`black_rhino` – A Financial Multi Agent Network Simulator

Co-Pierre Georg*

Date: 12/09/2012

Version: 0.9.5

License: GNU GPL v3

URL: http://sourceforge.net/projects/oxblackrhino/?source=directory

Introduction

black_rhino is an open source easy-to-use-and-adapt **financial network multi-agent simulation** (MAS) that serves two purposes. First, it can be used as a practical tool to simulate and analyse a model banking system. This is particularly handy for central banks and policy makers, as **black_rhino** fills a gap in the policy-toolbox. Second, and perhaps more importantly, it is a **python** module that can be easily adapted, changed, and modified for research purposes. It is intended to reduce the amount of work necessary to write a financial MAS and hence allows researchers to focus on the economic questions instead of worrying about code design patterns and basic functionality.

This software is intended for **educational and research purposes**. Despite my best efforts, I cannot fully rule out the possibility of errors and bugs. A number of tests are provided together with the software that aim to minimize this risk, but the use of **black_rhino** is entirely at your own risk.

This document is a short tutorial that will walk you through the first steps with **black_rhino** and elaborates on a number of the ideas behind the software. It is advisable that you have a look at Georg (2011) where I outline the basic algorithm and some of the notation that is used in this tutorial. The paper can be found here. After downloading and extracting **black_rhino.tar.gz**, you should find a folder **./black_rhino/** in your current directory. This folder contains the source code, a number of example files, and this tutorial.

Please note: **black_rhino** is published under the **GNU GPL v3**. If you are unsure about the implications, check the website of the Free Software Foundations.

1. Using black_rhino

First of all, you have to distinguish between a Python version based on **Windows** and a version based on **Linux**.

* Deutsche Bundesbank, Wilhelm-Epstein-Strasse 14, D-60431 Frankfurt am Main, Germany; and University of Oxford, Park End Street, Oxford OX1 1HP, United Kingdom. E-Mail: co-pierre.georg@keble.ox.ac.uk. This tutorial and the most recent version (0.9.5) benefitted substantially from excellent research assistance provided by Florian Urbschat.

Depending on your system you have to un/comment on statement in the main **black_rhino** start file. In order to execute **black_rhino** for **Windows** you have to use
args=['./black_rhino.py',"environments/","test10","log/","measurements/"]

When using **Linux**
args = sys.argv
has to be used and the parameters have to be passed from the command line. The respective other statement should be made inactive by marking it as a comment. The usage for linux is thus:
./black_rhino.py $environment_directory/ $environment_identifier $log_directory/ $measurement_directory/

Example: (being in **./black_rhino/Tutorial/)**:
./black_rhino.py environments/ test10 log/ measurements/

Even though **black_rhino** is intended to be easy to use and adapt, this does not make it entirely easy to use the first time. This section outlines the basic usage, a later section elaborates on the program's internal logic. First things first: **black_rhino** requires three inputs and generates two outputs. The three inputs are: (i) the network structure; (ii) the bank data; and (iii) the environment file.

The **network structure** is typically a file in **.gexf** format that holds information about all nodes (e.g. banks) and their linkages (e.g. interbank loans or mutual credit line agreements). An example file with a random network of ten banks is given in **networks/network-random-10-0.8.gexf**. The program **./br-generate_networks.py** can be used to generate other network configurations. At the moment only random and Barabasi-Albert networks are implemented, but thanks to **networkx**, this is easily changed. The **bank data** contains all the information that is necessary to characterize the different banks in the simulation. Example bank data can be found in **Tutorial/banks/10/** and consists of ten files **bank-0.xml** – **bank-9.xml**. Each bank has its own file to allow for high flexibility when characterizing individual banks by their parameters. The program **./br-generate_banks.py** can be used to generate further banks with a standard configuration. Finally, the **environment file** is the file that determines in which economic environment the simulation takes place. It is located in the **environment_directory/** and denoted by the **environment_identifier** (e.g. **test10**) and without the **.xml** extension. All the parameters that are specific to the macroeconomy and the regulatory environment are stored here. Also, a number of parameters that are common to all banks (or firms, households, and the central bank) are collected in the environment file.

