Banking, Shadow Banking, and Financial Regulation

An Agent-based Approach

Inaugural-Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Wirtschafts- und Sozialwissenschaften
der Wirtschafts- und Sozialwissenschaftlichen Fakultät
der Christian-Albrechts-Universität zu Kiel

vorgelegt von
Diplom-Volkswirt/Diplom-Kaufmann Sebastian Krug
aus Halle/Saale

Kiel, 2017

Gedruckt mit Genehmigung der Wirtschafts- und Sozialwissenschaftlichen Fakultät
der Christian-Albrechts-Universität zu Kiel

Dekan: Prof. Dr. Till Requate

Erstbegutachtung: Prof. Dr. Hans-Werner Wohltmann
Zweitbegutachtung: Prof. Dr. Thomas Lux
Drittbegutachtung: Prof. Dr. Christian Merkl

Tag der Abgabe der Arbeit: 09.08.2016
Tag der mündlichen Prüfung: 29.09.2016

This thesis entails the following contributions:

Publications

- Krug, Sebastian & Lengnick, Matthias & Wohltmann, Hans-Werner, 2015. "The Impact of Basel III on Financial (In)stability: An Agent-based Credit Network Approach", *Quantitative Finance*, vol. 15(12), pp. 1917– 1932.
- Lengnick, Matthias & Krug, Sebastian & Wohltmann, Hans-Werner, 2013. "Money Creation and Financial Instability: An Agent-based Credit Network Approach", *Economics – The Open-Access, Open-Assessment E-Journal*, Kiel Institute for the World Economy, vol. 7, pages 1-44.

Unpublished Working Paper

- Krug, Sebastian & Wohltmann, Hans-Werner, 2016. "Shadow Banking, Financial Regulation and Animal Spirits: An ACE Approach", Economics Working Papers 2016-08, Christian-Albrechts-University of Kiel, Department of Economics. (Currently under review at *Journal of Banking & Finance*)
- Krug, Sebastian, 2015. "The Interaction between Monetary and Macroprudential Policy: Should Central Banks 'Lean Against the Wind' to Foster Macro-financial Stability?", Economics Working Papers 2015-08, Christian-Albrechts-University of Kiel, Department of Economics. (Currently under review at *Journal of Money, Credit, and Banking*)

Acknowledgements

Although I have put only my name on this thesis, there are of course many people who contributed to its success in various ways.

First and foremost, I would like to mention my first supervisor *Prof. Dr. Hans-Werner Wohltmann* who gave me the opportunity to realize my dissertation project without hesitation. His sensitivity for strengths and weaknesses of his students enables him to give proper guidance in any situation. With his always positive and enthusiastic view on research projects, he created a work environment which was as motivating as it was stimulating and inspiring. But his view on his responsibility as a supervisor goes way beyond the academic part. Thus, I'm particularly grateful for his effort to support the financing of my post-graduate studies. The creative way in which he always tried to find solutions, e.g. by issuing several small projects, showed me how much these issues were also of his concern. In addition, I'm very grateful for his effort regarding my two-year scholarship since he was the driving force of receiving the grant. At the same time, his general open-door policy and his habit of giving tremendously fast, focused and valuable feedback facilitated the way in which I benefited from his experience.

In this regard, his encouragement to "learn by doing" by instantly throwing me into a research project with more experienced colleagues was especially helpful. This directly brings me to *Dr. Matthias Lengnick*. He significantly influenced my view on general equilibrium theory, computational economics and how important the simplicity of agent-based models is. Hence, I had the pleasure to work with him in the same field and the fact that two out of four papers of this thesis are co-authored with him best reflects how close the collaboration was during the last years. I'm also grateful to him that I quickly got in touch with agent-based computational economics (ACE) although learning this methodology just being equipped with merely rudimentary programming skills means a highly non-linear, iterative and sometimes frustrating process. This makes model development as well as model extension very time consuming but the upside is that the researcher, in turn, gets deep insights and intuitive understanding of the underlying system. So, I'm grateful for the countless questions I was able to ask and for always dealing with my frequently emerging technical and non-technical problems.

I also want to thank *Dr. Sven Offick* for kindly sharing an office with me, for several years full of interesting conversations on macroeconomic and non-macro topics and for being always there when I had questions about mathematical problems.

Of course, there are a lot of other people and colleagues I met during the last years which I have to thank. Among them, I want to mention *Jun.-Prof. Dr. Roland Winkler* from the TU Dortmund for his significant contribution to the grant of my scholarship by providing a positive evaluation letter. Furthermore, I have to thank *Jun.-Prof. Dr. Sander van der Hoog* from Bielefeld University for the stimulating discussions we had and for many hints concerning related literature. I also want to thank *Prof. Mark C. Lewis, Ph.D*, for acting on his conviction by writing the first and, so far, only book on introductory programming on the market which uses Scala as a first programming language. Without his book, it would have been a lot harder and much less effective to achieve programming skills enabling me to build agent-based models on my own. Also many thanks go to *Dr. Alexander Totzek* and *Dr. Stephen Sacht* who provided first insights into the procedures of Kiel University's PhD-program and, thus, nourish my wish to likewise start a dissertation project.

Finally, I want to thank my family for supporting me over such a long period of time and especially for listening so often to my monologs about the current state of our financial system, monetary policy as well as macroeconomics in general without even complaining once. Moreover, I'm very grateful to my close friends from which I received considerable support in various ways. Here, a very special and personal thank goes to my old friend *Benjamin Burghardt* for motivating me, for always being there and, of course, for being lenient with me that I have left the team for such a long time. I'll be back...

Contents

List of Acronyms

ACE	Agent-based Computational Economics
ABM	Agent-based Model
AS	Animal Spirits
BA	Bank (Agent)
BCBS	Basel Committee on Banking Supervision
BD	Broker-dealer
BoE	Bank of England
CAR	Capital Adequacy Requirement
CB	Central Bank
CConB	Capital Conservation Buffer
CCQ	Core Capital Quota
CCycB	Countercyclical Buffer
CET1	Common Equity Tier 1 Capital
CFSI	Composite Financial Stability Indicator
CsD	Cross-sectional Dimension of Systemic Risk
DSGE	Dynamic Stochastic General Equilibrium Model
ECB	European Central Bank
FASB	Financial Accounting Standards Board
FSB	Financial Stability Board
GDP	Gross Domestic Product
GE	General Equilibrium Model
G-SIB/G-SIFI	Global Systemically Important Bank / Financial Institution
HH	Household
HQLA	High-quality Liquid Assets
IDL	Intraday Liquidity
IMF	International Monetary Fund
LCR	Liquidity Coverage Ratio

LKW	Lengnick, Krug, Wohltmann (2013)
LOLR	Lender of Last Resort
LR	Leverage Ratio
MC	Monte Carlo (Simulation)
MMF	Money-market Mutual Fund
NSFR	Net Stable Funding Ratio
ODD	Overview, Design Concepts, Details
OSF	Operational Standing Facility
OSDF	Operational Standing Deposit Facility
OSLF	Operational Standing Lending Facility
OTC	Over-the-counter (market)
ROE	Return on Equity
RTGS	Real Time Gross Settlement (Payment System)
RWA	Risk-weighted Assets
SFC	Stock-flow Consistent
TR	Taylor Rule
TvD	Time-varying Dimension of Systemic Risk
VAT	Value-added Tax
ZIA	Zero Intelligence Agent

List of Tables

List of Figures

"Imagine how hard physics would be if electrons could think . . . "

[Murray Gell-Mann, Nobel Laureate in Physics from 1969]

Chapter 1

General Introduction

The global financial crisis has revealed serious gaps in the ability of standard macroeconomic models that were typically used for quantitative and empirical investigations to either define, measure and manage externalities resulting from recent developments within the financial intermediation process. The following deep recession emphasized the necessity to address these deficiencies and to improve the understanding of the linkages between financial sector activity and macroeconomic aggregates.

As a consequence, the field of *"macro-finance"*, i.e. the intersection of financial economics and macroeconomics, received much attention [Morley (2015)] through the integration of banking, corporate finance and financial markets into macroeconomic models using various methodologies. Although standard (equilibrium) macro-models are still used, they are typically just augmented with ad-hoc assumptions when it comes to financial sector activity. To push policy-orientated macroeconomic modeling beyond this approach, *agent-based computational economic (ACE) models* has been identified as a new class of models that is able to overcome these deficiencies by enabling the modeling of dynamics resulting from the endogenous formation of systemic risk, bubbles and contagion effects. Therefore, these models help to gain insights into newly identified sources of financial instability and serve as suitable experimental labs to test the performance of monetary, fiscal and financial stability policies that aim to mitigate the negative effects of such phenomena in order to provide proper guidance for decision makers in central banks and financial supervisory authorities. The ultimate goal of the field is to contribute to the development of a regulatory framework that ensures the stability of the financial system without suppressing its growth-supporting capacity.

This dissertation consists of papers that aim to contribute to the macro-finance area using agent-based computational (ACE) methods. It includes two already published articles (chapter 2 and 3) as well as two working papers that are submitted and currently within the peer-review process (chapter 4 and 5). In particular, the papers cover

- financial stability issues that has been identified as main sources of systemic risk being held responsible for the occurrence of the recent global financial crisis,

- potential extensions of the deficient regulatory framework to mitigate accompanied externalities as well as
- possible conflicts with monetary policy and
- the regulatory inclusion of shadow banking activities.

A more detailed description of the research done can be found below.

First paper (chapter 2) The second chapter presents a small-scale, stock-flow consistent agent-based computational model that covers a simple monetary economy based on the transactions among households, firms and banks. All agents follow very simple behavioral rules. The resulting model is well suited to explain money creation in line with the standard theory of fractional reserve banking. Instead of enforcing an equilibrium state by assumption, we show that it emerges endogenously from individual interactions in the long run. Therefore, the model represents a generalization of standard (equilibrium) theory. Moreover, it is novel in the sense that individual interactions also create an interconnected banking sector giving rise to systemic risk and bankruptcy cascades. Hence, financial instability, in this model, is inevitably interwoven with the creation of money and, thus, can be seen as an intrinsic property of modern monetary economies.

We find that the existence of an interbank market has a twofold effect: As a source of liquidity, it has stabilizing effects during normal times but amplifies systemic instability, contagion and bankruptcy cascades once a crises has been triggered. But even with no interbank market, indirect contagion can lead to bankruptcy cascades. We identify maturity mismatches between different assets and liabilities as the driving force that, first, builds up systemic risk and, second, triggers financial crises endogenously. We also find that the existence of large banks threatens financial stability and that regulatory policy should target large banks more strictly than small ones.

The chapter is based on a joint article with Dr. Matthias Lengnick and Prof. Dr. Hans-Werner Wohltmann entitled “Money Creation and Financial Instability: An Agent-based Credit Network Approach”. The article is published in the journal *Economics: The Open-Access, Open-Assessment E-Journal, Volume 7, Issue 32, pp. 1–44.* My contribution consists of substantial parts of the literature research and the theoretical model development. The entire programming was done by Dr. Matthias Lengnick.

Second paper (chapter 3) The third chapter deals with the current Basel accord on banking regulation, namely Basel III. With this proposal, regulators have reacted to the recent global financial crisis with, first, a revision of microprudential instruments and, second, the introduction of several new macroprudential instruments. This approach of cumulating several requirements bears the risk of single measures negating or even conflicting with each other, which might lessen their desired effects on financial stability. Hence, the question arises, whether the concurrent imposition of instruments leads to a regulatory environment in which they (perhaps partially) offset each other's individual contribution to financial stability.

We use the model proposed in chapter 2 to provide an impact study of Basel III which evaluates both, the isolated and joined impact, of most of its instruments. The literature, of course, has already evaluated most of them. Unfortunately, the majority of the available studies deal with single instruments only, thus, providing no insight into potential conflicts between them. To get the joined impact of several (or all) instruments, one can not simply sum up the contributions of individual instruments in isolation. Our model allows for the simultaneous imposition of several instruments. It also gives rise to the sources of systemic risk (cross-sectional and time-varying dimension) that Basel III aims to reduce. Hence, our model is well suited for an impact study of Basel III.

With respect to microprudential instruments, we find that the positive joint impact of all instruments is considerably larger than the sum of individual contributions, i.e. the standalone impacts are non-additive. Concerning the macroprudential overlay, the impacts are either marginal or even destabilizing except for the buffers (CConB and CCycB) which indeed represent indispensable instruments to counteract agents' pro-cyclical behavior. It is worth mentioning that two instruments contribute most to financial stability: The newly introduced liquidity coverage ratio (microprudential), and the flexible (i.e. buffered) capital requirement (macroprudential). Although the leverage ratio embodies a synthesis of both, non-risk sensitivity and simplicity, it falls short of expectations. The same holds for surcharges on systemically important institutions which have a quite moderate standalone and even destabilizing multi-dimensional impact. Hence, surcharges in their current implementation only contribute to financial regulation's complexity and not to the resilience of the system.

The chapter is based on a joint article with Dr. Matthias Lengnick and Prof. Dr. Hans-Werner Wohltmann entitled "The Impact of Basel III on Financial (In)stability: An Agent-based Credit Network Approach" published in *Quantitative Finance (2015), Volume 15, Issue 12, pp. 1917–1932.* My contribution consists of the development of the research question, the writing, the literature research as well as substantial parts of the theoretical model development. The entire programming was done by Dr. Matthias Lengnick.

Third paper (chapter 4) Chapter four presents a completely new agent-based macro-model with heterogeneous interacting agents and endogenous money developed for policy analysis in the macro-finance context. We show that the model is able to replicate common various stylized facts related to the macroeconomy, the credit market and financial crises, hence, making it a suitable experimental lab for this area.

After the recent financial crisis of 2007-09, two policies have been found adequate to increase the overall resilience of the financial system, i.e. monetary and macroprudential policy. Unfortunately, the Deutsche Bundesbank (2015) has acknowledged that *"[a]s both monetary policy and macroprudential policy measures initially affect the financial sector, interaction between these two policy areas is inevitable. However, at the current juncture, experience and knowledge of the functioning of macroprudential instruments [...] and the way in which they interact with each other and with monetary policy are rather limited."* Thus, in the present paper, the model serves as framework for the analysis in order to shed some light on the interaction between monetary policy and financial regulation. We do this by capturing the current debate on whether central banks should lean against financial imbalances and whether financial stability issues should be an explicit concern of monetary policy decisions or if these should be left to the macroprudential approach of financial regulation.

Our results provide three main findings. First, we find that extending the monetary policy mandate in order to achieve price, output and financial stability simultaneously, confirms the proposition of Tinbergen's "effective assignment principle" in the sense that it is not possible to improve financial stability *additionally* without negatively affecting the traditional goals of monetary policy using the same policy instrument. In contrast, using (macro)prudential regulation as an independent and unburdened policy instrument significantly improves the resilience of the system by restricting credit to the unsustainable and high-leveraged part of the real economy. Hence, our results strengthen the view that both policies are designed for their specific purpose and that they should be used accordingly in order to avoid excess macroeconomic volatility through overburdened policy instruments.

Second, "leaning against the wind" should only serve as a first line of defense in the absence of prudential financial regulation. Even in such a setting, a central bank response to financial sector imbalances just improves macroeconomic stability while the effect on financial stability is only marginal.

Third, our results confirm that, in line with Adrian and Shin (2008a,b), both policies are inherently connected and, thus, influence each other which emphasizes that an appropriate coordination is inevitably and that the prevailing dichotomy of the currently used linear quadratic framework may lead to misleading results.

The chapter is based on a single-authored working paper entitled "The Interaction Between Monetary and Macroprudential Policy: Should Central Banks 'Lean Against the Wind' to Foster Macro-financial Stability?". It was submitted to the *Journal of Money, Credit and Banking* in November 2015 and is still under review. The entire research project was done on my own.

FOURTH PAPER (CHAPTER 5) The aim of the last chapter is to shed some light on the transition the credit system has been through over the last decades and on the destabilizing externalities accompanied by this transformation, in particular, the substantial shift in market risks faced by financial institutions. Aggravating this situation, the permanent seek of market participants for regulatory arbitrage has led to the continuous build up of a parallel and unregulated banking system *"in the shadows"*, i.e. beyond the reach of regulators. Unfortunately, shadow banking does not only reduce the costs of the financial intermediation process but exhibits an extensive contribution to overall systemic risk due to i) the lack of prudential regulation, ii) the lack of access to a public safety net (liquidity and roll over risk) as well as iii) the reliance on extreme short-term funding sources (through the money market). The contribution of this paper is to analyze the effects of an inclusion of the shadow banking sector into the current regulatory framework on economic activity and whether such a proceeding would be suitable to internalize the described destabilizing externalities.

The underlying model extends the model developed in chapter 4 by a shadow banking sector representing an alternative investment opportunity for households. Following the seminal work of Akerlof and Shiller (2009), the investment decisions of households can be characterized by animal spirit-like, i.e. highly pro-cyclical, herd-like and myopic, behavior. The presented model is well suited to analyze the research question at hand since pro-cyclical behavior as well as sudden and common withdrawals of invested funds has been identified as one of the root causes of systemic failures of the recent past.

Our simulation experiments provide three main findings. First, our results suggest that switching the regulatory regime from a *"regulation by institutional form"* to a *"regulation by function"* meaning the inclusion of shadow banks into the regulatory framework, as proposed by Mehrling (2012), seems to be worthwhile in general terms.

Second, supervisory authorities should do so in a coordinated and complete manner. A unilateral inclusion, i.e. burdening the shadow banking sector with the same regulatory requirements as traditional banks but denying the access to the public safety net leads to inferior outcomes compared to the benchmark case (no shadow banking activity) and even to the case in which they are not regulated at all. The results of such cases include negative effects on monetary policy goals, significantly increases in the volatility of growth and financial and real sector default rates as well as a higher volatility in the credit-to-GDP gap.

Moreover, experiments with a full and complete inclusion, i.e. with access to a lender of last resort, lead to superior outcomes in terms of the central bank's dual mandate, economic growth and financial stability suggesting that a full inclusion of the shadow banking sector into the regulatory framework could indeed, from a theoretical point of view, lead to a significant mitigation of the destabilizing externalities accompanied by their fragile funding model and to a suitable exploitation of their liquidity provision capacity in terms of sustainable growth.

Finally, the paper is useful to understand why the accessibility of contagion-free, alternative sources of liquidity to the *whole* financial sector is of such great importance. Our results show that the massive risks originating from boundedly rational agents that interact freely in a prospering and completely unregulated part of the financial system without a liquidity backstop can lead to states of the system that are comparable to the recent financial crisis.

The chapter is based on a joint working paper with Prof. Dr. Hans-Werner Wohltmann entitled “Shadow Banking, Financial Regulation and Animal Spirits – An ACE Approach” and submitted to the *Journal of Banking & Finance* in June 2016. My contribution consists of the entire theoretical model development, the programming, the literature research and writing as well as the analysis. The original idea for the project came from Prof. Dr. Wohltmann.

References of Chapter 1

Adrian, T. and Shin, H. S. (2008a). Financial Intermediaries, Financial Stability, and Monetary Policy, *Staff Report no. 346*, Federal Reserve Bank of New York.

Adrian, T. and Shin, H. S. (2008b). Liquidity, Monetary Policy, and Financial Cycles, *Current Issues in Economics and Finance* **14**(1): 1–7.

Akerlof, G. A. and Shiller, R. J. (2009). *Animal Spirits: How Human Psychology Drives the Economy, and Why It Matters for Global Capitalism*, Princeton University Press, New Jersey.

Deutsche Bundesbank (2015). The Importance of Macroprudential Policy for Monetary Policy, *Monthly Report* **67**(3): 39–73.

Mehrling, P. (2012). Three Principles for Market-Based Credit Regulation, *American Economic Review* **102**(3): 107–112.

Morley, J. (2015). Macro-finance Linkages, *Journal of Economic Surveys* (forthcoming).

Chapter 2

Money Creation and Financial Instability
– An Agent-based Credit Network Approach –

Published in: Economics: The Open-Access, Open-Assessment E-Journal (2013), Volume 7, Issue 32, pp. 1–44.

Co-authors: Matthias Lengnick, Hans-Werner Wohltmann

Abstract

We develop a simple agent-based and stock flow consistent model of a monetary economy. Our model is well suited to explain money creation along the lines of mainstream theory. Additionally, it uncovers a potential instability that follows from a maturity mismatch of assets and liabilities. We analyze the impact of interbank lending on the stability of the financial sector and find that an interbank market stabilizes the economy during normal times but amplifies systemic instability, contagion and bankruptcy cascades during crises. But even with no interbank market, indirect contagion can lead to bankruptcy cascades. We also find that the existence of large banks threatens stability and that regulatory policy should target large banks more strictly than small.

Keywords: Financial Instability, Agent-based Macroeconomics, Stock-flow Consistency, Disequilibrium Analysis, Basel III.

JEL Classification: C63, E42, E51, G01

The full article can be downloaded via
http://dx.doi.org/10.5018/economics-ejournal.ja.2013-32

DOI: 10.5018/economics-ejournal.ja.2013-32

Chapter 3

The Impact of Basel III on Financial (In)stability – An Agent-based Credit Network Approach –

Published in: Quantitative Finance (2015), Volume 15, Issue 12, pp. 1917–1932.

Co-authors: Matthias Lengnick, Hans-Werner Wohltmann

Abstract

The Basel III accord reacts to the events of the recent financial crisis with a combination of revised micro- and new macroprudential regulatory instruments to address various dimensions of systemic risk. This approach of cumulating requirements bears the risk of individual measures negating or even conflicting with each other which might lessen their desired effects on financial stability. We provide an analysis of the impact of Basel III's main components on financial stability in a stock-flow consistent (SFC) agent-based computational economic (ACE) model. We find that the positive joint impact of the microprudential instruments is considerably larger than the sum of the individual contributions to stability, i.e. the standalone impacts are non-additive. However, except for the buffers, the macroprudential overlay's impact is either marginal or even destabilizing. Despite its simplicity, the leverage ratio performs poorly especially when associated drawbacks are explicitly taken into account. Surcharges on SIBs seem to rather contribute to financial regulations complexity than to the resilience of the system.

Keywords: Banking Supervision, Basel III, Liquidity Coverage Ratio, Macroprudential Regulation, Financial Instability, Agent-based Computational Economics.

JEL Classification: G01, G28, E40, C63

The full article can be downloaded via
http://dx.doi.org/10.1080/14697688.2014.999701

DOI: 10.1080/14697688.2014.999701

Chapter 4

The Interaction between Monetary and Macroprudential Policy: Should Central Banks "Lean Against the Wind" to Foster Macro-financial Stability?

Submitted to: Journal of Money, Credit, and Banking (2015), currently under review.

Abstract

The extensive harm caused by the financial crisis raises the question of whether policymakers could have done more to prevent the build-up of financial imbalances. This paper aims to contribute to the field of regulatory impact assessment by taking up the revived debate on whether central banks should "*lean against the wind*" or not. Currently, there is no consensus on whether monetary policy is, in general, able to support the resilience of the financial system or if this task should better be left to the macroprudential approach of financial regulation. We aim to shed light on this issue by analyzing distinct policy regimes within an agent-based computational macro-model with endogenous money. We find that policies make use of their comparative advantage leading to superior outcomes concerning their respective intended objectives. In particular, we show that "*leaning against the wind*" should only serve as first line of defense in the absence of a prudential regulatory regime and that price stability does not necessarily mean financial stability. Moreover, macroprudential regulation as unburdened policy instrument is able to dampen the build-up of financial imbalances by restricting credit to the unsustainable high-leveraged part of the real economy. In contrast, leaning against the wind seems to have no positive impact on financial stability which strengthens proponents of Tinbergen's principle arguing that both policies are designed for their specific purpose and that they should be used accordingly.

Keywords: Financial Stability, Monetary Economics, Macroprudential Policy, Financial Regulation, Central Banking, Agent-based Macroeconomics.

JEL Classification: E44, E50, G01, G28, C63

4.1 Introduction

In a competitive environment, banks' private choices concerning money creation are not socially optimal burdening the economy with externalities and leaving the system vulnerable to financial crises. In this context, the focus is on "*how to exploit the magic of credit for growth without inciting banks to imprudent lending practices*", as Giannini (2011) puts it, and how to avoid states of the financial system which are macro-economically destructive instead of growth-supportive.

Historically, central banks emerged as institutional counterbalance in order to be in control of the banking sector and to restrict the risk of financial imbalances [Haldane and Qvigstad (2014); Hellwig (2014); Stein (2012); Goodhart (1988)]. But over time, the focus more and more turned from (direct) crisis mitigation towards the current dual mandate since it was generally agreed that inflation represents one of the main sources of financial instability and that achieving price stability would be sufficient to ensure also financial stability [Schwartz (1995)]. The occurrence of the recent financial crisis disabused both practitioners as well as researchers.[1]

In the course of the recent resurgence of interest in the nexus of finance and macroeconomics [Morley (2015)], there are numerous invocations to put such considerations back on the research agenda emphasizing that the focus on inflation bears the potential of omitting other measures of economic health [Woodford (2012); Walsh (2014); Borio (2014); Stein (2014); Tarullo (2014); George (2014)]. As a consequence, many central banks face calls to expand their policy goals towards financial stability issues. The corresponding debate is mainly on whether to continue to entirely rely on financial regulation and macroprudential policy instruments to ensure financial stability [Hanson et al. (2011); Criste and Lupu (2014); Tomuleasa (2015)] or to respond directly to financial imbalances through monetary policy.

For the vast majority of central banks around the world, flexible inflation targeting has become the predominant monetary policy regime and proponents argue that financial stability issues can represent a natural extension [Olsen (2015)]. For example, Woodford (2012) states that central banks should implement a policy which is seeking

> *"to deter extreme levels of leverage and of maturity transformation in the financial sector"*. Even *"modest changes in short-term rates can have a significant effect on firm's incentives to seek high degrees of leverage or excessively short-term sources of funding. Again, this is something that we need to understand better than we currently do; acceptance that monetary policy deliberations should take account of the consequences of the policy decision for financial stability will require a sustained*

[1] Albeit even prior to the crisis there was some early awareness of the fact that this view is not correct [e.g. Borio (2006); Issing (2003)]. For empirical evidence on the missing positive correlation between price and financial stability, see Blot et al. (2015).

> *research effort, to develop the quantitative models that will be needed as a basis for such a discussion*".

Moreover, R. Bookstaber adds in his speech at the INET conference 2014 that "*we have to embed financial regulation deeply within macroeconomics and in particular monetary policy, the interface between those two is untried territory*". A similar kind of invocation was also made by Mishkin (2011) who states that "*research on the kind of quantitative models needed to analyze this issue should probably be a large part of the agenda for central-bank research staffs in the near term*".

But there are not only arguments in favor of an extended flexible inflation targeting since monetary and financial-stability policy are distinct and separate policies with different objectives and different instruments, as Svensson (2012) argues. Thus, a direct central bank response to, say, credit growth would inevitably suggest a violation of Tinbergen's famous *effective assignment principle* [Tinbergen (1952)], i.e. to assign only one objective to each independent policy instrument which, in turn, implies that policymakers cannot be "the servant of two masters". Therefore, Svensson emphasizes that "*[...] the policy rate is not the only available tool, and much better instruments are available for achieving and maintaining financial stability. Monetary policy should be the last line of defense of financial stability, not the first line*". Ignoring the principle of Tinbergen bears the risk of an overactive monetary policy leading to a highly volatile target rate which might entail destabilizing effects on the primary goals of the central bank. Also Yellen (2014); Giese et al. (2013) argue that using macroprudential policy would be the more effective and direct way while Smets (2014) emphasizes the importance of an appropriate coordination in order to avoid conflicts of interacting policies.

These considerations necessarily raise the question whether the analysis framework usually used by central banks is the right tool to consult for proper guidance. Existing research in this field is yet still dominated by studies using DSGE models as underlying framework for the analysis [Käfer (2014); Chatelain and Ralf (2014); Plosser (2014)]. In this context, Mishkin (2011) states that the underlying linear quadratic framework of pre-crisis theory of optimal monetary policy has a significant shortcoming, i.e. the financial sector does not play a special role for economic fluctuations. This naturally led to a *dichotomy between monetary and financial-stability policy* resulting in a situation in which both are conducted separately.[2] However, Adrian and Shin (2008a,b) argue against "*the common view that monetary policy and policies toward financial stability should be seen separately, they are inseparable*". Moreover, there are some early studies which have argued that the current monetary policy framework could fail to deal with financial instability because it largely ignores the development of variables that are usually linked to financial imbalances, e.g. credit growth or asset prices [Cecchetti et al. (2000); Bordo and Jeanne (2002); Borio and Lowe (2002, 2004)]. For a more recent critique see Gelain et al. (2012) who

[2]See Suh (2014) which shows the existence of the dichotomy in a New Keynesian model with credit.

state that the analysis of the nexus between monetary and macroprudential policy "*requires a realistic economic model that captures the links between asset prices, credit expansion, and real economic activity. Standard DSGE models with fully rational expectations have difficulty producing large swings in [private sector] debt that resemble the patterns observed*" in the data. Also Agénor and Pereira da Silva (2014) choose a simple dynamic macroeconomic model of a bank-dominated financial system for their analysis because it "*provides [...] a better starting point to think about monetary policy [...] compared to the New Keynesian model [...] which by now is largely discredited. The days of studying monetary policy in models without money (and credit) are over [...]*".[3]

Although the framework is continuously extended and meanwhile also the banking sector and financial frictions are taken into account,[4] relying entirely on a single kind of model to analyze policy issues might bear the risk of "backing the wrong horse".[5] Hence, the new insights gained in the aftermath of the crisis might be a good reason to approach monetary policy analysis within alternative frameworks. Moreover, Bookstaber (2013) strongly argues in favor of agent-based computational economic (ACE) frameworks to do research on financial stability issues.

We contribute to the literature on regulatory impact assessment and the interaction between monetary policy and financial stability in the following way: First, by providing an agent-based macro-model with endogenous money, we contribute to model pluralism in this area. Currently, we are not aware of any comparable studies using an ACE model in this field, except for Popoyan et al. (2015); da Silva and Lima (2015) and somewhat more broadly also Salle, Yıldızoğlu and Sénégas (2013); Salle, Sénégas and Yildizoglu (2013) who analyze the credibility of central bank's inflation target announcements. Second, instead of usually incorporating only single macroprudential policy instruments (e.g. loan-to-value ratio (LTV)), our experiments encompass complete regulatory regimes, i.e. Basel II and Basel III. This enables us to run counterfactual simulations of the model relative to a benchmark scenario which is comparable to the economic environment of the pre-crises period, i.e. a situation with a rather loose regulatory environment (Basel II) and a central bank focusing solely on price and output stability. Based on this benchmark scenario, we then test the impact of either a tightened financial regulation, of various degrees of a central bank's response to financial imbalances and a combination of both. As also done by Gelain et al. (2012), results are considered in terms of the two objectives

[3] See also Disyatat (2010).

[4] Recent examples would be Levine and Lima (2015); Gambacorta and Signoretti (2014); Badarau and Popescu (2015); Rubio and Carrasco-Gallego (2014). For a literature overview on monetary policy and financial stability using DSGE models with financial frictions as framework for the analysis, see Verona et al. (2014); Chatelain and Ralf (2014); Akram and Eitrheim (2008).

[5] Haldane and Qvigstad (2014) state that "*Model or epistemological uncertainty can to some extent be neutralized by using a diverse set of approaches. This, again, can avoid the catastrophic policy errors that might result from choosing a single model and it proving wrong. The workhorse macro-economic model, without banks and with little role for risk and asset prices, predictably showed itself completely unable to account for events during the crisis. Use of this singular framework for example, for gauging the output consequences of the crisis would have led policymakers seriously astray. Using a suite of models which emphasized bank, asset prices and risk transmission channels would generated far better forecasting performance through the crisis [...]*".

of both policies, (macro)economic and financial stability, in order to shed light on potential conflicts and crowding-out effects.

Our experiments provide three main findings. First, assigning more than one objective to the monetary policy instrument in order to achieve price, output and financial stability simultaneously, confirms the expected proposition of the Tinbergen principle in the sense that it is not possible to improve financial stability additionally to the traditional goals of monetary policy. The results of our experiment show that after a long phase of deregulation, "leaning against the wind" has a positive impact on price and output stability but affects the fragile financial system only marginally. Moreover, in a system in which banks have to comply with tight prudential requirements, a central banks' additional response to the build-up of financial imbalances does not lead to improved outcomes concerning both macroeconomic and financial stability. In contrast, using prudential regulation as an independent and unburdened policy instrument significantly improves the resilience of the system.

Second, "leaning against the wind" should only serve as a first line of defense in the *absence* of prudential financial regulation. If the activity of the banking sector is already guided by an appropriate regulatory framework, the results are in line with Svensson (2012) who argues that "the policy rate is not the only available tool, and much better instruments are available for achieving and maintaining financial stability. Monetary policy should be the last line of defense of financial stability, not the first line". Macroprudential policy dampens the build-up of financial imbalances and contributes to the resilience of the financial system by restricting credit to the unsustainable high-leveraged part of the real economy. This strengthens the view of opponents which argue that both policies are designed for their specific purpose and that they should be used accordingly.

Third, our results confirm that, in line with Adrian and Shin (2008a,b), both policies are inherently connected and, thus, influence each other which emphasizes that an appropriate coordination is inevitably and that the prevailing dichotomy of the currently used linear quadratic framework may lead to misleading results.

The remainder of the paper is organized as follows: in section 4.2, we give an overview of the structure of the underlying ACE model followed by a section concerning common macroeconomic stylized facts which can be simultaneously replicated by the model (4.3). Note that appendix A provides the underlying source code of the model. It follows a detailed description of the conducted experiments in section 4.4. Section 4.5 provides a discussion of the results for different monetary policy rules comparing their performance in terms of macroeconomic and financial stability. Section 4.6 concludes.

4.2 The Model

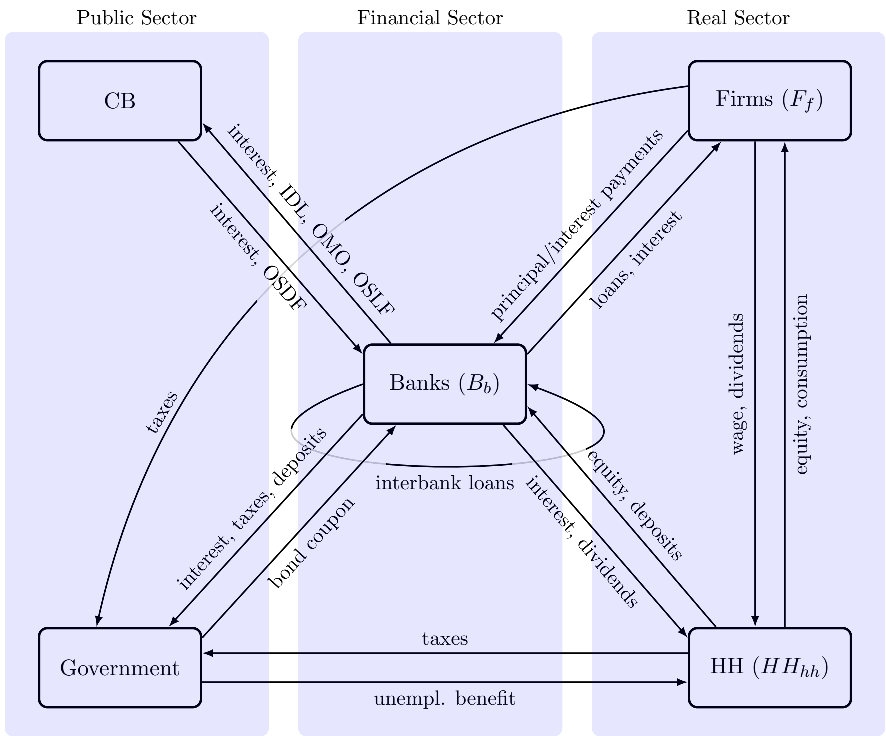


Figure 4.1: Monetary flows in the model

4.2.1 Purpose

The agent-based macroeconomic model presented in the following consists of six types of agents, i.e. households and firms representing the real sector, a central bank, a government and a financial supervisory authority forming the public sector and a set of traditional banks (financial sector). Agents are heterogeneous in their initial endowments of e.g. productivity, amount of employees or clients and interact through a goods, labor and money market in order to follow their own needs, like consuming or making profit. Figure 4.1 provides an overview of the relationships between types of agents on a monetary level.

As a result of the interaction of heterogeneous agents, the model exhibits common macroeconomic stylized facts emerging through the course of the simulation such as endogenous business cycles, GDP growth, unemployment rate fluctuations, balance sheet dynamics, leverage/credit cycles and constraints, bank defaults and financial crises, as well as the need for the public sector to stabilize the economy [Riccetti et al. (2015)] (see also section 4.3).

Since the model should serve as an experimental lab to analyze policies regarding monetary policy and banking regulation, we focus on the monetary system and model it in great detail. Therefore, we adopt as much as possible from the functionality of the real world template provided by the Bank of England's "UK Sterling Monetary Framework" [Bank of England (2014c)]. Here, the CB plays a crucial role since it implements monetary policy as usual in developed countries by setting a target rate which directly affects the whole set of existing interest rates, in particular the rates charged on loans to the real sector by means of increased refinancing costs. Through the resulting effect on credit demand, the CB's monetary policy transmits to overall economic activity, i.e. to production and price levels and, thus, to inflation and output. Therefore, the presented model is well suited to analyze the question of whether macro-financial stability issues should be an explicit concern of monetary policy decisions or if it should be better left to macroprudential regulation and banking supervision. The rest of the paragraph describes the fundamental design concepts of the model.

4.2.2 Design Concepts

The underlying time scheme is divided into ticks (one unit of time) whereas every tick t represents a week. In our model, every month has exactly 4 weeks which leads to an experimental quarter of 12 weeks and an experimental year that consists of only 48 (instead of 52) weeks. This means that variable x_t represents the value of x in tick t while x_{t-12} represents the value of x 12 weeks ago, i.e. the value of the previous quarter.

As stated above, a substantial part of agents' interaction takes place on markets through a matching process. To determine the specific set of matching pairs for a certain action between two agents, i.e. between households and firms on the labor and goods market or between two banks on the interbank market, a pre-selection mechanism is applied to the whole set of agents that generates subsets and, thus, constrains the interaction space in order to meet certain stylized facts. The pre-selection mechanisms as well as the matching mechanism applied to the subsets are randomized.

Concerning the underlying behavioral assumptions, we state that agent's in the model are purely backward looking. They do form expectations on e.g. the inflation rate but these expectations entirely depend on the past development of the inflation rate. Thus, agent's do not have the ability to collect and process massive amounts of data in order to perform (perfect) forecasts that guide their decisions. Moreover, agents also do not use any optimization procedures to follow their needs and to interact in a fully rationale way. Instead, they are boundedly rational and decision making is largely based on rules of thumb and heuristics. Our aim is to model agents that are restricted in their decision-making capabilities but still have to cope with a relative complex world. Furthermore, the current version of the model does not include any learning capabilities of agents, thus, the decision rules do not alter over time. Agents do know their own state variables but not those of other agents.

Concerning the exit and entry of agents, only corporations, i.e. firms and banks can go bankrupt. In such a case of a default of an agent, all its connections to other agents and to the network of claims are resolved appropriately until the agent has, again, a state that equals the state at its initialization. So, the agent-object does not vanish, nor is it deleted but when it reenters the market after a random amount of time and under certain preconditions it operates like a new firm or bank agent.

Finally, there are no external sources used as input during run-time.

The remainder of this chapter covers the description of the behavior of each type of agent in more detail.

4.2.3 Sequence of Simulated Economic Activity (Pseudo Code)

In this section, we show the economic activities as they occur during the simulation process. This should impart a rough idea of the functionality of the underlying agent-based macro-model and its consisting parts. The rest of the section describes these parts in more detail. The corresponding source code can be found in appendix A on p. 195 ff. (`Simulation.scala`). Note, that the provided code already includes the extensions concerning shadow banking activity used in chapter 5. The simulations consist of the following parts:

1. Start economic interaction of settlement period t ($t = 1, \ldots, 3000$)
   - Banks settle their overnight/short-term interbank liabilities (if any)
   - Banks settle their overnight/short-term standing facility liabilities with the CB (if any)
   - Banks set up repos with CB of maintenance period (if new periods starts)
2. Real sector activity (planning phase)
   - Reactivation of firms (if any)
   - Firms determine their production target
   - Firms determine their offered wage
   - Firms determine their credit demand (external financing)
   - Firms send credit requests to banks
   - Firms announce vacancies
   - Firms fire employees if they face an overproduction
3. Government pays unemployment benefit to unemployed HH

4. Real sector activity (production phase)
   - Unemployed HH search for a job / firms hire workers in case of a match
   - Firms produce and offer their bundle of goods
   - HH plan and conduct consumption
5. Real/public sector debt obligations
   - Firms pay wages and meet their debt obligations (risk for firm default due to illiquidity)
   - Government pays principal/interest on outstanding bonds
6. End of settlement period t
   - Test for firm defaults due to insolvency (annual report)
   - Banks repay intra day liquidity (IDL) to the CB (if any)
   - Banks conduct interbank lending (overnight; if necessary)
   - Banks use standing facility of the CB (if necessary)
   - CB pays interest on reserves
   - Banks determine their profit / pay taxes (if any) / pay dividends to HH (if any)
   - Test for insolvencies of banks (annual report)
   - Government bail out of systemically important banks
7. Monetary policy decisions
   - CB sets target rate and corresponding interest environment
   - CB/Supervisor set regulatory requirements (Basel III accord)

4.2.4 Start Economic Interaction of Settlement Period

4.2.4.1 Relationship Bank

The initial bilateral relationships between bank b (with $b = 1, \ldots, B$) and real sector agents are assigned randomly, i.e. each household and firm chooses a bank where it places its deposits and requests loans. These relationships do only change in the case of a default of an agent. In the case of a bank default, all clients of the insolvent bank randomly choose a new bank and if a new founded bank enters the market, clients of other banks have a small probability to switch. New firms also choose their banks randomly. The same holds for the ownership relationships since firms and banks are owned by households. Furthermore, we suppose that all economic transactions are conducted by only using scriptural money, i.e. there exist no banknotes (cashless economy).

Assets	Liabilities
Bank Deposits ($D_{G,t}$)	Public Debt ($B_{G,t}$)
CB Deposits ($D_{G,t}^{CB}$)	Equity ($E_{G,t}$)
Total Assets ($TA_{G,t}$)	

(a) Balance Sheet 1: Example Government

Assets	Liabilities
Business Loans ($BL_{b,t}$)	Retail Deposits ($RD_{b,t}$)
Wholesale Loans ($WL_{b,t}$)	Gov. Deposits ($GD_{b,t}$)
Gov. Bonds ($GB_{b,t}$)	Wholesale Liab. ($WO_{b,t}$)
Interest Receiv. ($IR_{b,t}$)	CB Liabilities ($CBL_{b,t}$)
CB Reserves ($R_{b,t}$)	Equity ($E_{b,t}$)
Total Assets ($TA_{b,t}$)	

(b) Balance Sheet 2: Example bank b

Figure 4.2: Balance sheet structure of government and banks

4.2.4.2 Public Debt

At the beginning of every simulation of the overdraft economy, the government brings money into the system by issuing bonds ($B_{G,t}$ and $GB_{b,t}$ increase) and selling them to the commercial banks and the central bank (CB) which pay by crediting the government's accounts ($D_{G,t}$ and $GD_{b,t}$ increase) [for the source code see appendix A, p. 409 (`issueNewGovBonds`)]. The bonds have a face value of 1000 monetary units and a duration of 5 years. The fix annual coupon orientates at the target rate of the central bank in period t (i_t^*), and lies slightly (15 basis points) above it [Choudhry (2010)]. The present value of each bond is determined by its clean price (neglecting accrued interest) using the standard textbook formula from Bodie et al. (2010) [source code can be found in appendix A on p. 403 (`case class govBond`)]

$$p_{k,t}^{clean} = \frac{\left(\frac{2+i_t^*}{2}\right)^{-n_{k,t}+\frac{\Omega_{k,t}}{\Upsilon_{k,t}}} \cdot FV_{k,t}\left[i_t^* + c_k\left(\left(\frac{2+i_t^*}{2}\right)^{n_{k,t}} - 1\right)\right]}{i_t^*} - \frac{c_k \Omega_{k,t} FV_{k,t}}{2\Upsilon_{k,t}} \tag{4.1}$$

where $FV_{k,t}$ denotes the face value of bond k in t, c_k the coupon, $n_{k,t}$ the amount of remaining coupon payments at t, $\Omega_{k,t}$ the amount of days since the last coupon payment, and $\Upsilon_{k,t}$ the total days in the coupon period.

The received deposits enable the government to spend and every time it runs out of deposits, it repeats this transaction in order to ensure its financial ability to act [Lavoie (2003)].[6] The issued public debt is tax-financed.

4.2.4.3 Monetary Framework

The underlying monetary framework of the model follows the post-keynesian theory of endogenous money [see Lavoie (2003) among others], i.e. the amount of money in the system is determined by the investment decisions of real sector agents (demand-driven) instead of the supply of the CB (supply-driven). Thus, we implement a monetary system along the lines of

[6] This leads to the fact that government bonds represent a large part of the banks' assets but this seems to be reasonable in times where the market-based non-traditional banking sector is larger than the traditional retail banking sector, e.g. in the U.S. [Mehrling (2012)].

the *UK Sterling Monetary Framework* of the Bank of England (BoE) using it as a template.[7] The orientation seems to be reasonable, since the BoE itself recently attracted attention in the field by implicitly accepting the endogenous money theory in their in-house journal, the *BoE Quarterly Bulletin* [McLeay et al. (2014a,b)].

At the heart of the UK reserve averaging scheme[8] is a *real-time gross settlement* (RTGS) system [Kelsey and Rickenbach (2014); Dent and Dison (2012); Nakajima (2011); Arciero et al. (2009)] which enables the CB to provide liquidity insurance to commercial banks via operational standing facilities (OSF) and, thus, to meet its lender of last resort (LOLR) function. This means that the settlement of a transaction between real sector agents takes place as soon as a payment is submitted into the system (real-time) and that payments can only be settled if the paying bank has enough liquidity to deliver the *full* amount in central bank money (gross settlement, i.e. no netting takes place) [Galbiati and Soramäki (2011)].

Reserve Target and Maintenance Period Since each bank has to pledge a sufficient amount of collateral for the reserves borrowed from the CB,[9] the initial endowment of reserves is efficient when it equals the bank's expected net transaction volume of the settlement day [see appendix A, p. 282 (`pledgeCollateral`)]. Hence, each bank chooses an amount of reserves that covers a fraction of its current interest-bearing deposits (i.e. liquidity that customers can potentially transfer to another bank). In our model this fraction is 1/15 according to Ryan-Collins et al. (2012) which state that this is a usual value for banks within the UK monetary system. The endowment is called *reserve target* ($R^*_{b,t}$) [see appendix A, p. 426 (`case class ReserveTarget`) and p. 288 (`setReserveTarget`)] and can be adjusted at the beginning of each maintenance period [see appendix A, p. 282 (`_currentReserveTarget`) and p. 285 (`monthlyRepoToAquireTargetReserve`)]

$$R^*_{b,t} = \frac{RD_{b,t} + GD_{b,t}}{15} \qquad \text{(see balance sheet 2).} \tag{4.2}$$

A maintenance period runs from one CB target rate decision to the next and, thus, has a duration of 4 weeks.

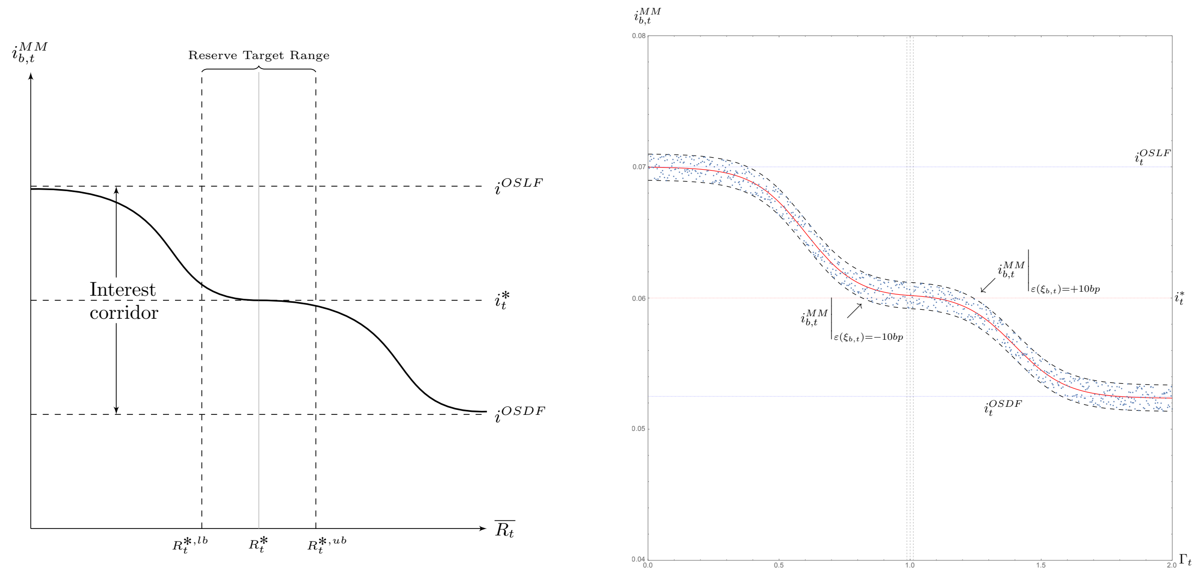
Liquidity Management Unfortunately, banks usually face an unpredictable stream of payments to execute during the settlement day meaning that it is likely for them to end up with an amount of reserves that lies either above (excess reserves) or below $R^*_{b,t}$ (reserve deficit). In

[7] A good description can be found in Bank of England (2014c); Ryan-Collins et al. (2012).

[8] Although it was suspended after the recent financial crisis in 2009 and a Quantitative Easing (QE) scheme is prevailing instead, the reserve averaging scheme can be considered as the default scheme implemented in normal times. With respect to the aim of the model, i.e. to evaluate monetary policies contribution to macro-financial stability, a scheme with a comparable setting to the pre-crises period of 2007/2008 seems to be a reasonable choice.

[9] Repos with the CB are conducted according to the international accounting standards, meaning that the bonds pledged as collateral still appear in the balance sheet of the borrower since he still faces the entire economic risk (also the coupon is paid to the borrower although the bonds are placed as collateral) [Choudhry (2010)].

order to ensure the proper functioning of the payment system, i.e. to ensure that each bank has enough reserves to conduct the payments of their customers, the CB incentivizes banks to manage their liquidity by only paying interest on the reserve holdings of a bank [see p. 431 (`payInterestOnReserves`)] if its maintenance-period average reserve holdings lie within a narrow 1%-band around $R^*_{b,t}$ (reserve target range). Hence, if a bank has met its reserve target range, it will be credited with the CB's target rate i^*_t against its average balance at the end of each maintenance period. The monetary system provides three liquidity management mechanisms for banks that they can use to compensate deviations from $R^*_{b,t}$ and to adjust their reserve accounts in such a way that they reach their reserve target range (see 4.3a). The following part describes the mechanisms of the RTGS system in more detail.



(a) Corridor System of reserves (theoret.) (b) System within the model

Figure 4.3: Interbank rate, banks' demand for reserves and the interest corridor of the CB [Bank of England (2014c); Ryan-Collins et al. (2012); Winters (2012)]

Intraday Liquidity (IDL) If a bank needs reserves during the course of the settlement day in order to process a payment of a customer because the transaction volume exceeds its current reserve balances, it can borrow the needed reserves from the CB via extreme short-term (intraday) repos [see appendix A, p. 287 (`getIntraDayLiquidity`)]. This *intraday liquidity (IDL)* has to be repaid at the beginning of the closing procedure of each settlement day [Bank of England (2014a); Dent and Dison (2012); Ryan-Collins et al. (2012)]. Thus, the provision of IDL ensures that any payment of banks' clients can be settled in real-time and on a gross basis.[10] Note, that the immediate repayment means that the CB does not provide any long-term finance for banks nor will it provide reserves or lend to insolvent banks (bailouts are exclusively conducted by the government). Of course,

[10]This mechanism implicitly assumes that there is no lack of collateral, which represents the current situation in financial markets. In such a case, the bank would simply securitize some assets to meet the need for collateral.

payments received from other banks can rebuild the reserve balances but more likely is a net in- or outflow of reserves after the settlement of intraday liquidity which requires the usage of further liquidity management mechanisms. Banks now have the opportunity to reallocate reserves through the interbank market or, if this is not possible for some reason, to use the standing facilities and borrow (deposit) the needed funds overnight from (at) the CB.

Interbank Lending Concerning the modeling of the interbank lending activity, the difficulty arose from the fact that the theoretical framework provided by the BoE only consists of a graphical representation as shown in figure 4.3a, i.e. without any mathematical description in form of a function or the like. Therefore, we decided to develop and implement such a formal representation of the interbank interest rate based on the provided logic of the BoE.

Hence, we model the interbank market as a (decentralized) over-the-counter (OTC) market which requires bank b (in need of reserves) to find a counterparty within the set of all other banks willing to lend reserves to b [Afonso and Lagos (2015)]. The conditions for overnight interbank repos are then based on bilateral negotiation about volume and interest charged. Whereas the volume depends on the counterparties current excess reserves, the costs of borrowing reserves on the interbank market $i_{b,t}^{MM}$ faced by bank b depend on three parts:

1. The first part is the CB's target rate i_t^* since its operating standing facility rates for borrowing reserves from (i^{OSLF}) and depositing reserves at the CB (i^{OSDF}) build a corridor around i_t^* and, thus, determine the overall level of the prevailing interest environment.
2. The second part is the aggregate amount of current average reserves holdings ($\overline{R_t}$) relative to the aggregate reserve targets (R_t^*), i.e. the current supply of excess reserves on the interbank market (Γ_t):

$$\Gamma_t = \frac{\sum_{b=1}^{B} \overline{R_{b,t}}}{\sum_{b=1}^{B} R_{b,t}^*} = \frac{\overline{R_t}}{R_t^*}. \tag{4.3}$$

   Γ_t serves as a measure of how far the current aggregate average reserves ($\overline{R_t}$) are away from the aggregate reserve target (R_t^*) or, put differently, the current potential for reserve reallocation. If there are a lot of excess reserves and the potential for reallocating reserves among banks is high, the interest on interbank loans ($i_{b,t}^{MM}$) is lower, i.e. close to the deposit facility rate of the CB (i^{OSDF}). If reserves are scarce, $i_{b,t}^{MM}$ is higher, i.e. closer to the rate charged for borrowing reserves overnight from the CB (lending facility rate i^{OSLF}).[11]

[11] Lavoie (2003) describes the situation in which the financial system only consists of two (highly specialized) banks whereas one of them only collects deposits while the other only grants loans to the real sector. As a result

3. The third part is a small risk premium that depends on bank b's current financial soundness $\varepsilon(\xi_{b,t})$. It is measured by its debt-to-equity ratio $\xi_{b,t}$ and it ranges between -10 and +10 basis points. Hence, realizations of $i^{MM}_{b,t}$ fall within the scope of a small band around $i^{MM}_{b,t}\Big|_{\varepsilon(\xi_{b,t})=0}$ (figure 4.3b shows this exemplary for $i^*_t = 0.06$ and $\Gamma_t \in (0,2)$).

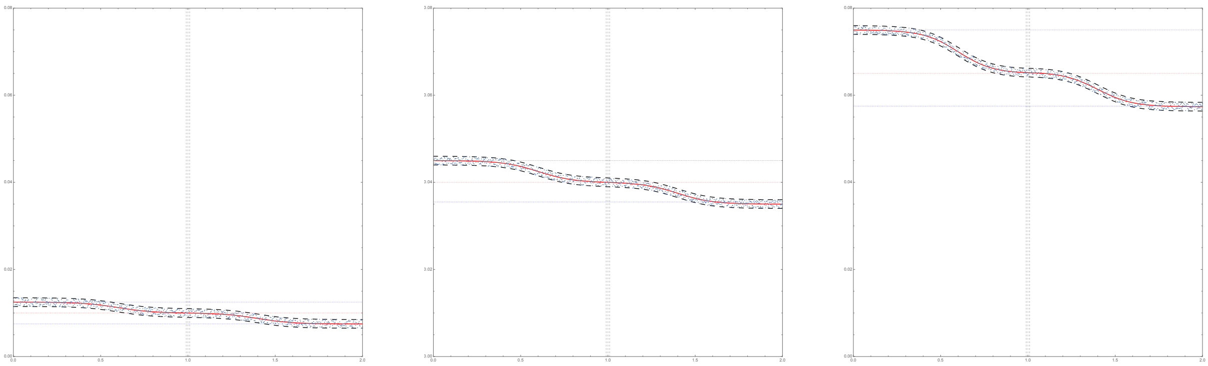
Thus, the prevailing incentive scheme shown in figure 4.3a/4.3b leads to an individual interbank rate for bank b of

$$i^{MM}_{b,t}(i^*_t, \Gamma_t, \xi_{b,t}) = \left\{ g(\Gamma_t)\left[\sigma_1 - \sigma_2 \cdot \tanh\left(\varphi\Gamma_t - \frac{3}{2}\varphi\right)\right] + \Big(1 - g(\Gamma_t)\Big)\left[\sigma_3 - \sigma_4 \cdot \tanh\left(\varphi\Gamma_t - \frac{\varphi}{2}\right)\right]\right\} - (0.06 - i^*_t) + \varepsilon(\xi_{b,t}) \tag{4.4}$$

with

$$g(\Gamma_t) = \frac{1}{2} + \frac{1}{2}\tanh\left(\frac{\Gamma_t - 1}{0.1}\right) \quad \text{and} \quad \varphi = 5 \tag{4.5}$$

[see appendix A, p. 289 (`lendOvernightFromIBM`) and p. 293 (`interestOnIBMLoans`)]. The parameters $\sigma_1, \sigma_2, \sigma_3$ and σ_4 are implemented to take the fact into account that it



(a) Low interest level (b) Mid interest level (c) High interest level

Figure 4.4: Interest corridor of the CB for varying target rate levels

seems to be a property of FED funds data[12] in the past that the CB's interest corridor or, put differently, the interest spread between borrowing from and depositing at the CB increases with the level of the target rate i^*_t. We guess that if monetary aggregates increase along with economic activity, the CB intents to provide more scope for banks to reallocate reserves among themselves through interbank lending before turning to the (more

of the incentive scheme framed by the interest corridor of the central bank's standing facilities, banks have a huge incentive to reallocate the amount of outstanding reserves among each other (through interbank lending) without involving the central bank's balance sheet.

[12]For example, the Federal Reserve Bank of St. Louis provides appropriate data sets of the federal funds rate showing such a feature (http://research.stlouisfed.org/fred2/).

expensive) standing facilities to ensure a smooth functioning of the interbank market. Therefore, we decided to (stepwise) widen the spread for higher levels of i_t^*, i.e. we define a low ($i_t^* < 3\%$), mid ($3\% \leq i_t^* \leq 5\%$), and high ($i_t^* > 5\%$) interest environment with appropriate spreads for the standing facility corridor. Figure 4.4 shows the corresponding plots for target rates lying within each of the three ranges. Therefore, the calculation of $i_{b,t}^{MM}$ in equation (4.4) is carried out accordingly by depending on $\sigma_1, \sigma_2, \sigma_3$ and σ_4. The corresponding parameterization can be found in table 4.1.

Table 4.1: Parameter sets determining the level of the CB's interest corridor

i_t^*	i_t^{OSDF}	i_t^{OSLF}	σ_1	σ_2	σ_3	σ_4
$i_t^* < 3\%$ (low)	$\max(i_t^* - 0.25\%, 0.25\%)$	$i_t^* + 0.25\%$	$\sigma_3 - 0.0025$	0.00125	0.06125	0.00125
$3\% \leq i_t^* \leq 5\%$ (mid)	$i_t^* - 0.45\%$	$i_t^* + 0.5\%$	$\sigma_3 - 0.005$	0.0025	0.0625	0.0025
$i_t^* > 5\%$ (high)	$i_t^* - 0.75\%$	$i_t^* + 1\%$	$\sigma_3 - 0.00865$	0.004	0.065	0.005

Moreover, figure 4.5 provides an overview of the possible spreads in the model whereas the area $B + C$ represents all possible locations of $i_{b,t}^{MM}$. These spreads form the incentive scheme for banks determining what to do with their liquidity, i.e. since $i_{b,t}^{Loan} > i_t^* > i_t^{OSDF}$ holds, meeting the real sector's demand for credit has the highest priority whereas lending excess reserves to peers or placing them at the CB plays a subordinated role.[13]

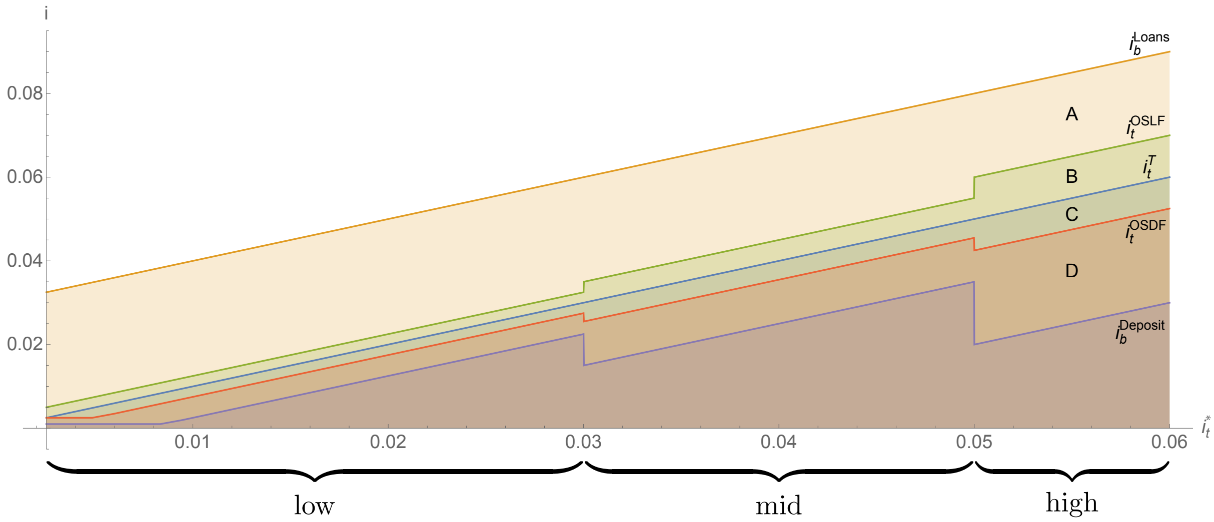


Figure 4.5: Overview on interest spreads

Finally, for the (unsecured) overnight interbank lending to take place, the borrowing bank sends a request to all peers [see p. 282 (`currentlyOfferedReservesOnIBM`)] whereas the ones with excess reserves respond with an offer consisting of the amount of reserves they

[13]This means that the modeled CB is, in general, able to stimulate banks' lending activity by lowering its target rate. In reality, this may not always be the case. The recent past has shown that the European Central Bank's (ECB) endeavor to foster lending to the real sector by providing an interest level near and even below the Zero Lower Bound (ZLB) was most widely unsuccessful due to paralyzed markets and the lack of confidence.

are willing to lend and the interest charged, i.e. $i_{b,t}^{MM}$. If the borrowing bank agrees on the offered conditions, the lending bank transfers the reserves to the borrower. At the beginning of the next settlement day, the borrower has to repay the borrowed reserves including the interest.

Operating Standing Facilities (OSF) Banks use the OSF for two reasons, either the amount of outstanding reserves (which is still only a fraction of interest-bearing deposits) is sufficient and the interbank lending is somehow distorted preventing an efficient reallocation of reserves or the transaction volume exceeds the amount of outstanding reserves or a combination of both. In such situations, the CB provides liquidity insurance for banks by means of standing facilities which can be used against collateral at the end of each settlement day [see appendix A, p. 290 (`useOSFifNecessary`)]. By charging a premium of $i^{OSLF} - i_t^*$ (discount of $i_t^* - i^{OSDF}$) on i_t^* for the usage of its lending (deposit) facility, the CB builds an interest corridor which ensures that banks seek money first in the open (interbank) money market and reallocate outstanding reserves through overnight repos with peers before turning to the CB's standing facilities [compare Lavoie (2003)].

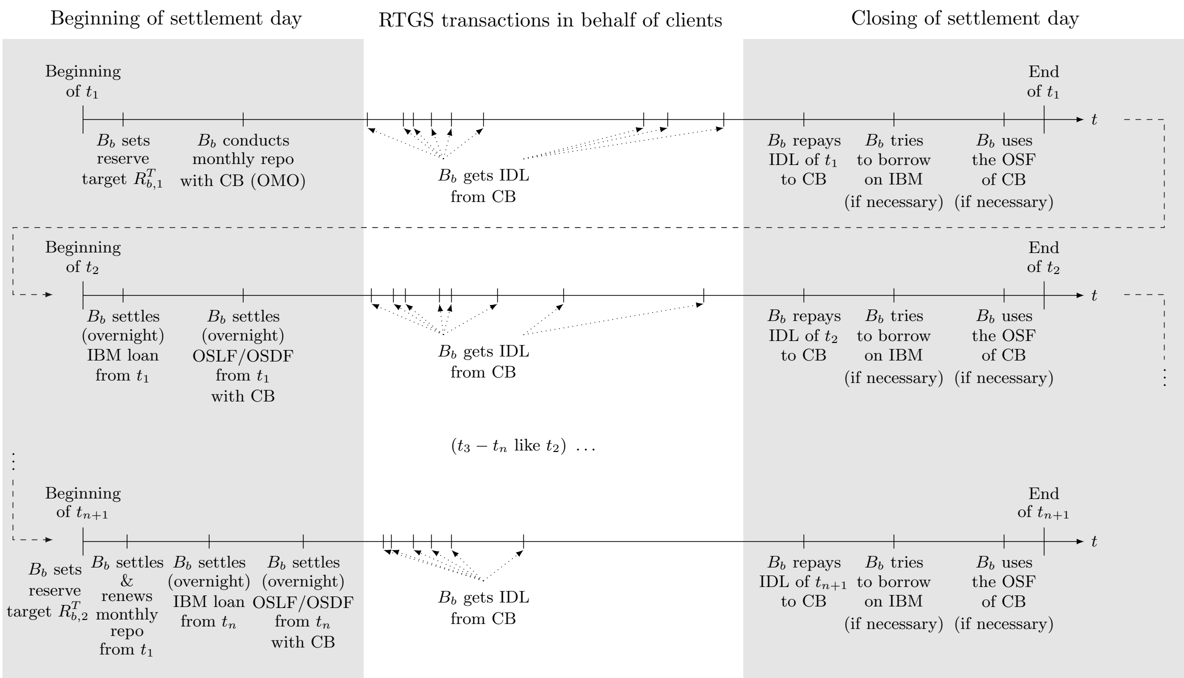


Figure 4.6: Reserve account settlement within the RTGS monetary system [see Bank of England (2014b)]

In summary, it can be said that the CB acts as settlement agent within the real time gross settlement (RTGS) system by providing settlement accounts for banks with access to intraday and overnight liquidity, i.e. the CB provides liquidity insurance [Bank of England (2014a); Dent and Dison (2012)]. In turn, these mechanisms frame the incentive for banks to internally reallocate reserves through the interbank market underpinning its central role within the monetary system since interbank rates are a key target of the CB's monetary policy implementation. Hence, banks have full control over their end-of-period reserve balances but not over the costs associated with the liquidity management mechanisms to achieve their individual reserve target range. Therefore, the underlying monetary framework empowers the CB to fully control the price for liquidity and, thus, economic activity within the model. By way of example, figure 4.6 shows how banks settle their reserve accounts with the CB during the maintenance period through the RTGS system and what options it provides.

4.2.5 Real Sector Activity (Planning Phase)

FIRM'S PRODUCTION TARGET The technology of firms follows the work of Stolzenburg (2015) where the author implements parts of the famous *Solow growth model* into an agent-based framework [Solow (1956)]. Hence, each firm f (with $f = 1,\ldots,F$) determines its production target $q^*_{f,t}$ in period t (the target stays fixed for the next quarter) according to a simple heuristic. This heuristic ensures that the capacity utilization is always slightly above the sales of the past quarter (s_f) in order to enable the firm to accommodate demand fluctuations. The target value for the firm's capacity utilization is set to

$$U^* = \frac{\sum_{s=t-12}^{t-1} s_{f,s}}{q^*_{f,t}} = 0.75, \tag{4.6}$$

i.e. $U^* < 1$ leads to an expected additional production capacity exceeding past sales $s_{f,t}$ by $\left(\frac{1}{U^*} - 1\right) s_{f,t}$. Hence, the firms production target is set according to [see appendix A, p. 371 (`determineProductionTarget`)]

$$q^*_{f,t} = \frac{\sum_{s=t-12}^{t-1} s_{f,s}}{U^*}. \tag{4.7}$$

FIRM'S OFFERED WAGE Every household (HH) h (with $h = 1,\ldots,H$) starts with an initial labor skill ψ_h that is a random draw from a truncated normal distribution, i.e. $\psi_h \in \max[0.5, \sim \mathcal{N}(2,\sigma^2)]$, and it determines both the household's individual initial productivity and its wage level. The wage per unit of labor skill $w_{f,t}$ offered by firms on the labor market also follows a simple heuristic with an update frequency of once per quarter. This means that the wage per unit of labor skill from the previous quarter, $w_{f,t-12}$, grows at the same rate as the labor productivity $\left(g_A^Q\right)$ and also takes current expected inflation (π_t^e) as well as the firm's weighted employment gap ($\Xi_{f,t}$) into account. Current expected inflation means a weighted sum of annualized monthly inflation rates of the past two years influenced by the CB's inflation target π^* times the CB's credibility parameter $\chi_\pi = 0.25$, i.e.

$$\pi_t^e = \chi_\pi \pi^* + (1 - \chi_\pi) \sum_{s=1}^{T_\pi} \pi_{t-s}^m \frac{1 + T_\pi - s}{\frac{1}{2} T_\pi (1 + T_\pi)}. \tag{4.8}$$

Moreover, $w_{f,t}$ also depends on the firm's weighted employment gap ($\Xi_{f,t}$) as an indicator of the firm's ability to hire enough workers to meet its production target given its current offered wage, i.e.

$$\Xi_{f,t} = 1 - \sum_{s=1}^{T_\Xi} \frac{q_{f,t-s}}{q^*_{f,t-s}} \cdot \frac{1 + T_\Xi - s}{\frac{1}{2} T_\Xi (1 + T_\Xi)}. \tag{4.9}$$

Thus, firm f sets its wage offered for a unit of labor skill according to

$$w_{f,t} = w_{f,t-12} \left[\exp\left(g_A^Q \right) + \pi_t^e + \omega_\Xi \Xi_{f,t} \right] \tag{4.10}$$

[see appendix A, p. 372 (`determineOfferedWageFactor`)].

Firm's Credit Demand In order to finance its planned production in advance, firms request loans $\mathcal{L}_{f,t}$ from banks with a maturity of 10 years. The volume of the requested loan mainly depends on the expected weekly labor costs that would occur if the firm would be able to hire a sufficient amount of workers to produce its previously planned production target $q^*_{f,t}$, i.e.

$$q^{-1}_{f,t} \left(q^*_{f,t} \right) w_{f,t}. \tag{4.11}$$

Here, $q^{-1}_{f,t}(\cdot)$ means the inverse production function giving the units of labor skill needed to produce a given amount of output (here the firm's production target $q^*_{f,t}$). So, the term just multiplies the needed units of labor skill with the wage offered per unit of labor skill [see appendix A, p. 374 (`expectedLaborCostsWeekly`)]. Since the weekly labor costs have to be paid during the next quarter, it has to be multiplied with twelve. Moreover, firms add a markup of 10% ($\kappa = 1.1$) on top of the expected labor costs to have an appropriate financial margin for their operational business:

$$\mathcal{L}_{f,t} = \max \left[0,\ 12\kappa \cdot q^{-1}_{f,t} \left(q^*_{f,t} \right) w_{f,t} - D_{f,t} \right], \tag{4.12}$$

Equation (4.12) shows that firms prioritize internal financing since they only have a positive demand for bank loans if their current funds ($D_{f,t}$) are insufficient to cover the expected labor costs [see appendix A, p. 375 (`determineExternalFinancing`)]. If this is the case, firm f sends a request for the loan to its relationship bank.

Firm Agents Request Bank Loans The endogenous provision of credit money to firms represents the heart of commercial banks' (traditional) business model. The granting of loans is based on a three-stage decision process:

1. After receiving a loan request from a firm [see appendix A, p. 375 (`aquireFunding`)], the bank proofs whether it would still comply with the regulatory requirements if it would grant the loan. Thus, the firm can only receive credit money if the bank's balance sheet

provides enough regulatory scope to make more loans without violating financial regulation. A violation can have several reasons and can violate either the non-risk based or risk-based capital requirements or both. Thus, the granting of the requested loan can either lead to a violation of the leverage ratio due to the loan volume or to an increase in bank's risk-weighted assets (RWA) which might become too large because the client already exhibits a very high indebtedness. In contrast, a violation of the capital buffers (capital conservation and countercyclical buffer) would not restrict any lending activity, since it would just lead to a (temporary) payout block of dividends [see appendix A, p. 295 (`proofRegulatoryRequirements`)].

2. In case of a positive finding, bank b, in a second step, decides on the interest to charge on the requested loan of firm f (i.e. $i_{b,f,t}$) by consulting a simple internal risk model to evaluate the firm's creditworthiness. Thus, $i_{b,f,t}$ moves in lock-step with the target rate i_t^* and includes a basic mark-up of 2% as well as a firm-specific risk premium. The risk premium reflects the firm's ability to generate sufficient revenues ($Rev_{f,t}$) to meet its future debt obligations ($Oblig_{f,t}$) during the fiscal year. The premium equals 10% if the firm has generated an amount of revenues that exactly equals its potential debt obligations and declines with the amount the revenues exceed the debt obligations as it decreases the risk of a credit default. The risk premium has a maximum of 15%. Hence, the offered interest on the requested loan is determined as

$$i_{b,f,t} = i_t^* + 0.02 + \underbrace{\min\left(0.1 \cdot \frac{\sum_{s=t}^{t+48} Oblig_{f,s}}{\sum_{s=t-48}^{t-1} Rev_{f,s}},\ 0.15\right)}_{\text{risk premium}}. \tag{4.13}$$

[see appendix A, p. 292 (`interestOnLoans`) as well as p. 295 (`proofCreditworthiness`)]. Note that, in the model, the actual firm-specific risk premiums are significantly lower than 10% which merely serves as a benchmark since the revenues usually exceed the firm's debt obligations. After this evaluation process, the bank responds to the loan request of the firm by offering the corresponding conditions.[14]

3. The third and final step involves the firm's evaluation on the profitability of the investment given the offered loan conditions. This decision is based on the internal rate of return which is represented by the fact that the probability to take the loan $\mathcal{L}_{f,t}$ under the offered conditions, negatively depends on the offered interest rate $i_{b,f,t}$, i.e.

$$\Pr\left(\mathcal{L}_{f,t} \mid i_{b,f,t}\right) = \max\left[1.8 - 7.5 i_{b,f,t},\ 0\right] \tag{4.14}$$

[14] There is also the possibility of only *partially* granting the requested loan, but following a survey of the ECB, these cases are only of minor importance. The decision process used here represents over 80% of decisions made by banks within the Euro area [ECB (2010)].

> [see appendix A, p. 377 (`loanIsProfitable`)]. Hence, there might be cases in which the firm does not take the loan due to the bank's high risk premium as a result of the firm's poor ability to generate a sufficient amount of revenues. In these cases of a loan rejection, the firm can only employ an amount of workers appropriate to its internal financing capacity. So, in line with the endogenous money theory, the money supply depends on the current indebtedness of the real sector (implicitly via the regulatory channel) and on the CB's current monetary policy decisions.

If the firm agrees with the offered loan conditions, the bank grants the requested loan and credits the firm's bank account and generates also a corresponding loan asset and interest receivable on its balance sheets [see appendix A, p. 294 (`case class Loan`) and p. 296 (`grantCredit2Firm`)].

Workforce Adjustment Now, the financial dimension of the planning phase is completed and firms head to the labor market to search for the appropriate amount of workers that they need to realize their planned production target [see appendix A, p. 379 (`announceCurrentJobs`) and (`affordableAdditionalLaborSkill`)].

Of course, there can also be the case in which a firm has too much employees and current production is higher than the newly planned production target, i.e. $q_{f,t} > q^*_{f,t}$. In such a case the firm fires an adequate amount of workers [see appendix A, p. 380 (`fireEmployees`)].

4.2.6 Government Pays Unemployment Benefit to Unemployed Households

Now the government pays unemployment benefit to all currently unemployed households [see appendix A, p. 419 (`payUnemploymentBenefit2HH`)]. The amount paid is adjusted every year to incorporate recent price developments in order to ensure that every household can afford a minimum amount of the good bundle [see appendix A, p. 418 (`updateUnemploymentBenefit`)].

Assets	Liabilities
Equity Stake ($ES_{h,t}$)	
Bank Deposits ($D_{h,t}$)	
Gov. Bonds ($B_{h,t}$)	Equity ($E_{h,t}$)
Total Assets ($TA_{h,t}$)	

(a) Balance Sheet 3: Example HH h

Assets	Liabilities
Inventory ($Inv_{f,t}$)	Debt Capital ($L_{f,t}$)
Bank Deposits ($D_{f,t}$)	Interest Obl. ($IO_{f,t}$)
	Equity ($E_{f,t}$)
Total Assets ($TA_{f,t}$)	

(b) Balance Sheet 4: Example firm f

Figure 4.7: Real sector agents' balance sheet structure

4.2.7 Real Sector Activity (Production Phase)

Labor Market Activity At this stage of the simulation, unemployed households start searching for a job out of a fraction ($\alpha = 0.95$) of all offered vacancies [see appendix A, p. 360

(`searchJob`)]. On the labor market, households offer their labor skill and firms search for an amount of workers that satisfies their specific labor skill demand. If there are any matchings, i.e. if the household faces vacancies in its currently observed subset of all vacancies that demand at least $\psi_{h,t}$, it is hired by a random firm from this individual subset and stays unemployed otherwise [see appendix A, p. 379 (`employHH`)].

PRODUCTION OF GOODS The production function for the weekly output faced by firm f ($q_{f,t}$) is of the Cobb-Douglas-type and depends on the aggregate labor skill currently employed by firm f ($\Psi_{f,t}$) as input and on the technology parameter A_t representing technological progress. Thus, the labor productivity of households grows at a constant exogenous rate of $g_A = 0.012$ annually (or $g_A^Q = 0.003$ per quarter), i.e. is adjusted every quarter (every 12 weeks) according to

$$A_t = A_{t-12} \exp\left(g_A^Q\right). \tag{4.15}$$

Hence, firms produce the amount of goods according to their production function of

$$q_{f,t} = \left(A_t \Psi_{f,t}\right)^{1-\alpha} \qquad \text{(with } \alpha = 0.2\text{)} \tag{4.16}$$

while it depends on the firm's ability to hire enough workers on the labor market whether it is able to meet its production target or not [see appendix A, p. 378 (`productionFunction`) and p. 382 (`produceGood`)]. Note, that one unit of the produced good represents a whole bundle of goods in order to also be able to consume continuous instead of just discrete amounts of the good.

PRICE SETTING To set the retail price for a unit of the produced bundle of goods, firms add a markup on expected unit costs ($\mu > 1$) and account for expected inflation (π_t^e)

$$p_{f,t} = \left(\mu + \pi_t^e\right) \cdot \frac{12 \cdot q_{f,t}^{-1}\left(q_{f,t}^*\right) w_{f,t} + \mathcal{L}_{f,t} i_{b,f,t}}{12 \cdot q_{f,t}^*}. \tag{4.17}$$

The expected unit costs consist of the expected labor costs for the production of the next quarter $\left(q_{f,t}^{-1}\left(q_{f,t}^*\right) w_{f,t}\right)$ and expenses for interest on bank loans ($\mathcal{L}_{f,t} i_{b,f,t}$). Again, $q_{f,t}^{-1}(\cdot)$ represents the inverse production function giving the units of labor skill needed to produce a given amount of output [see appendix A, p. 382 (`determinePrice`)].

Once the retail price is determined, the firm agents offer their produced goods and their inventory on the goods market [see appendix A, p. 383 (`offerGood`)]

CONSUMPTION Households plan their individual weekly consumption level ($c_{h,t}^p$) and update it once a quarter [source code can be found in appendix A on p. 362 (`planConsumption`)]. $c_{h,t}^p$

is composed of an autonomous part

$$c^a_{h,t} = 0.18 \cdot \frac{1}{F} \sum_{f=1}^{F} w_{f,t-12} \tag{4.18}$$

that co-varies with the average wage level of the firm sector from the previous quarter since it is a main driver of goods prices and the consumption level is expressed in monetary units.[15] Moreover, the planned consumption also depends on the current individual financial situation of household h, i.e. on the average weekly income of the previous quarter including received wages, interest on deposits as well as dividends on an accrual basis ($\overline{I_{h,t}}$). Households adjust their consumption plan in response to changes in the average income $\overline{I_{h,t}}$ according to the adjustment speed parameter $\eta = 0.9$:

$$c^p_{h,t} = \eta c^p_{h,t-12} + (1-\eta)\left(c^a_{h,t} + \eta \frac{\sum_{s=t-12}^{t} \overline{I_{h,s}}}{12}\right) \qquad \text{(with } \eta = 0.9\text{)} \tag{4.19}$$

[see appendix A, p. 362 (`planConsumption`)].

The actual consumption of household h in period t ($c_{h,t}$) only deviates from its planned consumption level $c^p_{h,t}$ in the case in which household h cannot afford to consume $c^p_{h,t}$ due to the lack of money or of supply. Thus, household h might be restricted by its current amount of bank deposits $D_{h,t}$ that depend on the surplus of income over expenditures since the beginning of the simulation. The household's sources of income include a mix of wages ($w_{h,t}$) and unemployment benefits ($UB_{h,s}$) (depending on how long it was unemployed until t) as well as received interest on its bank deposits ($i^D_{h,s}$). Furthermore, at the end of each fiscal year, firms and banks (partially) distribute their profits in form of dividends to the owning households ($d^F_{h,s}$ and $d^B_{h,s}$, respectively). These sources of income are tax deducted with taxes on income ($\tau^I = 0.3$) [see appendix A, p. 417 (`incomeTax`)], on capital gains ($\tau^{CG} = 0.25$) and on consumption ($\tau^{VAT} = 0.2$). From these sources of income, the household's expenditures consists of its previous consumption $c_{h,s}$ (until $t-1$) and the investments in a firm or bank if it is stakeholder of a corporation ($e^F_{h,s}$ and $e^B_{h,s}$, respectively). Hence, the bank deposits of household h in period t are determined as follows:

$$\begin{aligned} D_{h,t} = &\sum_{s=1}^{t}(1-\tau^I)w_{h,s} + \sum_{s=1}^{t} UB_{h,s} + \sum_{s=1}^{t} i^D_{h,s} + \sum_{s=1}^{t}(1-\tau^{CG})d^F_{h,s} + \sum_{s=1}^{t}(1-\tau^{CG})d^B_{h,s} \\ &- \left[\sum_{s=1}^{t-1}(1+\tau^{VAT})c_{h,s} + \sum_{s=1}^{t} e^F_{h,s} + \sum_{s=1}^{t} e^B_{h,s}\right] \end{aligned} \tag{4.20}$$

[15]Note, that this does not mean that households receive wages from every firm, it just ensures that the autonomous part of the planned consumption level adjusts to changes in the wage level of the firm sector.

Taking all this into account, the actual consumption of household h in period t follows

$$c_{h,t} = \min\left[D_{h,t},\ \eta c^{p}_{h,t-12} + (1-\eta)\left(c^{a}_{h,t} + \eta\frac{\sum_{s=t-12}^{t}\overline{I_{h,s}}}{12}\right)\right] \tag{4.21}$$

[see appendix A, p. 363 (`consume`)].

4.2.8 Real and Public Sector Debt Obligations

Firms Pay Out Wages Since employees work first before they get their well-deserved wages, we see the related payments to the employed household also comparable to a debt position which is why we put it in this section [see appendix A, p. 381 (`payOutWage2HH`)]. Wages are paid out at the end of each month so it doesn't have any influence on the consumer behavior just because of the fact that the payment is processed after the consumption of households in the simulation since they plan and smooth their consumption accordingly. Note, that if a firm is not able to pay all of its employees appropriately due to the lack of sufficient funds, it has to declare bankruptcy due to illiquidity reasons [see appendix A, p. 385 (`shutDownFirm`)].

Interest on Deposits Furthermore, we judge banks' interest payments on deposits in the same light [see appendix A, p. 296 (`payInterestOnDeposits`)]. The development of the interest on deposits and its dependency on the CB's target rate i^*_t can be reviewed in figure 4.5 which shows the prevailing interest environment [see also appendix A, p. 292 (`interestOnDeposits`)].

Firms Repay Bank Loans The generated revenues of the firms are now used to settle due parts of their obligations from loan contracts, i.e. they make principal payments and pay interest to the banks [see appendix A, p. 383 (`repayLoan`)]. Firms pay interest on their outstanding loans every month whereas principal payments are due once a year. The yearly principal equals 10% of the face value of the loan ($\mathcal{L}_{f,t}$) since the maturity of bank loans is 10 years. This means that the monthly interest on a bank loan declines over time. If a firm is not able to meet its debt obligations, it exits the market and all financial claims are cleared in such a way that banks have to depreciate the outstanding loans after receiving the proceeds of the liquidation of the firm's assets. Moreover, the owning households lose their share of the firm's equity [see appendix A, p. 385 (`shutDownFirm`)].

Bond Coupon Payment Also the public sector, i.e. the government, has debt obligations stemming from the issuance of government bonds. At this stage of the simulation, the government pays the yearly coupon on the outstanding government bonds and also repays the face value at maturity [see appendix A, p. 409 (`payCoupon`)]. Its expenditures for unemployment benefit to households and the interest on outstanding public debt are financed by raising income taxes on wages ($\tau^I = 30\%$), a VAT on the consumption of goods ($\tau^{VAT} = 20\%$), a corporate tax on profits of firms and banks ($\tau^C = 60\%$), and a tax on capital gains ($\tau^{CG} = 25\%$).

4.2.9 End of Settlement Period t

Real Sector At the end of the settlement day, all economic activity has been done and the time has come to evaluate on the associated results. If settlement period t is also the last settlement day of the fiscal year, the firm sector ends its settlement period by making annual reports [see appendix A, p. 394 (`makeAnnualReport`)]. If all went well and the firm f was able to meets its debt obligations during the fiscal year, it determines its profit before taxation $\Pi^{bt}_{f,t}$ as the difference of the period revenues and cost of goods sold (COGS). Revenues are calculated simply by sales (s_f) times corresponding prices of the period of production (p_f). The cost of goods sold include the amount of interest paid for outstanding loans $i^{\mathcal{L}}_f$ and labor costs of the fiscal year, i.e. the units of labor skill hired (Ψ_f) times the wage paid per unit of labor skill (w_f) [see appendix A, p. 393 (`determineProfit`)]:

$$\Pi^{bt}_{f,t} = s_f \cdot p_f - \left(i^{\mathcal{L}}_f + \Psi_f w_f\right) \tag{4.22}$$

In the case of $\Pi_{f,t} > 0$, firms are burdened by the government with a corporate tax so that the profit after tax results from

$$\Pi^{at}_{f,t} = (1 - \tau^C)\Pi^{bt}_{f,t} \qquad \text{(with } \tau^C = 0.6) \tag{4.23}$$

[see appendix A, p. 393 (`payTaxes`)]. From the remaining profit after taxation, $\theta\Pi^{at}_{f,t}$ serves as retained earnings to strengthen the internal financing capacity while the residual of $(1-\theta)\Pi^{at}_{f,t}$ (with $\theta = 0.9$) is distributed as dividends to equity holders [see appendix A, p. 394 (`payOutDividends2Owners`)].

So far, there was only the possibility for firms to go bankrupt due to illiquidity. During the process of the annual report and the updating of the balance sheet positions, it might also be the case that the firm has to shut down due to insolvency, i.e. due to insufficient or non-positive equity [see appendix A, p. 385 (`shutDownFirm`)]. Assuming that the bankruptcy of a firm happened in t, a new firm enters the market in $t + 24 + \varrho$ (where ϱ is a positive uniformly distributed integer between zero and 48) given that there exists a sufficiently large group of investors [see appendix A, p. 391 (`reactivateFirm`)].[16]

Financial Sector Now the financial sector also has to settle its accounts in order to end the settlement day. Section 4.2.4.3 already describes the following procedure for banks in great

[16]Firms which are shut down, do not vanish from the economy. In order to ensure the stock flow consistency of the model, these firms are just inactive until a new group of HH (investors) has enough capital to reactivate the firm [Dawid et al. (2014)].

detail. First of all, they have to repay the amount of intraday liquidity (IDL) if they have borrowed funds from the CB during the course of the settlement day in order to process a transaction of a customer which exceeded the bank's current reserve balances in volume [see appendix A, p. 288 (`repayIDL`)]. If this step is done, banks look at their actual reserve balance after the repayment of the IDL and evaluate its impact on their average reserve holdings over the whole maintenance period. If their current reserve balance would push their average holdings further away or not strongly enough towards their desired target range, they decide to take advantage of the liquidity management mechanisms.

Banks with a reserve deficit try to borrow an amount of reserves that would bring their average reserve holdings back to their target range using the interbank market. Banks have a huge incentive to reallocate reserves among each other before borrowing directly from the CB because this is much cheaper [see appendix A, p. 289 (`lendOvernightFromIBM`) and p. 293 (`interestOnIBMLoans`)].

Depending on the banks' ability to borrow from (or lend excess reserves to) peers, they might be forced to adjust their average reserve holdings using the standing facilities of the central bank [see appendix A, p. 290 (`useOSFifNecessary`)]. Since both liquidity management mechanisms involve just overnight loans, banks have to immediately repay the borrowed funds at the beginning of the next settlement day [see appendix A, p. 283 (`repayIBMloans`) and p. 284 (`repayOSF`)].

If the period t is also the end of the current maintenance period, the central bank pays interest on the banks that were able to achieve an average reserve holdings within their individual reserve target range. The average reserve holdings are remunerated at the central bank's target rate i_t^* [see appendix A, p. 431 (`payInterestOnReserves`)].

After the settlement of all accounts, the banking sector follows with its annual reposts [see appendix A, p. 309 (`makeAnnualReport`)]. First, every bank determines its profit before tax as a difference of the received and paid interest payments [see appendix A, p. 307 (`determineProfit`)]. The earned interest of banks include the interest on loans to firms and to other banks on the interbank market as well as the coupon payments of the government bonds, the interest on reserves from the central bank and the interest earned by depositing excess reserves using the central bank's standing deposit facility. Banks' interest expenditures include the amount paid on deposits and on the borrowed reserves from peers as well as on the usage of the standing facility of the central bank. After the identification of the fiscal year's profit, banks pay corporate taxes [see appendix A, p. 307 (`payTaxes`)]. Before they start to distribute the profit to their stakeholders, they evaluate whether they still comply with the regulatory requirements, i.e. in this case the compliance with the capital conservation buffer (CConB) imposed by the financial supervisory authority (also see 4.2.10.2 for more details on regulatory requirements). The aim of the CConB is that banks are able to use the additional (buffered) core capital to absorb unexpected losses (e.g. due to volatile valuation of collateral) in order to avoid harmful deleveraging processes. If a bank does not fulfill the requirement, it is burdened with a payout block according to the ratios shown in table 4.2 meaning that it is forced to retain (a fraction of) its (current and future) earnings instead of paying out dividends until the conservation buffer is restored [see appendix A, p. 308 (`payOutDividends2Owners` and `_currentShareOfRetainedEarnings`)].

Table 4.2: Individual bank minimum capital conservation standards of Basel III

Common Equity Tier 1 Ratio	Min. Capital Conservation Ratios (expressed as a percentage of earnings)	Unconstrained percentage of earnings for distribution
4.500% - 5.125%	100%	0%
5.125% - 5.750%	80%	20%
5.750% - 6.375%	60%	40%
6.375% - 7.000%	40%	60%
> 7.0%[a]	0%	100%

[a] The 7.0% CET1 ratio consists of the 4.5% CET1 minimum requirement and the 2.5% conservation buffer.

Of course, also financial institutions are monitored regarding their solvency at the end of fiscal year [see appendix A, p. 297 (`shutDownBank`)]. In the case of a threatening default of a systemically important bank (SIB), i.e. of a bank that has significant market share [see appendix A, p. 309 (`determineCurrentMarketShare`)] and, thus, plays a crucial role for the functioning of the payment system, the government bails out the institution in distress by waiving of deposits and the issuance of new government bonds. This behavior also leads to the fact, that in the case of a banking crisis that affects large parts of the financial system, the last bank is always bailed out by the government. Hence, the government prevents the artificial economy from a total failure of the financial system at any time [see appendix A, p. 420 (`bailOutLastBank`)]. Finally, the entry mechanism of new banks resembles the one for firms that is explained at the beginning of this section [see appendix A, p. 305 (`reactivateBank`)].

4.2.10 Monetary Policy and Financial Regulation

4.2.10.1 Monetary Policy Decisions

Since we have described how the CB uses the target rate i_t^* as key instrument to transmit monetary policy in the model (see section 4.2.4.3), we finally have to explain how decisions about its current level are made. The CB follows a standard *Taylor Rule* under flexible inflation targeting in order to ensure price and output stability. Equation (4.24) can be considered as a benchmark representing the case of conventional monetary policy which does not target any financial stability measure:

$$i_t^* = i^r + \pi^* + \delta_\pi(\pi_t - \pi^*) + \delta_x\left(x_t - x_t^n\right) \tag{4.24}$$

with $i^r = \pi^* = 0.02$ and x_t^n representing the long-term trend of real GDP measured by application of the Hodrick-Prescott-filter (with $\lambda = 1600/4^4 = 6.25$ for yearly data [Ravn and Uhlig (2002)]).

Assets	Liabilities
Loans to Banks ($L_{CB,t}$)	Reserves ($R_{CB,t}$)
Gov. Bonds ($B_{CB,t}$)	Gov. Acc. ($GA_{CB,t}$)
	Equity ($E_{CB,t}$)
Total Assets ($TA_{CB,t}$)	

Figure 4.8: Balance Sheet 5: Example CB

The scheme's inherent interest incentive for banks combined with being in full control of the target rate and, thus, of the prevailing interest corridor, enables the CB to perfectly steer interest rates, indebtedness of the real sector and, hence, economic activity [see appendix A, p. 428 (`setTargetRate`) and p. 430 (`setCentralBankInterestRates`)].

4.2.10.2 Regulatory Framework

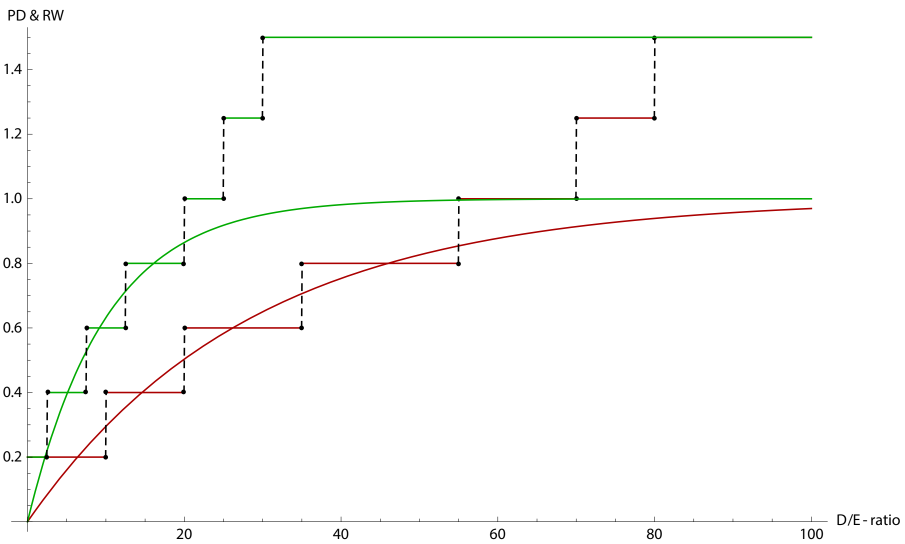


Figure 4.9: Assigned risk weights according to clients creditworthiness (red for banks, green for firms).

