agent-Based Computational Economics*. North Holland, Amsterdam., 2006.