The first type of output that **black_rhino** generates are **measurements** and can be found in the **measurement_directory/**. During a normal run, five files will be created. They all have the format **$identifier-histo$Observable.dat** where **$identifier** is the identifier that is specified in the environment file and **$Observable** is either **ActiveBanks** (the number of banks active at a certain point in time), **D** (the amount of deposits), **I** (the overall investment), **LC** (the central bank credit), or **L** (the interbank market volume). Each measurement file is a histogram where each line in the file is an individual simulation (remember that simulations have to be run a number of times to get a meaningful average) and each **i**-th entry in a line is an observation at time **i**. The histograms have to be transformed into result files that can be plotted e.g. using **gnuplot**. I usually use **R** to read the histograms and create mean, std deviation, etc. and then **gnuplot** for the plotting itself. But of

course you can also use **R** for plotting. An example **R** file is provided in **measurements**/ and called **doPlots.R**. It can be executed in a unix shell as: **cat doPlots.R | R --no-save --args $identifier $numBanks $numSweeps** (e.g. **doPlots.R | R --no-save --args test10 10 100**) where **$identifier** is the identifier given in the environment file, **$numBanks** is the number of banks defined in the environment file, and **$numSweeps** is the number of sweeps (i.e. the number of update steps) defined in the environment file.

The other type of output is a **log file**, created in **log_directory**/ (typically **log**/) that is named **$identifier.log** and contains information about the run. The logfile is useful in many circumstances, as it will report a number of errors that could occur (e.g. through misusage or because a certain exception was triggered in the run) and also give information how long a run takes (which is very useful, as you can start a run with one simulation only and then extrapolate how long 1000 simulations would take). The log file also provides some information about network statistics.

Finally, the flow of data during a run of **black_rhino** is shown in Figure 1, showing the two types of output that **black_rhino** generates.

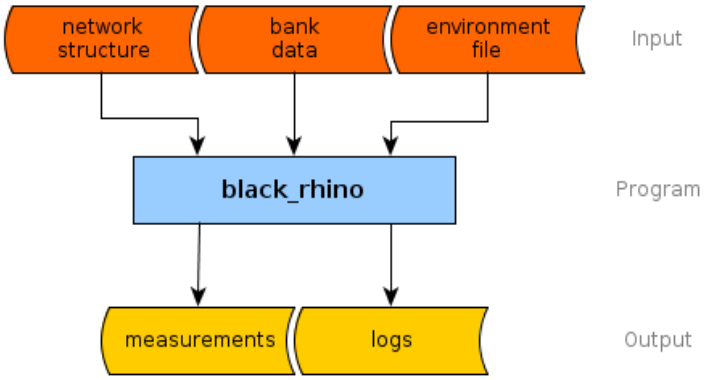

Figure 1: *The flow of data during a run of **black_rhino**.*

2. The Internal Oranization of black_rhino

Besides using **black_rhino** for financial multi-agent simulations, modifying the source code will be the most common task one faces. This section outlines the internal organization of **black_rhino** and explains some of its design principles.

black_rhino (other financial network multi-agent simulations will be designed similarly) can be structured in five elementary building blocks, shown in Figure 2. A flowchart of the **__main__()** program is given in Figure 3, showing the different stages of a typical run.

The **Environment** describes the world the simulation takes place in. It is implemented in the **class Environment** and consists of three other classes: the **class Parameters** contains all the parameters that are unchanged during the run of a simulation. This includes the simulation **identifier**, the number of simulations **numSimulations**, the number of update steps **numSweeps**, the number of banks at the start of a simulation **numBanks**, the directory where the initial bank data can be found: **bankDirectory**, and the location of the gexf-file that holds the network of

contracts information, **contractsNetworkFile**. This set of parameters is specified in the environment file **$environment_directory/$environment_identifier.xml**.

