Agent Coordination Simulator in Julia

The code developed in Julia was based on “Agent-Based Modeling and Social Coordination Through Money” paper, written by Paulo Garrido, where agent-base behaviours are seen from a computacional view and it's intended to simulate the basic process of operation. This simulator intends to, through a series of rules of production and consumption of products, as is in reality, to simulate the state of every agent involved in the process.

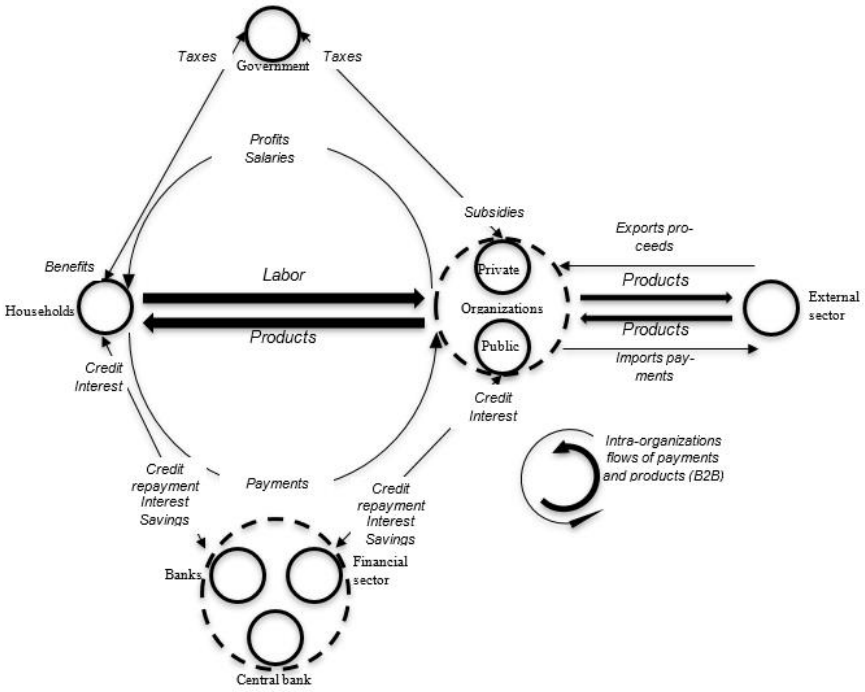

Figure 1 - Financial sector overview

A second paper has been published, under the conference CONTROLO 2016, that shows the initial simulation results achieved by the first version of the developed software. Some minor errors were fixed regarding the previous versions and some major changes were done.

First, a port to the new version of Julia was made, to be able to be developed under Atom IDE which offers critical features that help the development. The flow of relations, as rules consumption, production, selling and taxing were stabilized under the current 0.6.3 version. The data types were updated to the following, after the minor changes and the port:

```
#CONTROLLER AGENT
type Controller
  taxPercentage::Float64 # Taxes could be a list of taxes, indexed by Producer
  Goals::List{ControllerGoal}
  Prices::List{Float64}    #

  function Controller(tax = 1.0)
    this = new()
    this.taxPercentage = tax
    this.Goals = List{ControllerGoal}(ControllerGoal)
    this.Prices = List{Float64}(Float64)
    return this
  end
end
```

Figure 2 - Controller data structure

```
#PRODUCER AGENT
type Producer
  Numeraire::Float64
  InputStore::List{Symbol}
  OutputStore::List{Symbol}
  Enabled::Bool
  ID::Int64
  numeraireToBeTaxed::Float64
  Internal::Bool
```

Figure 3 - Producer data structure

```
type List{T}
  vec::Vector{T}
  addContent::Function
  deleteList::Function
  deleteContent::Function
  copyList::Function
  getSize::Function
```

Figure 4 - List data structure

```
type Symbol
  Symbol::AbstractString              # Strings of simbols (Unicode)
  Amount::Int64                 # Quantity
  announcedToProduction::Int64 #Amount to produce announced product
end
```

Figure 5 - Symbol data structure

```
type ControllerGoal
  Symbol::AbstractString
  Ymin::Int64
  Ynom::Int64
```

Figure 6 - Controller Goal data structure

```
#Defining Product Offer type
type ProductOffer
  Producer::Int64
  Units::Int64
  OrderedUnits::Int64
  UnitPrice::Float64
end
```

Figure 7 - Product Offer data structure

```
#Defining a Cell of the B List
type BList_Cell
  Product::AbstractString
  Producers::List{Int64}
  Offers::List{ProductOffer}
  AvgPriceSold::Float64
  Remaining::Int64
```

Figure 8 - BList Cell data structure

```
type Rule
  Antecedent::List{Symbol}     # A - the producer's copy of the antecendent multi-set
  Consequent::Symbol            # b - the producer's copy of the consequent string
  y::Int64                     # yA -> y(qb)
  Type::AbstractString          # a -> Possible&Saleable b -> Possible&Unsaleable c -> Necessary&Saleable
  Ymin::Int64                  # Ymin - minimum number of strings that the rule must produce in a k peri
  YnomList::List{Int64}        # YnomList - Nominal values are no longer expressed by Ynom, but are a fu
  Ymax::Int64                  # Ymax - maximal number of strings that the rule can produce in a period
  antecedentsReady::Bool
```

Figure 9 - Rule data structure

```
type SuperSystem
  C::Controller           #Controller Object
  P::List{Producer}       #List of Producers
  V::List{List{Rule}}     #List of Rules per Producer
  B::List{BList_Cell}     #B List
  K::Int64                #K cycles
  N::Int64                #N Producing Systems
  PricingList::List{StandardPriceCell}   #List of prices by Symbol
  ActiveProducers::Int64  # Current number of active producers in the super system
```

Figure 10 – Super System data structure

All the data structures presented followed the scheme on the software specification provided by Paulo Garrido, the author, and suffered changes throughout the development process. The main process of the simulation is now different, being the files hosted in different directories, to allow easier access.

A system identification must be provided, to allow the system to reach a determined configuration file and output a file (simulation result) regarding that configuration file. This change allowed a better testing procedure, where multiple situations can be created and tested using multiple configurations. The output file is populated directly with Scilab commands, that are read by a proprietary Scilab application that plots the generated data.

For simplicity, the system configuration is done in just one JSON file, to allow ease of configuration. To initialize and run a simulation, in the “main” file, one should provide the system ID and make sure that the corresponding configuration file is in the configuration directory.

```
SystemID = "TestD0"; #System Identifier (Simulation ID)
```

Figure 11 - System Identification

```
outputFileName = string("./outputs/simulation.",SystemID,".txt");
systemConfigFileName = string("./configs/System.",SystemID,".json");

S,K,N= SuperSystem(systemConfigFileName);
```

Figure 12 - Setup of the system

The file names are created and the SuperSystem is configured. The simulation procedure hasn't suffered any changes, as major changes will be taken when the simulation of macro-economies will take place, as credits, savings, external sector will be a major influence in the system's behavior.

Furthermore, efforts are being made to collect data from the National Institute of Statistics (INE) and the Bank of Portugal (Banco de Portugal) to achieve a model that represents the Portuguese economy, in order to have real significance to the simulation results. Although the system is not ready to implement this kind of simulations, the basis were tested and changes will be made further on to give support to the newer requirements.