The financial supervisory authority agent aims to ensure the growth-supportive capacity of the financial sector by imposing micro- and macroprudential capital requirements on banks according to the current Basel III accord of the Basel Committee of Banking Supervision (BCBS) [Krug et al. (2015)].[17] So, except for the leverage ratio of 3%, all capital requirements are *risk-based*, i.e. require a minimum amount of capital in relation to the riskiness of bank b's loan portfolio measured by its individual risk-weighted assets (RWA). Positive risk weights are

[17]We do not explicitly model Basel III's liquidity requirements (LCR and NSFR), since the literature identifies the capital regulation as the most effective pillar. For further analysis on the relationship between banks' liquidity regulation and monetary policy, see e.g. Scheubel and Körding (2013).

assigned to assets resulting from loan contracts whereas government bonds have a zero-risk weight. Hence, we calculate the $\text{RWA}_{b,t}$ of bank b in t by assigning risk weights to its granted loans that depend on the current probabilities of default ($PD_{j,t}$) of its debtor firms ($j = f$) and banks ($j = b$). It follows that the $\text{RWA}_{b,t}$ are an increasing function of the debtors' debt-to-equity ratios $\xi_{j,t}$. The debtors' probabilities of default (PD) are determined by

$$PD_{j,t} = 1 - \exp\{-\rho_j \xi_{j,t}\} \quad \text{with } j \in \{f, b\}, \rho_j \in \{0.1, 0.35\}. \tag{4.25}$$

Figure 4.9 shows the relationships between the PD (solid lines) and the assigned risk weights on granted loans (staircase-shaped lines). It also shows the qualitative differences between debtor firms and debtor banks due to their differing business models meaning that a loan to a debtor bank is typically associated with a much higher debt-to-equity ratio for the same risk weight than to a debtor firm. For instance, if bank b has a loan contract with firm f in its portfolio and $\xi_{f,t} = 8$ holds, it follows approximately that $PD_{f,t} = 0.55$ and the risk weight assigned to that particular loan is 60%. The underlying source code of the mechanism in figure 4.9 can be found in appendix A on p. 437 (`def riskWeightOfGrantedLoan`).

The imposed requirements consist of a required core capital of 4.5% extended by the capital conservation buffer (CConB) of 2.5%, a counter-cyclical Buffer (CCycB) of 2.5% that is set by the CB according to the rule described in Basel Committee on Banking Supervision (BCBS) (2010) and Drehmann and Tsatsaronis (2014); Agénor et al. (2013); Drehmann et al. (2010), i.e. according to the gap of the current credit-to-GDP ratio [see appendix A, p. 428 (`determineCreditToGDPgap`)] and its long term trend determined by applying the *Hodrick-Prescott filter*[18] with a smoothing parameter of $\lambda = 1600$ [Ravn and Uhlig (2002)]:

$$CCycB_{t+1} = [(\Lambda_t - \Lambda_t^n) - N] \cdot \frac{2.5}{M - N} \tag{4.26}$$

with the credit-to-GDP ratio

$$\Lambda_t = \frac{C_t}{GDP_t}. \tag{4.27}$$

In line with the regulatory proposal of the Bank of International Settlement (BIS), we set $N = 2$ and $M = 10$. The underlying source code can be found in appendix A on p. 432 (`def setCCycB`).

Finally, we impose surcharges on systemically important banks (SIB) using the banks' market share measured by total assets as indicator for their assignment to the buckets, i.e. if

$$\frac{TA_{b,t}}{\sum_{b=1}^{B} TA_{b,t}} \leq \frac{1 + 0.3z}{B} \tag{4.28}$$

[18]In line with the BCBS, the trend here is "*a simple way of approximating something that can be seen as a sustainable average of ratio of credit-to-GDP based on the historical experience of the given economy. While a simple moving average or a linear time trend could be used to establish the trend, the Hodrick-Prescott filter is used in this regime as it has the advantage that it tends to give higher weights to more recent observations. This is useful as such a feature is likely to be able to deal more effectively with structural breaks*" [Basel Committee on Banking Supervision (BCBS) (2010)].

holds, b is assigned to bucket $6 - z$ for $z \in \{0, \ldots, 4\}$. An assignment to bucket 6 means no surcharge and to bucket 2 an extension of the risk-based capital requirement of 2.5% (the highest bucket with a surcharge of 3.5% is empty by definition; compare table **??**) [for the corresponding source code see appendix A, p. 436 (`_surchargesOnSIBs`)].

4.3 Validation of the Model

In order to validate the output data and the results of the presented agent-based macro-model, we use this section to jointly replicate a wide range of common empirical regularities like it has been done for other ACE models that are already accepted in the field of policy advice. In this context, the Keynes+Schumpeter model developed in Dosi et al. (2006, 2008, 2010, 2013, 2014, 2015) or the model described in Riccetti et al. (2015) should be mentioned since both show that (decentralized) interactions among heterogeneous agents give rise to emergent macroeconomic properties.[19] In both cases, the authors are able to validate their results by showing in detail how the model's simulated macroeconomic dynamics lead to characteristic patterns and distributions within their experimental data that coincide with real macro data. According to Fagiolo et al. (2007); Fagiolo and Roventini (2012), this is the appropriate approach to show a robust empirical validation of the model framework and, hence, of the "computational lab" leading to plausible and comparable results when testing and analyzing various policy experiments.[20]

To the best of our knowledge, the list of stylized facts presented in table 4.3 is the list to be met by ACE models for policy evaluation in the macro-finance area. It can originally be found in Dosi et al. (2014) and we chose it as a guide for the validation process of our model because it is the most complete one. Moreover, the table is extended by some additional facts found in Riccetti et al. (2015). Furthermore, we set the number of Monte Carlo simulations to be 1000, i.e. the experiments are repeated with random seeds 1, . . . ,1000, in order to *"wash away [the] across-simulation variability"* resulting from *"non-linearities present in agents' decision rules and [...] interaction patterns"*. This approach enables us to *"analyze the properties of the stochastic processes governing the co-evolution of micro- and macro-variables"*.

Going through table 4.3 step-by-step, the first macroeconomic stylized facts (SF1) would be the ability of the model to produce endogenous and self-sustained GDP growth characterized by persistent fluctuations both in nominal and real terms. Figure 4.10a shows the average log of

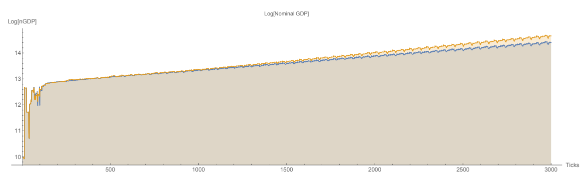
[19] Riccetti et al. (2015) state that *"[i]n particular, simulations show that endogenous business cycles emerge as a consequence of the interaction between real and financial factors: when firms profits are improving, they try to expand the production and, if banks extend the required credit, this results in more employment [;] the decrease of the unemployment rate leads to the rise of wages that, on the one hand, increases the aggregate demand, while on the other hand reduces firms profits, and this may cause the inversion of the business cycle, and then the recession is amplified by the deleveraging process"*.

[20] Dosi et al. (2014) emphasize that this way of model validation, i.e. matching a large number of stylized facts simultaneously, is the way to do it, although it is eminently costly and time-consuming. We can confirm this view.

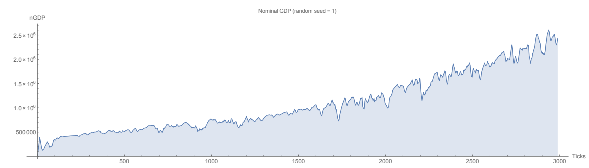
Table 4.3: Stylized facts replicated by the Keynes+Schumpeter-ACE model [Dosi et al. (2014)]

Code	Stylized fact	Empirical studies (among others)
SF1	Endogenous self-sustained growth with persistent fluctuations	Burns and Mitchell (1946); Kuznets and Murphy (1966); Zarnowitz (1985); Stock and Watson (1999)
SF2	Fat-tailed GDP growth-rate distribution	Fagiolo et al. (2008); Castaldi and Dosi (2009)
SF3	Recession duration exponentially distributed	Ausloos et al. (2004); Wright (2005)
SF4	Relative volatility of GDP/consum./invest.	Stock and Watson (1999); Napoletano et al. (2006)
SF5[a]	Pro-cyclical aggregate firm investment	Wälde and Woitek (2004)
SF6	Pro-cyclical bank profits/debt of firm sector	Lown and Morgan (2006)
SF7	Counter-cyclical credit defaults	Lown and Morgan (2006)
SF8	Lagged correlation between firm indebtedness & credit defaults	Foos et al. (2010); Mendoza and Terrones (2014)
SF9	Banking crises duration is right skewed	Reinhart and Rogoff (2009)
SF10	Fat-tailed distribution of fiscal costs of banking crises-to-GDP ratio	Laeven and Valencia (2013)
SF11[b]	the presence of the Phillips curve	Phillips (1958)

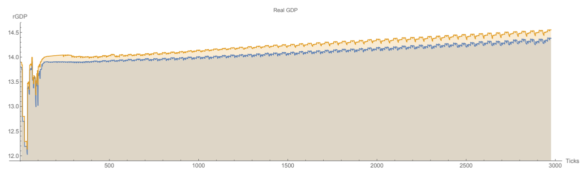
[a] In the original table of Dosi et al. (2014), aggregate R&D investments are used. We use, instead, the firm sector's requested amount of loans from banks as a proxy for their investment in the production of goods.
[b] Described as general characteristic of an economy, i.e. without explicit notion of empirical studies and found in Riccetti et al. (2015).



(a) Log Nominal GDP (avg. of 1000 runs; blue for Basel II, orange for Basel III)



(b) Nominal GDP (single run)



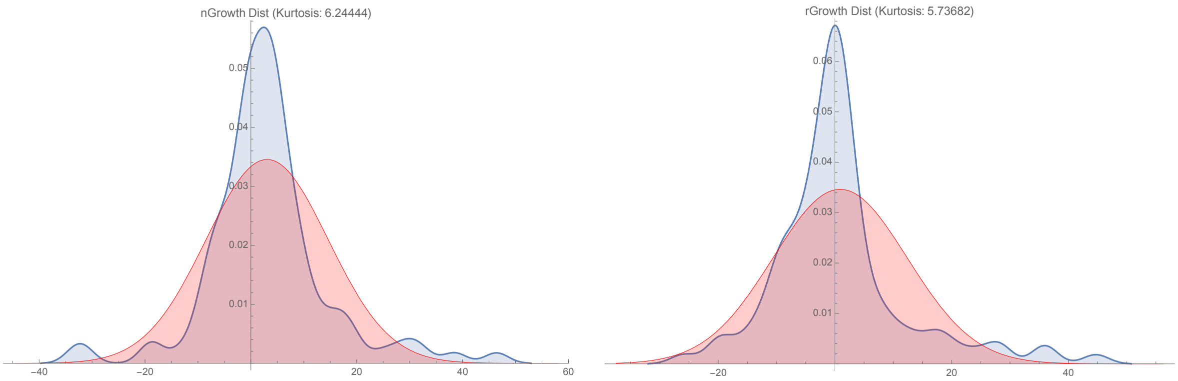
(c) Log Real GDP (avg. of 1000 runs; blue for Basel II, orange for Basel III)

(d) Real GDP (single run)

Figure 4.10: Endogenous nominal/real GDP growth with persistent fluctuations [SF1]

nominal GDP for simulations with random seeds 1, . . . ,1000 which is steadily growing whereas figure 4.10b shows exemplary the dynamics of nominal GDP of a single run. The right panel exhibits moderate fluctuations at the beginning of the simulation which are increasing with economic activity and overall size of the economy leading to business cycles including booms and deep downturns. The same holds for real GDP (see figure 4.10c/4.10d). Moreover, the comparison of both time series reveals the fact that the business cycles do not vanish when building the average of various simulation runs but are much more regular.

The second replicated stylized fact directly connects to the first one and follows the empirical



(a) Distribution of nominal GDP growth-rate

(b) Distribution of real GDP growth-rate

Figure 4.11: GDP growth-rate distribution (blue) compared to the Gaussian fit (red) [SF2]

studies of Fagiolo et al. (2008); Castaldi and Dosi (2009) where the authors have shown that real data sets of GDP-growth rates have the property of fat-tailed distributions compared to their Gaussian benchmarks. This also holds for our model in both nominal (figure 4.11a) and real terms (figure 4.11b).

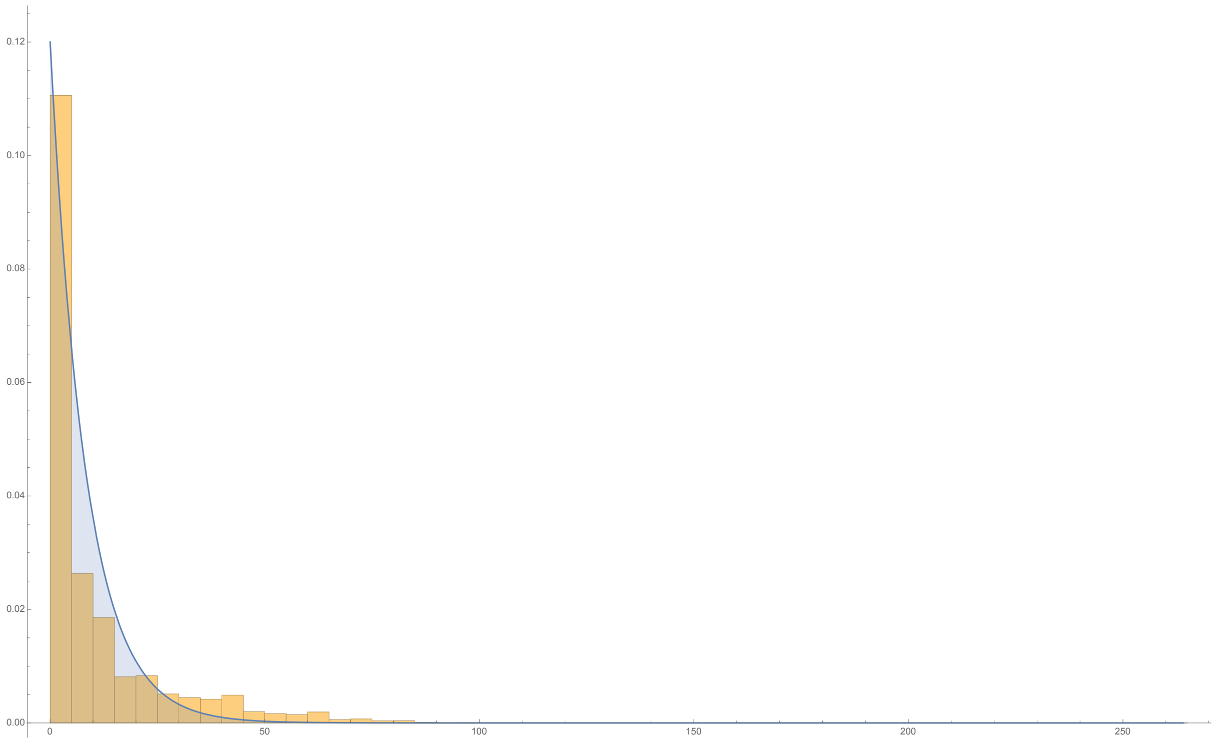


Figure 4.12: Exponentially distributed duration of recessions [SF3]
Bins represent the data from the model, blue is the exponential fit of the data.

Concerning the recessions occurring during the simulations, we can confirm that the majority lasts for rather short periods of time and that their frequency declines substantially with rising duration. Empirical data shows that they are approximately exponentially distributed which is also the case in our experimental data (see figure 4.12).

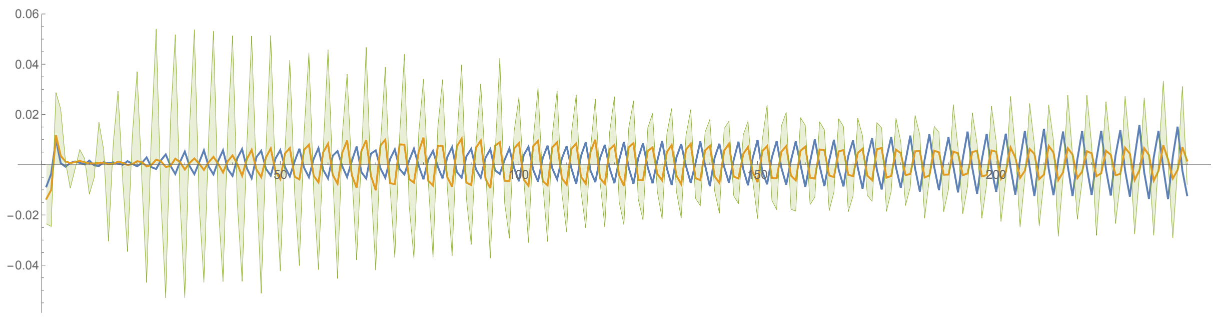


Figure 4.13: Bandpass filtered time series of GDP/consumption/investments to show their relative volatility [SF4]
Volatility of GDP (blue); of consumption (orange); of investments (green)

To verify whether our model can replicate SF4, we again follow Dosi et al. (2014) and bandpass filter the time series for GDP, consumption and firm investment in order to de-trend the data and to analyze their behavior at business cycle frequencies. As figure 4.13 shows, the data produced by our model is in line with the empirical findings since the fluctuations of consumption are slightly smaller compared to GDP while firm investments is much more volatile than output.

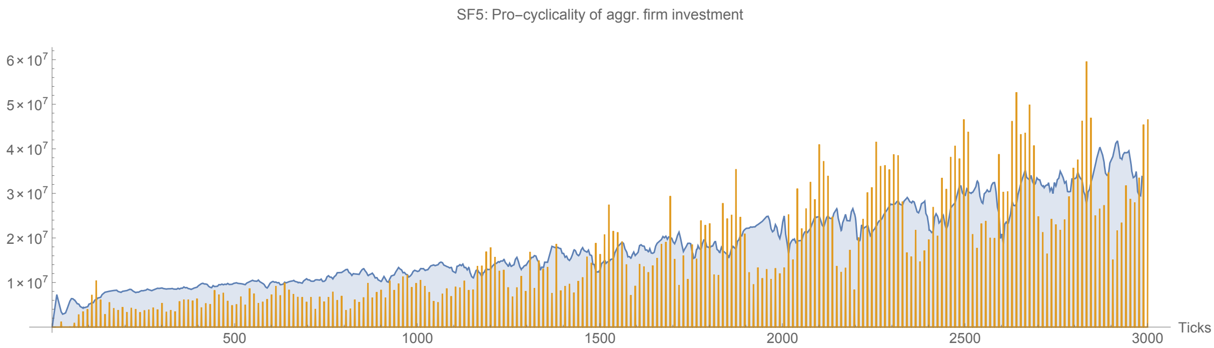
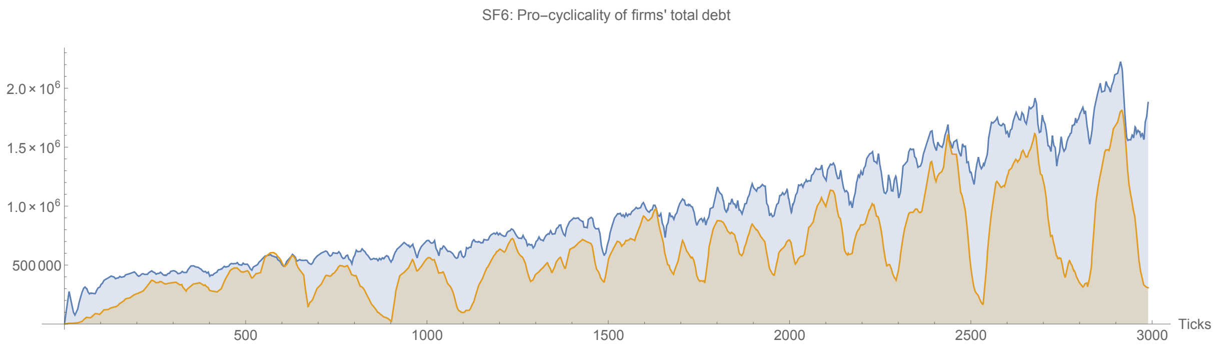


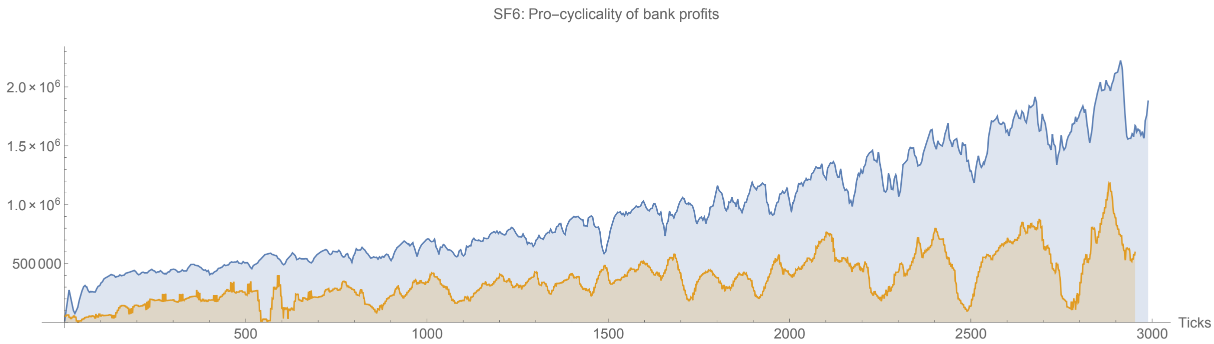
Figure 4.14: Pro-cyclicality of aggregate firm investments [SF5]
GPD (blue); Aggregate firm investment (orange)

While the stylized facts 1-4 have general macroeconomic character, the following focus on drivers of prevailing economic activity and, thus, the business cycle. This means that the pro- and counter-cyclicality of key variables is essential to ensure the proper functioning of the modeled monetary economy. Overall, they shed some light on the development of the lending activity and on the resulting financial stability dynamics over time. The first fact here is then the pro-cyclicality of firm's aggregate investment which tend to co-move with the business cycle (figure 4.14).

Moreover, Lown and Morgan (2006) have shown empirically, there exists a strong link between the total debt outstanding in the firm sector (4.15a) and the profits of the banking sector (4.15b) both being highly pro-cyclical. Hence, the lending activity co-moves with the business cycle whereas the experience from past financial crises suggests that the build-up of debt imbalances



(a) Pro-cyclicality of firms' total debt



(b) Pro-cyclicality of bank profits

Figure 4.15: Pro-cyclical lending activity [SF6]
Ordinate scale relates to GDP (blue); whereas credit related variables (orange) are scaled appropriately to emphasize their pro-cyclicality.

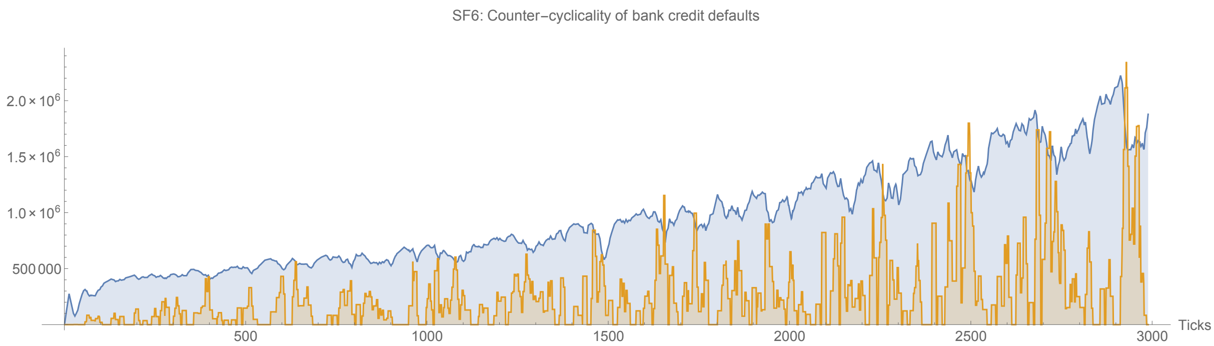


Figure 4.16: Counter-cyclical credit defaults [SF7]
GDP (blue); credit defaults are measured by loan losses of banks (orange).

leads to downturns triggered by peaks in default rates which, in turn, result in rather counter-cyclical behavior of credit defaults (4.16). Figure 4.16 shows that these facts are also features of our model and can be replicated simultaneously.

Moreover, the slightly lagged correlation between indebtedness of the firm sector and credit default rates can be replicated just as well. Figure 4.17 validates in a very clear manner that in our experimental data the build-up of real sector debt imbalances is accompanied by banks facing excessive risk of bad debt and, thus, are frequently paired with periods of financial distress translating into economic downturns.

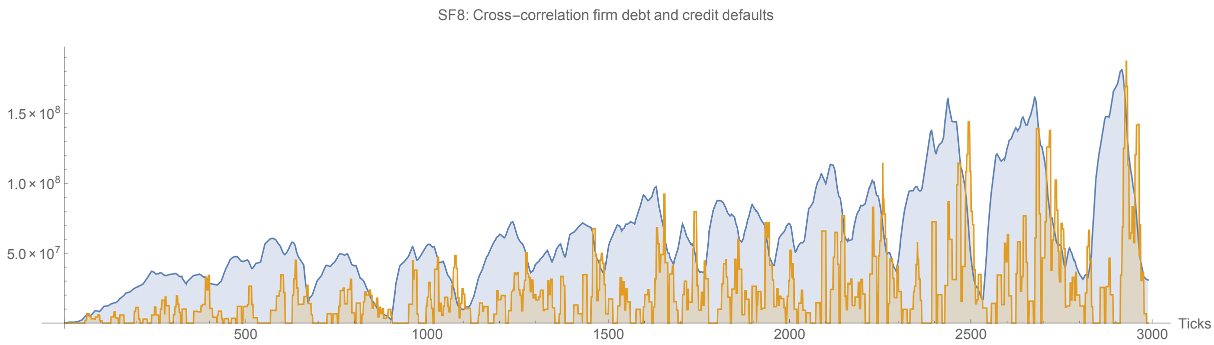


Figure 4.17: Lagged correlation of firm indebtedness and credit defaults [SF8]
Indebtedness of firm sector (blue); bad debt is measured by loan losses of banks (orange).

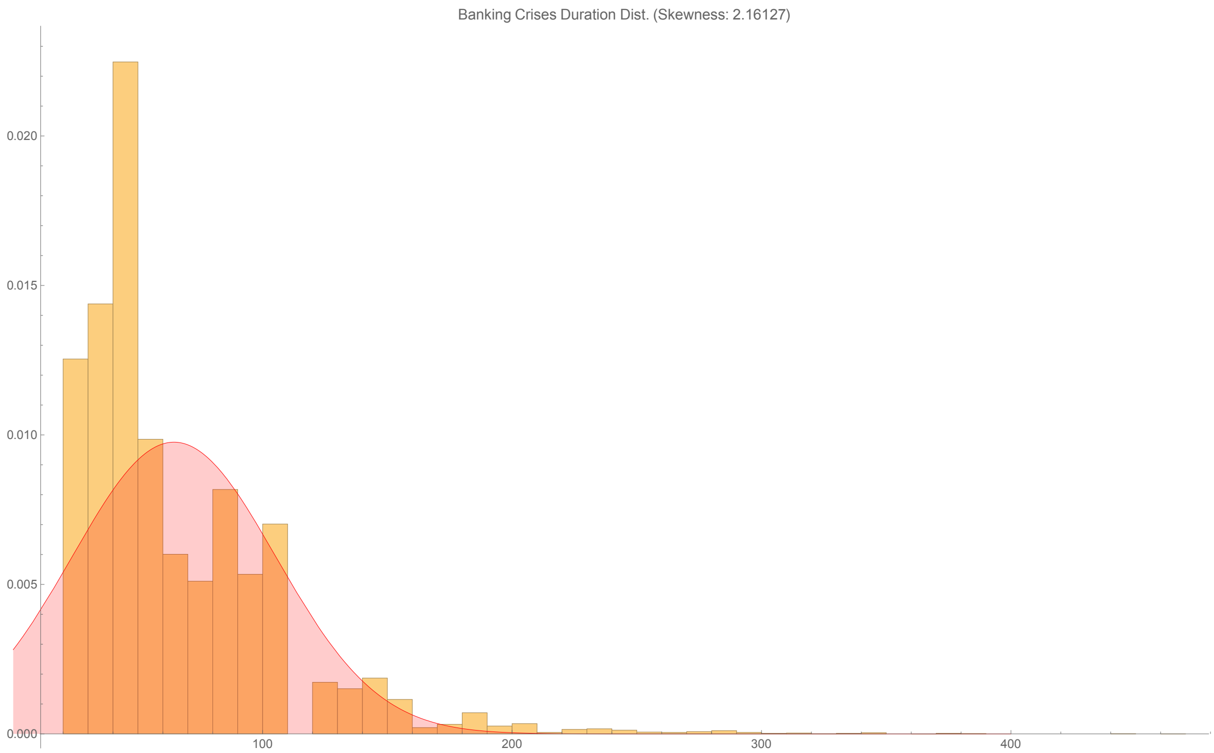


Figure 4.18: Banking crises duration is right-skewed compared to Gaussian data fit [SF9]

In order to cope with empirical regularities of financial crises data, we then define crises as periods from the first bank default until all banks B are back in their business. Thus, the empirical work of Reinhart and Rogoff (2009) suggests that the distribution of the duration of these periods is positively skewed (right skewed). This also holds for our model. Moreover, the ratio of fiscal costs-to-GDP is computed for such periods of financial distress. These fiscal or restructuring costs caused by financial crises mainly consists of recapitalization costs to stabilize the banking sector and, in reality, the distribution of the ratio is characterized by excess kurtosis (here above 12), i.e. fat tails, which is also the case in our experiments (see figure 4.19).[21] Last

[21] Laeven and Valencia (2013) define a significant support by the government if fiscal costs exceed 3% of GDP. This seems to be a reasonable choice for real data but the typical real economy of interest is considerably larger and consist of more agents compared to our small-scale ACE model. In fact, this affects the fiscal costs-to-GDP ratio since the size of our banking sector relative to GDP is much larger than in reality since our model has less

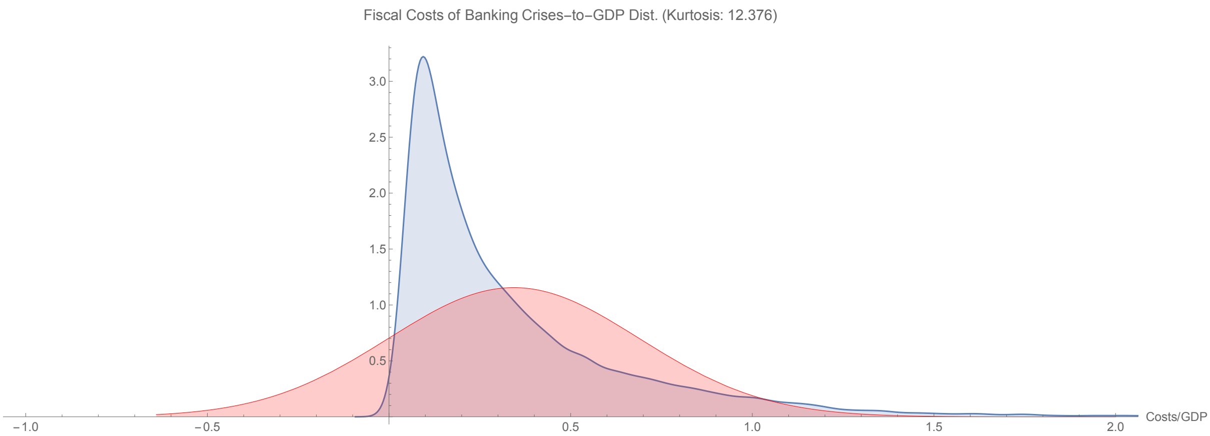


Figure 4.19: Fat-tailed distribution of fiscal costs of banking crises-to-GDP ratio [SF10]

but not least, our experimental data exhibits a Phillips curve (figure 4.20).

In summary, the replicated stylized facts shown above indicate the relevance of leverage cycles and credit constraints on economic performance as well as the importance of the government in its function as a compensating and balancing institutional agent providing stability to the economy. Furthermore, this section shows that the presented macro model is generally able to serve as framework for the analysis of research questions concerning banks lending activity, leverage, financial crises as well as monetary and macroprudential policy.

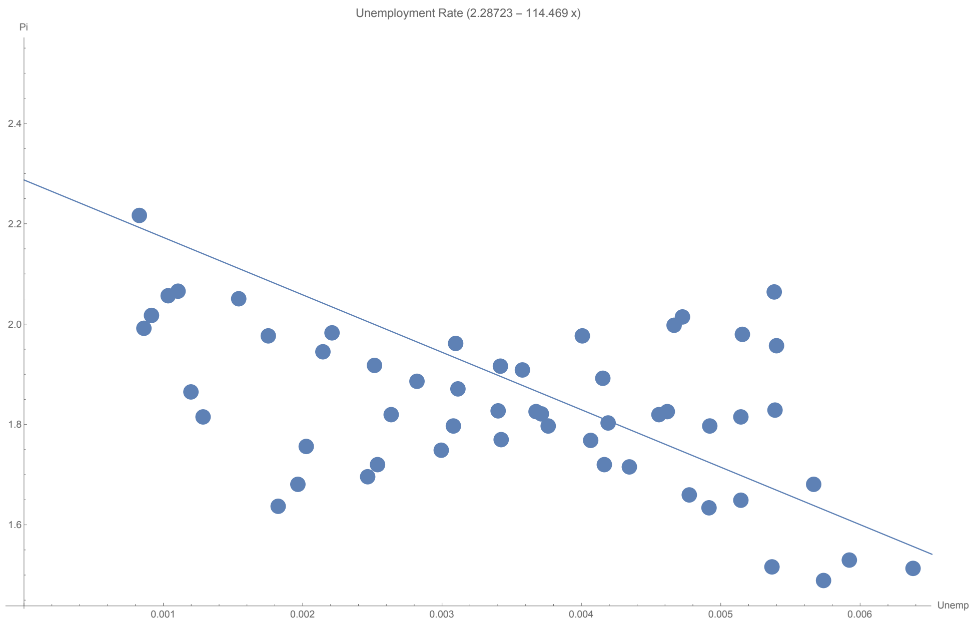


Figure 4.20: Phillips curve [SF11]

agents to contribute to GDP. Hence, this can lead to years in which the fiscal costs are twice or three times as high as GDP. These relatively high ratios might be comparable to the situation in small countries with large financial systems like Iceland or Ireland where the fiscal costs have reached very high levels amounting even to multiples of GDP.

4.4 Design of Experiments (DOE)

Mishkin (2011) states that, despite the occurrence of the recent financial crisis and the theoretical deficiencies of general equilibrium frameworks, there is no reason to turn away from traditional new keynesian theory of optimal monetary policy, which caused us to do so in order to measure monetary policy outcomes. According to Verona et al. (2014), the assessment of the research question formulated above entails three main issues, i.e.

(i) determination of a financial stability measure,

(ii) modeling of the CB's policy response,

(iii) determination of a criterion for policy effectiveness.

Then policy outcomes will be compared in order to show whether crisis mitigation is better achieved with a monetary policy reaction or with financial regulation, i.e. macroprudential policy.

In this regard, the indicator in use for the measurement of financial instability to which the CB should respond to, is, indeed, a crucial issue. Woodford (2012) suggests that, from a theoretical point of view, using financial sector's leverage would be the natural choice. However, Stein (2014) argues that this would be hard to measure in a comprehensive fashion and one should better stick to a broader measure of private sector leverage. He points to the work of Drehmann et al. (2012); Borio and Drehmann (2009); Borio and Lowe (2002) which show that the ratio of credit to the private non-financial sector relative to GDP (the credit-to-GDP ratio) has considerable predictive power for financial crises. Hence, we try to shed some light on these issues by comparing policy outcomes of CB's response to either a measure for the financial sector's leverage which targets a prudent balance sheet structure of the aggregate banking sector [Adrian and Shin (2008a,b)] as well as to the credit-to-GDP ratio.

In order to address (ii), the following paragraph describes the implementation in detail:

- In line with the literature on early warning indicators for financial crises [Babecký et al. (2013); Gadanecz and Jayaram (2009)], we construct a *composite financial stability indicator (CFSI)* and augment the standard instrument rule by the deviation from its target value $CFSI^*$:

$$i_t^* = i^r + \pi^* + \delta_\pi(\pi_t - \pi^*) + \delta_x\,(x_t - x_t^n) + \delta_s(CFSI_t - CFSI^*) \qquad (4.29)$$

with $i^r = \pi^* = 0.02$ and x_t^n representing the long-term trend of real GDP measured by application of the Hodrick-Prescott-filter (with $\lambda = 1600/4^4 = 6.25$ for yearly data [Ravn and Uhlig (2002)]). Moreover, the $CFSI_t$ consists of the average D/E-ratio of banking sector as well as of the inverse of banks' average equity ratio

$$CFSI_t = \log\left(\frac{1}{b}\sum_{i=1}^{b}\xi_{B_i,t}\right) + \log\left(\frac{1}{\frac{1}{b}\sum_{i=1}^{b}\frac{E_{B_i,t}}{RWA_{B_i,t}}}\right). \tag{4.30}$$

As a benchmark, we set $CFSI^* = 6$ which corresponds to an average D/E-ratio in the banking sector of 33 (or an average leverage ratio of approx. 3%) as well as an average equity ratio of 7% core capital, both representing current thresholds of the Basel III accord. This setup leads to an increasing (declining) CFSI if the banking sector gets more fragile (stable) over time.

- In experiments in which the CB responds to jumps in the credit-to-GDP ratio,[22] target rate decisions are guided by

$$i_t^* = i^r + \pi^* + \delta_\pi(\pi_t - \pi^*) + \delta_x\left(x_t - x_t^n\right) + \delta_s\left(\Lambda_t - \Lambda_t^n\right) \tag{4.31}$$

with Λ_t as defined in eq. (4.27). The credit-to-GDP gap $\Lambda_t - \Lambda_t^n$ is determined by the difference between the current credit-to-GDP ratio and its long-term trend measured by means of applying the *Hodrick-Prescott filter* with a smoothing parameter $\lambda = 6.25$ [Ravn and Uhlig (2002)].

Concerning (iii), there are two main traditions in the literature. The first one is to search for the policy that maximizes social welfare, i.e. maximizes the utility function of HH, but according to Verona et al. (2014) this approach has some drawbacks which is why we go for the second one, that is, the policy that best achieves the objective at hand by minimizing loss functions. For the sake of clarity, we take up the approach of Gelain et al. (2012) and differentiate between *(macro)economic* ($L^{MS}_{\delta_s,k,m}$) and *financial stability* ($L^{FS}_{\delta_s,k,m}$). Hence, we define two loss functions in order to easily evaluate outcomes in both dimensions whereby the former is usually defined as the weighted sum of the variances of inflation, output gap and of nominal interest rate changes,[23] i.e.

$$L^{MS}_{\delta_s,k,m} = \alpha_\pi\overline{\mathrm{Var}(\pi_{\delta_s,k,m})} + \alpha_x\overline{\mathrm{Var}(x_{\delta_s,k,m})} + \alpha_i\overline{\mathrm{Var}(i_{\delta_s,k,m})} \tag{4.32}$$

with $\alpha_\pi = 1.0$, $\alpha_x = 0.5$, $\alpha_i = 0.1$ [Agénor et al. (2013); Agénor and Pereira da Silva (2012)]. The latter, however, addressing financial stability ($L^{FS}_{\delta_s,k,m}$) is defined in terms of the weighted sum of the average burden for the public sector of a bank bailout measured as the fraction of the average bailout costs for the government and the average amount of bailouts ($\overline{\zeta_{\delta_s,k,m}}$) as

[22] This has also been analyzed using DSGE models in Cúrdia and Woodford (2010) and Quint and Rabanal (2014).

[23] For a deeper discussion of the effects of central bank's interest rate smoothing, see Driffill et al. (2006).

well as the average amount of bank and firm defaults ($\overline{\rho_{\delta_s,k,m}}$ and $\overline{\gamma_{\delta_s,k,m}}$, respectively), i.e.

$$L^{FS}_{\delta_s,k,m} = \alpha^{FS} \left(\overline{\zeta_{\delta_s,k,m}} + \overline{\rho_{\delta_s,k,m}} + \overline{\gamma_{\delta_s,k,m}}\right) \tag{4.33}$$

with $k \in \{CFSI,\, \Lambda_t - \Lambda^n_t\}$, $\alpha^{FS} = 0.01$ and
$m \in$ {Basel II (macroprudential policy off), Basel III (macroprudential policy on)}. Hence, the analyzed scenarios add up to 4 since the variables m and k have only two values. While m determines the prevailing regulatory regime, i.e. whether banks have to comply with regulatory requirements in line with the Basel III accord or with its predecessor, namely Basel II, variable k determines the central bank's response to the financial stability measure, which can either be the CFSI or the credit-to-GDP gap. For each of these 4 scenarios, we basically follow the idea of the recent *"model-based analysis of the interaction between monetary and macroprudential policy"* of the Deutsche Bundesbank [Deutsche Bundesbank (2015)] which searches for optimal values of the coefficients in the monetary policy rule using three differing DSGE models including a macroprudential rule. We apply the approach by doing a *grid search* within the three-dimensional parameter space spanned by $\delta_\pi \in$ [1,3], $\delta_x \in$ [0,3] and $\delta_s \in$ [0,2][24] (with a step size of 0.25) whereby the cases of $m =$ Basel II (no macroprudential policy) and $\delta_s = 0.0$ (no leaning against financial imbalances of the CB) represent the benchmark, i.e. a situation that is comparable to the pre-crisis period.

The analysis procedure for raw data produced by the model includes the following steps:

A. Grid Search We perform a grid search for minimum values of the loss function L and visualize the results using heat maps. Thus, the performance of parameter combinations or data points is evaluated in counterfactual simulations of the underlying agent-based (disequilibrium) macroeconomic model[25] using a set up of 125 HH, 25 firms and 5 banks.[26] Considering every combination of δ_π, δ_x, δ_s, m and k, this adds up to 4212 data points in total. We then conduct Monte Carlo simulations for every data point with random seeds $1, \ldots, 100$[27] while every of the 100 runs has a duration of $T = 3000$ periods or ticks. According to our setting, this duration can be translated into approximately 60 years since every tick represents a week and every month has 4 weeks which adds up to 48 weeks for an experimental year. Hence, for the analysis, we take the last 50 years (2400 periods) into account and use the first 600 periods as initialization phase.

B. Identification of Minimum Losses In a second step, we identify areas of best performing parameterizations (minimum losses) and of the corresponding policies.

[24] The monthly report of March 2015 of the Deutsche Bundesbank states this parameter space as commonly used for DSGE models and refers to Schmitt-Grohé and Uribe (2007) in this regard.
[25] The ACE Model is programmed in Scala 2.11.7.
[26] We have also conducted experiments with a set up which follows Riccetti et al. (2015) implementing 500 households, 80 firms and 10 banks but the results where qualitatively the same.
[27] We chose only 100 because of the pure amount of data points to simulate and the corresponding time restrictions.

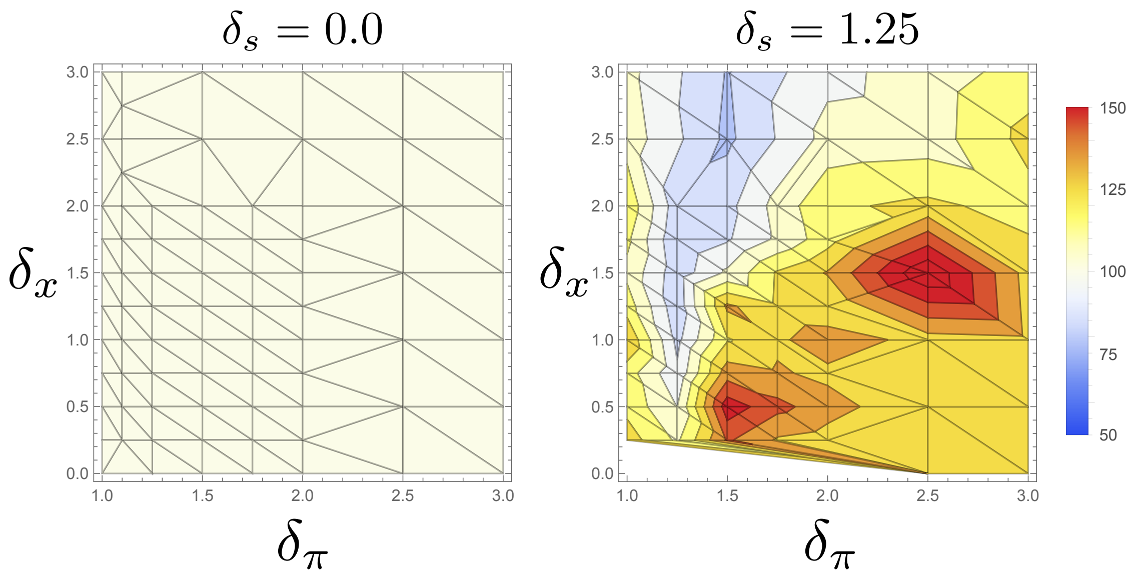


Figure 4.21: Example for benchmark (left panel) and non-benchmark losses (right panel)

After the generation of the raw output data, we compute the values for the two loss functions $L^{MS}_{\delta_s,k,m}$ and $L^{FS}_{\delta_s,k,m}$. In order to represent the results in two-dimensional space, we additionally compute a combination of $L^{MS}_{\delta_s,k,m}$ and $L^{FS}_{\delta_s,k,m}$:

$$L = \alpha_L L^{MS}_{\delta_s,k,m} + (1-\alpha_L) L^{FS}_{\delta_s,k,m} \tag{4.34}$$

where α_L represents the weight of the central bank's policy goals. With $\alpha_L = 1$, the CB would just consider its traditional goals of price and output stability whereas $\alpha_L = 0$ would be a solely focusing on financial stability issues. We show relative values for L in panels with δ_π on the abscissa and δ_x on the ordinate for every combination of δ_s, m, k and α_L. Thus, we get $|m| \cdot |k| = 4$ matrices containing $|\delta_s| \cdot |\alpha_L| = 45$ panels. To put the computed results in relation with the benchmark losses (representing 100%), all losses are expressed in percent of their corresponding benchmark loss using a heat map. The displayed range varies from 50% (blue) to 150% (red) of the benchmark. To make this clear, figure 4.21 shows a benchmark panel (left panel) and a non-benchmark panel (right panel). Of course, the benchmark panel does not show any blue or red color since it shows a comparison with itself (all data points represent exactly 100%). However, the data point $(\delta_\pi = 2.5, \delta_x = 1.5, \delta_s = 1.25)$ in the right panel lies in a dark red area which means that, according to our experiments, the underlying policy leads to a much higher loss relative to the corresponding benchmark loss $(\delta_\pi = 2.5, \delta_x = 1.5, \delta_s = 0.0)$. Now, we search for all data points lying in dark blue spots to identify minimum losses. The reader can find the results of the grid search for the four analyzed scenarios in figures 4.23, 4.25, 4.27 and 4.29.

C. Evaluation of Performance Gains We use violin plots to evaluate how performance gains (minimum losses) can be achieved via policy adjustments and in which way better performing policies differ from the benchmark. These kind of plots extends the usual descriptive statistics of box plots with density plots in order to provide a visualization of the whole distribution of the data. The width of the (rotated and mirrored) density plot represents the frequency of occurrence.

Hence, we show a violin plot for each part of the two loss functions $L^{MS}_{\delta_s,k,m}$ and $L^{FS}_{\delta_s,k,m}$ and, in every plot, we compare the distribution of the parts under the adjusted policy associated with the gain in performance (red density plot) with the corresponding benchmark (blue density plot). In order to avoid a cluttered graph and for the sake of clarity, we decided to forgo the box plot and just show the two density plots in each panel. The reader can find the comparisons of the data points in the figures 4.22, 4.24, 4.26, 4.28.

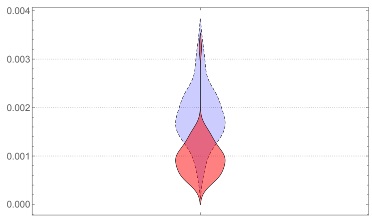
The next section presents the results of the described experiments.

4.5 Discussion of Results

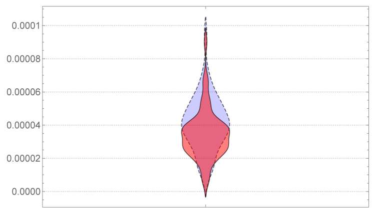
4.5.1 Scenario 1: A monetary policy response to financial sector leverage in a loose regulatory environment

Figure 4.23 shows the losses for the direct response to financial sector leverage in a rather loose regulatory environment (Basel II). If policy makers leave their focus on the traditional monetary policy goals of price and output stability ($\alpha_L = 1$; first row), “leaning against the wind” ($\delta_s \approx 1.0$) has a positive effect on these for common values of δ_π and δ_x. In terms of financial stability ($\alpha = 0.0$; 5th row), results show that such an extension of the central banks’ mandate only leads to minor improvements. This stems mainly from the already existing fragility of the system due to the lack of an appropriate regulatory environment. Of course, since there is no conflicting effect or trade-off in the case of $\delta_s > 0$, implementing an extended monetary policy which tries to incorporate also financial stability issues ($\alpha = 0.5$) still leads to a gain relative to the benchmark.

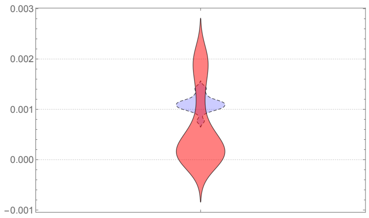
Figure 4.22 shows how the individual components of the loss functions react to the central bank response in detail. Here, the caution against the consequences of an overreacting monetary policy seem not to be valid. Indeed, the volatility in variances of the target rate increases significantly but at the same time the volatility in the variances of inflation and of the output gap decreases which seem to result in lower firm and considerably lower bank default rates. Also the tail risk for extremely high fiscal costs exhibit a large decline.



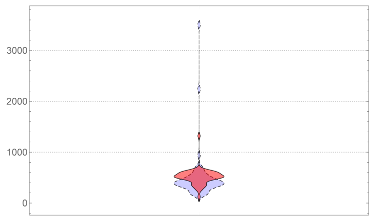
(a) Var($\pi_{\delta_s,k,m}$)



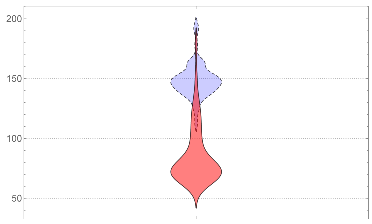
(b) Var($x_{\delta_s,k,m}$)



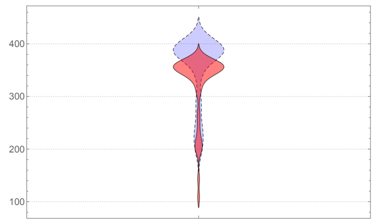
(c) Var($i_{\delta_s,k,m}$)



(d) $\zeta_{\delta_s,k,m}$ (fiscal costs)



(e) $\rho_{\delta_s,k,m}$ (bank defaults)



(f) $\gamma_{\delta_s,k,m}$ (firm defaults)

Figure 4.22: Minimum loss given a response to CFSI under Basel II; $\delta_\pi = 1.1; \delta_x = 0.25; \delta_s = 1.75; \alpha_L = 1$.
The blue, dashed distribution represents the benchmark scenario while the red, solid one represents the counterfactual scenario.

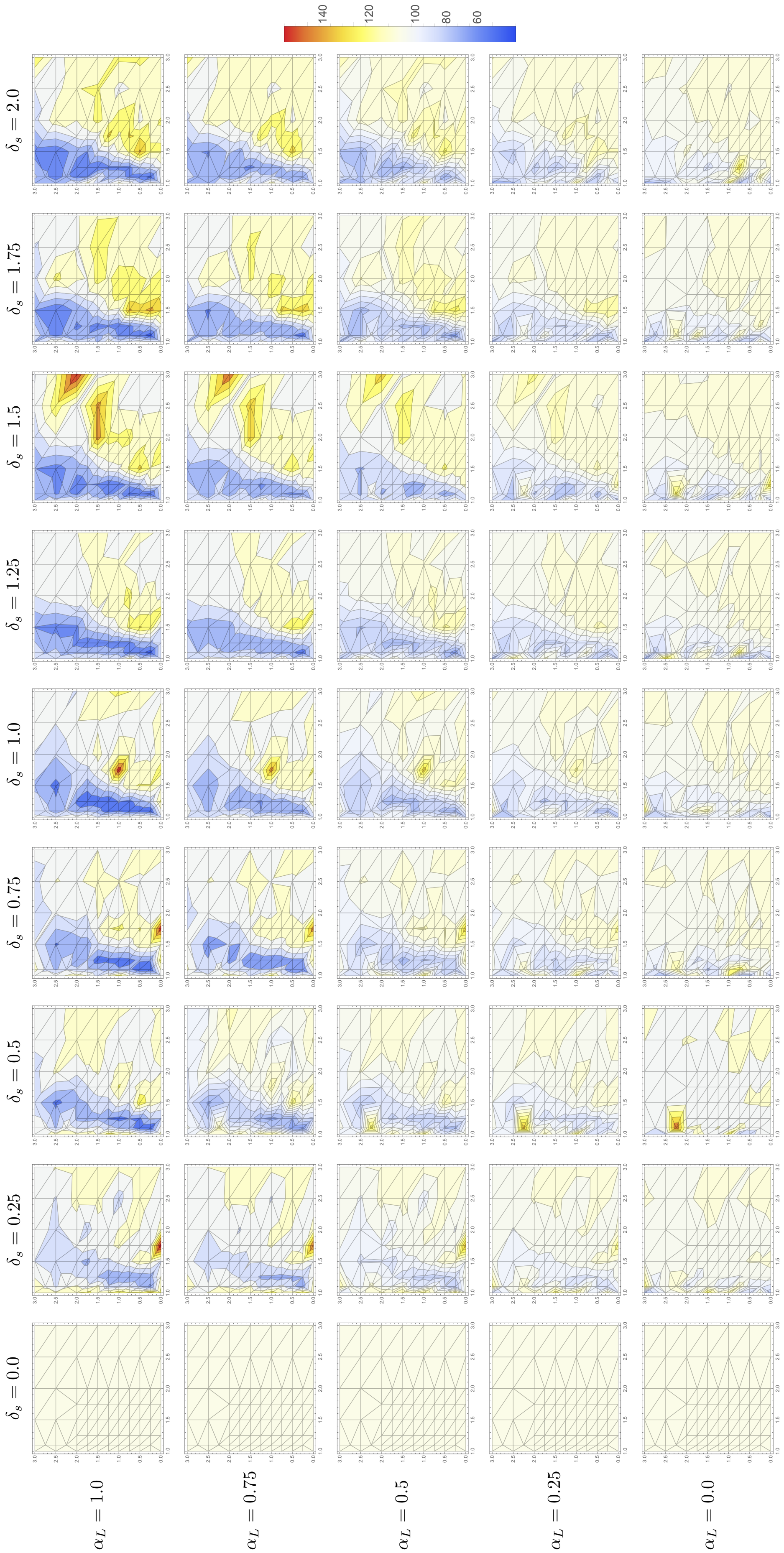
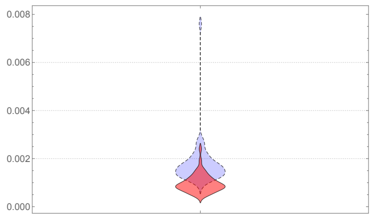


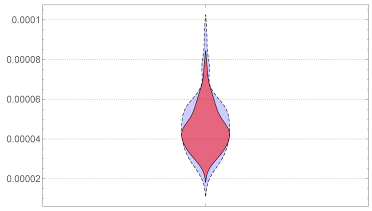
Figure 4.23: Relative loss of policy 1 (in % of the benchmark case); response to CFSI under Basel II with $\alpha_L \in [1,0]$ in $L = \alpha_L L^{MS}_{\delta_s,k,m} + (1-\alpha_L) L^{FS}_{\delta_s,k,m}$

4.5.2 Scenario 2: A monetary policy response to unsustainable credit growth in a loose regulatory environment

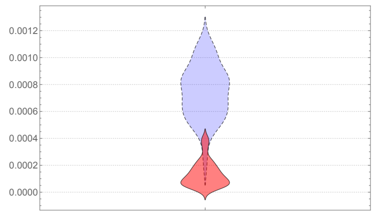
Figure 4.25 shows basically the same story for the response to the credit-to-GDP gap, meaning that in a poorly regulated financial system both analyzed transmission channels of monetary policy do not make much of a difference. Again, we can have a look at the composition of minimum losses. This time the volatility in the target rate reduces tremendously likewise with that of inflation. In opposition to the direct tackling of banks' balance sheet structure, a response to jumps in the credit-to-GDP ratio does only seem to have marginal effects on the resilience of the financial system. While the variance in firm and bank defaults increase, the fiscal costs of banking crises just seem to improve in the probability of extreme events. Again, there is no conflict between policy targets meaning that also with a response to unsustainable credit growth as an indicator for financial imbalances, "leaning against the wind" can contribute to the traditional targets of monetary policy.



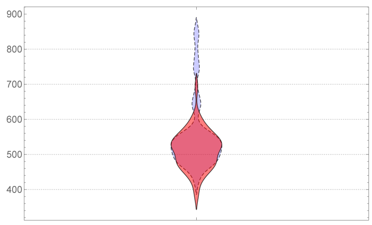
(a) Var($\pi_{\delta_s,k,m}$)



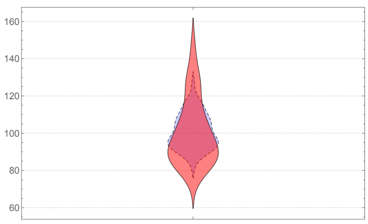
(b) Var($x_{\delta_s,k,m}$)



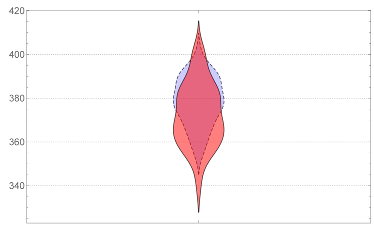
(c) Var($i_{\delta_s,k,m}$)



(d) $\zeta_{\delta_s,k,m}$ (fiscal costs)



(e) $\rho_{\delta_s,k,m}$ (bank defaults)



(f) $\gamma_{\delta_s,k,m}$ (firm defaults)

Figure 4.24: Minimum loss given a response to the credit-to-GDP gap under Basel II; $\delta_\pi = 1.5; \delta_x = 2.5; \delta_s = 1.0; \alpha_L = 1$.
The blue, dashed distribution represents the benchmark scenario while the red, solid one represents the counterfactual scenario.

To sum up, our results concerning a deregulated system confirm the expected proposition of the Tinbergen principle in the sense that it is not possible to improve financial stability additionally to the traditional goals of monetary policy when addressing both distinct goals (macro and financial stability) using only monetary policy as policy instrument.[28]

[28] In scenario 1 and 2 the authorities only have the target rate as a policy instrument, since banks are not required to comply with any prudential requirements, i.e. macroprudential policy is not available as a policy tool in these scenarios. This changes in scenarios 3 and 4.

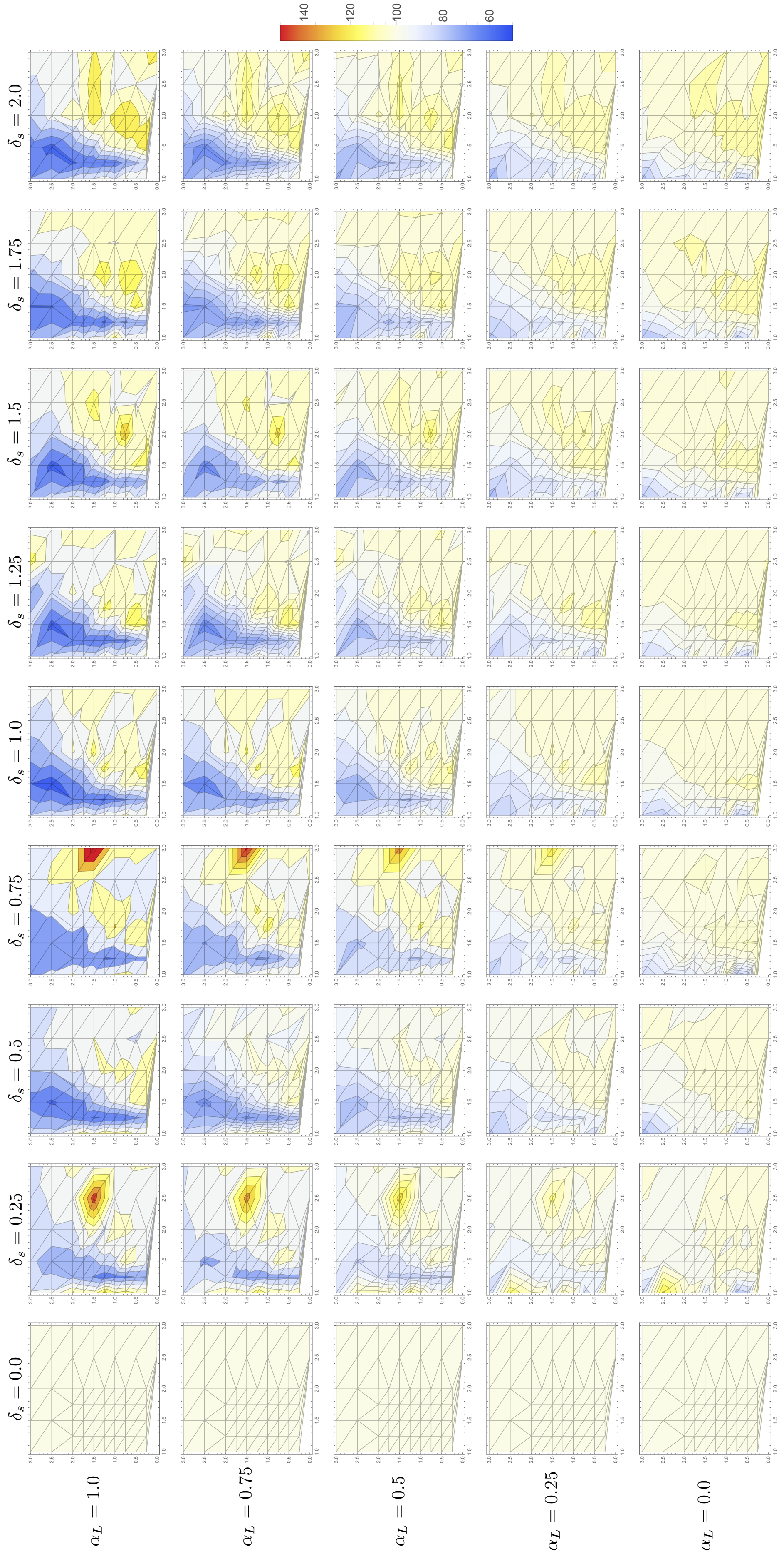
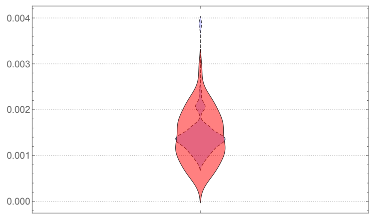


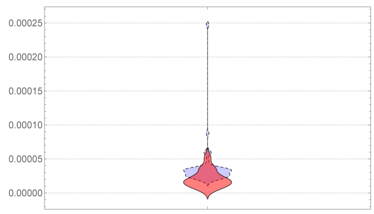
Figure 4.25: Relative loss of policy 2 (in % of the benchmark case); response to the Credit-to-GDP gap under Basel II with $\alpha_L \in [1,0]$ in $L = \alpha_L L^{MS}_{\delta_s,k,m} + (1-\alpha_L) L^{FS}_{\delta_s,k,m}$

4.5.3 Scenario 3: A monetary policy response to financial sector leverage in a tight regulatory environment

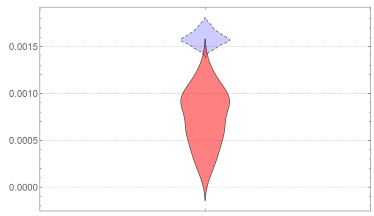
If now the supervisory authorities decide to terminate a period of significant financial deregulation by burdening financial intermediaries with various prudential requirements, as happened in the aftermath of the recent financial crisis, the picture is somewhat different. With macroprudential policy as a separate and independent policy instrument to tackle financial instability, a supplementary action by the central bank seems to be counterproductive (cf. figure 4.27). Given the setting of the current scenario, the loss is minimized if central bankers would use the monetary policy instrument exclusively to target traditional goals, i.e. the common dual mandate, because the tighter financial regulation already serves as first line of defense against banking crises. Thus, any additional intervention via the target rate has a negative impact on the traditional monetary policy goals. Moreover, the results show that without an active guidance of economic activity through monetary policy, financial stability cannot be achieved, i.e. losses for $\delta_\pi \approx 1.25$ significantly increase the fragility of the system which underpins the above mentioned common view that inflation can be seen as one of the main sources of financial instability. Hence, our results confirm that, in line with Adrian and Shin (2008a,b), both policy instruments are inherently connected and complementary, thus, influence each other which emphasizes that an appropriate coordination is inevitably and that the prevailing dichotomy of the currently used linear quadratic framework may lead to misleading results.



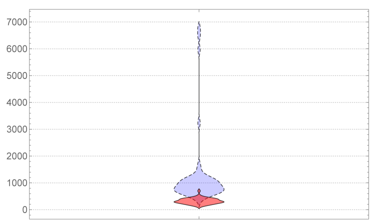
(a) Var($\pi_{\delta_s,k,m}$)



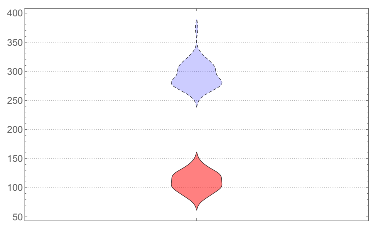
(b) Var($x_{\delta_s,k,m}$)



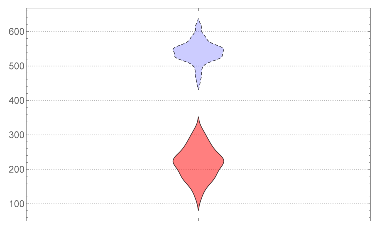
(c) Var($i_{\delta_s,k,m}$)



(d) $\zeta_{\delta_s,k,m}$ (fiscal costs)



(e) $\rho_{\delta_s,k,m}$ (bank defaults)



(f) $\gamma_{\delta_s,k,m}$ (firm defaults)

Figure 4.26: Minimum loss given a response to CFSI under Basel III; $\delta_\pi = 1.25; \delta_x = 2.5; \delta_s = 0.0; \alpha_L = 0$.
The blue, dashed distribution represents the benchmark scenario while the red, solid one represents the counterfactual scenario.

Having a closer look at the composition of the minimum loss, figure 4.26 shows that even without a central bank which leans against the wind, both the traditional goals of monetary policy as well as the goal of a much safer banking sector seem to be achievable simultaneously leading to positive effects on the real economy. Put differently, the results suggest that a tightening of financial regulation only comes at marginal costs in terms of the central bank's primary goals (macroeconomic stability) but can significantly improve financial stability within the artificial economy. Under the Basel III accord, volatility of inflation rises while volatility of output and interest rates decrease vastly. In contrast, figure 4.26d–4.26f highlight the considerable role of an appropriate degree of financial regulation for the resilience of the financial system. The fiscal costs caused by the need to recapitalize significantly large institutions (government bail outs of banks which are "too big to fail") could be lowered tremendously. This stems mainly from the fact that the tail risk concerning the occurrence of bankruptcy cascades massively boosting fiscal costs could be strikingly decreased by providing an incentive scheme which is sufficiently able to control for banks' risk appetite through the imposition of prudential regulatory requirements. While also the amount of bank defaults decreases significantly, the more interesting part of the results is the effect of a tightened banking regulation on the real sector. The relatively stable range of firm defaults under Basel II (≈ 550 defaults per run) turns into a range with slightly increased variance but with a significantly lower mean. This stems from the fact that banks under Basel III have less lending capacity per unit of capital and also tighter leverage restrictions. At the first glance one might argue that this may lead to non-exhausted growth potential but it rather seems to implicitly restrict lending activity to the already (unsustainable) high-leveraged part of the real sector, dampening the build-up of financial imbalances and, therefore, improving the overall sustainability of economic activity. Hence, the implementation of macroprudential policy has the effect that banks are more cautious in their lending activity since they have to ponder whether to grant a credit to a firm since their lending capacity is much more sensitive to a possible future non-performance of its customers.

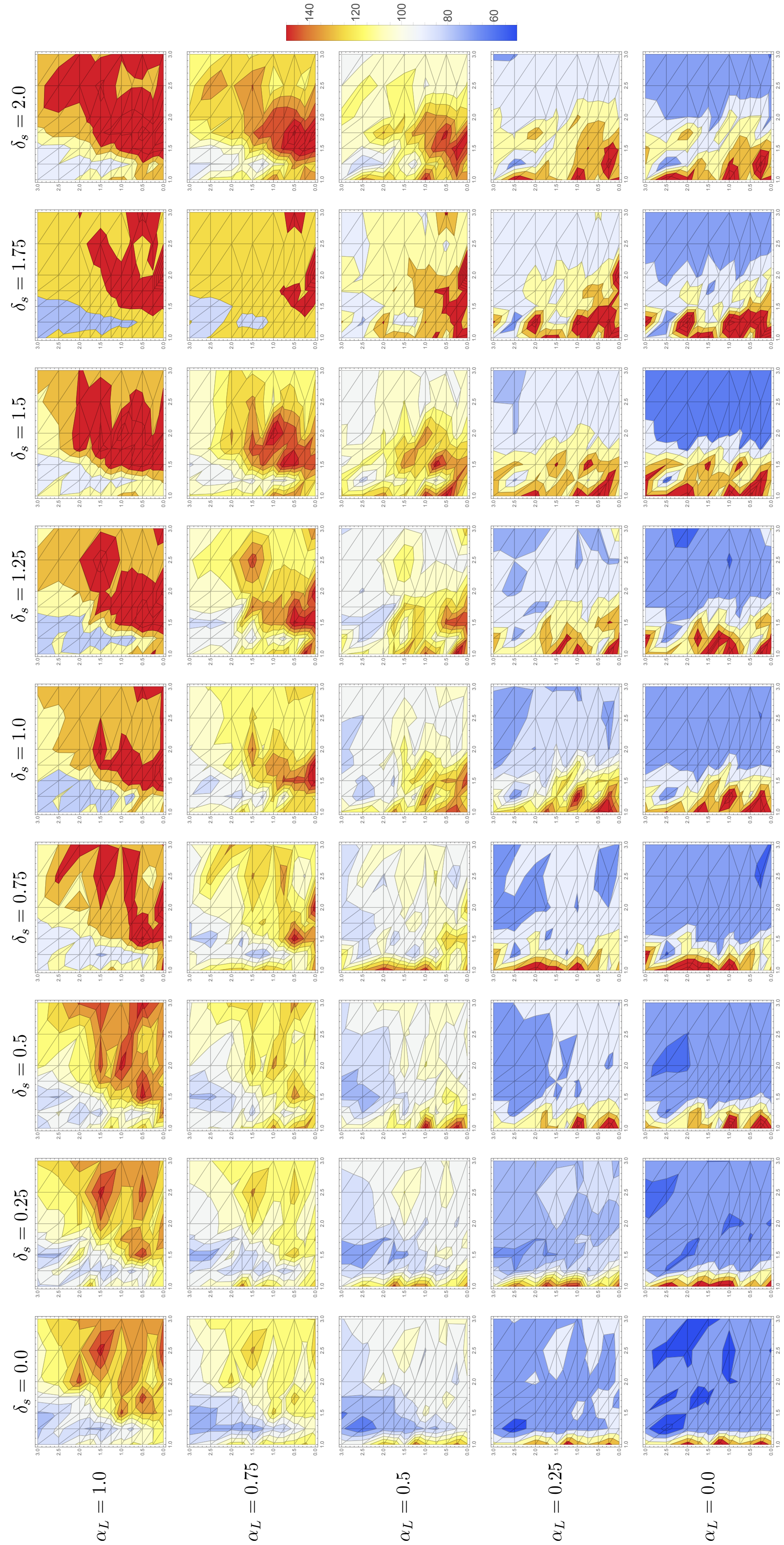
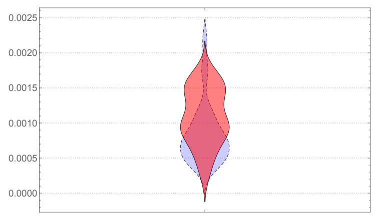


Figure 4.27: Relative loss of policy 3 (in % of the benchmark case); response to CFSI under Basel III with $\alpha_L \in [1{,}0]$ in $L = \alpha_L L^{MS}_{\delta_s,k,m} + (1-\alpha_L) L^{FS}_{\delta_s,k,m}$

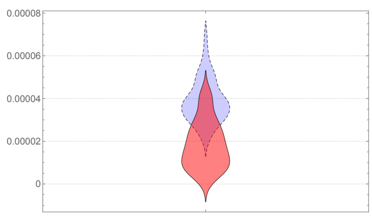
4.5.4 Scenario 4: A monetary policy response to unsustainable credit growth in a tight regulatory environment

For the response to the credit-to-GDP gap, qualitative results are similar to a direct response to unsustainable levels of leverage in the financial sector (scenario 3). $\delta_s > 0$ has almost the same negative impact on the traditional monetary policy goals. The major difference here is that the resilience of the financial system does improve slightly for moderate levels of δ_s, i.e. the minimum loss given the focus on L_{FS} ($\alpha_L = 0$) is achieved for $\delta_s = 0.5$. But since it is doubtlessly useful to search for the best compromise of both targets, $\delta_s = 0.0$ would be appropriate due to the negative effect on volatility of inflation rates.

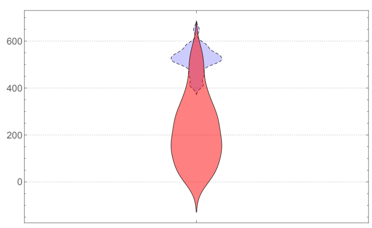
Also the composition of the minimum loss differs from a response to the CFSI, mainly in the higher amount of bank defaults although fiscal costs and firm defaults decline sharply. This phenomenon seems to stems from the conflicting effects of the presence of prudential requirements (positive) and the $\delta_s > 0$ (negative) on the financial system. Thus, there are still cases in which tax payers are burdened with high costs of banking crises but stricter lending standards are clearly beneficial in order to prevent from frequent massive public sector interventions which is in line with the findings of Rubio and Carrasco-Gallego (2014) and Gelain et al. (2012). Also in line with Gelain et al. (2012) is that a direct interest response to excessive credit growth in the central bank's interest rate rule can stabilize output but has the drawback of magnifying the volatility of inflation.



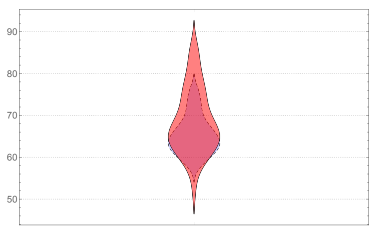
(a) Var($\pi_{\delta_s,k,m}$)



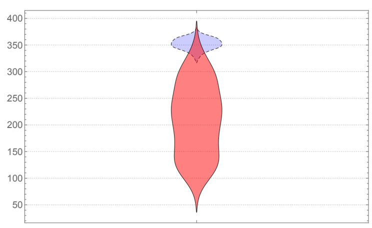
(b) Var($x_{\delta_s,k,m}$)



(c) $\zeta_{\delta_s,k,m}$ (fiscal costs)



(d) $\rho_{\delta_s,k,m}$ (bank defaults)



(e) $\gamma_{\delta_s,k,m}$ (firm defaults)

Figure 4.28: Minimum loss given a response to credit-to-GDP gap under Basel III; $\delta_\pi = 3.0$; $\delta_x = 0.5$; $\delta_s = 0.5$; $\alpha_L = 0$.
The blue, dashed distribution represents the benchmark scenario while the red, solid one represents the counterfactual scenario.

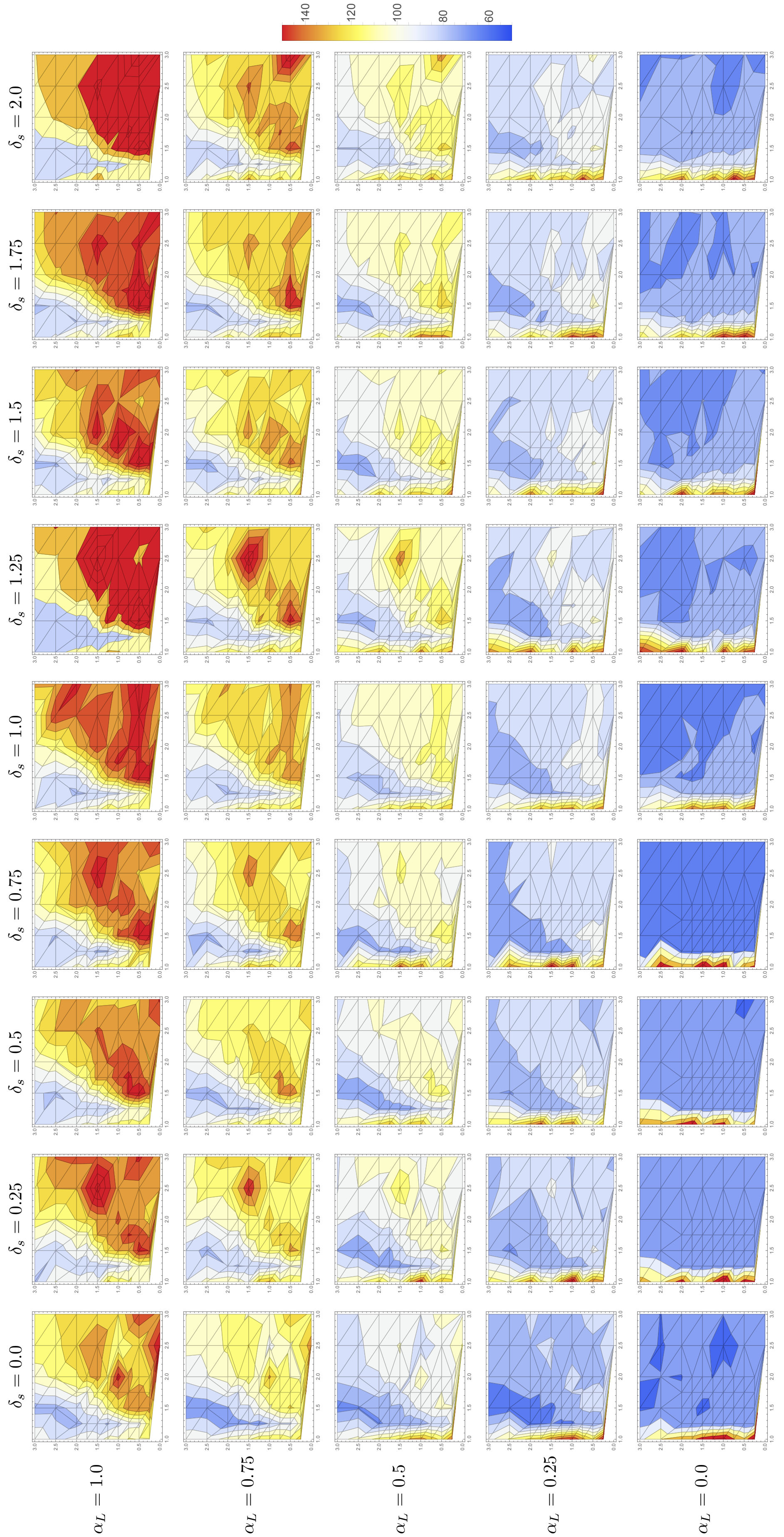


Figure 4.29: Relative loss of policy 4 (in % of the benchmark case); response to Credit-to-GDP gap under Basel III with $\alpha_L \in [1,0]$ in $L = \alpha_L L^{MS}_{\delta_s,k,m} + (1-\alpha_L)L^{FS}_{\delta_s,k,m}$

4.6 Concluding Remarks

The aim of this paper is to shed some light on the current debate on whether central banks should lean against financial imbalances and whether financial stability issues should be an explicit concern of monetary policy decisions or if these should be left to macroprudential regulation and banking supervision. Based on the pre-crisis situation in which financial regulation was way too loose and central banks just focused on their usual dual mandate, there are two policies that have been found adequate to increase the overall resilience of the financial system, i.e. either monetary or macroprudential policy (or a combination of both). So, we also shed some light on the nexus between financial regulation and monetary policy by considering the outcome of policy experiments in terms of macroeconomic and financial stability.

As a framework for the analysis, we present an agent-based macro-model with heterogeneous interacting agents and endogenous money. The central bank agent plays a particular role here since it controls market interest rates via monetary policy decisions which, in turn, affect credit demand and overall economic activity. Therefore, we think that the presented model is well suited to analyze the research question at hand.

Our simulation experiments provide three main findings. First, assigning more than one objective to the monetary policy instrument in order to achieve price, output and financial stability simultaneously, confirms the expected proposition of the Tinbergen principle in the sense that it is not possible to improve financial stability additionally to the traditional goals of monetary policy. The results of our experiments show that after a long phase of deregulation, leaning against the wind has a positive impact on price and output stability but affects the rather fragile financial system only marginally. Moreover, in a system in which banks have to comply with rather tight prudential requirements, a central bank's additional response to the build-up of financial imbalances does not lead to improved outcomes concerning both macroeconomic and financial stability. In contrast, using prudential regulation as an independent and unburdened policy instrument significantly improves the resilience of the system.

Second, leaning against the wind should only serve as a first line of defense in the *absence* of prudential financial regulation. If the activity of the banking sector is already guided by an appropriate regulatory framework, the results are in line with Svensson (2012) who argues that *"the policy rate is not the only available tool, and much better instruments are available for achieving and maintaining financial stability. Monetary policy should be the last line of defense of financial stability, not the first line"*. Macroprudential policy dampens the build-up of financial imbalances and contributes to the resilience of the financial system by restricting credit to the unsustainable high-leveraged part of the real economy. This strengthens the view of opponents which argue that both policies are designed for their specific purpose and that they should be used accordingly.

Third, our results confirm that, in line with Adrian and Shin (2008a,b), both policies are inherently connected and, thus, influence each other which emphasizes that an appropriate co-ordination is inevitably and that the prevailing dichotomy of the currently used linear quadratic framework may lead to misleading results.

Finally, the present paper is useful to understand that the famous principle of Tinbergen has indeed its justification since extending the objective of monetary policy in order to address additional goals merely transforms the target rate into an overburdened policy instrument that is not able to achieve its traditional policy goals. In this regard, Olsen (2015) is right when arguing that financial regulation probably cannot do it alone and that it needs support but without overburdening monetary policy's mandate. But this seems to be the crux of the matter. Indeed, there can be done too much when heading towards crises mitigation since additional central bank actions can also result in rather counterproductive activism merely contributing to unintended volatility than strengthening the resilience of the system. In any case, we think that additional research in this area is needed in order to further explore the nexus between monetary policy and financial regulation to avoid such tensions.

4.7 Model Parameterization

Table 4.4: Model parameterization

Symbol	Type	Description	Updating	Initialization
B	sub	# of banks	–	5
b	sub	bank b	–	
F	sub	# of firms	–	25
f	sub	firm f	–	
H	sub	# of households	–	125
h	sub	household h	–	
T	sub	# of ticks	–	3000
t	sub	ticks/periods	–	1
α	par	Exponent in firms Cobb-Douglas prod. fct.	–	0.2
α^{FS}	par	Weight of financial stability indicator in loss fct.	–	0.01
α_π	par	Weight of inflation variance in loss fct.	–	1.0
α_i	par	Weight of target rate variance in loss fct.	–	0.1
α_k	par	Weight of CFSI/Credit-to-GDP gap in loss fct.	–	1.0
α_x	par	Weight of output gap variance in loss fct.	–	0.5
χ_π	par	CB credibility parameter	–	0.25
δ_π	par	Instrument param. for price stability in TR	–	1.25
δ_s	par	Instrument param. for financial stability in TR	–	$\in (0, 0.5)$
δ_x	par	Instrument param. for output stability in TR	–	0.25

Continued on next page

Table 4.4 – *Continued from previous page*

Symbol	Type	Description	Updating	Initialization
η_h	par	consumption preference parameter	–	$\sim\mathcal{U}(0{,}0.5)$
κ_f	par	External finance factor of firms (10% buffer)	–	1.1
λ	par	Smoothing parameter for HP-filter	–	6.25 / 1600
μ	par	Price mark-up on production costs	–	1.1
ω_Ξ	par	Employment gap param. for wage decision	–	0.005
π^*	par	Inflation target of the CB	–	0.02
ψ_h	par	Labor skill of household h	–	$\max[0.5, \sim\mathcal{N}(2,\sigma^2)]$
τ^C	par	Corporate tax	–	0.6
τ^I	par	Tax on income	–	0.3
τ^{CG}	par	Tax on capital gains	–	0.25
τ^{VAT}	par	Value added tax (tax on consumption)	–	0.2
θ	par	Retained earnings parameter for firm sector	–	0.9
ϱ	par	Firm entry parameter	–	$\sim\mathcal{U}(0{,}48)$
φ	par	Money Market interest parameter	–	5
σ_1	par	Money Market interest parameter	–	see table 4.1
σ_2	par	Money Market interest parameter	–	see table 4.1
σ_3	par	Money Market interest parameter	–	see table 4.1
σ_4	par	Money Market interest parameter	–	see table 4.1
A_t	par	Firm technology parameter	quarterly	1.0
g_A	par	Annual technological progress of firms	–	0.012
g_A^Q	par	Monthly technological progress of firms	–	0.003
T_π	par	Expected inflation horizon	–	24
T_ψ	par	Employment gap horizon	–	12
U^*	par	Target utilization of firms	–	0.75
CAR	par	Capital adequacy requirement (Basel III)	–	0.045
CConB	par	Capital conservation buffer (Basel III)	–	0.025
M	par	Parameter for determination of CB's CCycB	–	10
N	par	Parameter for determination of CB's CCycB	–	2
Γ_t	var	Excess reserve supply on money market in t	w.n.	
Λ_t	var	Credit-to-GDP ratio in t		
Λ_t^n	var	Long-term trend of the Credit/GDP ratio in t		
$\Lambda_t - \Lambda_t^n$	var	Credit-to-GDP gap in t		
$\Omega_{k,t}$	var	# of days since last bond coupon paym.	weekly	
π_t	var	Annual inflation rate in t	yearly	0.0
π_t^e	var	Expected inflation rate	weekly	0.02
π_t^m	var	Annualized monthly inflation rate	monthly	
$\Pi_{f,t}^{at}$	var	Profit after tax of firm f in t	yearly	
$\Pi_{f,t}^{bt}$	var	Profit before tax of firm f in t	yearly	
$\Psi_{f,t}$	var	Aggregate labor input of firm f in t	weekly	
$\Upsilon_{k,t}$	var	Total days in coupon period of bond k in t	weekly	

Continued on next page

Table 4.4 – *Continued from previous page*

Symbol	Type	Description	Updating	Initialization
$\varepsilon(\xi_{b,t})$	var	Risk premium for interbank lending depending on D/E ratio of bank b	w.n.	
$\Xi_{f,t}$	var	Weighted employment gap of firm f		
$k_{\delta_s,m}$	var	Weight of TR-augmentation in loss fct.		
$\gamma_{\delta_s,k,m}$	var	Weight of bank/firm defaults in loss fct.		
$\rho_{\delta_s,k,m}$	var	Weight of bank bailouts in loss fct.		
$\zeta_{\delta_s,k,m}$	var	Weight of avg. fiscal costs in loss fct.		
$\mathcal{L}_{f,t}$	var	Need for external finance of firm f in t	quarterly	
$B_{CB,t}$	var	Government bonds hold by the CB in t	weekly	0.0
$B_{G,t}$	var	Issued public debt of government in t (bonds)	weekly	0.0
$BL_{b,t}$	var	Business loans of bank b in t	weekly	0.0
C_t	var	Outstanding credit to the real sector in t	weekly	0.0
$CBL_{b,t}$	var	CB liabilities of bank b in t	weekly	0.0
$CFSI^*$	var	CB's target for the CFSI in t	–	6.0
$CFSI_t$	var	Comp. financial stability indicator in t	every 6 weeks	
c_k	var	Coupon of bond k	–	
$c_{h,t}$	var	Actual consumption level of HH h in t	weekly	0.0
$c^a_{h,t}$	var	Autonomous consumption level of HH h	quarterly	0.0
$c^p_{h,t}$	var	Planned weekly consumption level of HH h in t	quarterly	0.0
$D_{f,t}$	var	Bank deposits of firm f in t	weekly	0.0
$D_{G,t}$	var	Bank deposits of the government in t	weekly	0.0
$D_{h,t}$	var	Bank deposits of HH h in t	weekly	0.0
$D^{CB}_{G,t}$	var	CB deposits of the government in t	weekly	0.0
$d^B_{h,s}$	var	Dividends received by HH h from bank b	yearly	
$d^F_{h,s}$	var	Dividends received by HH h from firm f	yearly	
$E_{h,t}$	var	Net wealth of HH h in t	weekly	0.0
$E_{f,t}$	var	Net wealth of firm f in t	weekly	0.0
$E_{b,t}$	var	Net wealth of bank b in t	weekly	0.0
$E_{G,t}$	var	Net wealth of the government in t	weekly	0.0
$E_{CB,t}$	var	Net wealth of CB in t	weekly	0.0
$ES_{h,t}$	var	HH h's share of firms/banks	w.n.	0.0
$e^B_{h,s}$	var	Investment of HH h for founding bank b	w.n.	
$e^F_{h,s}$	var	Investment of HH h for founding firm f	w.n.	
$FV_{k,t}$	var	Face value of bond k in t	weekly	
$GA_{CB,t}$	var	Government account at CB in t	weekly	0.0
$GB_{b,t}$	var	Government bonds of bank b in t	weekly	0.0
$GD_{b,t}$	var	Government deposits of bank b in t	weekly	0.0
i^r	var	Real interest rate (long-term)	w.n.	0.02
i^*_t	var	CB target rate in t	every 6 weeks	0.01
i^{OSDF}_t	var	Op. standing deposit facility of CB in t	every 6 weeks	0.0075

Continued on next page

Table 4.4 – *Continued from previous page*

Symbol	Type	Description	Updating	Initialization
i_t^{OSLF}	var	Op. standing lending facility of CB in t	every 6 weeks	0.0125
$i_f^{\mathcal{L}}$	var	Interest payments for outst. loans of firm f	w.n.	
$i_{b,f,t}$	var	Loan interest charged by bank b on firm f in t	w.n.	$i_t^* + 0.03$
$i_{b,t}^{Deposit}$	var	Interest on deposits paid by bank b in t	every 6 weeks	0.0025
$i_{b,t}^{MM}$	var	Money market int. rate faced by bank b in t	w.n.	
$i_{h,s}^{D}$	var	Interest received on $D_{h,t}$ by HH h in s	yearly	
$I_{h,t-12}$	var	Avg. weekly income of HH h of prev. quarter	quarterly	
$Inv_{b,t}$	var	Value of Inventory of firm f in t	weekly	0.0
$IO_{f,t}$	var	Interest Obligations of firm f in t	weekly	0.0
$IR_{b,t}$	var	Interest receivables of bank b in t	weekly	0.0
$L^{FS}_{\delta_s,k,m}$	var	Loss fct. to determine financial stability	–	
$L^{MS}_{\delta_s,k,m}$	var	Loss fct. to determine macroeconomic stability	–	
$L_{CB,t}$	var	CB loans to the banking sector in t	weekly	0.0
$L_{f,t}$	var	Debt capital of firm f in t	weekly	0.0
$n_{k,t}$	var	# of remaining coupon paym. of bond k at t	weekly	
$\Pr\left(\mathcal{L}_{f,t} \mid i_{b,f,t}\right)$		Probability that firm f takes $\mathcal{L}_{f,t}$ given $i_{b,f,t}$	quarterly	
$p_{f,t}$	var	Offered price of firm f in t	quarterly	200.0
$p_{k,t}^{clean}$	var	Clean price of government bonds	weekly	
$q_{f,t}$	var	Actual production of firm f in t	weekly	
$q_{f,t}^*$	var	Production target of firm f in t	quarterly	$2H$
$R_{b,t}$	var	Central bank reserves of bank b in t	weekly	0.0
$R_{b,t}^*$	var	Reserve target of bank b in t	weekly	0.0
$R_{CB,t}$	var	Outst. CB reserves hold by banking sector in t	weekly	0.0
$RD_{b,t}$	var	Retail deposits of bank b in t	weekly	0.0
$RWA_{b,t}$	var	Risk-weighted assets of bank b in t		
$s_{f,t}$	var	Sales of firm f in t	weekly	
$TA_{b,t}$	var	Total assets of bank b in t	weekly	0.0
$TA_{CB,t}$	var	Total assets of CB in t	weekly	0.0
$TA_{f,t}$	var	Total assets of firm f in t	weekly	0.0
$TA_{G,t}$	var	Total assets of the government in t		0.0
$TA_{h,t}$	var	Total assets of HH h in t	weekly	0.0
$UB_{h,s}$	var	Unemployment benefit received by HH h in t	yearly	
$w_{f,t}$	var	Wage per unit of labor skill offered by f in t	quarterly	1000.0
$w_{h,s}$	var	Wage received per unit of labor skill by h in s	quarterly	1000.0
$WL_{b,t}$	var	Wholesale loans of bank b in t	weekly	0.0
$WO_{b,t}$	var	Wholesale deposits of bank b in t	weekly	0.0
x_t	var	Output gap in t	yearly	0.0
x_t^n	var	Potential output in t	yearly	0.0
z	var	Surcharge-bucket assignment parameter		
CCycB_t	var	Countercyclical buffer set by the CB in t	6 weeks	0.0

Acknowledgements: I would like to thank Sigridur Benediktsdottir, Stijn Claessens, Matthias Lengnick, Sander van der Hoog, Hans-Werner Wohltmann, Benjamin Burghardt, Falko Schlaffke, Sven Offick and Stephen Sacht as well as the participants of the *3rd Macro, Banking and Finance Workshop* (Pavia/Milano, 2015), and the *International conference on "Capital Flows, Systemic Risk and Policy Responses"* (Reykjavik, Iceland, 28-29 April 2016) for a lot of stimulating comments.

References of Chapter 4

Adrian, T. and Shin, H. S. (2008a). Financial Intermediaries, Financial Stability, and Monetary Policy, *Staff Report no. 346*, Federal Reserve Bank of New York.

Adrian, T. and Shin, H. S. (2008b). Liquidity, Monetary Policy, and Financial Cycles, *Current Issues in Economics and Finance* **14**(1): 1–7.

Afonso, G. and Lagos, R. (2015). The Over-the-Counter Theory of the Fed Funds Market: A Primer, *Journal of Money, Credit and Banking* **47**(S2): 127–154.

Agénor, P.-R., Alper, K. and Pereira da Silva, L. A. (2013). Capital Regulation, Monetary Policy, and Financial Stability, *International Journal of Central Banking* **9**(3): 193–238.

Agénor, P.-R. and Pereira da Silva, L. A. (2012). Macroeconomic Stability, Financial Stability, and Monetary Policy Rules, *International Finance* **15**(2): 205–224.

Agénor, P.-R. and Pereira da Silva, L. A. (2014). Macroprudential Regulation and the Monetary Transmission Mechanism, *Journal of Financial Stability* **13**(0): 44–63.

Akram, Q. F. and Eitrheim, Ø. (2008). Flexible Inflation Targeting and Financial Stability: Is It Enough to Stabilise Inflation and Output?, *Journal of Banking & Finance* **32**: 1242–1254.

Arciero, L., Biancotti, C., D'Aurizio, L. and Impenna, C. (2009). Exploring Agent-based Methods for the Analysis of Payment Systems: A Crisis Model for StarLogo TNG, *Journal of Artificial Societies and Social Simulation* **12**(1): 2.

Ausloos, M., Mikiewicz, J. and Sanglier, M. (2004). The Durations of Recession and Prosperity: Does Their Distribution Follow a Power or an Exponential Law?, *Physica A: Statistical Mechanics and its Applications* **339**(34): 548 – 558.

Babecký, J., Havránek, T., Matějů, J., Rusnák, M., Šmídková, K. and Vašíček, B. (2013). Leading Indicators of Crisis Incidence: Evidence From Developed Countries, *Journal of International Money and Finance* **35**: 1–19.

Badarau, C. and Popescu, A. (2015). Monetary Policy and Financial Stability: What Role for the Interest Rate?, *International Economics and Economic Policy* **12**(3): 359–374.

Bank of England (2014a). Bank of England Settlement Accounts, *Technical report*, Bank of England.
URL: *http://www.bankofengland.co.uk/markets/Documents/paymentsystems/boesettlementaccounts.pdf*

Bank of England (2014b). Cut-off Times for Payments to and from Reserve Accounts, and for Operational Standing Facilities, *Technical report*, Bank of England.
URL: *http://www.bankofengland.co.uk/markets/Pages/money/reserves/default.aspx*

Bank of England (2014c). The Bank of England's Sterling Monetary Framework (The Red Book), *Technical report*, Bank of England.
URL: *http://www.bankofengland.co.uk/markets/Pages/sterlingoperations/redbook.aspx*

Basel Committee on Banking Supervision (BCBS) (2010). Guidance for National Authorities Operating the Countercyclical Capital Buffer, *Technical report*, Bank for International Settlements (BIS).
URL: *http://www.bis.org/publ/bcbs187.htm*

Blot, C., Creel, J., Hubert, P., Labondance, F. and Saraceno, F. (2015). Assessing the Link Between Price and Financial Stability, *Journal of Financial Stability* **16**(0): 71–88.

Bodie, Z., Kane, A. and Marcus, A. (2010). *Investments*, 9th edn, McGraw-Hill Education.

Bookstaber, R. (2013). *Using Agent-based Models for Analyzing Threats to Financial Stability*, DIANE Publishing Company.

Bordo, M. D. and Jeanne, O. (2002). Monetary Policy and Asset Prices: Does 'Benign Neglect' Make Sense?, *International Finance* **5**(2): 139–164.