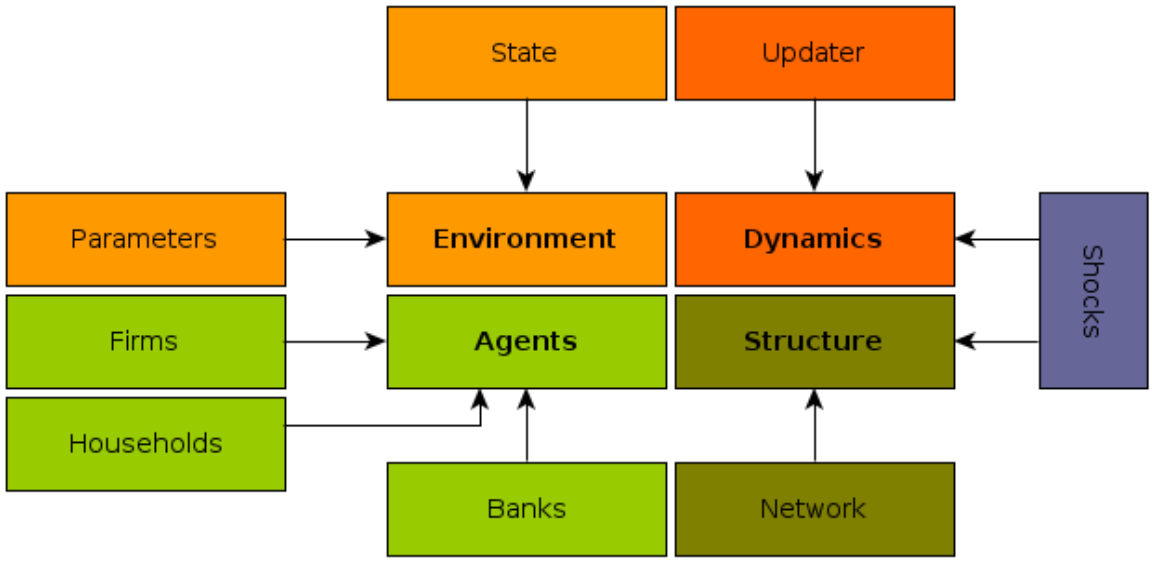

Figure 2: *The five elementary building blocks of a financial network multi-agent simulation.*

Typically, the structure of the environment file will look like this:

```
<environment title='test10'>
    <!-- PARAMETERS THAT STAY CONSTANT DURING THE SIMULATION -->
    <parameter type='numSweeps' value='100'></parameter>
    <parameter type='numSimulations' value='1'></parameter>
    <parameter type='numBanks' value='10'></parameter>
    <parameter type='bankDirectory' value='banks/10/'></parameter>
    <parameter type='graphType' value='gexf'></parameter>
   <parameter type='contractsNetworkFile' value='networks/network-random-
10-0.8'></parameter>

    <!-- PARAMETERS THAT CHANGE DURING THE SIMULATION -->
    <!-- parameters determining the payment flow of banks -->
   <parameter   type='changing'   name='rb'   value='0.02'   validity='0-
1000'></parameter>
   <parameter   type='changing'   name='rd'   value='0.04'   validity='0-
1000'></parameter>
    …
    …
</environment>
```

The **class Parameters** also contains a list **parameters[]** of parameters which may change during the run of a simulation. These are identified by **type='changing'** and the range for which a given parameter is valid is given by **validity='to-from'**. The **class State** is basically a container that holds all the parameters needed in the course of the simulation.