Borio, C. (2006). Monetary and Financial Stability: Here to Stay?, *Journal of Banking & Finance* **30**(12): 3407–3414.

Borio, C. (2014). Monetary Policy and Financial Stability: What Role in Prevention and Recovery?, *BIS Working Papers no. 440*, Bank for International Settlements.

Borio, C. and Drehmann, M. (2009). Assessing the Risk of Banking Crises - Revisited, *BIS Quarterly Review* **March**: 29–46.

Borio, C. and Lowe, P. (2002). Asset Prices, Financial and Monetary Stability: Exploring the Nexus, *BIS Working Papers no. 114*, Bank for International Settlements.

Borio, C. and Lowe, P. (2004). Securing Sustainable Price Stability: Should Credit Come Back From the Wilderness?, *BIS Working Papers no. 157*, Bank for International Settlements.

Burns, A. F. and Mitchell, W. C. (1946). *Measuring Business Cycles*, NBER Books, NBER.
URL: *http://www.nber.org/books/burn46-1*

Castaldi, C. and Dosi, G. (2009). The Patterns of Output Growth of Firms and Countries: Scale Invariances and Scale Specificities, *Empirical Economics* **37**(3): 475–495.

Cecchetti, S., Genberg, H., Lipsky, J. and Wadhwan, S. (2000). Asset Prices and Central Bank Policy, *The Geneva Report on the World Economy no. 02*, International Centre for Monetary and Banking Studies, Geneva.
URL: *http://down.cenet.org.cn/upfile/8/2010318204458149.pdf*

Chatelain, J.-B. and Ralf, K. (2014). Stability and Identification with Optimal Macroprudential Policy Rules, *Papers 1404.3347*, arXiv.org.
URL: *http://ideas.repec.org/p/arx/papers/1404.3347.html*

Choudhry, M. (2010). *The Repo Handbook*, 2nd edn, Butterworth-Heinemann, Oxford.

Criste, A. and Lupu, I. (2014). The Central Bank Policy Between the Price Stability Objective and Promoting Financial Stability, *Procedia Economics and Finance* **8**(0): 219–225.

Cúrdia, V. and Woodford, M. (2010). Credit Spreads and Monetary Policy, *Journal of Money, Credit and Banking* **42**(s1): 3–35.

da Silva, M. A. and Lima, G. T. (2015). Combining Monetary Policy and Prudential Regulation: an Agent-based Modeling Approach, *Banco Central Do Brasil, Working Papers No. 394* .

Dawid, H., Gemkow, S., Harting, P., van der Hoog, S. and Neugart, M. (2014). The Eurace@Unibi Model: An Agent-based Macroeconomic Model for Economic Policy Analysis, *University of Bielefeld Working Papers in Economics and Management* **2012**(5).

Dent, A. and Dison, W. (2012). The Bank of England's Real-Time Gross Settlement Infrastructure, *Bank of England Quartely Bulletin* **2012**(Q3): 234–243.

Deutsche Bundesbank (2015). The Importance of Macroprudential Policy for Monetary Policy.

Disyatat, P. (2010). Inflation Targeting, Asset Prices, and Financial Imbalances: Contextualizing the Debate, *Journal of Financial Stability* **6**(3): 145–155.

Dosi, G., Fagiolo, G., Napoletano, M. and Roventini, A. (2013). Income Distribution, Credit and Fiscal Policies in an Agent-based Keynesian Model, *Journal of Economic Dynamics and Control* **37**(8): 1598–1625.

Dosi, G., Fagiolo, G., Napoletano, M., Roventini, A. and Treibich, T. (2015). Fiscal and Monetary Policies in Complex Evolving Economies, *Journal of Economic Dynamics and Control* **52**(0): 166 – 189.

Dosi, G., Fagiolo, G. and Roventini, A. (2006). An Evolutionary Model of Endogenous Business Cycles, *Computational Economics* **27**(1): 3–34.

Dosi, G., Fagiolo, G. and Roventini, A. (2008). The Microfoundations of Business Cycles: an Evolutionary, Multi-agent Model, *Journal of Evolutionary Economics* **18**(3-4): 413–432.

Dosi, G., Fagiolo, G. and Roventini, A. (2010). Schumpeter Meeting Keynes: A Policy-friendly Model of Endogenous Growth and Business Cycles, *Journal of Economic Dynamics and Control* **34**(9): 1748–1767.

Dosi, G., Napoletano, M., Roventini, A. and Treibich, T. (2014). Micro and Macro Policies in the Keynes+Schumpeter Evolutionary Models, *LEM working paper series no. 2014/21*, Laboratory of Economics and Management (LEM), Sant'Anna School of Advanced Studies, Pisa.
URL: *http://www.lem.sssup.it/WPLem/2014-21.html*

Drehmann, M., Borio, C., Gambacorta, L., Jiménez, G. and Trucharte, C. (2010). Countercyclical Capital Buffers: Exploring Options, *BIS Working Papers no. 317*, Bank for International Settlements.

Drehmann, M., Borio, C. and Tsatsaronis, K. (2012). Characterising the Financial Cycle: Don't Lose Sight of the Medium Term!, *BIS Working Papers no. 380*, Bank for International Settlements.

Drehmann, M. and Tsatsaronis, K. (2014). The credit-to-GDP Gap and Counter-cyclical Capital Buffers: Questions and Answers, *BIS Quarterly Review* **2014**(March): 55–73.

Driffill, J., Rotondi, Z., Savona, P. and Zazzara, C. (2006). Monetary Policy and Financial Stability: What Role for the Futures Markets?, *Journal of Financial Stability* **2**: 95–112.

ECB (2010). Survey on the Access to Finance of Small and Medium-sized Enterprises in the Euro Area: Second Half of 2009, *Technical report*, European Central Bank.
URL: *https://www.ecb.europa.eu/stats/money/surveys/sme/html/index.en.html*

Fagiolo, G., Moneta, A. and Windrum, P. (2007). A Critical Guide to Empirical Validation of Agent-based Models in Economics: Methodologies, Procedures, and Open Problems, *Computational Economics* **30**(3): 195–226.

Fagiolo, G., Napoletano, M. and Roventini, A. (2008). Are Output Growth-rate Distributions Fat-tailed? Some Evidence From OECD Countries, *Journal of Applied Econometrics* **23**(5): 639–669.

Fagiolo, G. and Roventini, A. (2012). On the Scientific Status of Economic Policy: A Tale of Alternative Paradigms, *The Knowledge Engineering Review* **27**(2): 163–185.

Foos, D., Norden, L. and Weber, M. (2010). Loan growth and riskiness of banks, *Journal of Banking & Finance* **34**(12): 2929 – 2940. International Financial Integration.

Gadanecz, B. and Jayaram, K. (2009). Measures of Financial Stability - A Review, *in* Bank for International Settlements (ed.), *Proceedings of the IFC Conference on "Measuring financial innovation and its impact", Basel, 26-27 August 2008*, Vol. 31 of *IFC Bulletins chapters*, Bank for International Settlements, pp. 365–380.

Galbiati, M. and Soramäki, K. (2011). An Agent-based Model of Payment Systems, *Journal of Economic Dynamics and Control* **35**: 859–875.

Gambacorta, L. and Signoretti, F. M. (2014). Should Monetary Policy Lean Against the Wind? An Analysis Based on a DSGE Model with Banking, *Journal of Economic Dynamics and Control* **43**(C): 146–174.

Gelain, P., Lansing, K. J. and Mendicino, C. (2012). House Prices, Credit Growth, and Excess Volatility: Implications for Monetary and Macroprudential policy, *International Journal of Central Banking* **9**(2): 219–276.

George, E. L. (2014). Supervisory Frameworks and Monetary Policy, *Journal of Economic Dynamics and Control* **49**: 139–141.

Giannini, C. (2011). *The Age of Central Banks*, Edward Elgar Publishing.
URL: *http://dx.doi.org/10.4337/9780857932143*

Giese, J., Nelson, B., Tanaka, M. and Tarashev, N. (2013). How Could Macroprudential Policy Affect Financial System Resilience and Credit? Lessons From the Literature, *Bank of England Financial Stability Paper* **2013**(21).

Goodhart, C. A. E. (1988). *The Evolution of Central Banks*, MIT Press (MA).

Haldane, A. G. and Qvigstad, J. F. (2014). The Evolution of Central Banks – a Practitioner's Perspective, *Norges Bank Conference "Of the uses of central banks: Lessons from history", 4-5 June 2014*.

Hanson, S., Kashyap, A. K. and Stein, J. C. (2011). A Macroprudential Approach to Financial Regulation, *Journal of Economic Perspective* **25**(1): 3–28.

Hellwig, M. (2014). Financial Stability, Monetary Policy, Banking Supervision, and Central Banking, *Preprints of the Max Planck Institute for Research on Collective Goods* **2014**(9).

Issing, O. (2003). Monetary and Financial Stability: Is There a Trade-off?, *BIS Quarterly Review* **16**: 1–7.

Käfer, B. (2014). The Taylor Rule and Financial Stability A Literature Review with Application for the Eurozone, *Review of Economics* **65**(2): 159–192.

Kelsey, E. and Rickenbach, S. (2014). Enhancing the Resilience of the Bank of England's Real-Time Gross Settlement Infrastructure, *Bank of England Quartely Bulletin* **2014**(Q3): 316–320.

Krug, S., Lengnick, M. and Wohltmann, H.-W. (2015). The Impact of Basel III on Financial (In)stability: An Agent-based Credit Network Approach, *Quantitative Finance* **15**(12): 1917–1932.

Kuznets, S. and Murphy, J. (1966). *Modern Economic Growth: Rate, Structure and Spread: An Adaptation*, number 8 in *Current thought series*, Feffer and Simons.

Laeven, L. and Valencia, F. (2013). Systemic Banking Crises Database, *IMF Economic Review* **61**(2): 225–270.

Lavoie, M. (2003). A Primer on Endogenous Credit-Money, *in* L.-P. Rochon and S. Rossi (eds), *Modern Theories of Money: The Nature and Role of Money in Capitalist Economies*, Edward Elgar Publishing Limited, Northampton, pp. 506–543.

Levine, P. and Lima, D. (2015). Policy Mandates for Macro-prudential and Monetary Policies in a New Keynesian Framework, *ECB Working Paper Series no. 1784*, European Central Bank.

Lown, C. and Morgan, D. P. (2006). The Credit Cycle and the Business Cycle: New Findings Using the Loan Officer Opinion Survey, *Journal of Money, Credit and Banking* **38**(6): 1575–1597.

McLeay, M., Radia, A. and Thomas, R. (2014a). Money Creation in the Modern Economy, *Bank of England Quartely Bulletin* **54**(1): 14–27.

McLeay, M., Radia, A. and Thomas, R. (2014b). Money in the Modern Economy: An Introduction, *Bank of England Quartely Bulletin* **54**(1): 4–13.

Mehrling, P. (2012). Three Principles for Market-based Credit Regulation, *American Economic Review* **102**(3): 107–112.

Mendoza, E. G. and Terrones, M. E. (2014). An Anatomy of Credit Booms and their Demise, *in* M. Fuentes, C. E. Raddatz and C. M. Reinhart (eds), *Capital Mobility and Monetary Policy*, Vol. 18 of *Central Banking, Analysis, and Economic Policies Book Series*, Central Bank of Chile, chapter 6, pp. 165–204.

Mishkin, F. S. (2011). Monetary Policy Strategy: Lessons From The Crisis, *in* M. Jarocinski, F. Smets and C. Thimann (eds), *Approaches to monetary policy revisited – lessons from the crisis*, number cbc6 in *Contributions to the 6th ECB Central Banking Conference, 18-19 November 2010*, European Central Bank, chapter 3, pp. 67–118.
URL: *https://ideas.repec.org/h/ecb/chaptr/cbc6-03.html*

Morley, J. (2015). Macro-Finance Linkages, *Journal of Economic Surveys* **forthcoming**.
URL: *http://dx.doi.org/10.1111/joes.12108*

Nakajima, M. (2011). *Payment System Technologies and Functions: Innovations and Developments*, IGI Global, Hershey.

Napoletano, M., Roventini, A. and Sapio, S. (2006). Are Business Cycles All Alike? A Bandpass Filter Analysis of the Italian and US Cycles, *Rivista italiana degli economisti* **1**: 87–118.

Olsen, O. (2015). Integrating Financial Stability and Monetary Policy, Speech of the Governor of Norges Bank (Central Bank of Norway), at the Systemic Risk Center, London School of Economics, London, 27. April 2015.

Phillips, A. W. (1958). The Relation Between Unemployment and the Rate of Change of Money Wage Rates in the United Kingdom 1861-1971, *Economica* **25**(100): 283–299.

Plosser, C. I. (2014). Monetary Rules - Theory and Practice, Speech at the Frameworks for Central Banking in the Next Century Policy Conference, Hoover Institution, Stanford, CA, 30 May 2014.

Popoyan, L., Roventini, A. and Napoletano, M. (2015). Taming Macroeconomic Instability: Monetary and Macro Prudential Policy Interactions in an Agent-based Model, *Technical report*, Laboratory of Economics and Management (LEM), Sant'Anna School of Advanced Studies, Pisa, Italy, working paper No. 2015/33.

Quint, D. and Rabanal, P. (2014). Monetary and Macroprudential Policy in an Estimated DSGE Model of the Euro Area, *International Journal of Central Banking* **10**(2): 169–236.

Ravn, M. O. and Uhlig, H. (2002). On Adjusting the Hodrick-Prescott Filter for the Frequency of Observations, *Review of Economics and Statistics* **84**(2): 371–376.

Reinhart, C. M. and Rogoff, K. S. (2009). The Aftermath of Financial Crises, *American Economic Review* **99**(2): 466–472.

Riccetti, L., Russo, A. and Gallegati, M. (2015). An Agent-based Decentralized Matching Macroeconomic Model, *Journal of Economic Interaction and Coordination* **10**(2): 305–332.

Rubio, M. and Carrasco-Gallego, J. A. (2014). Macroprudential and Monetary Policies: Implications for Financial Stability and Welfare, *Journal of Banking & Finance* **49**(0): 326–336.

Ryan-Collins, J., Greenham, T., Werner, R. and Jackson, A. (2012). *Where Does Money Come From? A Guide to the UK Monetary and Banking System*, 2nd edn, The New Economics Foundation, London.

Salle, I., Sénégas, M.-A. and Yildizoglu, M. (2013). How Transparent About Its Inflation Target Should a Central Bank be? An Agent-based Model Assessment, *Cahiers du GREThA, No. 2013-24* pp. 1–41.

Salle, I., Yıldızoğlu, M. and Sénégas, M.-A. (2013). Inflation Targeting in a Learning Economy: An ABM Perspective, *Economic Modelling* **34**: 114–128.

Scheubel, B. and Körding, J. (2013). Liquidity Regulation, the Central Bank, and the Money Market, *Beitrage zur Jahrestagung des Vereins fur Socialpolitik Wettbewerbspolitik und Regulierung in einer globalen Wirtschaftsordnung - Session Liquidity and Regulation, No. E19-V3* .

Schmitt-Grohé, S. and Uribe, M. (2007). Optimal Simple and Implementable Monetary and Fiscal Rules, *Journal of Monetary Economics* **54**: 1702–1725.

Schwartz, A. J. (1995). Why Financial Stability Depends on Price Stability, *Economic Affairs* **15**(4): 21–25.

Smets, F. (2014). Financial Stability and Monetary Policy: How Closely Interlinked?, *International Journal of Central Banking* **10**(2): 263–300.

Solow, R. M. (1956). A Contribution to the Theory of Economic Growth, *The Quarterly Journal of Economics* pp. 65–94.

Stein, J. C. (2012). Monetary Policy as Financial-stability Regulation, *Quarterly Journal of Economics* **127**: 57–95.

Stein, J. C. (2014). Incorporating Financial Stability Considerations Into a Monetary Policy Framework, Speech at the International Research Forum on Monetary Policy, Washington, D.C.

Stock, J. H. and Watson, M. W. (1999). Business Cycle Fluctuations in US Macroeconomic Time Series (ch. 1), Vol. 1, Part A of *Handbook of Macroeconomics*, Elsevier, pp. 3 – 64.

Stolzenburg, U. (2015). The Agent-based Solow Growth Model with Endogenous Business Cycles, *Economics Working Papers 2015-01*, Christian-Albrechts-University of Kiel, Department of Economics.
URL: *https://ideas.repec.org/p/zbw/cauewp/201501.html*

Suh, H. (2014). Dichotomy Between Macroprudential Policy and Monetary Policy on Credit and Inflation, *Economics Letters* **122**(2): 144–149.

Svensson, L. E. O. (2012). Comment on Michael Woodford, "Inflation Targeting and financial Stability", *Sveriges Riksbank Economic Review* **2012**(1): 33–39.

Tarullo, D. K. (2014). Monetary Policy and Financial Stability, Speech at the 30th Annual National Association for Business Economics Economic Policy Conference, Arlington, Virginia.

Tinbergen, J. (1952). *On the Theory of Economic Policy*, North Holland Publishing Company, Amsterdam.
URL: *http://hdl.handle.net/1765/15884*

Tomuleasa, I.-I. (2015). Macroprudential Policy and Systemic Risk: An Overview, *Procedia Economics and Finance* **20**(0): 645–653. Globalization and Higher Education in Economics and Business Administration - GEBA 2013.

Verona, F., Martins, M. M. F. and Drumond, I. (2014). Financial Shocks and Optimal Monetary Policy Rules, *Research Discussion Papers 21/2014*, Bank of Finland.

Wälde, K. and Woitek, U. (2004). R&D Expenditure in G7 Countries and the Implications for Endogenous Fluctuations and Growth, *Economics Letters* **82**(1): 91 – 97.

Walsh, C. E. (2014). Multiple Objectives and Central Bank Trade-offs Under Flexible Inflation Targeting, Keynote address, 16th Annual Inflation Targeting Seminar, Banco Central do Brazil, May 15-16, 2014.

Winters, B. (2012). Review of the Bank of England's Framework for Providing Liquidity to the Banking System, *Report for the Court of the Bank of England*, Bank of England.

Woodford, M. (2012). Inflation Targeting and Financial Stability, *Sveriges Riksbank Economic Review* **2012**(1): 7–32.

Wright, I. (2005). The Duration of Recessions Follows an Exponential Not a Power Law, *Physica A: Statistical Mechanics and its Applications* **345**(34): 608 – 610.

Yellen, J. L. (2014). Monetary Policy and Financial Stability, Speech at the 2014 Michel Camdessus Central Banking Lecture of the International Monetary Fund, Washington, D.C.

Zarnowitz, V. (1985). Recent Work on Business Cycles in Historical Perspective: A Review of Theories and Evidence, *Journal of Economic Literature* **23**(2): 523–580.

Chapter 5

Shadow Banking, Financial Regulation and Animal Spirits: An Agent-based Approach

Submitted to: Journal of Banking & Finance (2016), currently under review.

Abstract

Over the past decades, the framework for financing has experienced a fundamental shift from traditional bank lending towards a broader market-based financing of financial assets. As a consequence, regulated banks increasingly focus on coping with regulatory requirements meaning that the resulting funding gap for the real economy is left to the unregulated part of the financial system, i.e. to shadow banks highly relying on securitization and repos. Unfortunately, economic history has shown that unregulated financial intermediation exposes the economy to destabilizing externalities in terms of excessive systemic risk. The arising question is now whether and how it is possible to internalize these externalities via financial regulation.

We aim to shed light on this issue by using an agent-based computational macro-model as experimental lab. The model is augmented with a shadow banking sector representing an alternative investment opportunity for the real sector which shows animal spirit-like, i.e. highly pro-cyclical and myopic, behavior in its investment decision.

We find that an unilateral inclusion of shadow banks into the regulatory framework, i.e. without access to central bank liquidity, has negative effects on monetary policy goals, significantly increases the volatility in growth rates and that its disrupting character materializes in increasing default rates and a higher volatility in the credit-to-GDP gap. However, experiments with a full inclusion, i.e. with access to a lender of last resort, lead to superior outcomes relative to the benchmark without shadow banking activity. Moreover, our results highlight the central role of the access to contagion-free, alternative sources of liquidity within the shadow banking sector.

Keywords: Shadow Banking, Financial Stability, Monetary Economics, Macroprudential Policy, Financial Regulation, Agent-based Macroeconomics.

JEL Classification: E44, E50, G01, G28, C63

5.1 Introduction

Over the past decades, the framework for financing has experienced a fundamental shift from the traditional bank-based towards a new and broader market-based credit system entailing new sources of systemic risk [Adrian and Shin (2008); Mehrling (2012); Mehrling et al. (2013)].[1]

As Hoenig (1996) puts it in his remarkable speech held in 1996, i.e. over 10 years before the described developments manifested in the global financial crisis:

> *"In recent years, financial markets around the world have experienced significant structural changes. Some of the more important changes are the growing importance of capital markets in credit intermediation, the emergence of markets for intermediating risks, changes in the activities and risk profiles of financial institutions, and the increasingly global nature of financial intermediation. [...] More than ever before, banks face greater competition from other financial institutions. Many businesses are turning away from banks and other depository institutions and directly toward capital markets and nonbank intermediaries for their funding needs. [...] As these changes occur, financial activities are increasingly taking place outside of the traditional bank regulatory framework. [...] The increased competition in traditional lines of business along with the opportunities in capital and derivatives markets have led the largest domestic and global banks to significantly alter their activities and products. Among the most significant of the new activities are trading and market-making in money markets, capital markets, foreign exchange, and derivatives. The rise in proprietary trading, market-making, and active portfolio management has also dramatically altered the risk profiles of financial institutions. If used properly for portfolio management, new financial instruments can certainly reduce an institution's risk exposure and raise its profitability and viability in the financial marketplace. If used improperly, however, they expose the institution to sudden, extraordinary losses, raising the likelihood of failure. Moreover, the risks and opportunities for failure are often exacerbated by the leverage associated with the new activities and the larger numbers of players and greater degree of anonymity in financial markets. Increased trading activity, for example, has significantly increased the exposure of banks to market risk – the risk of loss due to changes in asset prices and the volatility of asset prices. Like traditional credit risk, market risk can lead to significant losses and ultimately to failure if not managed appropriately. In contrast to credit-related losses, which can take time to develop, losses due to market risk can occur quickly."*

[1] In this context, Adrian and Shin (2008) state that *"[t]he rapid move toward a market-based financial system in recent years has accelerated the trend toward greater reliance on non-traditional, non-deposit-based funding and toward greater use of the interbank market, the market for commercial paper, and asset-backed securities"*.

As a consequence, traditional banks face significant competitive disadvantages forcing them to alter their business model and leaving the resulting funding gap for the real economy to highly specialized non-bank financial intermediaries that can provide liquidity at much lower costs [Martin et al. (2013); Gorton and Metrick (2012a,b); Sunderam (2015)]. The main problem is that such entities exhibit an extensive contribution to systemic risk by carrying out bank-like functions associated with bank-like risks but without being subject to bank-like regulation and without access to a lender of last resort or to public backstops like deposit insurance schemes. Hence, there is a latent risk of runs on these institutions comparable to the situation of traditional banks in the 19th century [Dombret (2014b); Haldane and Qvigstad (2014); Dombret (2013a)]. Such runs can lead to a materialization of idiosyncratic liquidity risk and may force single entities into harmful deleveraging processes. This can negatively affect asset prices bearing the risk of spreading financial distress through the highly interconnected system. Adrian and Ashcraft (2012a) describe the financial frictions involved in shadow banking in great detail. They emphasize that the inherent fragility of this sector is directly related to both sides of shadow banks' balance sheets, namely to the asset side due to poor underwriting standards while erratic and fickle wholesale funding affects the liabilities side. Paired with investor's fundamental ignorance of tale risks [Gennaioli et al. (2013)], their collective underestimation of asset correlations (e.g. fire sale externalities, leverage cycles [Geanakoplos (2009); Adrian and Boyarchenko (2012); Martin et al. (2013); Aymanns and Farmer (2015)]) and their animal spirit-like, highly pro-cyclical investment decisions (over-investment during booms and the excessive collapses during bust), unregulated credit intermediation establishes optimal conditions for systemic risk to materialize in the form of financial crises.

Hence, financial supervisory authorities have the difficult task to design an appropriate regulatory regime that restricts loan portfolios and prevents excessive risk-taking to ensure a constant stream of credit to the real sector [Luttrell et al. (2012); Schwarcz (2012); Financial Stability Board (2015)]. The arising question is now whether and how it is possible to internalize these externalities via financial regulation.

The still small but growing amount of studies in this strand is dominated by general equilibrium frameworks, thus, we contribute to the field by presenting an agent-based macro-model with heterogeneous interacting agents, endogenous money and a shadow banking sector representing an alternative investment opportunity for the real sector. The model comprises all main sources of systemic risk associated with unregulated credit intermediation such as animal spirit-like, sudden collective withdrawals of invested funds, runs, fire sales of assets, poor underwriting standards, the evaporation of whole sale funding as well as systemic under-capitalization making it well suited to analyze financial stability issues since these features have been identified as root causes of systemic failures of the past.[2] Our contribution is to get insights into the effects of an inclusion of the shadow banking sector into the regulatory framework on economic activity and whether such a proceeding would be suitable to internalize the described destabilizing externalities without limiting shadow banking activity per se, i.e. we shed light on how to make the most out of it. Moreover, the present paper is useful to understand the central role of the access to contagion-free, alternative sources of liquidity within the shadow banking sector.

Our experiments provide three main findings. First, our results suggest that switching the regulatory regime from *"regulation by institutional form"* to a *"regulation by function"* meaning the inclusion of shadow banks into the regulatory framework, as proposed by Mehrling (2012), seems to be worthwhile in terms of the internalization of systemic risk.

Second, supervisory authorities should do so in a coordinated and complete manner. A unilateral inclusion, i.e. burdening the shadow banking sector with the same regulatory requirements as traditional banks but denying the access to the public safety net leads to inferior outcomes compared to the benchmark case without shadow banking activity and even to the case in which they are not regulated at all. The results of such cases include negative effects on primary monetary policy goals, significantly increases in the volatility of growth rates as well as financial and real sector default rates. Moreover, a higher volatility in the credit-to-GDP gap can also be observed which is a common indicator for excessive credit growth and, thus, for financial crises.

Finally, experiments with a full and complete inclusion, i.e. with access to a lender of last resort, lead to superior outcomes in terms of the central bank's dual mandate, economic growth and financial stability suggesting that a full inclusion of the shadow banking sector into the regulatory framework could indeed, from a theoretical point of view, lead to a significant mitigation of the destabilizing externalities accompanied by their fragile funding model and to a suitable exploitation of their liquidity provision capacity in terms of sustainable growth.

The remainder of the paper is organized as follows: in section 5.2, we give a brief overview of the currently existing literature on the regulation of shadow banks. Then, in section 5.3, we present an overview of the structure of the underlying ACE macro model followed by a detailed description of the conducted experiments in section 5.4. Section 5.5 provides a discussion of experiment results in terms of macroeconomic and financial stability. Section 5.6 concludes.

5.2 Related Literature

Concerning the existing literature, Meeks et al. (2014) emphasizes in general that, *"[u]ntil now, few papers have attempted to model shadow banking in a macroeconomic context"*. In particular,

[2] Bookstaber (2012) and Battiston et al. (2016) strongly argue in favor of agent-based computational (ACE) frameworks to do research on financial stability and related policy issues. For a good overview on current DSGE models including shadow banking, see Meeks et al. (2014).

the strand on the regulation of shadow banking activity mainly includes either studies that develop principles aiming to guide future regulatory reforms or studies using simple two- or three-period models as well as DSGE models to shed some light on these issues. Hence, to the best of our knowledge, the set of model classes used to explore the effects of shadow banking on economic activity is yet limited to (general) *equilibrium* frameworks. An early three-state formal model is presented by Gennaioli et al. (2013) which builds on the production model from Gennaioli et al. (2012) and introduces shadow banking in order to show that financial innovation has contributed to the build up of systemic risk. Moreover, they show that in a world with shadow banking and myopic investors which systematically neglect tail risks, a sufficiently large degree of maturity transformation and leverage lead to credit booms and busts. di Iasio and Pozsar (2015) use a simple two-period model to analyze capital and liquidity regulation in a market-based intermediation system while Ricks (2010) studies potential approaches to policy intervention within a simple risk model and proposes a risk threshold for financial intermediaries. Additionally, the author discusses the externalities accompanied by the inherently fragile funding scheme of shadow banks. Furthermore, Plantin (2014) shows that the regulatory arbitrage-channel serves as explanation for the massive growth of the shadow banking sector using a simple two-state equilibrium model of optimal bank capital.

Concerning a possible future regulation of shadow banking, Schwarcz (2013, 2012) provides a general assessment of the trade-off between higher efficiency in the financial system through the existence of shadow banks and their contribution to systemic risk. The author argues not to limit shadow banking activity per se and, instead, favors an inclusion of shadow banking activity which should be conducted in such a way that efficiencies are maximized and the contribution to systemic risk is minimized. In this regard, Gorton and Metrick (2012a,b) describe two mechanisms that have led to the collapse of particular sectors in the shadow banking system and Gorton and Metrick (2013) emphasize the important role of the FED in their function as lender of last resort. Moreover, Gorton and Metrick (2010) identify three main factors of shadow banking activity, namely *i)* the emergence of money-market mutual funds (MMFs) that pool retail deposits, *ii)* the securitization process[3] to move assets off balance sheets, and *iii)* repurchase agreements (repos) that facilitated the use of securitized bonds as money. Further, the authors conclude that the key to a regulation of privately created money is a combination of strict guidelines on collateral for securitization and repos as well as a government-guaranteed insurance for MMFs. Finally, Adrian and Ashcraft (2012b) provide a conceptual framework for future regulatory reforms and describe the relevant financial frictions to consider in this regard.

There has also been increasing concern with introducing banking into the DGSE world. These few existing studies mainly focus on the role of credit-supply factors governing credit growth

[3]According to Adrian and Shin (2009), *"[s]ecuritization was intended as a way to transfer credit risk to those better able to absorb losses, but instead it increased the fragility of the entire financial system by allowing banks and other intermediaries to 'leverage up' by buying one another's securities".*

in business cycle fluctuations, i.e. they focus on the role of financial intermediaries rather than on the mechanisms of the borrower or demand-side as, for instance, in the seminal work of Bernanke et al. (1999). The first attempts in this direction are the studies of Gerali et al. (2010); Meh and Moran (2010) and Gertler and Karadi (2011). The authors show the presence of the bank balance sheet channel to improve the DSGE model's fit to the data. However, Meeks et al. (2014) criticize that in these papers, the entire financial system is represented by traditional intermediaries. Thus, they contribute to the literature by constructing a standard dynamic general equilibrium macro model that captures some key features of an economy in which traditional and shadow banks interact by implementing two types of financial intermediaries and a securitization process. In this setting, traditional banks are able to offload their risky loan portfolio onto the shadow banking sector and to trade the securitized assets which allows *"for heterogeneity and specialization in the functions of [financial] intermediaries, generating an additional source of dynamics"*. Within this framework, they analyze responses of aggregate economic activity, the supply of liquidity and credit spreads to business cycle and financial shocks. Another paper to mention is presented by Verona et al. (2013) who introduce shadow banking into a sticky price DSGE model by likewise adding a distinct class of financial intermediaries to study the effect of low interest rates environments on the financial system. However, the approach lacks securitization and there is no direct link between the regulated and unregulated part of the financial system. We also want to highlight the work of Goodhart et al. (2012) who study a wide range of macroprudential tools in a stylized two period general equilibrium model and show how fire sale dynamics can exacerbate financial constraints.

Finally, Arnold et al. (2012) provides an overview of the progress made in measuring systemic risk and of the remaining challenges in that field. Moreover, the authors also discuss in which sense shadow banks represent a significant factor that drives the build up of systemic risk. For a more general view on systemic risk in modern economies, see Montagna (2016).

To the best of our knowledge, there is yet no paper covering shadow banking and its prudential regulation using a comparable (agent-based) approach.

5.3 Model Summary

The paper is primarily focused on the impact of shadow banking on economic activity, excessive credit growth and the prudential regulation of this sector. Hence, due to space constraints, we do not want to burden the text with a full model description. Therefore, the following section only provides a brief overview of the essential parts of the model that are necessary to follow our analysis.

5.3.1 General Characteristics

The basic version of the used stock-flow-consistent agent-based macro model (SFC-ACE) was developed during the work of Krug (2015) where the author analyses the interaction between monetary and macroprudential policy. Figure 5.1 provides an overview of the modeled sectors and the corresponding relationships between types of agents on a monetary level. Thus, the artificial macroeconomy can be characterized by a high degree of financialization in which firms demand credit from the financial sector to finance their production.[4] It consists of six types of agents, i.e. households and firms (real sector), a central bank, a government and a financial supervisory authority[5] (public sector) and a set of traditional banks (financial sector). Agents are heterogeneous in their initial endowments of e.g. productivity, amount of employees or clients and interact through a goods, labor and money market in order to follow their own needs like consuming or making profit. Along the business cycle, the economy follows *Minskyan* dynamics with firms transitioning between various stages of financial soundness, i.e. hedge, speculative and Ponzi finance[6] [Minsky (1986)], representing the root cause for severe financial crises.[7]

Moreover, economic activity is guided by monetary policy which is implemented as usual in developed countries by setting a target rate that directly affects the whole set of existing interest rates, in particular the rates charged on loans to the real sector by means of increased refinancing costs. Through the resulting effect on credit demand, the CB's monetary policy transmits to overall economic activity, i.e. to production and price levels and, thus, to inflation and output.

As a result of the interaction of heterogeneous agents, the model exhibits common macroeconomic stylized facts emerging through the course of the simulation such as endogenous business cycles, GDP growth, unemployment rate fluctuations, balance sheet dynamics, leverage/credit cycles and constraints, bank defaults and financial crises, as well as the need for the public sector to stabilize the economy [shown in Krug (2015)].

For this paper, we extend the basic version of the model in the following way: beside the traditional and regulated banking sector with all its safety net-features like deposit insurance against

[4] Note that in this version of the model, households yet do not demand any credit from the banking sector. In order to be able to analyze the impact of a wider range of macroprudential tools concerning consumer credit, i.e. like the loan-to-value (LTV) or the debt-to-income (DTI) ratio, an extension of the model in this direction would be necessary.

[5] This type of agent is not depicted in figure 5.1 since it is not involved in any monetary flows.

[6] Shadow banking contributes to the shift towards more fragile Minskyan funding forms (speculative and Ponzi) since the lending activity of traditional banks focuses on hedge financed firms by charging a sufficiently high risk premium. However, shadow banks do not fully compensate for a higher default risk of their customers in the same manner and tend to have more lose underwriting standards. Hence, the fraction of fragile funding forms increases with the size of the shadow banking sector and so does overall systemic risk [Chernenko and Sunderam (2014)] .

[7] The share of the three financing schemes proposed by *Minsky* varies over time and is seen as a main source of fluctuations of the *financial cycle* [Drehmann et al. (2012); Adrian and Shin (2008); Claessens et al. (2012); Borio (2014); van der Hoog and Dawid (2015); Strohsal et al. (2015a,b); Galati et al. (2016)].

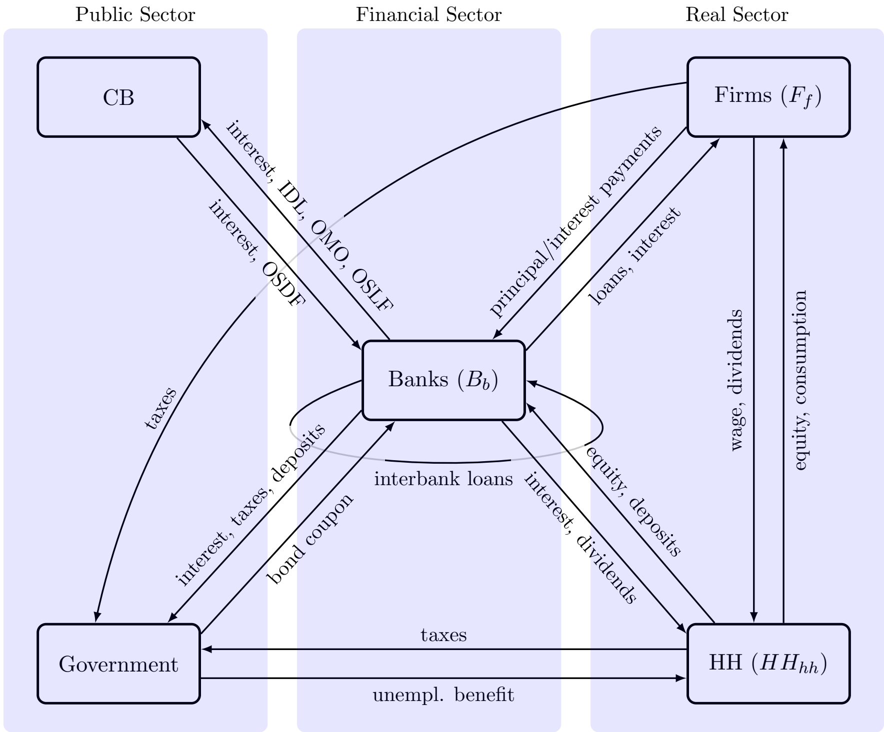


Figure 5.1: Monetary flows in the basic version of the underlying model developed in Krug (2015)

bank runs and the liquidity insurance given by the central bank (LOLR function), we implement a so-called *"parallel banking system"*, i.e. a co-existing financial sub-system comprising of various independent, specialist non-banks raising an interconnected network of balance sheets that operates completely external to regulated banks and the public safety net [Pozsar et al. (2010)]. This sub-system finances itself through investments of HHs since it represents an alternative investment opportunity with a higher yield compared to the interest on deposits paid by traditional banks [see subsubsection 5.3.4.3 for a detailed description of the HH's decision process]. The shadow banking activity is modeled in a way to implement the negative effects of extreme short-term funding structures (wholesale or money market funding), a high degree of pro-cyclicality and the on/off-character of the availability of liquidity in market-based credit systems. Of course, the manifestation of these effects depend on the relative size of the unregulated sub-system and, hence, shadow banking is not a bad thing in itself. Used in a prudential manner, it can even contribute to a prospering economy by serving as an alternative source of liquidity for parts of the real sector that would be credit rationed in the absence of shadow banks [Dombret (2013a, 2014a)]. Pozsar et al. (2010), among others, describe the shadow banking process in great detail, but due to the high degree of complexity and opaqueness, we do not model the whole process with all its dozens of specialist entities involved. For the sake of simplicity, we decide to model just the "head and tail" of the shadow banking process, i.e. we add two classes of agents, one being *"Money-market Mutual Funds (MMF)"* which serves as a cash pool for the investments of the households and *"Broker-dealers (BD)"* who grant loans to firms and finance these via secured (overnight) repos with the MMF. Figure 5.2 shows the extended parts in red color. Subsection 5.3.4 provides a detailed description of the way the shadow banking process is modeled.

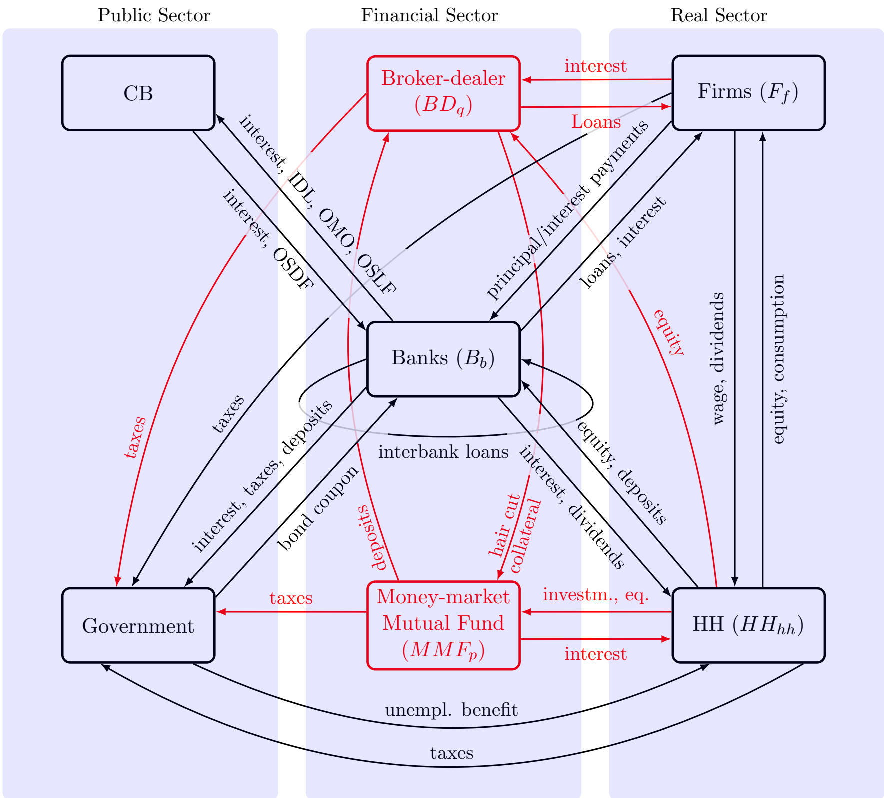


Figure 5.2: Monetary flows in the extended model with shadow banking

5.3.2 Sequence of Simulated Economic Activity (Pseudo Code)

In this section, we show the economic activities as they occur during the simulation process. This should impart a rough idea of the functionality of the underlying agent-based macro-model and its consisting parts. The rest of the section describes these parts in more detail.

1. Start economic interaction of settlement period t ($t = 1, \ldots, 3000$)
    - Banks settle their overnight/short-term interbank liabilities (if any)
    - Banks settle their overnight/short-term standing facility liabilities with the CB (if any)
    - Banks set up repos with CB of maintenance period (if new periods starts)
2. Shadow bank activity

- Reactivation of shadow banks (if any)
- HH adjust their speculative funds
- MMF decide about to roll over their repos
- BD repurchase collateral (if any)
- MMF repay withdrawn funds to HH (if any)
- BD securitize and sell loan portfolio
- BD do new overnight repos with MMF (if any)

3. Real sector activity (planning phase)
   - Reactivation of firms (if any)
   - Firms determine their production target
   - Firms determine their offered wage
   - Firms determine their credit demand (external financing)
   - Firms send credit requests to traditional and shadow banks (sequentially[8])
   - Firms announce vacancies
   - Firms fire employees if they face an overproduction (if any)
4. Government pays unemployment benefit to unemployed HH
5. Real sector activity (production phase)
   - Unemployed HH search for a job / firms hire workers in case of a match
   - Firms produce and offer their bundle of goods
   - HH plan and conduct consumption
6. Real/public sector debt obligations
   - Firms pay wages and meet their debt obligations (risk for firm default due to illiquidity)
   - Government pays principal/interest on outstanding bonds
   - Test for firm default due to insolvency
7. End of settlement period t
   - Banks determine their profit / pay taxes (if any) / pay dividends to HH (if any)
   - Banks repay intra day liquidity (IDL) to the CB (if any)
   - Banks conduct interbank lending (overnight)

[8]Here, sequentially means that firms send credit requests to traditional banks first and in the case of a refusal they try to use the shadow banking sector as alternative source of liquidity.

- Banks use standing facility of the CB
- CB pays interest on reserves
- Test for insolvencies of financial sector agents (trad. banks/shadow banks)
- Government bail out of systemically important (i.e. large traditional) banks

8. Monetary policy decisions
   - CB sets target rate
   - adjustment of the market sentiment parameter (PCL)
   - CB sets counter-cyclical buffer

5.3.3 Settlement Period

The underlying monetary framework of the model follows the theory of endogenous money [see Lavoie (2003) among others], i.e. the amount of money in the system is determined by the investment decisions of real sector agents (demand-driven) instead of the supply of the CB (supply-driven). To model this feature in the most consistent way, we decided to implement a monetary system along the lines of the *UK Sterling Monetary Framework* of the Bank of England (BoE) using it as a template.[9] The orientation seems to be reasonable, since the BoE itself recently attracted attention in the field by implicitly accepting endogenous money theory in their in-house journal, the *BoE Quarterly Bulletin* [McLeay et al. (2014a,b)].

At the heart of the UK reserve averaging scheme[10] lies a *real-time gross settlement* (RTGS) system [Kelsey and Rickenbach (2014); Dent and Dison (2012); Nakajima (2011); Arciero et al. (2009)] which enables the CB to provide liquidity insurance to commercial banks via operational standing facilities (OSF) and, thus, to meet its lender of last resort (LOLR) function. This means that the settlement of a transaction between real sector agents takes place as soon as a payment is submitted into the system (real-time) and that a payment can only be settled if the paying bank has enough funds to deliver the *full* amount in central bank money (gross settlement, i.e. no netting takes place) [Galbiati and Soramäki (2011)].[11] Banks have to finance their reserve accounts for the current maintenance period[12] in advance by setting a target average for their reserve holdings as a fraction of their current interest bearing deposits and by pledging a suitable

[9] A good description can be found in Bank of England (2014b); Ryan-Collins et al. (2012).

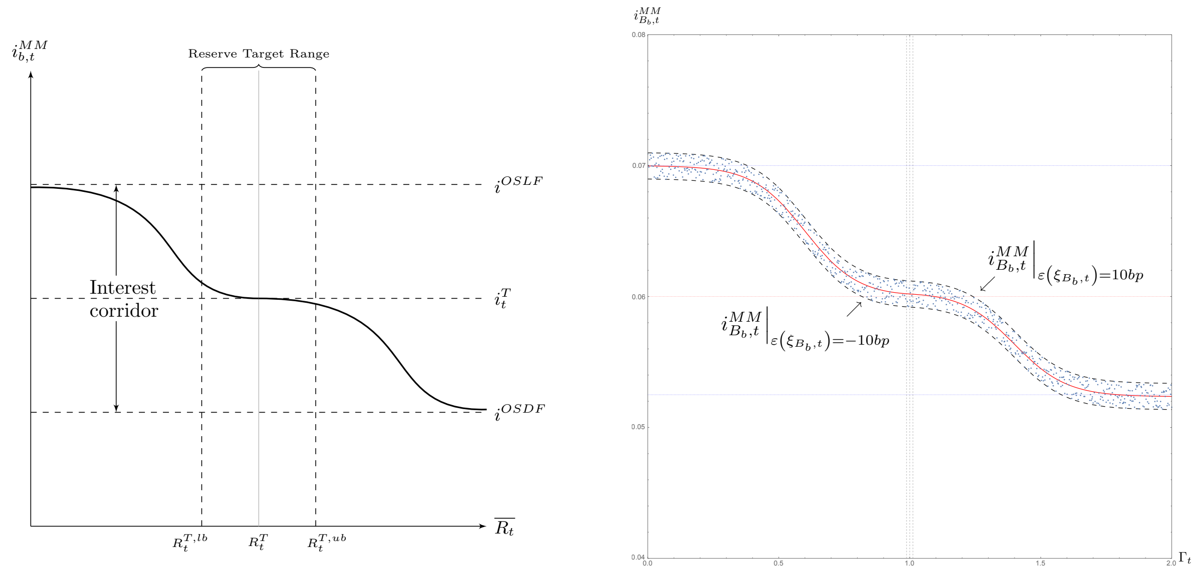
[10] Although it was suspended after the recent financial crisis in 2009 and a Quantitative Easing (QE) scheme is prevailing instead, the reserve averaging scheme can be considered as the default scheme implemented in normal times. With respect to the aim of the model, i.e. to evaluate monetary policies contribution to financial stability, a scheme with a comparable setting to the pre-crises period of 2007/2008 seems to be a reasonable choice.

[11] We suppose that all transactions in the overdraft economy are conducted by only using scriptural money, i.e. there exist no banknotes (cashless economy).

[12] The maintenance period means the time between the target rate decisions of CB. In reality, the maintenance period of the BoE lasts 4 weeks and banks have to settle their reserve accounts with the BoE at the end of each business day. Hence, the modeled maintenance period lasts for 4 settlement periods.

amount of collateral with the CB [Ryan-Collins et al. (2012)]. In turn, banks' reserve holdings are remunerated at the CB's target rate i_t^* on a period average basis. For that reason, the CB defines a narrow 1%-range around the individual target balance of each bank and depending on whether the bank has met its reserve target range or not, it will be credited with the interest earned against its average balance at the end of each maintenance period.

However, through the course of the maintenance period, each bank faces an unpredictable stream of transactions between real sector agents each affecting banks' reserve balances. Thus, economic activity usually leads banks to end up with an average reserve balance outside of their reserve target range, i.e. with either excess reserves or a reserve deficit. To ensure the compliance with the target range, banks are encouraged to appropriately manage their liquidity. By charging a premium (discount) on the target rate i_t^* for the usage of its lending (deposit) facility, the CB builds an interest corridor which ensures that banks seek money first in the open (interbank) money market and reallocate outstanding reserves through overnight repos with peers before turning to the CB's standing facilities[13] [compare Lavoie (2003)].



(a) Corridor System of reserves (theoretical)

(b) System within the model

Figure 5.3: Money market rate, banks' demand for reserves and the interest corridor of the CB [Bank of England (2014b); Ryan-Collins et al. (2012); Winters (2012)]

We model the interbank market as a (decentralized) over-the-counter (OTC) market which requires bank b (in need of reserves) to find a counterparty within the set of all other banks that is willing to lend reserves to b [Afonso and Lagos (2013)]. The conditions for overnight

[13]Beside the standing facilities, the liquidity insurance of the CB also encompasses secured short-term repos for banks in need of reserves *during the course of the settlement period*. These reserves are referred to *intraday liquidity* (IDL) and have to be repaid at the end of the settlement period just before banks take action to meet their individual reserve target range [Bank of England (2014a); Dent and Dison (2012); Ryan-Collins et al. (2012)]. So, the provision of IDL ensures that any payment of a banks' client can be settled in real-time and on a gross basis.

interbank repos are then based on bilateral negotiation about volume and interest charged ($i_{b,t}^{MM}$). Whereas the volume depends on the counterparty's current excess reserves, the money market rate $i_{b,t}^{MM}$ faced by b depends on i_t^*, on the current financial soundness of bank b and on the current supply of excess reserves on the money market expressed by

$$\Gamma_t = \frac{\sum_{b=1}^{B} \overline{R_{b,t}}}{\sum_{b=1}^{B} R_{b,t}^*} = \frac{\overline{R_t}}{R_t^*} \tag{5.1}$$

which serves as a measure for how far the current aggregate average reserves ($\overline{R_t}$) are away from the aggregate reserve target (R_t^*). Hence, the prevailing incentives scheme shown in figure 5.3a leads to an individual money market rate for bank b of

$$\begin{aligned} i_{b,t}^{MM}\left(i_t^*,\Gamma_t,\xi_{b,t}\right) = \\ &\left\{ g\left(\Gamma_t\right)\left[\sigma_1 - \sigma_2 \cdot \tanh\left(\varphi\Gamma_t - \frac{3}{2}\varphi\right)\right] + \Big(1 - g\left(\Gamma_t\right)\Big)\left[\sigma_3 - \sigma_4 \cdot \tanh\left(\varphi\Gamma_t - \frac{\varphi}{2}\right)\right]\right\} \\ &- \left(0.06 - i_t^*\right) + \varepsilon\left(\xi_{b,t}\right) \end{aligned} \tag{5.2}$$

with

$$g(\Gamma_t) = \frac{1}{2} + \frac{1}{2}\tanh\left(\frac{\Gamma_t - 1}{0.1}\right) \tag{5.3}$$

as well as $\varepsilon\left(\xi_{b,t}\right)$ representing a small risk premium/discount (between +10 and -10 basis points) depending on b's financial soundness measured by its D/E-ratio $\xi_{b,t}$. Hence, realizations of $i_{b,t}^{MM}$ fall within the scope of a small band around $i_{b,t}^{MM}\Big|_{\varepsilon\left(\xi_{b,t}\right)=0}$ (figure 5.3b shows this exemplary for $\Gamma_t \in (0,2)$). Table 5.1 shows the corresponding interest corridor build by the lending/deposit facility rates which depends on the current target rate i_t^* as well as the parameter sets for σ_1, σ_2, σ_3 and σ_4.[14]

Note that the reserve allocation process of the model's payment system is not perfect in the sense that the search for a counterparty with excess reserves is not always successful. This can be for various reasons, for instance, the banks with excess reserves do not want to lend to other banks because they have to offset a former deficit state or they show, in general, a highly risk-averse behavior in the aftermath of a default of a peer. Such a behavior corresponds with the freeze of the interbank market that could have been observed after the default of Lehman Brothers. Another reason could be that the bank in need of reserves has a very bad financial soundness and only this bank is forced to turn to the central bank while others are still able to obtain reserves from peers.

[14] We calibrated the parameters according to data on the interest rate corridor of the BoE and the FED which show that the corridor widens with an increasing target rate.

Table 5.1: Parameter sets determining the level of the CB's interest corridor

i_t^{OSDF}	i_t^*	i_t^{OSLF}	σ_1	σ_2	σ_3	σ_4
$i_t^* - 0.75\%$	$i_t^* \geq 5\%$	$i_t^* + 1\%$	$\sigma_3 - 0.00865$	0.004	0.065	0.005
$i_t^* - 0.45\%$	$i_t^* \leq 5\%$	$i_t^* + 0.5\%$	$\sigma_3 - 0.005$	0.0025	0.0625	0.0025
$\max(i_t^* - 0.25\%, 0.25\%)$	$i_t^* < 3\%$	$i_t^* + 0.25\%$	$\sigma_3 - 0.0025$	0.00125	0.06125	0.00125

5.3.4 Shadow Banking

Shadow Banking mimics the traditional financial intermediation process by disassembling it into its parts or services and by providing every service through a highly specialized and unregulated entity. This proceeding is not only very complex in nature, it is also accompanied by several sources of systemic risk well-known from banking in the 19$^{\text{th}}$ century when the first central banks where established to regulate the fully free operating banking sector, in particular, to mitigate the negative externalities of excessive maturity and liquidity mismatches [Haldane and Qvigstad (2014); Mehrling et al. (2013)].

Hence, these sources mainly include the susceptibility to runs due to the lack of an appropriate (deposit) insurance scheme [Gorton and Metrick (2012b)], extreme levels of leverage as well as the immense liquidity or roll-over risk faced by shadow banks in combination with the lacking access to a LOLR-institution. In particular, the predominant reliance on institutional funds and its concentration in wholesale funding markets play an important role. Unlike retail deposits, these funds are well-informed, herd-like, i.e. highly sensitive to news, and badly diversified. This mainly stems from the fact that the institutional investor's intention is yield rather than storing and security. Another issue contributing to the fragility of the shadow banking system is the form of withdrawals. The predictability of retail-deposit withdrawals is much higher since they require an active decision of the depositor to withdraw funds from its account. In wholesale funding markets where (overnight) repos are the contractual form of choice, it is the exact opposite, i.e. investors have to decide actively about the roll-over of their lent funds. For traditional banks, the analogous situation would be that every depositor would have to actively decide and communicate every evening whether he still agrees to place his funds with the bank until the next day or not, and moreover, if he does nothing at all, the money would automatically be withdrawn from the bank.

As such, we frame shadow banks as *unregulated and extremely leveraged entities without any link to resilient, contagion-free liquidity sources or insurance schemes that exhibit a wholesale funding model which is highly exposed to the fickle and herd-like decisions of investors and revulsions in overall market sentiment.*

According to [Pozsar et al. (2010); Pozsar (2014)] there is usually an entity which serves as an institutional cash pool, like a pension, hedge or money-market fund promising a relatively safe but higher yield compared to traditional banks. To earn the promised yield, the fund lends the collected funds against collateral (typically via secured overnight repos) to other entities that are in need of liquidity and have large amounts of securitized assets on their balance sheets [Chernenko and Sunderam (2014); Dombret (2014a)]. These entities build the core of the highly complex shadow banking process and for the sake of simplicity, we follow the approach of previous studies in the field and do not explicitly model this process in great detail [Meeks et al. (2014), among others]. At the other end of the process, one typically finds entities that provide liquidity to the real sector, like a broker-dealer [Rosengren (2014)], but do not want to hold the highly illiquid assets until maturity on their balance sheets in order to avoid the risks stemming from credit, liquidity and maturity transformation accompanied with traditional financial intermediation [Pozsar (2015)]. That is why these assets are distributed through the securitization process finally ending up at the cash pooling fund and the liquidity from the fund ends up at the broker-dealer completing the shadow banking intermediation process. Thus, we explicitly model the *head and tail* of this process by introducing two new classes of agents, i.e. a money-market mutual fund (MMF) that pools the cash of investors and a broker-dealer (BD) that serves as alternative source for credit for the real sector. The latter finances itself through extremely short-term (overnight) repos with the MMF. Figure 5.4 shows the differences between the traditional and shadow banking intermediation process in the model.

The rest of the section describes the business of these new types of agents and their range of activities in more detail, followed by a description of the investment decision of HHs.

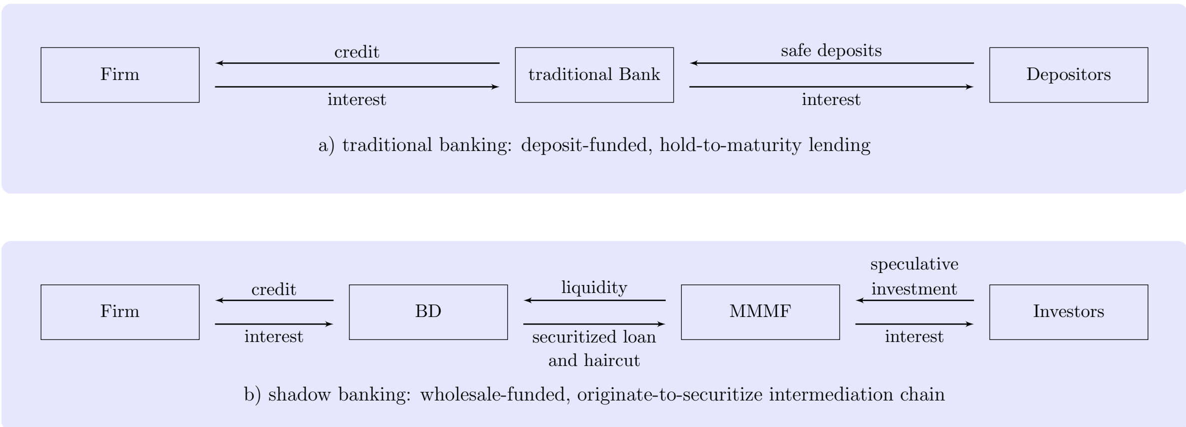


Figure 5.4: Lending activity in the traditional and shadow banking sector

5.3.4.1 Money-market Mutual Funds (MMF) – The Cash Pool

Dombret (2014a) vividly describes the fragility of MMF by mentioning that, from an investor's point of view, they bear a strong resemblance to traditional banks since there is very little difference between the investment into an MMF and a bank account. In general, both balances are available on demand. But he argues that

> *"the main problem comes with money market funds which operate with "constant share values", such that investor deposits have a constant value. With funds like this, losses are not distributed evenly across all investors. Instead, a first come first served rule applies. Those who withdraw their deposits first get back the full amount, while those who act too late have to accept corresponding losses. This rule makes such money market funds susceptible to runs".*

Moreover, real sector agents typically do not invest *directly* in the money market. Instead, they place their money with an MMF that pools (private and public) funds and then invests large volumes in the money market with the promise of redemption at par and on-demand. Nevertheless, this promise is not supported by any amount of capital.

Assets	Liabilities
Repos ($RC_{v,t}$)	Retail Deposits ($RD_{v,t}$)
Bank Deposits ($D_{v,t}$)	Interest Obl. ($IO_{v,t}$)
Gov. Bonds ($GB_{v,t}$)	
Interest Receiv. ($IR_{v,t}$)	Equity ($E_{v,t}$)
Total Assets ($TA_{v,t}$)	

(a) Balance Sheet 6: Example MMF v

Assets	Liabilities
Business Loans ($BL_{u,t}$)	Repos ($RL_{u,t}$)
Bank Deposits ($D_{u,t}$)	
Gov. Bonds ($GB_{u,t}$)	
Interest Receiv. ($IR_{u,t}$)	Equity ($E_{u,t}$)
Total Assets ($TA_{u,t}$)	

(b) Balance Sheet 7: Example BD u

Figure 5.5: Balance sheet structure of shadow banking agents

The initial investment of HHs is incentivized by the fact that the MMF offer slightly more interest than traditional banks. More detailed information about the interest level can be found in subsection 5.3.6.

If the MMF has collected a sufficient amount of funds at its account, it offers them at the money market for secured repo lending. The repo includes the exchange of securities against funds and the MMF earns a fee, namely the haircut, which can be seen as the interest on the loan to the broker-dealer. From an accounting point of view, this means the MMF raises a claim on the securities that still remain at the balance sheet of the broker-dealer. The BD only gets funds worth a fraction of the collateral whereas the difference is the haircut. The haircut usually lies about 100 basis points above the interest the MMF pays to its investors.

If, for any reason, some HHs decide to (full or partly) withdraw their investments from the MMF (the decision process of HH is described in subsubsection 5.3.4.3), the MMF checks whether it currently has the needed liquidity to meet the demand of the HHs. If it has not, it stops to roll-over a sufficient amount of repos which forces some broker-dealers to repurchase their pledged collateral. This might turn into financial pressure on the broker-dealer since its balance sheet typically shows a significant maturity mismatch. Unfortunately, it lacks the opportunity to get CB liquidity, thus, it is forced to fire sale some of its assets at a discount depending on the number of recent BD defaults. If the fire sale does not generate enough funds to repurchase the collateral, the broker-dealer is forced into default due to illiquidity and the MMF has the opportunity to fire sale the collateral and internalize the corresponding loss. If the MMF cannot meet the withdrawals of its investors, it also defaults and is resolved passing the loss over to the investors.

5.3.4.2 Broker-Dealer – The Non-bank Provider of Credit

Our aim is to implement the typical broker-dealer funding model with all associated risks as described in e.g. Rosengren (2014). It includes large balance sheets with risky long-term assets mainly funded at low costs, i.e. short-term fully collateralized loans at a quite low interest or haircut (repurchase agreements). Unfortunately, such a business model requires prospering and booming phases in order to be profitable and highly depends on the availability of liquidity to roll over the broker-dealer's debt. However, during times of financial distress, that low-cost funding quickly evaporates. In this regard, Rosengren (2014) states that

> *"[d]uring the financial crisis, we saw that many of those who traditionally lent to broker-dealers feared default by a broker-dealer – and did not want to risk having to take possession of the collateral associated with the repurchase agreement in the event of a default. In fact, money market mutual funds, one of the largest sources of lending to broker-dealers, are prohibited from purchasing the kind of long-term or high-credit-risk assets that are sometimes pledged as collateral for loans to broker-dealers. [...] The result is that broker-dealers can experience significant funding problems during times of financial stress".*

The economic activity of broker-dealers in the model can be described as follows: After its foundation, the broker-dealer grants initial loans to firms and securitizes the resulting long-term asset in order to place it as collateral for a repo with a MMF. The new liquidity can now be used for further loans proceeding in the same way while balance sheets expand and profit rise.

Regulatory tools are designed to prevent from greedy tendencies gaining the upper hand, in particular during prospering phases, and, hence, a significant share of the credit demand cannot be met by traditional banks. Due to the mentioned cost advantages of its intermediation strategy, the broker-dealer can offer loans at more favorable conditions to firms than traditional banks. More detailed information about the interest level can be found in subsection 5.3.6. Another point that increases the attractiveness of shadow banks is that they have rather loose underwriting standards since they are not forced to comply with corresponding regulatory requirements and usually distribute the originated assets through securitization. Hence, the modeled broker-dealer agents cover this feature by neglecting the evaluation of its client's creditworthiness. As a consequence and since every credit request represents an opportunity to make profit, the only channel that restricts the lending activity is the lack of sufficiently liquid MMFs. This comes to the fact that the shadow banking sector also finances the less creditworthy part of the real sector while traditional banks are incentivized not to lend to these firms through regulation. Thus, increasing shadow bank activity not just negatively affects the distribution of the Minskyan financing schemes towards instability by itself, but also by functioning as an amplifier through lending to financially unsound firms.

5.3.4.3 Investment Decision of Households

The extension of the model by shadow banking also includes an alternative investment opportunity for HHs in MMFs instead of just leaving their funds at traditional banks. This section describes the decision process involved.

Once a month, each HH decides on whether to adjust its investment into the shadow banking sector or not. This involves a two-stage-decision process where the result depends on both the recent development of the market sentiment and household's individual degree of risk aversion. The overall market sentiment[15] is modeled by a *public confidence level* (PCL),[16] i.e. the agents' expectations about the future economic activity within the artificial economy. This market sentiment negatively depends on the prevailing interest environment with the central banks' target rate at its core. This is in line with the risk channel-theory which says that a low-interest environment leads to a *seek-for-yield* behavior accompanied by a higher risk tolerance of market participants [Borio and Zhu (2012)].

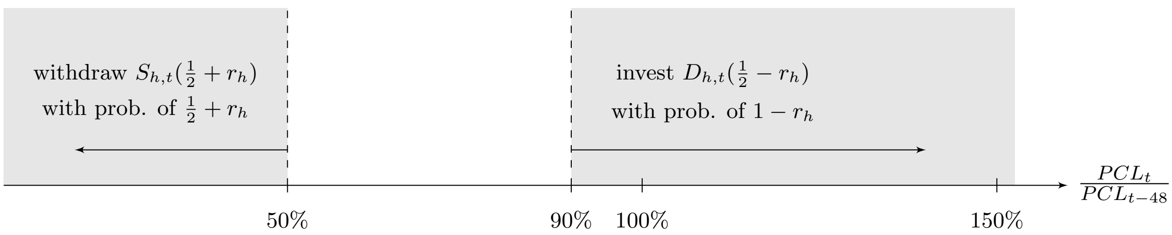


Figure 5.6: Investment decision of HH h in t
r_h represents the risk-aversion parameter of HH h which is randomly distributed between 0 and 0.5 and stays fixed for the rest of the simulation, $S_{h,t}$:= already invested funds of HH h in t, $D_{h,t}$:= fraction of deposits of HH h in t held at its traditional bank account available for speculative investments.

To model the typical inherent myopia of investor's decisions, we link the investor's assessment of the current market situation to the short-run development of the market sentiment, i.e. HHs compare the current level of market sentiment (PCL_t) with its development during the recent

[15]The approach of an endogenous market sentiment has some analogy with switching mechanisms resulting from agents' limited capacity to process information (bounded rationality of agents) used, for instance, in De Grauwe (2011); Lengnick and Wohltmann (2016), among others. In these papers, agents endogenously switch between optimistic and pessimistic sentiments or between acting as chartists and fundamentalists on the financial markets.

[16]A comparable index would be the German *Ifo-Index* of the Munich Economic Institute which also calls market participants and asks them for their current evaluation of the market sentiment.

past, i.e. with the level one year ago (PCL_{t-48}).[17] Hence, the PCL depends on and reacts to (short-run) changes of the central bank's target rate:

$$PCL_t\left(i_t^T\right) = 1.1 - 10 i_t^T. \tag{5.4}$$

In this regard, one could say that HHs act similar to chartists known from the financial markets literature and that their behavior is mainly driven by "animal spirits" [Keynes (1936); Akerlof and Shiller (2009)]. Figure 5.6 shows that if the change in market sentiment, either positive or negative, is relatively large, it then depends on the household's individual risk-aversion parameter r_h whether it immediately responds to the changes or not. For instance, if the overall market sentiment has declined sufficiently, the probability to *withdraw* its funds from the MMF increases with r_h, while the probability to *invest* negatively depends on r_h during euphoric times.

In a second step, after the HH has decided to react to the changes in market sentiment, it decides about the amount to invest/withdraw:

$$\frac{PCL_t}{PCL_{t-48}} = \begin{cases} > 0.9 & \Longrightarrow \text{invest } D_{h,t}(\frac{1}{2} - r_h) \text{ with prob. of } 1 - r_h \\ < 0.5 & \Longrightarrow \text{withdraw } S_{h,t}(\frac{1}{2} + r_h) \text{ with prob. of } \frac{1}{2} + r_h \\ \text{otherwise} & \Longrightarrow \text{do nothing} \end{cases}, \tag{5.5}$$

Hence, the HH's assessment represents a rather myopic and local consideration of the market which represents well-known phenomena like highly pro-cyclical and herding behavior of market participants. Since HHs make their investment decision in such a boundedly rational way, they also want to invest into the shadow banking sector at low interest levels as long as the PCL_t exceeds the PCL_{t-48} by a sufficient amount. HHs then decide to either invest more, withdraw a fraction of their already invested funds or leave their investment at the current level. Figure 5.7 shows the typical highly erratic development of funds invested in the shadow banking sector. A common decision to withdraw leads to runs on MMF triggering a highly contagious chain of deleveraging processes among financial sector agents.

5.3.5 Real Sector Activity

At first, firms plan their production for the period as well as the corresponding costs (including wages) which, in turn, determines their current credit demand. The planned production is based on a target value for the firm's capacity utilization, i.e. it depends on average sales of past periods and a surcharge to cope with demand fluctuations. Moreover, the production function

[17] Note, that the periods within the model represent weeks and that a modeled year has $12 * 4 = 48$ weeks. Thus, a value of the previous year has the index $t - 48$ while a value of the previous quarter has the index $t - 12$.

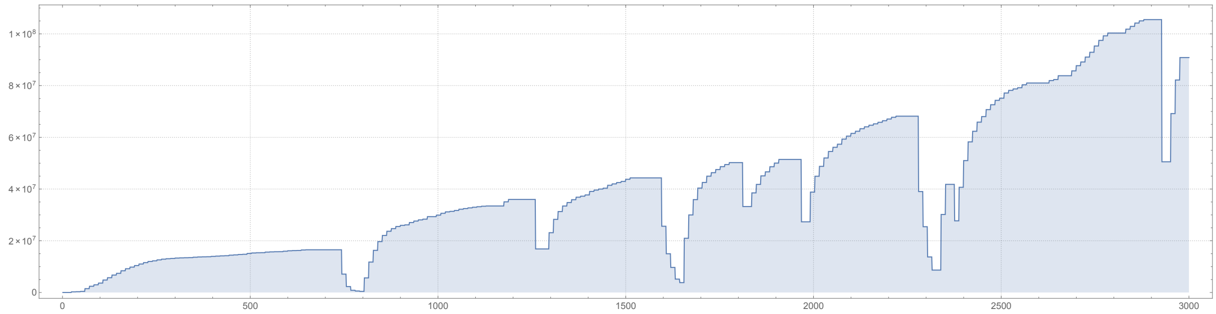


Figure 5.7: Typical development of invested funds in the shadow banking sector within the model

for the period output faced by each firm is of the Cobb-Douglas-type

$$q_{f,t} = (A_t \Psi_{f,t})^{1-\alpha} \tag{5.6}$$

with aggregate labor skill currently used by firm f ($\Psi_{f,t}$) as input and technology parameter A_t representing technological progress[18] since labor productivity of HHs grows at a constant exogenous rate of $g_A = 0.012$ annually (or $g_A^Q = 0.003$ per quarter), i.e.

$$A_t = A_{t-12} \exp\left(g_A^Q\right). \tag{5.7}$$

When plans are completed, firms request credit from traditional or shadow banks (this is described in more detail in subsection 5.3.6) and announce vacancies depending on their financial resources. The firm's ability to meet its labor demand influences the offered wage of the subsequent periods accordingly.

At this stage, unemployed HHs receive unemployment benefit from the government[19] and start searching for a job. If there is a match between the offered amount of labor skill of a HH and the labor demand of a firm, the HH is hired and stays unemployed otherwise. Then production takes place according to the firm's current production capacity. After production is completed, the output[20] is offered on the goods market at retail prices $p_{f,t}$ that account for (individual) expected unit costs including a mark-up ($\mu > 1$) as well as expected inflation (π_t^e)

$$p_{f,t} = (\mu + \pi_t^e) \cdot \frac{12 \cdot q_{f,t}^{-1}(q_{f,t}^*) w_{f,t} + \mathcal{L}_{f,t} i_{b,f,t}}{12 \cdot q_{f,t}^*}. \tag{5.8}$$

[18] The technology of firms follows the work of Stolzenburg (2015) where the author implements parts of the famous *Solow growth model* [Solow (1956)] into an agent-based framework.

[19] The government expenditures for unemployment benefit to HH and interest on outstanding public debt are financed by raising income taxes on wages ($\tau^I = 30\%$), a VAT on the consumption of goods ($\tau^{VAT} = 20\%$), a corporate tax on profits of firms, traditional and shadow banks ($\tau^C = 60\%$), and a tax on capital gains ($\tau^{CG} = 25\%$).

[20] One unit represents a whole bundle of goods in order to also be able to consume continuous instead of just discrete values of goods.

Expected unit costs include wages denoted by $w_{f,t}$ and scaled by the produced quantity $q^{-1}_{f,t}(q^{*}_{f,t})$ as well as cost of debt denoted by $\mathcal{L}_{f,t} i_{b,f,t}$. Price revisions occur once a year.

HHs plan their period consumption level, $c^{p}_{h,t}$, and update it once a quarter. It is composed of an autonomous part

$$c^{a}_{h,t} = 0.18 \cdot \frac{1}{F} \sum_{f=1}^{F} w_{f,t-12} \tag{5.9}$$

co-varying with the average wage of the previous quarter and a part depending more on the current individual financial situation of HH h, i.e.

$$c^{p}_{h,t} = \min \left[D_{h,t},\, \eta c^{p}_{h,t-12} + (1-\eta)(c^{a}_{h,t} + \eta \overline{I_{h,t-12}}) \right] \qquad \text{with } \eta = 0.9 \tag{5.10}$$

where η represents the HH's adjustment speed to new levels of income and $\overline{I_{h,t-12}}$ the average income of the previous quarter including received wages, interest on deposits as well as dividends on an accrual basis. The planned consumption level only deviates from the actual level $c_{h,t}$ in the case in which h cannot afford to consume $c^{p}_{h,t}$ due to the lack of money or it is not able to do so due to a lack of goods supply. The HH's sources of income include a mix of wages and unemployment benefits depending on how long it was unemployed until t as well as interest on its deposits. Moreover, at the end of each fiscal year, firms and banks (partially) distribute their profits in form of dividends to HHs.

Firms use the generated revenues to pay wages and, if any, to settle due parts of their obligations from loan contracts, i.e. they make principal payments and pay interest to the bank. If a firm is not able to meet its debt obligations, it exits the market and all financial claims are cleared in such a way that banks have to depreciate the outstanding loans after receiving the proceeds of the liquidation of the firm's assets, if any, and owners (HH) lose their share of the firm's equity. Moreover, all employees loose their jobs. Assuming that the bankruptcy of a firm happened in period t, a new firm enters the market in $t + 24 + \varrho$ (where ϱ is a positive uniformly distributed integer between zero and 48) given that there exists a sufficiently large group of investors.[21] If all goes well and the firm meets its obligations until the end of the fiscal year, it determines the profit before taxation

$$\Pi^{bt}_{f,t} = s_f \cdot p_f - \left(i^{debt}_{f} + \Psi_f w_f \right) \tag{5.11}$$

where the cost of goods sold include due interest on outstanding debt i^{debt}_{f} and labor costs of the fiscal year (for a detailed description of interest rates charged on loans, see section 5.3.6). In the case of $\Pi_{f,t} > 0$, firms are burdened by the government with a corporate tax so that the

[21] Firms which are shut down, do not vanish from the economy. In order to ensure the stock-flow consistency of the model, these firms are just inactive until a new group of HH (investors) has enough capital for reactivation [Dawid et al. (2014)].

profit after tax results from

$$\Pi^{at}_{f,t} = (1 - \tau^C)\Pi^{bt}_{f,t} \qquad (\text{with } \tau^C = 0.6). \tag{5.12}$$

From the remaining profit after taxation, $\theta\Pi^{at}_{f,t}$ serves as retained earnings to strengthen the internal financing capacity while the residual of $(1 - \theta)\Pi^{at}_{f,t}$ (with $\theta = 0.9$) is distributed as dividends to equity holders.

5.3.6 Credit Market and Interest Environment

Firms in need of external financing send a credit request to a (traditional) bank which then decides on the interest to charge on the loan. The interest depends on the firm's ability to generate sufficient cash flow during the past fiscal year in order to meet its potential future debt obligations.[22] Now firms can evaluate on the profitability of the investment given the offered loan conditions. This decision is based on the internal rate of return which is represented by the fact that the firm's probability to take the loan ($\mathcal{L}_{f,t}$) under the offered conditions negatively depends on the offered interest rate $i_{b,f,t}$, i.e.

$$\Pr\left(\mathcal{L}_{f,t} \mid i_{b,f,t}\right) = \max\left[1.8 - 7.5 i_{b,f,t},\, 0\right]. \tag{5.13}$$

Hence, there might be cases in which the added risk premium is so high (due to the inadequacy of the firm's latest cash flow statement) that it decides to refuse the loan offer. If a firm is credit rationed for this or any other reason[23] by a traditional bank, it tries to finance its planned production with funds from the shadow banking sector which is able to offer more attractive loan conditions than the regulated banking system.[24] Moreover, shadow banks have less incentives to ensure high quality underwriting standards because they do not hold their originated loans after its securitization. If the firm is not even able to acquire the needed funds from shadow banks, it can only employ an amount of workers appropriate to its internal financing capacity.

In addition to the liquidity provision to the real sector, traditional banks have also other opportunities to generate profits. In general, they do so by exploiting the prevailing interest spreads. We want to give a more intuitive picture of the interest environment into which agents are embedded by means of Figure 5.8. The shown spreads form an incentive scheme for the banking

[22] There is also the possibility of only *partially* granting the requested loan, but following a survey of the ECB, these cases are only of minor importance. The decision process used here represents over 80% of decisions made by banks within the Euro area [ECB (2010)]. The decision process of banks concerning the granting of loans is described in detail in subsection 5.3.6.

[23] Traditional banks may reject a loan request directly without evaluation of the firm's ability to create sufficient cash flows to repay the funds because of regulatory requirements.

[24] This is in line with empirical observations, since the unregulated part of the financial system exhibits much more flexibility compared to the traditional banking system facing increasing competitiveness instead [Hoenig (1996)].

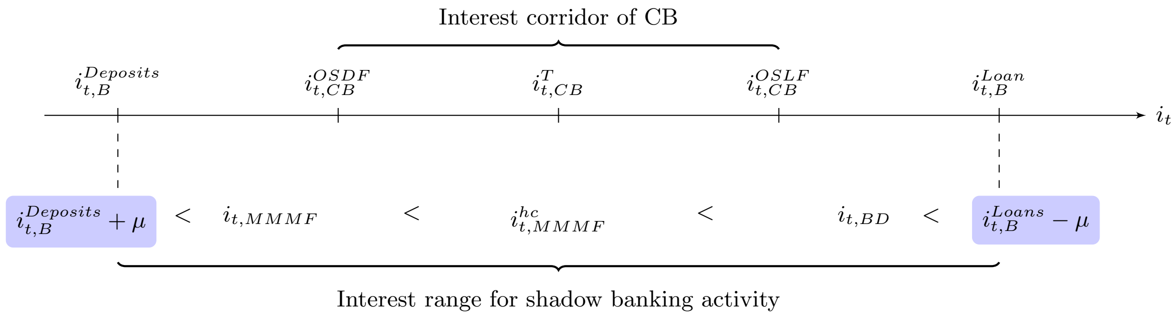


Figure 5.8: Interest spreads on the credit/money market

sector that determines what to do with its lending capacity, i.e. since $i_{t,B}^{Loan} > i_{t,CB}^{T} > i_{t,CB}^{OSDF}$ holds, meeting the real sector's demand for credit has the highest priority whereas lending excess reserves to peers or placing them at the CB are subordinated.[25] Hence, the larger the spread between the interest paid on deposits ($i_{t,B}^{Deposits}$) and the interest charged on loans ($i_{t,B}^{Loan}$) is, the more profitable is the traditional banking business. However, as a side-effect, this profit-maximizing behavior of traditional *universal* banks creates huge incentives for alternative forms of financial intermediaries to enter the market. Since shadow banking mimics traditional financial intermediation by providing every of the several services of the intermediation process through an independent, unregulated and highly specialized financial entity instead of providing the whole range of financial services by a single institution, they can do it at much lower costs[26] and, thus, are able to operate in a much more flexible business environment. As a consequence, the profit potential and the incentive to compete with universal banks for market share is huge which can be seen as an explanation for the boom in the shadow banking activity during the last two decades.

Hence, to complete the described incentive scheme for the traditional banks, we have to implement a corresponding scheme for shadow banks in a consistent way. Thus, assuming even similar operating costs, they make profit as long as their whole lending process includes an interest spread ranging between $i_{i,B}^{Deposits} + \mu$ and $i_{i,B}^{Loans} - \mu$ with $\mu > 0$. In order to attract funds from investors, shadow banks must pay a higher interest compared to the interest on deposits paid by traditional banks, i.e. $i_{i,B}^{Deposits} + \mu$. At the same time, the interest charged on loans should be marginally lower than the rates charged by traditional banks to attract credit demand from the real sector, i.e. $i_{i,B}^{Loans} - \mu$. Since the modeled shadow banking process consists of two

[25] A monetary framework with such an incentive scheme at its heart may have pitfalls. The recent past has shown that the European Central Bank's power to encourage the lending activity to the real sector in a low-interest environment (near the ZLB) is limited as the ECB actually wasn't able to force banks to use the provided liquidity for loans to the real sector even by charging instead of paying interest on excess reserves deposited at the central bank, i.e. $i_{t,CB}^{OSDF} < 0$ instead of $i_{t,CB}^{OSDF} > 0$.

[26] Due to the fact that shadow banks do not have to comply with regulatory requirements concerning their balance sheet structure, the types of asset classes they hold or their level of leverage, they are highly attractive because they usually are able to accomplish a much higher ROE since they make profits on a much smaller capital base, at least, as long as markets are liquid and the sensitivity to risk is low due to a euphoric market sentiment.

entities, the rates charged on each other for their specific services must also fall into this spread, i.e. the rate charged by the MMF for the (overnight) repo with the broker-dealer (haircut) must exceed the interest paid to investors. Accordingly, the interest charged by the broker-dealer on the loans must be lower than that of traditional banks but also higher than the haircut paid to the MMF for the repo.

5.3.7 Foundation and Bankruptcy

The initial bilateral relationships between financial and real sector agents are assigned randomly, i.e. each household and firm chooses a traditional/shadow bank where it places its deposits, requests loans or decides to place investments. These relationships do only change in the case of a default of an agent.

In general, there are two underlying causes for defaults of real and financial sector agents in the economy, i.e. illiquidity and insolvency. For instance, if a firm does not have sufficient funds to pay wages or it is not able to meet its debt obligations, it defaults due to illiquidity. Especially shadow banks face a significant liquidity risk due to the highly pro-cyclical and fragile character of their funding sources and the missing link to a liquidity backstop. Moreover, at the end of each settlement period, agents compute their profits, and update their income statements and balance sheets in order to determine their individual period obligations concerning debt financing, taxes and dividends. After these assessments, agents might conclude that the revenues of the last couple of periods might have been sufficiently low and that, as a consequence, the net worth has turned negative, i.e. the agent has to declare its default due to insolvency. In either case, the malfunction leads to a shut down of the firm's operating business entailing the resolution of all its economic relationships and commitments as well as its final liquidation.

In the case of a threatening default of a systemically important bank (SIB), i.e. of a bank that has significant market share[27] and, thus, a crucial role for the functioning of the payment system, the government bails out the institution in distress by issuing new government bonds and waiving of deposits in order to provide the needed capital. In turn, the government becomes a shareholder of the bailed out bank and tries to sell its shares to investors in future periods. In the case of a default of a (sufficiently small) bank, all clients of the insolvent bank randomly choose a new bank and if a new founded bank enters the market, clients of other banks have a small probability to switch. New firms also form their bank relationships randomly.

[27] For simplicity, the market share of a bank is approximated by its size in terms of total assets. The threshold for a bank being classified as systemically important is set at the inverse of the number of banks meaning that an insolvent bank lying above that threshold is bailed out since it represents a significant part of the payment system. As a result, the probability for banks to be bailed out by the government increases with the bank defaults that already happened. For five banks, this would be 20%.

5.3.8 Financial Regulation

The financial supervisory authority agent aims to ensure the growth-supportive capacity of the financial sector by imposing micro- and macroprudential capital requirements on traditional banks according to the Basel III accord [Krug et al. (2015)] while the shadow banking sector does not face any regulatory requirements at all.[28] Hence, traditional banks have to comply simultaneously with the risk-sensitive measures of

- a core capital ratio of 4.5%
- that is extended by the capital conservation buffer (CConB) of 2.5% and
- a counter-cyclical buffer (CCycB) of 2.5% which is set by the CB according to the rule described in Basel Committee on Banking Supervision (BCBS) (2010); Drehmann and Tsatsaronis (2014); Agénor et al. (2013); Drehmann et al. (2010),[29]
- surcharges on systemically important banks (SIB) using the banks' market share as an indicator as well as
- a (non-risk sensitive) leverage ratio of 3%.

The risk-sensitive measures require a minimum amount of capital in relation to the banks' exposure to (credit) risk, i.e. a fraction of its risk-weighted assets (RWA). The contribution of a loan to a banks' $\text{RWA}_{b,t}$ depends on the idiosyncratic probability of default of the borrower. Thus, the RWA are an increasing function of the borrower's D/E-ratio, i.e.

$$PD_{j,t} = 1 - \exp\{-\rho_j \xi_{j,t}\} \quad \text{with } j \in \{f,b\},\ \rho_j \in \{0.1,\ 0.35\} \tag{5.14}$$

for claims against firms ($j = f$) and banks ($j = b$), respectively. The qualitative differences concerning the business models of firms and banks, lead to the fact that the latter can have a much higher D/E-ratio for the same risk weight compared to firms. Positive risk weights are assigned to assets resulting from loan contracts whereas government bonds have a zero-risk weight.

[28] We do not explicitly modeled Basel III's liquidity requirements (LCR and NSFR), since the literature identifies the capital regulation as the most effective. For further analysis on the relationship between banks' liquidity regulation and monetary policy, see e.g. Scheubel and Körding (2013). For an overview on the effort to implement macroprudential policy in the EU see Gualandri and Noera (2015).

[29]

$$CCycB_{t+1} = [(\Lambda_t - \Lambda_t^n) - N] \cdot \frac{2.5}{M - N}$$

with the credit-to-GDP ratio

$$\Lambda_t = \frac{C_t}{GDP_t}.$$

In line with the regulatory proposal of the Bank of International Settlement (BIS), we set $N = 2$ and $M = 10$.

5.3.9 Monetary Policy

Since we have described how the CB uses the target rate as key instrument to transmit monetary policy in the model (subsection 5.3.3), we finally have to explain how decisions about its current level are made. The CB follows a standard Taylor Rule under flexible inflation targeting in order to ensure price and output stability:

$$i_t^* = i^r + \pi^* + \delta_\pi(\pi_t - \pi^*) + \delta_x\,(x_t - x_t^n) \tag{5.15}$$

with $i^r = \pi^* = 0.02$ and x_t^n representing the long-term trend of real GDP measured by application of the Hodrick-Prescott-filter (with $\lambda = 1600/4^4 = 6.25$ for yearly data [Ravn and Uhlig (2002)]).

The scheme's inherent interest incentive for banks combined with being in full control of the target rate and, thus, of the prevailing interest corridor, enables the CB to perfectly steer interest rates, indebtedness of the real sector and, hence, economic activity.

5.4 Design of Experiments (DOE)

The technical implementation of the experiments can be outlined as follows. In order to shed light on the question if and how shadow banking activity should be restricted by financial regulation, the performance of various cases (scenarios) is evaluated in counterfactual simulations of the underlying agent-based (disequilibrium) macroeconomic model.[30] Therefore, we conduct Monte Carlo simulations for random seeds $1, \dots, 1000$ while every run has a duration of $T = 3000$ periods and the chosen set up consists of 125 HH, 25 firms, 5 banks as well as 5 MMFs and Broker-dealers. According to our setting,[31] this duration can be translated into approx. 60 years. Hence, for the analysis, we take the last 50 years (2400 periods) into account and use the first 600 periods as initialization phase.

Within the previously explained model framework, we analyze the different outcomes of six scenarios which aim to represent the economy's development concerning the balancing of financialization and appropriate regulation. Hence, these scenarios are modeled in such a way that they represent states of the economy ranging from past ones (no shadow banking activity) over current ones (unregulated shadow banking sector) to some possible future states in which shadow banks also have to comply with regulatory requirements. In the following, we describe the scenarios in more detail:

[30] The extended ACE model is programmed in Scala 2.11.8 and the code is available upon request to *s.krug@economics.uni-kiel.de*.

[31] Within our model, every tick represents a week and every month has 4 weeks which adds up to 48 weeks for an experimental year. Compare also chapter 4.4.

Case A This scenario represents the baseline or benchmark case in which an entirely institution-based credit system prevails, i.e. only traditional and regulated (universal) banks exist. This means that there is no shadow banking activity at all and the real sector is credit rationed when the conditions offered by traditional banks as main source of liquidity lies outside the acceptable range of the requesting agent. Traditional banks have to comply with the Basel III accord and, thus, might not be able to offer suitable conditions due to their current balance sheet structure. A detailed description of the model's baseline version including a section on its validation can be found in Krug (2015).

Case B In a first extending step, shadow bank activity is introduced to the baseline scenario as we have it these days, meaning that traditional banks are still regulated while shadow banks are not. This step mimics the recent development towards a market-based credit intermediation system. Here, shadow banks serve as alternative and attractive source of liquidity. As a consequence, they can exploit their advantageous business environment to compete with traditional banks on the credit market and eventually crowd them out to a significant extend. The superior flexibility in terms of their balance sheet structure and their ability to provide low cost credit to the real sector let them gain market share but is also accompanied by increased systemic risk. This scenario can be seen as a good approximation of the current situation.

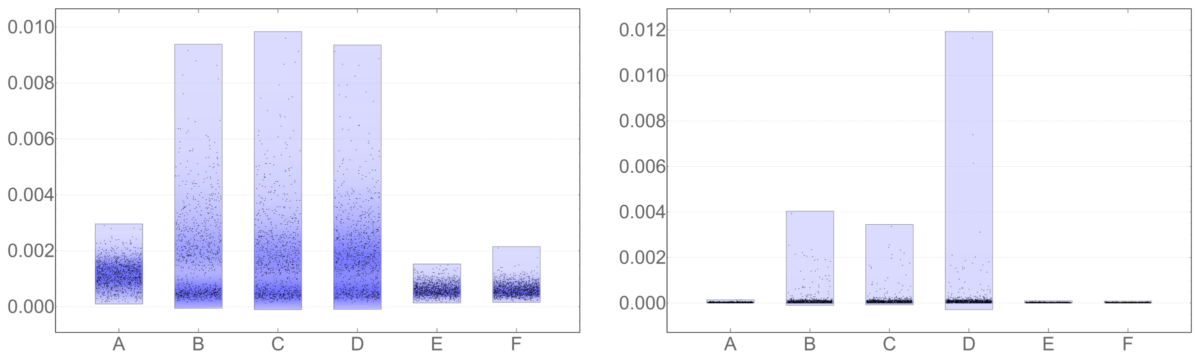
Case C An inherent part of the current debate about financial regulation relates to a fundamental reform of the way the requirements apply. The invocation to replace the current approach of a *"regulation by institutional form"* with a *"regulation by function"* moves more and more into the spotlight [Pozsar et al. (2010); Blinder (2010); Vento and Ganga (2013)]. Within our experimental lab, this means to make the transition from a regulatory framework that is only applicable to banks (from a legal point of view, shadow banks are not banks) and to proceed with one that regulates financial institutions by their functions, i.e. whether their business model includes credit/liquidity/maturity transformation or not. Thus, in case C, we start experimenting with the regulation of the shadow banking sector by burdening the so far unregulated part of the financial system to likewise comply with the Basel III accord in order to test whether a restriction of extremely leveraged entities would be sufficient to stabilize the economy to the desired extend. This means that, in this case, shadow banks are *equally* regulated compared to traditional banks which reduces the competitive advantage of shadow banks substantially. Moreover, in this scenario only traditional banks have access to central bank liquidity, i.e. there is no lender of last resort for shadow banks.

Case D Case D goes one step further by regulating the shadow banking sector even *stricter* than traditional banks. Here, we just tighten the requirements of the Basel III accord, i.e. the *capital adequacy ratio* for shadow banks is now 10% while it remains at 4.5% for traditional banks. The complementary risk-based requirement of *surcharges for systemically important financial institutions* (SIFI) is doubled leaving the process of assigning the institutions into the buckets stays untouched. An equivalent change is implemented for the non-risk sensitive *leverage ratio* which rises from 3% to 10% for shadow banks. Moreover, there is still no access to central bank liquidity for shadow banks.

Case E Mehrling (2012) (among others) questions the sufficiency of the public safety net's liquidity backstop because it is exclusively accessible for traditional banks. This criticism cause us to additionally analyze cases in which the now regulated shadow banking sector not only faces the downside of financial regulation but also has access to a lender of last resort. In order to isolate the effect on the stability of the system, case E is equivalent to case C except for the this detail. Hence, both traditional and shadow banks are equally regulated and, this time, solvent but illiquid institutions of both sectors have access to central bank liquidity.

Case F Case F is the corresponding equivalent to Case D, i.e. with the described tighter regulation of shadow banks but now with additional access to central bank liquidity.

In order to visualize the outcomes of the six scenarios as plain and disaggregated as possible, we use plots that show every single data point within a bin. This proceeding should enable the reader to get a proper intuition of the distribution of the simulated data. For instance, figure 5.9a shows the simulation results for the variance of the inflation rate and each of the six bins contains the corresponding 1000 realizations of Var(π) under the conditions described for the cases above. Every realization is represented by a small black dot and the bins show a blue background that gets darker in areas where realizations are more concentrated. The height of the bins represents the range of realizations. Finally, the ordinate always represents the values of the corresponding variable under consideration.



(a) Distributions of Var(π) (b) Distributions of Var(x)

Figure 5.9: Results for central bank's dual mandate

5.5 Discussion of Results

5.5.1 Macroeconomic Stability

We start the presentation of the simulation results[32] with a closer look at the standard parts of a central bank's loss function operating within a flexible inflation targeting regime, i.e. the variances of inflation rate π and output x. Table 5.2 shows the results for the different experiments and we see that the system without shadow banking activity (case A) endows the monetary policy makers with much more control to steer the economy onto a rather calm trajectory. When the economy passes through the transition towards a mainly market-based credit system

Table 5.2: Macroeconomic stability

Case	Var(π)	Var(x)
A	0.00116132 (100.00%)	0.0000231731 (100.00%)
B	0.00183051 (157.63%)	0.0001404550 (606.11%)
C	0.00178202 (153.45%)	0.0001050580 (453.36%)
D	0.00189498 (163.18%)	0.0001355790 (585.07%)
E	0.00063002 (54.25%)	0.0000156398 (67.49%)
F	0.00062860 (54.13%)	0.0000157170 (67.83%)

by introducing (unregulated) shadow banks, this changes dramatically and volatilities rise significantly. Such a parallel banking system, i.e. completely beyond the reach of regulators, seems to negatively affect the central bank's ability to achieve their policy goals as the occurrence of the recent global financial crises has harmfully shown. If the activity of this disrupting element would be restricted by incorporating shadow banks into the regulatory framework, this does not change much (case C) and the variance of inflation and output decline just slightly. Constraining the lending activity of shadow banks over-proportionally and trying to enhance the competitiveness of traditional banks through massive regulation, in turn, worsens the situation from a central bank's point of view. Note that until now, the incorporation of shadow banks into the regulatory framework is incomplete since they are burdened with financial regulation but still haven't access to a lender of last resort. This brings us to the results for case E and F, which suggest that the volatilities seem to be driven by the absence of the liquidity insurance of the central bank. The huge liquidity risk underlying the shadow banks' fragile funding model can be eliminated to a large extend if they would have also access to public safety net in return for their regulatory burden. Figure 5.9a and 5.9b show the distributions of the variances of inflation and of the output gap, respectively, in detail.

[32]Our results are robust in the sense that they do not alter qualitatively under different setups of the experiments. We conducted the same simulations either with significantly more agents following Riccetti et al. (2014) (i.e. 500 households, 80 firms and 10 banks), and we also varied the size of the shadow banking sector relative to the traditional banking sector. Concerning the latter experiments, we simulated both a much smaller (larger) shadow banking sector being half (twice) as large as the traditional one.

5.5.2 Economic Growth

The most fundamental dimension of interest concerning the impact of varying degrees of financialization is, of course, economic growth. Table 5.3 shows the average annual growth rates in both nominal and real terms. Although, on a bird's eye view, one would think that the different scenarios only have minor effects on growth, the reader should note that these are average growth rates per year over a time span of 50 years. So even rather small deviations from the benchmark case A mean significant deviations in the growth-path over the whole simulated period of time.

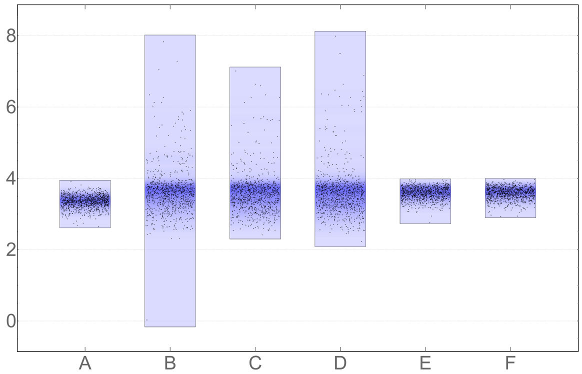
Table 5.3: Average annual growth rates (nominal/real)

Case	Avg. nominal growth (% p.a.)	Avg. real growth (% p.a.)
A	3.35398 (100.00%)	1.25396 (100.00%)
B	3.60575 (107.51%)	1.28218 (102.25%)
C	3.56649 (106.34%)	1.26385 (100.79%)
D	3.58598 (106.92%)	1.29978 (103.65%)
E	3.58371 (106.85%)	1.09079 (86.99%)
F	3.58683 (106.94%)	1.09223 (87.10%)

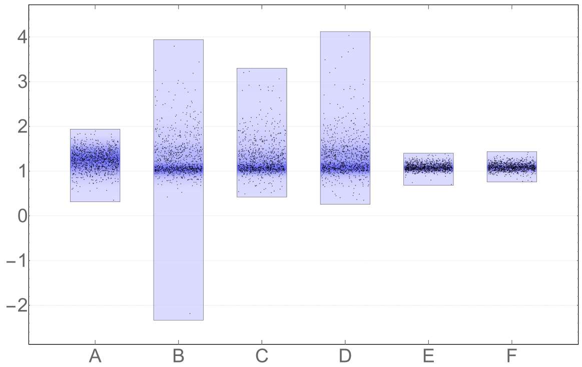
In nominal terms, the presence of alternative sources of liquidity seems to have (at least on average) an overall positive impact on growth, independent from the regulatory dimension. This is different for average real growth rates, since they drop when shadow banks have access to a lender of last resort while they show a moderate increase without. As we show in figure 5.10b, this phenomenon mainly stems from the fact that the volatility of real annual growth rates declines substantially in systems in which all institutions involved in the financial intermediation process are both subject to financial regulation (limiting systemic risk through the reduction of insolvency risk) and have a liquidity backstop (limiting the liquidity risk). Whereas leaving parts of the financial system completely unregulated (case B) can lead to strongly negative and harmful average growth rates. Despite the rarity of these events, policy makers definitely would choose to avoid such states in advance if they would be able to do so. Thus, our results show that the mitigation of systemic risk in as much dimensions as possible is directly linked to the *most stable*, although not growth-maximizing, trajectories of real growth, i.e. to preferred states from a central bank's point of view. This highlights the common trade-off between the primal (stability) goals of the central bank and the maximization of economic growth which can be typically found in this regard.

5.5.3 Financial Sector Stability

As we know from the recent past, a resilient financial system can be seen as a prerequisite for the achievement of primary monetary policy goals [Blanchard et al. (2010, 2013); Schularick



(a) Dist. of avg. annual growth rates (nominal)



(b) Dist. of avg. annual growth rates (real)

Figure 5.10: Distributions of mean annual growth rates

and Taylor (2012)]. Hence, it might be worthwhile to have a closer look at the development of some financial stability-related variables to get a better idea of what drives the results of section 5.5.1. Table 5.4 shows the default rates of financial sector agents across the experiments.

Table 5.4: Average default rates of financial sector agents

Case	trad. Bank	# bail outs	MMF	Broker-dealer	fiscal costs (in mio.)
A	63.8990 (100.00%)	26.1160 (100.00%)	–	–	326.442 (100.00%)
B	77.7692 (121.71%)	21.9990 (84.24%)	2.43623 (100.00%)	62.7257 (100.00%)	310.154 (95.01%)
C	75.4374 (118.06%)	21.9550 (84.07%)	3.51351 (144.22%)	13.9319 (22.21%)	308.129 (94.39%)
D	76.4724 (119.68%)	22.8372 (87.45%)	3.88844 (159.61%)	14.7930 (21.99%)	335.170 (102.67%)
E	81.5373 (127.60%)	18.6139 (71.27%)	1.09353 (44.89%)	0.0000 (0.00%)	118.879 (36.42%)
F	82.3736 (128.91%)	18.0819 (69.24%)	1.08691 (44.61%)	0.0000 (0.00%)	117.688 (36.05%)

The data on defaults of traditional banks reflects the increased competitiveness on the credit market due to the presence of shadow banks since more banks fail and even the expansion of the regulatory framework does not lead to a reversing effect. But one also has to incorporate the number of government bail outs through the course of the simulations which show an opposite development. Considering both variables, the data suggests that traditional banks do not fail more often but they lose in market share which makes them less systemically important and the government less often decides to jump in and to bail out the institution in distress.[33] Instead, it lets the bank fail and resolves it. Thus, although traditional banks are not regulated differently across the experiments, the regulation of shadow banks and the accompanied loss in market share due to the increased competitiveness on financial markets might lead to a mitigation of the moral hazard problem related to the "too-big-to-fail"-state of financial institutions. Moreover, our results show clearly that in the case of a regulation of shadow banks, in whatever form, the

[33]We do not implement the opportunity to bail out shadow banks, although the recent past has shown that this is, indeed, a quite realistic scenario. The reason is that the bail out of AIG was necessary because it was directly linked to the banking system meaning that its default would indirectly affect the payment system by bringing traditional banks in financial distress. In our model, this direct link is not present and without it, the default of a shadow bank affects economic activity but not the functioning of the payment system.

supervisory authorities have to take into account possible externalities on the already regulated part of the financial system although the regulation imposed on it does not change. Finally, the fiscal costs arising from government bail outs of banks decline tremendously when shadow banks are linked to the public safety net.

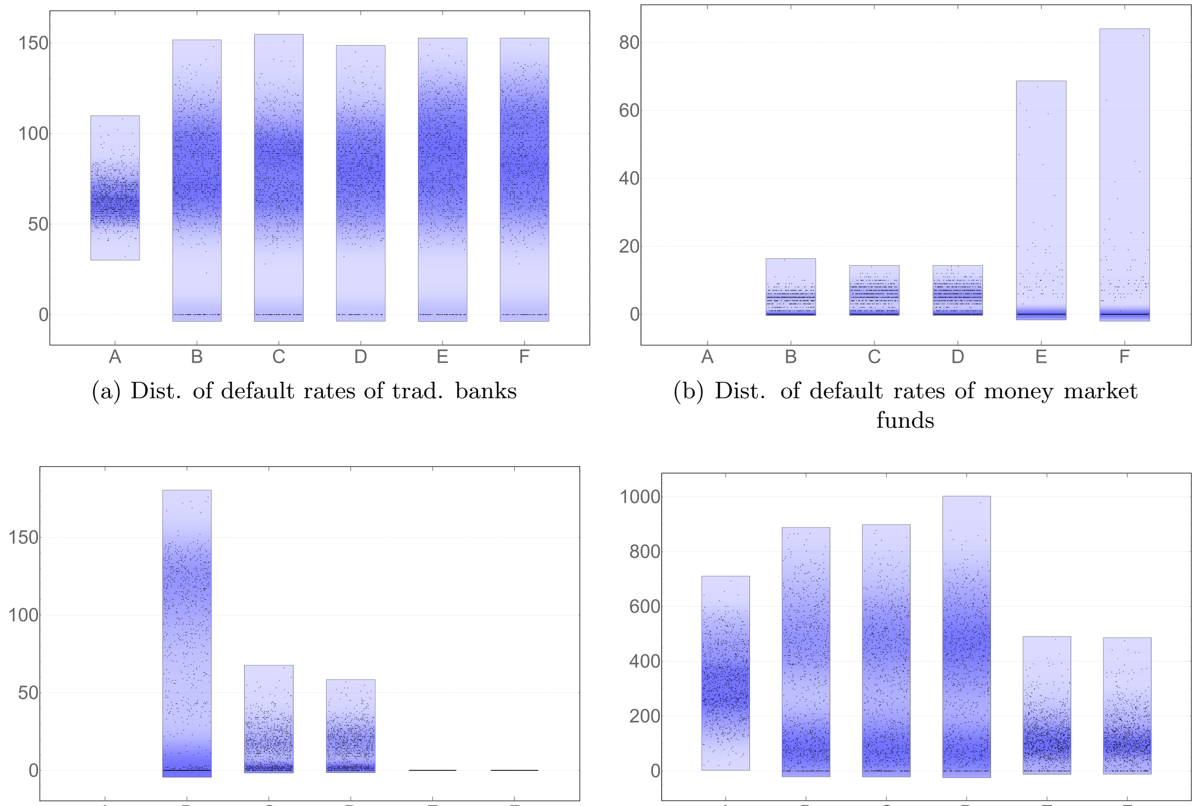


Figure 5.11: Distributions of financial sector agent default rates and fiscal costs

In addition to table 5.4, figure 5.11c emphasizes the relevance of restricting the balance sheet structure and the leverage of shadow banks by regulation. The average default rates, especially of Broker-dealer, decrease strongly and even drop to some tail events if liquidity and overall market risk is reduced by the central banks' liquidity insurance. For MMF, the effect is different, since their business model is indirectly affected by the restriction of the Broker-dealer's lending flexibility and they sometimes get in trouble due to the lack of investment opportunities and profit (see figure 5.11b).[34]

To underpin the results of this section, we also have a look at the volatility in the credit-to-GDP gap $(\Lambda_t - \Lambda_t^n)$ serving as a common early warning indicator for excessive and unsustainable credit growth and, thus, for financial crises [Drehmann and Tsatsaronis (2014); Giese et al.

[34]This is comparable with the current low or negative interest environment which has a similar effect on institutions with a business model based on returns on safe assets. For instance, home loans banks have serious problems to pay the contractually defined interest on deposits due to the lack of investment opportunities which yield a sufficiently safe and high return.

Table 5.5: Average variance in credit-to-GDP gap across cases

Case	Var(credit-to-GDP gap)
A	0.0241032 (100.00%)
B	0.1283510 (532.51%)
C	0.1054290 (437.41%)
D	0.0816890 (338.91%)
E	0.0178757 (74.16%)
F	0.0179898 (74.64%)

(2014)]. Table 5.5 shows that the variance in this indicator explodes due to the existence of an unregulated sources of liquidity (case B) and that it can be mitigated to some extend via regulatory requirements but still remains very high relative to the benchmark case (case C and D). The remarkable decline for the cases with a full inclusion of shadow banking activity into the regulatory framework can be explained by much more stable average growth paths (see figure 5.10b).

5.5.4 The Credit Market

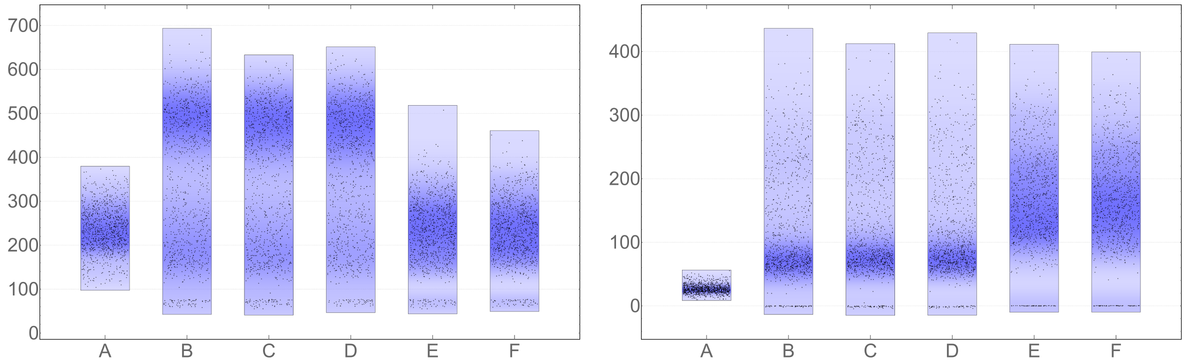
Our findings concerning the credit market meet the expectations of the literature in the sense that it clearly shows that shadow banking activity is not a bad thing per se [Dombret (2013a,b, 2014a)] but, by analogy with traditional banking of the 19th century [Adrian and Ashcraft (2012a)], it leads to negative externalities and, hence, has to be supervised properly [Pozsar (2014); Meeks et al. (2014); Pozsar et al. (2010)]. Table 5.6 reveals that the demand for liquidity could better be met with shadow banking activity and the indebtedness of the real sector rises accordingly. Unfortunately, the average default rate of firms (figure 5.12a) also increases due to the lack of proper regulation of private money creation. The free lending to the real sector including its financial unsound part, i.e to speculative and Ponzi financed firms in Minskyan terms,[35] leads to a widened set of possible growth paths (see figure 5.10a and 5.10b) and burdening shadow banks with regulatory requirements has a stabilizing effect in this regard by decreasing the average overall indebtedness of the real sector (case D). The most interesting results here are definitely delivered by the cases with full inclusion of shadow banking into the regulatory framework (case E and F). In these cases the default rate of firms declines to the level of an economy without shadow banking activity although much more liquidity is provided and the indebtedness of the real sector exceeds the debt of the benchmark case by far (figure 5.12b). These credit market data manifest in tremendously stable growth paths which suggests that a full inclusion of the shadow banking sector into the regulatory framework could indeed,

[35] Note, that the existence of broker dealers by itself also affects the prevailing shares of Minskyan financing schemes in the economy towards speculative ones, since it might be *solvent* enough to buy back the underlying collateral of a repo but usually not *liquid* enough and, hence, likewise contributing to systemic risk through two separate channels, i.e. its own highly leveraged and fragile balance sheet structure and the build up of financial sector imbalances as a result of its lending activity.

from a theoretical point of view, lead to a significant mitigation of the negative externalities accompanied by their fragile funding model and to a suitable exploitation of their liquidity provision capacity in terms of sustainable growth.

Table 5.6: Credit market data

Case	Avg. firm default rate	Avg. Firm Sector Demand for Credit (in mio.)	Avg. Firm Sector Debt (in mio.)
A	235.066 (100.00%)	494.582 (100.00%)	27.0044 (100.00%)
B	350.224 (148.99%)	123.074 (24.88%)	119.0050 (440.69%)
C	347.334 (147.76%)	124.342 (25.14%)	119.6300 (443.00%)
D	364.421 (155.03%)	128.818 (26.05%)	114.4270 (423.74%)
E	231.348 (98.42%)	103.119 (20.85%)	165.5840 (613.17%)
F	230.666 (98.13%)	102.681 (20.76%)	165.6510 (613.42%)



(a) Dist. of default rates of firms

(b) Dist of avg. firm sector indebtedness

Figure 5.12: Distribution of credit market related data

To summarize the results, we adopt the approach of Krug (2015) by using a combination of two loss functions to be able to compare the performance across cases. Hence, we define two loss functions concerning *(macro)economic* (L_k^{MS}) and *financial stability* (L_k^{FS}) in order to easily evaluate outcomes in both dimensions whereby the former is usually defined as the weighted sum of the variances of inflation, output gap and of nominal interest rate changes, i.e.

$$L_k^{MS} = \alpha_\pi \overline{\mathrm{Var}(\pi_k)} + \alpha_x \overline{\mathrm{Var}(x_k)} + \alpha_i \overline{\mathrm{Var}(i_k)} \tag{5.16}$$

with $\alpha_\pi = 1.0$, $\alpha_x = 0.5$, $\alpha_i = 0.1$ [Agénor et al. (2013); Agénor and Pereira da Silva (2012)]. The latter, however, addressing financial stability is defined in terms of the weighted sum of the average burden for the public sector of a bank bailout measured as the fraction of the average bailout costs for the government and the average amount of bailouts, as well as the average amount of bank and firm defaults ($\overline{\zeta_k}$, $\overline{\rho_k}$ and $\overline{\gamma_k}$, respectively), i.e.

$$L_k^{FS} = \alpha^{FS} \left(\overline{\zeta_k} + \overline{\rho_k} + \overline{\gamma_k} \right) \tag{5.17}$$

with $\alpha^{FS} = 0.01$ and $k \in \{A,B,C,D,E,F\}$. The combined loss L is expressed as

$$L = \alpha_L L_k^{MS} + (1 - \alpha_L) L_k^{FS}. \tag{5.18}$$

Table 5.7 shows the corresponding losses for each of the considered cases. The results make clear that when taking macroeconomic and financial stability issues into account (with $\alpha_L = 0.5$), the effort to fully include shadow banking activity into the regulatory framework seems to be worthwhile since the loss is much less even when compared to a situation in which traditional banking dominates. In contrast, a pure restriction of alternative activities in the financial sector leads to the highest losses across all scenarios.

Table 5.7: Combined losses for equally weighted objectives

Case	A	B	C	D	E	F
L	3.25823	3.80171	3.76522	3.99551	2.25223	2.24439

5.6 Concluding Remarks

The aim of this paper is to shed some light on the transition the credit system has been through over the last decades and on the destabilizing externalities accompanied by this, in particular, the substantial shift in market risks faced by financial institutions that is now much more in the focus of regulators. Aggravating this situation, the permanent seek of market participants for regulatory arbitrage has led to the continuous build up of a parallel and unregulated banking system *"in the shadows"*, i.e. beyond the reach of regulators, which roughly equals the traditional banking system in size.[36] Unfortunately, shadow banking does not only reduce the costs of the financial intermediation process but exhibits an extensive contribution to systemic risk due to

- the lack of regulation,
- the lack of access to a public safety net (liquidity and roll over risk) as well as
- the reliance on extreme short-term funding sources (through the money market).

Our contribution is to get some insights into the effects of an inclusion of the shadow banking sector into the current regulatory framework on economic activity and whether such a proceeding would be suitable to internalize the described destabilizing externalities.

As a framework for the analysis, we present an agent-based macro-model with heterogeneous interacting agents and endogenous money. The central bank agent plays a particular role since it controls market interest rates via monetary policy decisions which, in turn, affect credit demand and overall economic activity. Moreover, the model is augmented by a shadow banking sector representing an alternative investment opportunity for the real sector which is characterized by animal spirit-like, i.e. highly pro-cyclical and myopic, behavior in its investment decision. Therefore, we think that the presented model is well suited to analyze the research question at hand since pro-cyclical behavior as well as sudden and common withdrawals of invested funds has been identified as one of the root causes of systemic failures of the past.

Our simulation experiments provide three main findings. First, our results suggest that switching the regulatory regime from *"regulation by institutional form"* to a *"regulation by function"* meaning the inclusion of shadow banks into the regulatory framework, as proposed by Mehrling (2012), seems to be worthwhile in general terms.

Second, supervisory authorities should do so in a coordinated and complete manner. A unilateral inclusion, i.e. burdening the shadow banking sector with the same regulatory requirements as traditional banks but denying the access to the public safety net leads to inferior outcomes

[36] This is true for the US whereas the shadow banking sector accounts for approximately two-thirds of the traditional bank assets in Europe [Financial Stability Board (2014)].

compared to the benchmark case without shadow banking activity and even to the case in which they are not regulated at all. The results of such cases include negative effects on monetary policy goals, significantly increases in the volatility of growth and financial and real sector default rates as well as a higher volatility in the credit-to-GDP gap.

Moreover, experiments with a full and complete inclusion, i.e. with access to a lender of last resort, lead to superior outcomes in terms of the central bank's dual mandate, economic growth and financial stability suggesting that a full inclusion of the shadow banking sector into the regulatory framework could indeed, from a theoretical point of view, lead to a significant mitigation of the destabilizing externalities accompanied by their fragile funding model and to a suitable exploitation of their liquidity provision capacity in terms of sustainable growth.

Finally, the present paper is useful to understand why the access to central bank liquidity is so important: the main issue here is the extremely short-term funding maturity (typically overnight). The cash pools (MMF) have a huge incentive to minimize their own liquidity risk and to avoid runs by investors since they have promised the on-demand availability of the invested funds but this promise is not appropriately backed by a sufficient amount of capital which, in turn, creates massive roll-over risk for the broker-dealers. In addition, MMFs collectively tend to underestimate the associated risks with the repos they undertake since these are typically secured transactions signaling an alleged lack of risk due to the negligence of interconnectedness and interaction effects of operating on the same markets. This means, that in the case of a broker-dealer default resulting from a refusal to roll over the repo for another night, the MMF systemically neglects the fact that it will be forced to fire sale the collateral in order to serve the withdrawals from its investors. In such a situation, MMFs can only turn to financial markets since they control huge deposit volumes and have no link to a lender of last resort. The associated discount puts additional pressure on the badly capitalized funds triggering even more harmful collective actions. These features of financial crises originating in the shadow banking sector are fully covered by the presented version of our model and our results clearly show the negative effects on economic activity of a lack of contagion-free, alternative sources of liquidity within the shadow banking sector as it is nowadays.

These negative effects can be seen as a typical result of a coordination failure. Socially, it would be better if agents would avoid the negative externalities of their sudden collective withdrawals by appropriate coordination and the distribution of possible (collective) losses across *all* agents. Instead, their behavior is guided by selfishness and the attempt to maximize their individual utility by strictly acting to minimize *individual* losses. This reveals the need for an intervention of a superordinate institution like a financial supervisory authority to internalize negative effects exogenously and to prevent socially undesired states of the system, i.e. financial crises.

For future research, an extension towards the direct link between traditional and shadow banks would incorporate another highly relevant issue with regard to financial stability. In such a scenario, public sector bail outs of systemically important shadow banks would be of much interest. Furthermore, one could also test the performance of other macroprudential tools since the Basel III accord does only include a selection of the available tools which are related to financial institutions. Here, the impact of a loan-to-value ratio (LTV) or a debt-to-income ratio (DTI) applied on household credit could be interesting and it would similarly enable the researcher to extend the analysis towards the financial cycle. Finally, an extension of the model towards an open economy could also be an interesting task and would widen the range of research questions which can be addressed and analyzed using the underlying agent-based framework significantly.

Acknowledgements: I would like to thank Matthias Lengnick, Sander van der Hoog, Benjamin Burghardt, Falko Schlaffke, Sven Offick and Stephen Sacht for a lot of stimulating comments.

References of Chapter 5

Adrian, T. and Ashcraft, A. (2012a). Shadow Banking: A Review of the Literature, *Staff Reports no. 580*, Federal Reserve Bank of New York.

Adrian, T. and Ashcraft, A. (2012b). Shadow Banking Regulation, *Annu. Rev. Financ. Econ.* **4**(1): 99–140.

Adrian, T. and Boyarchenko, N. (2012). Intermediary Leverage Cycles and Financial Stability, *Federal Reserve Bank of New York Staff Reports no. 567* .

Adrian, T. and Shin, H. S. (2008). Liquidity, Monetary Policy, and Financial Cycles, *Current Issues in Economics and Finance* **14**(1): 1–7.

Adrian, T. and Shin, H. S. (2009). The Shadow Banking System: Implications for Financial Regulation, *Federal Reserve Bank of New York Staff Report No. 382* .

Afonso, G. and Lagos, R. (2013). The Over-the-Counter Theory of the Fed Funds Market: A Primer, *Staff Report no. 660*, Federal Reserve Bank of New York.
URL: *http://newyorkfed.org/research/staff_reports/sr660.html*

Agénor, P.-R., Alper, K. and Pereira da Silva, L. A. (2013). Capital Regulation, Monetary Policy, and Financial Stability, *International Journal of Central Banking* **9**(3): 193–238.

Agénor, P.-R. and Pereira da Silva, L. A. (2012). Macroeconomic Stability, Financial Stability, and Monetary Policy Rules, *International Finance* **15**(2): 205–224.

Akerlof, G. A. and Shiller, R. J. (2009). *Animal Spirits: How Human Psychology Drives the Economy, and Why It Matters for Global Capitalism*, Princeton University Press, New Jersey.

Arciero, L., Biancotti, C., D'Aurizio, L. and Impenna, C. (2009). Exploring Agent-based Methods for the Analysis of Payment Systems: A Crisis Model for StarLogo TNG, *Journal of Artificial Societies and Social Simulation* **12**(1): 2.

Arnold, B., Borio, C., Ellis, L. and Moshirian, F. (2012). Systemic Risk, Macroprudential Policy Frameworks, Monitoring Financial Systems and the Evolution of Capital Adequacy, *Journal of Banking & Finance* **36**(12): 3125 – 3132.

Aymanns, C. and Farmer, J. D. (2015). The Dynamics of the Leverage Cycle, *Journal of Economic Dynamics and Control* **50**: 155 – 179.

Bank of England (2014a). Bank of England Settlement Accounts, *Technical report*, Bank of England.
URL: *http://www.bankofengland.co.uk/markets/Documents/paymentsystems/boesettlementaccounts.pdf*

Bank of England (2014b). The Bank of England's Sterling Monetary Framework (The Red Book), *Technical report*, Bank of England.
URL: *http://www.bankofengland.co.uk/markets/Pages/sterlingoperations/redbook.aspx*

Basel Committee on Banking Supervision (BCBS) (2010). Guidance for National Authorities Operating the Countercyclical Capital Buffer, *Technical report*, Bank for International Settlements (BIS).
URL: *http://www.bis.org/publ/bcbs187.htm*

Battiston, S., Farmer, J. D., Flache, A., Garlaschelli, D., Haldane, A. G., Heesterbeek, H., Hommes, C., Jaeger, C., May, R. and Scheffer, M. (2016). Complexity Theory and Financial Regulation, *Science* **351**(6275): 818–819.

Bernanke, B. S., Gertler, M. and Gilchrist, S. (1999). The Financial Accelerator in a Quantitative Business Cycle Framework, *in* J. B. Taylor and M. Woodford (eds), *Handbook of Macroeconomics*, Vol. 1, Part C of *Handbook of Macroeconomics*, Elsevier, chapter 21, pp. 1341 – 1393.

Blanchard, O. J., Dell'Ariccia, G. and Mauro, P. (2010). Rethinking Macroeconomic Policy, *Journal of Money, Credit and Banking* **42**(s1): 199–215.

Blanchard, O. J., Dell'Ariccia, G. and Mauro, P. (2013). Rethinking Macro Policy II: Getting Granular, *IMF Staff Discussion Note SDN/13/03* pp. 1–26.

Blinder, A. S. (2010). Its Broke, Lets Fix It: Rethinking Financial Regulation, *International Journal of Central Banking* **6**(34): 277–330.

Bookstaber, R. (2012). Using Agent-based Models for Analyzing Threats to Financial Stability, *U.S. Department of the Treasury, Office of Financial Research, Working Paper #0003* .

Borio, C. (2014). Monetary Policy and Financial Stability: What Role in Prevention and Recovery?, *BIS Working Papers no. 440*, Bank for International Settlements.

Borio, C. and Zhu, H. (2012). Capital Regulation, Risk-taking and Monetary Policy: A Missing Link in the Transmission Mechanism?, *Journal of Financial Stability* **8**(4): 236 – 251.

Chernenko, S. and Sunderam, A. (2014). Frictions in Shadow Banking: Evidence from the Lending Behavior of Money Market Mutual Funds, *Review of Financial Studies* **27**(6): 1717–1750.

Claessens, S., Kose, M. A. and Terrones, M. E. (2012). How Do Business and Financial Cycles Interact?, *Journal of International Economics* **87**(1): 178 – 190.

Dawid, H., Gemkow, S., Harting, P., van der Hoog, S. and Neugart, M. (2014). The Eurace@Unibi Model: An Agent-based Macroeconomic Model for Economic Policy Analysis, *University of Bielefeld Working Papers in Economics and Management* **2012**(5).

De Grauwe, P. (2011). Animal Spirits and Monetary Policy, *Economic Theory* **47**(2): 423–457.

Dent, A. and Dison, W. (2012). The Bank of England's Real-Time Gross Settlement Infrastructure, *Bank of England Quartely Bulletin* **2012**(Q3): 234–243.

di Iasio, G. and Pozsar, Z. (2015). A Model of Shadow Banking: Crises, Central Banks and Regulation, *SSRN Electronic Journal* .

Dombret, A. (2013a). Systemic Risks of Shadow Banking, Speech at the Salzburg Global Seminar "Out of the Shadows: Should Non-banking Financial Institutions Be Regulated?", Salzburg, 20 August 2013.

Dombret, A. (2013b). Systemic risks of shadow banking, Speech held at the Salzburg Global Seminar "Out of the Shadows: Should Non-banking Financial Institutions Be Regulated?", Salzburg, 20th of August, 2013.
URL: *http://www.bis.org/review/r130821a.pdf*

Dombret, A. (2014a). Financial Market Regulation – Standing Still Means Falling Behind, Speech held at the 2014 Alternative Investor Conference of the Federal Association for Alternative Investments, Frankfurt am Main, 14th of May, 2014.
URL: *https://www.bis.org/review/r140519d.htm*

Dombret, A. (2014b). Shadow Banking and the Roots of International Cooperation, Speech at a reception to bid farewell to Winfried Liedtke, financial attach, and to welcome his successor, Thomas Notheis, Beijing, 29th of October, 2014.
URL: *https://www.bundesbank.de/Redaktion/EN/Reden/2014/2014_10_28_dombret.html?nn=2104*

Drehmann, M., Borio, C., Gambacorta, L., Jiménez, G. and Trucharte, C. (2010). Countercyclical Capital Buffers: Exploring Options, *BIS Working Papers no. 317*, Bank for International Settlements.

Drehmann, M., Borio, C. and Tsatsaronis, K. (2012). Characterising the Financial Cycle: Don't Lose Sight of the Medium Term!, *BIS Working Papers no. 380*, Bank for International Settlements.

Drehmann, M. and Tsatsaronis, K. (2014). The Credit-to-GDP Gap and Countercyclical Capital Buffers: Questions and Answers, *BIS Quarterly Review* **2014**(March): 55–73.

ECB (2010). Survey on the Access to Finance of Small and Medium-sized Enterprises in the Euro Area: Second Half of 2009, *Technical report*, European Central Bank.
URL: *https://www.ecb.europa.eu/stats/money/surveys/sme/html/index.en.html*

Financial Stability Board (2014). Global Shadow Banking Monitoring Report 2014, *Technical report*, Financial Stability Board (FSB).

Financial Stability Board (2015). Global Shadow Banking Monitoring Report 2015, *Technical report*, Financial Stability Board (FSB).

Galati, G., Koopman, S. J. and Vlekke, M. (2016). Measuring Financial Cycles in a Model-based Analysis: Empirical Evidence for the United States and the Euro Area, *SSRN Electronic Journal* .
URL: *http://dx.doi.org/10.2139/ssrn.2768697*

Galbiati, M. and Soramäki, K. (2011). An Agent-based Model of Payment Systems, *Journal of Economic Dynamics and Control* **35**: 859–875.

Geanakoplos, J. (2009). The Leverage Cycle, *Cowles Foundation Discussion Papers 1715*, Cowles Foundation for Research in Economics, Yale University.
URL: *https://ideas.repec.org/p/cwl/cwldpp/1715.html*

Gennaioli, N., Shleifer, A. and Vishny, R. (2012). Neglected Risks, Financial Innovation, and Financial Fragility, *Journal of Financial Economics* **104**(3): 452 – 468. Market Institutions, Financial Market Risks and Financial Crisis.

Gennaioli, N., Shleifer, A. and Vishny, R. W. (2013). A Model of Shadow Banking, *The Journal of Finance* **68**(4): 1331–1363.

Gerali, A., Neri, S., Sessa, L. and Signoretti, F. M. (2010). Credit and Banking in a DSGE Model of the Euro Area, *Journal of Money, Credit and Banking* **42**: 107–141.

Gertler, M. and Karadi, P. (2011). A Model of Unconventional Monetary Policy, *Journal of Monetary Economics* **58**(1): 17 – 34.

Giese, J., Andersen, H., Bush, O., Castro, C., Farag, M. and Kapadia, S. (2014). The Credit-To-GDP Gap And Complementary Indicators For Macroprudential Policy: Evidence From The Uk, *International Journal of Finance & Economics* **19**(1): 25–47.

Goodhart, C. A., Kashyap, A., Tsomocos, D. and Vardoulakis, A. (2012). Financial Regulation in General Equilibrium, *NBER Working Paper Series, Working Paper no. 17909* .
URL: *http://dx.doi.org/10.3386/w17909*

Gorton, G. and Metrick, A. (2010). Regulating the Shadow Banking System, *Brookings Papers on Economic Activity* **2010**(2): 261–297.

Gorton, G. and Metrick, A. (2012a). Securitization, *NBER Working paper Series, Working Paper 18611* .

Gorton, G. and Metrick, A. (2012b). Securitized Banking and the Run on Repo, *Journal of Financial Economics* **104**(3): 425–451.

Gorton, G. and Metrick, A. (2013). The Federal Reserve and Panic Prevention: The Roles of Financial Regulation and Lender of Last Resort, *The Journal of Economic Perspectives* **27**(4): 45–64.

Gualandri, E. and Noera, M. (2015). *Bank Risk, Governance and Regulation*, Palgrave Macmillan UK, London, chapter Towards a Macroprudential Policy in the EU, pp. 182–205.

Haldane, A. G. and Qvigstad, J. F. (2014). The Evolution of Central Banks – a Practitioner's Perspective, *Norges Bank Conference "Of the uses of central banks: Lessons from history", 4-5 June 2014*.

Hoenig, T. M. (1996). Rethinking Financial Regulation, *Economic Review-Federal Reserve Bank of Kansas City* **81**: 5–14.

Kelsey, E. and Rickenbach, S. (2014). Enhancing the Resilience of the Bank of England's Real-Time Gross Settlement Infrastructure, *Bank of England Quartely Bulletin* **2014**(Q3): 316–320.

Keynes, J. M. (1936). *General Theory of Employment, Interest and Money*, Cambridge University Press, Cambridge.

Krug, S. (2015). The Interaction between Monetary and Macroprudential Policy: Should Central Banks "Lean Against the Wind" to Foster Macro-financial Stability?, *Economics Working Papers 2015-08*, Christian-Albrechts-University of Kiel, Department of Economics.
URL: *http://www.vwl.uni-kiel.de/EconomicsWorkingPapers/abstract.php?stat_id=192*

Krug, S., Lengnick, M. and Wohltmann, H.-W. (2015). The Impact of Basel III on Financial (In)stability: An Agent-based Credit Network Approach, *Quantitative Finance* **15**(12): 1917–1932.

Lavoie, M. (2003). A Primer on Endogenous Credit-Money, *in* L.-P. Rochon and S. Rossi (eds), *Modern Theories of Money: The Nature and Role of Money in Capitalist Economies*, Edward Elgar Publishing Limited, Northampton, pp. 506–543.

Lengnick, M. and Wohltmann, H.-W. (2016). Optimal Monetary Policy in a New Keynesian Model with Animal Spirits and Financial Markets, *Journal of Economic Dynamics and Control* **64**: 148 – 165.

Luttrell, D., Rosenblum, H. and Thies, J. (2012). Understanding the Risks Inherent in Shadow Banking: A Primer and Practical Lessons Learned, *Dallas FED Staff Papers* **No. 18**.

Martin, A., Skeie, D. and von Thadden, E.-L. (2013). Repo Runs, *GESY Discussion Paper No 448* .

McLeay, M., Radia, A. and Thomas, R. (2014a). Money Creation in the Modern Economy, *Bank of England Quartely Bulletin* **54**(1): 14–27.

McLeay, M., Radia, A. and Thomas, R. (2014b). Money in the Modern Economy: An Introduction, *Bank of England Quartely Bulletin* **54**(1): 4–13.

Meeks, R., Nelson, B. and Alessandri, P. (2014). Shadow banks and macroeconomic instability, *Bank of England working papers 487*, Bank of England.
URL: *https://ideas.repec.org/p/boe/boeewp/0487.html*

Meh, C. A. and Moran, K. (2010). The Role of Bank Capital in the Propagation of Shocks, *Journal of Economic Dynamics and Control* **34**(3): 555 – 576.

Mehrling, P. (2012). Three Principles for Market-based Credit Regulation, *American Economic Review* **102**(3): 107–112.

Mehrling, P., Pozsar, Z., Sweeney, J. and Neilson, D. H. (2013). Bagehot was a Shadow Banker: Shadow Banking, Central Banking, and the Future of Global Finance, *SSRN Electronic Journal* .

Minsky, H. P. (1986). *Stabilizing an Unstable Economy*, New Haven, Yale University Press.

Montagna, M. (2016). *Systemic Risk in Modern Financial Systems*, PhD thesis, Kiel University, Kiel.

Nakajima, M. (2011). *Payment System Technologies and Functions: Innovations and Developments*, IGI Global, Hershey.

Plantin, G. (2014). Shadow Banking and Bank Capital Regulation, *The Review of Financial Studies* **0**(0): 1–30.

Pozsar, Z. (2014). Shadow Banking: The Money View, *Technical report*.

Pozsar, Z. (2015). A Macro View of Shadow Banking: Levered Betas and Wholesale Funding in the Context of Secular Stagnation, *SSRN Electronic Journal* .
URL: *http://dx.doi.org/10.2139/ssrn.2558945*

Pozsar, Z., Adrian, T., Ashcraft, A. and Boesky, H. (2010). Shadow Banking, *Available at SSRN 1640545* .

Ravn, M. O. and Uhlig, H. (2002). On Adjusting the Hodrick-Prescott Filter for the Frequency of Observations, *Review of Economics and Statistics* **84**(2): 371–376.

Riccetti, L., Russo, A. and Gallegati, M. (2014). An Agent-based Decentralized Matching Macroeconomic Model, *Journal of Economic Interaction and Coordination* **forthcoming**: 1–28.
URL: *http://link.springer.com/article/10.1007%2Fs11403-014-0130-8#page-1*

Ricks, M. (2010). Shadow Banking and Financial Regulation, *Columbia Law and Economics Working Paper* (370).

Rosengren, E. S. (2014). Broker-dealer Finance and Financial Stability. Remarks by Eric S. Rosengren, President and Chief Executive Officer, Federal Reserve Bank of Boston, at the Conference on the Risks of Wholesale Funding sponsored by the Federal Reserve Banks of Boston and New York, New York, New York, August 13, 2014.
URL: *http://EconPapers.repec.org/RePEc:fip:fedbsp:85*

Ryan-Collins, J., Greenham, T., Werner, R. and Jackson, A. (2012). *Where Does Money Come From? A Guide to the UK Monetary and Banking System*, 2nd edn, The New Economics Foundation, London.

Scheubel, B. and Körding, J. (2013). Liquidity Regulation, the Central Bank, and the Money Market, *Beitrage zur Jahrestagung des Vereins fur Socialpolitik Wettbewerbspolitik und Regulierung in einer globalen Wirtschaftsordnung - Session Liquidity and Regulation, No. E19-V3* .

Schularick, M. and Taylor, A. M. (2012). Credit Booms Gone Bust: Monetary Policy, Leverage Cycles, and Financial Crises, 1870-2008, *American Economic Review* **102**(2): 1029–61.
URL: *http://www.aeaweb.org/articles.php?doi=10.1257/aer.102.2.1029*

Schwarcz, S. L. (2012). Regulating Shadow Banking, *Review of Banking and Financial Law* **31**(1).

Schwarcz, S. L. (2013). Regulating Shadows: Financial Regulation and Responsibility Failure, *Washington and Lee Law Review* (forthcoming).

Solow, R. M. (1956). A Contribution to the Theory of Economic Growth, *The quarterly journal of economics* pp. 65–94.

Stolzenburg, U. (2015). The Agent-based Solow Growth Model with Endogenous Business Cycles, *Economics Working Papers 2015-01*, Christian-Albrechts-University of Kiel, Department of Economics.

Strohsal, T., Proano, C. R. and Wolters, J. (2015a). Characterizing the Financial Cycle - Evidence from a Frequency Domain Analysis, number 21 in *Sonderforschungsbereich 649: Ökonomisches Risiko*, Wirtschaftswissenschaftliche Fakultät.

Strohsal, T., Proano, C. R. and Wolters, J. (2015b). How Do Financial Cycles Interact? - Evidence from the US and the UK, number 24 in *Sonderforschungsbereich 649: Ökonomisches Risiko*, Wirtschaftswissenschaftliche Fakultät.

Sunderam, A. (2015). Money Creation and the Shadow Banking System, *Review of Financial Studies* **28**(4): 939–977.

van der Hoog, S. and Dawid, H. (2015). Bubbles, Crashes and the Financial Cycle, *University of Bielefeld Working Papers in Economics and Management* **2015**(1).

Vento, G. A. and Ganga, P. (2013). *Crisis, Risk and Stability in Financial Markets*, Palgrave Macmillan UK, London, chapter Shadow Banking and Systemic Risk: In Search of Regulatory Solutions, pp. 96–115.
URL: *http://dx.doi.org/10.1057/9781137001832_6*

Verona, F., Martins, M. M. F. and Drumond, I. (2013). (Un)anticipated Monetary Policy in a DSGE Model with a Shadow Banking System, *International Journal of Central Banking* **9**(3): 74 – 117.

Winters, B. (2012). Review of the Bank of England's Framework for Providing Liquidity to the Banking System, *Report for the court of the bank of england*, Bank of England.

Chapter 6

Outlook and Future Research

The effort put into future research projects should bifurcate. The first bifurcation is of methodological concerns, i.e. raising the acceptance and the validity of agent-based macro-models by improving the usability of the models in order to enable students to study macroeconomic phenomena on their own. The current form of most existing ACE models is far away from being “user-friendly” since it requires a huge amount of practical training and a level of programming and computer science skills that is far beyond that what the average student in economics exhibits. Unfortunately, this circumstance serves as an entrance barrier for lecturers because they usually do not have enough time to explain the whole model functionality in class and students are seldom willing to study a thick user manual (if it would exist). The goal should be apps that are runnable with a single click and that provide a graphical user interface (GUI) that can be used by students to change things like parameters, number of agents or their initial endowments. The model presented in chapter 2 and 3 is a first step in this direction by exhibiting all these features and vividly mimicking the text-book approach to money creation. At the same time it is developed in a framework that is quite easy to extend (NetLogo) by means of the provision of routines for the actions of agents. Moreover, in chapter 3 we demonstrate that, although simple, the model can be used to answer highly relevant policy questions.

This brings us to the second bifurcation, i.e. further extensions of the models presented during the course of this dissertation. The model extensions of chapter 3 (the implementation of the Basel III accord) and chapter 5 (the introduction of a shadow banking sector) show that the underlying baseline versions of the agent-based macro-models provide a broad foundation to further develop and adjust them according to the users current research question.

A further extension towards the direct link between traditional and shadow banks would incorporate another highly relevant issue with regard to financial stability. In such a scenario, public sector bail outs of systemically important shadow banks would be of much interest. Furthermore, one could also test the performance of other macroprudential tools since the Basel III accord does only include a selection of the available tools which are related to financial institutions. For instance, the impact of a loan-to-value ratio (LTV) or a debt-to-income ratio (DTI) applied on household credit could be interesting and it would similarly enable the researcher to extend the analysis towards the financial cycle. Such an extension would also suggest the introduction of a housing market while a stock market that makes the households' stakes in firms and banks tradable would complete the research that can be done on financial markets. Finally, an extension of the model towards an open economy could also be an interesting task and would widen the range of research questions that can be addressed and analyzed using the underlying agent-based framework significantly. Either way, the financial system's highly dynamic character, its cat-and-mouse game with financial regulation and the permanent seek for regulatory arbitrage of market participants will surely remain the driving forces for the emergence of new financial and macroeconomic phenomena that threaten the stability of future financial systems. In my view, the presented model frameworks can also be used to extend the analysis towards transmission channels of systemic risk that are yet unknown.

Curriculum Vitæ

Last Name:	Krug
First Name:	Sebastian
Address:	Wilhelm-Seelig-Platz 1
	24118 Kiel, Germany
E-Mail:	s.krug@economics.uni-kiel.de

Education

08/2010	Diploma in Business Administration ("Diplom-Kaufmann"), University of Kiel
01/2012	Participant of the PhD-Program "Quantitative Economics", University of Kiel
02/2012	Diploma in Economics ("Diplom-Volkswirt"), University of Kiel

Publications

- Krug, Sebastian & Lengnick, Matthias & Wohltmann, Hans-Werner, 2015. "The Impact of Basel III on Financial (In)stability: An Agent-based Credit Network Approach", *Quantitative Finance*, vol. 15(12), pp. 1917– 1932.
- Lengnick, Matthias & Krug, Sebastian & Wohltmann, Hans-Werner, 2013. "Money Creation and Financial Instability: An Agent-based Credit Network Approach", *Economics – The Open-Access, Open-Assessment E-Journal*, Kiel Institute for the World Economy, vol. 7, pages 1-44.

Working Paper

- Krug, Sebastian & Wohltmann, Hans-Werner, 2016. "Shadow Banking, Financial Regulation and Animal Spirits: An ACE Approach", Economics Working Papers 2016-08, Christian-Albrechts-University of Kiel, Department of Economics.
- Krug, Sebastian, 2015. "The Interaction between Monetary and Macroprudential Policy: Should Central Banks 'Lean Against the Wind' to Foster Macro-financial Stability?",

Economics Working Papers 2015-08, Christian-Albrechts-University of Kiel, Department of Economics.

- Krug, Sebastian & Lengnick, Matthias & Wohltmann, Hans-Werner, 2014. "The Impact of Basel III on Financial (In)stability: An Agent-based Credit Network Approach", Economics Working Papers 2014-13, Christian-Albrechts-University of Kiel, Department of Economics.

- Lengnick, Matthias & Krug, Sebastian & Wohltmann, Hans-Werner, 2012. "Money Creation and Financial Instability: An Agent-based Credit Network Approach", Economics Working Papers 2012-15, Christian-Albrechts-University of Kiel, Department of Economics.

Conferences

04/2016	Systemic Risk Centre (SRC) Conference on "Capital Flows, Systemic Risk, and Policy Responses" of the London School of Economics and the Central Bank of Iceland, Reykjavik, Iceland
09/2015	3rd International Workshop on "Macro, Banking, and Finance", University of Pavia, Italy
11/2013	1st Bordeaux Workshop on "Agent-based Macroeconomics", University of Bordeaux IV, France
10/2013	Deutsche Bundesbank/SAFE Conference on "Supervising Banks in Complex Financial Systems", University of Frankfurt, Germany
09/2013	1st Meeting of the German Network for New Economic Dynamics (GENED), University of Bielefeld, Germany
09/2012	3rd International Workshop on "Managing Financial Instability in Capitalist Economies (MAFIN)", University of Genova, Italy
08/2012	International Symposium on "Crisis in Macroeconomics" at University of Bochum, Germany

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich meine Doktorarbeit *"Banking, Shadow Banking, and Financial Regulation: An Agent-based Approach"* selbstständig und ohne fremde Hilfe angefertigt habe und dass ich als Koautor maßgeblich zu den weiteren Fachartikeln beigetragen habe. Alle von anderen Autoren wörtlich übernommenen Stellen, wie auch die sich an die Gedanken anderer Autoren eng anlehnenden Ausführungen der aufgeführten Beiträge wurden besonders gekennzeichnet und die Quellen nach den mir angegebenen Richtlinien zitiert.

Datum	Unterschrift

Appendix A

Source Code

A.1 Control Files

A.1.1 Main File

main.scala

```scala
/**
 * @author  Sebastian Krug
 * @version 0.1
 * @date    Tue Mai 14 14:48:41 EST 2013
 * @see     LICENSE (CAU style license file).
 * @compile scalac -cp ../../classes -d classes monEcon.scala
 * @run     scala -cp ../../classes:classes event.Bank
 */


package monEcon

import scala.Console
import java.io._
import java.util._
import scala.sys.process._


// --------------------------------- Start of Programm ----------------------------------------
object Main extends IO {


  def main (args: Array[String]) {
    val ticks                = args(0).toInt
    val seed                 = args(1).toInt
    val initialInterest      = args(2).toDouble
    val delta_pi             = args(3).toDouble
    val delta_x              = args(4).toDouble
    val delta_s              = args(5).toDouble
    val CFSItarget           = args(6).toDouble
    val finReg               = args(7).toBoolean
    val creditToGDPratioinTR = args(8).toBoolean
    val shadowBanks          = args(9).toBoolean
    val level                = 1
    val profilePerformance   = false
    val pln                  = false



    // initialize Set of Simulations
    val simul =
      level match {
        case 0 => Simulation(ticks = ticks, seed = seed, CFSItarget = CFSItarget, finReg = finReg, creditToGDPratioinTR = creditToGDPratioinTR, shadowBanks = shadowBanks)
        case 1 => Simulation(ticks = ticks, seed = seed, initialTargetRate = initialInterest, delta_pi = delta_pi, delta_x = delta_x, delta_s = delta_s, CFSItarget = CFSItarget, finReg = finReg, creditToGDPratioinTR = creditToGDPratioinTR, shadowBanks = shadowBanks)
        case 2 => Simulation(ticks = ticks, seed = seed, initialTargetRate = initialInterest, delta_pi = delta_pi, delta_x = delta_x, delta_s = delta_s, CFSItarget = CFSItarget, finReg = finReg, creditToGDPratioinTR = creditToGDPratioinTR, shadowBanks = shadowBanks)
        case _ => sys.error("Wrong level of financialization")
      }

    createDirectory(s"simData")
    Console.setOut(new FileOutputStream(s"simData/console_output_${simul.seed}.txt"))                 // redirects the console output into the specified log-file
    if(pln) println(s"New $simul starts")
    if(pln) println("--")
```

main.scala

```

    simul.start

    if(pln) println(s"$simul was successful")


  }// End of main-def



}// End of Main-object
```

A.1.2 Simulation Class

Simulation.scala

```
/**
 * @author Sebastian Krug
 *
 */

package monEcon

import monEcon.financialSector.Bank
import monEcon.financialSector.MMMF
import monEcon.financialSector.BrokerDealer
import monEcon.publicSector._
import monEcon.realSector._
import monEcon.Markets._

import collection.mutable.{ListBuffer, ArrayBuffer, Set}
import util.Random
import java.util._
import java.text.SimpleDateFormat




case class Simulation (numberOfHH          :Int      =   125,
                       numberOfFirms       :Int      =    25,
                       ticks               :Int,
                       seed                :Int      =     1,
                       initialTargetRate   :Double   =     0.03,
                       delta_pi            :Double   =     1.25,
                       delta_x             :Double   =     0.5,
                       delta_s             :Double   =     0.0,
                       CFSItarget          :Double,
                       finReg              :Boolean,
                       creditToGDPratioinTR:Boolean,
                       shadowBanks         :Boolean
                                                              ) extends SaveResults with round with entryExit with codeProfiling with accountManagement with bonds {

  val centralBankMoneyBD   = false
  val regulatedShadowBanks = false
  val stricterRegulatedSB  = false


  val tradBanks            = true
  val CFSIbackstop         = false
  val sim                  = this
  val profilePerformance   = false
  val test                 = false
  val testSB               = false
  val pln                  = false


  val zeitstempel      = new Date()
  val simpleDateFormat = new SimpleDateFormat("ddMMyyyy")
  val date             = simpleDateFormat.format(zeitstempel)

```

```
/*     ---------------     Set initial values (general parameters) --- policy variables depending on past values & expectations     ---------------*/
val random                              = new Random(seed)                                  // create Random Generator depending on seed
val profileMethods                      = true
val numberOfBanks:Int                   =    5
val numberOfMMMF                        = numberOfBanks
val numberOfBrokerDealer                = numberOfBanks
val numberOfShadowBanks                 = numberOfBanks
val initialPriceOfGood                  =   200.00
val initialWage                         =  1000.00
val mortalityRate                       =     0.00
val killingParameter                    =     0.10
val vacancyAvailabilityParameter        =     0.95
val goodAvailabilityParameter           =     1.00
val laborSkillUpdateParameter           =     0.00
val privateFundAvailabilityParameter    =     0.25
val initialCapital                      =  4000.0 * 25
val firmProductivityFactor              =     1.75
val retainedEarningsParameter           =     0.90
val updateFrequency                     =    12
val At                                  = ArrayBuffer[Double](1.0)
val fractionOfDebtBank                  =     0.0
val nbcParameter                        =     0.03
val initialMoney                        = (numberOfHH * initialWage * ticks)/numberOfBanks
val maxTargetRate                       =     0.10
val minTargetRate                       =     0.0025
val inflationTarget                     =     0.02
val yearsOfInactiveMP                   =    12.5
val years2TakeIntoAccountInTR           =     1
val reserveRequirement                  =     0.05
val taylorRule                          = true
val TRpathdependence                    = true
val CCycB                               = if(finReg) true else false
val lambdaCCycB                         = 1600
val CConB                               = if(finReg) true else false
val LR                                  = if(finReg) true else false
val surcharges                          = if(finReg) true else false
val publicConfidenceLevel               = ArrayBuffer[Double](0.5)
val withdrawFundsFromSBsector           = true

// initialize agents
val arge                                = new ARGE
val laborMarket                         = new LaborMarket(this)
val goodsMarket                         = new GoodsMarket(this, initialPriceOfGood)
```

```scala

val centralBank                         = new CentralBank(initialTargetRate,
                                                  maxTargetRate,
                                                  minTargetRate,
                                                  initialTargetRate + 0.0025,
                                                  math.max(initialTargetRate - 0.0025,  0.0025),
                                                  reserveRequirement,
                                                  delta_pi,
                                                  delta_x,
                                                  delta_s,
                                                  inflationTarget,
                                                  yearsOfInactiveMP,
                                                  years2TakeIntoAccountInTR,
                                                  this,
                                                  taylorRule,
                                                  TRpathdependence,
                                                  CCycB,
                                                  CFSItarget,
                                                  creditToGDPratioinTR
                                                                      )



val interbankMarket                     = new InterbankMarket(this,
                                                      centralBank
                                                                      )

val government                          = new Government(tradBanks,
                                                 initialMoney,
                                                 centralBank,
                                                 goodsMarket,
                                                 laborMarket,
                                                 interbankMarket,
                                                 4 * initialPriceOfGood,
                                                 0.2,
                                                 0.6,
                                                 0.25,
                                                 0.75,
                                                 nbcParameter,
                                                 this
                                                                       )

val supervisor                          = new Supervisor(this,
                                                 0.045,
                                                 0.03
                                                                       )

var bankList                            = Seq.tabulate(numberOfBanks)(n => Bank(s"$n",
                                                                      fractionOfDebtBank,
                                                                      random,
                                                                      centralBank,
                                                                      interbankMarket,
                                                                      this
                                                                                          ) )
```

```scala


var MMMFList = Seq.tabulate(numberOfMMMF)(n => MMMF(s"$n", random, centralBank, this, bankList(random.nextInt(numberOfBanks))) )
    MMMFList.foreach{mmmf => mmmf.houseBank.MMMFClients += mmmf}

var BrokerDealerList = Seq.tabulate(numberOfBrokerDealer)(n => BrokerDealer(s"$n", random, centralBank, this, bankList(random.nextInt(numberOfBanks))) )
    BrokerDealerList.foreach{BD => BD.houseBank.BDClients += BD}


var hhList                            = Seq.tabulate(numberOfHH)   (n => HH(s"$n",
                                                                            numberOfHH,
                                                                            numberOfFirms,
                                                                            numberOfBanks,
                                                                            tradBanks,
                                                                            random,
                                                                            bankList(random.nextInt(numberOfBanks)),
                                                                            MMMFList(random.nextInt(numberOfMMMF)),
                                                                            goodsMarket,
                                                                            laborMarket,
                                                                            interbankMarket,
                                                                            government,
                                                                            randomGaussian4truncatedND(1, 0.5),
                                                                            arge,
                                                                            randomProbability,
                                                                            randomProbability,
                                                                            randomProbability,
                                                                            vacancyAvailabilityParameter,
                                                                            goodAvailabilityParameter,
                                                                            this
                                                                                                            )        )




hhList.foreach{
  hh =>
    hh.reservationWage.update(0, initialPriceOfGood * 4 - (100 * hh.laborSkillFactor.head))
    hh.houseBank.retailClients += hh
}

val avgInitialLS = average( sim.hhList.map { _.laborSkillFactor.last } )

val initialProductionTarget = (2 * rounded( math.pow(hhList.map(_.laborSkillFactor.last).sum, 1-0.2)) ) / numberOfFirms

var firmList                          = Seq.tabulate(numberOfFirms)(n => Firm(s"$n",
                                                                              numberOfHH,
                                                                              numberOfFirms,
                                                                              numberOfBanks,
                                                                              tradBanks,
                                                                              arge,
                                                                              random,
                                                                              bankList(random.nextInt(numberOfBanks)),
                                                                              BrokerDealerList(random.nextInt(numberOfBrokerDealer)),
                                                                              goodsMarket,
```

```scala
                                                                              laborMarket,
                                                                              interbankMarket,
                                                                              government,
                                                                              0,
                                                                              rounded(initialPriceOfGood - 10 * random.nextDouble),
                                                                              rounded(initialWage         + random.nextDouble),
                                                                              initialProductionTarget,
                                                                              firmProductivityFactor,
                                                                              privateFundAvailabilityParameter,
                                                                              retainedEarningsParameter,
                                                                              initialCapital,
                                                                              this
                                                                                                                                   )    )

firmList.foreach{
  firm =>
    firm.houseBank.businessClients += firm
    firm.houseShadowBank.clients   += firm
 }

def unemploymentRate = 1.0 - sim.firmList.map { firm => firm.employees.size + firm.queuedEmployees.size }.sum / sim.numberOfHH
def faceValueOfBonds = 10000.0

val listOfCorporations   = if(tradBanks) bankList ++: firmList         else firmList
val listOfShadowBanks    = if(tradBanks) MMMFList ++: BrokerDealerList else Seq()

val corporations2BeFound = random shuffle(listOfCorporations).to[ArrayBuffer]
val  shadowBanks2BeFound = random shuffle(listOfShadowBanks ).to[ArrayBuffer]






// Initialization Methods
/**
 *
 *
 *  This method is to found the corporations in the model, i.e. firms, banks, BD and MMF.
 *
 *     */
 def assignCorporationOwners {
   val hhInvestment = collection.mutable.Map[HH, collection.mutable.Map[Corporation, Double]]()




   // assign owners to corporations
   hhList.foreach{
     hh =>
       val money = if(tradBanks) hh.bankDeposits.last else hh.cash.last
       if(money > 1000.0){
         if(corporations2BeFound.nonEmpty){
           corporations2BeFound.head.owners += hh
```

```
            hh.foundedCorporations += corporations2BeFound.head
            corporations2BeFound   -= corporations2BeFound.head
          } else {
            val banksNeedingOwners = if(tradBanks) bankList.filter(_.owners.size < (numberOfHH / 3)/numberOfBanks) else Seq[Bank]()
            val corp = if(banksNeedingOwners.nonEmpty) banksNeedingOwners(random nextInt(banksNeedingOwners length)) else listOfCorporations(random nextInt(listOfCorporations length))
            corp.owners += hh
            hh.foundedCorporations += corp
          }
        }
    }

    listOfShadowBanks.foreach {
      ShadowBank =>
        val newOwners = random.shuffle(hhList).take(10 + random.nextInt(numberOfHH/5) )
        newOwners.foreach {
          hh =>
            ShadowBank.owners += hh
            hh.foundedCorporations += ShadowBank
        }
    }
    if(test) listOfShadowBanks.foreach { shadowBank => shadowBank.owners.nonEmpty }

    // transfer money to founded corporations and store relationships in hhInvestment
    hhList.foreach{
      hh =>
        val money = if(tradBanks) hh.bankDeposits.last else hh.cash.last
        if(hh.foundedCorporations.nonEmpty){
          val investment = (money - 1000.0) / hh.foundedCorporations.size
          println(s"$hh founded ${hh.foundedCorporations} with $investment (${roundUpTo1000(money) - 1000}) of ${hh.bankDeposits.last}")
          if(test) require(hh.foundedCorporations.size == 1, s"$hh has founded more than 1 corp")
          hh.foundedCorporations.foreach {
            corp =>
              if(hhInvestment.contains(hh)) hhInvestment(hh) += corp -> investment else hhInvestment += hh -> collection.mutable.Map(corp -> investment)
              println(s"$hh transfers $investment to $corp and has bD of ${hh.bankDeposits.last}")
              corp match {
                case corpF:Firm        => transferMoney(hh, corpF, investment, "initialInvestmentF", sim, 1)
                case corpB:Bank        =>
                  corpB.foundMe(money)
                  transferMoney(hh, hh.houseBank, roundUpTo1000(money) - 1000, "initialInvestmentB", sim, 1)
                case mmmf:MMMF         => transferMoney(hh, mmmf, investment,  "foundMMMF",          sim, 1)
                case   bd:BrokerDealer => transferMoney(hh,   bd, investment,  "foundBrokerDealer",  sim, 1)
                case _ => sys.error("Currently only Firms and Banks can be found...")
              }
          }
        }
        if(hh.bankDeposits.last < 0){
          val missingFunds = -hh.bankDeposits.last + 1000.0
          deposit(hh.bankDeposits,             missingFunds, 0, sim)
          deposit(hh.houseBank.retailDeposits, missingFunds, 0, sim)
        }
    require(hh.bankDeposits.last >= 0, s"$hh has negative bankDeposits after founding corps: ${hh.bankDeposits.last}")
    }

```

```scala
    // calculate and assign share
    hhList.foreach{
      hh =>
        hh.foundedCorporations.foreach{
          foundedCorp =>
            val share = hhInvestment(hh)(foundedCorp) / foundedCorp.owners.map(owner => hhInvestment(owner)(foundedCorp)).sum
            hh.shareOfCorporations += foundedCorp -> share
            if(pln) println(s"$hh founded $foundedCorp with a share of $share")
            if(test) require(share <= 1, s"share has to be <= 1 but its not: $share")
            if(pln) println(hh.shareOfCorporations)
        }
    }

    // test of share
    if(test){
      hhList.foreach{
        hh =>
          hh.foundedCorporations.foreach{
            corp =>
              require( rounded(corp.owners.map(_.shareOfCorporations(corp)).sum) == 1, s"Owners of $corp own more than 100%: ${corp.owners.map(_.shareOfCorporations(corp)).sum}" )
          }
      }
      firmList.foreach(firm => assert(firm.owners.nonEmpty, s"$firm has no owners although it should!"))
    }

  }// method



  /**
   *
   *  This method adjusts the public confidence level according to the central banks monetary policy decisions.
   *
   */
  def adjustPublicConfidenceLevel (t:Int) = {
    if(withdrawFundsFromSBsector && t >= yearsOfInactiveMP * 48 && t % 48 % 6 == 0){
      val newPCL = 1.1 - 10 * centralBank.targetFFR.last
      publicConfidenceLevel(publicConfidenceLevel.size-1) = newPCL
    }
  }





  val numberOfActiveFirms   = ArrayBuffer[Int](numberOfFirms)
  val numberOfActiveBanks   = ArrayBuffer[Int](numberOfBanks)
  val numberOfActiveMMMF    = ArrayBuffer[Int](numberOfMMMF)
```

```scala
val numberOfActiveBD      = ArrayBuffer[Int](numberOfBrokerDealer)
val numberOfBailOuts      = ArrayBuffer[Int]()
val reserveFlows          = bankList.map(bank => bank -> bankList.filter(_ != bank).map(_ -> ArrayBuffer[Double](0.0)).toMap ).toMap
val IBMloanFlows          = bankList.map(bank => bank -> bankList.filter(_ != bank).map(_ -> ArrayBuffer[Double](0.0)).toMap ).toMap
val IDLflows              = bankList.map(bank => bank -> ArrayBuffer[Double](0.0) ).toMap
val bondIDs               = collection.mutable.Set[Long]()
val saveTickTime          = ArrayBuffer[Long]()
val expPi                 = ArrayBuffer[Double]()
val expRealIntRate        = ArrayBuffer[Double]()                              // r_t
val longRunRealIntRate    = ArrayBuffer[Double]()                              // r*
val investmentSBsector    = ArrayBuffer[Double]()                              //
val withdrawFromSBsector  = ArrayBuffer[Double]()                              //
val sizeTBsector          = ArrayBuffer[Double]()                              //
val sizeSBsector          = ArrayBuffer[Double]()                              //
val equityTBsector        = ArrayBuffer[Double]()                              //
val equitySBsector        = ArrayBuffer[Double]()                              //
val investedFundsSBsector = ArrayBuffer[Double]()                              //
val neededLiquidityFirms  = ArrayBuffer[Double]()                              //
val offeredFundsMMMF      = ArrayBuffer[Double]()
val offeredLiquidityBD    = ArrayBuffer[Double]()                              //
val creditGrantedByBD     = ArrayBuffer[Double]()                              //
val creditGrantedByTB     = ArrayBuffer[Double]()                              //
val liquidityGap          = ArrayBuffer[Double]()                              //


/**
 *
 *  This method determines the exogenous technological progress underlying the model.
 *
 *    */
def techProgress (a:Double = At.last) = At += a * math.exp(0.012 / (48 / updateFrequency))


/**
 *
 *  This method determines the expected inflation.
 *
 *    */
def determineExpInflation = {
  val T = 24
  val weights                         = collection.immutable.Vector.tabulate(T)(x => (T + 1 - (x+1)) / (0.5 * T * (T + 1)) )
  val avgMonthlyPrices                = goodsMarket.weightedAvgPriceOfMonth.takeRight(T+1).reverse
  val annualizedMonthlyInflation      = collection.immutable.Vector.tabulate(T)(n => 12 * ( math.log(avgMonthlyPrices(n)) - math.log(avgMonthlyPrices(n+1)) ) )
  val sumOfWeightedPastMonthlyInflation = collection.immutable.Vector.tabulate(T)(n => annualizedMonthlyInflation(n) * weights(n) ).sum
  expPi += 0.25 * inflationTarget + 0.75 * sumOfWeightedPastMonthlyInflation
}

def determineExpRealIntRate = expRealIntRate += centralBank.targetFFR.last - expPi.last
```

```
/*    --------------------------------------------------    START OF SIMULATION
------------------------------------------------------------------------------------------------------------    */
def start = {
    val startingTime = System.nanoTime
    if(pln) println("* Bring Money into the System\n")
    if(pln) println(sim)
    checkBankRetailDeposits(0, "before starting sim")
    government issueInitialGovBonds(initialMoney)
    checkBankRetailDeposits(0, "after initial issuance of GovBonds")
    government payUnemploymentBenefit2HH(); if(test) testSFC("gov_payUnemploymentBenefit", 1)
    checkBankRetailDeposits(0, "after initial payment of unemployment benefit")
    if(testSB) testAmountOfOutstandingBonds(1)
    assignCorporationOwners; if(test) testSFC("sim_assignCorporationOwners", 1)
    if(test) assume(corporations2BeFound.isEmpty, "There are Corporations which aren't yet founded: " + corporations2BeFound.toString)
    checkBankRetailDeposits(0, "after founding corps")




  /*    -------------------- MAIN
LOOP---------------------------------------------------------------------------------------------------------------------------------------------------    */
  for(t <- 1 to ticks){
    val startOfTick = System.nanoTime()
    if(pln) println(s"__Tick $t __  ")
    time(addTickValue(t), "addTickValue", this)
    testBonds(t, "At start of Tick", "BEFORE")
    if(sim.testSB) testAmountOfOutstandingBonds(t)
    if(pln) println(s"reserveFlows: ${reserveFlows.keys}")




    if(t>100 && (t-1) % 4 == 0){
      time(
          random.shuffle(hhList).take( (numberOfHH * killingParameter).toInt ).foreach{
            hh =>
              if(random.nextDouble < mortalityRate){
                hh.laborSkillFactor update(hh.laborSkillFactor.length-1, randomGaussian4truncatedND(1, 0.5))
                hh.employers.last match {
                  case employer:Firm    => employer.fireHH(hh)
                  case employer:ARGE    =>
                  case _                =>
              }
            }
        }
      , "hh_randomDeath", this)
    }





    if(test){
```

```scala
        bankList.filter(_.active).foreach{
          bank =>
            bank.listOfBonds.keys.foreach{                    id => if(t > government.findStackOfBondsByID(id).bond.maturity) sys.error(s"maturity of bond in listOfBonds of $bank is already over-due: ${government.findStackOfBondsByID(id).bond.maturity} / $t")}
            bank.bondsPledgedAsCollateralForOMO.keys.foreach{ id => if(t > government.findStackOfBondsByID(id).bond.maturity) sys.error(s"maturity of bond in bondsPledgedAsCollateralForOMO of $bank is already over-due: ${government.findStackOfBondsByID(id).bond.maturity} / $t")}
            bank.bondsPledgedAsCollateralForIDL.keys.foreach{ id => if(t > government.findStackOfBondsByID(id).bond.maturity) sys.error(s"maturity of bond in bondsPledgedAsCollateralForIDL of $bank is already over-due: ${government.findStackOfBondsByID(id).bond.maturity} / $t")}
            bank.bondsPledgedAsCollateralForOSLF.keys.foreach{id => if(t > government.findStackOfBondsByID(id).bond.maturity) sys.error(s"maturity of bond in bondsPledgedAsCollateralForOSLF of $bank is already over-due: ${government.findStackOfBondsByID(id).bond.maturity} / $t")}
        }
      }




      bankList.foreach(bank => if(bank.active == false && bank.periodOfReactivation == t) bank.reactivateBank(t)); if(test) testSFC("bank_reactivateBank", t)
      p(t, "reactivate Banks")
      bankList.foreach(bank => if(bank.active) bank.updateBankAge)
      numberOfActiveBanks +=         bankList.filter(_.active).size
      numberOfActiveMMMF  +=         MMMFList.filter(_.active).size
      numberOfActiveBD    += BrokerDealerList.filter(_.active).size

      if(t>1 && (t-1) % 4 == 0)           hhList.foreach(_.switchHouseBank(t))
      if(t>1 && (t-1) % 4 == 0)         firmList.foreach(_.switchHouseBank(t))
      p(t, "HH/Firms switchHB")

      val initialPVofBonds = if(test) bankList.filter(_.active).map(_.currentPVofSoBs(t)) else Seq[Double]()
      if(pln) println(s"initialPVofBonds at beginnig of tick $t: $initialPVofBonds")
      def testPVofBonds = initialPVofBonds.zip(bankList.filter(_.active).map(_.currentPVofSoBs(t))).foreach(
          tuple =>
            require(
                math.pow(rounded(tuple._2) - rounded(tuple._1), 2) <= math.max(100, tuple._1 * 0.0001),
                s"Deviation in PV of Bonds. Beginning of tick $t: ${tuple._1} vs. ${tuple._2}. This corresponds to a deviation of ${rounded(((tuple._2 - tuple._1) / tuple._1) * 100)}%"
            )
      )
      if(SE(bankList.map(_.currentPVofBonds(t)), initialPVofBonds) sys.error(s"Deviation in PV of Bonds. Beginning of tick $t: $initialPVofBonds vs. ${bankList.map(_.currentPVofBonds(t))}. This corresponds to a deviation of ${rounded(((bankList.map(_.currentPVofBonds(t)).head - initialPVofBonds.head) / initialPVofBonds.head) * 100)}% and ${rounded(((bankList.map(_.currentPVofBonds(t)).last - initialPVofBonds.last) / initialPVofBonds.last) * 100)}%")



      /*    -----    BEGIN OF SETTLEMENT DAY -------------------------------------------------------------------------------------------------------------------------------------------
       *     From here on, all transactions of the economy include the usage of the payment system:
       * */
      if(tradBanks){
        if((t-1) %  4 == 0)    bankList.filter(_.active).foreach(_.setReserveTarget);                    if(test) testSFC("banks_setReserveTarget", t); if(test) testPVofBonds
                               p(t, "Banks set reserveTarget")
                               bankList.filter(_.active).foreach(_.repayIBMloans(t));                    if(test) testSFC("banks_repayIBMloans",    t)
                               p(t, "Banks repayIBMloans")
                               bankList.filter(_.active).foreach(_.repayOSF(t));                         if(test) testSFC("banks_repayOSF",         t)
                               p(t, "Banks repayOSF to CB")
```

```scala
      if((t-1) %  4 == 0)    bankList.filter(_.active).foreach(_.monthlyRepoToAquireTargetReserve(t)); if(test) testSFC("banks_monthlyRepo",      t)
                             p(t, "Banks do omnthlyRepoForTargetReserves")
    }
    if(testSB) testBonds(t, "After beginn of Settlement Day", "AFTER")


    if(t>12 && t % 12 == 0) {
      hhList.foreach( hh => hh.payBankAccountFee(t) )
      government.payBankAccountFee(t)
    }



    if(testSB) testAmountOfExistingIDs(t, "")
    if(test) {
      testAmountOfOutstandingFirmDebt(t, "before", "aquireFunding", false)
    }

    if(t>1 && t % updateFrequency == 0) techProgress()


 if(shadowBanks){

   if(testSB) testBonds(t, "Before shadow banking activity", "BEFORE")
    BrokerDealerList.foreach(BD => if(BD.active == false && BD.periodOfReactivation == t) BD.reactivateBrokerDealer(t))
    BrokerDealerList.foreach(BD => if(BD.active) BD.updateAge)
    p(t, "Reactivate BD")
    MMMFList.foreach(mmmf => if(mmmf.active == false && mmmf.periodOfReactivation == t) mmmf.reactivateMMMF(t))
    MMMFList.foreach(mmmf => if(mmmf.active) mmmf.updateAge)
    if(regulatedShadowBanks || stricterRegulatedSB) if(centralBankMoneyBD) centralBank.liquidityInsuranceDebtBD.keys.foreach(BD => BD.repayCBdebt(t))
    p(t, "Reactivate MMMF")

    if(testSB) testBonds(t, "Before HH adjust specFunds", "BEFORE")
    // 1.
    if(t>12 && t % 12 == 0) hhList.foreach(_.adjustSpeculativeFunds(t))
    p(t, "HH adjust specFunds")
    if(testSB) testBonds(t, "After HH adjust specFunds and before MMMF decide2RollOverRepos", "AFTER")
    // 2.
    MMMFList.foreach { mmmf => if(mmmf.active) mmmf.Decide2RollOverRepos(t) }
    if(testSB) testBonds(t, "After MMMF decide2RollOverRepos and before BD repurchaseCollateral", "AFTER")
    p(t, "MMMF decide about repo roll over")
    // 3.
    println(s"Before SB activity/before repurchase collateral: ${BrokerDealerList.map { _.bankDeposits.last}}")
    BrokerDealerList.foreach { BD => if(  BD.active)   BD.repurchaseCollateral(t) }
    println(s"After repurchase collateral: ${BrokerDealerList.map { _.bankDeposits.last}}")
    if(testSB) testBonds(t, "After BD repurchaseCollateral and before MMMF repayFunds", "AFTER")
    p(t, "BD repurchase collateral (if any)")
    // 4.
    MMMFList.foreach { mmmf => if(mmmf.active) mmmf.repayFunds(t) }
    MMMFList.foreach { mmmf => if(mmmf.active && mmmf.funds2repay.nonEmpty) sys.error(s"funds2repay of $mmmf are not empty after repay in tick $t.") }
    if(testSB) testBonds(t, "After MMMF repayFunds and before BD buy more GovBonds", "AFTER")
    p(t, "MMMF repay withdrawn funds to investors/hh")
    // 5.
    println(s"Before buy add bonds: ${BrokerDealerList.map { _.bankDeposits.last}}")
```

```
    BrokerDealerList.filter(_.active).foreach { _.securitizeAndSellLoans(t) }
    println(s"After buy add bonds: ${BrokerDealerList.map { _.bankDeposits.last}}")
    if(testSB) testBonds(t, "After BD buy more GovBonds and before BD doOvernightRepos", "AFTER")
    p(t, "BD decide about buying new Gov bonds")
    MMMFList.filter(_.active).foreach { mmmf => deposit(offeredFundsMMMF, mmmf.offeredAmountOfFunds, t, this) }
    println(s"before repo: ${BrokerDealerList.map { _.bankDeposits.last}}")
    BrokerDealerList.foreach {BD => if(BD.active)   BD.doOvernightRepo(t) }
    println(s"After repo: ${BrokerDealerList.map { _.bankDeposits.last}}")
    if(testSB) testBonds(t, "After shadow banking activity", "AFTER")
    p(t, "BD doOvernightRepos with MMMF")

  }



    firmList.foreach(firm => if(firm.active == false && firm.periodOfReactivation == t) firm.reactivateFirm(t)); if(test) testSFC("firm_reactivateFirm", t)
    firmList.foreach(firm => if(firm.active) firm.updateFirmAge)
    p(t, "Reactivation Firm")
    numberOfActiveFirms  += firmList.filter(_.active).size
    offeredLiquidityBD += BrokerDealerList.filter(_.active).map {
      bd =>
        val repoFees2payTomorrow = rounded(bd.outstandingRepos.map { _.overnightFee }.sum)
        math.max(0, rounded(bd.bankDeposits.last - repoFees2payTomorrow - 1000))
    }.sum
    random.shuffle(firmList).foreach{
      firm =>
        if(firm.active){
          if(t>12 && t % 12 == 0)     firm payBankAccountFee(t)
          p(t, "Firm pay bank account fee")
          if(t>1  && (t-1) %  4 == 0) firm employHH(t)
          p(t, "Firm employ hh")
          if(t>1 && (t-1) % updateFrequency == 0)    firm determineProductionTarget(t)
          p(t, "Firm determine production target")
          if(t>1 && (t-1) % updateFrequency == 0)    firm determineOfferedWageFactor(t)
          p(t, "Firm determine offered wages")
          if(t>1 && (t-1) % updateFrequency == 0)    firm updateWages(t)
          p(t, "Firm update wages")
          if(t>1 && (t-1) % updateFrequency == 0)    firm determineExternalFinancing(t)
          p(t, "Firm determine external financing")
                                                     firm aquireFunding(t);             if(test) testSFC("firm_aquireFunding", t)
                                                     p(t, "Firm aquire funding")
                                                     firm announceCurrentJobs(t)
                                                     p(t, "Firm announce jobs")
          if(t>1 && (t-1) % updateFrequency == 0)    firm fireEmployees(t)
                                                     p(t, "Firm fire empoyees")
```

```
      } else {
        if(t>1 && (t-1) % updateFrequency == 0)    firm.productionTarget         += 0.0
        if(t>1 && (t-1) % updateFrequency == 0)    firm.offeredWages             += 0.0
        if(t>1 && (t-1) % updateFrequency == 0)    firm.needForExternalFinancing += 0.0
                                                   firm.announceCurrentJobs(t, 0)
      }
    }
    if(test) {
      testAmountOfOutstandingFirmDebt(t, "after", "aquireFunding", false)
    }
    if(testSB) testAmountOfExistingIDs(t, "After Firm period planning methods")

    if(t>1 && (t-1) % 48 == 0) government.updateUnemploymentBenefit
    if(t>1 && (t-1) % 4  == 0) government payUnemploymentBenefit2HH(t); if(test) testSFC("gov_payUnemploymentBenefit2", t)
    p(t, "Gov pay unemployment benefit")
    if(testSB) testAmountOfExistingIDs(t, "After Gov paying unemployment benefit")

    if(t>1 && (t-1) % 48 == 0) random.shuffle(hhList).take( (numberOfHH * laborSkillUpdateParameter).toInt ).foreach(hh => hh.updateLaborSkill)
                               random.shuffle(hhList).foreach(hh => if(hh.currentEmployer == arge) hh.searchJob(t) )
                               p(t, "hh search job")

      random.shuffle(firmList).foreach{
        firm =>
          firm.numberOfEmployees +=    firm.employees.size + firm.queuedEmployees.size
          if(firm.active){
                                                  firm.produceGood(t)
            if(t>1 && (t-1) % updateFrequency == 0) firm.determinePrice(t)
                                                  firm.offerGood(t)
          } else {
                                                  firm.producedGoods     += 0.0
                                                  firm.amountOfInventory += 0.0
          if(t>1 && (t-1) % updateFrequency == 0)   firm.price             += 0.0
                                                  firm.offerGood(t)
          }
    }
    p(t, "Firm production")
    if(testSB) testAmountOfExistingIDs(t, "After Firm production")


    goodsMarket.determineWeightedAvgPriceOfTick
    goodsMarket.setCurrentSupply
    government.determineNominalGDP

    if(t > 48 && t % 48 == 0){
      if(tradBanks) centralBank.updateBaseYear(t)
                     government.determineRealGDP(t)
                     government.calcGDPdeflator
                     government.calcGDPdeflatorMP
                    centralBank.determineInflation
                    centralBank.determineInflationMP
    }
    if(t>100 && t % 4 == 0) determineExpInflation
    if(testSB) testAmountOfExistingIDs(t, "After some Gov/CB stuff")

```

```

    if(t>1 && (t-1) % updateFrequency == 0) hhList.foreach(_.planConsumption(t))
    random.shuffle(hhList).foreach(hh => hh.consume(t)); if(test) testSFC("hh_consume", t); if(testSB) testAmountOfExistingIDs(t, "After HH consumption")
    p(t, "hh consume")

    if(tradBanks) random.shuffle(bankList).filter(_.active).foreach(_.payInterestOnDeposits(t)); if(test) testSFC("banks_payInterestOnDeposits", t); if(testSB)
testAmountOfExistingIDs(t, "After Banks pay interest on deposits")
    if(tradBanks && testSB) random.shuffle(bankList).filter(_.active).foreach(bank => require(bank.COGS.last >= 0, s"$bank must have non-negative COGS: ${bank.COGS.last}"))
    p(t, "Banks pay interest on deposits")

    if(test) testAmountOfOutstandingFirmDebt(t, "before", "repayLoan", false)
    random.shuffle(firmList).foreach{
      firm =>
        if(firm.active){
          if(t>1 && t % 4 == 0) firm.payOutWage2HH(t); if(test) testSFC("firm_payWages",  t)
          p(t, "Firms pay wages")
          firm.repayLoan(t);     if(test) testSFC("firm_repayLoan", t)
          p(t, "Firms repay Loans")
        }
    }
    if(testSB) testAmountOfExistingIDs(t, "After Firms repay loans")
    if(test){
      testAmountOfOutstandingFirmDebt(t, "after", "repayLoan", true)
    }

            bankList.filter(_.active).foreach(_.deleteDueBusinessLoans(t))
    BrokerDealerList.filter(_.active).foreach(_.deleteDueBusinessLoans(t))

    // set labor status of hh
    hhList.foreach{
          hh =>
            if(hh.currentEmployer == arge) hh.unemployed += true else hh.unemployed += false
                                           hh.employers  += hh.currentEmployer
    }
    if(testSB) testAmountOfExistingIDs(t, "After setting labor status of HH")
    firmList.foreach(firm => firm.currentProdCap += firm.actualProductionCapacity)

    if(testSB) testBonds(t, "Before Gov payCoupon", "BEFORE")
    government payCoupon(t); if(test) testSFC("gov_payCoupon", t)
    if(testSB) testBonds(t, "After Gov payCoupon", "AFTER")
    p(t, "Gov payCoupon")


    firmList.foreach(firm => firm.makeAnnualReport(t) ); if(test) testSFC("firm_AR", t)
    p(t, "Firm AR")


    bankList.foreach{
      bank =>
        bank.determineProfit
        p(t, "Bank determineProfit")
        if(t>1 && t % 48 == 0) bank.payTaxes(t)
        p(t, "Bank payTaxes")
        if(t>1 && t % 48 == 0) bank.payOutDividends2Owners(t)
```

```
        p(t, "Bank payOutDividends2Owners")
    }

    /*    -----    END OF SETTLEMENT DAY ------------------------------------------------------------------------------------------------------------------------------------    */
    bankList.filter(_.active).foreach(_.repayIDL(t));                                                                                   if(test) testSFC("banks_repayIDL", t);
    p(t, "Banks repayIDL")
    random.shuffle(bankList.filter(bank => bank.active == true && bank._reserveDeficit > 0)).foreach(_.lendOvernightFromIBM(t)); if(test) testSFC("bank_lendOvernightFromIBM", t);
    p(t, "Banks lend on IBM")
    bankList.filter(_.active).foreach(_.useOSFifNecessary(t));                                                                           if(test) testSFC("banks_useOSF", t);
    p(t, "Banks use OSF if necessary")

    if(tradBanks && t % 4 == 0){
      centralBank payInterestOnReserves(t); if(test) testSFC("CB_payInterestOnReserves", t)
    }
    p(t, "CB pays interest on reserves")


    bankList.filter(_.active).foreach{
      bank =>
        if(!centralBank.avgReserves.contains(bank))     centralBank.avgReserves     += bank -> ArrayBuffer(bank._currentAvgReserves) else centralBank.avgReserves(bank)     += bank._currentAvgReserves
        if(!centralBank.deficitReserves.contains(bank)) centralBank.deficitReserves += bank -> ArrayBuffer(bank._reserveDeficit)     else centralBank.deficitReserves(bank) += bank._reserveDeficit
        if(!centralBank.excessReserves.contains(bank))  centralBank.excessReserves  += bank -> ArrayBuffer(bank._excessReserves)     else centralBank.excessReserves(bank)  += bank._excessReserves
    }

  /*    -----    Banking Sector Annual Report    -----    */
  if(tradBanks)                        bankList.foreach(bank => bank.makeAnnualReport(t) ); if(test) testSFC("bank_AR", t)
  p(t, "Bank AR")
  if(tradBanks && shadowBanks)         MMMFList.foreach(mmmf => mmmf.makeAnnualReport(t) )
  p(t, "MMMF AR")
  if(tradBanks && shadowBanks) BrokerDealerList.foreach(BD   =>   BD.makeAnnualReport(t) )
  p(t, "BD AR")
                                                          government.makeAnnualReport(t)
  p(t, "Gov AR")
  if(tradBanks)                        bankList.foreach(_ determineCurrentMarketShare)

  // Monetary Policy
  if(tradBanks){
    centralBank.outstandingPrivateSectorDebt += firmList.map(firm => firm.debtCapital.last + firm.interestOnDebt.last).sum
    centralBank setCentralBankInterestRates(t)
    adjustPublicConfidenceLevel(t)
    if(expPi.nonEmpty) determineExpRealIntRate
    if(CCycB) centralBank setCCycB(t)
  }

  // Monetary Aggregates M0, M1, M3
  time({
    tradBanks match {
      case true        =>
        government.M0(government.M0.size-1) +=                      bankList.filter(_.active).map(_.cbReserves.last     ).sum
```

```scala
        government.M1(government.M1.size-1) +=                      bankList.filter(_.active).map(_.retailDeposits.last ).sum
        government.M3(government.M3.size-1) += government.M1.last + bankList.filter(_.active).map(_.OSFused.last._2     ).sum

      case false    =>
        government.M0(government.M0.size-1) +=   hhList.map(_.cash.last).sum
        government.M0(government.M0.size-1) += firmList.map(_.cash.last).sum

    }
  }, "track_agg_money", this)


  if(t == ticks) bankList.foreach(bank => numberOfBailOuts += bank.bailOutCounter.size)
                    sizeTBsector(sizeTBsector.size-1) +=          bankList.filter(_.active).map { _.totalAssets.last }.sum
  if(shadowBanks)   sizeSBsector(sizeSBsector.size-1) +=          MMMFList.filter(_.active).map { _.totalAssets.last }.sum
  if(shadowBanks)   sizeSBsector(sizeSBsector.size-1) +=  BrokerDealerList.filter(_.active).map { _.totalAssets.last }.sum
                  equityTBsector(sizeTBsector.size-1) +=          bankList.filter(_.active).map { _.equity.last }.sum
  if(shadowBanks) equitySBsector(sizeSBsector.size-1) +=          MMMFList.filter(_.active).map { _.equity.last }.sum
  if(shadowBanks) equitySBsector(sizeSBsector.size-1) +=  BrokerDealerList.filter(_.active).map { _.equity.last }.sum
  investedFundsSBsector(investedFundsSBsector.size-1) += hhList.map { hh => hh.speculativeFunds.map{ case(mmmf, abWithTuple) => abWithTuple.map(_._1).sum }.sum }.sum



  time(saveEndOfTickData(t), "sim_saveEndOfTickData", this)

  val tickTime  = System.nanoTime - startOfTick
  saveTickTime += tickTime
  if(pln) println(f"Time elapsed during tick $t: ${tickTime.toDouble / 1000000000}%1.2f seconds or ${(tickTime.toDouble / 1000000000)/60}%1.2f minutes [seed: $seed]")
  println(s"seed $seed --> numberOfExisting SoBs: ${government.numberOfExistingBonds.last}")
  if(pln) println(f"numberOfActiveBanks: ${bankList.filter(_.active).size} / noeb: ${government.numberOfExistingBonds.last}")

  // memory info
  val mb       = 1024.0 * 1024.0
  val runtime  = Runtime.getRuntime
  val usedRAM  = (runtime.totalMemory - runtime.freeMemory) / mb
  val freeRAM  = runtime.freeMemory / mb
  val totalRAM = runtime.totalMemory / mb
  val maxRAM   = runtime.maxMemory / mb
  if(usedRAM > 0.9 * maxRAM && pln){
    println( "##### Heap utilization statistics [MB] #####")
    println(s"** Used Memory: $usedRAM")
    println(s"** Free Memory: $freeRAM")
    println(s"** Total Memory: $totalRAM")
    println(s"** Max Memory:  $maxRAM")
  }
} // -------------------- End of Main Loop --------------------








  government.determineEconomicGrowth
  if(CCycB) centralBank.HPfilterData(centralBank.credit2GDPratio, lambdaCCycB).foreach(centralBank.credit2GDPtrend += _)
```

```
    time(saveEndOfSimulationData, "sim_saveEndOfSimulationData", this)

    // deviations through roundings
    if(pln) println(s"Cummulated deviation of govDeposits:     ${govDepositsDev.head} (${govDepositsDev.head / government.bankDeposits.last}%)")
    if(pln) println(s"Cummulated deviation of retailDeposits:  ${retailDepositsDev.head}")
    if(pln) println(s"Cummulated deviation of reserveAccounts: ${reserveAccountDev.head}")



    // timing
    val elapsedTime = System.nanoTime - startingTime
    println("\n        " + elapsedTime                 + " ns")
    println(f"        ${elapsedTime.toDouble / 1000000}%2.2f ms")
    println(f"        Total simulation time: ${elapsedTime.toDouble / 1000000000}%1.2f seconds or ${(elapsedTime.toDouble / 1000000000)/60}%1.2f minutes")
    saveSimulationData(sim.toString, "duration", "simulation", s"${elapsedTime.toDouble / 1000000000}%1.2f seconds", sim.government, 1)

  }// -------------------- End of start-method --------------------




   /**
    *
    *    */
    def testSFC (method:String, t:Int) {
      // aggregate
      if(centralBank.reserves.last > 1000) require(
          SE(bankList.filter(_.active).map(_.cbReserves.last).sum, centralBank.reserves.last),
          s"reserves are not consistent, deviation is ${rounded(bankList.filter(_.active).map(_.cbReserves.last).sum) - centralBank.reserves.last} (CB);\n ${bankList.map(bank => bank -> (bank.active, bank.cbReserves.last))} "
          )
      try{
        require(
            math.pow(rounded(bankList.map(_.govDeposits.last).sum) - government.bankDeposits.last, 2) <= 5,
            s"govDeposits are not consistent: the deviation is ${rounded(bankList.map(_.govDeposits.last).sum) - government.bankDeposits.last} (BankAcc - GovAcc)"
        )
      } catch {
        case a:java.lang.IllegalArgumentException =>
          rounded(bankList.map(_.govDeposits.last).sum) - government.bankDeposits.last match {
            case deviation:Double if deviation > 0 => deposit( government.bankDeposits,  deviation, t)
            case deviation:Double if deviation < 0 => withdraw(government.bankDeposits, -deviation, t)
          }
      } finally {
        require(
            math.pow(rounded(bankList.map(_.govDeposits.last).sum) - government.bankDeposits.last, 2) <= 25,
            s"govDeposits are not consistent: pos. deviation of ${rounded(bankList.map(_.govDeposits.last).sum) - government.bankDeposits.last} (BankAcc - GovAcc) after adjustment."
        )
      }
      checkAndAdjust("govDeposits", "after", method, bankList.filter(_.active).map(_.govDeposits.last).sum, government.bankDeposits, t)
      // individual
      bankList.filter(_.active).foreach{
        bank =>
          if(!Seq("banks_repayIDL", "bank_lendOvernightFromIBM").contains(method)) require(
```

```scala
            bank.cbReserves.last >= -0.2, s"checkReserveAccounts (testSFC) failed after $method: $bank has negative reserve account: ${rounded(bank.cbReserves.last)}"
            )
        val firmDepositsOfBank = bank.businessClients.map(_.bankDeposits.last).sum
        val   hhDepositsOfBank = bank.retailClients.map(  _.bankDeposits.last).sum
        checkAndAdjust("checkBankDeposits", "after", method, firmDepositsOfBank + hhDepositsOfBank, bank.retailDeposits, t)
        require(
            math.pow(rounded(firmDepositsOfBank + hhDepositsOfBank) - bank.retailDeposits.last, 2) < 5,
            s"checkBankDeposits (testSFC) failed after $method: deviation is ${rounded(firmDepositsOfBank + hhDepositsOfBank) - bank.retailDeposits.last} ($bank)"
        )
    }
}





def testBonds (t:Int, cause:String, when:String) = {
  if(testSB){
    testAmountOfOutstandingBonds
    bankList.filter(_.active).foreach(_.checkExistenceOfIDs(when, cause))
    BrokerDealerList.filter(_.active).foreach(_.checkExistenceOfIDs(when, cause))
    testAmountOfExistingIDs(t, cause)
  }
}









def testAmountOfOutstandingBonds(t:Int) {
    println(government.govLOB)
    bankList.foreach{bank => println(s"LoB of $bank: ${bank.listOfBonds.map{case(id, fraction) => government.findStackOfBondsByID(id)}}");
    println(s"IDL of $bank: ${bank.bondsPledgedAsCollateralForIDL.map{case(id, fraction) => government.findStackOfBondsByID(id)}}") ;
    println(s"OMO of $bank: ${bank.bondsPledgedAsCollateralForOMO.map{case(id, fraction) => government.findStackOfBondsByID(id)}}");
    println(s"OSLF of $bank: ${bank.bondsPledgedAsCollateralForOSLF.map{case(id, fraction) => government.findStackOfBondsByID(id)}}")}
    BrokerDealerList.foreach{bd => println(s"LoB of $bd: ${bd.listOfBonds.map{case(id, fraction) => government.findStackOfBondsByID(id)}}");
    println(s"pledged4Repo of $bd: ${bd.bondsPledgedAsCollateralForRepo.map{case(id, fraction) => government.findStackOfBondsByID(id)}}") }
    val amountOfBondsAtGov   = government.govLOB.map(_.amountOfBondsInStack).sum
    val bankIDs              = bankList.filter(_.active).map{
        bank =>
          val LOB_IDs           = bank.listOfBonds.map{ case(id, fraction) => id }.toBuffer
          val OMO_IDs           = bank.bondsPledgedAsCollateralForOMO.map{ case(id, fraction) => id }.toBuffer
          val OSLF_IDs          = bank.bondsPledgedAsCollateralForOSLF.map{ case(id, fraction) => id }.toBuffer
          val IDL_IDs           = bank.bondsPledgedAsCollateralForIDL.map{ case(id, fraction) => id }.toBuffer
          ArrayBuffer[Long]() ++= LOB_IDs ++= OMO_IDs ++= OSLF_IDs ++= IDL_IDs
      }
    val bdIDs = BrokerDealerList.filter(_.active).map{
      bd =>
        val LOB_IDs           = bd.listOfBonds.map{ case(id, fraction) => id }.toBuffer
        val Repo_IDs          = bd.bondsPledgedAsCollateralForRepo.map{ case(id, fraction) => id }.toBuffer
```

```
        ArrayBuffer[Long]() ++= LOB_IDs ++= Repo_IDs
      }
    val cbIDs                = centralBank.listOfBonds.map{ case(id, fraction) => id }.toBuffer
    val listOfAllIDs         = ArrayBuffer[Long]() ++= bankIDs.flatten ++= cbIDs ++= bdIDs.flatten
    val listOfDistinctIDs    = listOfAllIDs.distinct.map(id => government.findStackOfBondsByID(id)).map(_.amountOfBondsInStack).sum
    println(s"Gov (in $t): $amountOfBondsAtGov")
    println(s"Bank (in $t): $bankIDs")
    println(s"BD (in $t): $bdIDs")
    println(s"CB (in $t): $cbIDs")
    require(amountOfBondsAtGov == listOfDistinctIDs, s"bonds@gov ($amountOfBondsAtGov) != bonds@banks/CB/BD ($listOfDistinctIDs)")
  }





  def testAmountOfExistingIDs (t:Int, cause:String) = {
    val govIDs       = government.govLOB.size
    val govListOfIDs = government.govLOB.map { repo => repo.id }
    val bankIDs = bankList.filter(_.active).map{
        bank =>
          val LOB_IDs  = bank.listOfBonds.keys
          val OMO_IDs  = bank.bondsPledgedAsCollateralForOMO.keys
          val OSLF_IDs = bank.bondsPledgedAsCollateralForOSLF.keys
          val IDL_IDs  = bank.bondsPledgedAsCollateralForIDL.keys
          ArrayBuffer[Long]() ++= LOB_IDs ++= OMO_IDs ++= OSLF_IDs ++= IDL_IDs
      }
    val BDIDs = BrokerDealerList.filter(_.active).map {
      BD =>
        val LOB_IDs  = BD.listOfBonds.keys
        val repo_IDs = BD.bondsPledgedAsCollateralForRepo.keys
        ArrayBuffer[Long]() ++= LOB_IDs ++= repo_IDs
      }
    val cbIDs             = centralBank.listOfBonds.keys
    val listOfAllIDs      = ArrayBuffer[Long]() ++= bankIDs.flatten ++= cbIDs ++= BDIDs.flatten
    val listOfDistinctIDs = listOfAllIDs.distinct.size
    val missingIDs = govListOfIDs --= listOfAllIDs
    require(
      govIDs == listOfDistinctIDs,
      s"t:$t --> $cause --> amount of IDs does not correspond to amount of outstanding IDs. Gov: $govIDs / Bank/CB/BD (distinct): $listOfDistinctIDs; ${missingIDs}")
  }



  def checkBankRetailDeposits (t:Int, cause:String) = bankList.foreach{
    bank =>
      val firmDep = bank.businessClients.map(_.bankDeposits.last).sum
      val hhDep   =   bank.retailClients.map(_.bankDeposits.last).sum
      val MMMFDep =     bank.MMMFClients.map(_.bankDeposits.last).sum
      val BDDep   =       bank.BDClients.map(_.bankDeposits.last).sum
      val sumOfDeps = rounded( firmDep + hhDep + MMMFDep + BDDep )
      require(sumOfDeps < bank.retailDeposits.last + 50 && sumOfDeps > bank.retailDeposits.last - 50,
          s"checkBankRetailDeposits [SumOfBanks] of $bank [seed $seed/t=$t] failed after $cause:\n firm: $firmDep + hh: $hhDep + MMMF: $MMMFDep + BD: $BDDep ($sumOfDeps) == ${bank.retailDeposits.last} --> diff: ${bank.retailDeposits.last - sumOfDeps}")
```

```
    }




    def checkGovDeposits (cause:String, t:Int, seed:Int = seed) =
      require(
        rounded(bankList.filter(_.active).map(_.govDeposits.last).sum) < rounded(government.bankDeposits.last) + 10 &&
        rounded(bankList.filter(_.active).map(_.govDeposits.last).sum) > rounded(government.bankDeposits.last) - 10,
        s"Wrong gDeposits after $cause in t=$t [seed $seed] --> Diff: ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum) - rounded(government.bankDeposits.last)}; bs: ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / g: ${government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} "
      )


    def p (t:Int, cause:String, marker:Boolean = false) = {
      BrokerDealerList.filter(_.active).foreach{bd => require(bd.businessLoans.last >= 0, s"businessLoans of $bd are negative after $cause [t=$t|seed $seed] (${bd.businessLoans.last})")}
      println(s"$cause in t=$t")
      MMMFList.filter(_.active).foreach {
        mmmf =>
          require(
              mmmf.interestOnDebt.last < Seq(mmmf.claimsFromRepos.last, mmmf.bankDeposits.last, bonds.last, mmmf.interestReceivables.last).sum,
              s"intOnDebt of $mmmf is very large after $cause [t=$t|seed $seed]: ${mmmf.interestOnDebt.last} vs. TA of ${Seq(mmmf.claimsFromRepos.last, mmmf.bankDeposits.last, bonds.last, mmmf.interestReceivables.last).sum}"
          )
      }
      BrokerDealerList.foreach {
        bd =>
          bd.listOfBonds.foreach {
            case(id, fraction) =>
              sim.government.findStackOfBondsByID(id).bond.ticksOfCouponPayment.filter{_ > t }.foreach {
                tick =>
                  require(sim.government.coupon2PayBD.contains(tick), s"coupon2PayBD does not contain couponPayment to $bd in tick $tick [t=$t] $cause: $id -> $fraction")
                  }
              }
          }
      BrokerDealerList.foreach { bd => bd.listOfBonds.values.foreach{ fraction => require(roundTo5Digits(fraction) > 0.0, s"fraction of SoB of $bd [seed $seed | t=$t] is almost zero $cause: $fraction") } }
      checkGovDeposits(cause, t)
      if(marker){
        val faultTolerance = 5
        bankList.foreach { bank => if(bank.active) require(bank.retailDeposits.last >= -faultTolerance, s"$bank [seed $seed] has negative rd in t=$t: ${bank.retailDeposits.last} after $cause") }
          hhList.foreach {   hh =>                 require(  hh.bankDeposits.last   >= -faultTolerance, s"$hh   [seed $seed] has negative bd in t=$t:     ${hh.bankDeposits.last} after $cause") }
        firmList.foreach { firm =>                 require(firm.bankDeposits.last   >= -faultTolerance, s"$firm [seed $seed] has negative bd in t=$t:   ${firm.bankDeposits.last} after $cause") }
        checkBankRetailDeposits(t, cause)
        if(government.govLOB.size > 5000){
          sys.error(s"Current code position of seed $seed in t=$t: after $cause [nob: ${government.govLOB.size}]")
                MMMFList.foreach(mmmf => if(mmmf active) require(mmmf.bankDeposits.last >= -faultTolerance, s"$mmmf has negative bankDeposits before $cause in t=$t [seed $seed]: ${mmmf.bankDeposits.last}\n ${mmmf.printBSP}"))
          BrokerDealerList.foreach(  bd => if(  bd active) require(  bd.bankDeposits.last >= -faultTolerance, s"  $bd has negative bankDeposits before $cause in t=$t [seed
```

```
$seed]: ${bd.bankDeposits.last}\n ${bd.printBSP}"))
          }
        }
      }







      def testAmountOfOutstandingFirmDebt (t:Int, when:String, cause:String, repaidInCurrentTick:Boolean) = {
        repaidInCurrentTick match {
          case false => testBeforeRepay
          case true  => testAfterRepay
        }

        def testBeforeRepay = {
            firmList.foreach{
              firm =>
                if(firm.houseBank.listOfDebtors.contains(firm)) require(
                    SEc(rounded(firm.houseBank.listOfDebtors(firm).map(loan => loan.principalPayments.filter(_._1 >= t).values.sum).sum), firm.debtCapital.last,    1),
                    s"$firm debtCapital (${firm.debtCapital.last}) in t = $t $when $cause is not equal to its outstanding interest payments (${rounded(firm.houseBank.listOfDebtors(firm).map(loan => loan.principalPayments.filter(_._1 >= t).values.sum).sum)}) "
                )
                if(firm.houseBank.listOfDebtors.contains(firm)) require(
                    SEc(rounded(firm.houseBank.listOfDebtors(firm).map(loan =>  loan.interestPayments.filter(_._1 >= t).values.sum).sum), firm.interestOnDebt.last, 1),
                    s"$firm interestOnDebt (${firm.interestOnDebt.last}) in t = $t $when $cause is not equal to its outstanding principal payments (${rounded(firm.houseBank.listOfDebtors(firm).map(loan =>  loan.interestPayments.filter(_._1 >= t).values.sum).sum)}) "
                    )
            }
        }

        def testAfterRepay = {
          firmList.foreach{
            firm =>
              if(firm.houseBank.listOfDebtors.contains(firm)) require(
                  SEc(rounded(firm.houseBank.listOfDebtors(firm).map(loan => loan.principalPayments.filter(_._1 > t).values.sum).sum), firm.debtCapital.last,    1),
                  s"$firm debtCapital (${firm.debtCapital.last}) in t = $t $when $cause is not equal to its outstanding interest payments (${rounded(firm.houseBank.listOfDebtors(firm).map(loan => loan.principalPayments.filter(_._1 > t).values.sum).sum)}) "
                  )
              if(firm.houseBank.listOfDebtors.contains(firm)) require(
                  SEc(rounded(firm.houseBank.listOfDebtors(firm).map(loan =>  loan.interestPayments.filter(_._1 > t).values.sum).sum), firm.interestOnDebt.last, 1),
                  s"$firm interestOnDebt (${firm.interestOnDebt.last}) in t = $t $when $cause is not equal to its outstanding principal payments (${rounded(firm.houseBank.listOfDebtors(firm).map(loan =>  loan.interestPayments.filter(_._1 > t).values.sum).sum)}) "
                  )
            }
        }

      }// end of method


}/*    -----    End of class: Simulation    -----    */
```