The **Structure** of the financial network is captured in the **class Network**. In **black_rhino** this class is part of the **class Environment** since the network structure contains information about the world

the simulation takes place in. The network class is a class that contains much functionality and uses the **networkx** library, which can be downloaded from http://networkx.lanl.gov (**black_rhino** 0.9 uses **networkx** 1.6). Besides an identifier, the network class contains two networks: **contracts = nx.DiGraph()** is a (in general directed, in this version undirected) network of contracts between banks. It represents mutual agreements between banks to engage in interbank lending and is only changed when a bank is removed from the simulation (due to default). The other network, **exposures = nx.DiGraph()** is the network of actual exposures between banks. It rapidly changes during the course of the simulation, representing interbank linkages.

The **Agents** are described in the class Banks. The only agents in this version of **black_rhino** are banks. Each bank has a list **accounts[]** where all the transactions (which are objects of the **class Transaction**) a bank has performed is stored. The balance sheet of a bank is effectively expressed through the set of transactions which a bank has performed (and which are still on the bank's books at the current point in time).

The optimization behavior of banks is described in Georg (2011), as is the update algorithm that describes how the system evolves from one state to another. This update algorithm is part of the **class Updater** and constitutes the main part of the model **Dynamics**. The update algorithm is depicted in Figure 4, where the colors correspond to the different elementary building blocks. To simplify debugging, the updater is not called directly from **__main__()**, there rather is a **class Runner** which contains a **runner.do_run()** that is executed in a loop within **__main__()**.

The runner does three things. First, it takes care of the actual update step by calling **updater.do_update()**. Second, it checks whether there is a **Shock** in the current update step. If so, **class Shock** can execute a shock (currently, only two types of shock are implemented, but this is easily expanded). And third, the runner takes care of the **Measurements**, implemented in **class Measurement**. At each update step, there will be a measurement of all the state variables, stored in a histogram that is written to the hard disk at the end of the run.

This structure ensures that the code is easier to debug and adapt. Details about the interface of each class can be found within the actual .py files. Moreover, if you are rather **new to python or object-oriented-programming** in general, Florian Urbschat has provided some **FAQs** which might help you to understand **black_rhino** in greater detail. You can find the FAQs here **black_rhino\tutorial\faq**

3. Test Your Code

One of the most critical steps in developing extensions and modifications to **black_rhino** is writing tests. Typically, a researcher will try to avoid this *„unecessary“* step in the code development as no *„new“* results are being produced. New modifications to the code will only be accepted, however, if they are properly tested to ensure the integrity of the code. There are two types of tests: (extended) **unit tests**, and **simulation tests**. While unit tests ensure the proper functioning of a small piece of code, simulation tests are larger tests that simulate extreme economic conditions. The behavior and economic interpretation of the observables and hence the economic transmission channels of the model can be understood in these extreme situations.

Writing a number of unit tests serves two purposes and its relevance cannot be underestimated. First of all, only proper unit testing can ensure that the code does exactly what it is supposed to do. It will save you a lot of time later on if you write your test at the same time you write a modification and extension to **black_rhino**. Unit tests should be written frequently for smaller parts of codes and once these parts are individually tested, also for larger parts. The other important reason for using unit tests is that they guarantee that a routine (a method or a larger part of code) do exactly the same when they are refactorized. This becomes relevant if you refactorize for instance the interbank lending routines for higher speed (this is the innermost loop, where the simulation spends most of the time).

black_rhino has a separate program called **./br-make_tests.py** that is used to execute individual tests. Useful example data is provided in the directory **tests**/. In **src**/**tests.py** a number of tests for different parts of the code is already provided. Of course, there could always be more tests and with future revisions of **black_rhino** further tests will be provided.

References

Georg, Co-Pierre, *„The Effect of the Interbank Network Structure on Contagion and Common Shocks“*, Deutsche Bundesbank Working Paper Series 2, 12-2011, (**2011**)

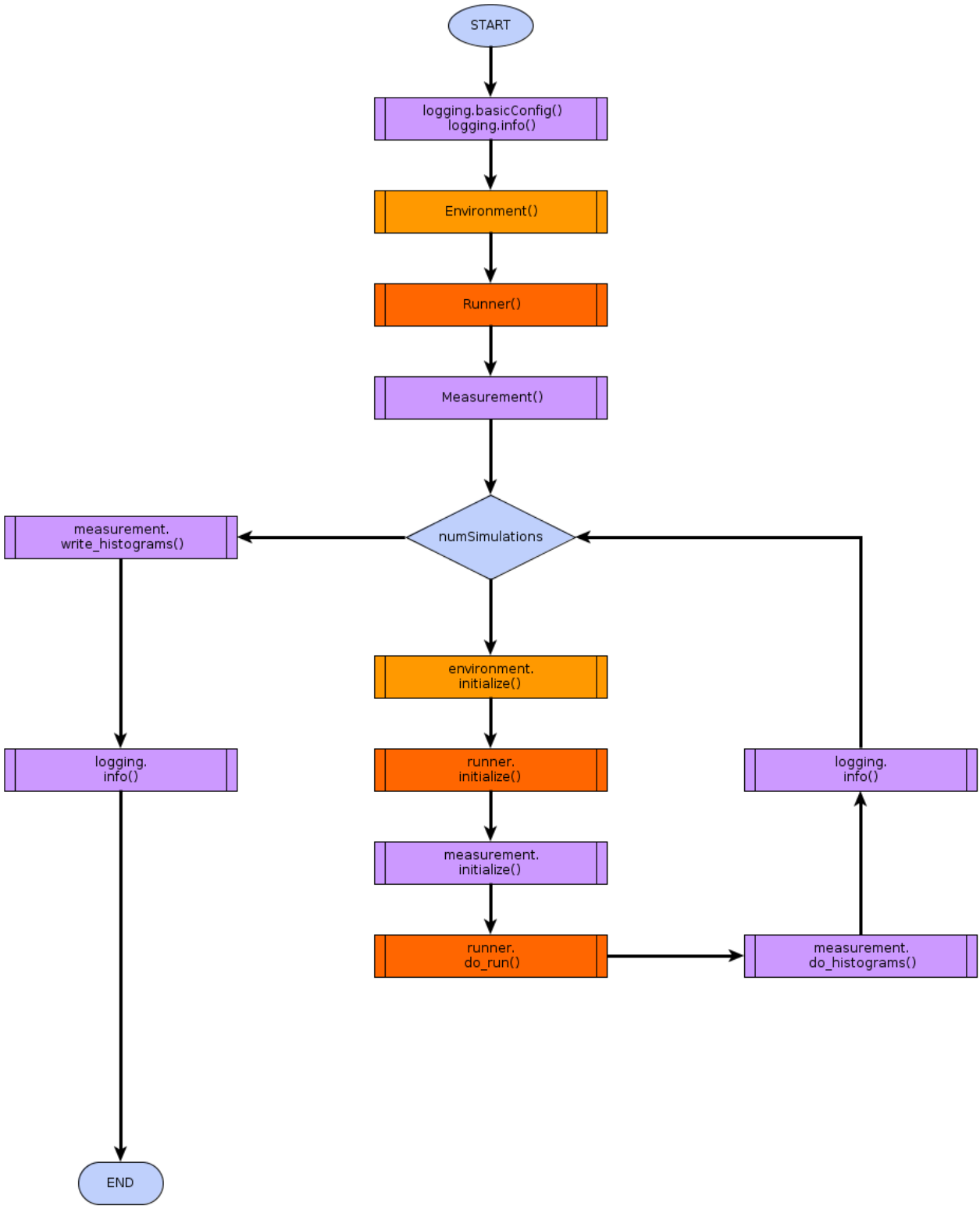

Figure 3*: Flowchart of the main program* ***__main__()****. Colors correspond to the elementary building blocks.*