A.1.3 Traits

Traits.scala

```scala
/**
 * Routines that are used by multiple classes of agents
 */

package monEcon

import monEcon._
import monEcon.financialSector._
import monEcon.publicSector._
import monEcon.realSector._
import monEcon.Markets._

import collection.mutable.Map
import collection.mutable.ArrayBuffer

import scalax.io._              // scalax
import scalax.io.Codec
import scalax.io.JavaConverters._
import scala.io._
import scala.util.{Try,Success,Failure}
import scala.sys.process._
import math._
import util.Random

import java.io._
import java.util.Date
import java.util.Calendar
import java.text.SimpleDateFormat

import org.apache.commons.io.FileUtils
import org.apache.commons.io.filefilter.WildcardFileFilter
import org.apache.commons.math3._
import org.apache.commons.math3.stat.regression.SimpleRegression





/**
 * @author Sebastian Krug
 *
 */

// -------------------- Trait for input/output ----------------------------------------------------------------------------------------------------------
trait IO {

  val zeitstempel2      = new Date()
  val simpleDateFormat2 = new SimpleDateFormat("ddMMyyyy")
  val date2             = simpleDateFormat2.format(zeitstempel2)


  def using[A <: {def close(): Unit}, B](param: A)(f: A => B): B =
    try {
      f(param)
```

```scala
  } finally {
    param.close()
    }


//  Method to evaluate the right fileName for outputs
def SetFileName (folder:String, fileName:String, objectType:String, seed:Int):String = {
  createDirectory(s"simData/$objectType")     // create sub-directories according to agents
  s"simData/$objectType/${fileName}.csv"      // create csv-files       according to data
}


def writeToFile (folder:String, fileName:String, data:Seq[Any], objectType:String, seed:Int) = {
  using (new FileWriter(SetFileName(folder, fileName, objectType, seed))) {
    fileWriter =>
      fileWriter.write( data.mkString( ", " ) )
  }
}


def saveSimulationData (folder:String, fileName:String, objectType:String, textData:Seq[_], agent:Agent, seed:Int) = {
  using (new FileWriter(SetFileName(folder, fileName, objectType, seed), true)) {
    fileWriter =>
      using (new PrintWriter(fileWriter)) {printWriter => printWriter.println( textData.mkString( ", " ) )}
  }
}


def saveTickData (folder:String, fileName:String, tick:Int, objectType:String, textData:Seq[_], agent:Agent, seed:Int) = {
  using (new FileWriter(SetFileName(folder, fileName, objectType, seed), true)) {
    fileWriter =>
      using (new PrintWriter(fileWriter)) {
        printWriter =>
          printWriter.println( textData.mkString( "{", ", ", s" (* $agent of tick $tick *)}" ) )
      }
  }
}


def savePerformanceData (folder:String, fileName:String, objectType:String, nanoTime:Long, seed:Int) = {
  using (new FileWriter(SetFileName(folder, fileName, objectType, seed), true)) {
    fileWriter =>
      using (new PrintWriter(fileWriter)) {printWriter => printWriter.println(nanoTime)}
  }
}


def formatMap (map:collection.immutable.Map[_, _], mapType:String, folder:String, fileName:String, seed:Int) = {
  mapType match {

    case "Agent_map_Array" =>
      val regex = "Bank\\(\\d\\) -> ArrayBuffer\\(\\d\\.\\d, \\d\\.\\d\\)".r
      val data  = map.mkString
      using (new FileWriter( s"simData/$folder/${fileName}_$seed.csv" )) {fileWriter => fileWriter.write( data )}
```

```scala

    case "Agent_map_Map_Agent_map_Array" =>
      val data = map.mkString(", ").replace(")), ", "))\n").replace("ArrayBuffer", "").replace("Map", "").replace("->", "").replaceAll("[()]", "")
      using (new FileWriter( s"simData/$folder/${fileName}_$seed.csv" )) {fileWriter => fileWriter.write( data )}

    case _ => sys.error("format map")

  }
}


def copyFile (fileName: String) = {
  val fis = new FileInputStream(fileName)
  val fos = new FileOutputStream(fileName.split('.')(0)+"_copy."+fileName.split('.')(1))
  var byte = fis.read()
  while(byte>=0){
    fos.write(byte)
    byte = fis.read()
  }
  fis.close()
  fos.close()
}


def deleteFile (fileName: String) = {
  val file = new File("output/"+fileName+"_"+this+".m").delete
}



def deleteOldSimulationOutput (directoryName:String) {
  val dir = new File(directoryName)
  if(dir.exists()){
    println(directoryName + " with old simulation data detected and deleted.")
    try{
      FileUtils.deleteDirectory(new File(directoryName))
    } catch {
      case ioe:IOException => ioe.printStackTrace()           // log the exception here
                              throw ioe
      }
  } else println("No old simultaion data detected.")
}




// Method to create a directory
def createDirectory (path:String) = {
  val dir                = new File(path)
  val successful:Boolean = dir.mkdir()
  if(successful){
    println(s"Directory $path was created successfully.  ")
  }
}
```

```scala
  def readCSV (file:String) = {
    val bufferedSource = io.Source.fromFile("GDP_HP.csv")
    for (line <- bufferedSource.getLines) {
      val cols = line.split(",").map(_.trim)
    }
  }


} // -------------------- End of Trait IO ------------------------------------------------------------------------------------------------


trait SaveResults extends IO {
  val sim             :Simulation
  val laborMarket     :LaborMarket
  val goodsMarket     :GoodsMarket
  val centralBank     :CentralBank
  val interbankMarket :InterbankMarket
  val government      :Government
  val supervisor      :Supervisor


  /*
   * ---------- save LISTBUFFER-DATA -> is mutated during the period (e.g. balance sheet positions etc.)  ----------
   * addTickValue creates opening balance sheet in every tick (incl. the first with all zeros in t=1)
   *
   * */
  def addTickValue (t:Int) = {

```

```scala
  def addMapValue (map:Map[String, ArrayBuffer[Double]]) = {
    map.foreach{
      case (nameOfBSP, data) =>
        if(data.isEmpty) data += 0.0 else if(data.size < t) data += data.last else if(data.size > t) sys.error(nameOfBSP + "has too many values in the Buffer.") }
  }

  def addValue (buffer:ArrayBuffer[Double]) = if(buffer.isEmpty) buffer += 0.0 else if(buffer.size < t) buffer += 0.0 else if(buffer.size > t) sys.error(buffer + " has too many values in the Buffer.")

  sim.bankList.foreach{
    bank =>
      addMapValue(bank.bankBSP)
      addValue(bank.COGS)
      addValue(bank.earnings)
      addValue(bank.loanLosses)
      if((t-1) % 48 == 0) bank.insolvencies += 0
  }

  sim.MMMFList.foreach{
    MMMF =>
      addMapValue(MMMF.MMMFBSP)
      addValue(MMMF.earnings)
      if((t-1) % 48 == 0) MMMF.insolvencies += 0
  }

  sim.BrokerDealerList.foreach{
    BD =>
      addMapValue(BD.brokerDealerBSP)
      addValue(BD.loanLosses)
      addValue(BD.earnings)
      if((t-1) % 48 == 0) BD.insolvencies += 0
  }

  sim.firmList.foreach{
    firm =>
      addMapValue(firm.firmBSP)
      addValue( firm.sales)
      if((t-1) % 48 == 0) firm.insolvencies += 0
  }








  sim.hhList.foreach{
    hh =>
      addMapValue(hh.hhBSP)
      if((t-1) % 4  == 0) hh.amount2Spend          += 0.0
      if((t-1) % 48 == 0) hh.laborSkillFactor      += hh.laborSkillFactor.last
                          hh.riskAversionParameter += hh.riskAversionParameter.last
      if((t-1) % 48 == 0) hh.interestOnDeposits    += 0.0
```

```
      if((t-1) % 48 == 0) hh.dividendsReceived     += 0.0
  }

  addMapValue(sim.centralBank.centralBankBSP)
  addValue(   sim.centralBank.credit2privateSector)

  if(t > 1){
    sim.reserveFlows.foreach{ case (from, mapOfTos) => mapOfTos.foreach{ case (to, buffer) => buffer += 0.0}}
    sim.IBMloanFlows.foreach{ case (from, mapOfTos) => mapOfTos.foreach{ case (to, buffer) => buffer += 0.0}}
    sim.IDLflows.foreach(pair => pair._2 += 0.0)
  }

  addValue(sim.investmentSBsector)
  addValue(sim.withdrawFromSBsector)
  addValue(sim.sizeTBsector)
  addValue(sim.sizeSBsector)
  addValue(sim.equityTBsector)
  addValue(sim.equitySBsector)
  addValue(sim.investedFundsSBsector)
  addValue(sim.neededLiquidityFirms)
  addValue(sim.offeredFundsMMMF)
  addValue(sim.creditGrantedByBD)
  addValue(sim.creditGrantedByTB)
  addValue(sim.government.GDP)
  addValue(sim.government.VATrevenue)
  addValue(sim.government.corporateTaxRevenue)
  addValue(sim.government.capitalGainsTaxRevenue)
  addValue(sim.government.incomeTaxRevenue)
  addValue(sim.government.govSpending)
  addValue(sim.government.M0)
  addValue(sim.government.M1)
  addValue(sim.government.M3)
  addValue(sim.government.productionOfTick)
  addValue(sim.government.lossFromBailOut)
  addMapValue(sim.government.governmentBSP)
  sim.publicConfidenceLevel += sim.publicConfidenceLevel.last
}




def saveEndOfTickData (t:Int) = {

  def saveAndClear (map:Map[String, _], t:Int, objectType:String, agent:Agent) = {
    map.foreach{
      case (name, data) =>
        data match {
          case data:Map[_, _]      => saveTickData(sim.toString, name, t, objectType, data.toSeq, agent, sim.seed)
          case data:ArrayBuffer[_] => saveTickData(sim.toString, name, t, objectType, data.asInstanceOf[Seq[_]], agent, sim.seed)
          case _                   => println("Cannot clear the data for saving tick data. Add another case to saveAndClear")
        }
    }
  }
}
```

```
def saveEndOfSimulationData = {

  def save(map:Map[String, _], objectType:String, agent:Agent) = {
    map.foreach{
      case (name, data) =>
        data match {
          case data:Map[_, _]      => saveSimulationData(sim.toString, name, objectType, data.toSeq,                agent, sim.seed)
          case data:ArrayBuffer[_] => saveSimulationData(sim.toString, name, objectType, data.asInstanceOf[Seq[_]], agent, sim.seed)
          case data:List[_]        => saveSimulationData(sim.toString, name, objectType, data.asInstanceOf[Seq[_]], agent, sim.seed)
          case data:Int            => saveSimulationData(sim.toString, name, objectType, Seq[Int](data),            agent, sim.seed)
          case data:Double         => saveSimulationData(sim.toString, name, objectType, Seq[Double](data),         agent, sim.seed)
          case _                   => sys.error("save-Method in saveEndOfSimulationData does not contain enough cases! Add a case for the missing data structure!")
        }
    }
  }


  saveSimulationData(sim.toString, "numberOfActiveFirms",   "simulation", sim.numberOfActiveFirms,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "numberOfActiveBanks",   "simulation", sim.numberOfActiveBanks,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "numberOfActiveMMMF  ",  "simulation", sim.numberOfActiveMMMF,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "numberOfActiveBD",      "simulation", sim.numberOfActiveBD,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "numberOfBailouts",      "simulation", sim.numberOfBailOuts,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "expectedInflation",     "simulation", sim.expPi,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "expRealIntRate",        "simulation", sim.expRealIntRate,
sim.centralBank, sim.seed)
  saveSimulationData(sim.toString, "saveTickTime",          "simulation", sim.saveTickTime,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "investmentSBsector",    "simulation", sim.investmentSBsector,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "withdrawSBsector",      "simulation", sim.withdrawFromSBsector,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "sizeTBsector",          "simulation", sim.sizeTBsector,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "sizeSBsector",          "simulation", sim.sizeSBsector,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "equityTBsector",        "simulation", sim.equityTBsector,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "equitySBsector",        "simulation", sim.equitySBsector,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "investedFundsSBsector", "simulation", sim.investedFundsSBsector,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "publicConfidenceLevel", "simulation", sim.publicConfidenceLevel,
sim.government,  sim.seed)
  saveSimulationData(sim.toString, "neededLiquidityFirms",  "simulation", sim.neededLiquidityFirms,
```

```
sim.government,  sim.seed)
    saveSimulationData(sim.toString, "offeredFundsMMMF",      "simulation", sim.offeredFundsMMMF,
sim.government,  sim.seed)
    saveSimulationData(sim.toString, "creditGrantedByBD",     "simulation", sim.creditGrantedByBD,
sim.government,  sim.seed)
    saveSimulationData(sim.toString, "creditGrantedByTB",     "simulation", sim.creditGrantedByTB,
sim.government,  sim.seed)
    saveSimulationData(sim.toString, "offeredLiquidityBD",    "simulation", sim.offeredLiquidityBD,
sim.government,  sim.seed)
    saveSimulationData(sim.toString, "liquidityGap",          "simulation", sim.liquidityGap,
sim.government,  sim.seed)
    saveSimulationData(sim.toString, "reserveFlows",          "simulation", sim.reserveFlows.toSeq,
sim.government,  sim.seed)
    saveSimulationData(sim.toString, "IDLflows",              "simulation", sim.IDLflows.toSeq,
sim.government,  sim.seed)
    saveSimulationData(sim.toString, "reserveFlows",          "simulation", formatMap(sim.reserveFlows, "Agent_map_Map_Agent_map_Array", "CentralBank", "reserveFlows"),
sim.government,  sim.seed)
    saveSimulationData(sim.toString, "IDLflows",              "simulation", formatMap(sim.IDLflows,     "Agent_map_Array",               "CentralBank", "IDLflows"),
sim.government,  sim.seed)

    formatMap(sim.reserveFlows, "Agent_map_Map_Agent_map_Array", "simulation", "reserveFlows", sim.seed)
    formatMap(sim.IBMloanFlows, "Agent_map_Map_Agent_map_Array", "simulation", "IBMloanFlows", sim.seed)
    formatMap(sim.IDLflows,     "Agent_map_Array",               "simulation", "IDLflows",     sim.seed)


    sim.bankList.foreach{
      bank =>
        save(bank.bankEndOfSimulationData, "Bank", bank)
        save(bank.bankBSP,                 "Bank", bank)
    }

    sim.MMMFList.foreach{
      MMMF =>
        save(MMMF.MMMFEndOfSimulationData, "MMMF", MMMF)
        save(MMMF.MMMFBSP,                 "MMMF", MMMF)
    }





    sim.BrokerDealerList.foreach{
      brokerDealer =>
        save(brokerDealer.brokerDealerEndOfSimulationData, "BrokerDealer", brokerDealer)
        save(brokerDealer.brokerDealerBSP,                 "BrokerDealer", brokerDealer)
    }

    sim.hhList.foreach{
      hh =>
        save(hh.hhEndOfSimulationData, "HH", hh)
        save(hh.hhBSP,                 "HH", hh)
    }

    sim.firmList.foreach{
```

```scala
      firm =>
        save(firm.firmEndOfSimulationData, "Firm", firm)
        save(firm.firmBSP,                 "Firm", firm)
    }

    save(sim.supervisor.supervisorEndOfSimulationData,   "Supervisor",      sim.supervisor)        // Supervisor
    save(sim.centralBank.centralBankEndOfSimulationData, "CentralBank",     sim.centralBank)       // Central Bank
    save(                sim.centralBank.centralBankBSP, "CentralBank",     sim.centralBank)       // Central Bank
    save(  sim.government.governmentEndOfSimulationData, "Government",      sim.government)        // Government
    save(                  sim.government.governmentBSP, "Government",      sim.government)        // Government
    save(    sim.interbankMarket.IBMEndOfSimulationData, "InterbankMarket", sim.interbankMarket)   // IBM
    save(sim.goodsMarket.goodsMarketEndOfSimulationData, "GoodsMarket",     sim.goodsMarket)       // goodsMarket

  }
} // -------------------- End Trait saveResults ----------------------------------------------------------------------------------------------------



































/*  --------------------  Begin Trait for Search & Matching -------------------------------------------------------------------------------------  */
trait searchAndMatching extends round {
```

```scala

def determineFirmSizeDistribution(simulation:Simulation, tick:Int, list:Seq[Corporation]) = {
  // determine current relative firm size
  val absoluteFirmSize:Map[Corporation, Double] = Map()
  if(tick < 2){
        list.foreach{corp => corp match {
          case corp:Firm => absoluteFirmSize += corp -> corp.bankDeposits.last
          case corp:Bank => absoluteFirmSize += corp -> corp.cash.last
          }
        }
      } else {
          list.foreach{corp => corp match {
            case corp:Firm => absoluteFirmSize += corp -> corp.totalAssets.last
            case corp:Bank => absoluteFirmSize += corp -> corp.totalAssets.last
            }
          }
        }
  println("absolute firm sizes: " + absoluteFirmSize)
  val sectorSize = absoluteFirmSize.values.sum
  val relativeFirmSize:Map[Corporation, Double] = Map()
  absoluteFirmSize.keys.foreach(corp => relativeFirmSize += corp -> absoluteFirmSize(corp)/sectorSize)
  if(rounded(relativeFirmSize.values.sum) != 1) sys.error("Sum of relative Firm sizes must be 1: " + relativeFirmSize.values.sum)
  println("relative firm sizes: " + relativeFirmSize)

  // let hh choose considered firms for random search
  val firmDistribution:ArrayBuffer[Corporation] = ArrayBuffer()
  relativeFirmSize.keys.foreach(corp => firmDistribution ++= List.fill( math.rint(relativeFirmSize(corp)*100).toInt )(corp) )
  println("Distribution of Firms according to their current size (100): " + firmDistribution.size)
  val consideredFirms = simulation.random.shuffle(firmDistribution).take(50).toSet.toList
  println("Considered Firms: " + consideredFirms)
  consideredFirms
}

def randomSubsampleOfFirms(sim:Simulation, tick:Int, fraction:Double) = {
  sim.random.shuffle(sim.firmList).take( (sim.numberOfFirms * fraction).toInt )
}


} // -------------------- End Trait for Search & Matching ----------------------------------------------------------------------------------------------
```

```scala
/*  --------------------  Begin Trait for OLS Regression  ---------------------------------------------------------------------------------------  */
trait simpleRegression {

  def addRegData (reg:SimpleRegression, list1:Seq[Double], list2:Seq[Double]) = {
    require(list1.length == list2.length, "reg data have unequal size")     // test whether both lists have the same length
    val lista:Seq[Seq[Double]]     = combineSeq(list1, list2)               // List(a,b) & List(c,d) => List(List(a,c), List(b,d))
    val listb:Seq[Array[Double]]   = lista map(n => n.toArray)              // List[List[Double]]    => List[Array[Double]]
    val listc:Array[Array[Double]] = listb.toArray                          // List[Array[Double]]   => Array[Array[Double]]
    reg.addData(listc)
  }

  def filterRegressionData (a:Seq[Double], b:Seq[Double]) = a zip b filter(tuple => tuple._1 != 0.0 && tuple._2 != 0.0) unzip

  def doRegression (reg:SimpleRegression, list1:Seq[Double], list2:Seq[Double], tickOfPrediction:Int) = {
    reg.addData(list1.last, list2.last)
    println(s"${reg.toString()} = $list1, $list2")
    val regResult = reg.predict(tickOfPrediction)
    println(s"Regression using ${reg.getN()} 2D-data points results: $regResult")
    regResult
  }

  def tupleToList[T](t: (T,T)): Seq[T] = Seq(t._1, t._2)


  def combineSeq (seq1:Seq[Double], seq2:Seq[Double]) = {
    val seq = seq1 zip seq2
    seq map(n => tupleToList(n))
  }

} // -------------------- End Trait for OLS Regression -----------------------------------------------------------------------------------------------
```

```scala
/*  --------------------  Begin Trait for firm entry  --------------------------------------------------------------------------------  */
trait entryExit extends round {
  val numberOfBanks:Int
  val numberOfFirms:Int
  val numberOfHH:Int
  val random:Random
  val tradBanks:Boolean
  val goodsMarket:GoodsMarket
  val laborMarket:LaborMarket
  val interbankMarket:InterbankMarket
  val government:Government
  val centralBank:CentralBank
  val fractionOfDebtBank:Double
  val sim:Simulation

  val arge:ARGE
  var hhList:Seq[HH]
  var bankList:Seq[Bank]
  var firmList:Seq[Firm]

  var bankCounter = numberOfBanks - 1
  var firmCounter = numberOfFirms - 1


  def removeBank(bank:Bank) = bankList = bankList diff Seq(bank)
  def removeFirm(firm:Firm) = firmList = firmList diff Seq(firm)


  def randomProbability = {
    var p = 0.00
    do{ p = rounded( random.nextFloat.toDouble ) }while(p > 0.5)
       p
  }

```

```scala
  def randomGaussian4truncatedND(mean:Double, lowerBoundary:Double):Double = {
    var result = 0.0
    do{
      result = rounded(random.nextGaussian + mean)
    }while(result < lowerBoundary)
    result
  }

} // ---------------------------------------------------------------------------------------------------------------------------------------

trait codeProfiling extends IO {

  def time[R](block: => R, method:String, sim:Simulation): R = {
    if(sim.profilePerformance){
      val t0          = System.nanoTime()
      val result      = block                         // call-by-name
      val t1          = System.nanoTime()
      val elapsedTime = t1 - t0
      savePerformanceData(sim.toString, method, "sim_performance", elapsedTime, sim.seed)
      println(method + " in t = " + t + " needed " + elapsedTime + " ns")
      result
    } else {
      block
    }
  }

}
```

```
trait random {
  val random:Random

  def randomDoubleBetween (lowerBound:Double, upperBound:Double) = {
    val rnd = new scala.util.Random
  }


}// end trait random




















trait round extends codeProfiling {

  def rounded        (number:Double) = math.rint(number *        100) /        100    // round to 2 digits
  def roundTo3Digits (number:Double) = math.rint(number *       1000) /       1000    // round to 3 digits
  def roundTo4Digits (number:Double) = math.rint(number *      10000) /      10000    // round to 4 digits
  def roundTo5Digits (number:Double) = math.rint(number *     100000) /     100000    // round to 5 digits
  def roundTo9Digits (number:Double) = math.rint(number * 1000000000) / 1000000000


  def roundUpTo1000 (amount:Double) = if(amount %  1000 != 0) amount + ( 1000 - (amount %  1000)) else amount
  def roundUpTo5k   (amount:Double) = if(amount %  5000 != 0) amount + ( 5000 - (amount %  5000)) else amount
  def roundUpTo10k  (amount:Double) = if(amount % 10000 != 0) amount + (10000 - (amount % 10000)) else amount
```

```
def roundUpXk     (amount:Double, k:Double) = if(amount % k != 0) amount + (k - (amount % k)) else amount
def roundDownXk   (amount:Double, k:Double) = if(amount % k != 0) amount - (amount % k) else amount

// square error
def SE (a:Double, b:Double) = {
  val c = math.max(10, b * 0.000001)
  math.pow(rounded(a) - rounded(b), 2) <= c
}

def SEc (a:Double, b:Double, c:Double) = math.pow(rounded(a) - rounded(b), 2) <= c

def squareDeviation (a:Double, b:Double) = math.pow(a-b, 2)

val govDepositsDev    = ArrayBuffer[Double](0.0)
val retailDepositsDev = ArrayBuffer[Double](0.0)
val reserveAccountDev = ArrayBuffer[Double](0.0)

def checkAndAdjust (checkableAccount:String, when:String, cause:String, a:Double, b:ArrayBuffer[Double], t:Int) {
  val c   = math.max(25, b.last * 0.000001)
  val dev = rounded(a) - rounded(b.last)
  if(math.pow(dev, 2) > 0){
    if(math.pow(dev, 2) <= c){
      dev match {
        case dev:Double if dev > 0 => b(b.size-1) = rounded(b.last +   dev)
        case dev:Double if dev < 0 => b(b.size-1) = rounded(b.last - (-dev))
      }
      checkableAccount match {
        case "govDeposits"       => govDepositsDev(0)    += dev
        case "checkBankDeposits" => retailDepositsDev(0) += dev
        case "reserveAccounts"   => reserveAccountDev(0) += dev
        case _                   => sys.error("")
      }
    } else sys.error(s"$checkableAccount are not consistent $when $cause in $t: No adjustment conducted because deviation is too large -> ${rounded(a)} - ${b.last} = $
{rounded(a) - rounded(b.last)}.")
  }
  require( SE(a, b.last), s"$checkableAccount failed $when $cause: ${rounded(a)} - ${b.last} = ${rounded(a) - rounded(b.last)} after adjustment" )
}

}
```

Traits.scala

```scala
trait accountManagement extends round with IOU {

  def deposit (account:ArrayBuffer[Double], amount:Double, tick:Int, sim:Simulation) = {
    if(sim.test){
      require(rounded(amount) >= 0, s"You cannot deposit a negative value at an account: $amount")
      require(account.size == tick, s"You try to update the wrong number, length: ${account.size}:${account} / $tick")
    }
    account(account.size-1) = rounded(account.last + amount)
  }

  def withdraw (account:ArrayBuffer[Double], amount:Double, tick:Int, sim:Simulation) = {
    if(sim.test){
      require(rounded(amount) >= 0, s"The amount to withdraw cannot be negative: $amount")
      require(account.size == tick, "You try to update the wrong number, length: " + account.size + " / " + tick)
      assert(rounded(amount) <= account.last + math.max(1, amount * 0.000001), "You don't have enough money on your account! You cannot deduct " + amount + " from " + account.last)
    }
    account(account.size-1) = rounded(account.last - amount)
  }


  def average (list:Seq[Double]) = {
    require(list.nonEmpty, "You cannot take the average of an empty list!")
    list.foldLeft(0.0)(_+_) / list.foldLeft(0.0)((r,c) => r+1)
  }

  def stdDev (data:ArrayBuffer[Double]):Double = {
     val sum:Double = if(data.length >= 2){
       val mean = average(data)
       val factor:Double = 1.0/(data.length.toDouble-1)
       factor * data.foldLeft(0.0){ (acc,x) => acc + math.pow(x - mean,2) }
     } else 0.0
     math.sqrt(sum)
   }


  def sumOfPastPeriods( list:ArrayBuffer[Double], sim:Simulation):Double = if(list.length < sim.updateFrequency) list.sum else list.slice(list.length - sim.updateFrequency, list.length).sum
  def sumOfNPastPeriods(list:ArrayBuffer[Double], periods:Int   ):Double = if(list.length < periods)             list.sum else list.slice(list.length - periods, list.length).sum


  def map2ListOfRelationships (mapOfMaps:Map[Bank, Map[Bank, Double]], listOfRelationships:ArrayBuffer[List[String]]) {
    val list:ArrayBuffer[String] = ArrayBuffer()
    mapOfMaps.foreach{ case (creditor, mapOfCreditors) => mapOfCreditors.foreach{ case (debtor, amount) => list += s"$creditor -> ($debtor, $amount)" }}
    listOfRelationships += list.toList
  }



  /**
   *
   *    */
  def IBMrelationship(debtor:Bank, claimholder:Bank, amount:Double, sim:Simulation) = {
```

```
    if(sim.interbankMarket.grossInterbankLiabilitiesOfCurrentTick.contains(debtor)){
      if(sim.interbankMarket.grossInterbankLiabilitiesOfCurrentTick(debtor).contains(claimholder)){
        sim.interbankMarket.grossInterbankLiabilitiesOfCurrentTick(debtor) += claimholder -> ( rounded(sim.interbankMarket.grossInterbankLiabilitiesOfCurrentTick(debtor)(claimholder) + amount) )
      } else sim.interbankMarket.grossInterbankLiabilitiesOfCurrentTick(debtor) += claimholder -> rounded(amount)
    } else sim.interbankMarket.grossInterbankLiabilitiesOfCurrentTick += debtor -> Map( claimholder -> rounded(amount) )
    if(sim.interbankMarket.grossInterbankLoansOfCurrentTick.contains(claimholder)){
      if(sim.interbankMarket.grossInterbankLoansOfCurrentTick(claimholder).contains(debtor)){
        sim.interbankMarket.grossInterbankLoansOfCurrentTick(claimholder) += debtor -> ( rounded(sim.interbankMarket.grossInterbankLoansOfCurrentTick(claimholder)(debtor) + amount) )
      } else sim.interbankMarket.grossInterbankLoansOfCurrentTick(claimholder) += debtor -> rounded(amount)
    } else sim.interbankMarket.grossInterbankLoansOfCurrentTick += claimholder -> Map( debtor -> rounded(amount) )
  }



  /**
   *   Funds are tranfered either
   *   - internal -> one of the counterparty is a housebank or both counterparties are customer of the same bank
   *   - external -> transfer includes 2 distinct banks
   *    */
  def transferMoney (from:Agent, to:Agent, amount:Double, cause:String, sim:Simulation, t:Int, interest:Double = 0.0) = {
    from match {
      /* -----  business Firms --------------------------------------------------------------------------------------------------------------------------------------------------------------------- */
      case from:Firm =>
        to match {
          case to:Firm =>
            cause match {
              case "" =>

            }

          case to:Bank =>
            cause match {
              // always internal
              case "payBankAccountFee" =>
                withdraw(from.bankDeposits,   amount, t, sim)
                withdraw(  to.retailDeposits, amount, t, sim)
                deposit(   to.earnings,       amount, t, sim)

              case "payInterestOnBankLoan" =>
                if(sim.test) checkBankDeposits(to, from, "payInterestOnBankLoan", "before")
                withdraw(         from.bankDeposits,                                              amount, t, sim)  //
                withdraw(         from.interestOnDebt,                                            amount, t, sim)  //
                withdraw(           to.retailDeposits,                                            amount, t, sim)  //
                withdraw(           to.interestReceivables,                                       amount, t, sim)  //
                deposit(            to.earnings,                                                  amount, t, sim)  //
                if(sim.test) checkBankDeposits(to, from, "payInterestOnBankLoan", "after")

              // always internal
              case "payInterestOnBankLoanPartially" =>
                if(sim.test) checkBankDeposits(to, from, "payInterestOnBankLoanPartially", "before")
                withdraw(from.bankDeposits,        amount, t, sim)  //
```

```
          withdraw(from.interestOnDebt,      amount, t, sim)  //
          withdraw(  to.retailDeposits,      amount, t, sim)  //
          withdraw(  to.interestReceivables, amount, t, sim)  //
          deposit(   to.earnings,            amount, t, sim)  //
          if(sim.test) checkBankDeposits(to, from, "payInterestOnBankLoanPartially", "after")

        case "repayBankLoan" =>
          if(sim.test) checkBankDeposits(to, from, "repayBankLoan", "before")
          withdraw(from.bankDeposits,   amount, t, sim)
          withdraw(from.debtCapital,    amount, t, sim)
          withdraw(  to.retailDeposits, amount, t, sim)
          withdraw(  to.businessLoans,  amount, t, sim)
          if(sim.test) checkBankDeposits(to, from, "repayBankLoan", "after")

        case "repayBankLoanPartially" =>
          if(sim.test) checkBankDeposits(to, from, "repayBankLoanPartially", "before")
          withdraw(from.bankDeposits,   amount, t, sim)
          withdraw(from.debtCapital,    amount, t, sim)
          withdraw(  to.retailDeposits, amount, t, sim)
          withdraw(  to.businessLoans,  amount, t, sim)
          if(sim.test) checkBankDeposits(to, from, "repayBankLoanPartially", "after")







        // always internal
        case "negativeEquity1" =>
          if(sim.test) checkBankDeposits(to, from, "negativeEquity1", "before")
          if(amount + interest > from.bankDeposits.last) deposit(to.loanLosses, amount + interest - from.bankDeposits.last, t, sim)
          withdraw(  to.retailDeposits,      math.min(amount + interest, from.bankDeposits.last), t, sim)
          withdraw(from.bankDeposits,        math.min(amount + interest, from.bankDeposits.last), t, sim)
          if(sim.test) require(from.bankDeposits.last < 0.1, s"$from has deposits ($amount / ${from.bankDeposits.last}) left after shut down...")
          withdraw(  to.businessLoans,       amount,                                              t, sim)
          withdraw(  to.interestReceivables, interest,                                            t, sim)
          if(sim.test) checkBankDeposits(to, from, "negativeEquity1", "after")

        case _ => sys.error("error in Firm -> Bank payment.")
      }




    case to:BrokerDealer =>
      cause match {

        case "payInterestOnBrokerDealerLoan" =>
          if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "payInterestOnBrokerDealerLoan", "before")
          withdraw(from.bankDeposits,        amount, t, sim)  //
          withdraw(from.interestOnDebt,      amount, t, sim)  //
          if(from.houseBank != to.houseBank){
            if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
```

```
        withdraw(from.houseBank.retailDeposits, amount, t, sim)
        deposit(   to.houseBank.retailDeposits, amount, t, sim)
        withdraw(from.houseBank.cbReserves,     amount, t, sim)
        deposit(   to.houseBank.cbReserves,     amount, t, sim)
        registerReserveFlow(from.houseBank, to.houseBank, amount, t)
      }
      deposit( to.bankDeposits, amount, t, sim)
      withdraw(to.interestReceivables, amount, t, sim)  //
      deposit( to.earnings,            amount, t, sim)  //
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "payInterestOnBrokerDealerLoan", "after")


    case "payInterestOnBrokerDealerLoanPartially" =>
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "payInterestOnBrokerDealerLoanPartially", "before")
      withdraw(from.bankDeposits,        amount, t, sim)  //
      withdraw(from.interestOnDebt,      amount, t, sim)  //
      if(from.houseBank != to.houseBank){
        if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
        withdraw(from.houseBank.retailDeposits, amount, t, sim)
        deposit(   to.houseBank.retailDeposits, amount, t, sim)
        withdraw(from.houseBank.cbReserves,     amount, t, sim)
        deposit(   to.houseBank.cbReserves,     amount, t, sim)
        registerReserveFlow(from.houseBank, to.houseBank, amount, t)
      }
      deposit( to.bankDeposits, amount, t, sim)
      withdraw(to.interestReceivables, amount, t, sim)  //
      deposit( to.earnings,            amount, t, sim)  //
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "payInterestOnBrokerDealerLoanPartially", "after")











    case "repayBrokerDealerLoan" =>
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayBrokerDealerLoan", "before")
      withdraw(from.bankDeposits, amount, t, sim)
      withdraw(from.debtCapital,  amount, t, sim)
      if(from.houseBank != to.houseBank){
        if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
        withdraw(from.houseBank.retailDeposits, amount, t, sim)
        deposit(   to.houseBank.retailDeposits, amount, t, sim)
        withdraw(from.houseBank.cbReserves,     amount, t, sim)
        deposit(   to.houseBank.cbReserves,     amount, t, sim)
        registerReserveFlow(from.houseBank, to.houseBank, amount, t)
      }
      deposit( to.bankDeposits,  amount, t, sim)
      withdraw(to.businessLoans, math.min(to.businessLoans.last, amount), t, sim)
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayBrokerDealerLoan", "after")
```

```scala



        case "repayBrokerDealerLoanPartially" =>
          if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayBrokerDealerLoanPartially", "before")
          withdraw(from.bankDeposits, amount, t, sim)
          withdraw(from.debtCapital,  amount, t, sim)
          if(from.houseBank != to.houseBank){
            if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
            withdraw(from.houseBank.retailDeposits, amount, t, sim)
            deposit(   to.houseBank.retailDeposits, amount, t, sim)
            withdraw(from.houseBank.cbReserves,     amount, t, sim)
            deposit(   to.houseBank.cbReserves,     amount, t, sim)
            registerReserveFlow(from.houseBank, to.houseBank, amount, t)
          }
          deposit( to.bankDeposits,  amount, t, sim)
          withdraw(to.businessLoans, math.min(to.businessLoans.last, amount), t, sim)
          if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayBrokerDealerLoanPartially", "after")



        case "negativeEquity1" =>
          if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "negativeEquity1", "before")
          val funds2pay = math.min(amount + interest, from.bankDeposits.last)
          if(amount + interest > from.bankDeposits.last) deposit(to.loanLosses, amount + interest - from.bankDeposits.last, t, sim)
          withdraw(from.bankDeposits,       funds2pay, t, sim)
          deposit(   to.bankDeposits,       funds2pay, t, sim)
          if(from.houseBank != to.houseBank){
            if(from.houseBank.cbReserves.last < funds2pay) from.houseBank.getIntraDayLiquidity(funds2pay, t)
            withdraw(from.houseBank.retailDeposits, funds2pay,  t, sim)
            deposit(   to.houseBank.retailDeposits, funds2pay,  t, sim)
            withdraw(from.houseBank.cbReserves,     funds2pay,  t, sim)
            deposit(   to.houseBank.cbReserves,     funds2pay,  t, sim)
            registerReserveFlow(from.houseBank, to.houseBank, funds2pay, t)
          }
          if(sim.test) require(from.bankDeposits.last < 0.1, s"$from has deposits ($amount / ${from.bankDeposits.last}) left after shut down...")
          withdraw(  to.businessLoans,      math.min(to.businessLoans.last, amount),  t, sim)
          withdraw(  to.interestReceivables, interest,                                t, sim)
          if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "negativeEquity1", "after")


        case _ => sys.error("error in payments concerning >> Firm -> BrokerDealer <<.")

      }







    case to:HH =>
      cause match {
        case "payWage0" =>
          withdraw(         from.cash,             amount,                                    t, sim)
```

```
        deposit(            to.cash,             amount - sim.government.incomeTax(amount), t, sim)
        deposit(sim.government.cash,             sim.government.incomeTax(amount),          t, sim)
        deposit(sim.government.incomeTaxRevenue, sim.government.incomeTax(amount),          t, sim)

      case "payWage1" =>
        if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "payWage1", "before")
        withdraw(         from.bankDeposits, amount,                                       t, sim)
        deposit(            to.bankDeposits, amount - sim.government.incomeTax(amount),   t, sim)
        deposit(sim.government.bankDeposits, sim.government.incomeTax(amount),            t, sim)
        deposit(  to.houseBank.govDeposits,  sim.government.incomeTax(amount),            t, sim)
        if(from.houseBank != to.houseBank){
          if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
          withdraw(from.houseBank.retailDeposits, amount,                                    t, sim)
          deposit(   to.houseBank.retailDeposits, amount - sim.government.incomeTax(amount), t, sim)
          withdraw(from.houseBank.cbReserves,     amount,                                    t, sim)
          deposit(   to.houseBank.cbReserves,     amount,                                    t, sim)
          registerReserveFlow(from.houseBank, to.houseBank, amount, t)
        } else withdraw(  to.houseBank.retailDeposits,   sim.government.incomeTax(amount),   t, sim)
        deposit(sim.government.incomeTaxRevenue, sim.government.incomeTax(amount),           t, sim)
        if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "payWage1", "after")

      case "repayPrivateLoan" =>
        withdraw(         from.cash,                   amount + interest,                                                t, sim)
        deposit(            to.cash,                   amount + (1 - sim.government.capitalGainsTax.last) * interest,   t, sim)
        deposit(sim.government.cash,                   sim.government.capitalGainsTax.last * interest,                   t, sim)
        deposit(sim.government.capitalGainsTaxRevenue, sim.government.capitalGainsTax.last * interest,                   t, sim)
        withdraw(           to.loans,                  amount + interest,                                                t, sim)
        withdraw(         from.debtCapital,            amount,                                                           t, sim)
        withdraw(         from.interestOnDebt,         interest,                                                         t, sim)

      case "repayPrivateLoanPartially" =>
        withdraw(from.cash,           from.cash.last,    t, sim)
        deposit(   to.cash,           from.cash.last,    t, sim)
        withdraw(  to.loans,          amount + interest, t, sim)
        withdraw(from.debtCapital,    amount,            t, sim)
        withdraw(from.interestOnDebt, interest,          t, sim)

      case "dividends0" =>
        withdraw(         from.cash,                   amount,                                             t, sim)
        deposit(            to.cash,                   amount * (1 - sim.government.capitalGainsTax.last), t, sim)
        deposit(sim.government.cash,                   amount *      sim.government.capitalGainsTax.last,  t, sim)
        deposit(sim.government.capitalGainsTaxRevenue, amount *      sim.government.capitalGainsTax.last,  t, sim)

      case "dividends1" =>
        if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "dividends1", "before")
        withdraw(         from.bankDeposits, amount,                                             t, sim)
        deposit(            to.bankDeposits, amount * (1 - sim.government.capitalGainsTax.last), t, sim)
        deposit(sim.government.bankDeposits, amount *      sim.government.capitalGainsTax.last,  t, sim)
        deposit(  to.houseBank.govDeposits,  amount *      sim.government.capitalGainsTax.last,  t, sim)
        if(from.houseBank != to.houseBank){
          if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
          withdraw(from.houseBank.retailDeposits,   amount,                                             t, sim)
          deposit(   to.houseBank.retailDeposits,   amount * (1 - sim.government.capitalGainsTax.last), t, sim)
          withdraw(from.houseBank.cbReserves,       amount,                                             t, sim)
```

```
              deposit(   to.houseBank.cbReserves,       amount,                                              t, sim)
              registerReserveFlow(from.houseBank, to.houseBank, amount, t)
            } else withdraw(  to.houseBank.retailDeposits, amount * sim.government.capitalGainsTax.last,    t, sim)
            deposit(sim.government.capitalGainsTaxRevenue, amount * sim.government.capitalGainsTax.last,    t, sim)
            to.dividendsReceived.update(to.dividendsReceived.size-1, to.dividendsReceived.last + amount * (1 - sim.government.capitalGainsTax.last))
            if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "dividends1", "after")

          case "negativeEquity0" =>
            withdraw(from.cash, amount, t, sim)
            deposit(   to.cash, amount, t, sim)

          case "repayCapital" =>
            if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayCapital", "before")
            withdraw(from.bankDeposits, amount, t, sim)
            deposit(   to.bankDeposits, amount, t, sim)
            if(from.houseBank != to.houseBank){
              if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
              withdraw(from.houseBank.retailDeposits, amount, t, sim)
              deposit(   to.houseBank.retailDeposits, amount, t, sim)
              withdraw(from.houseBank.cbReserves,     amount, t, sim)
              deposit(   to.houseBank.cbReserves,     amount, t, sim)
              registerReserveFlow(from.houseBank, to.houseBank, amount, t)
            }
            if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayCapital", "after")

          case _ => sys.error("error in Firm -> HH payment.")
        }

      case to:Government =>
        cause match {
          case "corporateTax0" =>
            withdraw(         from.cash,                amount, t, sim)
            deposit(            to.cash,                amount, t, sim)
            deposit(sim.government.corporateTaxRevenue, amount, t, sim)

          case "corporateTax1" =>
            withdraw(          from.bankDeposits,        amount, t, sim)
            withdraw(from.houseBank.retailDeposits,      amount, t, sim)
            deposit( sim.government.bankDeposits,        amount, t, sim)
            deposit( from.houseBank.govDeposits,         amount, t, sim)
            deposit( sim.government.corporateTaxRevenue, amount, t, sim)

          case _ => sys.error("error in Firm -> Gov payment.")
        }
    }

  /* -----  commercial
Banks ----------------------------------------------------------------------------------------------------------------------------- */
  case from:Bank =>
    to match {
      case to:Firm =>
        cause match {
          case "grantLoan" =>
            if(sim.test) checkBankDeposits(from, to, "grantLoan", "before")
```

```
            deposit(  to.bankDeposits,        amount,   t, sim)
            deposit(  to.debtCapital,         amount,   t, sim)
            deposit(  to.interestOnDebt,      interest, t, sim)
            deposit(from.retailDeposits,      amount,   t, sim)
            deposit(from.businessLoans,       amount,   t, sim)
            deposit(from.interestReceivables, interest, t, sim)
            if(sim.test) checkBankDeposits(from, to, "grantLoan", "after")

          case "interestOnRetailDeposits" =>
            if(sim.test) checkBankDeposits(from, to, "interestOnRetailDeposits", "before")
            deposit(from.retailDeposits, amount, t, sim)
            deposit(  to.bankDeposits,   amount, t, sim)
            deposit(from.COGS,           amount, t, sim)
            if(sim.test) checkBankDeposits(from, to, "interestOnRetailDeposits", "after")

          case _ => sys.error("error in Bank -> Firm payment.")
        }


      case to:Bank =>
        cause match {
          case "overnightIBMloan" =>
            withdraw(from.cbReserves,          amount,             t, sim)
            deposit( from.interbankLoans,      amount,             t, sim)
            deposit( from.interestReceivables, interest,           t, sim)
            deposit(   to.cbReserves,          amount,             t, sim)
            deposit(   to.interbankLiabilities, amount + interest, t, sim)
            registerIBMloanFlow(from, to,      amount,             t)

          case "repayOvernightIBMloan" =>
            from.cbReserves(from.cbReserves.length-1) -= amount + interest
            withdraw(from.interbankLiabilities,          amount + interest, t, sim)
            deposit(   to.cbReserves,                    amount + interest, t, sim)
            withdraw(  to.interbankLoans,                amount,            t, sim)
            withdraw(  to.interestReceivables,           interest,          t, sim)
            deposit( from.COGS,                          interest,          t, sim)
            deposit(   to.earnings,                      interest,          t, sim)

          case "depreciateOvernightIBMloan" =>
            withdraw(from.interbankLiabilities,          amount + interest, t, sim)
            withdraw(  to.interbankLoans,                amount,            t, sim)
            withdraw(  to.interestReceivables,           interest,          t, sim)

          case "cleanOvernightIBMloan2InsolventBank" =>
            withdraw(from.interbankLiabilities,          amount + interest, t, sim)
            withdraw(  to.interbankLoans,                amount,            t, sim)
            withdraw(  to.interestReceivables,           interest,          t, sim)


          case "transferGovDeposits"  =>
            if(from.cbReserves.last < amount) from.getIntraDayLiquidity(amount, t)
            withdraw(from.govDeposits, amount, t, sim)
            withdraw(from.cbReserves,  amount, t, sim)
            deposit(   to.govDeposits, amount, t, sim)
```

```
                deposit(   to.cbReserves,  amount, t, sim)
                registerReserveFlow(from, to, amount, t)

              case _ => sys.error("error in Bank -> Bank payment.")
            }

          case to:HH =>
            cause match {
              case "interestOnRetailDeposits" =>
                if(sim.test) checkBankDeposits(from, to, "interestOnRetailDeposits", "before")
                deposit(from.retailDeposits, amount, t, sim)
                deposit(  to.bankDeposits,   amount, t, sim)
                deposit(from.COGS,           amount, t, sim)
                to.interestOnDeposits.update(to.interestOnDeposits.size-1, to.interestOnDeposits.last + amount)
                if(sim.test) checkBankDeposits(from, to, "interestOnRetailDeposits", "after")

              case "dividends1" =>
                deposit(          from.govDeposits,            amount *      sim.government.capitalGainsTax.last,  t, sim)
                deposit(            to.bankDeposits,           amount * (1 - sim.government.capitalGainsTax.last), t, sim)
                deposit(sim.government.bankDeposits,           amount *      sim.government.capitalGainsTax.last,  t, sim)
                deposit(sim.government.capitalGainsTaxRevenue, amount *      sim.government.capitalGainsTax.last,  t, sim)
                if(from != to.houseBank){
                  if(from.cbReserves.last < amount * (1 - sim.government.capitalGainsTax.last)) from.getIntraDayLiquidity(amount * (1 - sim.government.capitalGainsTax.last), t, test = false)
                  deposit(   to.houseBank.retailDeposits,   amount * (1 - sim.government.capitalGainsTax.last), t, sim)
                  withdraw(from.cbReserves,                 amount * (1 - sim.government.capitalGainsTax.last), t, sim)
                  deposit(   to.houseBank.cbReserves,       amount * (1 - sim.government.capitalGainsTax.last), t, sim)
                  registerReserveFlow(from, to.houseBank,   amount * (1 - sim.government.capitalGainsTax.last), t)
                } else deposit(from.retailDeposits, amount * (1 - sim.government.capitalGainsTax.last), t, sim)
                to.dividendsReceived.update(to.dividendsReceived.size-1, to.dividendsReceived.last + amount * (1 - sim.government.capitalGainsTax.last))

              case _ => sys.error("error in Bank -> HH payment.")
            }

          case to:CentralBank =>
            cause match {
              case "repayIDL" =>
                from.cbReserves(from.cbReserves.length-1) -= amount
                withdraw(from.cbLiabilities,          amount, t, sim)
                withdraw(  to.reserves,               amount, t, sim)
                withdraw(  to.loans2CommercialBanks,  amount, t, sim)

              case "OSDF" =>
                withdraw(from.cbReserves, amount,            t, sim)
                deposit( from.OSDF,       amount,            t, sim)
                deposit( from.interestReceivables, interest, t, sim)
                to.reserves(to.reserves.size-1) -= amount
                deposit(   to.OSDF,       amount + interest, t, sim)

              case "repayOSLF" =>
                from.cbReserves(from.cbReserves.size-1) -=  amount + interest
                       withdraw(from.cbLiabilities,         amount + interest, t, sim)
                  to.reserves(    to.reserves.size-1  ) -=  amount + interest
                      withdraw(   to.loans2CommercialBanks, amount + interest, t, sim)
```

```
                  deposit(  from.COGS,                  interest,          t, sim)

          case "repayMonthlyOMO" =>
              from.cbReserves(from.cbReserves.length-1) -= amount + interest
                      withdraw(from.cbLiabilities,         amount + interest, t, sim)
                      withdraw(  to.reserves,              amount + interest, t, sim)
                      withdraw(  to.loans2CommercialBanks, amount + interest, t, sim)
                      deposit( from.COGS,                  interest,          t, sim)

          case _ => sys.error("error in Bank -> CB payment.")
        }


      case to:Government =>
        cause match {
          case "buyInitialGovBonds" =>
            deposit(from.govDeposits,  amount, t, sim)
            deposit(  to.bankDeposits, amount, t, sim)

          case "buyGovBonds" =>
            deposit(from.govDeposits,  amount, t, sim)
            deposit(  to.bankDeposits, amount, t, sim)

          case "recapitalizeBank" =>

          case "interestOnRetailDeposits" =>
            deposit(from.govDeposits,  amount, t, sim)
            deposit(  to.bankDeposits, amount, t, sim)
            deposit(from.COGS,         amount, t, sim)

          case "corporateTax1" =>
            deposit(           from.govDeposits,         amount, t, sim)
            deposit( sim.government.bankDeposits,        amount, t, sim)
            deposit( sim.government.corporateTaxRevenue, amount, t, sim)

          case _ => sys.error("error in Bank -> Gov payment.")
        }




      case to:MMMF =>
        cause match {
          case "fireSaleCollateral" =>
            deposit(from.retailDeposits, amount, t, sim)
            deposit(  to.bankDeposits,   amount, t, sim)


          case "interestOnRetailDeposits" =>
            if(sim.test) checkBankDeposits(from, to, "interestOnRetailDeposits", "before")
            deposit(from.retailDeposits, amount, t, sim)
            deposit(  to.bankDeposits,   amount, t, sim)
            deposit(from.COGS,           amount, t, sim)
```

```
            if(sim.test) checkBankDeposits(from, to, "interestOnRetailDeposits", "after")

          case _ => sys.error("error in Bank -> MMMF payment.")
      }

      case to:BrokerDealer =>
        cause match {
          case "fireSaleBonds" =>
            deposit(from.retailDeposits, amount, t, sim)
            deposit(  to.bankDeposits,   amount, t, sim)

          case "interestOnRetailDeposits" =>
            if(sim.test) checkBankDeposits(from, to, "interestOnRetailDeposits", "before")
            deposit(from.retailDeposits, amount, t, sim)
            deposit(  to.bankDeposits,   amount, t, sim)
            deposit(from.COGS,           amount, t, sim)
            if(sim.test) checkBankDeposits(from, to, "interestOnRetailDeposits", "after")

          case _ => sys.error("error in Bank -> BrokerDealer payment.")
        }


    }



  /* ----- HH ---------------------------------------------------------------------------------------------------------------------------------------------------------------------------- */
  case from:HH =>
    to match {
      case to:Firm =>
        cause match {

          case "privateLending" =>
            withdraw(from.cash,            amount,                                    t, sim)
            deposit(   to.cash,            amount,                                    t, sim)
            deposit( from.loans,           amount * (1 + from.interestOnLoans.last),  t, sim)
            deposit(   to.debtCapital,     amount,                                    t, sim)
            deposit(   to.interestOnDebt,  amount * from.interestOnLoans.last,        t, sim)

          case "consumption0" =>
            withdraw(from.cash, amount, t, sim)
            deposit(   to.cash, amount, t, sim)
```

```
    case "initialInvestmentF" =>
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "initialInvestmentF", "before")
      withdraw(from.bankDeposits, amount, t, sim)
      deposit(   to.bankDeposits, amount, t, sim)
      if(from.houseBank != to.houseBank){
        if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
        withdraw(from.houseBank.retailDeposits, amount,                                         t, sim)
        deposit(   to.houseBank.retailDeposits, amount,                                         t, sim)
        withdraw(from.houseBank.cbReserves,     amount,                                         t, sim)
        deposit(   to.houseBank.cbReserves,     amount,                                         t, sim)
        registerReserveFlow(from.houseBank, to.houseBank, amount,                               t)
      }
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "initialInvestmentF", "after")

    case "consumption1" =>
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "consumption1", "before")
      withdraw(from.bankDeposits, amount, t, sim)
      deposit(   to.bankDeposits, amount, t, sim)
      if(from.houseBank != to.houseBank){
        if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
        withdraw(from.houseBank.retailDeposits, amount,                                         t, sim)
        deposit(   to.houseBank.retailDeposits, amount,                                         t, sim)
        withdraw(from.houseBank.cbReserves,     amount,                                         t, sim)
        deposit(   to.houseBank.cbReserves,     amount,                                         t, sim)
        registerReserveFlow(from.houseBank, to.houseBank, amount, t)
      }
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "consumption1", "after")

    case "reactivateFirm0" =>
      withdraw(from.cash, amount, t, sim)
      deposit(   to.cash, amount, t, sim)

    case "reactivateFirm1" =>
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "reactivateFirm1", "before")
      withdraw(from.bankDeposits, amount, t, sim)
      deposit(   to.bankDeposits, amount, t, sim)
      if(from.houseBank != to.houseBank){
        if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
        withdraw(from.houseBank.retailDeposits, amount,                                         t, sim)
        deposit(   to.houseBank.retailDeposits, amount,                                         t, sim)
        withdraw(from.houseBank.cbReserves,     amount,                                         t, sim)
        deposit(   to.houseBank.cbReserves,     amount,                                         t, sim)
        registerReserveFlow(from.houseBank, to.houseBank, amount, t)
      }
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "reactivateFirm1", "after")

    case _ => sys.error("error in HH -> Firm payment.")
}


case to:Bank =>
  cause match {
    case "initialInvestmentB" =>
      if(sim.test) checkBankDeposits(to, from, "initialInvestmentB", "before")
```

```scala
          withdraw(         from.bankDeposits,   amount, t, sim)
          withdraw(           to.retailDeposits, amount, t, sim)
          deposit(            to.govDeposits,    amount, t, sim)
          deposit(sim.government.bankDeposits,   amount, t, sim)
          if(sim.test) checkBankDeposits(to, from, "initialInvestmentB", "after")

        case "payBankAccountFee" =>
          withdraw(from.bankDeposits, amount, t, sim)
          withdraw(to.retailDeposits, amount, t, sim)
          deposit( to.earnings,       amount, t, sim)

        case "reactivateBank" =>
          withdraw(from.bankDeposits, amount, t, sim)
          if(from.houseBank != to){
              if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
              withdraw(from.houseBank.retailDeposits, amount,                                        t, sim)
              withdraw(from.houseBank.cbReserves,     amount,                                        t, sim)
              deposit(   to.cbReserves,               amount,                                        t, sim)
              registerReserveFlow(from.houseBank, to, amount,                                        t)
            } else withdraw(to.retailDeposits, amount, t, sim)


        case _ => sys.error("error in HH -> Bank payment.")
      }


    case to:MMMF =>
      cause match {
        case "foundMMMF" =>
          if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "foundMMMF", "before")
          withdraw(from.bankDeposits, amount, t, sim)
          deposit(   to.bankDeposits, amount, t, sim)
          if(from.houseBank != to.houseBank){
            if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
            withdraw(from.houseBank.retailDeposits, amount,                                        t, sim)
            deposit(   to.houseBank.retailDeposits, amount,                                        t, sim)
            withdraw(from.houseBank.cbReserves,     amount,                                        t, sim)
            deposit(   to.houseBank.cbReserves,     amount,                                        t, sim)
            registerReserveFlow(from.houseBank, to.houseBank, amount,                              t)
          }
          if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "foundMMMF", "after")


        case "investDeposits@MMMF"  =>
          if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "investDeposits@MMMF", "before")
          withdraw(from.bankDeposits, amount, t, sim)
          deposit(   to.bankDeposits, amount, t, sim)
          if(from.houseBank != to.houseBank){
            if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
            withdraw(from.houseBank.retailDeposits, amount,                                        t, sim)
            deposit(   to.houseBank.retailDeposits, amount,                                        t, sim)
            withdraw(from.houseBank.cbReserves,     amount,                                        t, sim)
            deposit(   to.houseBank.cbReserves,     amount,                                        t, sim)
            registerReserveFlow(from.houseBank, to.houseBank, amount,                              t)
```

```
      }
      deposit(from.loans,          amount + interest, t, sim)
      deposit(  to.deposits,       amount,            t, sim)
      deposit(  to.interestOnDebt, interest,          t, sim)
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "investDeposits@MMMF", "after")


    case "reactivateMMMF" =>
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "reactivateMMMF", "before")
      withdraw(from.bankDeposits, amount, t, sim)
      deposit(   to.bankDeposits, amount, t, sim)
      if(from.houseBank != to.houseBank){
        if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
        withdraw(from.houseBank.retailDeposits, amount,                                        t, sim)
        deposit(   to.houseBank.retailDeposits, amount,                                        t, sim)
        withdraw(from.houseBank.cbReserves,     amount,                                        t, sim)
        deposit(   to.houseBank.cbReserves,     amount,                                        t, sim)
        registerReserveFlow(from.houseBank, to.houseBank, amount,                              t)
      }
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "reactivateMMMF", "after")


    case _ => sys.error("error in payments concerning >> HH -> MMMF <<.")
  }



case to:BrokerDealer =>
  cause match {
    case "foundBrokerDealer" =>
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "foundBrokerDealer", "before")
      withdraw(from.bankDeposits, amount, t, sim)
      deposit(   to.bankDeposits, amount, t, sim)
      if(from.houseBank != to.houseBank){
        if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
        withdraw(from.houseBank.retailDeposits, amount,                                        t, sim)
        deposit(   to.houseBank.retailDeposits, amount,                                        t, sim)
        withdraw(from.houseBank.cbReserves,     amount,                                        t, sim)
        deposit(   to.houseBank.cbReserves,     amount,                                        t, sim)
        registerReserveFlow(from.houseBank, to.houseBank, amount,                              t)
      }
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "foundBrokerDealer", "after")


    case "reactivateBrokerDealer" =>
      if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "reactivateBrokerDealer", "before")
      withdraw(from.bankDeposits, amount, t, sim)
      deposit(   to.bankDeposits, amount, t, sim)
      if(from.houseBank != to.houseBank){
        if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
        withdraw(from.houseBank.retailDeposits, amount,                                        t, sim)
        deposit(   to.houseBank.retailDeposits, amount,                                        t, sim)
        withdraw(from.houseBank.cbReserves,     amount,                                        t, sim)
        deposit(   to.houseBank.cbReserves,     amount,                                        t, sim)
```

```
                registerReserveFlow(from.houseBank, to.houseBank, amount,                               t)
              }
              if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "reactivateBrokerDealer", "after")


            case _ => sys.error("error in payments concerning >> HH -> BrokerDealer <<.")
          }


        case to:HH =>
          cause match {
            case "" =>
          }


      case to:Government =>
        cause match {
          case "VAT0" =>
            withdraw(from.cash,       amount, t, sim)
            deposit(   to.cash,       amount, t, sim)
            deposit(   to.VATrevenue, amount, t, sim)

          case "VAT1" =>
            withdraw(from.bankDeposits,               amount, t, sim)
              withdraw(from.houseBank.retailDeposits, amount, t, sim)
              deposit(   to.bankDeposits,             amount, t, sim)
              deposit( from.houseBank.govDeposits,    amount, t, sim)
              deposit(   to.VATrevenue,               amount, t, sim)

          case "buyBonds" =>
            withdraw(from.cash, amount, t, sim)
            deposit(   to.cash, amount, t, sim)

          case _ => sys.error("error in HH -> Gov payment.")
        }
      }


    /* ----- MMMF ----------------------------------------------------------------------------------------------------------------------------------------------------- */
    case from:MMMF =>
      to match {
        case to:HH =>
          cause match {
            case "interestOnRetailDeposits" =>
              if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "interestOnRetailDeposits", "before")
              withdraw(from.bankDeposits, amount, t, sim)
              deposit(   to.bankDeposits, amount, t, sim)
              if(from.houseBank != to.houseBank){
                if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
                withdraw(from.houseBank.retailDeposits, amount,                                         t, sim)
                deposit(   to.houseBank.retailDeposits, amount,                                         t, sim)
                withdraw(from.houseBank.cbReserves,     amount,                                         t, sim)
                deposit(   to.houseBank.cbReserves,     amount,                                         t, sim)
```

```
      registerReserveFlow(from.houseBank, to.houseBank, amount,                              t)
    }
    if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "interestOnRetailDeposits", "after")


  case "payDividends" =>


  case "withdrawDepositsFromMMMF_A" =>
    if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "withdrawDepositsFromMMMF_A", "before")
    withdraw(from.bankDeposits, amount, t, sim)
    deposit(   to.bankDeposits, amount, t, sim)
    if(from.houseBank != to.houseBank){
      if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
      withdraw(from.houseBank.retailDeposits, amount,                                        t, sim)
      deposit(   to.houseBank.retailDeposits, amount,                                        t, sim)
      withdraw(from.houseBank.cbReserves,     amount,                                        t, sim)
      deposit(   to.houseBank.cbReserves,     amount,                                        t, sim)
      registerReserveFlow(from.houseBank, to.houseBank, amount,                              t)
    }
    withdraw(from.deposits, amount, t, sim)
    withdraw(  to.loans,    amount, t, sim)
    if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "withdrawDepositsFromMMMF_A", "after")


  case "withdrawDepositsFromMMMF_B" =>
    if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "withdrawDepositsFromMMMF_B", "before")
    withdraw(from.interestOnDebt,    amount, t, sim)
    withdraw(  to.loans,             amount, t, sim)
    if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "withdrawDepositsFromMMMF_B", "after")


  case "partiallyRepayInvestedDepositsDue2BankruptMMMF" =>
    if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "partiallyRepayInvestedDepositsDue2BankruptMMMF", "before")
    withdraw(from.bankDeposits, amount, t, sim)
    deposit(   to.bankDeposits, amount, t, sim)
    if(from.houseBank != to.houseBank){
      if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
      withdraw(from.houseBank.retailDeposits, amount,                                        t, sim)
      deposit(   to.houseBank.retailDeposits, amount,                                        t, sim)
      withdraw(from.houseBank.cbReserves,     amount,                                        t, sim)
      deposit(   to.houseBank.cbReserves,     amount,                                        t, sim)
      registerReserveFlow(from.houseBank, to.houseBank, amount,                              t)
    }
    if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "partiallyRepayInvestedDepositsDue2BankruptMMMF", "after")



  case "repayInvestedDepositsDue2BankruptMMMF" =>
    if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayInvestedDepositsDue2BankruptMMMF", "before")
    withdraw(from.bankDeposits, amount, t, sim)
    deposit(   to.bankDeposits, amount, t, sim)
    if(from.houseBank != to.houseBank){
      if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
```

```
          withdraw(from.houseBank.retailDeposits, amount,                                          t, sim)
          deposit(   to.houseBank.retailDeposits, amount,                                          t, sim)
          withdraw(from.houseBank.cbReserves,     amount,                                          t, sim)
          deposit(   to.houseBank.cbReserves,     amount,                                          t, sim)
          registerReserveFlow(from.houseBank, to.houseBank, amount,                                t)
        }
        if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayInvestedDepositsDue2BankruptMMMF", "after")



      case "repayCapital" =>
        if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayCapital", "before")
        withdraw(from.bankDeposits, amount, t, sim)
        deposit(   to.bankDeposits, amount, t, sim)
        if(from.houseBank != to.houseBank){
          if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
          withdraw(from.houseBank.retailDeposits, amount, t, sim)
          deposit(   to.houseBank.retailDeposits, amount, t, sim)
          withdraw(from.houseBank.cbReserves,     amount, t, sim)
          deposit(   to.houseBank.cbReserves,     amount, t, sim)
          registerReserveFlow(from.houseBank, to.houseBank, amount, t)
        }
        if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayCapital", "after")


      case _ => sys.error("error in payments concerning >> MMMF -> HH <<.")
    }


  case to:BrokerDealer =>
    cause match {

      case "overnightRepo" =>
        withdraw(from.bankDeposits, amount, t, sim)
        deposit(   to.bankDeposits, amount, t, sim)
        if(from.houseBank != to.houseBank){
          if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
          withdraw(from.houseBank.retailDeposits, amount,                                          t, sim)
          deposit(   to.houseBank.retailDeposits, amount,                                          t, sim)
          withdraw(from.houseBank.cbReserves,     amount,                                          t, sim)
          deposit(   to.houseBank.cbReserves,     amount,                                          t, sim)
          registerReserveFlow(from.houseBank, to.houseBank, amount,                                t)
        }
        deposit( from.claimsFromRepos, interest, t, sim)
        deposit(   to.liabsFromRepos,  interest, t, sim)


      case _ => sys.error("error in payments concerning >> MMMF -> BrokerDealer <<.")
  }

}
```

```
      /* -----
BrokerDealer  ----------------------------------------------------------------------------------------------------------------------------------------------------------------------
*/
      case from:BrokerDealer =>
        to match {

          case to:Firm =>
            cause match {

              case "grantLoan" =>
                if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "grantLoan", "before")
                withdraw(from.bankDeposits, amount, t, sim)
                deposit(   to.bankDeposits, amount, t, sim)
                if(from.houseBank != to.houseBank){
                  if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
                  withdraw(from.houseBank.retailDeposits, amount,                                         t, sim)
                  deposit(   to.houseBank.retailDeposits, amount,                                         t, sim)
                  withdraw(from.houseBank.cbReserves,     amount,                                         t, sim)
                  deposit(   to.houseBank.cbReserves,     amount,                                         t, sim)
                  registerReserveFlow(from.houseBank, to.houseBank, amount,                               t)
                }
                deposit(  to.debtCapital,         amount,   t, sim)
                deposit(  to.interestOnDebt,      interest, t, sim)
                deposit(from.businessLoans,       amount,   t, sim)
                deposit(from.interestReceivables, interest, t, sim)
                if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "grantLoan", "after")


              case _ => sys.error("error in payments concerning >> BrokerDealer -> Firm <<.")
            }


          case to:MMMF =>
            cause match {

              case "repurchaseCollateral" =>
                withdraw(from.bankDeposits, amount, t, sim)
                deposit(   to.bankDeposits, amount, t, sim)
                if(from.houseBank != to.houseBank){
                  if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
                  withdraw(from.houseBank.retailDeposits, amount,                                         t, sim)
                  deposit(   to.houseBank.retailDeposits, amount,                                         t, sim)
                  withdraw(from.houseBank.cbReserves,     amount,                                         t, sim)
                  deposit(   to.houseBank.cbReserves,     amount,                                         t, sim)
                  registerReserveFlow(from.houseBank, to.houseBank, amount,                               t)
                }
                withdraw(from.liabsFromRepos,  amount, t, sim)
                withdraw(  to.claimsFromRepos, amount, t, sim)


              case "payOvernightFee4RolledOverRepos" =>
```

```
          withdraw(from.bankDeposits, amount, t, sim)
          deposit(   to.bankDeposits, amount, t, sim)
          if(from.houseBank != to.houseBank){
            if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
            withdraw(from.houseBank.retailDeposits, amount,                                         t, sim)
            deposit(   to.houseBank.retailDeposits, amount,                                         t, sim)
            withdraw(from.houseBank.cbReserves,     amount,                                         t, sim)
            deposit(   to.houseBank.cbReserves,     amount,                                         t, sim)
            registerReserveFlow(from.houseBank, to.houseBank, amount,                               t)
          }




        case "quitRepoDue2BankruptMMMF" =>
          withdraw(from.bankDeposits, amount, t, sim)
          deposit(   to.bankDeposits, amount, t, sim)
          if(from.houseBank != to.houseBank){
            if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
            withdraw(from.houseBank.retailDeposits, amount,                                         t, sim)
            deposit(   to.houseBank.retailDeposits, amount,                                         t, sim)
            withdraw(from.houseBank.cbReserves,     amount,                                         t, sim)
            deposit(   to.houseBank.cbReserves,     amount,                                         t, sim)
            registerReserveFlow(from.houseBank, to.houseBank, amount,                               t)
          }
          withdraw(from.liabsFromRepos,  interest, t, sim)
          withdraw(  to.claimsFromRepos, interest, t, sim)



        case _ => sys.error("error in payments concerning >> BrokerDealer -> MMMF <<.")
    }



    case to:HH =>
      cause match {
        case "payDividends" =>


        case "repayCapital" =>
          if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayCapital", "before")
          withdraw(from.bankDeposits, amount, t, sim)
          deposit(   to.bankDeposits, amount, t, sim)
          if(from.houseBank != to.houseBank){
            if(from.houseBank.cbReserves.last < amount) from.houseBank.getIntraDayLiquidity(amount, t)
            withdraw(from.houseBank.retailDeposits, amount, t, sim)
            deposit(   to.houseBank.retailDeposits, amount, t, sim)
            withdraw(from.houseBank.cbReserves,     amount, t, sim)
            deposit(   to.houseBank.cbReserves,     amount, t, sim)
            registerReserveFlow(from.houseBank, to.houseBank, amount, t)
          }
          if(sim.test) checkBankDepositsBetweenNonBanks(from, to, from.houseBank, to.houseBank, "repayCapital", "after")
```

```scala

              case _ => sys.error("error in payments concerning >> BrokerDealer -> HH <<.")
          }


          case to:Government =>
            cause match {
              case "buyGovBonds" =>
                withdraw(from.bankDeposits,             amount, t, sim)
                withdraw(from.houseBank.retailDeposits, amount, t, sim)
                deposit(   to.bankDeposits,             amount, t, sim)
                deposit( from.houseBank.govDeposits,    amount, t, sim)


              case _ => sys.error("error in payments concerning >> BrokerDealer -> Gov <<.")
          }











          case to:CentralBank =>
            cause match {
              case "repayCBdebt"  =>
                withdraw(  to.loans2CommercialBanks,    amount, t, sim)
                withdraw(  to.reserves,                 amount, t, sim)
                withdraw(from.houseBank.cbReserves,     amount, t, sim)
                withdraw(from.houseBank.retailDeposits, amount, t, sim)
                withdraw(from.bankDeposits,             amount, t, sim)
                withdraw(from.liabsFromRepos,           amount, t, sim)
                to.liquidityInsuranceDebtBD(from) -=    amount


              case _ => sys.error("error in CB -> BD payment.")
            }


        }




      /* -----
  CB  ---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  - */
      case from:CentralBank =>
        to match {

          case to:Bank =>
```

```
        cause match {

          case "payInterestOnReserves" =>
            deposit(from.reserves,   amount,   t, sim)
            deposit(  to.cbReserves, amount,   t, sim)
            deposit(  to.earnings,   amount,   t, sim)

          case "provideIDL" =>
            deposit(from.reserves,              amount, t, sim)
            deposit(from.loans2CommercialBanks, amount, t, sim)
            deposit(  to.cbReserves,            amount, t, sim)
            deposit(  to.cbLiabilities,         amount, t, sim)
            registerIDLflow(to, amount, t)

          case "OMO" =>
            deposit(from.reserves,              amount,            t, sim)
            deposit(  to.cbReserves,            amount,            t, sim)
            deposit(from.loans2CommercialBanks, amount + interest, t, sim)
            deposit(  to.cbLiabilities,         amount + interest, t, sim)

          case "repayOSDF" =>
            deposit( from.reserves,           amount + interest, t, sim)
            withdraw(from.OSDF,               amount + interest, t, sim)
            deposit(  to.cbReserves,          amount + interest, t, sim)
            withdraw( to.OSDF,                amount,            t, sim)
            withdraw( to.interestReceivables, interest,          t, sim)
            deposit(  to.earnings,            interest,          t, sim)

          case "repayOSDFwoInterest" =>
            deposit( to.cbReserves,          amount,            t, sim)
            withdraw(to.OSDF,                amount,            t, sim)
            withdraw(to.interestReceivables, interest,          t, sim)
            deposit( from.reserves,          amount,            t, sim)
            withdraw(from.OSDF,              amount + interest, t, sim)




          case "OSLF" =>
            deposit(from.reserves,              amount,            t, sim)
            deposit(  to.cbReserves,            amount,            t, sim)
            deposit(from.loans2CommercialBanks, amount + interest, t, sim)
            deposit(  to.cbLiabilities,         amount + interest, t, sim)

          case _ => sys.error("error in CB -> Bank payment.")
        }


      case to:Government =>
        cause match {
          case "buyInitialGovBonds" =>
            deposit(from.governmentsAccount, amount, t, sim)
            deposit(  to.cbDeposits,         amount, t, sim)

```

```scala
              case _ => sys.error("error in CB -> Gov payment.")
            }


          case to:BrokerDealer =>
            cause match {

              case "liquidityInsuranceBD" =>
                deposit(from.loans2CommercialBanks,    amount, t, sim)
                deposit(from.reserves,                 amount, t, sim)
                deposit(  to.houseBank.cbReserves,     amount, t, sim)
                deposit(  to.houseBank.retailDeposits, amount, t, sim)
                deposit(  to.bankDeposits,             amount, t, sim)
                deposit(  to.liabsFromRepos,           amount, t, sim)
                if(from.liquidityInsuranceDebtBD.contains(to)) from.liquidityInsuranceDebtBD(to) += amount else from.liquidityInsuranceDebtBD += to -> amount


              case _ => sys.error("error in CB -> BD payment.")
            }


        }




      /* -----
Government  ----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
*/
      case from:Government =>
        to match {

          case to:Firm =>
            cause match {

              case "govConsumption0" =>
                withdraw(         from.cash,        amount, t, sim)
                deposit(            to.cash,        amount, t, sim)
                deposit(sim.government.govSpending, amount, t, sim)

              case "govConsumption1" =>
                if(sim.test) checkBankDeposits(to.houseBank, from, "govConsumption1", "before")
                if(to.houseBank.govDeposits.last < amount) from.getGovDeposits(to.houseBank, amount, t)
                withdraw(         from.bankDeposits,   amount, t, sim)
                withdraw( to.houseBank.govDeposits,    amount, t, sim)
                deposit(            to.bankDeposits,   amount, t, sim)
                deposit(  to.houseBank.retailDeposits, amount, t, sim)
                deposit(sim.government.govSpending,    amount, t, sim)
                if(sim.test) checkBankDeposits(to.houseBank, from, "govConsumption1", "after")

              case _ => sys.error("error in Gov -> Firm payment.")
            }

```

```

        case to:HH =>
          cause match {

            case "unemploymentBenefit0" =>
              if(from.cash.last >= amount){
                withdraw(from.cash,                   amount, t, sim)
                deposit(   to.cash,                   amount, t, sim)
                deposit(  sim.government.govSpending, amount, t, sim)
              }

            case "unemploymentBenefit1" =>
              if(sim.test) checkBankDeposits(to.houseBank, from, "unemploymentBenefit1", "before")
              if(to.houseBank.govDeposits.last < amount) from.getGovDeposits(to.houseBank, amount, t)
              withdraw(        from.bankDeposits,   amount, t, sim)
              withdraw(to.houseBank.govDeposits,    amount, t, sim)
              deposit(           to.bankDeposits,   amount, t, sim)
              deposit( to.houseBank.retailDeposits, amount, t, sim)
              if(sim.test) checkBankDeposits(to.houseBank, from, "unemploymentBenefit1", "after")

            case "payCoupon" => withdraw(from.cash,                   amount * (1 - from.capitalGainsTax.last), t, sim)
                                withdraw(  to.cash,                   amount * (1 - from.capitalGainsTax.last), t, sim)
                                deposit( from.capitalGainsTaxRevenue, amount * from.capitalGainsTax.last,       t, sim)

            case "repayDuePublicDebt0" =>
              withdraw(from.cash, amount, t, sim)
              deposit(   to.cash, amount, t, sim)

            case _ => sys.error("error in Gov -> HH payment.")
          }



        case to:Bank =>
          cause match {
            case "payBankAccountFee" =>
              if(to.govDeposits.last < amount) from.getGovDeposits(to, amount, t)
              withdraw(from.bankDeposits, amount, t, sim)
              withdraw(to.govDeposits,    amount, t, sim)
              deposit( to.earnings,       amount, t, sim)

            case "payCoupon" =>
              if(to.govDeposits.last < amount) from.getGovDeposits(to, amount, t)
              withdraw(from.bankDeposits,           amount, t, sim)
              withdraw(  to.govDeposits,            amount, t, sim)
              deposit(   to.earnings,               amount, t, sim)

            case "repayDuePublicDebt1" =>
              if(to.govDeposits.last < amount) from.getGovDeposits(to, amount, t)
              withdraw(from.bankDeposits, amount, t, sim)
              withdraw(  to.govDeposits,  amount, t, sim)

            case _ => sys.error("error in Gov -> Bank payment.")
          }
```

```
case to:BrokerDealer =>
  cause match {
    case "payCoupon" =>
      if(to.houseBank.govDeposits.last < amount) from.getGovDeposits(to.houseBank, amount, t)
      withdraw(from.bankDeposits,             amount, t, sim)
      deposit(   to.bankDeposits,             amount, t, sim)
      withdraw(  to.houseBank.govDeposits,    amount, t, sim)
      deposit(   to.houseBank.retailDeposits, amount, t, sim)
      deposit(   to.earnings,                 amount, t, sim)


    case "repayDuePublicDebt1" =>
      if(to.houseBank.govDeposits.last < amount) from.getGovDeposits(to.houseBank, amount, t)
      withdraw(from.bankDeposits,             amount, t, sim)
      deposit(   to.bankDeposits,             amount, t, sim)
      withdraw(  to.houseBank.govDeposits,    amount, t, sim)
      deposit(   to.houseBank.retailDeposits, amount, t, sim)

    case "securitizeLoans" =>
      if(to.houseBank.govDeposits.last < amount) from.getGovDeposits(to.houseBank, amount, t)
      withdraw(to.businessLoans,            amount, t, sim)
      deposit( to.bankDeposits,             amount, t, sim)
      deposit( to.houseBank.retailDeposits, amount, t, sim)
      withdraw(to.houseBank.govDeposits,    amount, t, sim)
      withdraw(sim.government.bankDeposits, amount, t, sim)

    case _ => sys.error("error in Gov -> BrokerDealer payment.")
}



case to:CentralBank =>
  cause match {

    case "payCoupon" =>
      val netAmount = amount * (1 - from.capitalGainsTax.last)
      if(to.governmentsAccount.last < netAmount) sim.government.createBondRelationship(to, netAmount, "buyInitialGovBonds", t, false)
      withdraw(from.cbDeposits,            netAmount,                          t, sim)
      withdraw(  to.governmentsAccount,    netAmount,                          t, sim)
      deposit(from.capitalGainsTaxRevenue, amount * from.capitalGainsTax.last, t, sim)

    case "repayDuePublicDebt1" =>
      if(to.governmentsAccount.last < amount) sim.government.createBondRelationship(to, amount, "buyInitialGovBonds", t, false)
      withdraw(from.cbDeposits,         amount, t, sim)
      withdraw(  to.governmentsAccount, amount, t, sim)
```

```scala

            case _ => sys.error("error in Gov -> CB payment.")
          }
        }


    case _  => sys.error("The transfer of money can only occur between agents. This either no Firm, Bank, HH, CB or Government!")
  }













/**
 *
 *    */
  def checkBankDepositsBetweenNonBanks (from:Agent, to:Agent, fromHouseBank:Bank, toHouseBank:Bank, cause:String, when:String) {
    val firmDepositsOfFromHouseBank = fromHouseBank.businessClients.map(_.bankDeposits.last).sum
    val   hhDepositsOfFromHouseBank = fromHouseBank.retailClients.map(  _.bankDeposits.last).sum
    val   firmDepositsOfToHouseBank =   toHouseBank.businessClients.map(_.bankDeposits.last).sum
    val     hhDepositsOfToHouseBank =   toHouseBank.retailClients.map(  _.bankDeposits.last).sum
    from match {

      case from:Firm =>
        to match {
          case to:HH =>
            require(
                SD(sim.bankList.map(_.cbReserves.last).sum, sim.centralBank.reserves.last, 2),
                s"reserves are not consistent $when $cause, deviation is ${rounded(sim.bankList.map(_.cbReserves.last).sum) - sim.centralBank.reserves.last}"
            )
            checkAndAdjust("reserveAccounts",   when, cause, sim.bankList.filter(_.active).map(_.cbReserves.last).sum, sim.centralBank.reserves,     t)
            checkAndAdjust("checkBankDeposits", when, cause, firmDepositsOfFromHouseBank + hhDepositsOfFromHouseBank,  fromHouseBank.retailDeposits, t)
            checkAndAdjust("checkBankDeposits", when, cause, firmDepositsOfToHouseBank   + hhDepositsOfToHouseBank,    toHouseBank.retailDeposits,   t)
            require(
                SD(firmDepositsOfFromHouseBank + hhDepositsOfFromHouseBank, fromHouseBank.retailDeposits.last, 5),
                s"checkBankDeposits failed    $when $cause: deviation is ${rounded(firmDepositsOfFromHouseBank + hhDepositsOfFromHouseBank) - fromHouseBank.retailDeposits.last}"
            )
            require(
                SD(firmDepositsOfToHouseBank   + hhDepositsOfToHouseBank,   toHouseBank.retailDeposits.last,   5),
                s"checkBankDeposits failed    $when $cause: deviation is ${rounded(firmDepositsOfToHouseBank   + hhDepositsOfToHouseBank)   - toHouseBank.retailDeposits.last}"
            )
        }

      case from:HH =>
        to match {
          case to:Firm =>
            require(
```

```scala
                SD(sim.bankList.map(_.cbReserves.last).sum, sim.centralBank.reserves.last, 2),
                s"reserves are not consistent $when $cause, deviation is ${rounded(sim.bankList.map(_.cbReserves.last).sum) - sim.centralBank.reserves.last}"
            )
            checkAndAdjust("reserveAccounts",   when, cause, sim.bankList.filter(_.active).map(_.cbReserves.last).sum, sim.centralBank.reserves,     t)
            checkAndAdjust("checkBankDeposits", when, cause, firmDepositsOfFromHouseBank + hhDepositsOfFromHouseBank,  fromHouseBank.retailDeposits, t)
            checkAndAdjust("checkBankDeposits", when, cause, firmDepositsOfToHouseBank   + hhDepositsOfToHouseBank,    toHouseBank.retailDeposits,   t)
            require(
                SD(firmDepositsOfFromHouseBank + hhDepositsOfFromHouseBank, fromHouseBank.retailDeposits.last, 5),
                s"checkBankDeposits failed    $when $cause: deviation is ${rounded(firmDepositsOfFromHouseBank + hhDepositsOfFromHouseBank) - fromHouseBank.retailDeposits.last}"
            )
            require(
                SD(firmDepositsOfToHouseBank   + hhDepositsOfToHouseBank,   toHouseBank.retailDeposits.last,   5),
                s"checkBankDeposits failed    $when $cause: deviation is ${rounded(firmDepositsOfToHouseBank   + hhDepositsOfToHouseBank)   - toHouseBank.retailDeposits.last}"
            )
      }
    }
  }



/**
 *
 *    */
  def checkBankDeposits (houseBank:Bank, client:Agent, cause:String, when:String) {
    val firmDepositsOfHouseBank = houseBank.businessClients.map(_.bankDeposits.last).sum
    val   hhDepositsOfHouseBank =   houseBank.retailClients.map(_.bankDeposits.last).sum
    checkAndAdjust("reserveAccounts",   when, cause, sim.bankList.filter(_.active).map(_.cbReserves.last).sum,                 sim.centralBank.reserves,     t)
    require(
        math.pow(rounded(sim.bankList.map(_.cbReserves.last).sum) - sim.centralBank.reserves.last, 2) <  5,
        s"reserves are not consistent $when $cause, deviation is ${rounded(sim.bankList.map(_.cbReserves.last).sum) - sim.centralBank.reserves.last}"
    )
    client match {

      case client:Firm =>
        checkAndAdjust("checkBankDeposits", when, cause, firmDepositsOfHouseBank + hhDepositsOfHouseBank, houseBank.retailDeposits, t)
        require(
            SD(firmDepositsOfHouseBank + hhDepositsOfHouseBank, houseBank.retailDeposits.last, 5),
            s"checkBankDeposits failed before $cause: deviation is ${rounded(firmDepositsOfHouseBank + hhDepositsOfHouseBank) - houseBank.retailDeposits.last}"
        )

      case client:HH   =>
        checkAndAdjust("checkBankDeposits", when, cause, firmDepositsOfHouseBank + hhDepositsOfHouseBank, houseBank.retailDeposits, t)
        require(
            SD(firmDepositsOfHouseBank + hhDepositsOfHouseBank, houseBank.retailDeposits.last, 5),
            s"checkBankDeposits failed before $cause: deviation is ${rounded(firmDepositsOfHouseBank + hhDepositsOfHouseBank) - houseBank.retailDeposits.last}"
        )

      case _           =>
    }
  }
```

```scala


 /**
 *
 *    */
def checkBankDepositsAfterTransaction (houseBank:Bank, client:Agent, cause:String) {
  val firmDepositsOfHouseBank = houseBank.businessClients.map(_.bankDeposits.last).sum
  val   hhDepositsOfHouseBank =   houseBank.retailClients.map(_.bankDeposits.last).sum
  require(
      math.pow(rounded(sim.bankList.map(_.cbReserves.last).sum) - sim.centralBank.reserves.last, 2) <  5,
      s"reserves are not consistent, deviation is ${rounded(sim.bankList.map(_.cbReserves.last).sum) - sim.centralBank.reserves.last}"
  )
  client match {

    case client:Firm =>
      println( houseBank.retailClients.map(_.bankDeposits.last) + " / " + houseBank.businessClients.map(_.bankDeposits.last) )
      println(houseBank.retailDeposits.last)
      require(
          math.pow(rounded(firmDepositsOfHouseBank + hhDepositsOfHouseBank) - houseBank.retailDeposits.last, 2) <= 5,
          s"checkBankDeposits failed after $cause: deviation is ${rounded(firmDepositsOfHouseBank + hhDepositsOfHouseBank) - houseBank.retailDeposits.last}"
      )

    case client:HH   =>
      println(s"$houseBank : ${houseBank.retailClients}")
      println(s"$houseBank : ${houseBank.retailClients.map(_.bankDeposits.last)}")
      println(s"$houseBank : ${houseBank.businessClients}")
      println(s"$houseBank : ${houseBank.businessClients.map(_.bankDeposits.last)}")
      println(s"$houseBank : ${houseBank.retailDeposits.last}")
      require(
          math.pow(rounded(firmDepositsOfHouseBank + hhDepositsOfHouseBank) - houseBank.retailDeposits.last, 2) <= 5,
          s"checkBankDeposits failed after $cause: deviation is ${rounded(firmDepositsOfHouseBank + hhDepositsOfHouseBank) - houseBank.retailDeposits.last}"
      )

    case _           =>
  }
}






def registerReserveFlow (from:Bank, to:Bank, amountOfReserves:Double, t:Int) {
    require(sim.reserveFlows(from)(to).size == t, s"registerReserveFlow failed because of too many entries in Array")
    println(s"reserveFlows: ${sim.reserveFlows.keys}")
    println(s"Transferring reserves of $amount from $from to $to.")
    sim.reserveFlows(from)(to)(t-1) += amountOfReserves
  }


  def registerIBMloanFlow (from:Bank, to:Bank, amountOfReserves:Double, t:Int) {
    require(sim.IBMloanFlows(from)(to).size == t, s"registerIBMloanFlow failed because of too many entries in Array")
    sim.IBMloanFlows(from)(to)(t-1) += amountOfReserves
    println(s"IBMloanFlows: ${sim.IBMloanFlows}")
```

```scala
    }

    def registerIDLflow (to:Bank, amountOfReserves:Double, t:Int) {
      require(sim.IDLflows(to).size == t, s"registerIDLflow failed because of too many entries in Array: ${sim.IDLflows(to).size} / $t")
      sim.IDLflows(to)(t-1) += amountOfReserves
      println(s"IDLflows: ${sim.IDLflows}")
    }


  }

} // End of Trait: accountManagement ------------------------------------------------------------------------------------------








//  Wholesale Funding Instruments (WFI)
trait IOU extends round {

  case class OMO               (                   borrowingBank:Bank, amountOfReserves:Double, interest:Double, tickOfRepayment:Int)       // short-term repo (1 month) with CB
  case class IBMloan           (lendingBank:Bank, borrowingBank:Bank, amountOfReserves:Double, interest:Double)                            // this is used at the end of each settlement day
  case class OvernightOSLFloan (                   borrowingBank:Bank, amountOfReserves:Double, interest:Double, haircut:Double = 0.0)      // overnight operational standing lending facility loan

  val riskAversionParameterWholesaleClients = 0.035
  val riskAversionParameterRetailClients    = 0.1

  /*  Creditworthiness  */

  /** tested
   *
   * returns the PD in dependence of the client's D/E-ratio used in the internal risk model of the deciding bank   */
  def PD (client:Corporation) = {
    client match {
      case client:Bank  =>  rounded( 1 - exp( -riskAversionParameterWholesaleClients * client.debt2EquityRatio) )
      case client:Firm  =>  rounded( 1 - exp( -riskAversionParameterRetailClients    * client.debt2EquityRatio) )
      case _      =>  error("To calculate the PD, the client must be either a Bank or a Firm!")
    }
  }


  /** tested
   *
   * returns probability for a loan to be granted in dependence of the client's D/E-ratio   */
  def probOfGrantingLoan2Client (client:Corporation) = {
    rounded( 1 - PD(client) )
  }