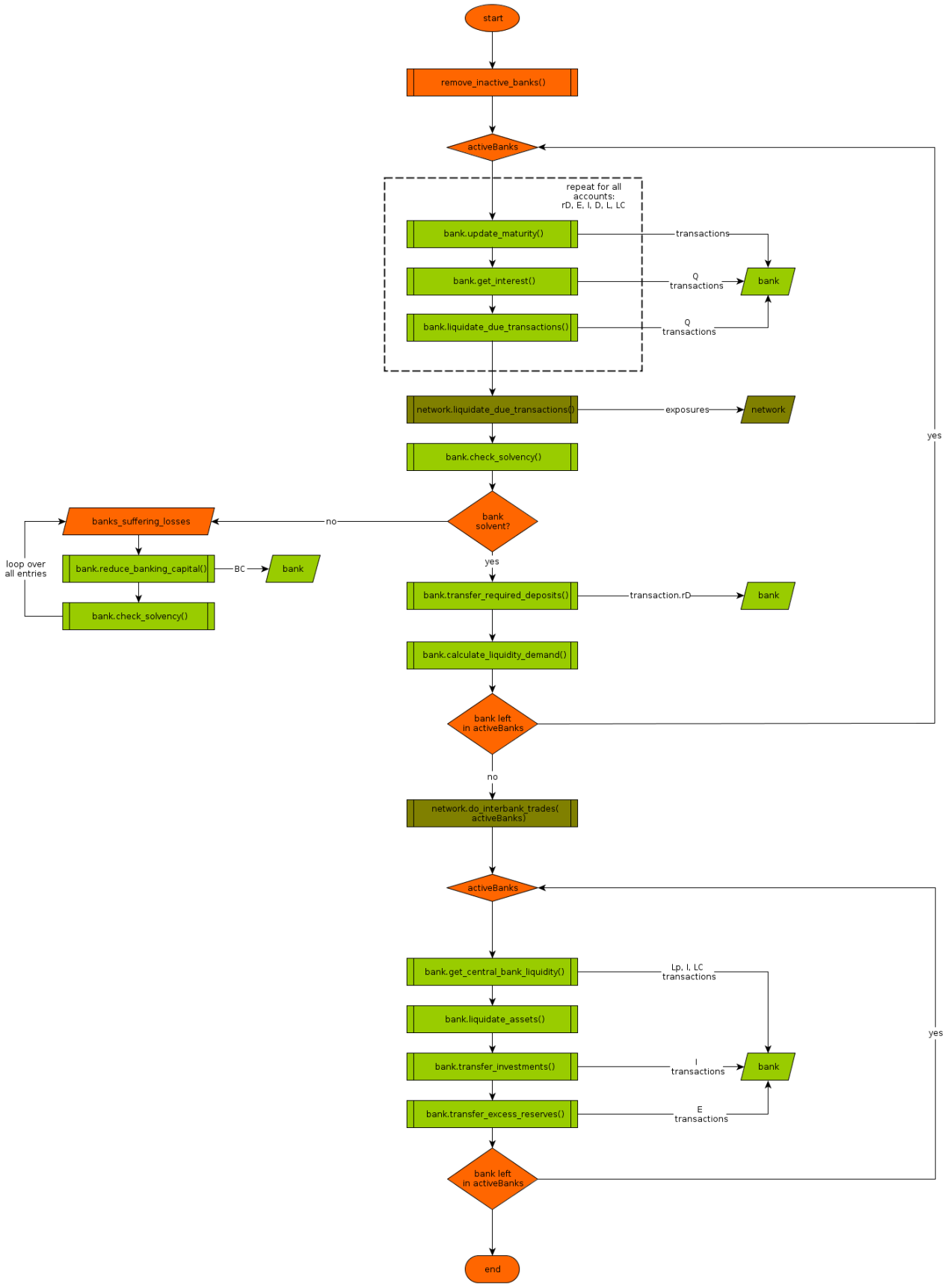

Figure 4: The flow diagram of **updater.do_udpate()**. Colors correspond to colors of the elementary building blocks.