```

```
}// end of trait IOU














trait bonds extends round {

  val sim:Simulation

  // data of all agents which have the power to deal with bonds
  private val _bonds                             = ArrayBuffer[Double](0.0)
  private val _listOfBonds                       = Map[Long, Double]()
  private val _bondsPledgedAsCollateralForOMO    = Map[Long, Double]()     // (ID, Fraction pledged); maturity: one month
  private val _bondsPledgedAsCollateralForOSLF   = Map[Long, Double]()     // (ID, Fraction pledged); maturity: overnight
  private val _bondsPledgedAsCollateralForIDL    = Map[Long, Double]()     // (ID, Fraction pledged); maturity: settlement day
  private val _bondsPledgedAsCollateralForRepo   = Map[Long, Double]()     // (ID, Fraction pledged); maturity: overnight


  // getter
  def listOfBonds                     = _listOfBonds
  def bonds                           = _bonds
  def bondsPledgedAsCollateralForOMO  = _bondsPledgedAsCollateralForOMO
  def bondsPledgedAsCollateralForOSLF = _bondsPledgedAsCollateralForOSLF
  def bondsPledgedAsCollateralForIDL  = _bondsPledgedAsCollateralForIDL
  def bondsPledgedAsCollateralForRepo = _bondsPledgedAsCollateralForRepo

  def setID:Long = abs(sim.random.nextLong)


  def PV_LoB  (t:Int) = rounded( _listOfBonds.map{                    case(id, fraction) => PVofSoB(sim.government.findStackOfBondsByID(id), t) * fraction}.sum )
  def PV_OMO  (t:Int) = rounded( _bondsPledgedAsCollateralForOMO.map{ case(id, fraction) => PVofSoB(sim.government.findStackOfBondsByID(id), t) * fraction}.sum )
  def PV_OSLF (t:Int) = rounded( _bondsPledgedAsCollateralForOSLF.map{case(id, fraction) => PVofSoB(sim.government.findStackOfBondsByID(id), t) * fraction}.sum )
  def PV_IDL  (t:Int) = rounded( _bondsPledgedAsCollateralForIDL.map{ case(id, fraction) => PVofSoB(sim.government.findStackOfBondsByID(id), t) * fraction}.sum )
  def PV_Repo (t:Int) = rounded( _bondsPledgedAsCollateralForRepo.map{case(id, fraction) => PVofSoB(sim.government.findStackOfBondsByID(id), t) * fraction}.sum )


  def currentPVofSoBs   (t:Int):Double = time({ rounded(PV_LoB(t) + PV_OMO(t) + PV_OSLF(t) + PV_IDL(t)) }, "bank_currentPVofSoBs",         sim)
  def currentPVofSoBsBD (t:Int):Double = time({ rounded(PV_LoB(t) + PV_Repo(t)) },                               "BrokerDealer_currentPVofSoBs", sim)

  def printCompositionOfBonds (t:Int):Unit = {
    if(sim.test){
      try{
        require(
          SEc(rounded(PV_LoB(t) + PV_OMO(t) + PV_OSLF(t) + PV_IDL(t)), currentPVofSoBs(t), 5),
```

```
        s"Composition of bond valuation is not correct: ${PV_LoB(t)} + ${PV_OMO(t)} + ${PV_OSLF(t)} + ${PV_IDL(t)} / ${currentPVofSoBs(t)}"
      )
    } catch {
      case iae:java.lang.IllegalArgumentException =>
    }
  }
  if(sim.pln)  println(s"list: ${PV_LoB(t)} + OMO: ${PV_OMO(t)} + OSLF: ${PV_OSLF(t)} + IDL: ${PV_IDL(t)} = ${currentPVofSoBs(t)}")
}




def printCompositionOfBondsBD (t:Int):Unit = {
  if(sim.test){
    try{
      require(
          SEc(rounded(PV_LoB(t) + PV_Repo(t)), currentPVofSoBsBD(t), 5),
          s"Composition of bond valuation is not correct: ${PV_LoB(t)} + ${PV_Repo(t)} / ${currentPVofSoBsBD(t)}"
      )
    } catch {
      case iae:java.lang.IllegalArgumentException =>
    }
  }
  if(sim.pln)  println(s"list: ${PV_LoB(t)} + Repo: ${PV_Repo(t)} = ${currentPVofSoBsBD(t)}")
}

def currentPVofPledgeableBonds (t:Int):Double = time({ if(_listOfBonds.nonEmpty) PV_LoB(t) else 0.0 }, "bank_currentPVofPledgeableBonds", sim)

def updatePVofSoBs   (t:Int):Unit = _bonds(_bonds.size-1) = currentPVofSoBs(t)
def updatePVofSoBsBD (t:Int):Unit = _bonds(_bonds.size-1) = currentPVofSoBsBD(t)

def checkExistenceOfIDs (when:String, cause:String) = {
  if(_listOfBonds.nonEmpty){
    _listOfBonds.foreach{
      case(id, fraction) => require(sim.government.govLOB.map(_.id).contains(id), s"checkExistenceOfID (LOB) failed $when $cause: _govLOB does not contain ID $id of $this.")
    }
  }

  if(_bondsPledgedAsCollateralForOMO.nonEmpty){
    _bondsPledgedAsCollateralForOMO.foreach{
      case(id, fraction) => require(sim.government.govLOB.map(_.id).contains(id), s"checkExistenceOfID (OMO) failed $when $cause: _govLOB does not contain ID $id of $this.")
    }
  }
  if(_bondsPledgedAsCollateralForOSLF.nonEmpty){
    _bondsPledgedAsCollateralForOSLF.foreach{
      case(id, fraction) =>
        require(sim.government.govLOB.map(_.id).contains(id), s"checkExistenceOfID (OSLF) failed $when $cause: _govLOB does not contain ID $id (OSLF: $
{_bondsPledgedAsCollateralForOSLF}) of $this.")
    }
  }
  if(_bondsPledgedAsCollateralForIDL.nonEmpty){
    _bondsPledgedAsCollateralForIDL.foreach{
      case(id, fraction) =>
        require(sim.government.govLOB.map(_.id).contains(id), s"checkExistenceOfID (IDL) failed $when $cause: _govLOB does not contain ID $id of $this.")
```

```scala
    }
  }
  if(_bondsPledgedAsCollateralForRepo.nonEmpty){
    _bondsPledgedAsCollateralForRepo.foreach{
      case(id, fraction) =>
        require(sim.government.govLOB.map(_.id).contains(id), s"checkExistenceOfID (Repo) failed $when $cause: _govLOB does not contain ID $id of $this.")
    }
  }
}


def checkBankSoBCompleteness (bank:Bank) = {

  bank.listOfBonds.foreach{
    case(id, fraction) =>
      val fractions = ArrayBuffer[Double](fraction)
      if(bank.bondsPledgedAsCollateralForOMO.nonEmpty  && bank.bondsPledgedAsCollateralForOMO.contains(id))  fractions += bondsPledgedAsCollateralForOMO(id)
      if(bank.bondsPledgedAsCollateralForOSLF.nonEmpty && bank.bondsPledgedAsCollateralForOSLF.contains(id)) fractions += bondsPledgedAsCollateralForOSLF(id)
      if(bank.bondsPledgedAsCollateralForIDL.nonEmpty  && bank.bondsPledgedAsCollateralForIDL.contains(id))  fractions += bondsPledgedAsCollateralForIDL(id)
      require(roundTo9Digits(fractions.sum) == 1.0, s"SoB of $bank are not complete: ${fractions.sum} < 1.0")
  }



  bank.bondsPledgedAsCollateralForOMO.foreach{
    case(id, fraction) =>
      val fractions = ArrayBuffer[Double](fraction)
      if(bank.listOfBonds.nonEmpty           && bank.listOfBonds.contains(id))             fractions += listOfBonds(id)
      if(bank.bondsPledgedAsCollateralForOSLF.nonEmpty && bank.bondsPledgedAsCollateralForOSLF.contains(id)) fractions += bondsPledgedAsCollateralForOSLF(id)
      if(bank.bondsPledgedAsCollateralForIDL.nonEmpty  && bank.bondsPledgedAsCollateralForIDL.contains(id))  fractions += bondsPledgedAsCollateralForIDL(id)
      require(roundTo9Digits(fractions.sum) == 1.0, s"SoB of $bank are not complete: ${fractions.sum} < 1.0")
  }

  bank.bondsPledgedAsCollateralForOSLF.foreach{
    case(id, fraction) =>
      val fractions = ArrayBuffer[Double](fraction)
      if(bank.bondsPledgedAsCollateralForOMO.nonEmpty  && bank.bondsPledgedAsCollateralForOMO.contains(id))  fractions += bondsPledgedAsCollateralForOMO(id)
      if(bank.listOfBonds.nonEmpty           && bank.listOfBonds.contains(id))             fractions += listOfBonds(id)
      if(bank.bondsPledgedAsCollateralForIDL.nonEmpty  && bank.bondsPledgedAsCollateralForIDL.contains(id))  fractions += bondsPledgedAsCollateralForIDL(id)
      require(roundTo9Digits(fractions.sum) == 1.0, s"SoB of $bank are not complete: ${fractions.sum} < 1.0")
  }

  bank.bondsPledgedAsCollateralForIDL.foreach{
    case(id, fraction) =>
      val fractions = ArrayBuffer[Double](fraction)
      if(bank.bondsPledgedAsCollateralForOMO.nonEmpty  && bank.bondsPledgedAsCollateralForOMO.contains(id))  fractions += bondsPledgedAsCollateralForOMO(id)
      if(bank.bondsPledgedAsCollateralForOSLF.nonEmpty && bank.bondsPledgedAsCollateralForOSLF.contains(id)) fractions += bondsPledgedAsCollateralForOSLF(id)
      if(bank.listOfBonds.nonEmpty               && bank.listOfBonds.contains(id))             fractions += listOfBonds(id)
      require(roundTo9Digits(fractions.sum) == 1.0, s"SoB of $bank are not complete: ${fractions.sum} < 1.0")
  }

}
```

```
  def removeStackOfBondsFromGovLOB (ID:Long) {
    sim.government.govLOB.find(_.id == ID) match {
      case Some(sob) => sim.government.govLOB.remove( sim.government.govLOB.indexWhere(_.id == ID) )  //  evtl. filter?
      case None      => sys.error("Cannot remove bond, ID $ID of bond does not exist")
    }
  }

  def PVofOutstandingBonds (t:Int) = sim.government.govLOB.map(sob => PVofSoB(sob, t)).sum




  /** tested
   *  amount of money that would have to be invested today to generate the same future cash flow of the bond
   *  factors: targetRate, maturity, coupon
   *    */
  def PVofSoB (stackOfBonds:sim.government.stackOfBonds, t:Int) = {
    val par        = stackOfBonds.bond.faceValue                                                                      // face value
    val i          = stackOfBonds.bond.couponRate                                                                     // coupon rate
    val y          = sim.centralBank.targetFFR.last                                                                   // annual yield (market)
    val n          = (stackOfBonds.bond.maturity + stackOfBonds.bond.DIC - t) / stackOfBonds.bond.DIC                 // remaining coupons payable to maturity
    val DCS        = if(t % stackOfBonds.bond.DIC == 0) stackOfBonds.bond.DIC else t % stackOfBonds.bond.DIC          // days from last coupon to settlement fo coupon? (day of buy)
    val DIC        = stackOfBonds.bond.DIC.toDouble                                                                   // days in coupon period containing settlement (day of buy)
    val cleanPrice = rounded( ( math.pow( (2+y)/2 , -n+(DCS/DIC) ) * ( y + i*(-1 + math.pow((2+y)/2, n)) ) * par ) / y - (par * i * DCS) / (DIC * 2)  ) // cleanPrice = dirtyPrice - accrued interest
    val dirtyPrice = rounded( ( math.pow( (2+y)/2 , -n+(DCS/DIC) ) * ( y + i*(-1 + math.pow((2+y)/2, n)) ) * par ) / y                                ) // dirtyPrice = cleanPrice + accrued interest
    if(sim.test){
      require(cleanPrice >= 0, s"cleanPrice of PVofSoB cannot be negative: ${cleanPrice} / $t / $y / $stackOfBonds.bond")
      require(dirtyPrice >= 0, s"dirtyPrice of PVofSoB cannot be negative: ${dirtyPrice} / $t / $y / $stackOfBonds.bond")
    }
    cleanPrice * stackOfBonds.amountOfBondsInStack
 // dirtyPrice * stackOfBonds.amountOfBondsInStack
  }



  case class overnightRepo (borrower:BrokerDealer, lender:MMMF, tickOfIssuance:Int, repurchasePrice:Double, amountOfMoney:Double, tradBanks:Boolean = sim.tradBanks) {
    val hairCut_pa        = lender.hairCut
    val overnightFee      = repurchasePrice - amountOfMoney
    val overnightRepoRate = lender.hairCut / 365
    val linkedBondIDs     = Map[Long, Double]()
  }


  /**
  *
  *    */
  def repoRate (borrowedCash:Double, repurchasePrice:Double) = {
    rounded( (repurchasePrice - borrowedCash) / borrowedCash )
  }
```

```

 def annualRepoRate () = 0
 def overnightRepoRate (purchasePrice:Double, annualRepoRate:Double) = (purchasePrice * annualRepoRate) / 365



}// end of Trait bonds







trait hpFilter {

 import java.util._


 /**
  *    This function reads the data to filter and applies the HP-filter on it.
  *
  *    */
 def HPfilterData (inputData:ArrayBuffer[Double], lambda:Double = 100):ArrayBuffer[Double] = {
   val N = inputData.size
   val a = ArrayBuffer[Double]( 1 + lambda, 5 * lambda + 1) ++= Array.fill[Double](N-4)( 6 * lambda + 1) ++= Array[Double]( 5 * lambda + 1, 1 + lambda)
   val b = ArrayBuffer[Double](-2 * lambda                ) ++= Array.fill[Double](N-3)(-4 * lambda    ) ++= Array[Double](-2 * lambda,     0         )
   val c = ArrayBuffer[Double](                           ) ++= Array.fill[Double](N-2)(     lambda    ) ++= Array[Double]( 0,              0         )
   pentas(a, b, c, inputData, N)
 }



 /**
  * This function solves the linear equation system BxX=Y with B being a pentadiagonal matrix.
  */
 def pentas (a:ArrayBuffer[Double], b:ArrayBuffer[Double], c:ArrayBuffer[Double], d:ArrayBuffer[Double], N:Int):ArrayBuffer[Double] = {
   val data = d.clone
   var H1  = 0.0
   var H2  = 0.0
   var H3  = 0.0
   var H4  = 0.0
   var H5  = 0.0
   var HH1 = 0.0
   var HH2 = 0.0
   var HH3 = 0.0
   var HH5 = 0.0
   var Z   = 0.0
   var HB  = 0.0
   var HC  = 0.0

   for(x <- 0 until N){
     Z    = a(x) - H4 * H1 - HH5 * HH2
     HB   = b(x)
```

```
      HH1  = H1
      H1   = (HB - H4 * H2) / Z
      b(x) = H1
      HC   = c(x)
      HH2  = H2
      H2   = HC / Z
      c(x) = H2
      a(x) = (data(x) - HH3 * HH5 - H3 * H4) / Z
      HH3  = H3
      H3   = a(x)
      H4   = HB - H5 * HH1
      HH5  = H5
      H5   = HC
    }
    H2 = 0
    H1 = a(N - 1)
    data(N - 1) = H1
    for(x <- N-2 to 0 by -1){
      data(x) = a(x) - b(x) * H1 - c(x) * H2
      H2 = H1
      H1 = data(x)
    }
    data
  }


}// end of trait IOU
```

A.2 Superclasses

A.2.1 Agent Class

Agent.scala

```
/**
 *
 */
package monEcon

import monEcon.financialSector._

/**
 * @author Sebastian Krug
 *
 */



/**
 *
 * main (super) class that is super class of all other (sub)classes ensuring that they are able to use the code contained in the extended traits (accountManagement, IOU, etc.)
 *
 *    */
class Agent extends accountManagement with IOU with searchAndMatching with simpleRegression with codeProfiling {

}
```

A.2.2 Corporation Class

Corporation.scala

```scala
/**
 *
 */
package monEcon

import collection.mutable._
import monEcon.realSector._

/**
 * @author Sebastian Krug
 *
 */



/**
 *
 *  The corporation class defines attributes and methods that all sub-corporation classes commonly share.
 *
 *    */
class Corporation extends Agent {

  private val _owners = new ArrayBuffer[HH]     // HH that (partially) own the Corporation and receive dividends from it
  private val _profit = new ArrayBuffer[Double] // revenues - COGS

  //  getter
  def owners = _owners
  def profit = _profit

  // setter

}



/**
 *
 *  This class defines the agent that "employs" all unemployed HH through the simulation.
 *
 *    */
case class ARGE() extends Corporation {
  override def toString = "-"
}
```

A.3 Markets

A.3.1 Goods Market Class

GoodsMarket.scala

```scala
/**
 * @author Sebastian Krug
 *
 */

package monEcon.Markets

import monEcon.Agent
import monEcon.Corporation
import monEcon.realSector._
import monEcon.financialSector._
import monEcon.Simulation

import scala.collection.mutable._




/**
 *
 *  The GoodsMarket class (of which only one single agent exists during each simulation) is merely a passive agent that only collects data
 *  concerning the economic activity related to the trade of the good bundle. There is no Walrasian auctioneer or a comparable mechanism in this model.
 *  Trading goods is an entirely decentralized process here.
 *
 *    */
case class GoodsMarket (sim:Simulation, initialPriceOfGood:Double) extends Agent {
  override def toString = "goodsMarket"

  private val _priceIndex             = Map[Corporation, Double]()
  private val _producerPriceLevel     = ArrayBuffer[Double]()
  private val _weightedAvgPriceOfTick = ArrayBuffer[Double](initialPriceOfGood)
  private val _quarterlyOffers        = Map[Corporation, Double]()
  private val _offeredGoods           = Map[Corporation, Double]()
  private val _currentOffers          = Map[Corporation, Offer]()

  // getter
  def priceIndex             = _priceIndex
  def weightedAvgPriceOfTick = _weightedAvgPriceOfTick
  def producerPriceLevel     = _producerPriceLevel
  def quarterlyOffers        = _quarterlyOffers
  def offeredGoods           = _offeredGoods
  def currentOffers          = _currentOffers

  // setter
  def priceIndex_+=   (corp:Corporation, value:Double):Unit = _priceIndex   += (corp -> value)
  def offeredGoods_+= (corp:Corporation, value:Int)   :Unit = _offeredGoods += (corp -> value)


  // -------------------- Methods --------------------

  /**
   *
   *  This method calculates the weighted average price of the good bundle based on the prices of sold goods during the current period/tick.
   *
```

```
   *    */
  def determineWeightedAvgPriceOfTick = {time({
    val price =
      try{
        if(sim.firmList.filter{ _.active }.map(_offeredGoods(_)).sum > 0){
          rounded( sim.firmList.filter{ _.active }.map(firm => _priceIndex(firm) * _offeredGoods(firm)).sum / sim.firmList.filter{ _.active }.map(_offeredGoods(_)).sum )
        } else {
          average(sim.firmList.filter{ _.active }.map(_.price.last))
        }
    } catch {
      case e:Exception => weightedAvgPriceOfTick.last
    }
    if(sim.pln) println(sim.firmList.map(_offeredGoods(_)).sum + "/" + _priceIndex + " price added: " + price)
    if(sim.test) require(!price.isNaN(), "weightedAvgPriceOfTick is NaN and cannot be added")
    weightedAvgPriceOfTick += price
   }, "GM_determineWeightedAvgPriceOfTick", sim)
  }





  /**
   *
   *  This method produces a list of average prices for varying frequencies, i.e. monthly, quarterly, yearly.
   *
   *    */
  def getListOfAvgPrices (frequence:Int):Buffer[Double] = {
    val prices = if(_weightedAvgPriceOfTick.size < frequence) ArrayBuffer(average(_weightedAvgPriceOfTick)) else _weightedAvgPriceOfTick.grouped(frequence).toBuffer.filter(_.size
== frequence).map(average(_))
    if(sim.test) require(!prices.map(_.isInfinite).contains(true), s"list of prices ($frequence) contains Infinity: $prices")
    if(sim.test) require(!prices.map(_.isNaN     ).contains(true), s"list of prices ($frequence) contains NaN: $prices")
    prices
  }



  def weightedAvgPriceOfMonth   = getListOfAvgPrices( 4)      // weighted average monthly   price levels
  def weightedAvgPriceOfQuarter = getListOfAvgPrices(12)      // weighted average quarterly price levels
  def weightedAvgPriceOfYear    = getListOfAvgPrices(48)      // weighted average yearly    price levels




  def determinePriceLevel (p:Seq[Double] = priceIndex.values.filter(_ > 0).toBuffer) = {
    if(p.size > 1){
      _producerPriceLevel += rounded( average(p) )
    } else if(p.nonEmpty){
      _producerPriceLevel += p.head
    } else _producerPriceLevel += 0.0
  }
```

GoodsMarket.scala

```
  def setCurrentSupply = time(offeredGoods.keys.foreach(corp => currentOffers += corp -> Offer(corp, offeredGoods(corp), priceIndex(corp))), "GM_setCurrentSupply", sim)



  case class Offer(vendor:Corporation, quantity:Double, price:Double)



  /**
   *
   *  These values are jsut for data saving purposes.
   *
   *    */
  val goodsMarketEndOfTickData = Map("priceIndex"       -> _priceIndex,
                                     "quarterlyOffers"  -> _quarterlyOffers,
                                     "offeredGoods"     -> _offeredGoods,
                                     "currentOffers"    -> _currentOffers
                                                                              )

  val goodsMarketEndOfSimulationData = Map(
                                           "producerPriceLevel"     -> _producerPriceLevel,
                                           "weightedAvgPriceOfTick" -> _weightedAvgPriceOfTick,
                                           "weightedAvgPriceOfYear" ->  weightedAvgPriceOfYear
                                                                                              )

}// end of class
```

A.3.2 Labor Market Class

LaborMarket.scala

```scala
/**
 * @author Sebastian Krug
 *
 */

package monEcon.Markets

import scala.collection.mutable.Map

import monEcon.Agent
import monEcon.Corporation
import monEcon.Simulation
import monEcon.realSector._



/**
 *
 *  The LaborMarket class (of which only one single agent exists during each simulation) is merely a passive agent that only collects data
 *  concerning HH's search for a job. There is no Walrasian auctioneer or a comparable mechanism in this model.
 *  Searching for a job is an entirely decentralized process here.
 *
 *    */
case class LaborMarket (sim:Simulation) extends Agent {
  override def toString = "laborMarket"

  private val _vacancies:  Map[Corporation, Job] = Map()
  private val _laborDemand:Map[Firm, Double]     = Map()
  private val _wageFactors:Map[Firm, Double]     = Map()


  // getter
  def vacancies   = _vacancies
  def laborDemand = _laborDemand
  def wageFactors = _wageFactors

  // setter
  def vacancies_+= (firm:Firm, value:Job)    :Unit = _vacancies   += (firm -> value)
  def wages_+=     (firm:Firm, value:Double) :Unit = _wageFactors += (firm -> value)



  case class Job(laborDemand:Double, wageFactor:Double)


  /**
   *
   *  These values are jsut for data saving purposes.
   *
   *    */
  val laborMarketEndOfTickData = Map("laborDemand"  -> laborDemand,
                                     "wageFactors"  -> wageFactors
                                                              )

```

```
}
```

A.4 Financial Sector

A.4.1 Bank Class

Bank.scala

```
/**
 * @author Sebastian Krug
 * @constructor
 * @param name
 * @param numberOfHH
 *
 */

package monEcon.financialSector

import monEcon.Corporation
import monEcon.bonds
import monEcon.realSector._
import monEcon.publicSector._
import monEcon.publicSector.Supervisor
import monEcon.Markets._
import monEcon.PaymentSystem
import monEcon.Simulation

import collection.mutable._
import collection.immutable.SortedMap

import scala.util.Random


/**
 *
 *
 */


// ---------- Class for Bank-Objects ----------
case class Bank (name              :String,                     //
                 fractionOfDebtBank:Double,                     //
                 random            :Random,                     //
                 CB                :CentralBank,                //
                 IBM               :InterbankMarket,            //
                 sim               :Simulation                  //
                                                                                             ) extends Corporation with bonds {

  override def toString = s"Bank($name)"

  /*    ----------------------------------------    bank balance sheet positions    ----------------------------------------    */
  // ----- Asset Side -----
  private val _businessLoans        = ArrayBuffer(0.0)
  private val _interbankLoans       = ArrayBuffer(0.0)
  // private val bonds              = ArrayBuffer(0.0)
  private val _interestReceivables  = ArrayBuffer(0.0)
  private val _OSDF                 = ArrayBuffer(0.0)
  private val _cbReserves           = ArrayBuffer(0.0)
  private val _totalAssets          = ArrayBuffer[Double]()

  // ----- Liabilities Side -----
  private val _retailDeposits       = ArrayBuffer(0.0)
```

```scala
private val _govDeposits          = ArrayBuffer(0.0)
private val _cbLiabilities        = ArrayBuffer(0.0)
private val _interbankLiabilities = ArrayBuffer(0.0)
private val _equity               = ArrayBuffer[Double]()


val bankBSP                  = Map("businessLoans"        -> _businessLoans,              // this is just for io-reasons, i.e. saving of bank data.
                                   "interbankLoans"       -> _interbankLoans,
                                   "bonds"                -> bonds,
                                   "interestReceivables"  -> _interestReceivables,
                                   "OSDF"                 -> _OSDF,
                                   "cbReserves"           -> _cbReserves,
                                   "retailDeposits"       -> _retailDeposits,
                                   "govDeposits"          -> _govDeposits,
                                   "cbLiabilities"        -> _cbLiabilities,
                                   "interbankLiabilities" -> _interbankLiabilities
                                   "totalAssets"          -> _totalAssets,
                                   "equity"               -> _equity
                                                                                                                                  )

/**
 *
 *  method for testing of bank balance sheets
 *
 *    */
def checkDeposits = if(sim.test){
  require(
      sim.bankList.map(_.retailDeposits.last).sum == sim.firmList.map(_.bankDeposits.last).sum + sim.hhList.map(_.bankDeposits.last).sum,
      s"retailDeposits are not correct: ${sim.bankList.map(_.retailDeposits.last).sum} / ${sim.firmList.map(_.bankDeposits.last).sum + sim.hhList.map(_.bankDeposits.last).sum}"
    )
}




// other data
private var _active                              = true
private var _periodOfReactivation                = 0
private var _age                                 = 0
private val _insolvencies                        = ArrayBuffer[Int](0)
private val _bailOutCounter                      = Map[Int, Double]()
private val _loanLosses                          = ArrayBuffer[Double](0.0)
private val _businessClients                     = ArrayBuffer[Firm]()
private val _MMMFClients                         = ArrayBuffer[MMMF]()
private val _BDClients                           = ArrayBuffer[BrokerDealer]()
private val _retailClients                       = ArrayBuffer[HH]()
private val _goods2Liquidate                     = Map[Firm, (Double, Double)]()

/*    -----   Refinancing and MP    -----   */
// 1. monthly OMO/repo to meet target        -> get, repay done
private val _reserveTarget                    = ArrayBuffer[Double]()
private val _outstandingOMOpayables           = Queue[OMO]()
// 2. overnight IBM loans (banks set int)    -> get and repay done
private val _outstandingIBMpayables           = Queue[IBMloan]()
```

```scala
private val _outstandingIBMreceivables       = Map[Bank, IBMloan]()
private val _reservesCurrentlyOfferedOnIBM    = Map[Bank, (Double,Double)]()
// 3. overnight OSF loans                   -> get/use done, repay done
private val _outstandingOSLFpayables          = Queue[OvernightOSLFloan]()
private val _OSFused                          = ArrayBuffer[(Double, Double)]()
private var _interestOnOSDFrepos              = 0.0
// 4. IDL (free of charge)                   -> get and repay done
private var _borrowedIntraDayLiquidity        = 0.0
private var _bondsAddedWithBondRelationship   = 0

// interest spread
private val _interestOnRetailDeposits         = ArrayBuffer[Double]()
private val _interestOnRetailLoans            = ArrayBuffer[Double]()
private val _interestOnInterbankLoans         = ArrayBuffer[Double]()
private val _riskPremium4DoubtfulCredits      = ArrayBuffer[Double]()

// bank performance
private val _NIM                              = ArrayBuffer[Double]()
private val _ROE                              = ArrayBuffer[Double]()
private val _ROA                              = ArrayBuffer[Double]()
private val _earnings                         = ArrayBuffer[Double]()
private val _COGS                             = ArrayBuffer[Double]()
private val _marketShare                      = ArrayBuffer[Double]()

// regulatory data
private val _RWA                              = ArrayBuffer[Double]()
private val _equityRatio                      = ArrayBuffer[Double]()
private val _equityOfRWA                      = ArrayBuffer[Double]()

// test
val test                                      = ArrayBuffer[Long]()
val _equityAfterReactivation                  = ArrayBuffer[Double]()
val _tickOfInsolvency                         = ArrayBuffer[Int]()

// getter
def active                         = _active
def periodOfReactivation           = _periodOfReactivation
def age                            = _age
def bondsAddedWithBondRelationship = _bondsAddedWithBondRelationship
def insolvencies                   = _insolvencies
def bailOutCounter                 = _bailOutCounter
def loanLosses                     = _loanLosses
def interestOnRetailDeposits       = _interestOnRetailDeposits
def interestOnRetailLoans          = _interestOnRetailLoans
def interestOnInterbankLoans       = _interestOnInterbankLoans
def riskPremium4DoubtfulCredits    = _riskPremium4DoubtfulCredits

  // BSP
def businessLoans                  = _businessLoans
def interbankLoans                 = _interbankLoans
def interestReceivables            = _interestReceivables
def OSDF                           = _OSDF
def cbReserves                     = _cbReserves
def totalAssets                    = _totalAssets
```

```scala
def retailDeposits                  = _retailDeposits
def govDeposits                     = _govDeposits
def cbLiabilities                   = _cbLiabilities
def interbankLiabilities            = _interbankLiabilities
def equity                          = _equity

// other data
def listOfDebtors                   = _listOfDebtors
def businessClients                 = _businessClients
def MMMFClients                     = _MMMFClients
def BDClients                       = _BDClients
def retailClients                   = _retailClients
def goods2Liquidate                 = _goods2Liquidate
def earnings                        = _earnings
def COGS                            = _COGS
def reservesCurrentlyOfferedOnIBM  = _reservesCurrentlyOfferedOnIBM
def reserveTarget                   = _reserveTarget
def borrowedIntraDayLiquidity       = _borrowedIntraDayLiquidity
def OSFused                         = _OSFused

def RWA                             = _RWA
def equityRatio                     = _equityRatio
def equityOfRWA                     = _equityOfRWA
def outstandingIBMpayables          = _outstandingIBMpayables
def outstandingIBMreceivables       = _outstandingIBMreceivables
def interestOnOSDFrepos             = _interestOnOSDFrepos
def NIM                             = _NIM
def ROE                             = _ROE
def ROA                             = _ROA
def marketShare                     = _marketShare
def excessReserves                  = _excessReserves
def minReserves                     = _minReserves
def reserveDeficit                  = _reserveDeficit
def currentAvgReserves              = _currentAvgReserves




def finishTick (t:Int) = {
  if(sim.pln) println(" ----- Banks make annual report ----- ")
  makeAnnualReport(t)
}


def updateBankAge   = _age += 1
def updateBondsAddedWithRelationship (i:Int) = _bondsAddedWithBondRelationship += i

def foundMe (investment:Double) {
  val amountOfBonds = (roundUpXk(investment,sim.faceValueOfBonds)/sim.faceValueOfBonds - 1).toInt
  val newSoB = sim.government.stackOfBonds(amountOfBonds)
  sim.government.govLOB += newSoB
  listOfBonds           += newSoB.id -> 1.0
  sim.government.addPublicDebt4Repayment(this, newSoB)
}
```

```


  /*    --------------------------------------------------------------- Bank Refinancing ------------------------------------------------------
   *
   *    The following methods are written to enable bank agents to interact with the CB and among each other for refinancing purposes throughout the settlement day.
   *
   *      */
  def _currentReserveTarget  = CB.reserveTargetBalances(this).reserveTargetBalance
  def _excessReserves:Double = rounded( math.max(0, _cbReserves.grouped(4).toBuffer.takeRight(1)(0).map(_ - CB.reserveTargetBalances(this).upperBound).sum) )    // reserves above upperbound (on avg)
  def _reserveDeficit        = rounded( math.max(0, _cbReserves.grouped(4).toBuffer.takeRight(1)(0).map(CB.reserveTargetBalances(this).lowerBound - _).sum) )    // reserves below lowerBonud (on avg)
  def _currentAvgReserves    = average( _cbReserves.grouped(4).toBuffer.last )

  def _minReserves:Double    = rounded( CB.minReserveRequirement.last * (_retailDeposits.last + _govDeposits.last) )
    def debt2EquityRatio     = {
      if(_equity.nonEmpty){
        _equity.last match {
          case equity:Double if equity == 0 => if((_retailDeposits.last + _govDeposits.last + _interbankLiabilities.last + _cbLiabilities.last) == 0) 0.0 else 100
          case equity:Double if equity  > 0 =>    (_retailDeposits.last + _govDeposits.last + _interbankLiabilities.last + _cbLiabilities.last) / _equity.last
          case _                            => 100
        }
      } else 0.0
  }


  /**
   *  using this method, banks can check how much (excess) reserves are currently offered on the interbank market
   *
   *    */
  def currentlyOfferedReservesOnIBM :Map[Bank, (Double,Double)] = {
    _reservesCurrentlyOfferedOnIBM.clear
    sim.bankList.filter(_.active).foreach{
      bank =>
        val interestChargedOnBorrowingBank = bank.interestOnIBMLoans(this)
        val currentOffer                   = ( if(interestChargedOnBorrowingBank > CB.depositFacilityRate.last) math.min(bank._excessReserves, bank._cbReserves.last) else 0, interestChargedOnBorrowingBank )
        if(currentOffer._1 > 0) _reservesCurrentlyOfferedOnIBM += bank -> currentOffer
    }
    _reservesCurrentlyOfferedOnIBM
  }



  /**
   *
   *  banks pledge collateral at the ECB as part of the repo agreement in order to receive reserves
   *
   *    */
  def pledgeCollateral (map2PutBonds:Map[Long, Double], amount:Double, t:Int):Unit = {time({
    if(amount > 0){
      val cPVPB = currentPVofPledgeableBonds(t)
      if(cPVPB < amount) sim.government.issueNewGovBonds(this, amount - cPVPB, t)
      val testPVbefore  = if(sim.test) currentPVofSoBs(t) else 0.0
```

```
      var amount2Pledge = amount
      var loopCounter   = 0
      do{
        if(listOfBonds.isEmpty) sim.government.issueNewGovBonds(this, amount2Pledge, t)
        val SoB    = listOfBonds.head
        val PV_SoB = PVofSoB(sim.government.findStackOfBondsByID(SoB._1), t)
        val fractionOfStack2Pledge:Double = amount2Pledge / ( PV_SoB * SoB._2)
        if(map2PutBonds.contains(SoB._1)) map2PutBonds(SoB._1) += math.min(fractionOfStack2Pledge * SoB._2, SoB._2) else map2PutBonds += SoB._1 -> math.min(fractionOfStack2Pledge * SoB._2, SoB._2)
        roundTo9Digits(map2PutBonds(SoB._1))
        val updatedFractionLOB = SoB._2 - math.min(fractionOfStack2Pledge * SoB._2, SoB._2)
        if(fractionOfStack2Pledge >= 1) listOfBonds -= SoB._1 else listOfBonds += SoB._1 -> updatedFractionLOB
        amount2Pledge -= (PV_SoB * SoB._2)
        loopCounter   += 1
      }while(amount2Pledge > 0)
    }
   }, "bank_pledgeCollateral", sim)
  }




  /**
   *
   *  re-buy of the collateral (part of the repo agreement)
   *
   *    */
  def dePledgeCollateral (map2TakeBonds:Map[Long, Double]):Unit = {
    map2TakeBonds.foreach{ case(id:Long, fraction:Double) => if(listOfBonds.contains(id)) listOfBonds(id) += fraction else listOfBonds += id -> fraction }
    map2TakeBonds.clear()
  }







  /*     -----     Phase A. Begin of Settlement Day     -----     */
  /**
   *
   *  bank agents check whether they have to repay interbank loans from the previous period.
   *
   *    */
  def repayIBMloans (t:Int) {time({
     while(_outstandingIBMpayables.nonEmpty){
       val loanToRepay = _outstandingIBMpayables.dequeue
       if(loanToRepay.lendingBank.active == true){
         if(sim.pln){
           println(s"$this repays ${loanToRepay.amountOfReserves} plus interest of ${rounded(loanToRepay.amountOfReserves * (loanToRepay.interest/360))} to ${loanToRepay.lendingBank} to settle its overnight IBM loan.")
         }
         if(sim.test) require(loanToRepay.borrowingBank == this, "IBMloan has not the right borrowingBank. Check repayIBMloans...")
         transferMoney(this, loanToRepay.lendingBank, loanToRepay.amountOfReserves, "repayOvernightIBMloan", sim, t, rounded(loanToRepay.amountOfReserves * (loanToRepay.interest/
```

```
360)))
        loanToRepay.lendingBank.outstandingIBMreceivables -= this
        if(sim.pln) println(s"$this's reserve account: ${_cbReserves.last}")
        if(_cbReserves.last < 0.0) getIntraDayLiquidity(-_cbReserves.last, t, "nonNegativeReserveAccount")
        if(sim.test) require(_cbReserves.last >= 0.0, s"$this has negative reserve account after repayIBMloans: ${_cbReserves.last}")
      } else {
        if(sim.pln){
          println(s"$this repays ${loanToRepay.amountOfReserves} plus interest of ${rounded(loanToRepay.amountOfReserves * (loanToRepay.interest/360))} to ${loanToRepay.lendingBank} to settle its overnight IBM loan.")
        }
        if(sim.test) require(loanToRepay.borrowingBank == this, "IBMloan has not the right borrowingBank. Check repayIBMloans...")
        transferMoney(this, loanToRepay.lendingBank, loanToRepay.amountOfReserves, "cleanOvernightIBMloan2InsolventBank", sim, t, rounded(loanToRepay.amountOfReserves * (loanToRepay.interest/360)))
      }
    }// while
    if(sim.test) require(_outstandingIBMpayables.isEmpty, s"_outstandingIBMpayables is not empty: ${_outstandingIBMpayables}")
  }, "bank_repayIBMloans", sim)}







  /**
   *
   *  bank agents check whether they have to settle (overnight) liquidity agreements from the standing facility of the CB
   *
   *    */
   def repayOSF (t:Int):Unit = {time({
     if(sim.test) require(_outstandingOSLFpayables.size < 2, s"There are more than one outstanding OvernightOSLFloans, but there should only be one! Check repayOSF & useOSF: ${_outstandingOSLFpayables}")
     val PVB = if(sim.test) currentPVofSoBs(t) else 0.0
     if(sim.test) require(    _claimsFromOSDFrepos.size < 2, s"There are more than one claims from OSDF repos, but there should only be one! Check repayOSF & useOSF: ${_claimsFromOSDFrepos}")
     if(_outstandingOSLFpayables.nonEmpty && _OSDF.last == 0){
       if(sim.test) require(_outstandingOSLFpayables.size == 1, "There are more than one OSLFpayables.")
       val overnightOSLFloanToRepay = _outstandingOSLFpayables.dequeue
       if(sim.test) require(_outstandingOSLFpayables.isEmpty, "OSLFpayables is not empty.")
       if(sim.test) require(CB.outstandingOSLFreceivables.contains(this))
       if(sim.test) require(overnightOSLFloanToRepay.amountOfReserves == CB.outstandingOSLFreceivables(this).amountOfReserves, "Amount to repay of OvernightOSLFloan is not the same at bank & CB.")
       transferMoney(this, CB, overnightOSLFloanToRepay.amountOfReserves, "repayOSLF", sim, t, overnightOSLFloanToRepay.amountOfReserves * (overnightOSLFloanToRepay.interest/360) )
       if(sim.pln){
         println(s"$this has repaid its OSLF payable of ${overnightOSLFloanToRepay.amountOfReserves} plus interest of ${overnightOSLFloanToRepay.amountOfReserves * (overnightOSLFloanToRepay.interest/360)} to the CB; cbLiabs: ${_cbLiabilities.last}.")
       }
       CB.outstandingOSLFreceivables -= this
       dePledgeCollateral(bondsPledgedAsCollateralForOSLF)
       if(sim.test) require(bondsPledgedAsCollateralForOSLF.isEmpty, "bondsPledgedAsCollateralForOSLF is not empty after repyOSF.")
       if(sim.test) require(!listOfBonds.contains(null), s"listOfBonds of $this contains null")
       if(_cbReserves.last < 0.0) getIntraDayLiquidity(-_cbReserves.last, t, "nonNegativeReserveAccount")
       if(sim.test) require(_cbReserves.last >= 0.0, s"$this has negative reserve account after repayOSF: ${_cbReserves.last}")
       if(sim.test) require(SEc(PVB, currentPVofSoBs(t), 5), s"currentPVofBonds of $this is not correct")
     } else if(_outstandingOSLFpayables.isEmpty && _OSDF.last > 0){
```

```scala
      if(sim.pln) println(s"$this gets back its overnight OSDF deposits at the CB of ${_OSDF.last}")
      transferMoney(CB, this, _OSDF.last, "repayOSDF", sim, t, _interestOnOSDFrepos)
      _interestOnOSDFrepos = 0.0
    } else if(_outstandingOSLFpayables.isEmpty && _OSDF.last == 0){
      if(sim.pln) println(s"$this does not need to repayOSF since hasn't used it: ${_OSFused}")                                    // OSF not used at all
    } else sys.error(s"$this cannot lend and deposit at the CB at the same time. Check repayOSF and useOSFifNecessary: OSLF ${_outstandingOSLFpayables} / OSDF ${_OSDF.last}")
  }, "bank_repayOSF", sim)
 }

 /**
  *
  *
  *  bank agents perform a frequent repo with the CB to meet their (individual) reserve target
  *
  *    */
 def monthlyRepoToAquireTargetReserve (t:Int, amountOfReservesNeeded:Double = CB.reserveTargetBalances(this).reserveTargetBalance):Unit = {time({
   if(_outstandingOMOpayables.isEmpty){
     pledgeCollateral(bondsPledgedAsCollateralForOMO, amountOfReservesNeeded, t)
     transferMoney(CB, this, amountOfReservesNeeded, "OMO", sim, t, amountOfReservesNeeded * (CB.RePoRate.last/12))
     if(sim.pln){
       println(s"$this has conducted monthly repo according to its current reserve target (${CB.reserveTargetBalances(this).reserveTargetBalance}) and the CB supplies $amountOfReservesNeeded of reserves; cbLiabs: ${_cbLiabilities.last}.")
     }
     if(sim.test) require(!CB.outstandingOMOreceivabels.contains(this), s"$this cannot get another OMO, it already exists in CB.outstandingOMOreceivables")
     CB.outstandingOMOreceivabels += this -> CB.OMO(this, amountOfReservesNeeded, CB.RePoRate.last, t+4)
     _outstandingOMOpayables.enqueue(OMO(this, amountOfReservesNeeded, CB.RePoRate.last, t+4))
     if(sim.test) require(_cbReserves.last >= _currentReserveTarget, s"$this does not have enough reserve in its accounts to meet its target after OMO: ${_cbReserves.last} / ${_currentReserveTarget}")
     if(sim.test) require(PV_OMO(t) >= _currentReserveTarget, "PV of Bonds in bondsPledgedAsCollateralForOMO is not enough to secure monthly repo with CB.")
   } else {
     println(s"++++++++++++++++++++++++ $this [seed ${sim.seed}/t=$t] does its monthly repo with the CB ++++++++++++++++++++++++")
     if(sim.test) require(_outstandingOMOpayables.size < 2, "There are more than the outstanding OMO payable from last period.")
     println(s"$this repays its monthly repo of previous period (${_outstandingOMOpayables.head.amountOfReserves} + interest of ${_outstandingOMOpayables.head.amountOfReserves * (_outstandingOMOpayables.head.interest/12)}) to the CB.")
     val currentPVofBondsBeforeRepaymentOfMonthlyRepo = if(sim.test) currentPVofSoBs(t) else 0.0
     println(s"currentPVofBonds of $this before repay of monthly repo: $currentPVofBondsBeforeRepaymentOfMonthlyRepo")
     printBSP
     transferMoney(this, CB, _outstandingOMOpayables.head.amountOfReserves, "repayMonthlyOMO", sim, t, _outstandingOMOpayables.head.amountOfReserves * (_outstandingOMOpayables.head.interest/12))
     println(s"$this has repaid monthly repo (OMO) (${_outstandingOMOpayables.head.amountOfReserves} + ${_outstandingOMOpayables.head.amountOfReserves * (_outstandingOMOpayables.head.interest/12)}) to the CB leading to a reserve account of ${_cbReserves.last} and cbLiabs of ${_cbLiabilities.last}.")
     dePledgeCollateral(bondsPledgedAsCollateralForOMO)
     val OMOofLastPeriod = _outstandingOMOpayables.dequeue
     CB.outstandingOMOreceivabels -= this
     if(sim.test) require(_outstandingOMOpayables.isEmpty, "_outstandingOMOpayables is not empty.")
     val currentPVofBondsAfterRepaymentAndBeforePlacementOfNewMonthlyRepo = if(sim.test) currentPVofSoBs(t) else 0.0
```

```
    println(s"currentPVofBonds of $this after repay and before placement of new monthly repo: $currentPVofBondsAfterRepaymentAndBeforePlacementOfNewMonthlyRepo")

    if(_cbReserves.last < 0.0) getIntraDayLiquidity(-_cbReserves.last, t, "nonNegativeReserveAccount")
    if(sim.test) require(_cbReserves.last >= 0.0, s"$this has negative reserve account after repayOSF: ${_cbReserves.last}")
    pledgeCollateral(bondsPledgedAsCollateralForOMO, amountOfReservesNeeded, t)
    if(sim.test) require(!listOfBonds.contains(null), s"listOfBonds of $this contains null")
    if(sim.test){
      require(
          collateral.map(stackOfBonds => stackOfBonds._1 * PVofBond(stackOfBonds._2, t) ).sum >= _currentReserveTarget - math.max(1, _currentReserveTarget * 0.000001),
          s"Provided collateral of $this for OMO is insufficient: ${collateral.map(stackOfBonds => stackOfBonds._1 * PVofBond(stackOfBonds._2, t) ).sum} < ${_currentReserveTarget - _currentReserveTarget * 0.000001}"
        )
    }
    if(sim.test) require(!collateral.contains(null), s"collateral of $this contains null")
    if(sim.test) require(!bondsPledgedAsCollateralForOMO.contains(null), s"bondsPledgedAsCollateralForOMO of $this contains null")
    transferMoney(CB, this, amountOfReservesNeeded, "OMO", sim, t, amountOfReservesNeeded * (CB.RePoRate.last/12))
    println(s"$this has conducted monthly repo according to its current reserve target (${_currentReserveTarget}) and the CB supplies $amountOfReservesNeeded of reserves leading to a reserve account of ${_cbReserves.last}; cbLiabs: ${_cbLiabilities.last}.")
    if(sim.test) require(!CB.outstandingOMOreceivabels.contains(this), s"$this cannot get another OMO, it already exists in CB.outstandingOMOreceivables")
    CB.outstandingOMOreceivabels += this -> CB.OMO(this, amountOfReservesNeeded, CB.RePoRate.last, t+4)
    _outstandingOMOpayables.enqueue(OMO(this, amountOfReservesNeeded, CB.RePoRate.last, t+4))
    if(sim.test){
      require(PV_OMO(t)        >= _currentReserveTarget - math.max(1, _currentReserveTarget * 0.000001),
          s"PV of Bonds in bondsPledgedAsCollateralForOMO is not enough to secure monthly repo with CB: ${PV_OMO(t)} - ${_currentReserveTarget} = ${PV_OMO(t) - _currentReserveTarget}."
        )
    }
    if(sim.test){
      require(
          _cbReserves.last >= _currentReserveTarget - math.max(1, _currentReserveTarget * 0.000001),
          s"$this does not have enough reserve in its accounts to meet its target after OMO: ${_cbReserves.last} / ${_currentReserveTarget}"
        )
    }
    val currentPVofBondsAfterPlacementOfNewMonthlyRepo = if(sim.test) currentPVofSoBs(t) else 0.0
      println(s"currentPVofBonds of $this after placement of monthly repo: $currentPVofBondsAfterPlacementOfNewMonthlyRepo")
    if(sim.test){
      require(
          SEc(currentPVofBondsBeforeRepaymentOfMonthlyRepo, currentPVofBondsAfterRepaymentAndBeforePlacementOfNewMonthlyRepo, 5),
          s"PV of bonds changed during monthly repo: PVbeforeRepayment ($currentPVofBondsBeforeRepaymentOfMonthlyRepo) / in between ($currentPVofBondsAfterRepaymentAndBeforePlacementOfNewMonthlyRepo)"
        )
    }
    println(s"++++++++++++++++++++++++ $this [seed ${sim.seed}/t=$t] is done with monthly repo ++++++++++++++++++++++++")
    if(sim.test){
      require(
          bondsPledgedAsCollateralForOMO.map(stackOfBonds => stackOfBonds._1 * PVofBond(stackOfBonds._2, t)).sum >= deficitAboveTarget + _currentReserveTarget,
          "PV of Bonds in bondsPledgedAsCollateralForOMO is not enough to secure monthly repo with CB."
        )
    }
    if(sim.pln){
      println(s"$this has not enough collateral (${currentPVofPledgeableBonds(t)}) to place with the CB to conduct monthly OMO to meet deficit before target ($deficitAboveTarget) and new target (${_currentReserveTarget})")
    }
    if(sim.test) require(!listOfBonds.contains(null), s"listOfBonds of $this contains null")
```

```
    }

   }, "bank_monthlyRepoToAquireTargetReserve", sim)
  }

  /*    -----    Phase B. During Settlement Day    -----    */
  /**
   *
   *   If bank agents run out of reserves during the settlement day they can borrow sufficient amounts of reserves in order to be able to settle they accounts with other banks on a gross basis and in real-time
   *   as intended in RGTS payment systems.
   *   As for the majority of payment systems over the world (i.e. except for the U.S. Fedwire), intra day liquidity is provided by the CB at no charge since banks, in turn, have to pledge sufficient
   *   amounts of collateral at the CB.
   *
   *    */
  def getIntraDayLiquidity (amountToTransfer:Double, t:Int, cause:String = "IDL", test:Boolean = true):Unit = {time({
    if(sim.test) require(_cbReserves.last < amountToTransfer)
    val deficit = if(cause == "nonNegativeReserveAccount") amountToTransfer else if(cause == "IDL") amountToTransfer - _cbReserves.last else sys.error("wrong cause")
    pledgeCollateral(bondsPledgedAsCollateralForIDL, deficit, t)
    val ra = if(sim.pln) _cbReserves.last else 0.0
    transferMoney(CB, this, deficit, "provideIDL", sim, t)
    if(sim.pln){
      if(cause == "nonNegativeReserveAccount"){
        println(s"$this has requested IDL of $deficit inorder to settle its negative reserve account of $ra (after getting IDL ${_cbReserves.last}); cbLiabs: ${_cbLiabilities.last}.")
      } else {
        println(s"$this has requested IDL of $deficit since it has to transfer $amountToTransfer but only had an reserve account of $ra (now ${_cbReserves.last}) leading to a deficit of $deficit; cbLiabs: ${_cbLiabilities.last}.")
      }
    }
    if(CB.intraDayLiquidity.contains(this)) CB.intraDayLiquidity(this) += deficit else CB.intraDayLiquidity += this -> deficit
    _borrowedIntraDayLiquidity += deficit
    if(sim.test) require(PVB == currentPVofBonds(t), s"currentPVofBonds of $this is not correct.")
   }, "bank_getIntraDayLiquidity", sim)
  }
```

```
  /**
   *
   *   For their monthly repo with the CB, bank agents make a guess for their requested amount of reserves of the upcoming maintenance period (i.e. month) based on their current interest-bearing deposits.
   *
   *    */
  def setReserveTarget:Unit = {
    val interestBearingDeposits = _retailDeposits.last + _govDeposits.last
    if(sim.test) require(interestBearingDeposits >= 0, s"interestBearingDeposits of $this: ${_retailDeposits.last} + ${_govDeposits.last} = $interestBearingDeposits must be non-negative.")
    CB.reserveTargetBalances += this -> CB.ReserveTarget( roundUpTo10k( math.max(interestBearingDeposits * CB.minReserveRequirement.last, interestBearingDeposits / 15) ) )
    _reserveTarget += CB.reserveTargetBalances(this).reserveTargetBalance
    if(sim.pln){
      println(s"$this sets a reserve target of ${CB.reserveTargetBalances(this).reserveTargetBalance} sine it has interest bearing deposits of $interestBearingDeposits with is ${rounded((CB.reserveTargetBalances(this).reserveTargetBalance/interestBearingDeposits) * 100)}% (required: ${rounded(100.0/15)}%)")
    }
  }









  /*     -----     Phase C. End of Settlement Day     -----     */

  /**
   *
   *   bank agents have to repay the daylight loan at the end of the settlement day meaning that their intra-settlement-day repos with the CB have to be settled at this stage.
   *
   *    */
  def repayIDL (t:Int, borrowedReserves:Double = _borrowedIntraDayLiquidity) {time({
    if(borrowedReserves > 0){
      if(sim.test) require(_borrowedIntraDayLiquidity == sim.IDLflows(this).last, s"IDL to repay are not correct: _borrowedIntraDayLiquidity ${_borrowedIntraDayLiquidity} / IDLflows ${sim.IDLflows(this).last}")
      if(sim.pln) println(s"$this has _borrowedIntraDayLiquidity of ${_borrowedIntraDayLiquidity} / CB.intraDayLiquidity of ${CB.intraDayLiquidity} / IDLfows of ${sim.IDLflows(this)} so it must repay IDL")
      if(sim.test) require(borrowedReserves == CB.intraDayLiquidity(this), s"Amount of borrowed IDL is not correct ($borrowedReserves / ${CB.intraDayLiquidity(this)} ). Check bank.repayIDL of $this")
      val PVB = if(sim.test) currentPVofSoBs(t) else 0.0
      dePledgeCollateral(bondsPledgedAsCollateralForIDL)
      if(sim.test){
        require(!listOfBonds.contains(null), s"listOfBonds of $this contains null")
        require(!bondsPledgedAsCollateralForIDL.contains(null), s"bondsPledgedAsCollateralForIDL of $this contains null")
        require(bondsPledgedAsCollateralForIDL.isEmpty, s"bondsPledgedAsCollateralForIDL is not empty after bank.repayIDL: $bondsPledgedAsCollateralForIDL")
      }
      transferMoney(this, CB, borrowedReserves, "repayIDL", sim, t)
      if(sim.pln) println(s"$this has repaid IDL of $borrowedReserves (current reserve account after repay IDL: ${_cbReserves.last}; cbLiabs: ${_cbLiabilities.last})")
      CB.intraDayLiquidity       -= this
```

```
      _borrowedIntraDayLiquidity  = 0.0
     if(sim.test) require(SEc(PVB, currentPVofSoBs(t), 5), s"currentPVofBonds of $this is not correct: $PVB / ${currentPVofSoBs(t)}")
    }
   }, "bank_repayIDL", sim)
 }












  /**
   *
   *   If bank agents face a reserve deficit depending on their current (individual) average reserve account at the CB, they usually try to
   *   reallocate outstanding reserves in the system by interbank lending of reserves. Of course, a situation of low or even no supply of
   *   excess reserves by other banks is possible. In such a situation, bank agents have the incetive to use the standing facilities of the CB.
   *
   *    */
   def lendOvernightFromIBM (t:Int) {time({
     val reservesOffered = currentlyOfferedReservesOnIBM
     if(reservesOffered.nonEmpty){
       if(sim.pln) println(reservesOffered.map(offer => (offer._1, offer._1.cbReserves.last) -> offer._2))
       val acceptableOffers = reservesOffered.filter(_._2._2 < CB.lendingFacilityRate.last)
       if(acceptableOffers.nonEmpty){
         var reservesToBorrow = _reserveDeficit
         while(reservesToBorrow > 0 && acceptableOffers.nonEmpty){
           val bestOffer         = acceptableOffers.minBy(_._2._2)
           val borrowedReserves  = math.min(reservesToBorrow, bestOffer._2._1)
           val overnightLoan     = IBMloan(bestOffer._1, this, borrowedReserves, bestOffer._2._2)
           _outstandingIBMpayables.enqueue( overnightLoan )
           bestOffer._1.outstandingIBMreceivables += this -> overnightLoan.asInstanceOf[bestOffer._1.IBMloan]
           if(CB.reservesLendOvernightOnIBM.contains(t)) CB.reservesLendOvernightOnIBM(t) += overnightLoan.asInstanceOf[CB.IBMloan] else CB.reservesLendOvernightOnIBM += t -> ArrayBuffer(overnightLoan.asInstanceOf[CB.IBMloan])
           if(sim.pln){
             println(s"$this lends $borrowedReserves (+ ${rounded(borrowedReserves * (bestOffer._2._2/360))} interest) overnight from ${bestOffer._1} because it has a reserve deficit of ${_reserveDeficit} resulting from ${_cbReserves.grouped(4).toBuffer.takeRight(1)(0)}")
           }
           transferMoney(bestOffer._1, this, borrowedReserves, "overnightIBMloan", sim, t, rounded(borrowedReserves * (bestOffer._2._2/360)))
           reservesToBorrow -= borrowedReserves
           acceptableOffers -= bestOffer._1
         }// while
       } else if(sim.pln) println(s"$this wants to borrow on IBM but the offered interests are too high compared to the OSLF rate of ${CB.lendingFacilityRate.last}")
     }
    }, "bank_lendOvernightFromIBM", sim)
   }
```

```scala
/**
 *
 *  If bank agents with a reserve deficit weren't able to borrow\lend a sufficient amount of reserves from/to peers,
 *  they use the lending (OSLF) or deposit (OSDF) facility of the CB depending on their current avg. reserve accounts.
 *
 *    */
 def useOSFifNecessary (t:Int):Unit = {time({
   (t-1) % 4 match {
     case 0 =>
       _OSFused                += {(0.0, 0.0)}        // do not use OSF at all since it is too early
    assureNonNegativeReserveAccount   // lend if reserve account is negative

     case 1 =>
       assureNonNegativeReserveAccount
       if(random.nextDouble < 0.10) if(_reserveDeficit > 0) useOSLF() else if(_excessReserves > 0) useOSDF

     case 2 =>
       assureNonNegativeReserveAccount
       if(random.nextDouble < 0.5) if(_reserveDeficit > 0) useOSLF() else if(_excessReserves > 0) useOSDF

     case 3 =>
       assureNonNegativeReserveAccount
       if(_reserveDeficit > 0) useOSLF() else if(_excessReserves > 0) useOSDF

     case _ =>
       assureNonNegativeReserveAccount
   }
   if(sim.test) require(PVB == currentPVofBonds(t), s"currentPVofBonds of $this is not correct.")


   def assureNonNegativeReserveAccount:Unit = {
     if(_cbReserves.last < 0.0){
       val amountNeeded = -_cbReserves.last
       if(sim.pln) println(s"$this is using the OSLF since it has a negative reserve account (${_cbReserves.last}) and borrows ${amountNeeded}")
       useOSLF(amountNeeded, "assureNonNegativeReserveAccount")
       if(sim.test) require(_outstandingOSLFpayables.size == 1, "There more than one OSLFpayable after assureNonNegativeReserveAccount")
       if(sim.test){
         require(_outstandingOSLFpayables.head.amountOfReserves == amountNeeded, s"_outstandingOSLFpayables is wrong after assureNonNegativeReserveAccount: ${_outstandingOSLFpayables.head.amountOfReserves} / $amountNeeded")
       }
       if(sim.test){
         require(CB.outstandingOSLFreceivables(this).amountOfReserves == amountNeeded, s"CB.outstandingOSLFreceivables is wrong after assureNonNegativeReserveAccount: ${CB.outstandingOSLFreceivables(this).amountOfReserves} / $amountNeeded")
       }
     }
     if(sim.test) require(_cbReserves.last >= 0.0, s"$this has negative reserve account: ${_cbReserves.last}")
   }
```

```scala


  def useOSLF (amountToBorrowOvernight:Double = _reserveDeficit, usage:String = "useToMeetReserveTarget") {
    if(sim.pln) println(s"$this must use the OSLF since it didn't managed to lend fund from IBM: $currentlyOfferedReservesOnIBM")
    pledgeCollateral(bondsPledgedAsCollateralForOSLF, amountToBorrowOvernight, t)
    if(sim.test){
      require(
          collateral.map(stackOfBonds => stackOfBonds._1 * PVofBond(stackOfBonds._2, t) ).sum >= deficit - math.max(1, deficit * 0.000001),
          s"Provided collateral of $this for IDL is insufficient: ${collateral.map(stackOfBonds => stackOfBonds._1 * PVofBond(stackOfBonds._2, t) ).sum - deficit}."
        )
    }
    if(sim.test) require(collateral.isEmpty, "collateral is not empty.")
    val ra = _cbReserves.last
    transferMoney(CB, this, amountToBorrowOvernight, "OSLF", sim, t, amountToBorrowOvernight * (CB.lendingFacilityRate.last/360))
    if(sim.pln){
      println(s"$this has borrowed $amountToBorrowOvernight from OSLF since it has a deficit of $amountToBorrowOvernight causing either from neg. res. acc. $ra (now ${_cbReserves.last}) or from deficit ${_reserveDeficit} resulting from ${_cbReserves.grouped(4).toBuffer.takeRight(1)(0)}; cbLiabs: ${_cbLiabilities.last}")
    }
    _OSFused(_OSFused.size-1) = {(_OSFused(_OSFused.size-1)._1 + amountToBorrowOvernight, _OSFused(_OSFused.size-1)._2)}
    if(usage == "assureNonNegativeReserveAccount"){
      if(sim.test) require(_outstandingOSLFpayables.isEmpty, s"$this has _outstandingOSLFpayables although it should not when assuring non-negative reserve account.")
      _outstandingOSLFpayables.enqueue( OvernightOSLFloan(this, amountToBorrowOvernight, CB.lendingFacilityRate.last) )
      if(sim.test){
        require(
            !CB.outstandingOSLFreceivables.contains(this),
            s"$this should not already have a OSLF loan when assuring a non-negative reserve account but there already exists one in CB.outstandingOSLFreceivables"
          )
      }
      CB.outstandingOSLFreceivables += this -> CB.OvernightOSLFloan(this, amountToBorrowOvernight, CB.lendingFacilityRate.last)
    } else if(usage == "useToMeetReserveTarget") {
      if(_outstandingOSLFpayables.isEmpty){
        _outstandingOSLFpayables.enqueue( OvernightOSLFloan(this, amountToBorrowOvernight, CB.lendingFacilityRate.last) )
        if(sim.test){
          require(
              !CB.outstandingOSLFreceivables.contains(this),
              s"$this should not already have a OSLF loan when assuring a non-negative reserve account but there already exists one in CB.outstandingOSLFreceivables"
           )
        }
        CB.outstandingOSLFreceivables += this -> CB.OvernightOSLFloan(this, amountToBorrowOvernight, CB.lendingFacilityRate.last)
      } else {
        if(sim.test) require(_outstandingOSLFpayables.size == 1, s"$this has more than one _outstandingOSLFpayables.")
        val currentOutstandingOSLFpayable = _outstandingOSLFpayables.dequeue
        if(sim.test) require(_outstandingOSLFpayables.isEmpty, s"_outstandingOSLFpayables of $this should be empty now.")
        _outstandingOSLFpayables.enqueue( OvernightOSLFloan(this, currentOutstandingOSLFpayable.amountOfReserves + amountToBorrowOvernight, CB.lendingFacilityRate.last) )
        if(sim.test) require(_outstandingOSLFpayables.size == 1, s"$this has more than one _outstandingOSLFpayables.")
        if(sim.test) require(CB.outstandingOSLFreceivables.contains(this), s"$this does not exist in CB.outstandingOSLFreceivables although it already assured a non-negative reserve account.")
        CB.outstandingOSLFreceivables(this) = CB.OvernightOSLFloan(this, CB.outstandingOSLFreceivables(this).amountOfReserves + amountToBorrowOvernight, CB.lendingFacilityRate.last)
      }
    } else sys.error("There is no other usage of useOSLF.")
  }


```

```
        def useOSDF {
          if(_outstandingOSLFpayables.isEmpty){
            val amountToDepositAtCB = math.min(_excessReserves, math.max(0, _cbReserves.last))
            if(sim.pln) println(s"$this deposits $amountToDepositAtCB at OSDF since it has a surplus of ${_excessReserves} resulting from ${_cbReserves.grouped(4).toBuffer.takeRight(1)(0)}")
            _interestOnOSDFrepos = amountToDepositAtCB * (CB.depositFacilityRate.last/360)
            transferMoney(this, CB, amountToDepositAtCB, "OSDF", sim, t, _interestOnOSDFrepos)
            _OSFused(_OSFused.size-1) = {(_OSFused(_OSFused.size-1)._1, _OSFused(_OSFused.size-1)._2 + amountToDepositAtCB)}
          }
        }

    }, "bank_useOSFifNecessary", sim)
   }// method








  /*    ------------------------------------------------------------ Bank Interest Setting -----------------------------------------------------    */

  /**
   *
   *  bank agents' interest on deposits moves in perfect lock-step with the target rate of the CB
   *
   *    */
  def interestOnDeposits = CB.targetFFR.last match {
    case i:Double if(i <  0.03) => math.max(i - 0.0075, 0.001)
    case i:Double if(i <= 0.05) => i - 0.015
    case i:Double if(i >  0.05) => i - 0.03
  }





  /**
   *
   *  bank agents' interest on loans moves in perfect lock-step with the target rate of the CB
   *
   *    */
  def interestOnLoans = roundTo3Digits( CB.targetFFR.last + 0.03 )





  /**
   *
   *   bank agents lend reserves to other banks at market rates of interest applicable to them which vary by bank and depend on the target rate and the amount of outstanding reserves
```

```
 *   relative to the aggregate reserve target.
 *
 *    */
def interestOnIBMLoans (borrower:Bank) = {
  val deviationFromAggregateReserveTargtet = sim.bankList.filter(_.active).map(_._currentAvgReserves).sum / CB.reserveTargetBalances.filter(_._1.active == true).values.map(_.reserveTargetBalance).sum
  def s                                = 0.5 + 0.5 * math.tanh( (deviationFromAggregateReserveTargtet - 1) / 0.1 )
  def demandHigh (d:Double, e:Double) = d   - e   * math.tanh( 5 * deviationFromAggregateReserveTargtet - 7.5 )
  def demandLow  (a:Double, b:Double) = a   - b   * math.tanh( 5 * deviationFromAggregateReserveTargtet - 2.5 )
  val bankRate = CB.targetFFR.last match {
    case i:Double if(i <  0.03 ) => roundTo4Digits( ( s * demandHigh(0.06125 - 0.0025,  0.00125) + (1 - s) * demandLow(0.06125, 0.00125) ) - (0.06 - CB.targetFFR.last) ) //  low interest level
    case i:Double if(i <= 0.05 ) => roundTo4Digits( ( s * demandHigh(0.0625  - 0.005,   0.0025)  + (1 - s) * demandLow(0.0625,  0.0025 ) ) - (0.06 - CB.targetFFR.last) ) //  mid interest level
    case i:Double if(i >  0.05 ) => roundTo4Digits( ( s * demandHigh(0.065   - 0.00865, 0.004)   + (1 - s) * demandLow(0.065,   0.005  ) ) - (0.06 - CB.targetFFR.last) ) // high interest level
  }
  bankRate + riskPremium(borrower)
}



/**
 *
 *  bank agents add an idiosyncratic, i.e. bank specific (either positive or negative) risk premium on top of the rate determined by the target rate and the
 *  amount of outstanding reserves relative to the aggregate reserve target.
 *
 *    */
def riskPremium (borrower:Bank, randomDeviation:Boolean = true, crisis:Boolean = if(sim.bankList.map(_.active).contains(false)) true else false) = {
  crisis match {

    case false =>
      randomDeviation match {

        case false =>
          borrower debt2EquityRatio match {
            case ratio:Double if ratio < 2    => -borrower.debt2EquityRatio
            case ratio:Double if ratio > 1    => 0.01
            case _                            => 0.01
          }// match

        case true =>
              random.nextDouble/500 match {
                case i:Double if(i > 0.1) => -(i-0.1)
                case i:Double if(i < 0.1) => i

              }// match
      }// match

    case true => 0.1

  }// end match crisis

}
```

```scala
/**
 *
 *  This method is just to save data.
 *
 *    */
 def storeInterestRates {
   _interestOnRetailDeposits += interestOnDeposits
   _interestOnRetailLoans    += interestOnLoans
 }

/*    ------------------------------------------------------------- Bank Lending Activity -----------------------------------------------------
 *
 *    The following section describes the methods concering bank agents' activity on the credit market.
 *
 *    */

/**
 *
 *  This class defines a bank loan as well as the inherent data like:
 *  - the amount of interest to pay by the firm
 *  - the amounts and periods in which the firm has to pay interest
 *  - the amounts and periods in which the firm has to make principal payments
 *
 *    */
case class Loan (tickOfBorrowing:Int, borrower:Firm, loan:Double, interestRate:Double, maturity:Int = 480) {
  def amountOfInterest (t:Int) = (interestRate * (loan - ( (loan/(maturity/48)) * ((t-tickOfBorrowing)/48)) )) / 12
  val interestPayments  = SortedMap( Vector.tabulate(maturity/ 4)(n => tickOfBorrowing - 1 + (n+1) *  4).map{t => (t, amountOfInterest(t)) }:_* )
  val principalPayments = SortedMap( Vector.tabulate(maturity/48)(n => tickOfBorrowing - 1 + (n+1) * 48).map{t => (t, loan/(maturity/48))  }:_* )
  }

/**
 *
 *  This bank agent-specific list contains all current debtors of the bank agent.
 *
 *    */
private val _listOfDebtors = Map[Firm, ArrayBuffer[Loan]]()

/**
 *
```

```
   *  This method is just to delete loans from the bank agents' listOfDebtors when the firm has succesfully repayd its current debt obligations.
   *
   *    */
  def deleteDueBusinessLoans (t:Int) = _listOfDebtors.foreach{ case (firm, listOfLoans) => _listOfDebtors += firm -> listOfLoans.filterNot(_.principalPayments.last._1 <= t) }



  /**
   *
   *  Bank agents' check the creditworthiness of a firm in case of a loan request. The resulting interest offered to the firm depends on the firm's ability to create sufficient amounts of cash flows
   *  in the past and whether these cash flows would be enough to repay the loan if it would be granted by the bank agent. In a further step, the requesting firm can decide to take the loan or not
   *  depending on the offered interest of the bank.
   *
   *    */
  def proofCreditworthiness (corporation:Corporation, requestedAmountOfMoney:Double, t:Int) = {time({
    val statusOfCreditworthiness = Seq("unrestricted", "restricted", "denied")
    corporation match{
      case corp:Firm    =>
        val ppNow = if(_listOfDebtors.contains(corp)) _listOfDebtors(corp).map(_.principalPayments.filterKeys(_ >= t).filterKeys(_ <= t + math.min(corp.age, 48)).values.sum).sum else 0.0
        val ipNow = if(_listOfDebtors.contains(corp)) _listOfDebtors(corp).map( _.interestPayments.filterKeys(_ >= t).filterKeys(_ <= t + math.min(corp.age, 48)).values.sum).sum else 0.0
        val requestedLoan = Loan(t, corp, requestedAmountOfMoney, interestOnLoans)
        val ppThen = ppNow + requestedLoan.principalPayments.filterKeys(_ <= t + math.min(corp.age, 48)).values.sum
        val ipThen = ipNow +  requestedLoan.interestPayments.filterKeys(_ <= t + math.min(corp.age, 48)).values.sum
        val riskPremium = math.min( ((ppThen + ipThen) / corp.revenues.takeRight(math.min(corp.age, 48)).sum) / 100, 0.15)
        if(t < 50 || riskPremium.isNaN()) ( statusOfCreditworthiness.head, requestedAmountOfMoney, interestOnLoans ) else ( statusOfCreditworthiness.head, requestedAmountOfMoney, interestOnLoans - 0.01 + riskPremium )
      case corporation:Bank    => (statusOfCreditworthiness.head, requestedAmountOfMoney, interestOnIBMLoans(corporation))
      case _                   => sys.error("Creditworthiness cannot be proofed. Corporation must either be a Firm or a Bank!")
    }
   }, "bank_proofCreditworthiness", sim)
  }




  /**
   *   Tests whether the bank complies with
   *   1. the min CAR of 4.5% of RWA
   *   2. the Capital Conservation Buffer (CConB) of 2.5% of RWA on top of CAR
   *   3. the Countercyclical Buffer (CCycB) of 2.5% of RWA on top of CAR + CConB
   *   4. the surcharges on SIBs (1%-2.5%) on top of CAR + CConB + CCycB
   *   5. the non-risk sensitive LR (3%)
   *
   *    */
  def proofRegulatoryRequirements (t:Int):Boolean = {time({
    // risk-based measures
    val numberOfActiveBanks = sim.bankList.filter(_.active).size
    val currentMarketShare = _totalAssets.last / sim.bankList.filter(_.active).map(_.totalAssets.last).sum
    val surchargeBucket:Int = if(sim.surcharges) currentMarketShare match {
      case marketShare:Double if marketShare <= 1.0 / numberOfActiveBanks => 6     // [20% @ 5 banks] -> equal market share, same size as peers
```

Bank.scala

```scala
      case marketShare:Double if marketShare <= 1.3 / numberOfActiveBanks => 5     // [26% @ 5 banks] -> 40,54% larger than avg. peer
      case marketShare:Double if marketShare <= 1.6 / numberOfActiveBanks => 4     // [32% @ 5 banks] -> 88,24% larger than avg. peer
      case marketShare:Double if marketShare <= 1.9 / numberOfActiveBanks => 3     // [20% @ 5 banks]
      case marketShare:Double if marketShare <= 2.2 / numberOfActiveBanks => 2     // [20% @ 5 banks]
      case _                                                             => 1     // [20% @ 5 banks]
    } else 6
    val testCAR        = if(_currentEquityOfRWA(t) < sim.supervisor.CAR + sim.supervisor.surchargesOnSIBs(surchargeBucket)) false else true

    // non-risk based  measure
    val testLR         = if(sim.LR) _currentEquityRatio match {
      case eRatio:Double if eRatio >= sim.supervisor.minLeverageRatio => true
      case eRatio:Double if eRatio <  sim.supervisor.minLeverageRatio => false
    } else true

    if(Seq(testCAR, testLR).contains(false)) false else true
   }, "bank_proofRegulatoryRequirements", sim)
  }






  /**
   *
   *  If the requesting firm has sufficient cash flow in the past, the bank grants the loan.
   *
   *    */
  def grantCredit2Firm (firm:Firm, amount:Double, interest:Double, t:Int):Unit = {time({
    if(sim.test) require(amount >= 0.0, s"The requested amount of $firm is negative: $amount.")
    val grantedLoan = Loan(t, firm, amount, interest)
    if(_listOfDebtors.contains(firm)){
      listOfDebtors(firm) += grantedLoan
    } else listOfDebtors += firm -> ArrayBuffer( grantedLoan )
    CB.credit2privateSector(CB.credit2privateSector.size-1) += amount + grantedLoan.interestPayments.values.sum
    deposit(sim.creditGrantedByTB, amount, t, sim)
    if(sim.pln) println(s"$this grants credit of $amount to $firm since it is creditworthy enough (D/E of ${firm.debt2EquityRatio})")
    transferMoney(this, firm, amount, "grantLoan", sim, t, grantedLoan.interestPayments.values.sum )
   }, "bank_grantCredit2Firm", sim)
  }






  /**
   *
   *  Bank agents pay interest on interest bearing deposits of their customers once a year.
   *
   *    */
  def payInterestOnDeposits (t:Int) = {time({
    if(t % 48 == 0)
{
// only yearly
```

```
        _businessClients.foreach(firm => if(firm active) transferMoney(this, firm,                     firm.bankDeposits.last * interestOnDeposits, "interestOnRetailDeposits", sim, t))           // firms
          _retailClients.foreach(hh   =>                 transferMoney(this, hh,                         hh.bankDeposits.last * interestOnDeposits, "interestOnRetailDeposits", sim, t))           // hh
                                                          transferMoney(this, sim.government, sim.government.bankDeposits.last * interestOnDeposits, "interestOnRetailDeposits", sim, t)            // gov
          _MMMFClients.foreach(mmmf   => if(mmmf active) transferMoney(this, mmmf,                     mmmf.bankDeposits.last * interestOnDeposits, "interestOnRetailDeposits", sim, t))           // MMMF
              _BDClients.foreach(bd   => if(  bd active) transferMoney(this, bd,                         bd.bankDeposits.last * interestOnDeposits, "interestOnRetailDeposits", sim, t))           // BD
    }
   }, "bank_payInterestOnDeposits", sim)
  }











  /**
   *
   *  If a bank agent is not able to meet its debt obligations, it is either bailed out by the government agent if it is of systematical importance and otherwise it is resolved and exits the market.
   *
   *    */
  def shutDownBank (t:Int) = {time({

    def transferBondClaims2CB (listOfIDs:Map[Long, Double]) = {
      val couponClaimsBeforePurchaseCB = if(sim.test) rounded( sim.government.coupon2PayCB.filterKeys(_ > t).values.sum ) else 0.0
      val FVClaimsBeforePurchaseCB     = if(sim.test) rounded(    sim.government.dueDebtCB.filterKeys(_ > t).values.sum ) else 0.0
      var transferedCouponClaimsCB = 0.0
      var transferedFVClaimsCB     = 0.0
      listOfIDs.foreach{
        case(id, fraction) =>
          val purchasedSoB = sim.government.findStackOfBondsByID(id)
          purchasedSoB.bond.ticksOfCouponPayment.filter(_ > t).foreach{
            tick =>
              if(sim.government.coupon2PayCB.contains(tick)){
                sim.government.coupon2PayCB(tick) += purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
              } else sim.government.coupon2PayCB += tick -> purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
              transferedCouponClaimsCB += purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
          }
          if(sim.government.dueDebtCB.contains(purchasedSoB.bond.maturity)){
            sim.government.dueDebtCB(purchasedSoB.bond.maturity) += purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
          } else sim.government.dueDebtCB += purchasedSoB.bond.maturity -> purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
          transferedFVClaimsCB += purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
      }
      val couponClaimsAfterPurchaseCB = if(sim.test) rounded( sim.government.coupon2PayCB.filterKeys(_ > t).values.sum ) else 0.0
      val FVClaimsAfterPurchaseCB     = if(sim.test) rounded(    sim.government.dueDebtCB.filterKeys(_ > t).values.sum ) else 0.0
```

```scala
      if(sim.test){
        require(
            SEc(couponClaimsAfterPurchaseCB, rounded(couponClaimsBeforePurchaseCB + transferedCouponClaimsCB), 5),
            s"CB buys fire saled bonds of insolvent $this but COUPON claims are not consistent: claims after purchase ($couponClaimsAfterPurchaseCB) are not equal to claims before ($couponClaimsBeforePurchaseCB) plus transferedCouponCLaims ($transferedCouponClaimsCB)"
          )
        require(
            SEc(    FVClaimsAfterPurchaseCB, rounded(FVClaimsBeforePurchaseCB     + transferedFVClaimsCB), 5),
            s"CB buys fire saled bonds of insolvent $this but FACEVALUE claims are not consistent: claims after purchase ($FVClaimsAfterPurchaseCB) are not equal to claims before ($FVClaimsBeforePurchaseCB) plus transferedFVCLaims ($transferedFVClaimsCB)"
          )
      }
    }




    if(sim.pln) println(s"""     $this is shut down (negative equity).     """)
    sim.checkGovDeposits(s"before shutting down $this", t)
    if(sim.pln) println(s"Before resolution of $this: rD = ${_retailDeposits.last}")
    if(sim.pln) println(s"rD: ${_retailDeposits.last} (before transfering to peer); part of firms: ${_businessClients.map(_.bankDeposits.last).sum}; part of hh: ${_retailClients.map(_.bankDeposits.last).sum}")
    if(sim.pln) println(s"Reserves --> ${rounded(sim.bankList.filter(_.active).map(_.cbReserves.last).sum)} / ${CB.reserves.last} (CB);\n ${sim.bankList.map(bank => bank -> (bank.active, bank.cbReserves.last))} ")
    println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
      if(sim.pln) printBSP
      _active                       = false
      _tickOfInsolvency += t
      _periodOfReactivation = t - (t % 4) + 24 + 4 * random.nextInt(10) + 1
      if(sim.test) require( (_periodOfReactivation-1) % 4 == 0, s"$this has an incorrect _periodOfReactivation: ${_periodOfReactivation}" )
      _insolvencies(_insolvencies.size-1) += 1

      _listOfDebtors.foreach{
        case (firm, listOfLoans) =>
          listOfLoans.foreach{
            loan =>
              val principal2Repay = rounded(loan.principalPayments.filter(_._1 > t).values.sum)
              val  interest2Repay = rounded(loan.interestPayments.filter( _._1 > t).values.sum)
              val     liquidFunds = math.min(firm.bankDeposits.last, principal2Repay + interest2Repay)
              if(sim.pln) println(s"$firm --> principal2Repay: $principal2Repay (BSP: ${firm.debtCapital.last}) + interest2Repay: $interest2Repay (BSP: ${firm.interestOnDebt.last})")
              withdraw(firm.interestOnDebt,  interest2Repay,  t, sim)
              withdraw(firm.debtCapital,     principal2Repay, t, sim)
              withdraw(firm.bankDeposits,    liquidFunds,     t, sim)
              withdraw(_retailDeposits,      liquidFunds,     t, sim)
              withdraw(_interestReceivables, interest2Repay,  t, sim)
              withdraw(_businessLoans,       principal2Repay, t, sim)
          }
      }
      if(sim.pln) println(s"-- after payBack of businessLoans: rD = ${_retailDeposits.last}")
      _listOfDebtors.clear()
      if(sim.pln) println("After cleaning outstanding business loans:")
      if(sim.pln) println(s"rD: ${_retailDeposits.last} (before transfering to peer); part of firms: ${_businessClients.map(_.bankDeposits.last).sum}; part of hh: $
```

```scala
{_retailClients.map(_.bankDeposits.last).sum}")
      if(sim.pln) println(s"Reserves --> ${rounded(sim.bankList.filter(_.active).map(_.cbReserves.last).sum)} / ${CB.reserves.last} (CB);\n ${sim.bankList.map(bank => bank -> (bank.active, bank.cbReserves.last))} ")
      println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
      if(sim.pln) printBSP
      sim.checkGovDeposits(s"clearing businessLoans", t)

      // clear client's houseBankRelationship
      if(sim.pln) println(s"${_businessClients.map(firm => firm -> firm.bankDeposits.last)}")
      _businessClients.foreach(_.getNewHouseBank)
      if(sim.pln) println(s"${_businessClients.map(firm => firm -> firm.bankDeposits.last)}")
      if(sim.pln) println(s"${_retailClients.map(hh => hh -> hh.bankDeposits.last)}")
      _retailClients.foreach(_.getNewHouseBank)
      if(sim.pln) println(s"${_retailClients.map(hh => hh -> hh.bankDeposits.last)}")
      _MMMFClients.foreach(_.getNewHouseBank)
        _BDClients.foreach(_.getNewHouseBank)
      if(sim.pln) println("After assigning a new houseBank to clients:")
      if(sim.pln) println(s"rD: ${_retailDeposits.last} (before transfering to peer); part of firms: ${_businessClients.map(_.bankDeposits.last).sum}; part of hh: ${_retailClients.map(_.bankDeposits.last).sum}")
      if(sim.pln) println(s"Reserves --> ${rounded(sim.bankList.filter(_.active).map(_.cbReserves.last).sum)} / ${CB.reserves.last} (CB);\n ${sim.bankList.map(bank => bank -> (bank.active, bank.cbReserves.last))} ")
      if(sim.pln){
        println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
      }
      if(sim.pln) printBSP
      sim.checkGovDeposits(s"getNewHB for clients", t)

      // IDL (secured)
      if(sim.test) require(!CB.intraDayLiquidity.contains(this), s"CB.intraDayLiquidity contains $this while is shouldn't: ${CB.intraDayLiquidity(this)}")
      if(bondsPledgedAsCollateralForIDL.nonEmpty){
        dePledgeCollateral(bondsPledgedAsCollateralForIDL)
        if(sim.test) require(bondsPledgedAsCollateralForIDL.isEmpty, s"bondsPledgedAsCollateral are not empty: $bondsPledgedAsCollateralForIDL")
      }
      if(CB.intraDayLiquidity.contains(this)) CB.intraDayLiquidity -= this
      _borrowedIntraDayLiquidity = 0.0
      sim.checkGovDeposits(s"clearing IDL", t)

      if(listOfBonds.nonEmpty){
        val bankruptFractionOfFinancialSystem =
          (sim.bankList.filterNot(_.active).size + sim.MMMFList.filterNot(_.active).size + sim.BrokerDealerList.filterNot(_.active).size) / (sim.numberOfBanks + sim.numberOfMMMF + sim.numberOfBrokerDealer)
        val discount = math.min(0.5, bankruptFractionOfFinancialSystem)
        val price    = PV_LoB(t) * (1 - discount)
        val listOfLiquidBanks = sim.random.shuffle( sim.bankList.filter(bank => bank.active && bank.cbReserves.last >= price) )
        if(listOfLiquidBanks.nonEmpty){
          val bankWhichBuysBonds:Bank = listOfLiquidBanks.head
        val couponClaimsBeforePurchase = if(sim.test) rounded(sim.government.coupon2Pay.filterKeys(_ > t).filter(_._2.contains(bankWhichBuysBonds)).map(_._2(bankWhichBuysBonds)).sum) else 0.0
        val FVClaimsBeforePurchase     = if(sim.test) rounded(   sim.government.dueDebt.filterKeys(_ > t).filter(_._2.contains(bankWhichBuysBonds)).map(_._2(bankWhichBuysBonds)).sum) else 0.0

        var transferedCouponClaims = 0.0
```

```
        var transferedFVClaims     = 0.0
        listOfBonds.foreach{
          case(id, fraction) =>
            val purchasedSoB = sim.government.findStackOfBondsByID(id)
            purchasedSoB.bond.ticksOfCouponPayment.filter(_ > t).foreach{
              tick =>
                if(sim.government.coupon2Pay.contains(tick)) {
                  if(sim.government.coupon2Pay(tick).contains(bankWhichBuysBonds)){
                    sim.government.coupon2Pay(tick)(bankWhichBuysBonds) += purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
                  } else sim.government.coupon2Pay(tick) += bankWhichBuysBonds -> purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
                } else sim.government.coupon2Pay += tick -> Map(bankWhichBuysBonds -> purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction)
                transferedCouponClaims += purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
            }
            if(sim.government.dueDebt.contains(purchasedSoB.bond.maturity)) {
              if(sim.government.dueDebt(purchasedSoB.bond.maturity).contains(bankWhichBuysBonds)){
                sim.government.dueDebt(purchasedSoB.bond.maturity)(bankWhichBuysBonds) += purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
              } else sim.government.dueDebt(purchasedSoB.bond.maturity) += bankWhichBuysBonds -> purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
            } else sim.government.dueDebt += purchasedSoB.bond.maturity -> Map(bankWhichBuysBonds -> purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction)
            transferedFVClaims += purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
        }
        val couponClaimsAfterPurchase = if(sim.test) rounded(sim.government.coupon2Pay.filterKeys(_ > t).filter(_._2.contains(bankWhichBuysBonds)).map(_._2(bankWhichBuysBonds)).sum) else 0.0
        val FVClaimsAfterPurchase     = if(sim.test) rounded(   sim.government.dueDebt.filterKeys(_ > t).filter(_._2.contains(bankWhichBuysBonds)).map(_._2(bankWhichBuysBonds)).sum) else 0.0
        if(sim.test){
          require(
              SEc(couponClaimsAfterPurchase, rounded(couponClaimsBeforePurchase + transferedCouponClaims), 5),
              s"$bankWhichBuysBonds buys fire saled bonds of insolvent $this but COUPON claims are not consistent: claims after purchase ($couponClaimsAfterPurchase) are not equal to claims before ($couponClaimsBeforePurchase) plus transferedCouponCLaims ($transferedCouponClaims)"
            )
          require(
              SEc(    FVClaimsAfterPurchase, rounded(FVClaimsBeforePurchase     + transferedFVClaims),     5),
              s"$bankWhichBuysBonds buys fire saled bonds of insolvent $this but FACEVALUE claims are not consistent: claims after purchase ($FVClaimsAfterPurchase) are not equal to claims before ($FVClaimsBeforePurchase) plus transferedFVCLaims ($transferedFVClaims)"
            )
        }
        bankWhichBuysBonds.listOfBonds ++= listOfBonds
        listOfBonds.clear()
          withdraw(bankWhichBuysBonds.cbReserves, price, t, sim)
          deposit(_cbReserves, price, t, sim)
      } else {
        if(sim.test) sim.government.testCBBondPayments(t, false)
        transferBondClaims2CB(listOfBonds)
        listOfBonds.foreach{ case(id, fraction) => if(CB.listOfBonds.contains(id)) CB.listOfBonds(id) += fraction else CB.listOfBonds += id -> fraction}
        listOfBonds.clear()
        if(sim.test) sim.government.testCBBondPayments(t, false)
        deposit(CB.reserves, price, t, sim)
        deposit(_cbReserves, price, t, sim)
      }
      }
      updatePVofSoBs(t)
      if(sim.pln) println("After fire sale of bonds:")
      if(sim.pln) println(s" Reserves --> ${rounded(sim.bankList.filter(_.active).map(_.cbReserves.last).sum)} / ${CB.reserves.last} (CB);\n ${sim.bankList.map(bank => bank -> (bank.active, bank.cbReserves.last))} ")
```

```
    if(sim.pln) println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
    if(sim.pln) printBSP
    sim.checkGovDeposits(s"fireSale rest of bonds in LoB", t)


    println(s"rD: ${_retailDeposits.last} (before transfering to peer); part of firms: ${_businessClients.map(_.bankDeposits.last).sum}; part of hh: ${_retailClients.map(_.bankDeposits.last).sum}")
    println(s"-- before transferring deposits from old to new houseBank of clients: rD = ${_retailDeposits.last}")
    if(_businessClients.map(_.bankDeposits.last).sum > 0){
      _businessClients.foreach{
        firm =>
          val amount2Transfer = math.min(_retailDeposits.last, firm.bankDeposits.last)
          val reserves        = math.min(_cbReserves.last, amount2Transfer)
          if(sim.pln) println(s"$firm --> transfering $amount2Transfer from $this to ${firm.houseBank} paying with reserves of $reserves")
          withdraw(_retailDeposits,             amount2Transfer, t, sim)
          withdraw(_cbReserves,                 reserves,        t, sim)
          deposit(firm.houseBank.retailDeposits, amount2Transfer, t, sim)
          deposit(firm.houseBank.cbReserves,     reserves,        t, sim)
          firm.bankDeposits(firm.bankDeposits.size-1) = amount2Transfer
      }
    }
    println(s"-- after transfer of firm and before transfer of HH deposits: rD = ${_retailDeposits.last}")
    _retailClients.foreach{
      hh =>
        val amount2Transfer = math.min(_retailDeposits.last, hh.bankDeposits.last)
        val reserves        = math.min(_cbReserves.last, amount2Transfer)
        if(sim.pln) println(s"$hh --> transfering $amount2Transfer from $this to ${hh.houseBank} paying with reserves of $reserves")
        withdraw(_retailDeposits,            amount2Transfer, t, sim)
        withdraw(_cbReserves,                reserves,        t, sim)
        deposit(hh.houseBank.retailDeposits, amount2Transfer, t, sim)
        deposit(hh.houseBank.cbReserves,     reserves,        t, sim)
        hh.bankDeposits(hh.bankDeposits.size-1) = amount2Transfer
    }
    println(s"-- after transferring deposits from old to new houseBank of clients: rD = ${_retailDeposits.last}")
    if(sim.test) require(_retailDeposits.last < 1, s"There are retailDeposits left after transfering them to the client's new houseBank: ${_retailDeposits.last}")
    println("After tranfering retailDeposits of clients to their new houseBanks:")
    if(sim.pln){
      println(s"Reserves --> ${rounded(sim.bankList.filter(_.active).map(_.cbReserves.last).sum)} / ${CB.reserves.last} (CB);\n ${sim.bankList.map(bank => bank -> (bank.active, bank.cbReserves.last))} ")
    }
    println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
    if(sim.pln) printBSP
    // MMMF
    println(s"-- after transfer of firm/hh and before transfer of MMMF deposits: rD = ${_retailDeposits.last}")
    _MMMFClients.foreach{
      mmmf =>
        val amount2Transfer = math.min(_retailDeposits.last, mmmf.bankDeposits.last)
        val reserves        = math.min(_cbReserves.last, amount2Transfer)
        if(sim.pln) println(s"$mmmf --> transfering $amount2Transfer from $this to ${mmmf.houseBank} paying with reserves of $reserves")
        withdraw(  _retailDeposits,            amount2Transfer, t, sim)
        withdraw(  _cbReserves,                reserves,        t, sim)
        deposit(mmmf.houseBank.retailDeposits, amount2Transfer, t, sim)
```

```
        deposit(mmmf.houseBank.cbReserves,     reserves,        t, sim)
        mmmf.bankDeposits(mmmf.bankDeposits.size-1) = amount2Transfer
    }
    println(s"-- after transfer of firm/hh/MMMF and before transfer of BD deposits: rD = ${_retailDeposits.last}")
    _BDClients.foreach{
      bd =>
        val amount2Transfer = math.min(_retailDeposits.last, bd.bankDeposits.last)
        val reserves        = math.min(_cbReserves.last, amount2Transfer)
        if(sim.pln) println(s"$bd --> transfering $amount2Transfer from $this to ${bd.houseBank} paying with reserves of $reserves")
        withdraw(_retailDeposits,            amount2Transfer, t, sim)
        withdraw(_cbReserves,                reserves,        t, sim)
        deposit(bd.houseBank.retailDeposits, amount2Transfer, t, sim)
        deposit(bd.houseBank.cbReserves,     reserves,        t, sim)
        bd.bankDeposits(bd.bankDeposits.size-1) = amount2Transfer
    }
    println(s"-- after transfer of firm/hh/MMMF/BD: rD = ${_retailDeposits.last}")
    _retailDeposits(_retailDeposits.size-1) = 0.0
    require(_retailDeposits.last == 0.0)
    sim.checkGovDeposits(s"transfering retailDeposits to clients", t)


     while(_outstandingIBMpayables.nonEmpty){
       val loanToRepay = _outstandingIBMpayables.dequeue
       transferMoney(this, loanToRepay.lendingBank, loanToRepay.amountOfReserves, "depreciateOvernightIBMloan", sim, t, rounded(loanToRepay.amountOfReserves * (loanToRepay.interest/360)))
       loanToRepay.lendingBank.outstandingIBMreceivables -= this
     }
     if(sim.test) require(_outstandingIBMpayables.isEmpty, s"_outstandingIBMpayables is not empty: ${_outstandingIBMpayables}")
    _outstandingIBMreceivables.clear()
    if(sim.test) require(_outstandingIBMreceivables.isEmpty)
    if(sim.pln) println("After cleaning IBM claims:")
    if(sim.pln){
      println(s"Reserves --> ${rounded(sim.bankList.filter(_.active).map(_.cbReserves.last).sum)} / ${CB.reserves.last} (CB);\n ${sim.bankList.map(bank => bank -> (bank.active, bank.cbReserves.last))} ")
    }
    if(sim.pln){
      println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
    }
    if(sim.pln) printBSP
    sim.checkGovDeposits(s"clearing IBM claims", t)

    if(sim.test) sim.government.testCBBondPayments(t, false)
    if(sim.pln) println(s"CB_LOB before OMO-bonds are transfered: ${CB.listOfBonds}")
    if(sim.pln) println(s"$this OMO before OMO-bonds are transfered: $bondsPledgedAsCollateralForOMO")

    CB.outstandingOMOreceivabels -= this
    if(sim.test) require(!CB.outstandingOMOreceivabels.contains(this))
    transferBondClaims2CB(bondsPledgedAsCollateralForOMO)
    bondsPledgedAsCollateralForOMO.foreach{ case(id, fraction) => if(CB.listOfBonds.contains(id)) CB.listOfBonds(id) += fraction else CB.listOfBonds += id -> fraction}
    bondsPledgedAsCollateralForOMO.clear
    withdraw(CB.loans2CommercialBanks, _outstandingOMOpayables.head.amountOfReserves + _outstandingOMOpayables.head.amountOfReserves * (_outstandingOMOpayables.head.interest/12), t, sim)
    withdraw(_cbLiabilities,           _outstandingOMOpayables.head.amountOfReserves + _outstandingOMOpayables.head.amountOfReserves * (_outstandingOMOpayables.head.interest/12),
```

```scala
t, sim)
      _outstandingOMOpayables.clear()
      sim.checkGovDeposits(s"clearing OMO", t)

      if(sim.pln) println(s"CB_LOB after OMO-bonds are transfered: ${CB.listOfBonds}")
      if(sim.pln) println(s"$this OSLF before OSLF-bonds are transfered: $bondsPledgedAsCollateralForOSLF")
      if(_outstandingOSLFpayables.nonEmpty){
        if(sim.test) require(_outstandingOSLFpayables.size == 1)
        CB.outstandingOSLFreceivables -= this
        transferBondClaims2CB(bondsPledgedAsCollateralForOSLF)
        bondsPledgedAsCollateralForOSLF.foreach{ case(id, fraction) => if(CB.listOfBonds.contains(id)) CB.listOfBonds(id) += fraction else CB.listOfBonds += id -> fraction}
        bondsPledgedAsCollateralForOSLF.clear
        withdraw(CB.loans2CommercialBanks, math.max(_outstandingOSLFpayables.head.amountOfReserves + _outstandingOSLFpayables.head.amountOfReserves * (_outstandingOSLFpayables.head.interest/360), _cbLiabilities.last), t, sim)
        withdraw(_cbLiabilities,           math.max(_outstandingOSLFpayables.head.amountOfReserves + _outstandingOSLFpayables.head.amountOfReserves * (_outstandingOSLFpayables.head.interest/360), _cbLiabilities.last), t, sim)
          _outstandingOSLFpayables.clear()                                                             //
      }
       updatePVofSoBs(t)
       if(sim.pln) println("After transfer of collateral (OMO/OSLF) to CB:")
       if(sim.pln) printBSP
       sim.government.coupon2Pay.filterKeys(_ > t).filter(_._2.contains(this)).foreach{case(tick, mapOfClaims) => sim.government.coupon2Pay(tick) -= this}
          sim.government.dueDebt.filterKeys(_ > t).filter(_._2.contains(this)).foreach{case(tick, mapOfClaims) => sim.government.dueDebt(tick)    -= this}
    if(sim.test){
      sim.bankList.foreach(bank => sim.government.testBankBondPayments(bank, t, false))
      if(sim.pln) println(s"CB_LOB: ${CB.listOfBonds}")
      sim.government.testCBBondPayments(t, false)
    }
    sim.checkGovDeposits(s"clearing OSLF", t)

    if(_OSDF.last > 0){
      transferMoney(CB, this, _OSDF.last, "repayOSDFwoInterest", sim, t, _interestOnOSDFrepos)
      _interestOnOSDFrepos = 0.0
    }
    if(sim.pln) println("After cleaning CB claims:")
    if(sim.pln) println(s"Reserves --> ${rounded(sim.bankList.filter(_.active).map(_.cbReserves.last).sum)} / ${CB.reserves.last} (CB);\n ${sim.bankList.map(bank => bank -> (bank.active, bank.cbReserves.last))} ")
    if(sim.pln){
      println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
    }
    if(sim.pln) printBSP
    sim.checkGovDeposits(s"clearing OSDF", t)


    val gDepositsAtBankruptBank = _govDeposits.last
    val gDeposits2Transfer      = math.min(_govDeposits.last, _cbReserves.last)
    val newGovBank              = sim.bankList.filter(_.active)( sim.random.nextInt(sim.bankList.filter(_.active).size) )
    withdraw(_govDeposits,          gDeposits2Transfer, t, sim)
    withdraw(_cbReserves,           gDeposits2Transfer, t, sim)
    deposit(newGovBank.govDeposits, gDeposits2Transfer, t, sim)
    deposit(newGovBank.cbReserves,  gDeposits2Transfer, t, sim)
    withdraw(sim.government.bankDeposits, gDepositsAtBankruptBank, t, sim)
    deposit( sim.government.bankDeposits, gDeposits2Transfer,      t, sim)
```

```
    if(_govDeposits.last < _cbReserves.last){
      if(sim.test){
        require(
            _govDeposits.last < 1,
            "There are govDeposits left after transfering them to another bank.") else if(sim.test) require(_cbReserves.last < 1, "There are reserves left after transfering govDeposits to another bank."
          )
      }
    }
    if(sim.pln) println(s"Reserves --> ${rounded(sim.bankList.filter(_.active).map(_.cbReserves.last).sum)} / ${CB.reserves.last} (CB);\n ${sim.bankList.map(bank => bank -> (bank.active, bank.cbReserves.last))} ")
    if(sim.pln){
      println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
    }
    _govDeposits(_govDeposits.size-1) = 0.0
    sim.checkGovDeposits(s"tranfering govDeposits to other bank", t)

    if(_cbReserves.last > 0){
      withdraw(CB.reserves, _cbReserves.last, t, sim)
      withdraw(_cbReserves, _cbReserves.last, t, sim)
    }
    if(_businessLoans.last > 0) withdraw(_businessLoans, _businessLoans.last, t, sim)
    if(_cbLiabilities.last > 0) withdraw(_cbLiabilities, _cbLiabilities.last, t, sim)
    sim.checkGovDeposits(s"clearing rest of BSP", t)


    if(sim.pln) println("After resolving claims and BSP:")
    if(sim.pln) println(s"Reserves --> ${rounded(sim.bankList.filter(_.active).map(_.cbReserves.last).sum)} / ${CB.reserves.last} (CB);\n ${sim.bankList.map(bank => bank -> (bank.active, bank.cbReserves.last))} ")
    if(sim.pln){
      println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
    }
    if(sim.pln) printBSP


    do{
      owners.foreach{
        hh =>
          if(hh != null){
            if(sim.test) assert(hh.foundedCorporations.contains(this), hh.foundedCorporations + " does not include " + this + "?")
            hh.foundedCorporations -= this
            if(sim.test) assert(hh.shareOfCorporations.contains(this), hh.shareOfCorporations + " does not include " + this + "?")
            hh.shareOfCorporations -= this
            if(sim.test) assert(owners.contains(hh), owners + " does not include " + hh + "?")
            owners                          -= hh
          }
      }
    } while (owners.nonEmpty)
    if(sim.test) assert(owners.isEmpty, {if(sim.pln) println(owners);    sys.error("There are owners left after shut down")})


      _businessClients.clear()
```

```
        _retailClients.clear()
          _MMMFClients.clear()
            _BDClients.clear()


     }, "bank_shutDownFirm", sim)
    }






    /**
     *
     *  After a resolution of a non-SIB, there is a possibility that a new bank enters the market (from a technical point of view, the entirely cleaned but already existing bank object is reactivated) if there are
     *  enough HH that provide sufficient liquidity to found a new bank.
     *
     *    */
    def reactivateBank (t:Int) {time({
      if(sim.pln) println( s"TA of inactive $this --> bL: ${_businessLoans.last}, IBM: ${_interbankLoans.last}, b: ${bonds.last}, intR: ${_interestReceivables.last}, OSDF: ${_OSDF.last}, R: ${_cbReserves.last}" )
      if(sim.pln) println( s"TL of inactive $this --> rD: ${_retailDeposits.last}, gD: ${_govDeposits.last}, cbL: ${_cbLiabilities.last}, IBM: ${_interbankLiabilities.last}" )
      if(_interestReceivables.last > 0) withdraw(_interestReceivables, interestReceivables.last, t, sim)
      if(sim.test){
        require(
            rounded( Seq(_businessLoans.last, _interbankLoans.last, bonds.last, _interestReceivables.last, _OSDF.last, _cbReserves.last).sum ) < 1,
            s"""Reactivated bank has assets left from bankruptcy:\n ${if(sim.pln) printBSP}"""
         )
      }
      if(sim.test){
        require(
            rounded( Seq(_retailDeposits.last, _govDeposits.last, _cbLiabilities.last, _interbankLiabilities.last).sum)                         < 1,
            s"""Reactivated bank has liabs left from bankcruptcy:\n ${if(sim.pln) printBSP}"""
         )
      }

      // renew owners
      if(sim.test) require(owners.isEmpty, {if(sim.pln) println(owners);    sys.error("new activated bank should not have any owners yet")})
      _age = 0
      val minEquity  = 250000.0
      val investment =   5000.0
      val minOfNewInvestors = (1 * sim.numberOfHH) / sim.numberOfBanks
      val newOwners              = random.shuffle(sim.hhList.filter(_.bankDeposits.last >= investment))
      val newOwnersContribution = newOwners.map(no => no -> investment).toMap
      val recapitalizationGap   = minEquity - newOwnersContribution.values.sum
      if(newOwners.nonEmpty) {
        _active = true
        newOwners.foreach{
          hh =>
            owners                  += hh
            hh.foundedCorporations += this
            hh.shareOfCorporations += this -> newOwnersContribution(hh) / newOwnersContribution.values.sum
```

```
          if(sim.pln) println(s"$hh founded $this with a share of ${newOwnersContribution(hh) / newOwnersContribution.values.sum}")
        }
        if(sim.test) require(rounded(owners.map(_.shareOfCorporations(this)).sum) == 1, s"${owners.map(_.shareOfCorporations(this)).sum}")
        owners.foreach(owner => transferMoney(owner, this, newOwnersContribution(owner), "reactivateBank", sim, t))
        if(sim.pln) println(s"The BS of the reactivated $this")
        updatePVofSoBs(t)
        if(sim.pln) printBSP
        val TA = rounded( Seq(_businessLoans.last, _interbankLoans.last, bonds.last, _interestReceivables.last, _OSDF.last, _cbReserves.last).sum )
        if(sim.pln) println( s"TA of activated $this --> bL: ${_businessLoans.last}, IBM: ${_interbankLoans.last}, b: ${bonds.last}, intR: ${_interestReceivables.last}, OSDF: $
{_OSDF.last}, R: ${_cbReserves.last}" )
        val TL = rounded( Seq(_retailDeposits.last, _govDeposits.last, _cbLiabilities.last, _interbankLiabilities.last).sum)
        if(sim.pln) println( s"TL of activated $this --> rD: ${_retailDeposits.last}, gD: ${_govDeposits.last}, cbL: ${_cbLiabilities.last}, IBM: ${_interbankLiabilities.last}" )
        _equityAfterReactivation += rounded( TA - TL )
      } else {
        _periodOfReactivation = t + 24
        if(sim.pln) println("Currently no entrepreneurs around here to reactivate " + this)
      }
     }, "bank_reactivateBank", sim)
    }










  /**
   *
   *  thows back the bank agent's current equity ratio
   *
   *    */
  def _currentEquityRatio = {
    if(_equity.nonEmpty) {
      _equity.last match {
        case equity:Double if equity == 0.0 => if( (_retailDeposits.last + _govDeposits.last + _interbankLiabilities.last + _cbLiabilities.last) == 0 ) 1.0 else 0.0
        case equity:Double if equity  > 0.0 => _equity.last / _totalAssets.last
        case _                              => 0.0
      }
    } else 1.0
  }




  /**
   *
   *  throws back the bank agent's equity-to-RWA ratio
   *
   *    */
  def _currentEquityOfRWA (t:Int) = {
    if(_equity.nonEmpty){
      val cRWA = _currentRWA(t)
```

```
    if(cRWA > 0.0) _equity.last / cRWA else 1.0
  } else 1.0
}




/**
 *
 *  throws back the bank agent's current amount of RWA
 *
 *    */
def _currentRWA (t:Int):Double = {
  val riskWeightedBusinessLoans = if(_listOfDebtors.isEmpty) 0.0 else _listOfDebtors.map{
    case (firm, listOfLoans) =>
      listOfLoans.map{
        loan =>
          sim.supervisor.riskWeightOfGrantedLoan(loan.borrower) * (loan.principalPayments.filter{
            case (tick, amount) =>
              tick >= t
          }.values.sum + loan.interestPayments.filter{
            case (tick, amount) =>
              tick >= t
          }.values.sum) }.sum
  }.sum
  val riskWeightedIBMLoans = if(_outstandingIBMreceivables.isEmpty) 0.0 else _outstandingIBMreceivables.map{
    case (borrowingBank, loan) =>
      sim.supervisor.riskWeightOfGrantedLoan(borrowingBank) * ((1 + loan.interest) * loan.amountOfReserves) }.sum
  riskWeightedBusinessLoans + riskWeightedIBMLoans
}





/**
 *
 *       Profit and Loss Statement of Banks (every tick)
 *
 *    */
def determineProfit = {
  if(_COGS.isEmpty) profit += _earnings.last else profit += rounded( _earnings.last - _COGS.last )
}





/**
 *
 *  Bank agent's pay corporate tax on their profit of the fiscal year to the government agent.
 *
 *    */
def payTaxes (t:Int, tax:Double = sumOfNPastPeriods(profit, 48) * sim.government.corporateTax.last, cause:String = "corporateTax1") = {
```

```scala
    if(tax > 0){
      println(s"$this has to pay corporate tax of $tax")
      transferMoney(this, sim.government, tax, cause, sim, t)
    }
  }





    /**
     *
     *  Bank agent's distribute the profit after tax in the form of dividends among its owners (households).
     *
     *    */
    def payOutDividends2Owners (t:Int, profitAfterTax:Double = sumOfNPastPeriods(profit, 48) * (1 - sim.government.corporateTax.last), cause:String = "dividends1") {
      val share2Distribute = _currentShareOfRetainedEarnings(t)
      if(share2Distribute > 0 && profitAfterTax > 0){
        if(sim.test) require(owners.nonEmpty, this + " has no owners to pay out dividends!")
        if(sim.pln) println(s"$this has a _currentEquityOfRWA of ${_currentEquityOfRWA(t)} and, thus, can distribute ${_currentShareOfRetainedEarnings(t)} of its profitAfterTax 
($profitAfterTax) to its owners.")
        owners.foreach(hh => transferMoney(this, hh, (share2Distribute * profitAfterTax) * hh.shareOfCorporations(this), cause, sim, t))
      }
    }





    /**
     *
     *  Depending on the imposed regulatory capital buffers, bank agents might be burdened with a temporary payout block of dividends because of
     *  an insufficent loss absobency capacity. If so, they are required to build up their capital until the buffer is fully available for unexpected losses
     *  (according to the basel III accord).
     *
     *    */
    def _currentShareOfRetainedEarnings (t:Int):Double = {
      val capitalBuffer = sim.supervisor.CConB + sim.supervisor.CCycB.last
      if(sim.pln) println(s"age: ${_age}, Eq: ${_equity.last}, currentEqOfRWA or CCQ: ${_currentEquityOfRWA(t)}")
      _currentEquityOfRWA(t) match {
        case eRatio:Double if                                                        eRatio <  sim.supervisor.CAR + 0.25 * capitalBuffer => 0.0
        case eRatio:Double if eRatio >= sim.supervisor.CAR + 0.25 * capitalBuffer && eRatio <  sim.supervisor.CAR + 0.50 * capitalBuffer => 0.2
        case eRatio:Double if eRatio >= sim.supervisor.CAR + 0.50 * capitalBuffer && eRatio <  sim.supervisor.CAR + 0.75 * capitalBuffer => 0.4
        case eRatio:Double if eRatio >= sim.supervisor.CAR + 0.75 * capitalBuffer && eRatio <= sim.supervisor.CAR + 1.00 * capitalBuffer => 0.6
        case eRatio:Double if                                                        eRatio >  sim.supervisor.CAR +        capitalBuffer => 0.8
      }
    }






  /**
```

```
   *
   *  This method throws back the current market share of the bank agent.
   *
   *    */
  def determineCurrentMarketShare = if(_active) _marketShare += roundTo4Digits(_totalAssets.last / sim.bankList.filter(_.active).map(_.totalAssets.last).sum) else marketShare += 0.0



  /**
   *
   *  At the end of each fiscal year, the bank agent makes an annual report to update its balance sheets statements in order to check its solvency and financial soundness.
   *
   *    */
  def makeAnnualReport (t:Int) {time({
    if(_active){
      storeInterestRates
      if(sim.test) checkBankSoBCompleteness(this)
      if(sim.pln) printCompositionOfBonds(t)

      // AR
      updatePVofSoBs(t)
      val TA = rounded( Seq(_businessLoans.last, _interbankLoans.last, bonds.last, _interestReceivables.last, _OSDF.last, _cbReserves.last).sum )
      val TL = rounded( Seq(_retailDeposits.last, _govDeposits.last, _cbLiabilities.last, _interbankLiabilities.last).sum)
      _totalAssets += TA                                                                        // calculate total assets
      if(sim.pln) println("Total assets of " + this + ": " + businessLoans.last + " + " + interbankLoans.last + " + " + bonds.last + " + " + interestReceivables.last + " = " + totalAssets.last)
      _equity     += rounded( TA - TL )                                                         // calculate equity / net worth
      if(sim.pln) println("Equity of " + this + ": " + totalAssets.last + " - (" + retailDeposits.last + " + " + interbankLiabilities.last + ") = " + equity.last)
      if(sim.pln) println("BS after AR:")
      if(sim.pln) printBSP
      if(TA > 1) if(sim.test) require( SE(TA, TL + equity.last), s"Annual Report of $this is not correct: (A) $TA / (L) ${rounded( TL + equity.last )}")


      if(t % 48 == 0 && equity.last < 0){
        if(sim.pln) printBSP
        val currentMarketShare = _totalAssets.last / sim.bankList.filter(_.active).map(_.totalAssets.last).sum
        if(sim.test) require(currentMarketShare <= 1, s"Market share cannot be more than 100%")
        if(currentMarketShare < 0.25 && sim.bankList.filter(_.active).size > 1){
          println(s"$this is shut down due to negative equity...")
          shutDownBank(t)
        } else {
          if(sim.pln) println(s"$this has negative equity and has to be bailed out by the government.")
          _bailOutCounter += t -> currentMarketShare
          sim.government.bailOutLastBank(this, t)
        }
      }
      // store regulatory data
      _RWA         += _currentRWA(t)
      _equityRatio += _currentEquityRatio
      _equityOfRWA += _currentEquityOfRWA(t)
```

```scala
    } else {
      _totalAssets += 0.0
      _equity      += 0.0
    }

 }, "bank_makeAnnualReport", sim)
}




private val A_buL  = ArrayBuffer[Double](0.0)
private val A_baL  = ArrayBuffer[Double](0.0)
private val A_b    = ArrayBuffer[Double](0.0)
private val A_i    = ArrayBuffer[Double](0.0)
private val A_OSDF = ArrayBuffer[Double](0.0)
private val A_r    = ArrayBuffer[Double](0.0)
private val  tA    = ArrayBuffer[Double](0.0)
private val L_rD   = ArrayBuffer[Double](0.0)
private val L_gD   = ArrayBuffer[Double](0.0)
private val L_cbL  = ArrayBuffer[Double](0.0)
private val L_baL  = ArrayBuffer[Double](0.0)
private val   e    = ArrayBuffer[Double](0.0)


val BSPchanges = Map("businessLoans" -> A_buL, "ibLoans" -> A_baL, "bonds" -> A_b, "interest" -> A_i, "OSDF" -> A_OSDF, "reserves" -> A_r, "TA" -> tA,
                     "rDeposits" -> L_rD, "gDeposits" -> L_gD, "cbLiabilities" -> L_cbL, "ibLiabilities" -> L_baL, "equity" -> e
                                                                                                                                   )




/**
 *
 *  This method is just for the convenience of the programmer and enables him to print the current balance sheet of the bank agent as well as the changes of each position
 *  relative to the previous period (in %).
 *
 *    */
def printBSP:Unit = {
  println(f"""
         A                                    $this [seed ${sim.seed}]                              P
        ---------------------------------------------------------------------------------------------
         busiLoan  ${_businessLoans.last}%15.2f ${pDev(_businessLoans, A_buL)} | rDep    ${_retailDeposits.last}%15.2f ${pDev(_retailDeposits, L_rD)}
         ibLoan    ${_interbankLoans.last}%15.2f ${pDev(_retailDeposits, A_baL)} | gDep    ${_govDeposits.last}%15.2f ${pDev(_govDeposits, L_gD)}
         bonds     ${bonds.last}%15.2f ${pDev(bonds, A_b)} | cbLiab  ${_cbLiabilities.last}%15.2f ${pDev(_cbLiabilities, L_cbL)}
         interest  ${_interestReceivables.last}%15.2f ${pDev(_interestReceivables, A_i)} | ibLiab  ${_interbankLiabilities.last}%15.2f ${pDev(_interbankLiabilities, L_baL)}
         OSDF      ${_OSDF.last}%15.2f ${pDev(_OSDF, A_OSDF)} |
         reserves  ${_cbReserves.last}%15.2f ${pDev(_cbReserves, A_r)} | equity  ${if(_equity.nonEmpty) f"${_equity.last}%15.2f" else "NaN"} ${pDev(_equity, e)}
        ---------------------------------------------------------------------------------------------
         TA        ${if(_totalAssets.nonEmpty) f"${_totalAssets.last}%15.2f" else "NaN"} ${pDev(_totalAssets, tA)} |

        insolvecnies: ${_insolvencies.sum}
        bailOuts: ${_bailOutCounter.size}
```

```
        age: ${_age}
        amount of retailClients: ${_retailClients.size}
        amount of businessClients: ${_businessClients.size}
                                                                                  """)

  def pDev (a:ArrayBuffer[Double], b:ArrayBuffer[Double]):String = {
    if(a.size > 1){
      val change  = ((a.last - a.init.last) /        a.init.last) * 100
      val changeE = ((a.last - a.init.last) / _equity.init.last) * 100
      b += changeE
      f"(${change}%+7.2f%%, ${changeE}%+7.2f%%)"
    } else f"(${0}%+7.2f%%, ${0}%+7.2f%%)"
  }


}







/**
 *
 *  This is just to save data produced by the bank agent.
 *
 *    */
  val bankEndOfTickData        = Map()

  val bankEndOfSimulationData = Map(
      "interestOnRetailDeposits"    -> _interestOnRetailDeposits,     // AB[Double]
      "interestOnRetailLoans"       -> _interestOnRetailLoans,        // AB[Double]
      "interestOnInterbankLoans"    -> _interestOnInterbankLoans,     // AB[Double]
      "riskPremium4DoubtfulCredits" -> _riskPremium4DoubtfulCredits,  // AB[Double]
      "reserveTarget"               -> _reserveTarget,                // AB[Double]
      "businessClients"             -> _businessClients,              // AB[Firm]
      "retailClients"               -> _retailClients,                // AB[HH]
      "owners"                      -> owners,                        // AB[HH]
      "profit"                      -> profit,                        // AB[Double]
      "listOfBonds"                 -> listOfBonds,                   // AB[govBond]
      "earnings"                    -> _earnings,                     // AB[Double]
      "NIM"                         -> _NIM,                          // AB[Double]
      "ROE"                         -> _ROE,                          // AB[Double]
      "ROA"                         -> _ROA,                          // AB[Double]
      "RWA"                         -> _RWA,                          // AB{Double]
      "businessLoans"               -> _businessLoans,                // AB[Double]
      "loanLosses"                  -> _loanLosses,                   // AB[Double]
      "bonds"                       -> bonds,                         // AB[Double]
      "interbankLoans"              -> _interbankLoans,               // AB[Double]
      "interestReceivables"         -> _interestReceivables,          // AB[Double]
      "cbReserves"                  -> _cbReserves,                   // AB[Double]
      "totalAssets"                 -> _totalAssets,                  // AB[Double]
      "retailDeposits"              -> _retailDeposits,               // AB[Double]
```

```scala
        "govDeposits"              -> _govDeposits,               // AB[Double]
        "cbLiabilities"            -> _cbLiabilities,             // AB[Double]
        "interbankLiabilities"     -> _interbankLiabilities,      // AB[Double]
        "insolvencies"             -> _insolvencies               // AB[Int]
        "bailOutCounter"           -> _bailOutCounter.size,       // Int
        "equity"                   -> _equity,                    // AB[Double]
        "equityRatio"              -> _equityRatio,               // AB[Double]
        "equityOfRWA"              -> _equityOfRWA                // AB[Double]
        "equityAfterReactivation"  -> _equityAfterReactivation,   // AB[Double]
        "tickOfInsolvency"         -> _tickOfInsolvency,          // AB[Int]
        "marketShare"              -> _marketShare                // AB[Double]
        "test"                     -> test                        // AB[Long]
                                                                                          )


}// End of Bank-Class
```

A.4.2 MMF Class

```scala
/**
 * @author Sebastian Krug
 * @constructor
 * @param name
 * @param numberOfHH
 *
 */

package monEcon.financialSector

import monEcon.Corporation
import monEcon.bonds
import monEcon.realSector._
import monEcon.publicSector._
import monEcon.Simulation

import collection.mutable._
import scala.util.Random
import util.control._




case class MMMF (name            :String,                      //
                 random          :Random,                      //
                 CB              :CentralBank,                 //
                 sim             :Simulation,                  //
                 initialHouseBank:Bank                         //
                                                                                              ) extends Corporation with bonds {

  override def toString = s"MMMF($name)"


  /*    ----------------------------------------    MMMF balance sheet positions    ----------------------------------------    */
  // ----- Assets -----
  private val _claimsFromRepos     = ArrayBuffer(0.0)
  private val _bankDeposits        = ArrayBuffer(0.0)            //
  //  private val bonds                = ArrayBuffer(0.0)        //
  private val _interestReceivables = ArrayBuffer(0.0)            //
  //------------------------------------------------------
  private val _totalAssets         = ArrayBuffer[Double]()       //

  // ----- Liabilities -----
  private val _deposits            = ArrayBuffer(0.0)            //
  private val _interestOnDebt      = ArrayBuffer(0.0)            //
  private val _equity              = ArrayBuffer[Double](1.0)    //




  /**
   *
   *  to save MMMF balance sheet data
   *
```

```scala
 *    */
val MMMFBSP = Map("claimsFromRepos"      -> _claimsFromRepos,
                  "bankDeposits"         -> _bankDeposits,
                  "bonds"                -> bonds,
                  "interestReceivables"  -> _interestReceivables,
                  "deposits"             -> _deposits,
                  "interestOnDebt"       -> _interestOnDebt
                  "totalAssets"          -> _totalAssets,
                  "equity"               -> _equity
                                                                          )





// other data
private val _houseBank                    = ArrayBuffer[Bank](initialHouseBank)
private val _funds2repay                  = collection.mutable.Map[HH,Double]()
private var _active                       = true
private var _periodOfReactivation         = 0
private var _age                          = 0
private val _insolvencies                 = ArrayBuffer[Int](0)
private val _retailClients                = Map[HH, Double]()

// interest spread
private val _interestOnRetailDeposits     = ArrayBuffer[Double]()
private val _feeOnRepos                   = ArrayBuffer[Double]()
private val _haircut                      = ArrayBuffer[Double]()
private val _outstandingRepos             = Queue[overnightRepo]()
private val _earnings                     = ArrayBuffer[Double]()
private val _equityAfterReactivation      = ArrayBuffer[Double]()
private val _causeOfBankruptcy            = Map[String, Int]("ne" -> 0, "illiquidity" -> 0)    // ne = negative, i.e non-positive, equity
private val _BSP                          = Map[Int, String]()

// getter
def houseBank                             = _houseBank.last
def funds2repay                           = _funds2repay
def active                                = _active
def periodOfReactivation                  = _periodOfReactivation
def age                                   = _age
def insolvencies                          = _insolvencies
def interestOnRetailDeposits              = _interestOnRetailDeposits

// BSP
def bankDeposits                          = _bankDeposits
def interestReceivables                   = _interestReceivables
def claimsFromRepos                       = _claimsFromRepos
def outstandingRepos                      = _outstandingRepos
def deposits                              = _deposits
def interestOnDebt                        = _interestOnDebt
def totalAssets                           = _totalAssets
def equity                                = _equity

// other data
```

```scala
def retailClients                          = _retailClients
def earnings                               = _earnings
def equityAfterReactivation                = _equityAfterReactivation
def causeOfBankruptcy                      = _causeOfBankruptcy
def BSP                                    = _BSP










//  ==================================================================================================================================
//  ================================================== PART 1: Investment from HH ==================================================
//  ==================================================================================================================================

/**
 *
 *  In its function of a cash pool, the MMF pays intrest on deposits/invested funds to its investors/households according to the
 *  current interest environment.
 *
 *    */
def interestOnDeposits = CB.targetFFR.last match {
  case i:Double if(i <  0.03) => math.max(i - 0.005, 0.001)
  case i:Double if(i <= 0.05) =>          i - 0.01
  case i:Double if(i >  0.05) =>          i - 0.02
}





/**
 *
 *  test of invested funds of households
 *
 *    */
def checkHHinvestmentRelationship (cause:String) = {time({
    if(_retailClients.nonEmpty && sim.testSB){
      _retailClients.keys.foreach{
        hh =>
          require(
              rounded(retailClients(hh)) == rounded(hh.speculativeFunds(this).map(_._1).sum),
              s"There's a mismatch of claims between $hh (${rounded(hh.speculativeFunds(this).map(_._1).sum)}) /$this (${rounded(retailClients(hh))})"
          )
      }
    }
 }, "MMMF_checkHHinvestmentRelationship", sim)
}


```

```scala
/**
 *
 *  MMF agents pay interest on interest bearing deposits of their customers once a year.
 *
 *     */
def payInterestOnDeposits (t:Int) = {time({
  if(t % 48 == 0){
    _retailClients.keys.foreach{
      hh =>
        require(
            hh.speculativeFunds(this).map(inv => inv._1).sum == _retailClients(hh),
            s"There is a difference between investment info at HH and at MMMF: ${hh.speculativeFunds(this)} (HH) vs. ${_retailClients(hh)} (MMMF)"
        )
    }
    _retailClients.keys.foreach{
        hh =>
          hh.speculativeFunds(this).foreach(investment => transferMoney(this, hh, investment._1 * (1 + investment._2), "interestOnRetailDeposits", sim, t))
    }
  }
 }, "MMMF_payInterestOnDeposits", sim)
}

/**
 *
 *  If investors demand their investments back, the MMF has sufficient liquidity to meet its debt obligations or it is forced to
 *  refuse to roll over repos with BD agents in order to get the needed liquidity the next day. Such a scenario usually results
 *  in BD agents being in serious financial distres.
 *
 *     */
def repayFunds (t:Int) = {time({
  println(s"Before repayFunds of $this in $t: ${_retailClients}")
  if(_funds2repay.nonEmpty){
    val b = new Breaks
    b.breakable{
      _funds2repay.keys.foreach{
        hh =>
          println(s"Before repayFunds of $this with $hh: ${_retailClients}")
          if(_bankDeposits.last >= _funds2repay(hh)) {
            transferMoney(this, hh, _funds2repay(hh), "withdrawDepositsFromMMMF_A", sim, t)
          } else {
            shutDownMMMF(t, "illiquidity")
            b.break
          }
          if(sim.test) require(!_funds2repay.contains(hh), s"funds2repay of $this still contains $hh after repayment of funds...")
      }
      _funds2repay.clear()
    }// breakable
```

```scala
    }
    println(s"After repayFunds of $this in $t: ${_retailClients}")
    checkHHinvestmentRelationship("after repayFunds")
   }, "MMMF_repayFunds", sim)
  }





//  ========================================================================================================================
//  ======================================================= PART 2: Repo with BD =======================================================
//  ========================================================================================================================


  /**
   *
   *  The MMF finances the intrerst paid on investors investments by chargin a haircut on reops with BD agents. Of course,
   *  the haircut earned has to be higher than the intrest paid.
   *
   *    */
  def hairCut = interestOnDeposits + 0.01
  def haircut (ValueOfCollateral:Double, receivedFunds:Double) = rounded( (ValueOfCollateral - receivedFunds) / ValueOfCollateral )



  /**
   *
   *
   *
   *    */
  def offeredAmountOfFunds = math.max(0, _bankDeposits.last - 100)

  def addRepoClaim (repo:overnightRepo)                                = _outstandingRepos.enqueue( repo )
  def getAndRemoveAllRepoClaimsOfBrokerDealer     (  bd:BrokerDealer ) = _outstandingRepos.dequeueAll(_.borrower == bd)

  def getAndRemoveSpecificRepoClaimOfBrokerDealer (repo:overnightRepo):overnightRepo = {
    val cancelledRepo = _outstandingRepos.dequeueAll(_ == repo)
    if(sim.testSB) require(cancelledRepo.size == 1, "There is more than one repo with this identity... ")
    cancelledRepo.head
  }



  /**
   *
   *  The amount of repos which are not rolled over another night, depends on the current liquidity situation of the MMF agents and the
   *  amount of requested funds demanded by investors.
   *
   *    */
  def Decide2RollOverRepos (t:Int) = {time({
    if(_funds2repay.isEmpty || _bankDeposits.last >= _funds2repay.values.sum) {
      if(_outstandingRepos.nonEmpty){
        println(s"$this has outstandingRepos of ${_outstandingRepos}")
```

```
        val repoClients = _outstandingRepos.map { _.borrower }.toSet
        val repos = _outstandingRepos.clone()
        val b = new Breaks
        repoClients.foreach {
          BD =>
            b.breakable{
              repos.dequeueAll { _.borrower == BD }.foreach {
                repo =>
                  require(BD == repo.borrower, s"Damn, here's something wrong: mismatch of $BD in charge of the fee and repo.borrower ${repo.borrower}")
                  if(rounded(repo.overnightFee) > 0.0){
                    if(BD.bankDeposits.last >= repo.overnightFee){
                      transferMoney(BD, this, rounded(repo.overnightFee), "payOvernightFee4RolledOverRepos", sim, t)
                    } else if(sim.regulatedShadowBanks || sim.stricterRegulatedSB){
                      if(sim.centralBankMoneyBD) {
                        val missingLiquidity = rounded(repo.overnightFee - _bankDeposits.last)
                        if(missingLiquidity > 0.0){
                          transferMoney(CB, BD, missingLiquidity, "liquidityInsuranceBD", sim, t)
                          transferMoney(BD, this, rounded(repo.overnightFee), "payOvernightFee4RolledOverRepos", sim, t)
                        }
                      } else {
                        BD.shutDownBrokerDealer(t, "illiquidity")
                        b.break
                      }
                    } else {
                      BD.shutDownBrokerDealer(t, "illiquidity")
                      b.break
                    }
                  }
              }
            }// breakable
        }
      }
    } else {
      println(s"++++++++++++++++++++ $this starts to not roll over the following repos ++++++++++++++++++++")
      var missingAmountOfMoney    = _funds2repay.values.sum - _bankDeposits.last
      println(s"need: $missingAmountOfMoney vs. have: ${_outstandingRepos.map{ repo => repo.repurchasePrice }.sum}")
      val repos2Withdraw          = ArrayBuffer[overnightRepo]()
      var loopCounter             = 0
      val outstandingRepos        = _outstandingRepos.clone()
      while(missingAmountOfMoney > 0 && outstandingRepos.nonEmpty){
        val repoWithClosestVolume = outstandingRepos.map(repo => repo -> squareDeviation(repo.repurchasePrice, missingAmountOfMoney)).toBuffer.toMap.minBy { case(repo, sqDev) => sqDev }._1
        repos2Withdraw           += repoWithClosestVolume
        missingAmountOfMoney     -= repoWithClosestVolume.repurchasePrice
        val repos2drop            = outstandingRepos.dequeueAll(repo => repo == repoWithClosestVolume)
        require(repos2drop.size == 1, s"There has to only a single repo to drop")
        println(s"outstandingRepos after dropping: $outstandingRepos ($loopCounter)")
        loopCounter += 1
      }

      repos2Withdraw.foreach {
        repo2Repay =>
          val cancelledRepo = repo2Repay.borrower.outstandingRepos.dequeueAll { outstandingR => outstandingR == repo2Repay }
          if(sim.test) require(cancelledRepo.size == 1, "There is more than one repo with this identity... ")
```

```scala
        repo2Repay.borrower.notRolledOverRepos.enqueue(repo2Repay.asInstanceOf[repo2Repay.borrower.overnightRepo])
      }
      if(sim.testSB){
        println(s"t=$t: $this doesn't want to roll over repos with the following IDs:")
      }
    }
  }, "MMMF_decide2RollOverRepos", sim)
}





/**
 *
 *  If an MMF agents refuses to roll over a repo agreement with a BD agent in order to meet the liquidity demand
 *  of its investors and the BD agent is not liquid enough to buy back its pledged collateral, then the MMF becomes the
 *  legal owner of the pledged collateral and tries to liquidate it on the financial markets. During the recent financial crisis,
 *  one could observe fire sales with massive declines in asset prices. We tried to incorporate these phenomenons by
 *  selling fire saled collateral at a discount that increases with the amount of BD agent defaults that already happend at
 *  the time of the current fire sale. The logic behind this mechanism is the following: the more BD defaults already happended, the
 *  higher the amount of fire saled assets, the higher the demand on the markets, the lower the price.
 *
 *
 *    */
def fireSaleCollateral (repo:overnightRepo, t:Int) = {time({
  val overnightRepo2FireSale = getAndRemoveSpecificRepoClaimOfBrokerDealer(repo)
  require(overnightReposOfBD.size == 1, "There is more than one outstanding Repo to fireSale")
  fireSaleCollateralForThisRepo2Bank(overnightRepo2FireSale, overnightRepo2FireSale.borrower, t)
 }, "MMMF_fireSaleCollateral", sim)
}




/**
 *
 *    */
def fireSaleCollateralForThisRepo2Bank (repo:overnightRepo, BD:BrokerDealer, t:Int) = {time({
  val buyingBank                        = houseBank
  val bankruptFractionOfFinancialSystem = (sim.bankList.filterNot(_.active).size + sim.MMMFList.filterNot(_.active).size + sim.BrokerDealerList.filterNot(_.active).size) /
(sim.numberOfBanks + sim.numberOfMMMF + sim.numberOfBrokerDealer)
  val discount                          = math.min(0.5, bankruptFractionOfFinancialSystem)
  val price                             = repo.linkedBondIDs.map {
                                            case (id, fraction) =>
                                              BD.PVofSoB(BD.sim.government.findStackOfBondsByID(id), t) * fraction
                                          }.sum * (1 - discount)
  val couponClaimsBeforePurchase        = if(sim.testSB) rounded(sim.government.coupon2Pay.filterKeys(_ > t).filter(_._2.contains(buyingBank)).map(_._2(buyingBank)).sum) else 0.0
  val FVClaimsBeforePurchase            = if(sim.testSB) rounded(   sim.government.dueDebt.filterKeys(_ > t).filter(_._2.contains(buyingBank)).map(_._2(buyingBank)).sum) else 0.0

  var transferedCouponClaims = 0.0
  var transferedFVClaims     = 0.0
  repo.linkedBondIDs.foreach{
    case(id, fraction) =>
```

```
        val purchasedSoB = sim.government.findStackOfBondsByID(id)
        purchasedSoB.bond.ticksOfCouponPayment.filter(_ > t).foreach{
          tick =>
            if(sim.government.coupon2Pay.contains(tick)) {
              if(sim.government.coupon2Pay(tick).contains(buyingBank)){
                sim.government.coupon2Pay(tick)(buyingBank) += purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
              } else sim.government.coupon2Pay(tick) += buyingBank -> purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
            } else sim.government.coupon2Pay += tick -> Map(buyingBank -> purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction)
            sim.government.coupon2PayBD(tick)(BD) -= rounded( purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction )
            transferedCouponClaims += purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
        }
        if(sim.government.dueDebt.contains(purchasedSoB.bond.maturity)) {
          if(sim.government.dueDebt(purchasedSoB.bond.maturity).contains(buyingBank)){
            sim.government.dueDebt(purchasedSoB.bond.maturity)(buyingBank) += purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
          } else sim.government.dueDebt(purchasedSoB.bond.maturity) += buyingBank -> purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
        } else sim.government.dueDebt += purchasedSoB.bond.maturity -> Map(buyingBank -> purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction)
        sim.government.dueDebtBD(purchasedSoB.bond.maturity)(BD) -= rounded( purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction )
        transferedFVClaims += purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
    }
    val couponClaimsAfterPurchase = if(sim.testSB) rounded(sim.government.coupon2Pay.filterKeys(_ > t).filter(_._2.contains(buyingBank)).map(_._2(buyingBank)).sum) else 0.0
    val FVClaimsAfterPurchase     = if(sim.testSB) rounded(   sim.government.dueDebt.filterKeys(_ > t).filter(_._2.contains(buyingBank)).map(_._2(buyingBank)).sum) else 0.0
    if(sim.testSB){
      require(
          SEc(couponClaimsAfterPurchase, rounded(couponClaimsBeforePurchase + transferedCouponClaims), 5),
          s"$buyingBank (${buyingBank.active}) buys fire saled bonds of insolvent $this but COUPON claims are not consistent: claims after purchase ($couponClaimsAfterPurchase) are not equal to claims before ($couponClaimsBeforePurchase) plus transferedCouponCLaims ($transferedCouponClaims)"
          )
      require(
          SEc(    FVClaimsAfterPurchase, rounded(FVClaimsBeforePurchase     + transferedFVClaims),     5),
          s"$buyingBank (${buyingBank.active}) buys fire saled bonds of insolvent $this but FACEVALUE claims are not consistent: claims after purchase ($FVClaimsAfterPurchase) are not equal to claims before ($FVClaimsBeforePurchase) plus transferedFVCLaims ($transferedFVClaims)"
          )
    }

    repo.linkedBondIDs.foreach {
      case (id, fraction) =>
        if(buyingBank.listOfBonds.contains(id)) buyingBank.listOfBonds(id) += fraction else buyingBank.listOfBonds += id -> fraction
        if(fraction < BD.bondsPledgedAsCollateralForRepo(id)) BD.bondsPledgedAsCollateralForRepo(id) -= fraction else BD.bondsPledgedAsCollateralForRepo -= id
    }
    transferMoney(buyingBank, this, price, "fireSaleCollateral", sim, t)
            BD.updatePVofSoBsBD(t)
    buyingBank.updatePVofSoBs(t)
    if(sim.pln) println("After fire sale of bonds:")
    if(sim.pln){
      println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
    }
   }, "MMMF_fireSaleCollateralForThisRepo2Bank", sim)
  }
```

```scala

//  ==================================================================================================================================
//  ============================================== PART 3: Shutdown / reactivate MMMF ==============================================
//  ==================================================================================================================================


  /**
   *
   *  Due to the highly fragile funding model of the money market fund agent, it is likely that the bank-like risks (mainly liquidity risk steming from maturity mismatch) materialize
   *  and the MMF agent is either insolvent or illiquid during the course of the simulation. In such a case, it is resolved and shut down. The current version of the model does not provide a
   *  mechanism that enables the government to bail out systemically important MMF agents.
   *
   *    */
  def shutDownMMMF (t:Int, cause:String) = {time({
    println(s"$this is shut down in $t ($cause)")
    checkHHinvestmentRelationship("before shutDown")


    def repayCapital2Owners = {
      val shareOfDeposits = owners.map(owner => owner -> _bankDeposits.last * owner.shareOfCorporations(this)).toMap
       if(sim.test) require(_bankDeposits.last == shareOfDeposits.values.sum, s"dev is ${_bankDeposits.last} / ${shareOfDeposits.values.sum}")
      owners.foreach{
        owner =>
          if(sim.pln) println(s"Since $this is bankrupt due to neg equity and deposits left it repays ${shareOfDeposits(owner)} to $owner according to its share of the Firm (${owner.shareOfCorporations(this)}).")
          transferMoney(this, owner, shareOfDeposits(owner), "repayCapital", sim, t)
      }
    }



    cause match {
      case "negativeEquity" =>
        _active = false
        _periodOfReactivation = t - (t % 4) + 24 + 4 * random.nextInt(10) + 1
        _insolvencies(_insolvencies.size-1) += 1
        _causeOfBankruptcy("ne") += 1
        storeBSP(t, "ne")

        sim.p(t, "before shut down")

        val BDclients = _outstandingRepos.map { _.borrower }.toSet
        BDclients.foreach {
          bd =>
            println(s"govC before depledge ${sim.government.coupon2PayBD.filter{ case(tick, map) => map.contains(bd) }.map{ case(tick, map) => tick}.toList.sorted}")
            println(s"Before depleding (ticks from repos with $this) ${bd.outstandingRepos.filter( _.lender == this ).map{ _.linkedBondIDs.map{
              case(id, fraction) =>
                sim.government.findStackOfBondsByID(id).bond.ticksOfCouponPayment } }}"
            )
            println(s"Before depleding (ticks from LoB) ${bd.listOfBonds.map{ case(id, fraction) => sim.government.findStackOfBondsByID(id).bond.ticksOfCouponPayment } }")
            bd.outstandingRepos.filter( _.lender == this ).foreach {
              repo =>
```

```
            transferMoney(bd, this, math.min(repo.amountOfMoney, bd.bankDeposits.last), "quitRepoDue2BankruptMMMF", sim, t, repo.repurchasePrice)
            sim.p(t, s"after tranferMoney from ${repo.borrower}")
            bd.dePledgeCollateralOfSpecificRepo(repo, bd.bondsPledgedAsCollateralForRepo)
            println(s"After depleding (ticks from repos with $this) ${bd.outstandingRepos.filter( _.lender == this ).map{ _.linkedBondIDs.map{
              case(id, fraction) =>
                sim.government.findStackOfBondsByID(id).bond.ticksOfCouponPayment } }}"
            )
            println(s"After depleding (ticks from LoB) ${bd.listOfBonds.map{ case(id, fraction) => sim.government.findStackOfBondsByID(id).bond.ticksOfCouponPayment } }")
            println(s"govC before depledge ${sim.government.coupon2PayBD.filter{ case(tick, map) => map.contains(bd) }.map{ case(tick, map) => tick}.toList.sorted}")
            sim.p(t, s"after dePledging from ${repo.borrower}")
            bd.outstandingRepos.dequeueAll { _.lender == this }
            sim.p(t, s"after dequeueing from ${repo.borrower}")
        }
    }
    _outstandingRepos.clear()
    sim.p(t, "collect money from BDs")


    checkHHinvestmentRelationship("before cleaning investment relationship (ne)")
    _retailClients --= _retailClients.filter{ case(claimholder, claim) => claim == 0 }.keys
    _retailClients.filter{ case(claimholder, claim) => claim > 0 }.keys.foreach {
      hh =>
        if(sim.pln) println(s"$hh specFunds: ${hh.speculativeFunds}")
          hh.speculativeFunds(this).toMap.foreach{
            case(investedMoney, interest) =>
              withdraw(        hh.loans, investedMoney + (investedMoney * interest), t, sim)
              withdraw(_deposits,        investedMoney,             t, sim)
              withdraw(_interestOnDebt, (investedMoney * interest), t, sim)
          }
    }
    _retailClients.keys.foreach {
      hh =>
        hh.speculativeFunds -= this
    }
    val sumOfInvestmentsByHH = retailClients.values.sum
    if(bankDeposits.last < sumOfInvestmentsByHH){
      val relativeDebt = retailClients.map{ case (hh, investedMoney) => hh -> rounded( (investedMoney/sumOfInvestmentsByHH) * sumOfInvestmentsByHH) }.toMap
      relativeDebt.foreach{ case (hh, shareOfCash) => transferMoney(this, hh, shareOfCash, "partiallyRepayInvestedDepositsDue2BankruptMMMF", sim, t) }
    } else {
      retailClients.foreach{ case (hh, investment) => transferMoney(this, hh, investment,           "repayInvestedDepositsDue2BankruptMMMF", sim, t) }
    }
    sim.p(t, "repay hh investments")

    if(_bankDeposits.last > 0) repayCapital2Owners
    if(sim.test) require(_bankDeposits.last < 1, s"$this has deposits left after serving debt and equity holders (${_bankDeposits.last})")
    sim.p(t, "repay idle funds to owners")


    if(sim.pln) println( s"TA of shutDown $this --> cFR: ${claimsFromRepos.last}, bD: ${_bankDeposits.last}, b: ${bonds.last}, intR: ${_interestReceivables.last} (all should be 0.0 now)" )
    if(sim.pln) println( s"TL of shutDown $this -->   d: ${_deposits.last},      ioD: ${_interestOnDebt.last}  (all should be 0.0 now)" )

    do{
      owners.foreach{
```

```
          hh =>
            if(hh != null){
              if(sim.test) assert(hh.foundedCorporations.contains(this), hh.foundedCorporations + " does not include " + this + "?")
              hh.foundedCorporations -= this
              if(sim.test) assert(hh.shareOfCorporations.contains(this), hh.shareOfCorporations + " does not include " + this + "?")
              hh.shareOfCorporations -= this
              if(sim.test) assert(owners.contains(hh), owners + " does not include " + hh + "?")
              owners                 -= hh
            }
        }
      } while (owners.nonEmpty)
      if(sim.test) assert(owners.isEmpty, {if(sim.pln) println(owners);    sys.error("There are owners left after shut down")})

      _retailClients.keys.foreach(_.getNewHouseShadowBank)
      _retailClients.clear()
      require(_retailClients.isEmpty, s"_retailClients of $this is not empty: ${_retailClients}")
      _funds2repay.clear()
      require(_funds2repay.isEmpty, s"_funds2repay of $this is not empty: ${_funds2repay}")
      sim.p(t, s"shut down of $this")


    case "illiquidity"    =>
      _active = false
      _periodOfReactivation = t - (t % 4) + 24 + 4 * random.nextInt(10) + 1
      _insolvencies(_insolvencies.size-1) += 1
      _causeOfBankruptcy("illiquidity") += 1
      storeBSP(t, "illiq.")


      val BDclients = _outstandingRepos.map { _.borrower }.toSet
      BDclients.foreach {
        bd =>
          bd.outstandingRepos.filter( _.lender == this ).foreach {
            repo =>
              transferMoney(bd, this, math.min(repo.amountOfMoney, bd.bankDeposits.last), "quitRepoDue2BankruptMMMF", sim, t, repo.repurchasePrice)
              sim.p(t, s"after tranferMoney from ${repo.borrower}")
              bd.dePledgeCollateralOfSpecificRepo(repo, bd.bondsPledgedAsCollateralForRepo)
              sim.p(t, s"after dePledging from ${repo.borrower}")
              bd.outstandingRepos.dequeueAll { _.lender == this }
              sim.p(t, s"after dequeueing from ${repo.borrower}")
          }
      }
      _outstandingRepos.clear()


      println(s"$this is illiquid -> repay HH investments: _retailClients: ${_retailClients}; ")
      checkHHinvestmentRelationship("before cleaning investment relationship (illiquidity)")
      _retailClients --= _retailClients.filter{ case(claimholder, claim) => claim == 0 }.keys
      _retailClients.filter{ case(claimholder, claim) => claim > 0 }.keys.foreach {
        hh =>
          if(sim.pln) println(s"$hh specFunds: ${hh.speculativeFunds}")
            hh.speculativeFunds(this).toMap.foreach{
              case(investedMoney, interest) =>
                withdraw(        hh.loans, investedMoney + (investedMoney * interest), t, sim)
```

```scala
                withdraw(_deposits,        investedMoney,             t, sim)
                withdraw(_interestOnDebt, (investedMoney * interest), t, sim)
            }
      }
      _retailClients.keys.foreach {
        hh =>
          hh.speculativeFunds -= this
      }
      val sumOfInvestmentsByHH = retailClients.values.sum
      if(bankDeposits.last < sumOfInvestmentsByHH){
        val relativeDebt = retailClients.map{ case (hh, investedMoney) => hh -> rounded( (investedMoney/sumOfInvestmentsByHH) * sumOfInvestmentsByHH) }.toMap
        relativeDebt.foreach{ case (hh, shareOfCash) => transferMoney(this, hh, shareOfCash, "partiallyRepayInvestedDepositsDue2BankruptMMMF", sim, t) }
      } else {
        retailClients.foreach{ case (hh, investment) => transferMoney(this, hh, investment,           "repayInvestedDepositsDue2BankruptMMMF", sim, t) }
      }


      if(_bankDeposits.last > 0) repayCapital2Owners
      if(sim.test) require(_bankDeposits.last < 1, s"$this has deposits left after serving debt and equity holders (${_bankDeposits.last})")


      if(sim.pln) println( s"TA of shutDown $this --> cFR: ${claimsFromRepos.last}, bD: ${_bankDeposits.last}, b: ${bonds.last}, intR: ${_interestReceivables.last} (all should be 
0.0 now)" )
      if(sim.pln) println( s"TL of shutDown $this -->   d: ${_deposits.last},      ioD: ${_interestOnDebt.last}  (all should be 0.0 now)" )

      do{
        owners.foreach{
          hh =>
            if(hh != null){
              if(sim.test) assert(hh.foundedCorporations.contains(this), hh.foundedCorporations + " does not include " + this + "?")
              hh.foundedCorporations -= this
              if(sim.test) assert(hh.shareOfCorporations.contains(this), hh.shareOfCorporations + " does not include " + this + "?")
              hh.shareOfCorporations -= this
              if(sim.test) assert(owners.contains(hh), owners + " does not include " + hh + "?")
              owners                 -= hh
            }
        }
      } while (owners.nonEmpty)
      if(sim.test) assert(owners.isEmpty, {if(sim.pln) println(owners);    sys.error("There are owners left after shut down")})

      _retailClients.keys.foreach(_.getNewHouseShadowBank)
      _retailClients.clear()
      require(_retailClients.isEmpty, s"_retailClients of $this is not empty: ${_retailClients}")
      _funds2repay.clear()
      require(_funds2repay.isEmpty, s"_funds2repay of $this is not empty: ${_funds2repay}")



    case _ => sys.error(s"$this has to be shut down since it is bankrupt, but the cause delivered to the shutDownMethod is not correct.")

   }
  }, "MMMF_shutDownMMMF", sim)
 }
```

```scala
/**
 *
 *  A new agent of the MMF-type enters the market if there exist enough HH with sufficient liquidity to found a new one.
 *
 *    */
def reactivateMMMF (t:Int) = {time({
  if(sim.pln) println( s"TA of inactive $this --> cFR: ${_claimsFromRepos.last}, bD: ${_bankDeposits.last}, b: ${bonds.last}, intR: ${_interestReceivables.last}" )
  if(sim.pln) println( s"TL of inactive $this -->   d: ${_deposits.last}, ioD: ${_interestOnDebt.last}" )
  if(_interestReceivables.last > 0) withdraw(_interestReceivables, _interestReceivables.last, t, sim)
  if(      _interestOnDebt.last > 0) withdraw(_interestOnDebt,           _interestOnDebt.last, t, sim)
  if(sim.test){
    require(
        rounded( Seq(_claimsFromRepos.last, _bankDeposits.last, bonds.last, _interestReceivables.last).sum ) < 1,
        s"""Reactivated $this has assets left from bankruptcy:\n ${Seq(_claimsFromRepos, _bankDeposits.last, bonds.last, _interestReceivables.last)}"""
      )
  }
  if(sim.test){
    require(
        rounded( Seq(_deposits.last, _interestOnDebt.last).sum)                                              < 1,
        s"""Reactivated $this has liabs  left from bankruptcy:\n ${Seq(_deposits.last, _interestOnDebt.last)}"""
      )
  }
  if(sim.test) require(_funds2repay.isEmpty, s"$this has still funds2repay after shut down")

  if(sim.test) require(owners.isEmpty, {if(sim.pln) println(owners);    sys.error(s"new activated $this should not have any owners yet")})
  _age                       = 0
  val minEquity              = 250000.0
  val investment             =   5000.0
  val minOfNewInvestors      = (1 * sim.numberOfHH) / sim.numberOfMMMF
  val newOwners              = random.shuffle(sim.hhList.filter(_.bankDeposits.last >= investment))//.take(random.nextInt(sim.numberOfHH/sim.numberOfBanks))
  val newOwnersContribution = newOwners.map(no => no -> investment).toMap
  val recapitalizationGap    = minEquity - newOwnersContribution.values.sum
  if(newOwners.nonEmpty) {
    println(s"$this is reactivated in $t")
    _active = true
    newOwners.foreach{
      hh =>
        owners                 += hh
        hh.foundedCorporations += this
        hh.shareOfCorporations += this -> newOwnersContribution(hh) / newOwnersContribution.values.sum
        if(sim.pln) println(s"$hh founded $this with a share of ${newOwnersContribution(hh) / newOwnersContribution.values.sum}")
    }
    if(sim.test) require(rounded(owners.map(_.shareOfCorporations(this)).sum) == 1, s"${owners.map(_.shareOfCorporations(this)).sum}")
    owners.foreach(owner => transferMoney(owner, this, newOwnersContribution(owner), "reactivateMMMF", sim, t))
    val TA = rounded( Seq(_claimsFromRepos.last, _bankDeposits.last, bonds.last, _interestReceivables.last).sum )
    if(sim.pln) println( s"TA of activated $this --> cFR: ${_claimsFromRepos.last}, bD: ${_bankDeposits.last}, b: ${bonds.last}, intR: ${_interestReceivables.last}" )
    val TL = rounded( Seq(_deposits.last, _interestOnDebt.last).sum)
    if(sim.pln) println( s"TL of activated $this --> rD: ${_deposits.last}, ioD: ${_interestOnDebt.last}" )
```

```
      _equityAfterReactivation += rounded( TA - TL )
    } else {
      _periodOfReactivation = t + 24
      if(sim.pln) println(s"Currently no entrepreneurs around here to reactivate $this, next try will be in t = ${_periodOfReactivation}")
    }
   }, "MMMF_reactivateMMMF", sim)
  }







  /**
   *
   *  This method increases the counter "age" every tick. The counter is reset after a default of the agent. The counter shows the time the agent was able to operate in die
markets.
   *
   *    */
  def updateAge   = _age += 1





  /**
   *
   *  Since MMF agents are also customers of traditional bank agents, they have to search for another bank agent if their
   *  house bank is bankrupt.
   *
   *    */
  def getNewHouseBank = {time({
    val newHouseBank = sim.bankList.filter(_.active)( sim.random.nextInt(sim.bankList.filter(_.active).size) )
    if(sim.test) require(newHouseBank != houseBank && newHouseBank.active)
    _houseBank            += newHouseBank
    houseBank.MMMFClients += this
   }, "MMMF_getNewHouseBank", sim)
  }





  /**
   *
   *  In order to endow newly entered bank agents with some initial demand for their financial services, every customer has a small probability
   *  to switch its house bank every once in while.
   *
   *    */
  def switchHouseBank (t:Int) = {time({
    val listOfNewAndSmallBanks = sim.bankList.filter(bank => bank.active && bank.retailClients.size < (sim.numberOfHH / sim.numberOfBanks) * 0.25)
    val probability2Switch = if(listOfNewAndSmallBanks.nonEmpty) 1.0/sim.numberOfBanks else 0.1
    if(listOfNewAndSmallBanks.nonEmpty && sim.random.nextDouble <= probability2Switch){
```

```
      val newHouseBank = sim.random.shuffle(listOfNewAndSmallBanks).head
      val rDeposits2Transfer = math.max(_bankDeposits.last, 0)
      if(houseBank.cbReserves.last < rDeposits2Transfer) houseBank.getIntraDayLiquidity(rDeposits2Transfer, t)
      withdraw(  houseBank.retailDeposits, rDeposits2Transfer, t, sim)
      withdraw(  houseBank.cbReserves,     rDeposits2Transfer, t, sim)
      deposit(newHouseBank.retailDeposits, rDeposits2Transfer, t, sim)
      deposit(newHouseBank.cbReserves,     rDeposits2Transfer, t, sim)
      houseBank.MMMFClients -= this
      _houseBank            += newHouseBank
      houseBank.MMMFClients += this
    }
   }, "MMMF_switchHouseBank", sim)
  }


// ==================================================================================================================================
// ===================================================== PART 4: Annual Report =====================================================
// ==================================================================================================================================


  /**
   *
   *  At the end of each fiscal year, the MMF agent makes an annual report to update its balance sheets statements in order to check its solvency and financial soundness.
   *
   *    */
  def makeAnnualReport (t:Int) {time({
    if(_active){
      if(sim.test) checkBankSoBCompleteness(this)
      if(sim.pln) printCompositionOfBonds(t)

    // AR
    val TA = rounded( Seq(_claimsFromRepos.last, _bankDeposits.last, bonds.last, _interestReceivables.last).sum )
    val TL = rounded( Seq(_deposits.last, _interestOnDebt.last).sum)
    _totalAssets += TA
    if(sim.pln) println("Total assets of " + this + ": " + businessLoans.last + " + " + interbankLoans.last + " + " + bonds.last + " + " + interestReceivables.last + " = " + totalAssets.last)
    _equity      += rounded( TA - TL )                                                     // calculate equity/net worth
    if(sim.pln) println("Equity of " + this + ": " + totalAssets.last + " - (" + retailDeposits.last + " + " + interbankLiabilities.last + ") = " + equity.last)
    if(TA > 1) if(sim.test) require( SE(TA, TL + _equity.last), s"Annual Report of $this is not correct: (A) $TA / (L) ${rounded( TL + _equity.last )}")
    // check for insolvency
    if(t % 48 == 0 && _equity.last < 0) shutDownMMMF(t, "negativeEquity")
    } else {
      _totalAssets += 0.0
      _equity      += 0.0
    }
   }, "MMMF_makeAnnualReport", sim)
  }
```

```scala


/**
 *
 *  This method prints the MMF agent's current balance sheet.
 *
 *    */
def printBSP = {
println(f"""
         A                 $this                P
       ---------------------------------------------
       cR ${_claimsFromRepos.last}%15.2f  | dep ${_deposits.last}%15.2f
       bd ${_bankDeposits.last}%15.2f  | int  ${_interestOnDebt.last}%15.2f
       iR ${_interestReceivables.last}%15.2f  | eq.  ${if(_equity.nonEmpty) f"${_equity.last}%15.2f" else "NaN"}
       ---------------------------------------------
       TA  ${if(_totalAssets.nonEmpty) f"${_totalAssets.last}%15.2f" else "NaN"}  |
                                                                                                         """)
}



def storeBSP (t:Int, cause:String) = {
_BSP += t -> f"""
         A                 $this [$cause / seed ${sim.seed}]                P
       ---------------------------------------------
       cR ${_claimsFromRepos.last}%15.2f  | dep ${_deposits.last}%15.2f
       bd ${_bankDeposits.last}%15.2f  | int  ${_interestOnDebt.last}%15.2f
       iR ${_interestReceivables.last}%15.2f  | eq.  ${if(_equity.nonEmpty) f"${_equity.last}%15.2f" else "NaN"}
       ---------------------------------------------
       TA  ${if(_totalAssets.nonEmpty) f"${_totalAssets.last}%15.2f" else "NaN"}  |
                                                                                                         """
}



 /**
 *
 *  These values are jsut for data saving purposes.
 *
 *    */
  val MMMFEndOfTickData        = Map()

  val MMMFEndOfSimulationData = Map(
      "interestOnRetailDeposits"    -> _interestOnRetailDeposits,     // AB[Double]
      "interestOnRetailLoans"       -> _interestOnRetailLoans,        // AB[Double]
      "interestOnInterbankLoans"    -> _interestOnInterbankLoans,     // AB[Double]
      "riskPremium4DoubtfulCredits" -> _riskPremium4DoubtfulCredits,  // AB[Double]
      "businessClients"             -> _businessClients,              // AB[Firm]
      "retailClients"               -> _retailClients,                // AB[HH]
      "owners"                      -> owners,                        // AB[HH]
      "profit"                      -> profit,                        // AB[Double]
      "listOfBonds"                 -> listOfBonds,                   // AB[govBond]
      "earnings"                    -> _earnings,                     // AB[Double]
```

```
      "NIM"                        -> _NIM,                          // AB[Double]
      "ROE"                        -> _ROE,                          // AB[Double]
      "ROA"                        -> _ROA,                          // AB[Double]
      "RWA"                        -> _RWA,                          // AB{Double]
      "businessLoans"              -> _businessLoans,                // AB[Double]
      "loanLosses"                 -> _loanLosses,                   // AB[Double]
      "bonds"                      -> bonds,                         // AB[Double]
      "interestReceivables"        -> _interestReceivables,          // AB[Double]
      "totalAssets"                -> _totalAssets,                  // AB[Double]
      "retailDeposits"             -> _retailDeposits,               // AB[Double]
      "insolvencies"               -> _insolvencies,                 // AB[Int]
      "causeOfBankruptcy"          -> _causeOfBankruptcy,            // Map[String, Int]
      "BSP"                        -> _BSP,                          // Map[String, Int]
      "equity"                     -> _equity                        // AB[Double]
      "equityRatio"                -> _equityRatio,                  // AB[Double]
      "equityOfRWA"                -> _equityOfRWA                   // AB[Double]
      "equityAfterReactivation"    -> _equityAfterReactivation,      // AB[Double]
      "tickOfInsolvency"           -> _tickOfInsolvency,             // AB[Int]
      "marketShare"                -> _marketShare                   // AB[Double]
      "test"                       -> test                           // AB[Long]
                                                                                              )






}
```

A.4.3 Broker-dealer Class

```scala
/**
 * @author Sebastian Krug
 * @constructor
 * @param name
 * @param numberOfHH
 *
 */

package monEcon.financialSector

import monEcon.Corporation
import monEcon.bonds
import monEcon.realSector._
import monEcon.publicSector._
import monEcon.Simulation

import collection.mutable._
import collection.immutable.SortedMap

import scala.util.Random
import util.control._




case class BrokerDealer (name            :String,                      //
                         random          :Random,                      //
                         CB              :CentralBank,                 //
                         sim             :Simulation,                  //
                         initialHouseBank:Bank                         //
                                                                                              ) extends Corporation with bonds {

  override def toString = s"BrokerDealer($name)"


  private val _houseBank                 = ArrayBuffer[Bank](initialHouseBank)



  /*    ---------------------------------------    Broker-Dealer balance sheet positions    ---------------------------------------    */
  // ----- Assets -----
      private val _bankDeposits        = ArrayBuffer(0.0)            //
      private val _businessLoans       = ArrayBuffer(0.0)
  //  private val bonds                = ArrayBuffer(0.0)            //
      private val _interestReceivables = ArrayBuffer(0.0)            //
      //------------------------------------------------------
      private val _totalAssets         = ArrayBuffer[Double]()       //

  // ----- Liabilities -----
      private val _deposits            = ArrayBuffer(0.0)            //
      private val _liabsFromRepos      = ArrayBuffer(0.0)            //
      private val _equity              = ArrayBuffer[Double](1.0)    //


```

BrokerDealer.scala

```scala

/**
 *
 *  This is just to save balance sheet data.
 *
 *    */
val brokerDealerBSP = Map("bankDeposits"        -> _bankDeposits,
                          "bonds"               -> bonds,
                          "businessLoans"       -> _businessLoans,
                          "interestReceivables" -> _interestReceivables,
                          "deposits"            -> _deposits,
                          "liabsFromRepos"      -> _liabsFromRepos
                          "totalAssets"         -> _totalAssets,
                          "equity"              -> _equity
                                                                           )



// other data
private val _loanLosses               = ArrayBuffer[Double](0.0)
private val _interestOnRetailLoans    = ArrayBuffer[Double]()
private var _active                   = true
private var _periodOfReactivation     = 0
private var _age                      = 0
private val _insolvencies             = ArrayBuffer[Int](0)
private val _earnings                 = ArrayBuffer[Double]()
private val _outstandingRepos         = Queue[overnightRepo]()
private val _notRolledOverRepos       = Queue[overnightRepo]()
private val _repoVolume               = ArrayBuffer[Double]()
private val _equityAfterReactivation  = ArrayBuffer[Double]()
private val _clients                  = ArrayBuffer[Firm]()
private val _listOfDebtors            = Map[Firm, ArrayBuffer[Loan]]()
private val _causeOfBankruptcy        = Map[String, Int]("ne" -> 0, "illiquidity" -> 0)
private val _BSP                      = Map[Int, String]()


// getter
def houseBank                 = _houseBank.last
def bankDeposits              = _bankDeposits
def businessLoans             = _businessLoans
def interestReceivables       = _interestReceivables
def liabsFromRepos            = _liabsFromRepos
def loanLosses                = _loanLosses
def interestOnRetailLoans     = _interestOnRetailLoans
def listOfDebtors             = _listOfDebtors
def active                    = _active
def periodOfReactivation      = _periodOfReactivation
def age                       = _age
def insolvencies              = _insolvencies
def earnings                  = _earnings
def repoVolume                = _repoVolume
def equityAfterReactivation   = _equityAfterReactivation
def clients                   = _clients
def outstandingRepos          = _outstandingRepos
```

BrokerDealer.scala

```scala
  def notRolledOverRepos        = _notRolledOverRepos
  def totalAssets               = _totalAssets
  def equity                    = _equity
  def causeOfBankruptcy         = _causeOfBankruptcy
  def BSP                       = _BSP










//  ==========================================================================================================
//  ===================================================== PART 1: Repo with MMMF =====================================================
//  ==========================================================================================================

  /**
   *
   *  This method enables the BD to buy government bonds
   *
   *    */
  def buyGovBonds (amount:Double, t:Int) = sim.government.issueNewGovBondsBD(this, amount, t)



  /**
   *
   *  The BD securitizes its loan assets and pledges them as collateral at a money market fund (MMF) that is willing to provide overnight liquidity
   *  via an overnight repo.
   *
   *    */
  def doOvernightRepo (t:Int):Unit = {time({
    if(sim.MMMFList.filter(_.active).size > 0){
      val amountOfCollateral = currentPVofPledgeableBonds(t)
      val mmmf               = sim.MMMFList.filter(_.active).map(monetaryFund => monetaryFund -> squareDeviation(monetaryFund.offeredAmountOfFunds, amountOfCollateral)).minBy{ case (fund, sqDev) => sqDev }._1
      val offeredFunds       = mmmf.offeredAmountOfFunds
      val (repurchasePrice, borrowedFunds) = if(amountOfCollateral <= offeredFunds) ( amountOfCollateral, rounded(amountOfCollateral * (1 - (mmmf.hairCut/365))) ) else ( rounded(offeredFunds * (1 + (mmmf.hairCut/365))), offeredFunds )
      if(borrowedFunds > 1000){
        if(sim.pln) println(s"t:$t -> $this want to do an overnightRepo with: $mmmf; active: ${mmmf.active}; offeredFunds: ${offeredFunds}; borrowedFunds: $borrowedFunds; repurchasePrice: $repurchasePrice but PVofLoB: ${amountOfCollateral}")
        if(sim.pln) println(s"before $this pledgingCollateral: LoB $listOfBonds / $bondsPledgedAsCollateralForRepo")
        val newRepo = overnightRepo(this, mmmf, t, repurchasePrice, borrowedFunds)
        pledgeCollateral(bondsPledgedAsCollateralForRepo, repurchasePrice, newRepo, t)
        transferMoney(mmmf, this, borrowedFunds, "overnightRepo", sim, t, repurchasePrice)
        _outstandingRepos.enqueue( newRepo )
        mmmf.addRepoClaim( newRepo.asInstanceOf[mmmf.overnightRepo] )
        if(sim.pln) println(s"after $this pledgingCollateral: LoB $listOfBonds / $bondsPledgedAsCollateralForRepo")
        if(sim.pln) println(s"$this has borrowed $borrowedFunds from $mmmf: ${newRepo.linkedBondIDs}")
        _repoVolume += repurchasePrice
```

```
    }
  }
 }, "brokerDealer_overnightRepo", sim)
}// method





/**
 *
 *  This method is part of the BD agent's securitization process.
 *
 *    */
def securitizeAndSellLoans (t:Int) = {
  val bL = _businessLoans.last
  require(_businessLoans.last >= 0, s"businessLoans of $this are negative before securitization of Loan portfolio (${_businessLoans.last})")
  val bondValue2Buy = roundDownXk(_businessLoans.last, sim.faceValueOfBonds)
  if(bondValue2Buy > 0){
    transferMoney(sim.government, this, bondValue2Buy, "securitizeLoans", sim, t)
    buyGovBonds (bondValue2Buy, t)
    if(sim.test) println(s"$this has securitized $bondValue2Buy of loans and sold them on the financial markets (before: $bL / now: ${_businessLoans.last})")
  }
  require(_businessLoans.last >= 0, s"businessLoans of $this are negative after securitization of Loan portfolio (${_businessLoans.last})")
}





/**
 *
 *    */
private var _bondsAddedWithBondRelationship  = 0
def bondsAddedWithBondRelationship           = _bondsAddedWithBondRelationship
def updateBondsAddedWithRelationship (i:Int) = _bondsAddedWithBondRelationship += i





/**
 *
 *  BD agents pledge bonds as collateral for overnight repos with an MMF.
 *
 *    */
def pledgeCollateral (map2PutBonds:Map[Long, Double], amount:Double, repo:overnightRepo, t:Int) = {time({
  require(amount > 0, s"You cannot pledge collateral for an amount of $amount.")
  val cPVPB = currentPVofPledgeableBonds(t)
  if(sim.pln) println(s"$this has to pledge $amount and a cPVPB of ${currentPVofPledgeableBonds(t)}; LoB: $listOfBonds")
  val notSufficientPledgableCollateral = if(cPVPB < amount) true else false
  var ability2PledgeCollateral = true
  require(cPVPB >= amount, s"$this does not have enough collateral to pledge")
```

```scala
    if(notSufficientPledgableCollateral){
      val bondValue2Buy = roundUpXk(amount - cPVPB, sim.faceValueOfBonds)
      if(_bankDeposits.last >= bondValue2Buy){
        buyGovBonds (bondValue2Buy, t)
        println(s"LoB after buying new bonds: $listOfBonds")
      } else if(sim.regulatedShadowBanks || sim.stricterRegulatedSB){
        if(sim.centralBankMoneyBD){
          val missingLiquidity = rounded(bondValue2Buy - _bankDeposits.last)
          transferMoney(CB, this, missingLiquidity, "liquidityInsuranceBD", sim, t)
          buyGovBonds (bondValue2Buy, t)
        } else ability2PledgeCollateral = false
      } else ability2PledgeCollateral = false
    }
    if(ability2PledgeCollateral){
      require(
          _bankDeposits.last >= amount - cPVPB,
          s"""$this does not have enough collateral to pledge and not enough deposits to buy enough bonds from Gov: t=$t, amount2pledge: $amount, bD: ${_bankDeposits.last},
            currentPVofPledgeableBonds: ${currentPVofPledgeableBonds(t)}, repo: $repo     """
          )
      val testPVbefore = if(sim.test) currentPVofSoBsBD(t) else 0.0
      var amount2Pledge = amount
      var loopCounter    = 0
      println(s"      $amount has to be pledged.")
      do{
        if(sim.testSB){
          println(s"Is there a head in $this's LOB? ${listOfBonds}; PVofAll: ${currentPVofPledgeableBonds(t)}; PVofHead: ${rounded(PVofSoB(sim.government.findStackOfBondsByID(listOfBonds.head._1), t) * listOfBonds.head._2)}; amount2Pledge: $amount2Pledge")
        }
        if(listOfBonds.head._2 <= 0.0) println(s"LOB.head has a non-positive fraction: ${listOfBonds.head}")
        val SoB                           = listOfBonds.head
        val IDofPledgedSoB:Long           = SoB._1
        val fraction                      = SoB._2
        val PV_SoB                        = PVofSoB(sim.government.findStackOfBondsByID(IDofPledgedSoB), t)
        val fractionOfStack2Pledge:Double = amount2Pledge / ( PV_SoB * fraction )
        println(s"fractionOfStack2Pledge ($fractionOfStack2Pledge) of SoB $SoB: amount2Pledge / ( PV_SoB * fraction) = $amount2Pledge / ( $PV_SoB * $fraction ) ")
        if(sim.pln) println(s"      amount2Pledge: $amount2Pledge | map2PutBonds BEFORE pledging: $map2PutBonds | SoB: ${listOfBonds.head} | PV_SoB: $PV_SoB | fractionOfStack2Pledge: $fractionOfStack2Pledge")
        if(sim.pln) printCompositionOfBonds(t)
        if(map2PutBonds.contains(IDofPledgedSoB)){
          map2PutBonds(IDofPledgedSoB) += math.min(fractionOfStack2Pledge * fraction, fraction) else map2PutBonds += IDofPledgedSoB -> math.min(fractionOfStack2Pledge * fraction, fraction)
        }
        roundTo9Digits(map2PutBonds(IDofPledgedSoB))
        repo.linkedBondIDs += IDofPledgedSoB -> math.min(fractionOfStack2Pledge * fraction, fraction)
        val updatedFractionLOB = fraction - math.min(fractionOfStack2Pledge * fraction, fraction)
        if(fractionOfStack2Pledge >= 1) listOfBonds -= IDofPledgedSoB else listOfBonds += IDofPledgedSoB -> updatedFractionLOB
        amount2Pledge -= (PV_SoB * fraction)
        loopCounter   += 1
        if(sim.pln) println(s"      amount2Pledge: $amount2Pledge | map2PutBonds AFTER pledging: $map2PutBonds | SoB: ${listOfBonds} | PV_SoB: $PV_SoB")
      }while(rounded(amount2Pledge) > 0)
        if(sim.test){
          require( listOfBonds.values.filterNot(_ > 0).isEmpty, s"LOB of $this contains fraction equal to zero: ${listOfBonds}")
          require(map2PutBonds.values.filterNot(_ > 0).isEmpty, s"IDlist of $this contains fraction equal to zero: ${map2PutBonds} (loopCounter = $loopCounter)")
        }
```

```scala
    val testPVafter = if(sim.test) currentPVofSoBs(t) else 0.0
    if(sim.pln) printCompositionOfBonds(t)
    if(sim.test) require(SEc(testPVbefore, testPVafter, 5), s"amount2Pledge: $amount --> PVbefore $testPVbefore | PVafter $testPVafter")
  } else shutDownBrokerDealer(t, "illiquidity")
 }, "BD_pledgeCollateral", sim)
}




/**
 *
 *  This method clears the pledged collateral on the next settlement day, i.e. when the BD has to buy back the pledged collateral.
 *
 *    */
def dePledgeAllCollateral (map2TakeBonds:Map[Long, Double]):Unit = {time({
  map2TakeBonds.foreach{ case(id:Long, fraction:Double) => if(listOfBonds.contains(id)) listOfBonds(id) += fraction else listOfBonds += id -> fraction }
  map2TakeBonds.clear()
 }, "BD_dePledgeCollateral", sim)
}




/**
 *
 *  This method clears the pledged collateral on the next settlement day, i.e. when the BD has to buy back the pledged collateral.
 *
 *    */
def dePledgeCollateralOfSpecificRepo (repo:overnightRepo, map2TakeBonds:Map[Long, Double]):Unit = {time({
  require(repo.linkedBondIDs.nonEmpty)
  repo.linkedBondIDs.foreach {
    case (id, fraction) =>
      if(listOfBonds.contains(id))       listOfBonds(id) += fraction else   listOfBonds += id -> fraction
      if(fraction < map2TakeBonds(id)) map2TakeBonds(id) -= fraction else map2TakeBonds -= id
  }
 }, "BD_dePledgeCollateralOfSpecificRepo", sim)
}





/**
 *
 *  When the repo agreement with a MMF has to be settled on the next settlement day, the BD has to buy back the pledged collateral and repay the
 *  borrowed liquidity to the MMF as long as the MMF decides to not roll over the repo for another night.
 *
 *    */
def repurchaseCollateral (t:Int):Unit = {time({
  if(sim.testSB) sim.testBonds(t, s"Directly before $this is starting to repurchase collateral", "BEFORE")
  val b = new Breaks
  b.breakable{
```

```
      while(_notRolledOverRepos.nonEmpty){
        if(_bankDeposits.last >= _notRolledOverRepos.head.repurchasePrice){
          val overnightRepoToRepay = _notRolledOverRepos.dequeue
          overnightRepoToRepay.lender.getAndRemoveSpecificRepoClaimOfBrokerDealer(overnightRepoToRepay.asInstanceOf[overnightRepoToRepay.lender.overnightRepo])
          transferMoney(this, overnightRepoToRepay.lender, overnightRepoToRepay.repurchasePrice, "repurchaseCollateral", sim, t)
          if(sim.testSB) println(s"$this has repurchased its collateral from ${overnightRepoToRepay.lender} of ${overnightRepoToRepay.repurchasePrice}; IDs: ${overnightRepoToRepay.linkedBondIDs}.")
          dePledgeCollateralOfSpecificRepo(overnightRepoToRepay, bondsPledgedAsCollateralForRepo)
          if(sim.testSB) sim.testBonds(t, s"Directly after $this has repurchased collateral corresponding to $overnightRepoToRepay", "AFTER")
        } else if(sim.regulatedShadowBanks || sim.stricterRegulatedSB){
          if(sim.centralBankMoneyBD){
            val missingLiquidity = rounded(_notRolledOverRepos.head.repurchasePrice - _bankDeposits.last)
              transferMoney(CB, this, missingLiquidity, "liquidityInsuranceBD", sim, t)
              val overnightRepoToRepay = _notRolledOverRepos.dequeue
              overnightRepoToRepay.lender.getAndRemoveSpecificRepoClaimOfBrokerDealer(overnightRepoToRepay.asInstanceOf[overnightRepoToRepay.lender.overnightRepo])
              transferMoney(this, overnightRepoToRepay.lender, overnightRepoToRepay.repurchasePrice, "repurchaseCollateral", sim, t)
              if(sim.testSB) println(s"$this has repurchased its collateral from ${overnightRepoToRepay.lender} of ${overnightRepoToRepay.repurchasePrice}; IDs: ${overnightRepoToRepay.linkedBondIDs}.")
              dePledgeCollateralOfSpecificRepo(overnightRepoToRepay, bondsPledgedAsCollateralForRepo)
              if(sim.testSB) sim.testBonds(t, s"Directly after $this has repurchased collateral corresponding to $overnightRepoToRepay", "AFTER")
          } else {
            shutDownBrokerDealer(t, "illiquidity")
            b.break
          }
        } else {
          if(sim.testSB) println(s"$this cannot repurchase its collateral and must be shut down in $t")
          if(sim.testSB) sim.testBonds(t, s"Directly before shutting $this down since it cannot repurchase its collateral", "BEFORE")
          shutDownBrokerDealer(t, "illiquidity")
          b.break
        }
      }// while
    }// breakable
   }, "BD_repurchaseCollateral", sim)
  }



  def repayCBdebt (t:Int) {
    val amount = math.min(CB.liquidityInsuranceDebtBD(this), _bankDeposits.last)
    if(amount > 0) transferMoney(this, CB, amount, "repayCBdebt", sim, t)
  }





//  ==================================================================================================================================
//  ================================================== PART 2: grant Loans to Firms ==================================================
//  ==================================================================================================================================

  /**
   *
   *  This class defines a BD loan as well as the inherent data like:
   *  - the amount of interest to pay by the firm
```

```scala
 *  - the amounts and periods in which the firm has to pay interest
 *  - the amounts and periods in which the firm has to make principal payments
 *
 *    */
case class Loan (tickOfBorrowing:Int, borrower:Firm, loan:Double, interestRate:Double, maturity:Int = 480) {
  def amountOfInterest (t:Int) = (interestRate * (loan - ( (loan/(maturity/48)) * ((t-tickOfBorrowing)/48)) )) / 12
  val interestPayments  = SortedMap( Vector.tabulate(maturity/ 4)(n => tickOfBorrowing - 1 + (n+1) *  4).map{t => (t, amountOfInterest(t)) }:_* )
  val principalPayments = SortedMap( Vector.tabulate(maturity/48)(n => tickOfBorrowing - 1 + (n+1) * 48).map{t => (t, loan/(maturity/48))  }:_* )
}



/**
 *
 *  BD agents' interest on loans moves in perfect lock-step with the target rate of the CB.
 *  Moreover, the corridor ismore narrow compared to traditional universal banks (i.e. bank agents).
 *
 *    */
def interestOnLoans = CB.targetFFR.last match {
  case i:Double if(i <  0.03) => math.max(i - 0.005, 0.001) + 0.02
  case i:Double if(i <= 0.05) =>          i - 0.01          + 0.02
  case i:Double if(i >  0.05) =>          i - 0.02          + 0.02
}





/**
 *
 *  BD agents request a repo from money market funds and they decide on the hair cut depending on the
 *  BD agent's financial sondness. Then the BD agent can either accept or reject the offer from the fund.
 *
 *    */
def acceptHairCut (haircut:Double, firm:Firm) = if(haircut < interestOnLoans) true else false




/**
 *
 *  BD agents request a repo from money market funds and they decide on the hair cut depending on the
 *  BD agent's financial sondness. Then the BD agent can either accept or reject the offer from the fund.
 *
 *    */
 def decideAboutLoanRequest (corporation:Corporation, requestedAmountOfMoney:Double, t:Int):(Boolean, Double, Double) = {time({
  val repoFees2payTomorrow = rounded(_outstandingRepos.map { _.overnightFee }.sum)
  (true, math.min(requestedAmountOfMoney, math.max(0, _bankDeposits.last - repoFees2payTomorrow - 1000)), interestOnLoans)
   }, "BD_decideAboutLoanRequest", sim)
}



  /**
```

```scala
   *   Tests whether the BD agent complies with
   *   1. the min CAR of 4.5% of RWA
   *   2. the Capital Conservation Buffer (CConB) of 2.5% of RWA on top of CAR
   *   3. the Countercyclical Buffer (CCycB) of 2.5% of RWA on top of CAR + CConB
   *   4. the surcharges on SIBs (1%-2.5%) on top of CAR + CConB + CCycB
   *   5. the non-risk sensitive LR (3%)
   *
   *   Note that these requirements are only imposed in some scenarios, not in general since
   *   BD agents are (currently and so far) part of the unregulated shadow banking sector of the monetary economy.
   *   Moreover, the method contains booleans to either implement
   *   - no financial regulation at all
   *   - the same regulatory framework as bank agents (basel III)
   *   - and an even stricter regulation with quantitatively tightened requirements of basel III.
   *
   *
   *    */
  def proofRegulatoryRequirements (t:Int):Boolean = {time({
    if(sim.regulatedShadowBanks){
      // risk-based measures
      val numberOfActiveBanks = sim.BrokerDealerList.filter(_.active).size
      val currentMarketShare  = _totalAssets.last / sim.BrokerDealerList.filter(_.active).map(_.totalAssets.last).sum
      val surchargeBucket:Int = if(sim.surcharges) currentMarketShare match {
        case marketShare:Double if marketShare <= 1.0 / numberOfActiveBanks => 6     // [20% @ 5 banks] -> equal market share, same size as peers
        case marketShare:Double if marketShare <= 1.3 / numberOfActiveBanks => 5     // [26% @ 5 banks] -> 40,54% larger than avg. peer
        case marketShare:Double if marketShare <= 1.6 / numberOfActiveBanks => 4     // [32% @ 5 banks] -> 88,24% larger than avg. peer
        case marketShare:Double if marketShare <= 1.9 / numberOfActiveBanks => 3     // [20% @ 5 banks]
        case marketShare:Double if marketShare <= 2.2 / numberOfActiveBanks => 2     // [20% @ 5 banks]
        case _                                                               => 1     // [20% @ 5 banks]
      } else 6
      val testCAR = if(sim.stricterRegulatedSB){
        if(_currentEquityOfRWA(t) < 0.1 + 2 * sim.supervisor.surchargesOnSIBs(surchargeBucket)) false else true  // 10% CAR und 2 * 0.035, 0.03, 0.025, 0.015, 0.01 (surcharges)
      } else if(_currentEquityOfRWA(t) < sim.supervisor.CAR + sim.supervisor.surchargesOnSIBs(surchargeBucket)) false else true

      // non-risk based measure
      val testLR = if(sim.LR){
        if(sim.stricterRegulatedSB){
          _currentEquityRatio match {
            case eRatio:Double if eRatio >= 0.1 => true
            case eRatio:Double if eRatio <  0.1 => false
          }
        } else {
          _currentEquityRatio match {
            case eRatio:Double if eRatio >= sim.supervisor.minLeverageRatio => true
            case eRatio:Double if eRatio <  sim.supervisor.minLeverageRatio => false
          }
        }
      } else true

      if(Seq(testCAR, testLR).contains(false)) false else true
    } else true
   }, "BD_proofRegulatoryRequirements", sim)
  }
```

```
/**
 *
 *  throws back the BD agent's equity-to-RWA ratio
 *
 *    */
def _currentEquityOfRWA (t:Int) = {time({
  if(_equity.nonEmpty){
    val cRWA = _currentRWA(t)
    if(cRWA > 0.0) _equity.last / cRWA else 1.0
  } else 1.0
 }, "BD_currentEquityOfRWA", sim)
}





/**
 *
 *  throws back the BD agent's current amount of RWA
 *
 *    */
def _currentRWA (t:Int):Double = {time({
  val riskWeightedBusinessLoans = if(_listOfDebtors.isEmpty) 0.0 else _listOfDebtors.map{
    case (firm, listOfLoans) =>
      listOfLoans.map{
        loan =>
          sim.supervisor.riskWeightOfGrantedLoan(loan.borrower) * (loan.principalPayments.filter{
            case (tick, amount) =>
              tick >= t
          }.values.sum + loan.interestPayments.filter{
            case (tick, amount) =>
              tick >= t
          }.values.sum) }.sum
  }.sum
  riskWeightedBusinessLoans
 }, "BD_currentRWA", sim)
}






/**
 *
 *  throws back the BD agent's current equity ratio
 *
 *    */
def _currentEquityRatio = {time({
  if(_equity.nonEmpty) {
    _equity.last match {
```

```
        case equity:Double if equity == 0.0 => if( (_deposits.last + _liabsFromRepos.last) == 0 ) 1.0 else 0.0
        case equity:Double if equity  > 0.0 => _equity.last / _totalAssets.last
        case _                              => 0.0
      }
    } else 1.0
   }, "BD_currentEquityRatio", sim)
  }





  /**
   *
   *  In analogy to bank agents, BD agents lend money to the real sector. The difference is that they do not have access to and, thus, are not part of the reserve settlement (payment) system.
   *  They are mor like highly risky customers of bank that expose themselves to the same bank-like riks by providing bank0like services withput having access to the public safety net. This
   *  behavior contributes to the build up of systemic risk in the system.
   *  BD agents also do not implement the same assessment concerning the firm's creditworthiness, since their strategy is to distribute the credit risk of the granted loans in their books
   *  by selling the securitized assets on the financial markets. Thus, they have little incentive to invest into due diligence processes.
   *
   *    */
  def grantCredit2Firm (firm:Firm, amount:Double, interest:Double, t:Int):Unit = {time({
    if(sim.test) require(amount >= 0.0, s"The requested amount of $firm is negative: $amount.")
    val grantedLoan = Loan(t, firm, amount, interest)
    if(_listOfDebtors.contains(firm)){
      listOfDebtors(firm) += grantedLoan
    } else listOfDebtors += firm -> ArrayBuffer( grantedLoan )
    CB.credit2privateSector(CB.credit2privateSector.size-1) += amount + grantedLoan.interestPayments.values.sum
    deposit(sim.creditGrantedByBD, amount, t, sim)
    if(sim.pln) println(s"$this grants credit of $amount to $firm since it is creditworthy enough (D/E of ${firm.debt2EquityRatio})")
    transferMoney(this, firm, amount, "grantLoan", sim, t, grantedLoan.interestPayments.values.sum )
   }, "bank_grantCredit2Firm", sim)
  }





  /**
   *
   *  This method serves just to clear repayed debt positions int eh BD's listOfDebtors.
   *
   *    */
  def deleteDueBusinessLoans (t:Int) = _listOfDebtors.foreach{ case (firm, listOfLoans) => _listOfDebtors += firm -> listOfLoans.filterNot(_.principalPayments.last._1 <= t) }
```

```scala
//  ================================================================================================================================
//  =============================================== PART 3: Shut down / reactivate BD ==============================================
//  ================================================================================================================================


  /**
   *
   *  Due to the highly fragile funding model of the BD, it is likely that the bank-like risks materialize in some way and the BD agent is either insolvent or illiquid during the course of the simulation.
   *  In such a case, it is resolved and shut down. The current version of the model does not provide a mechanism that enables the government to bail out systemically important BD agents.
   *
   *    */
  def shutDownBrokerDealer (t:Int, cause:String) = {time({


    def repayCapital2Owners = {
      val shareOfDeposits = owners.map(owner => owner -> _bankDeposits.last * owner.shareOfCorporations(this)).toMap
      if(sim.test) require(_bankDeposits.last == shareOfDeposits.values.sum, s"dev is ${_bankDeposits.last} / ${shareOfDeposits.values.sum}")
      owners.foreach{
        owner =>
          if(sim.pln) println(s"Since $this is bankrupt due to neg equity and deposits left it repays ${shareOfDeposits(owner)} to $owner according to its share of the Firm (${owner.shareOfCorporations(this)}).")
          transferMoney(this, owner, shareOfDeposits(owner), "repayCapital", sim, t)
      }
    }



    def clearFirmLoans = {
      _listOfDebtors.foreach{
        case (firm, listOfLoans) =>
          listOfLoans.foreach{
            loan =>
              val principal2Repay = rounded(loan.principalPayments.filter(_._1 > t).values.sum)
              val  interest2Repay = rounded(loan.interestPayments.filter( _._1 > t).values.sum)
              val     liquidFunds = math.min(firm.bankDeposits.last, principal2Repay + interest2Repay)
              if(sim.pln) println(s"$firm --> principal2Repay: $principal2Repay (BSP: ${firm.debtCapital.last}) + interest2Repay: $interest2Repay (BSP: ${firm.interestOnDebt.last})")
              withdraw(firm.interestOnDebt,  interest2Repay,  t, sim)
              withdraw(firm.debtCapital,     principal2Repay, t, sim)
              withdraw(firm.bankDeposits,    liquidFunds,     t, sim)
```

```
            deposit(     _bankDeposits,    liquidFunds,     t, sim)
            withdraw(_interestReceivables, interest2Repay,  t, sim)
            withdraw(_businessLoans,       math.min(_businessLoans.last, principal2Repay), t, sim)
            if(firm.houseBank != houseBank){
              if(firm.houseBank.cbReserves.last < liquidFunds) firm.houseBank.getIntraDayLiquidity(liquidFunds, t)
              withdraw(firm.houseBank.retailDeposits, liquidFunds,        t, sim)
              deposit(      houseBank.retailDeposits, liquidFunds,        t, sim)
              withdraw(firm.houseBank.cbReserves,     liquidFunds,        t, sim)
              deposit(      houseBank.cbReserves,     liquidFunds,        t, sim)
            }
        }
    }
    _listOfDebtors.clear()
  }




  def fireSaleBonds2Bank (t:Int) = {
    val buyingBank = houseBank
    if(listOfBonds.nonEmpty){
      println(s"$this fireSales LoB to $buyingBank in t=$t: $listOfBonds")
      val bankruptFractionOfFinancialSystem = (sim.bankList.filterNot(_.active).size + sim.MMMFList.filterNot(_.active).size + sim.BrokerDealerList.filterNot(_.active).size) / (sim.numberOfBanks + sim.numberOfMMMF + sim.numberOfBrokerDealer)
      val discount                          = math.min(0.5, bankruptFractionOfFinancialSystem)
      val price                             = PV_LoB(t) * (1 - discount)
      val couponClaimsBeforePurchase        = if(sim.testSB) rounded(sim.government.coupon2Pay.filterKeys(_ > t).filter(_._2.contains(buyingBank)).map(_._2(buyingBank)).sum) else 0.0
      val FVClaimsBeforePurchase            = if(sim.testSB) rounded(   sim.government.dueDebt.filterKeys(_ > t).filter(_._2.contains(buyingBank)).map(_._2(buyingBank)).sum) else 0.0

      var transferedCouponClaims = 0.0
      var transferedFVClaims     = 0.0
      listOfBonds.foreach{
        case(id, fraction) =>
          val purchasedSoB = sim.government.findStackOfBondsByID(id)
          println(s"fireSaledSoB: $purchasedSoB")
          purchasedSoB.bond.ticksOfCouponPayment.filter(_ > t).foreach{
            tick =>
              println(s"t=$t => ticks of couponPayment to transfer in this SoB ${purchasedSoB.bond.ticksOfCouponPayment.filter(_ > t)}\n sim.government.coupon2PayBD: ${sim.government.coupon2PayBD.filter{
                case(tick, map) =>
                  map.contains(this) }.map{ case(tick, map) => tick}.toList.sorted}"
              )
              if(sim.government.coupon2Pay.contains(tick)) {
                if(sim.government.coupon2Pay(tick).contains(buyingBank)){
                  sim.government.coupon2Pay(tick)(buyingBank) += purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
                } else sim.government.coupon2Pay(tick) += buyingBank -> purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
              } else sim.government.coupon2Pay += tick -> Map(buyingBank -> purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction)
              sim.government.coupon2PayBD(tick)(this) -= rounded( purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction )
              transferedCouponClaims += purchasedSoB.bond.coupon * purchasedSoB.amountOfBondsInStack * fraction
          }
          if(sim.government.dueDebt.contains(purchasedSoB.bond.maturity)) {
```

```scala
              if(sim.government.dueDebt(purchasedSoB.bond.maturity).contains(buyingBank)){
                sim.government.dueDebt(purchasedSoB.bond.maturity)(buyingBank) += purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
              } else sim.government.dueDebt(purchasedSoB.bond.maturity) += buyingBank -> purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
            } else sim.government.dueDebt += purchasedSoB.bond.maturity -> Map(buyingBank -> purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction)
            sim.government.dueDebtBD(purchasedSoB.bond.maturity)(this) -= rounded( purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction )
            transferedFVClaims += purchasedSoB.bond.faceValue * purchasedSoB.amountOfBondsInStack * fraction
        }
        val couponClaimsAfterPurchase = if(sim.testSB) rounded(sim.government.coupon2Pay.filterKeys(_ > t).filter(_._2.contains(buyingBank)).map(_._2(buyingBank)).sum) else 0.0
        val FVClaimsAfterPurchase     = if(sim.testSB) rounded(   sim.government.dueDebt.filterKeys(_ > t).filter(_._2.contains(buyingBank)).map(_._2(buyingBank)).sum) else 0.0
        if(sim.testSB){
          require(
              SEc(couponClaimsAfterPurchase, rounded(couponClaimsBeforePurchase + transferedCouponClaims), 5),
              s"$buyingBank buys fire saled bonds of insolvent $this but COUPON claims are not consistent: claims after purchase ($couponClaimsAfterPurchase) are not equal to claims before ($couponClaimsBeforePurchase) plus transferedCouponCLaims ($transferedCouponClaims)"
              )
          require(
              SEc(    FVClaimsAfterPurchase, rounded(FVClaimsBeforePurchase     + transferedFVClaims),     5),
              s"$buyingBank buys fire saled bonds of insolvent $this but FACEVALUE claims are not consistent: claims after purchase ($FVClaimsAfterPurchase) are not equal to claims before ($FVClaimsBeforePurchase) plus transferedFVCLaims ($transferedFVClaims)"
              )
        }
        buyingBank.listOfBonds ++= listOfBonds
        listOfBonds.clear()
        transferMoney(houseBank, this, price, "fireSaleBonds", sim, t)
                   updatePVofSoBsBD(t)
        buyingBank.updatePVofSoBs(t)
        if(sim.pln) println("After fire sale of bonds:")
        if(sim.pln){
          println(s"gDeposits --> ${rounded(sim.bankList.filter(_.active).map(_.govDeposits.last).sum)} / ${sim.government.bankDeposits.last} (Gov); ${sim.bankList.map(bank => bank -> (bank.active, bank.govDeposits.last))} ")
        }
      }
    }







    cause match {
      case "negativeEquity" =>
        _active = false
        _periodOfReactivation = t - (t % 4) + 24 + 4 * random.nextInt(10) + 1
        _insolvencies(_insolvencies.size-1) += 1
        _causeOfBankruptcy("ne") += 1
        storeBSP(t, "ne")

        clearFirmLoans

        while(_outstandingRepos.nonEmpty){
          val repoToRepay = _outstandingRepos.dequeue
          repoToRepay.lender.fireSaleCollateral(repoToRepay.asInstanceOf[repoToRepay.lender.overnightRepo], t)
        }
```

```
        if(sim.testSB) require(_outstandingRepos.isEmpty, s"_outstandingRepos is not empty: ${_outstandingRepos}")
        while(_notRolledOverRepos.nonEmpty){
          val repoToRepay = _notRolledOverRepos.dequeue
          repoToRepay.lender.fireSaleCollateral(repoToRepay.asInstanceOf[repoToRepay.lender.overnightRepo], t)
        }
        if(sim.testSB) require(_notRolledOverRepos.isEmpty, s"_notRolledOverRepos is not empty: ${_notRolledOverRepos}")
        withdraw(_liabsFromRepos, _liabsFromRepos.last, t, sim)
        if(sim.testSB) require(bondsPledgedAsCollateralForRepo.isEmpty, s"bondsPledgedAsCollateralForRepo is not empty after fireSale2Bank (shutDown/ne);
$bondsPledgedAsCollateralForRepo")

        fireSaleBonds2Bank(t)
        if(sim.testSB) require(listOfBonds.isEmpty, s"listOfBonds is not empty after fireSale2Bank (shutDown/ne): $listOfBonds")

        if(_bankDeposits.last > 0) repayCapital2Owners
        if(sim.test) require(_bankDeposits.last < 1, s"$this has deposits left after serving debt and equity holders (${_bankDeposits.last})")

        if(sim.pln) println( s"TA of shutDown $this --> bL: ${_bankDeposits.last}, BL: ${_businessLoans.last}, b: ${bonds.last}, intR: ${_interestReceivables.last} (all should be
0.0 now)" )
        if(sim.pln) println( s"TL of shutDown $this -->  d: ${_deposits.last},    lFR: ${_liabsFromRepos.last}  (all should be 0.0 now)" )

        do{
          owners.foreach{
            hh =>
              if(hh != null){
                if(sim.test) assert(hh.foundedCorporations.contains(this), hh.foundedCorporations + " does not include " + this + "?")
                hh.foundedCorporations -= this
                if(sim.test) assert(hh.shareOfCorporations.contains(this), hh.shareOfCorporations + " does not include " + this + "?")
                hh.shareOfCorporations -= this
                if(sim.test) assert(owners.contains(hh), owners + " does not include " + hh + "?")
                owners                 -= hh
              }
          }
        } while (owners.nonEmpty)
        if(sim.test) assert(owners.isEmpty, {if(sim.pln) println(owners);    sys.error("There are owners left after shut down")})

        _clients.foreach(_.getNewHouseShadowBank)
        _clients.clear()




      case "illiquidity"    =>
        _active = false
        _periodOfReactivation = t - (t % 4) + 24 + 4 * random.nextInt(10) + 1
        _insolvencies(_insolvencies.size-1) += 1
        _causeOfBankruptcy("illiquidity")   += 1
        storeBSP(t, "illiq.")


        clearFirmLoans

        while(_outstandingRepos.nonEmpty){
          val repoToRepay = _outstandingRepos.dequeue
```

```
        repoToRepay.lender.fireSaleCollateral(repoToRepay.asInstanceOf[repoToRepay.lender.overnightRepo], t)
      }
      if(sim.test) require(_outstandingRepos.isEmpty, s"_outstandingRepos is not empty: ${_outstandingRepos}")
      while(_notRolledOverRepos.nonEmpty){
        val repoToRepay = _notRolledOverRepos.dequeue
        repoToRepay.lender.fireSaleCollateral(repoToRepay.asInstanceOf[repoToRepay.lender.overnightRepo], t)
      }
      if(sim.testSB) require(_notRolledOverRepos.isEmpty, s"_notRolledOverRepos is not empty: ${_notRolledOverRepos}")
      withdraw(_liabsFromRepos, _liabsFromRepos.last, t, sim)
      bondsPledgedAsCollateralForRepo.clear()
      if(sim.testSB) require(bondsPledgedAsCollateralForRepo.isEmpty, s"bondsPledgedAsCollateralForRepo is not empty after fireSale2Bank (shutDown/illiq.); $bondsPledgedAsCollateralForRepo")

      fireSaleBonds2Bank(t)
      if(sim.testSB) require(listOfBonds.isEmpty, s"listOfBonds is not empty after fireSale2Bank (shutDown/illiq.): $listOfBonds")

      if(_bankDeposits.last > 0) repayCapital2Owners
      if(sim.test) require(_bankDeposits.last < 1, s"$this has deposits left after serving debt and equity holders (${_bankDeposits.last})")

      if(sim.pln) println( s"TA of shutDown $this --> bL: ${_bankDeposits.last}, BL: ${_businessLoans.last}, b: ${bonds.last}, intR: ${_interestReceivables.last} (all should be 0.0 now)" )
      if(sim.pln) println( s"TL of shutDown $this -->  d: ${_deposits.last},    lFR: ${_liabsFromRepos.last}  (all should be 0.0 now)" )

      do{
        owners.foreach{
          hh =>
            if(hh != null){
              if(sim.test) assert(hh.foundedCorporations.contains(this), hh.foundedCorporations + " does not include " + this + "?")
              hh.foundedCorporations -= this
              if(sim.test) assert(hh.shareOfCorporations.contains(this), hh.shareOfCorporations + " does not include " + this + "?")
              hh.shareOfCorporations -= this
              if(sim.test) assert(owners.contains(hh), owners + " does not include " + hh + "?")
              owners                 -= hh
            }
        }
      } while (owners.nonEmpty)
      if(sim.test) assert(owners.isEmpty, {if(sim.pln) println(owners);    sys.error("There are owners left after shut down")})

      _clients.foreach(_.getNewHouseShadowBank)
      _clients.clear()




    case _ => sys.error(s"$this has to be shut down since it is bankrupt, but the cause delivered to the shutDownMethod is not correkt.")

  }// match

  require(_businessLoans.last >= 0, s"businessLoans of $this are negative after shutDown of ($cause) (${_businessLoans.last})")

 }, "BD_shutDownBrokerDealer", sim)
}// method
```

```scala






/**
 *
 *  After a resolution of a BD agent, there is a possibility that a new BD enters the market (from a technical point of view, the entirely cleaned but already existing BD object is reactivated) if there are
 *  enough HH that provide sufficient liquidity to found a new BD.
 *
 *    */
def reactivateBrokerDealer (t:Int) = {time({
  println( s"TA of inactive $this --> bD: ${_bankDeposits.last}, bL: ${_businessLoans.last}, b: ${bonds.last}, intR: ${_interestReceivables.last}" )
  println( s"TL of inactive $this -->  d: ${_deposits.last},    lFR: ${_liabsFromRepos.last}" )
  println("current BSP:")
  printBSP
  println(s"LOB of $this before reactivation: ${listOfBonds}")
  println(s"bondsPledgedAsCollateralForRepo of $this before reactivation: ${bondsPledgedAsCollateralForRepo}")
  println(s"${_causeOfBankruptcy}")
  println("BSP at shutdown")
  println(s"${_BSP.last}")
  println("..........................................................................................")
  if(_interestReceivables.last > 0) withdraw(_interestReceivables, interestReceivables.last, t, sim)
  if(sim.test){
    require(
        rounded( Seq(_bankDeposits.last, _businessLoans.last, bonds.last, _interestReceivables.last).sum ) < 1,
        s"""Reactivated $this has assets left from bankruptcy:\n ${Seq(_bankDeposits.last, _businessLoans.last, bonds.last, _interestReceivables.last)}"""
     )
  }
  if(sim.test){
    require(
        rounded( Seq(_deposits.last, _liabsFromRepos.last).sum)                                            < 1,
        s"""Reactivated $this has liabs  left from bankruptcy:\n ${Seq(_deposits.last, _liabsFromRepos.last)}"""
      )
  }


  if(sim.test) require(owners.isEmpty, {if(sim.pln) println(owners);    sys.error(s"new activated $this should not have any owners yet")})
  _age                      = 0
  val minEquity             = 250000.0
  val investment            =   5000.0
  val minOfNewInvestors     = (1 * sim.numberOfHH) / sim.numberOfBrokerDealer
  val newOwners             = random.shuffle(sim.hhList.filter(_.bankDeposits.last >= investment))
  val newOwnersContribution = newOwners.map(no => no -> investment).toMap
  val recapitalizationGap   = minEquity - newOwnersContribution.values.sum
  if(newOwners.nonEmpty) {
    _active = true
    newOwners.foreach{
      hh =>
        owners                 += hh
        hh.foundedCorporations += this
        hh.shareOfCorporations += this -> newOwnersContribution(hh) / newOwnersContribution.values.sum
```

```scala
        if(sim.pln) println(s"$hh founded $this with a share of ${newOwnersContribution(hh) / newOwnersContribution.values.sum}")
    }
    if(sim.test) require(rounded(owners.map(_.shareOfCorporations(this)).sum) == 1, s"${owners.map(_.shareOfCorporations(this)).sum}")
    owners.foreach(owner => transferMoney(owner, this, newOwnersContribution(owner), "reactivateBrokerDealer", sim, t))
    println(s"$this is reactivated with LOB: ${listOfBonds}")
    println(s"$this is reactivated with bondsPledgedAsCollateralForRepo: ${bondsPledgedAsCollateralForRepo}")
    println(s"${_causeOfBankruptcy}")
    printBSP
    println("==============================================================================================================================================")
    updatePVofSoBsBD(t)
    val TA = rounded( Seq(_bankDeposits.last, _businessLoans.last, bonds.last, _interestReceivables.last).sum )
    if(sim.pln) println( s"TA of activated $this --> bD: ${_bankDeposits.last}, bL: ${_businessLoans.last}, b: ${bonds.last}, intR: ${_interestReceivables.last}" )
    val TL = rounded( Seq(_deposits.last, _liabsFromRepos.last).sum)
    if(sim.pln) println( s"TL of activated $this --> rD: ${_deposits.last}, lFR: ${_liabsFromRepos.last}" )
    _equityAfterReactivation += rounded( TA - TL )
  } else {
    _periodOfReactivation = t + 24
    if(sim.pln) println(s"Currently no entrepreneurs around here to reactivate $this")
  }
 }, "BD_reactivateBrokerDealer", sim)
}


/**
 *
 *    */
def updateAge   = _age += 1


/**
 *
 *  Since BD agents are also customers of traditional bank agents, they have to search for another bank agent if their
 *  house bank is bankrupt.
 *
 *    */
def getNewHouseBank = {time({
  val newHouseBank = sim.bankList.filter(_.active)( sim.random.nextInt(sim.bankList.filter(_.active).size) )
  if(sim.test) require(newHouseBank != houseBank && newHouseBank.active)
  _houseBank          += newHouseBank
  houseBank.BDClients += this
 }, "BD_getNewHouseBank", sim)
}
```

```scala

  /**
   *
   *  In order to endow newly entered bank agents with some initial demand for their financial services, every customer has a small probability
   *  to switch its house bank every once in while.
   *
   *    */
  def switchHouseBank (t:Int) = {time({
    val listOfNewAndSmallBanks = sim.bankList.filter(bank => bank.active && bank.retailClients.size < (sim.numberOfHH / sim.numberOfBanks) * 0.25)
    val probability2Switch = if(listOfNewAndSmallBanks.nonEmpty) 1.0/sim.numberOfBanks else 0.1
    if(listOfNewAndSmallBanks.nonEmpty && sim.random.nextDouble <= probability2Switch){
      val newHouseBank = sim.random.shuffle(listOfNewAndSmallBanks).head
      val rDeposits2Transfer = math.max(_bankDeposits.last, 0)
      if(houseBank.cbReserves.last < rDeposits2Transfer) houseBank.getIntraDayLiquidity(rDeposits2Transfer, t)
      withdraw(  houseBank.retailDeposits, rDeposits2Transfer, t, sim)
      withdraw(  houseBank.cbReserves,     rDeposits2Transfer, t, sim)
      deposit(newHouseBank.retailDeposits, rDeposits2Transfer, t, sim)
      deposit(newHouseBank.cbReserves,     rDeposits2Transfer, t, sim)
      houseBank.BDClients -= this
      _houseBank          += newHouseBank
      houseBank.BDClients += this
    }
   }, "BD_switchHouseBank", sim)
  }















//  ==================================================================================================================================
//  ===================================================== PART 4: Annual Report ======================================================
//  ==================================================================================================================================







  /**
   *
   *  At the end of each fiscal year, the BD agent makes an annual report to update its balance sheets statements in order to check its solvency and financial soundness.
   *
   *    */
  def makeAnnualReport (t:Int) {time({
```

```
    if(_active){
      if(sim.test) checkBankSoBCompleteness(this)
      if(sim.pln) printCompositionOfBonds(t)

      // AR
      updatePVofSoBsBD(t)
      val TA = rounded( Seq(_bankDeposits.last, _businessLoans.last, bonds.last, _interestReceivables.last).sum )
      val TL = rounded( Seq(_deposits.last, _liabsFromRepos.last).sum)
      _totalAssets += TA
      if(sim.pln) println("Total assets of " + this + ": " + businessLoans.last + " + " + interbankLoans.last + " + " + bonds.last + " + " + interestReceivables.last + " = " + totalAssets.last)
      _equity      += rounded( TA - TL )                                                  // calculate equity / net worth
      if(sim.pln) println("Equity of " + this + ": " + totalAssets.last + " - (" + retailDeposits.last + " + " + interbankLiabilities.last + ") = " + equity.last)
      if(TA > 1) if(sim.test) require( SE(TA, TL + _equity.last), s"Annual Report of $this is not correct: (A) $TA / (L) ${rounded( TL + _equity.last )}")
      // check for insolvency
      if(t % 48 == 0 && _equity.last < 0) shutDownBrokerDealer(t, "negativeEquity")
    } else {
      _totalAssets += 0.0
      _equity      += 0.0
    }
   }, "BD_makeAnnualReport", sim)
  }






  /**
   *
   *  This method prints the BD agent's current balance sheet.
   *
   *    */
    def printBSP = {
  println(f"""
          A                 $this                P
        ---------------------------------------------
        bL ${_businessLoans.last}%15.2f  | dep  ${_deposits.last}%15.2f
        bd ${_bankDeposits.last}%15.2f  | lfR  ${_liabsFromRepos.last}%15.2f
        b ${bonds.last}%15.2f  |
        iR ${_interestReceivables.last}%15.2f  | eq.  ${if(_equity.nonEmpty) f"${_equity.last}%15.2f" else "NaN"}
        ---------------------------------------------
        TA   ${if(_totalAssets.nonEmpty) f"${_totalAssets.last}%15.2f" else "NaN"}  |
                                                                                                   """)
}







    def storeBSP (t:Int, cause:String) = {
  _BSP += t -> f"""
```

```
            A                  $this [$cause / seed ${sim.seed}]                P
          ---------------------------------------------
          bL ${_businessLoans.last}%15.2f  | dep  ${_deposits.last}%15.2f
          bd ${_bankDeposits.last}%15.2f  | lfR  ${_liabsFromRepos.last}%15.2f
          b ${bonds.last}%15.2f  |
          iR ${_interestReceivables.last}%15.2f  | eq.  ${if(_equity.nonEmpty) f"${_equity.last}%15.2f" else "NaN"}
          ---------------------------------------------
          TA   ${if(_totalAssets.nonEmpty) f"${_totalAssets.last}%15.2f" else "NaN"}  |
                                                                                                        """
}









  /**
   *
   *  These values are jsut for data saving purposes.
   *
   *    */

  val brokerDealerEndOfTickData        = Map()

  val brokerDealerEndOfSimulationData = Map(
      "interestOnRetailDeposits"    -> _interestOnRetailDeposits,     // AB[Double]
      "interestOnRetailLoans"       -> _interestOnRetailLoans,        // AB[Double]
      "interestOnInterbankLoans"    -> _interestOnInterbankLoans,     // AB[Double]
      "riskPremium4DoubtfulCredits" -> _riskPremium4DoubtfulCredits,  // AB[Double]
      "reserveTarget"               -> _reserveTarget,                // AB[Double]
      "businessClients"             -> _businessClients,              // AB[Firm]
      "retailClients"               -> _retailClients,                // AB[HH]
      "owners"                      -> owners,                        // AB[HH]
      "profit"                      -> profit,                        // AB[Double]
      "listOfBonds"                 -> listOfBonds,                   // AB[govBond]
      "earnings"                    -> _earnings,                     // AB[Double]
      "NIM"                         -> _NIM,                          // AB[Double]
      "ROE"                         -> _ROE,                          // AB[Double]
      "ROA"                         -> _ROA,                          // AB[Double]
      "RWA"                         -> _RWA,                          // AB{Double]
      "businessLoans"               -> _businessLoans,                // AB[Double]
      "loanLosses"                  -> _loanLosses,                   // AB[Double]
      "bonds"                       -> bonds,                         // AB[Double]
      "interbankLoans"              -> _interbankLoans,               // AB[Double]
      "interestReceivables"         -> _interestReceivables,          // AB[Double]
      "cbReserves"                  -> _cbReserves,                   // AB[Double]
      "totalAssets"                 -> _totalAssets,                  // AB[Double]
      "retailDeposits"              -> _retailDeposits,               // AB[Double]
      "govDeposits"                 -> _govDeposits,                  // AB[Double]
      "cbLiabilities"               -> _cbLiabilities,                // AB[Double]
      "interbankLiabilities"        -> _interbankLiabilities,         // AB[Double]
      "insolvencies"                -> _insolvencies,                 // AB[Int]
```

BrokerDealer.scala

```
        "causeOfBankruptcy"          -> _causeOfBankruptcy,          // Map[String, Int]
        "BSP"                        -> _BSP,                        // Map[String, Int]
        "bailOutCounter"             -> _bailOutCounter.size,        // Int
        "equity"                     -> _equity                      // AB[Double]
        "equityRatio"                -> _equityRatio,                // AB[Double]
        "equityOfRWA"                -> _equityOfRWA                 // AB[Double]
        "equityAfterReactivation"    -> _equityAfterReactivation,    // AB[Double]
        "tickOfInsolvency"           -> _tickOfInsolvency,           // AB[Int]
        "marketShare"                -> _marketShare                 // AB[Double]
        "test"                       -> test                         // AB[Long]
                                                                                                  )

}
```

A.5 Real Sector

A.5.1 Household Class

HH.scala

```
/**
 *
 */
package monEcon.realSector

import monEcon.Agent
import monEcon.Simulation
import monEcon.Corporation
import monEcon.ARGE
import monEcon.publicSector._
import monEcon.financialSector._
import monEcon.Markets._
import monEcon.bonds          // trait

import scala.util.Random

import scala.collection.mutable._


/**
 * @author Sebastian Krug
 *
 */

case class HH (name                        :String,          //
               numberOfHH                  :Int,             //
               numberOfFirms               :Int,             //
               numberOfBanks               :Int,             //
               tradBanks                   :Boolean,         //
               random                      :Random,          //
               initialHouseBank            :Bank,            //
               initialHouseShadowBank      :MMMF,            //
               goodsMarket                 :GoodsMarket,     //
               laborMarket                 :LaborMarket,     //
               interbankMarket             :InterbankMarket, //
               government                  :Government,      //
               initialLaborSkill           :Double,          //
               initialEmployer             :Corporation,     //
               willingness2Spend           :Double,          //
               initialRiskAversionParameter:Double,          //
               initialConfidenceLevel      :Double,          //
               vacancyAvailabilityParameter:Double,          //
               goodAvailabilityParameter   :Double,          //
               sim                         :Simulation       //
                                                                   ) extends Agent with bonds {

  override def toString = s"HH($name)"


  /*  ----------------------------------------  hh balance sheet positions  ----------------------------------------  */
  // Asset Side
    private val _equityStake  = ArrayBuffer(0.0)      //
    private val _loans        = ArrayBuffer(0.0)      //
```

```
    private val _bankDeposits = ArrayBuffer(0.0)      //
//  private val bonds         = ArrayBuffer(0.0)      //
    private val _cash         = ArrayBuffer(0.0)      //
    //--------------------------------------------------
    private val _totalAssets  = ArrayBuffer[Double]()   //

  // Liability Side
  private val _equity         = ArrayBuffer[Double]()   // net worth/wealth of HH



  /**
   *
   *  This is just to save balance sheet data.
   *
   *    */
  val hhBSP = Map("equityStake"   -> _equityStake,
                  "loans"         -> _loans,
                  "bankDeposits"  -> _bankDeposits,
                  "bonds"         -> bonds,
                  "cash"          -> _cash
                                                    )


  private val _foundedCorporations = ArrayBuffer[Corporation]()
  private val _shareOfCorporations = Map[Corporation, Double]()
  private val _speculativeFunds    = Map[MMMF, ArrayBuffer[(Double,Double)]]()          // ArrayBuffer[(amount, interest)]

  // Other Data
  private val _houseBank              = ArrayBuffer[Bank](initialHouseBank)
  private val _houseShadowBank        = ArrayBuffer[MMMF](initialHouseShadowBank)
  private val _reservationWage        = ArrayBuffer(0.0)                               //
  private val _periodsOfUnemployment  = ArrayBuffer(0)                                 //
  private val _unemployed             = ArrayBuffer(true)                              //
          var  currentEmployer        = initialEmployer                                //
  private val _employers              = ArrayBuffer[Corporation]()                     //
  private val _laborSkillFactor       = ArrayBuffer(initialLaborSkill)                 //

  private val _privateBorrower        = Map[Firm,(Double, Double, Int,Double)]()       //
  private val _interestOnLoans        = ArrayBuffer(0.05)                              //

  private val _willingness2Consume    = ArrayBuffer(willingness2Spend)                 //
  private val _riskAversionParameter  = ArrayBuffer((initialRiskAversionParameter, sim.initialTargetRate, initialConfidenceLevel))
  private val _amount2Spend           = ArrayBuffer(0.0)                               //

  // getter
  def houseBank                       = _houseBank.last                                //
  def houseShadowBank                 = _houseShadowBank.last                          //
  def equityStake                     = _equityStake
  def loans                           = _loans
  def bankDeposits                    = _bankDeposits
  def cash                            = _cash
  def equity                          = _equity
```

```
def reservationWage                 = _reservationWage
def periodsOfUnemployment           = _periodsOfUnemployment
def unemployed                      = _unemployed
def employers                       = _employers
def foundedCorporations             = _foundedCorporations
def shareOfCorporations             = _shareOfCorporations
def speculativeFunds                = _speculativeFunds
def laborSkillFactor                = _laborSkillFactor
def privateBorrower                 = _privateBorrower
def interestOnLoans                 = _interestOnLoans
def willingness2Consume             = _willingness2Consume
def riskAversionParameter           = _riskAversionParameter
def amount2Spend                    = _amount2Spend

// setter
def reservationWage_+=       (value:Double) :Unit = _reservationWage        += value                                //
def periodsOfUnemployment_+= (value:Int)    :Unit = _periodsOfUnemployment  += _periodsOfUnemployment.last + value  //
def unemployed_+=            (value:Boolean):Unit = _unemployed             += value                                //



/*  --------------------  Tick Routine of HH  --------------------  */
def tickRoutineHH (t:Int) = {
  if(sim.pln) println(" ----- HH search a Job -----")
  searchJob(t)
}
/*  ----------------------------------------------------------------  */


/*  Methods of
HH  ----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  */


/**
 *
 *  The individual labor skill of each HH improves over time according to its ability to get employed.
 *  --> less time of employment --> less learining on the job --> less improvement of the labor skill over time.
 *
 *    */
def updateLaborSkill {time({
  laborSkillFactor update(laborSkillFactor.length-1, rounded( laborSkillFactor.last * (1 + (2.4 / (96 + 2 * periodsOfUnemployment.last) )) ) )
  periodsOfUnemployment += 0
 }, "hh_updateLaborSkill", sim)
}



/**
 *
 *  If HH has cash, he looks at his confidence level an invests accordingly.
 *  The volume and probability of the investment depends on the individual risk aversion and the current publicConfidenceLevel
 *  to mimick common market phenomena like herding, myopia, procyclicality...
```

```scala
   *
   *    */
  def adjustSpeculativeFunds (t:Int) = {time({
    val currentSpeculativeFunds = _speculativeFunds.values.map(AB => AB.map(_._1).sum).sum
    val PCL                     = sim.publicConfidenceLevel.last
    val PCLbenchmark            = if(sim.publicConfidenceLevel.size >= 48) sim.publicConfidenceLevel(sim.publicConfidenceLevel.size - 48) else sim.publicConfidenceLevel.last / 2
    val upperBound              = 0.9    // 90%
    val lowerBound              = 0.5    // 50%
    if(houseShadowBank.active){
      println(s"+++++++++++++++ $this is adjusting its specFunds +++++++++++++++")
      println(s"investing instead of withdrawing: PCL $PCL >= ${PCLbenchmark * upperBound} PCLbenchmark * 0.9")
      if(PCL >= PCLbenchmark * upperBound){
        if(t <= 48 || random.nextDouble <= 1 - _riskAversionParameter.last._1){
          println(s"t=$t <= 48 || prob <= 1-${_riskAversionParameter.last._1}")
          val amount2Invest = if(_bankDeposits.last > 1000) (_bankDeposits.last - 1000) * (0.5 - _riskAversionParameter.last._1) else 0.0
          val interest      = houseShadowBank.interestOnDeposits
          println(s"$this wants to invest $amount2Invest at $interest% (int2pay: ${interest * amount2Invest})")
          if(amount2Invest > 0){
            transferMoney(this, houseShadowBank, amount2Invest, "investDeposits@MMMF", sim, t, interest * amount2Invest)
            deposit(sim.investmentSBsector, amount2Invest, t, sim)
            if(_speculativeFunds.isEmpty){
              _speculativeFunds              += houseShadowBank -> ArrayBuffer((amount2Invest, interest))
              houseShadowBank.retailClients += this -> amount2Invest
            } else {
              _speculativeFunds(houseShadowBank)  += {(amount2Invest, interest)}
              houseShadowBank.retailClients(this) += amount2Invest
            }
          }
        }
      } else if(currentSpeculativeFunds > 0 && PCL < PCLbenchmark * lowerBound) {
        if(random.nextDouble <= 0.5 + _riskAversionParameter.last._1){
          val amount2Withdraw = if(currentSpeculativeFunds > 1000) currentSpeculativeFunds * (0.5 + _riskAversionParameter.last._1) else 0.0
          deposit(sim.withdrawFromSBsector, amount2Withdraw, t, sim)
          require(!houseShadowBank.funds2repay.contains(this), s"${houseShadowBank} [${houseShadowBank.active}] has already funds to repay to $this (t=$t); repay: ${houseShadowBank.funds2repay}; amount2Withdraw $amount2Withdraw; invest: ${houseShadowBank.retailClients(this)}; specFundsOfHH: ${_speculativeFunds}; current: $currentSpeculativeFunds")
          houseShadowBank.funds2repay += this -> amount2Withdraw
          if(amount2Withdraw < houseShadowBank.retailClients(this)) houseShadowBank.retailClients(this) -= amount2Withdraw else houseShadowBank.retailClients -= this
          var x = amount2Withdraw
          try{
            println(s"amount2Withdraw of $this from $houseShadowBank: $x")
            do{
              val (amount, interestRate) = _speculativeFunds(houseShadowBank).head
              println(s"$this withdraws $amount from specFunds related to $houseShadowBank: ${_speculativeFunds}: before")
              if(amount >= x) {
                transferMoney(houseShadowBank, this, amount * interestRate, "withdrawDepositsFromMMMF_B", sim, t)
                _speculativeFunds(houseShadowBank)(0) = ( amount - x, interestRate )
                x -= amount
              } else {
                transferMoney(houseShadowBank, this, amount * interestRate, "withdrawDepositsFromMMMF_B", sim, t)
                x -= amount
                _speculativeFunds(houseShadowBank) -= _speculativeFunds(houseShadowBank)(0)
              }
```

```scala
            println(s"$this withdraws $amount (x: $x) from specFunds related to $houseShadowBank: ${_speculativeFunds}: after")
          }while(x > 0)
        } catch {
          case e:Exception => sys.error(s"t=$t, bD: ${_bankDeposits.last}, currentSF: $currentSpeculativeFunds; _speculativeFunds: ${_speculativeFunds}")
        }//
      }//
    }
  }//
  if(houseShadowBank.retailClients.contains(this) && houseShadowBank.retailClients(this) == 0.0) houseShadowBank.retailClients -= this
  if(houseShadowBank.retailClients.contains(this)) require(houseShadowBank.retailClients(this) > 0, s"The invested amount in MMMF must be positive: ${houseShadowBank.retailClients(this)}")
 }, "hh_adjustSpeculativeFunds", sim)
}




/**
 *
 *  Since HH agents are customers of traditional bank agents, they have to search for another bank agent if their
 *  house bank goes bankrupt.
 *
 *    */
def getNewHouseBank = {time({
  val newHouseBank = sim.bankList.filter(_.active)( sim.random.nextInt(sim.bankList.filter(_.active).size) )
  if(sim.test) require(newHouseBank != houseBank && newHouseBank.active == true)
    _houseBank += newHouseBank
    houseBank.retailClients += this
 }, "hh_getNewHouseBank", sim)
}



/**
 *
 *  Since HH agents are customers of money market fund agents, they have to search for another MMF agent if their
 *  house MMF goes bankrupt.
 *
 *    */
def getNewHouseShadowBank = {time({
  if(sim.MMMFList.filter(_.active).size > 0){
    val newHouseShadowBank = sim.MMMFList.filter(_.active)( sim.random.nextInt(sim.MMMFList.filter(_.active).size) )
    if(sim.test) require(newHouseShadowBank != houseShadowBank && newHouseShadowBank.active)
    _houseShadowBank += newHouseShadowBank
  }
 }, "hh_getNewHouseShadowBank", sim)
}





/**
```

```
  *
  *  In order to endow newly entered bank agents with some initial demand for their financial services, every customer has a small probability
  *  to switch its house bank every once in while.
  *
  *    */
 def switchHouseBank (t:Int) = {time({
   val listOfNewAndSmallBanks = sim.bankList.filter(bank => bank.active == true && bank.retailClients.size < (sim.numberOfHH / sim.numberOfBanks) * 0.25)
   val probability2Switch     = if(listOfNewAndSmallBanks.nonEmpty) 1.0/sim.numberOfBanks else 0.1
   if(listOfNewAndSmallBanks.nonEmpty && sim.random.nextDouble <= probability2Switch){
     val newHouseBank         = sim.random.shuffle(listOfNewAndSmallBanks).head
     val rDeposits2Transfer   = math.max(_bankDeposits.last, 0)
     if(rDeposits2Transfer > 0){
       if(houseBank.cbReserves.last < rDeposits2Transfer) houseBank.getIntraDayLiquidity(rDeposits2Transfer, t)
       withdraw(  houseBank.retailDeposits, rDeposits2Transfer, t, sim)
       withdraw(  houseBank.cbReserves,     rDeposits2Transfer, t, sim)
       deposit(newHouseBank.retailDeposits, rDeposits2Transfer, t, sim)
       deposit(newHouseBank.cbReserves,     rDeposits2Transfer, t, sim)
     } else if(rDeposits2Transfer < 0) sys.error(s"_bankDeposits of $this are negative: ${_bankDeposits.last}")
     houseBank.retailClients -= this                                 // old HB side
     _houseBank              += newHouseBank                         // client side
     houseBank.retailClients += this                                 // new HB side
   }
  }, "hh_switchHouseBank", sim)
 }









 /**
  *
  *   Every unemployed HH starts searching for a job until it gehts hired by a firm with a matching vacancy.
  *   1. They start looking if there are any vacancies at all
  *   2. Then they choose a fraction of the available vacancies
  *   3. Then they choose the one with the most attractive wage
  *   4. then they get hired if the labor skill demand of the firm match the labor skill of the HH
  *
  *    */
 def searchJob (t:Int) {time({
   if(laborMarket.vacancies.maxBy(_._2.laborDemand)._2.laborDemand > 0){                                                          // are there any vacancies at all?
     val consideredJobs = random.shuffle(laborMarket.vacancies).take( (numberOfFirms * vacancyAvailabilityParameter).toInt ).toMap       // choose random fraction of available
vacancies
     if(consideredJobs.maxBy(_._2.laborDemand)._2.laborDemand > 0){
       val firmWithTopWage = sim.random.shuffle(consideredJobs).head._1
       val topWage = consideredJobs(firmWithTopWage).wageFactor
       if(laborMarket.vacancies(firmWithTopWage).laborDemand >= _laborSkillFactor.last){
         if(laborMarket.vacancies(firmWithTopWage).wageFactor >= _reservationWage.last){
           currentEmployer = firmWithTopWage
           firmWithTopWage match {
             case firmWithTopWage:Firm =>
```

```
            if(sim.test) require(firmWithTopWage.offeredWages.last == topWage, s"topWage is not correct")
            firmWithTopWage.queuedEmployees += this -> topWage
          case firmWithTopWage:Bank =>  sys.error(this + " cannot take a Job at a Bank yet.")
        }
        periodsOfUnemployment.update(periodsOfUnemployment.length-1, periodsOfUnemployment.last + (4 - (t-1) % 4))
        laborMarket.vacancies += firmWithTopWage -> laborMarket.Job(math.max(0.0, rounded( laborMarket.vacancies(firmWithTopWage).laborDemand - _laborSkillFactor.last )), laborMarket.vacancies(firmWithTopWage).wageFactor)
        if(sim.pln) println(this + " takes a Job at " + firmWithTopWage)
        } else writeData4Unemployment
      } else writeData4Unemployment
    } else writeData4Unemployment
  } else writeData4Unemployment

  def writeData4Unemployment { periodsOfUnemployment.update(periodsOfUnemployment.length-1, periodsOfUnemployment.last + 1) }
}, "hh_searchJob", sim)}

/**
 *
 *  [deprecated]
 *
 *
 *    */
def changeEmployer (t:Int, oldFirm:Firm, prob:Double = 1.0) {time({
  val shareOfOldFirm = oldFirm.currentProductionShare
  val equalShare     = (100.0 / sim.firmList.filter(_.active).size) / 100
  if(shareOfOldFirm > equalShare && random.nextDouble <= prob && laborMarket.vacancies.maxBy(_._2.laborDemand)._2.laborDemand >= _laborSkillFactor.last){
```

```scala
    val consideredOffers =
      random.shuffle(laborMarket.vacancies).toMap.filter {
        case(firm, job) =>
          val share = firm match {
            case f:Firm => f.currentProductionShare
            case _      => 0.0
      };
      share < equalShare }.filter {
        case(firm, job) =>
          job.laborDemand >= _laborSkillFactor.last
      }
    if(consideredOffers.nonEmpty){
      val newFirm:Firm = consideredOffers.head._1 match {
        case f:Firm => f
        case _ => sys.error("newFirm must be a firm...")
      }
      val newWage              = laborMarket.vacancies(newFirm).wageFactor
      laborMarket.vacancies   += newFirm -> laborMarket.Job(math.max(0.0, rounded( laborMarket.vacancies(newFirm).laborDemand - _laborSkillFactor.last )),
laborMarket.vacancies(newFirm).wageFactor)
      currentEmployer          = newFirm
      newFirm.queuedEmployees += this -> newWage

    // clear olfFirm relationships
      oldFirm.employees -= this
      oldFirm.wageBill  -= this
    }
  }
 }, "hh_changeEmployer", sim)
}

private val _interestOnDeposits = ArrayBuffer[Double](0.0)
private val _dividendsReceived  = ArrayBuffer[Double](0.0) // yearly
private val _plannedConsumption = ArrayBuffer[Double](2 * rounded( math.pow(_laborSkillFactor.last, 1-0.2) ) ) // 0.85
def interestOnDeposits = _interestOnDeposits
def dividendsReceived  = _dividendsReceived
def plannedConsumption = _plannedConsumption


/**
 *
 *  HH plan their consumption of the upcoming quarter depending on their individual expected income.
 *
 *
 *    */
def planConsumption (t:Int, lambda:Double = 0.9, c:Double = 0.95) = {time({
  val wageOfPreviousMonth = average( sim.firmList.map{ _.offeredWages.init.last } )
```

```scala
    val autonomousConsumption  = 0.18 * wageOfPreviousMonth
    val wage =
      currentEmployer match {
        case employer:Firm =>
          if(employer.employees.contains(this)) {
            _laborSkillFactor.last * employer.wageBill(this) - sim.government.incomeTax(_laborSkillFactor.last * employer.wageBill(this))
          } else if(employer.queuedEmployees.contains(this)) {
            _laborSkillFactor.last * employer.queuedEmployees(this) - sim.government.incomeTax(_laborSkillFactor.last * employer.queuedEmployees(this))
          } else sys.error(s"Error in plannedConsumption of $this")
        case employer:ARGE => government.unemploymentBenefit.last / 4.0
      }
    val interest  = _interestOnDeposits.last / 48
    val dividends = _dividendsReceived.last  / 48
    val netIncome = wage + interest + dividends
    _plannedConsumption += lambda * _plannedConsumption.last + (1 - lambda) * ( autonomousConsumption + lambda * netIncome )
   }, "hh_planConsumption", sim)
  }





  /**
   *
   *  HH consume according to their plan except for the case that their liquidity is insufficient to consume according to their plan. In such a case, they restrict their consumption appropriately.
   *  1. HH search for offers of the goods bundle.
   *  2. If the market provides offers, they chose the one with the lowest price and
   *  3. consume the planed quantity if they are liquid enough.
   *
   *    */
  def consume (t:Int, causeFirm:String = if(tradBanks) "consumption1" else "consumption0", causeTax:String = if(tradBanks) "VAT1" else "VAT0") {time({
    var consideredOffers         = random.shuffle(goodsMarket.currentOffers).take( (numberOfFirms * goodAvailabilityParameter).toInt ).toMap.filter(_._2.quantity > 0)
    var currentConsumptionDemand = _plannedConsumption.last

    while(currentConsumptionDemand > 0 && _bankDeposits.last > 0.0 && consideredOffers.nonEmpty) {
      val (bestFirm:Firm, bestOffer) = sim.random.shuffle(consideredOffers).head
      val p            = bestOffer.price
      val q            = bestOffer.quantity
      val quantity2Buy = Seq(q, currentConsumptionDemand / (p * (1 + government.VAT.last)), _bankDeposits.last / (p * (1 + government.VAT.last))).min
      if(sim.pln) println(this + " has " + cash.last + " and wants to spend " + amountToSpend + " to spend and the offered price is " + lowestPrice + " (incl. VAT), so he can consume " + affordableQuantity + " of " + consideredOffers(corpWithLowestPrice).quantity + " / " + corpWithLowestPrice.amountOfInventory.last + " offered by " + corpWithLowestPrice)
      deposit( bestFirm.sales,             quantity2Buy, t, sim)
      withdraw(bestFirm.amountOfInventory, quantity2Buy, t, sim)
      _amount2Spend.update(_amount2Spend.size-1, _amount2Spend.last + quantity2Buy * p)
      transferMoney(this, bestFirm,   quantity2Buy * p,                        causeFirm, sim, t)
      transferMoney(this, government, quantity2Buy * p * government.VAT.last, causeTax,  sim, t)
      goodsMarket.currentOffers += bestFirm -> goodsMarket.Offer(bestFirm, rounded(q - quantity2Buy), p)
      consideredOffers          -= bestFirm
      currentConsumptionDemand  -= quantity2Buy * p * (1 + government.VAT.last)
    }
   }, "hh_consume", sim)
```

```
}

/**
 *
 *  [deprecated]
 *
 *    */
def proofCreditworthinessOfFirm (firm:Firm) = if(random.nextDouble < probOfGrantingLoan2Client(firm)) true else false

/**
 *
 *  HH have to pay a fee to their house bank, since they use the payment system through their bank account.
 *
 *    */
def payBankAccountFee (t:Int) = if(_bankDeposits.last >= 20) transferMoney(this, houseBank, 20, "payBankAccountFee", sim, t)

/**
 *
 *  At the end of each fiscal year, the HH agent makes an annual report to update its balance sheets statements.
 *
 *    */
def makeAnnualReport (t:Int) {time({
  _shareOfCorporations.foreach{
    case(firm, share) =>
      firm match {
        case firm:Firm => _equityStake(_equityStake.size-1) += rounded( share * firm.equity.last )
        case firm:Bank =>
      }
  }
  updatePVofSoBs(t)
  // AR
  _totalAssets += rounded( Seq(_equityStake.last, _loans.last, _bankDeposits.last, bonds.last, _cash.last).sum )
  if(sim.pln) println("Total assets of " + this + ": " + inventory.last + " + " + bankDeposits.last + " + " + cash.last + " = " + totalAssets.last)
    _equity      += rounded( _totalAssets.last )
  if(sim.pln) println("Equity of " + this + ": " + totalAssets.last + " - (" + debtCapital.last + " + " + interestOnDebt.last + ") = " + equity.last)

    if(equity.last < 0){
      if(sim.pln) println(s"""
      A             P
      ----------------------------
      inve ${_equityStake.last} |
      bd   ${_bankDeposits.last} |
      bonds ${bonds.last} |
      cash ${_cash.last} | eq.  ${_equity.last}
```

```
      -----------------------------
      TA   ${_totalAssets.last} |
      """)
    }
  if(sim.test) require(_equity.last >= -1, "Equity of " + this + " is < 0. That must be an sys.error.")
 }, "hh_AR", sim)
 }




 /*  Save Data */
  val hhEndOfTickData = Map()

  val hhEndOfSimulationData = Map(
                                  "reservationWage"          -> _reservationWage,
                                  "periodsOfUnemployment"    -> _periodsOfUnemployment,
                                  "unemployed"               -> _unemployed,
                                  "employer"                 -> _employers,
                                  "foundedCorporations"      -> _foundedCorporations,
                                  "laborSkillFactor"         -> _laborSkillFactor,
                                  "shareOfCorporations"      -> _shareOfCorporations,
                                  "willingness2Consume"      -> _willingness2Consume,
                                  "amount2Spend"             -> _amount2Spend,
                                  "totalAssets"              -> _totalAssets,
                                  "plannedConsumption"       -> _plannedConsumption,
                                  "equity"                   -> _equity,
                                  "riskAversionParameter"    -> _riskAversionParameter,
                                  "speculativeFunds"         -> _speculativeFunds
                                                                                                    )



} // end of HH class
```

A.5.2 Firm Class

Firm.scala

```scala
/**
 * @author Krugman
 * @constructor
 * @param name
 * @param numberOfHH
 *
 */


package monEcon.realSector

import collection.mutable._
import math._
import util.Random
import util.control._

import monEcon.financialSector.Bank
import monEcon.financialSector.BrokerDealer
import monEcon.Corporation
import monEcon.publicSector._
import monEcon.ARGE
import monEcon.Markets._
import monEcon.Simulation

import org.apache.commons.math3._
import org.apache.commons.math3.stat.regression.SimpleRegression






// ---------- Class for the Firm-Objects ----------
case class Firm (name                            :String,          //
                 numberOfHH                      :Int,             //
                 numberOfFirms                   :Int,             //
                 numberOfBanks                   :Int,             //
                 tradBanks                       :Boolean,         //
                 arge                            :ARGE,            //
                 random                          :Random,          //
                 initialHouseBank                :Bank,            //
                 initialHouseShadowBank          :BrokerDealer,
                 goodsMarket                     :GoodsMarket,     //
                 laborMarket                     :LaborMarket,     //
                 interbankMarket                 :InterbankMarket, //
                 government                      :Government,      //
                 initialInventory                :Double,          //
                 initialprice                    :Double,          //
                 initialWageFactor               :Double,          //
                 initialProductionTarget         :Double,          //
                 firmProductivityFactor          :Double,          //
                 privateFundAvailabilityParameter:Double,          //
                 retainedEarningsParameter       :Double,          //
```

```
               initialCapital                  :Double,           //
               firmDebt2EquityTarget           :Double            //
               sim                             :Simulation        //
                                                                       ) extends Corporation {

  override def toString = s"Firm($name)"




/*  ----------------------------------------  firm balance sheet positions  ----------------------------------------  */
// Asset Side
private val _inventory      = ArrayBuffer(0.0)
private val _bankDeposits   = ArrayBuffer(0.0)
private val _cash           = if(tradBanks) ArrayBuffer(0.0) else ArrayBuffer(initialCapital)
//-----------------------------------------------
private val _totalAssets  = ArrayBuffer[Double]()

// Liability Side
private val _debtCapital    = ArrayBuffer(0.0)
private val _interestOnDebt = ArrayBuffer(0.0)
private val _equity         = ArrayBuffer[Double]()



/**
 *
 *  This is just to save balance sheet data.
 *
 *    */
val firmBSP = Map("inventory"        ->  inventory,
                  "bankDeposits"     ->  bankDeposits,
                  "cash"             ->  cash,
                  "totalAssets"      ->  totalAssets,
                  "debtCapital"      ->  debtCapital,
                  "interestOnDebt"   ->  interestOnDebt
                  "equity"           ->  equity
                                                         )





// ----- Other Data -----
private val _houseBank                  = ArrayBuffer[Bank](initialHouseBank)
private val _houseShadowBank            = ArrayBuffer[BrokerDealer](initialHouseShadowBank)
private var _active                     = true
private var _periodOfReactivation       = 0
private var _age                        = 0
private val _debtToEquityTarget         = ArrayBuffer[Double]()
private val _insolvencies               = ArrayBuffer(0)
private val _costOfGoodsSold            = new ArrayBuffer[Double]
```

```
private val _revenues                  = new ArrayBuffer[Double]
private val _doubtfulCredit            = new ArrayBuffer[Boolean]
private val _privateLender             = Map[HH,(Double,Double,Int,Double)]()
private val _productionTarget          = ArrayBuffer(initialProductionTarget)
private val _producedGoods             = new ArrayBuffer[Double]
private val _amountOfInventory         = ArrayBuffer(initialInventory)
private val _offeredWages              = ArrayBuffer(initialWageFactor)
private val _queuedEmployees           = Map[HH, Double]()
private val _employees                 = Set[HH]()
private val _wageBill                  = Map[HH, Double]()
private val _numberOfEmployees         = ArrayBuffer[Int]()
private val _needForExternalFinancing  = ArrayBuffer[Double](10.000)
private val _interestOfferedOnBankLoan = ArrayBuffer[Double]()
private val _price                     = ArrayBuffer(initialprice)
private val _sales                     = ArrayBuffer(0.0)
private val _valuedInventory           = new LinkedHashMap[Int, (Double,Double)]
private val _ptDecision                = ArrayBuffer[Double]()
private val _pastInv                   = ArrayBuffer[Double]()
private val _pastProd                  = ArrayBuffer[Double]()
private val _vacancies                 = ArrayBuffer[(Double,Double)]()
private val _currentProdCap            = ArrayBuffer[Double]()
private val _creditRationed            = ArrayBuffer[Int](0, 0, 0, 0, 0, 0, 0)






// getter
def houseBank                = _houseBank.last                            //
def houseShadowBank          = _houseShadowBank.last                      //
def active                   = _active                                    //
def periodOfReactivation     = _periodOfReactivation                      //
def insolvencies             = _insolvencies                              //
def costOfGoodsSold          = _costOfGoodsSold                           //
def revenues                 = _revenues                                  //
def inventory                = _inventory                                 //
def bankDeposits             = _bankDeposits                              //
def cash                     = _cash                                      //
def totalAssets              = _totalAssets                               //
def debtCapital              = _debtCapital                               //
def interestOnDebt           = _interestOnDebt                            //
def equity                   = _equity                                    //
def doubtfulCredit           = _doubtfulCredit                            //
def amountOfInventory        = _amountOfInventory                         //
def needForExternalFinancing = _needForExternalFinancing                  //
def productionTarget         = _productionTarget                          //
def producedGoods            = _producedGoods                             //
def offeredWages             = _offeredWages                              //
def queuedEmployees          = _queuedEmployees                           //
def employees                = _employees                                 //
def wageBill                 = _wageBill                                  //
def numberOfEmployees        = _numberOfEmployees                         //
```

```scala
def vacancies                 = _vacancies                                         //
def price                     = _price                                             //
def sales                     = _sales                                             //
def valuedInventory           = _valuedInventory                                   //
def privateLender             = _privateLender                                     //
def age                       = _age                                               //
def debtToEquityTarget        = _debtToEquityTarget                                //
def interestOfferedOnBankLoan = _interestOfferedOnBankLoan
def ptDecision                = _ptDecision
def pastInv                   = _pastInv
def pastProd                  = _pastProd
def currentProdCap            = _currentProdCap
def creditRationed            = _creditRationed

// setter
def inventory_+=                (value:Int)     :Unit = _inventory                += value  //
def bankDeposits_+=             (value:Double)  :Unit = _bankDeposits             += value  //
def debtCapital_+=              (value:Double)  :Unit = _debtCapital              += value  //
def doubtfulCredit_+=           (value:Boolean) :Unit = _doubtfulCredit           += value  //
def needForExternalFinancing_+= (value:Double)  :Unit = _needForExternalFinancing += value  //
def productionTarget_+=         (value:Int)     :Unit = _productionTarget         += value  //
def offeredWages_+=             (value:Double)  :Unit = _offeredWages             += value  //
def employees_+=                (value:HH)      :Unit = _employees                += value  //

/*  --------------------  Methods of Firms  --------------------  */
/**
 *
 *  This method increases the counter "age" every tick. The counter is reset after a default of the agent. The counter shows the time the agent was able to operate in die markets.
 *
 *    */
def updateFirmAge   = _age += 1

/**
 *
 *  Since firm agents are customers of traditional bank agents, they have to search for another bank agent if their
 *  house bank is bankrupt.
 *
 *    */
def getNewHouseBank = {
  val newHouseBank = sim.bankList.filter(_.active)( sim.random.nextInt(sim.bankList.filter(_.active).size) )
  if(sim.test) require(newHouseBank != houseBank && newHouseBank.active)
  _houseBank                += newHouseBank
  houseBank.businessClients += this
```

```
}



/**
 *
 *  Since firm agents are customers of BD agents, they have to search for another BD agent if their
 *  house BD goes bankrupt.
 *
 *    */
def getNewHouseShadowBank = {
  if(sim.BrokerDealerList.filter(_.active).size > 0){
    val newHouseShadowBank = sim.BrokerDealerList.filter(_.active)( sim.random.nextInt(sim.BrokerDealerList.filter(_.active).size) )
    if(sim.test) require(newHouseShadowBank != houseShadowBank && newHouseShadowBank.active)
    _houseShadowBank        += newHouseShadowBank
    houseShadowBank.clients += this
  }
}




/**
 *
 *  In order to endow newly entered bank agents with some initial demand for their financial services, every customer has a small probability
 *  to switch its house bank every once in while.
 *
 *    */
def switchHouseBank (t:Int) = {
  val free2Switch = if(!houseBank.listOfDebtors.contains(this) || houseBank.listOfDebtors(this).isEmpty) true else false
  val listOfNewAndSmallBanks = sim.bankList.filter(bank => bank.active == true && bank.businessClients.size < (sim.numberOfFirms / sim.numberOfBanks) * 0.2)
  val probability2Switch = if(listOfNewAndSmallBanks.nonEmpty) 1.0/sim.numberOfBanks else 0.1
  if(listOfNewAndSmallBanks.nonEmpty && sim.random.nextDouble <= probability2Switch && free2Switch){
    val newHouseBank = sim.random.shuffle(listOfNewAndSmallBanks).head
    val rDeposits2Transfer = _bankDeposits.last
    if(houseBank.cbReserves.last < rDeposits2Transfer) houseBank.getIntraDayLiquidity(rDeposits2Transfer, t)
    withdraw(houseBank.retailDeposits,   rDeposits2Transfer, t, sim)
    withdraw(houseBank.cbReserves,       rDeposits2Transfer, t, sim)
    deposit(newHouseBank.retailDeposits, rDeposits2Transfer, t, sim)
    deposit(newHouseBank.cbReserves,     rDeposits2Transfer, t, sim)
    if(houseBank.listOfDebtors.contains(this)) houseBank.listOfDebtors -= this
    houseBank.businessClients -= this
    _houseBank                += newHouseBank
    houseBank.businessClients += this
  }
}




private val utilizationTarget = 0.75

/**
```

```
 *
 *  Firms plan their production according to their past sales plus a mark up to cope with demand fluctuations.
 *
 *    */
def determineProductionTarget (t:Int, pastInventory:Double = sumOfPastPeriods(_amountOfInventory, sim), pastProduction:Double = sumOfPastPeriods(_producedGoods, sim)) {
  time(
    val pastSales = sumOfPastPeriods(_sales, sim)
    if(pastSales == 0.0) _productionTarget += math.max(initialProductionTarget, _productionTarget.last) else _productionTarget += pastSales / utilizationTarget
    _pastInv    += pastInventory
    _pastProd   += pastProduction
    _ptDecision += pastInventory - pastProduction
  , "firm_determineProductionTarget", sim)
  if(sim.pln) println(s"$this has a new productionTarget of ${productionTarget.last}")
}





def currentProductionShare          = potentialProductionCapacity / productionFunction(sim.hhList.map(_.laborSkillFactor.last).sum)
def determineCurrentMarketShareCFSI = if(_active) roundTo4Digits( _totalAssets.last / sim.firmList.filter(_.active).map(_.totalAssets.last).sum ) else 0.0





/**
 *
 *  Firms set their wage offered per unit of labor skill according to their ability to hire a sufficient amount of workers in the past.
 *
 *    */
def determineOfferedWageFactor (t:Int, pastTarget:Double = sumOfPastPeriods(productionTarget, sim), pastProduction:Double = sumOfPastPeriods(producedGoods, sim)) {
  time({
    if(t < 250){
      _offeredWages += math.max( initialWageFactor, offeredWages.last * math.exp( 0.012 / (48 / sim.updateFrequency) ) )
    } else {
      _offeredWages += math.max( _offeredWages.reverse.find { _ > 0.0 }.head, offeredWages.last * (math.exp(0.01 / (48 / sim.updateFrequency)) + sim.expPi.last + 0.005 *
weightedEmploymentGap) )
    }
  }, "firm_determineOfferedWage", sim)
}




private val _utilizationGap         = ArrayBuffer[Double]()
private val _employmentGap          = ArrayBuffer[Double]()
private val _utilizationGapWeighted = ArrayBuffer[Double]()
private val _employmentGapWeighted  = ArrayBuffer[Double]()
def utilizationGap                  = _utilizationGap
def employmentGap                   = _employmentGap
def utilizationGapWeighted          = _utilizationGapWeighted
def employmentGapWeighted           = _employmentGapWeighted
```

```scala
/**
 *
 *  Determines whether there is a current excess production or not.
 *
 *    */
def determineUtilGapOfTick = {
  val utilization = potentialProductionCapacity / _productionTarget.last
  if(utilization.isNaN || utilization.isInfinity) _utilizationGap += 0.0 else _utilizationGap += utilization
}

/**
 *
 *  Determines whether there is a current excess employment or not.
 *
 *    */
def determineEmployGapOfTick = {
  val currentLaborSkill = rounded( _employees.map(_.laborSkillFactor.last).sum + _queuedEmployees.map{ case(hh, wage) => hh.laborSkillFactor.last}.sum )
  val employment        = currentLaborSkill / production2skill(_productionTarget.last)
  if(employment.isNaN || employment.isInfinity) _employmentGap += 0.0 else _employmentGap += employment
}

/**
 *
 *  Determines the weighted utilization gap. Newer gaps have a higher weight compared to gaps that lie far in the past.
 *
 *    */
def weightedUtilizationGap = {
  val T                     = sim.updateFrequency
  val weights               = collection.immutable.Vector.tabulate(T)(x => (T + 1 - (x+1)) / (0.5 * T * (T + 1)) )
  val utilGaps              = _utilizationGap.takeRight(T+1).reverse
  val sumOfWeightedUtilGaps = collection.immutable.Vector.tabulate(T)(n => utilGaps(n) * weights(n) ).sum
  _utilizationGapWeighted  += sumOfWeightedUtilGaps - utilizationTarget
  _utilizationGapWeighted.last
}

/**
 *
 *  Determines the weighted employment gap. Newer gaps have a higher weight compared to gaps that lie far in the past.
 *
 *    */
def weightedEmploymentGap = {
```

```scala
    val T                      = sim.updateFrequency
    val weights                = collection.immutable.Vector.tabulate(T)(x => (T + 1 - (x+1)) / (0.5 * T * (T + 1)) )
    val empGaps                = _employmentGap.takeRight(T+1).reverse
    val sumOfWeightedEmpGaps  = collection.immutable.Vector.tabulate(T)(n => empGaps(n) * weights(n) ).sum
    _employmentGapWeighted   += 1 - sumOfWeightedEmpGaps
    _employmentGapWeighted.last
  }





  /**
   *
   *  Determines the expected weekly labor costs in order to determine how much external finance or debt is needed from traditional/shadow banks.
   *
   *    */
  def expectedLaborCostsWeekly = {
    if(sim.test) require(
        math.pow(firmProductivityFactor, currentLaborSkill + math.max(0, production2skill(currentProductionDemand))) + 1 >= productionTarget.last,
        s"expectedLaborCosts are not correct: ${math.pow(firmProductivityFactor, currentLaborSkill + math.max(0, production2skill(currentProductionDemand))) + 1} is not >= ${productionTarget.last}"
       )
    1.1 * potentialLaborSkillCostsWeekly + ( production2skill(currentProductionDemand) * _offeredWages.last )
  }





  /**
   *
   *   [deprecated]
   *
   *    */
//  def debt2EquityTarget (t:Int) = {
//    if(t > 10000){
//      _age match {
//        case age:Int if(age <  144) => 4.00
//        case age:Int if(age <  336) => 2.00
//        case age:Int if(age <  576) => 1.00
//        case age:Int if(age <  960) => 0.50
//        case age:Int if(age < 2400) => 0.33
//        case _                      => 0.25
//      }
//    } else 0.0
//  }




  /**
   *
```

```
  *  Determines whether the firm has to request loans in order to meet its production target. If the firm has enough internal liquidity, it does not request loans.
  *
  *    */
 def determineExternalFinancing (t:Int, moneyAccount:ArrayBuffer[Double] = if(tradBanks) _bankDeposits else _cash) {
   time({
//      if(sim.test) assert(debt2EquityRatio >= 0, "D/E ratio cannot be negative: " + debt2EquityRatio)
//      debt2EquityRatio match {
//        case ratio:Double if ratio < debt2EquityTarget(t) =>  val amount2Borrow = max(0, debt2EquityTarget(t) * _equity.last - (_debtCapital.last + _interestOnDebt.last) )

//                                                             needForExternalFinancing += rounded( max(amount2Borrow, sim.updateFrequency * expectedLaborCostsWeekly - moneyAccount.last) )
//        case _                                           =>  needForExternalFinancing += rounded( max(            0, sim.updateFrequency * expectedLaborCostsWeekly - moneyAccount.last) )
//      }
     needForExternalFinancing        += rounded( max(0, sim.updateFrequency * expectedLaborCostsWeekly - moneyAccount.last) )
     _debtToEquityTarget             += debt2EquityTarget(t)
     _currentNeedForExternalFinancing = needForExternalFinancing.last
     if(sim.test) assert(needForExternalFinancing.last >= 0, "needForExternalFinancing can't be < 0!")
   }, "firm_determineExternalFinancing", sim)
 }




 /**
  *
  *    */
 private var _currentNeedForExternalFinancing:Double = 0.0
 def currentNeedForExternalFinancing                 = _currentNeedForExternalFinancing

 private val debtFinancing = ArrayBuffer[String]()





 /**
  *
  *    Firms request loans from traditional/shadow banks.
  *
  *    */
 def aquireFunding (t:Int, minMoney:Double = math.max(1000, 4 * goodsMarket.weightedAvgPriceOfYear.last)) {
   time({
     deposit(sim.neededLiquidityFirms, _currentNeedForExternalFinancing, t, sim)

     def try2getFundsFromSB {
       if(sim.BrokerDealerList.filter(_.active).size > 0){
         val listOfBrokerDealerWithIdleFunds = Map[BrokerDealer,(Double, Double)]()
         sim.BrokerDealerList.foreach{
           BD =>
             val (decisionOnLoanRequest, acceptedAmount, interestCharged) = BD.decideAboutLoanRequest(this, _currentNeedForExternalFinancing, t)
             if(decisionOnLoanRequest && acceptedAmount > 0) listOfBrokerDealerWithIdleFunds += BD -> {(acceptedAmount, interestCharged)}
         }
```

```scala
          if(listOfBrokerDealerWithIdleFunds.nonEmpty){
            val BDsWithSufficientFunds = listOfBrokerDealerWithIdleFunds.filter{ case (brokerDealer, (offeredAmount, intRate)) => offeredAmount >=
_currentNeedForExternalFinancing}
            if(BDsWithSufficientFunds.nonEmpty){
              val chosenBD = BDsWithSufficientFunds.minBy{ case (brokerDealer, (offeredAmount, intRate)) => intRate}._1
              val (acceptedLoan, offeredInterest) = BDsWithSufficientFunds(chosenBD)
              if(chosenBD.proofRegulatoryRequirements(t)){
                if(loanIsProfitable(offeredInterest)) {
                  chosenBD.grantCredit2Firm(this, acceptedLoan, offeredInterest, t)
                  _currentNeedForExternalFinancing -= acceptedLoan
                } else _creditRationed(6) += 1
              } else _creditRationed(5) += 1
            } else {
              _creditRationed(4) += 1
              val chosenBD = listOfBrokerDealerWithIdleFunds.maxBy{ case (brokerDealer, (offeredAmount, intRate)) => offeredAmount}._1
              debtFinancing += s"$t --> $this requests ${_currentNeedForExternalFinancing} but Sb sector only offers: ${listOfBrokerDealerWithIdleFunds} and so he chooses
$chosenBD \n"
              val (acceptedLoan, offeredInterest) = listOfBrokerDealerWithIdleFunds(chosenBD)
              if(chosenBD.proofRegulatoryRequirements(t)){
                if(loanIsProfitable(offeredInterest)) {
                  chosenBD.grantCredit2Firm(this, acceptedLoan, offeredInterest, t)
                  _currentNeedForExternalFinancing -= acceptedLoan
                } else _creditRationed(6) += 1
              } else _creditRationed(5) += 1
            }
          } else _creditRationed(3) += 1
        }
      }



      tradBanks match {

        case true  =>
          if(_currentNeedForExternalFinancing > 0){
            _creditRationed(0) += 1
            if(houseBank.proofRegulatoryRequirements(t)){
              // request a loan
              val (decisionOnLoanRequest, acceptedAmount, interestCharged) = houseBank.proofCreditworthiness(this, _currentNeedForExternalFinancing, t)
              if(sim.test) require(acceptedAmount >= 0, s"The requested amount of $this is negative: $acceptedAmount.")
              decisionOnLoanRequest match {
                case "unrestricted"  =>
                  _interestOfferedOnBankLoan += interestCharged
                  if(loanIsProfitable(interestCharged)) {
                    houseBank.grantCredit2Firm(this, acceptedAmount, interestCharged, t)
                    _currentNeedForExternalFinancing -= acceptedAmount
                  } else {
                    _creditRationed(2) += 1
                    try2getFundsFromSB
                  }

                case "restricted"    =>
                case "denied"        => if(sim.pln) println(s"$houseBank denies the request of $this for ${_currentNeedForExternalFinancing} because it is not creditworthy
```

```
enough (D/E of ${debt2EquityRatio})")
            }
          } else {
            _creditRationed(1) += 1
            try2getFundsFromSB
          }
        }



      case false =>
        var need = _currentNeedForExternalFinancing
        random.shuffle(sim.hhList).take((numberOfHH * privateFundAvailabilityParameter).toInt).filter(_.cash.last - minMoney > 0).foreach{
          hh =>
            val loan = math.min(need, hh.cash.last - minMoney)
            if(loan > 0) {
              if(hh.proofCreditworthinessOfFirm(this)){
                hh.privateBorrower += this -> (rounded(loan), rounded(loan * hh.interestOnLoans.last), t+24, hh.interestOnLoans.last)
                privateLender      += hh   -> (rounded(loan), rounded(loan * hh.interestOnLoans.last), t+24, hh.interestOnLoans.last)
                transferMoney(hh, this, loan, "privateLending", sim, t)
                need                  -= loan
                _currentNeedForExternalFinancing -= loan
                if(sim.pln) println(this + " needs " + need + " / " + currentNeedForExternalFinancing + " and borrows " + loan + " from " + hh)
              } else if(sim.pln) println(hh + " does not want to lend money to " + this + " because of its high D/E-ratio of `" + debt2EquityRatio + "`  ")
            }//
        }//
      }// match
    }, "firm_aquireFunding", sim)
  }






  /**
   *
   *  After requesting a loan from a traditional/shadow bank, the firm agent decides on the offered conditions, i.e. to take the loan or not.
   *
   *    */
  def loanIsProfitable (offeredInterestByBank:Double) = {
    if(random.nextDouble < math.max(0, 10 * (0.18 - 0.75 * offeredInterestByBank)) ) true else false
  }





  val _profitabilityOfOB = ArrayBuffer[(Double, Double, Double, Int)]()
  /**
   *
   *  This method checks whether the operating business of the firm agent is profitable over time or not.
   *    exp(Rev) - costs
```

```
   *
   *    */
  def profitabilityOfOperatingBusiness = {
    val expRev = _price.last * _producedGoods.last
    val costs  = actualLaborSkillCostsWeekly
    val LS     = rounded( _employees.map(_.laborSkillFactor.last).sum )
    val NOE    = _employees.size
    _profitabilityOfOB += {(expRev, costs, LS, NOE)}
  }





  /**
   *
   *  Calculates the current debt-to-equity ratio of the firm.
   *
   *    */
  def debt2EquityRatio = {
    if(_equity.nonEmpty){
      if(_equity.last == 0){
        if((_debtCapital.last + _interestOnDebt.last) == 0) 0.0 else 100.0
      } else if((_debtCapital.last + _interestOnDebt.last) / _equity.last >= 0) {
        (_debtCapital.last + _interestOnDebt.last) / _equity.last
      } else 100.0
    } else 0.0
  }






  /*  ------------------------------------------------------------ Labor Supply of Firm -----------------------------------------------------  */
  // production function of the Cobb-Douglas type
  def productionFunction (labor:Double, capital:Double = 1, A:Double = sim.At.last, alpha:Double = 0.2) = rounded( math.pow(capital, alpha) * math.pow(A * labor, 1-alpha) )
  def production2skill (production:Double, alpha:Double = 0.2)       = if(production >= 0) rounded( math.pow( production / sim.At.last, 1 / (1-alpha) ) ) else 0.0
  def potentialProductionCapacity                                    = rounded( productionFunction(_employees.map(_.laborSkillFactor.last).sum + _queuedEmployees.map{ case(hh, wage) => hh.laborSkillFactor.last}.sum ) )
  def actualProductionCapacity                                       = rounded( productionFunction(_employees.map(_.laborSkillFactor.last).sum ) )
  def potentialLaborSkillCostsWeekly                                 = rounded( _employees.map(hh => hh.laborSkillFactor.last * _wageBill(hh)).sum + _queuedEmployees.map{ case(hh, wage) => hh.laborSkillFactor.last * wage}.sum )
  def actualLaborSkillCostsWeekly                                    = rounded( _employees.map(hh => hh.laborSkillFactor.last * _wageBill(hh)).sum )
  def potentialProductionCapacity                                    = rounded( _employees.map(hh => productivityOfEmployee(hh)).sum + _queuedEmployees.map{ case(hh, wage) => productivityOfEmployee(hh)}.sum )
  def actualProductionCapacity                                       = rounded( _employees.map(hh => productivityOfEmployee(hh)).sum )
  def currentProductionDemand                                        = rounded( _productionTarget.last - potentialProductionCapacity )


  /**
   *
   *  Determines how many units of labor skill the firm agent is able to finance based on his current offered wage.
```

```scala
   *
   *    */
  def affordableAdditionalLaborSkill (t:Int) = {
    if(tradBanks){
      val remainingWeeks = (sim.updateFrequency - t % sim.updateFrequency)
      val wage2Pay       = remainingWeeks * potentialLaborSkillCostsWeekly
      val pp2Pay         = if(houseBank.listOfDebtors.contains(this)) houseBank.listOfDebtors(this).map(_.principalPayments.filterKeys(_ >= t).filterKeys(_ < t + remainingWeeks).values.sum).sum else 0.0
      val ip2Pay         = if(houseBank.listOfDebtors.contains(this)) houseBank.listOfDebtors(this).map( _.interestPayments.filterKeys(_ >= t).filterKeys(_ < t + remainingWeeks).values.sum).sum else 0.0
      val RevOfProduc    = remainingWeeks * (_price.last * actualProductionCapacity)
      math.max(0, rounded( (_bankDeposits.last - wage2Pay - pp2Pay - ip2Pay + RevOfProduc) / _offeredWages.last ) )
    } else math.max(0, rounded( (_cash.last - potentialLaborSkillCostsWeekly) / _offeredWages.last ) ) // incl. external financing
  }


  /**
   *
   *  Firm agent offers new vacancies on the labor market.
   *
   *    */
  def announceCurrentJobs (t:Int, s:Double = 1) {
      time({
        val laborSkillDemand     = if(s != 0) math.min( affordableAdditionalLaborSkill(t), production2skill(currentProductionDemand) ) else 0.0
        _vacancies              += laborSkillDemand -> _offeredWages.last
        laborMarket.laborDemand += this             -> laborSkillDemand
        laborMarket.wageFactors += this             -> _offeredWages.last
        laborMarket.vacancies   += this             -> laborMarket.Job(laborSkillDemand, _offeredWages.last)
      }, "firm_announceCurrentJobs", sim)
    }


  /**
   *
   *  Firms hire workers (i.e. HH) at the beginning of each month.
   *
   *    */
  def employHH (t:Int) {
    time({
      _queuedEmployees.foreach{
        case(hh, wage) =>
          _employees += hh
          _wageBill  += hh -> wage
      }
      if(sim.test) require(_employees.size == _wageBill.size, s"Unequal amount of HH employed and payed: ${_employees} = ${_employees.size} / ${_wageBill} = ${_wageBill.size}")
      _queuedEmployees.clear
```

```scala
  }, "firm_employHH", sim)
}






/**
 *
 *  If the firm agent faces an overcapacity of laborskill, it fires a sufficient amount of employees.
 *
 *    */
def fireEmployees (t:Int, currentEmployees:Map[HH, Double] = Map()) {
  time({
    if(1.05 * _productionTarget.last - potentialProductionCapacity < 0){
      var overCapacity = 1.05 * _productionTarget.last - potentialProductionCapacity
      _employees.foreach(hh => currentEmployees += hh -> hh.laborSkillFactor.last)
      val stupidEmployees = currentEmployees.retain((hh, skill) => (-overCapacity - skill) >= 0)
      while(stupidEmployees.nonEmpty && -overCapacity >= stupidEmployees.valuesIterator.min){
        val employee2Fire = stupidEmployees.maxBy(_._2)._1
        if(sim.pln) println(employee2Fire + " will be fired since currentLaborDemand is " + overCapacity + " and its skillFactor is " + stupidEmployees(employee2Fire))
        fireHH(employee2Fire)
        overCapacity    += stupidEmployees(employee2Fire)
        stupidEmployees -= employee2Fire
        if(sim.pln) println(employee2Fire + " with skill " + employee2Fire.laborSkillFactor.last + " is fired because " + this + " has an overcapacity of " + -overCapacity)
      }
    }
  }, "firm_fireEmployees", sim)
}






/**
 *
 *    */
def updateWages (t:Int) { time({ _employees.foreach{hh => _wageBill += hh -> offeredWages.last }    }, "firm_updateWages", sim) }





/**
 *
 *    */
def fireHH (hh:HH) {
  _employees -= hh
  _wageBill  -= hh
  hh.currentEmployer = arge
}
```

```
/**
 *
 *  Each month, firms pay wages to their employed HH.
 *
 *    */
def payOutWage2HH (t:Int) {
  time({
    if(sim.pln) println(this + " has exp. labor costs of: " + expectedLaborCostsMonthly + ", current cash is " + cash.last + " which means I can afford an add. skill of " + affordableAdditionalLaborSkill + " but current labor skill costs of " + currentLaborSkillCosts + ", my demand is " + currentProductionDemand + " " + vacancies.last)
      tradBanks match {

        case true =>
          val b = new Breaks
          b.breakable{
            employees.foreach{
              hh =>
                val wage = 4 * _wageBill(hh) * hh.laborSkillFactor.last
                if(_bankDeposits.last >= wage){
                  if(sim.test) assert(_bankDeposits.last >= 4 * _wageBill(hh) * hh.laborSkillFactor.last, transferMoney(this, hh, _bankDeposits.last, "payWage1", sim, t))
                  transferMoney(this, hh, wage, "payWage1", sim, t)
                } else {
                  transferMoney(this, hh, math.min(wage, _bankDeposits.last), "payWage1", sim, t)
                  shutDownFirm(t, "illiquidyWage")
                  b.break
                }
            }// foreach
          }

        case false  =>
          employees.foreach{
            hh =>
              if(sim.pln) println(this + " has " + cash.last + " cash and pays a wage of " + offeredWages.last * hh.laborSkillFactor.last + " to " + hh)
              transferMoney(this, hh, min(_cash.last, 4 * _wageBill(hh) * hh.laborSkillFactor.last), "payWage0", sim, t)
              if(sim.test) assert(_cash.last > 0)
          }// foreach

      }// tradBanks match

}, "firm_payWages", sim)
}







/*  ------------------------------------------------------------ Sales of Goods  --------------------------------------------------------  */
```

Firm.scala

```scala
/**
 *
 *  Firms produce each period/tick an amount of the good bundle that depends on its current amount of employees and their labor skill.
 *
 *    */
def produceGood (t:Int) {
  time({
    _producedGoods += actualProductionCapacity
    if(t<2) deposit(_amountOfInventory, _producedGoods.last, t, sim) else _amountOfInventory += rounded( _amountOfInventory.last + _producedGoods.last )
  }, "firm_produceGood", sim)
  }






/**
 *
 *  Firms set the price for their produced goods of the current period according to the corresponding costs (labor costs, external financing etc.) plus a mark up.
 *
 *    */
def determinePrice (t:Int) {
  time({
    val expFee           = 50.0
    val expLaborCosts    = (potentialLaborSkillCostsWeekly + production2skill(currentProductionDemand) * _offeredWages.last ) * sim.updateFrequency
    val expInterestCosts = if(houseBank.listOfDebtors.contains(this)) houseBank.listOfDebtors(this).map( _.interestPayments.filterKeys(_ >= t).filterKeys(_ < t + sim.updateFrequency).values.sum).sum else 0.0
    val expFixCosts      = expFee
    val expVariableCosts = expInterestCosts + expLaborCosts

    tradBanks match {
      case true =>
          val expTotalCosts = expFixCosts + expVariableCosts
          val expUnitCosts  = expTotalCosts / (_productionTarget.last * sim.updateFrequency)
          val markUp        = 0.1
          val newPrice      = if(t < 100) rounded(expUnitCosts * ( markUp + math.exp( 0.012 / (48 / sim.updateFrequency) ))) else rounded(expUnitCosts * (1 + markUp + sim.expPi.last))
          if(newPrice / _price.last < 0.985 || newPrice / _price.last > 1.015) _price += newPrice else _price += _price.last

      case false =>
        val expTotalCosts = expFixCosts + expVariableCosts
        val expUnitCosts  = expTotalCosts / _productionTarget.last
        val markUp        = 0.3
        if(employees.nonEmpty) price += rounded(expUnitCosts * (1 + markUp)) else price += goodsMarket.weightedAvgPriceOfTick.last

    }// end match
  }, "firm_determinePrice", sim)
}
```

```
/**
 *
 *    */
def offerPrice {goodsMarket.priceIndex   += (this -> _price.last)}

/**
 *
 *  After the production of the period, firms offer their goods in stock on the goods market.
 *
 *    */
def offerGood (t:Int) = {
  time( {
    goodsMarket.offeredGoods += (this -> _amountOfInventory.last)
    offerPrice
  }, "firm_offerGood", sim)
}

/**
 *
 *  At the end of each tick, firms check whether they have debt obligations to meet, i.e. whether they have to pay interest or principal payments.
 *
 *    */
def repayLoan (t:Int) {
  time({
  val b = new Breaks
  val c = new Breaks
    tradBanks match {
      case false =>
        b.breakable{
          _privateLender.foreach{
            loan =>
              if(loan._2._3 == t){
                if(sim.pln) println(_privateLender)
                if(sim.pln) println(this + " has to repay " + Seq(loan._2._1, loan._2._2).sum + " and has cash of " + _cash.last)
                if(loan._2._1 + loan._2._2 <= _cash.last){
                  transferMoney(this, loan._1, loan._2._1, "repayPrivateLoan", sim, t, loan._2._2)
                  loan._1.privateBorrower -= this
                  privateLender           -= loan._1
                } else {
                  transferMoney(this, loan._1, loan._2._1, "repayPrivateLoanPartially", sim, t, loan._2._2)
                  loan._1.privateBorrower -= this
                  privateLender           -= loan._1
                  shutDownFirm(t, "illiquidy")
                  b.break
```

```
              }// else
            }// if
        }// foreach
      }// breakable


    case true =>

      // traditional Banks
      b.breakable{
        if(houseBank.listOfDebtors.contains(this) && houseBank.listOfDebtors(this).nonEmpty){
          if(sim.test) require(!houseBank.listOfDebtors.contains(null), s"listOfDebtors of $houseBank contains null")
          val listOfLoans = houseBank.listOfDebtors(this).clone()

          for(loan <- listOfLoans){
            // interest
            if(loan.interestPayments.contains(t)){
              if(rounded(loan.interestPayments(t)) <= _bankDeposits.last) transferMoney(this, houseBank, loan.interestPayments(t), "payInterestOnBankLoan", sim, t) else {
                transferMoney(this, houseBank, _bankDeposits.last, "payInterestOnBankLoanPartially", sim, t)
                shutDownFirm(t, "illiquidy")
                b.break
              }// else
            }// if

            // principal payments
            if(loan.principalPayments.contains(t)){
              if(rounded(loan.principalPayments(t)) <= _bankDeposits.last){
                transferMoney(this, houseBank, loan.principalPayments(t), "repayBankLoan", sim, t)
              } else {
                transferMoney(this, houseBank, _bankDeposits.last, "repayBankLoanPartially", sim, t)
                shutDownFirm(t, "illiquidy")
                b.break
              }// else
            }// if
          }//for-loop
        }// if
      }// breakable

      // Shadow Banks
      c.breakable{
        if(houseShadowBank.listOfDebtors.contains(this) && houseShadowBank.listOfDebtors(this).nonEmpty){
          if(sim.test) require(!houseShadowBank.listOfDebtors.contains(null), s"listOfDebtors of $houseShadowBank contains null")
          val listOfLoans = houseShadowBank.listOfDebtors(this).clone()
          for(loan <- listOfLoans){
            // interest
            if(loan.interestPayments.contains(t)){
              println(s"$this has to pay interest of ${loan.interestPayments(t)}")
              if(rounded(loan.interestPayments(t)) <= _bankDeposits.last){
                println(s"$this has enough bD: ${_bankDeposits.last} --> payInterestOnBrokerDealerLoan")
                transferMoney(this, houseShadowBank, loan.interestPayments(t), "payInterestOnBrokerDealerLoan", sim, t)
              } else {
                println(s"$this has not enough bD: ${_bankDeposits.last} --> payInterestOnBrokerDealerLoanPartially + shutDown")
                if(_bankDeposits.last > 0) transferMoney(this, houseShadowBank, _bankDeposits.last, "payInterestOnBrokerDealerLoanPartially", sim, t)
```

```scala
                  shutDownFirm(t, "illiquidy")
                  c.break
                }// else
              }// if
              require(houseShadowBank.bankDeposits.last > 0, s"neg bD after $this pays interest on Loans: ${houseShadowBank.printBSP}")
              // principal payments
              if(loan.principalPayments.contains(t)){
                println(s"$this has to repay loan of ${loan.principalPayments(t)}")
                if(rounded(loan.principalPayments(t)) <= _bankDeposits.last){
                  println(s"$this has enough bD: ${_bankDeposits.last} --> repayBrokerDealerLoan")
                  transferMoney(this, houseShadowBank, loan.principalPayments(t), "repayBrokerDealerLoan", sim, t)
                } else {
                  println(s"$this has not enough bD: ${_bankDeposits.last} --> repayBrokerDealerLoanPartially + shutDown")
                  if(_bankDeposits.last > 0) transferMoney(this, houseShadowBank, _bankDeposits.last, "repayBrokerDealerLoanPartially", sim, t)
                  shutDownFirm(t, "illiquidy")
                  c.break
                }// else
              }// if
            }//for-loop
          }// if
        }// breakable
        require(houseShadowBank.bankDeposits.last > 0, s"neg bD after $this repays prinicipal: ${houseShadowBank.printBSP}")

  }// match
 }, "firm_repayLoans", sim)
}



/**
 *
 *    */
def deleteDueLoans (t:Int) = {

}





/**
 *
 *  If a firm is not able to meets its debt obligations, it exits the market by shutting down its production process.
 *  After a random amount of time, the firm enters the market (is reactivated) as a new firm and starts the production of goods again.
 *
 *    */
def shutDownFirm (t:Int, cause:String) {time({
  _active               = false                                                   // deactivate firm
  _periodOfReactivation = t + 24
  _insolvencies(_insolvencies.size-1) += 1                                        // increase firm insolvency counter by 1
```

```scala
  def clearBDLoans (cause:String) = {
    if(houseShadowBank.listOfDebtors.contains(this) && houseShadowBank.listOfDebtors(this).nonEmpty) {
      require(houseShadowBank.businessLoans.last >= 0, s"businessLoans of $houseShadowBank are negative before shutDown of $this (ne) (${houseShadowBank.businessLoans.last})")
      transferMoney(
          this,
          houseShadowBank,
          houseShadowBank.listOfDebtors(this).map(_.principalPayments.filterKeys(_ > t).values.sum).sum,
          "negativeEquity1",
          sim,
          t,
          houseShadowBank.listOfDebtors(this).map(_.interestPayments.filterKeys(_ > t).values.sum).sum
      )
      require(houseShadowBank.businessLoans.last >= 0, s"businessLoans of $houseShadowBank are negative after shutDown of $this (ne) (${houseShadowBank.businessLoans.last})")
      houseShadowBank.listOfDebtors -= this
    }
  }



  def repayCapital2Owners = {
    val shareOfDeposits = owners.map(owner => owner -> _bankDeposits.last * owner.shareOfCorporations(this)).toMap
    if(sim.test) require(_bankDeposits.last == shareOfDeposits.values.sum, s"dev is ${_bankDeposits.last} / ${shareOfDeposits.values.sum}")
    owners.foreach{
      owner =>
        if(sim.pln) println(s"Since $this is bankrupt due to neg equity and deposits left it repays ${shareOfDeposits(owner)} to $owner according to its share of the Firm (${owner.shareOfCorporations(this)}).")
        transferMoney(this, owner, shareOfDeposits(owner), "repayCapital", sim, t)
    }
  }



  // clear firms financial claims
  cause match {
    case "negative equity"  =>
      tradBanks match {

        case true =>
          // for trad. Banks
          if(sim.pln) println(s"""
            ####################################################################################################################################################
            $this is shut down: negative equity. ${houseBank} must write off all outstanding loans to $this. There are bank deposits left (${_bankDeposits.last})
            ####################################################################################################################################################""")
          if(houseBank.listOfDebtors.contains(this) && houseBank.listOfDebtors(this).nonEmpty) {
            if(sim.pln){
              printBSP
              houseBank.printBSP
            }
            if(sim.pln) println(s"$t -> ${houseBank.listOfDebtors(this)}")
            houseBank.listOfDebtors(this).foreach(x => if(sim.pln) println(x.principalPayments))
            houseBank.listOfDebtors(this).foreach(x => if(sim.pln) println(x.interestPayments))
            // clear Loans from trad. Banks
```

```scala
              transferMoney(
                  this,
                  houseBank,
                  houseBank.listOfDebtors(this).map(_.principalPayments.filterKeys(_ > t).values.sum).sum,
                  "negativeEquity1",
                  sim,
                  t,
                  houseBank.listOfDebtors(this).map(_.interestPayments.filterKeys(_ > t).values.sum).sum
              )
              houseBank.listOfDebtors -= this                                           // clear this loan at houseBank
            }
            if(sim.pln) printBSP
            if(sim.pln) println(s"$this has money left (${_bankDeposits.last}) and distributes it to owners ($owners)")
            if(sim.test) require(rounded(owners.map(_.shareOfCorporations(this)).sum) == 1, s"Owners of $this own more than 100%: ${owners.map(_.shareOfCorporations(this)).sum} (from $owners)")
            if(sim.test) require(_bankDeposits.last < math.max(10, _bankDeposits.last * 0.000001), s"$this has deposits left after serving equity holders (${_bankDeposits.last})")
            clearBDLoans("ne")
            if(_bankDeposits.last > 0) repayCapital2Owners
            if(sim.test) require(_bankDeposits.last < 1, s"$this has deposits left after serving debt and equity holders (${_bankDeposits.last})")
            clearFirmDebt
            if(sim.test) assert(!houseBank.listOfDebtors.contains(this) || houseBank.listOfDebtors(this).isEmpty, s"There are bank loans left after shut down of $this")
            if(sim.test) testOfBSP("negative equity")




          case false  =>
            if(sim.pln) println(this + " is shut down: negative equity. Private Lenders have to get their share of cash back: " + _privateLender + "  ")
            if(_privateLender.nonEmpty) {
              val sumOfClaims = _privateLender.map(loan => loan._2._1 + loan._2._2).sum
              val restOfCash = _cash.last
              _privateLender.foreach{loan =>
                val lendersShareOfMoney = (loan._2._1 + loan._2._2) / sumOfClaims
                transferMoney(this, loan._1, lendersShareOfMoney * restOfCash, "negativeEquity0", sim, t)
                withdraw(loan._1.loans,  loan._2._1 + loan._2._2, t, sim)
                loan._1.privateBorrower -= this
                _privateLender          -= loan._1
              }// foreach
              clearFirmDebt
            }// if
            if(sim.test) assert(_privateLender.isEmpty, "There are private lenders left after shut down of " + this)
            if(_cash.last>0) owners.foreach(owner => transferMoney(this, owner, _cash.last * owner.shareOfCorporations(this), "negativeEquity0", sim, t))
            if(sim.test) testOfBSP("negative equity")

        }// tradBanks match



      case "illiquidy" =>
        tradBanks match {

          case true =>
```

```
            if(sim.pln) println(s"""
              ##############################################################################################################################################
              $this is shut down: cannot repay its bankLoans. ${houseBank} must write off all outstanding loans to $this. There are bank deposits left (${_bankDeposits.last})
              ##############################################################################################################################################""")
            if(houseBank.listOfDebtors.contains(this) && houseBank.listOfDebtors(this).nonEmpty){
              val ppLoss = houseBank.listOfDebtors(this).map(_.principalPayments.filterKeys(_ > t).values.sum).sum
              val ipLoss = houseBank.listOfDebtors(this).map( _.interestPayments.filterKeys(_ > t).values.sum).sum
              deposit( houseBank.loanLosses, ppLoss + ipLoss, t, sim)
              withdraw(houseBank.businessLoans,       ppLoss, t, sim)
              withdraw(houseBank.interestReceivables, ipLoss, t, sim)
              houseBank.listOfDebtors -= this
            }
            if(houseShadowBank.listOfDebtors.contains(this) && houseShadowBank.listOfDebtors(this).nonEmpty){
              val ppLoss = houseShadowBank.listOfDebtors(this).map(_.principalPayments.filterKeys(_ > t).values.sum).sum
              val ipLoss = houseShadowBank.listOfDebtors(this).map( _.interestPayments.filterKeys(_ > t).values.sum).sum
              deposit( houseShadowBank.loanLosses, ppLoss + ipLoss, t, sim)
              require(houseShadowBank.businessLoans.last >= 0, s"businessLoans of $houseShadowBank are negative before shutDown of $this (illiquidity) (${houseShadowBank.businessLoans.last})")
              withdraw(houseShadowBank.businessLoans,       math.min(ppLoss, houseShadowBank.businessLoans.last), t, sim)
              require(houseShadowBank.businessLoans.last >= 0, s"businessLoans of $houseShadowBank are negative after shutDown of $this (illiquidity) (${houseShadowBank.businessLoans.last})")
              withdraw(houseShadowBank.interestReceivables, ipLoss, t, sim)
              houseShadowBank.listOfDebtors -= this
            }
            clearFirmDebt
            if(sim.test) testOfBSP("illiquidy")


          case false  =>
            if(sim.pln) println(this + " is shut down: cannot repay its private loan.  ")
            if(_privateLender.nonEmpty) _privateLender.foreach{loan =>
              withdraw(loan._1.loans,  loan._2._1 + loan._2._2, t, sim)
              loan._1.privateBorrower -= this
              _privateLender      -= loan._1
            }
            clearFirmDebt
            if(sim.test) testOfBSP("illiquidy")

        }// tradBanks match


      case "illiquidyWage" =>
        tradBanks match {

          case true =>
            if(sim.pln) println(s"""
              ##############################################################################################################################################
              $this is shut down: cannot pay wages. ${houseBank} must write off all outstanding loans to $this. There are bank deposits left (${_bankDeposits.last})
              ##############################################################################################################################################""")
              if(houseBank.listOfDebtors.contains(this) && houseBank.listOfDebtors(this).nonEmpty){
                val currentDeposits = bankDeposits.last
                val ppLoss = houseBank.listOfDebtors(this).map(_.principalPayments.filterKeys(_ > t).values.sum).sum
                val ipLoss = houseBank.listOfDebtors(this).map( _.interestPayments.filterKeys(_ > t).values.sum).sum
```

```
                withdraw(          bankDeposits,        currentDeposits, t, sim)
                withdraw(houseBank.retailDeposits,      currentDeposits, t, sim)
                if(ppLoss + ipLoss > currentDeposits) deposit(houseBank.loanLosses,  ppLoss + ipLoss - currentDeposits, t, sim)
                withdraw(houseBank.businessLoans,       ppLoss, t, sim)
                withdraw(houseBank.interestReceivables, ipLoss, t, sim)
                houseBank.listOfDebtors -= this
              }
              if(houseShadowBank.listOfDebtors.contains(this) && houseShadowBank.listOfDebtors(this).nonEmpty){
                val currentDeposits = bankDeposits.last
                val ppLoss = houseShadowBank.listOfDebtors(this).map(_.principalPayments.filterKeys(_ > t).values.sum).sum
                val ipLoss = houseShadowBank.listOfDebtors(this).map( _.interestPayments.filterKeys(_ > t).values.sum).sum
                withdraw(               bankDeposits, currentDeposits, t, sim)
                deposit(houseShadowBank.bankDeposits, currentDeposits, t, sim)
                if(houseBank != houseShadowBank.houseBank){
                  if(houseBank.cbReserves.last < currentDeposits) houseBank.getIntraDayLiquidity(currentDeposits, t)
                  withdraw(                houseBank.retailDeposits, currentDeposits, t, sim)
                  deposit( houseShadowBank.houseBank.retailDeposits, currentDeposits, t, sim)
                  withdraw(                houseBank.cbReserves,     currentDeposits, t, sim)
                  deposit( houseShadowBank.houseBank.cbReserves,     currentDeposits, t, sim)
                  require(sim.reserveFlows(houseBank)(houseShadowBank.houseBank).size == t, s"registerReserveFlow failed because of too many entries in Array")
                  sim.reserveFlows(houseBank)(houseShadowBank.houseBank)(t-1) += currentDeposits
                }
                if(ppLoss + ipLoss > currentDeposits) deposit(houseShadowBank.loanLosses,  ppLoss + ipLoss - currentDeposits, t, sim)
                require(houseShadowBank.businessLoans.last >= 0, s"businessLoans of $houseShadowBank are negative before shutDown of $this (illiquidityWage) (${houseShadowBank.businessLoans.last})")
                withdraw(houseShadowBank.businessLoans,       math.min(ppLoss, houseShadowBank.businessLoans.last), t, sim)
                require(houseShadowBank.businessLoans.last >= 0, s"businessLoans of $houseShadowBank are negative after shutDown of $this (illiquidityWage) (${houseShadowBank.businessLoans.last})")
                withdraw(houseShadowBank.interestReceivables, ipLoss, t, sim)
                houseShadowBank.listOfDebtors -= this
              }
              clearFirmDebt
              if(sim.test) testOfBSP("illiquidyWage")


          case false  =>
            if(sim.pln) println(this + " is shut down: cannot pay wages.  ")
            if(_privateLender.nonEmpty) _privateLender.foreach{loan =>
              withdraw(loan._1.loans,  loan._2._1 + loan._2._2, t, sim)
              loan._1.privateBorrower -= this
              _privateLender          -= loan._1
            }
            clearFirmDebt
            if(sim.test) testOfBSP("illiquidyWage")

        }// tradBanks match
    }// cause match

    do{
      owners.foreach{
        hh =>
          if(hh != null){
            if(sim.test) assert(hh.foundedCorporations.contains(this), hh.foundedCorporations + " does not include " + this + "?")
```

```scala
          hh.foundedCorporations -= this
          if(sim.test) assert(hh.shareOfCorporations.contains(this), hh.shareOfCorporations + " does not include " + this + "?")
          hh.shareOfCorporations -= this
          if(sim.test) assert(owners.contains(hh), owners + " does not include " + hh + "?")
          owners                 -= hh
        }
    }
  } while (owners.nonEmpty)
    if(sim.test) assert(owners.isEmpty, {if(sim.pln) println(owners); sys.error("There are owners left after shut down")})

  // fire all current employees
  do _employees.foreach( hh => if(hh != null) fireHH(hh) ) while (_employees.nonEmpty)
  if(sim.test) assert(_employees.isEmpty, this + " is shut down but there are unfired employees left!")
  if(sim.test) assert(_wageBill.isEmpty)
  _queuedEmployees.keys.foreach(_.currentEmployer = arge)
  _queuedEmployees.clear

  printFirmData

  def testOfBSP (cause:String) {
    if(tradBanks) if(sim.test){
      assert(_bankDeposits.last < math.max(10, _bankDeposits.last * 0.000001), s"$this has deposits left after shut down ($cause): ${_bankDeposits.last}")
    } else if(sim.test){
      assert(_cash.last < 0.1, s"$this has cash left after shut down ($cause): " + _cash.last)
    }
    if(sim.test) assert(_debtCapital.last    < 0.1, s"$this has debt left after shut down: " + _debtCapital.last)
    if(sim.test) assert(_interestOnDebt.last < 0.1, s"$this has debt left after shut down: " + _interestOnDebt.last)
  }

  def clearFirmDebt {
    _debtCapital(_debtCapital.size-1)       = 0.0
    _interestOnDebt(_interestOnDebt.size-1) = 0.0
  }

  }, "firm_shutDownFirm", sim)
}








def printFirmData = {
  // BSP
  if(sim.pln){
    println(this + " ----- BSP -----  ")
    println(this + " inventory: " + _inventory.last + "  ")
    println(this + " bankDeposits: " + _bankDeposits.last + "  ")
    println(this + " cash: " + _cash.last + "  ")
    println(this + " debtCapital: " + _debtCapital.last + "  ")
    println(this + " interestOnDebt: " + _interestOnDebt.last + "  ")
```

```
    println(this + " TA: " + _totalAssets.last + "  ")
    println(this + " equity: " + _equity.last + "  ")
    // other data
    println(this + " ----- Other data -----  ")
    println(this + " insolvencies: " + _insolvencies.last + "  ")
    println(this + " costOfGoodsSold: " + _costOfGoodsSold.last + "  ")
    println(this + " revenues: " + _revenues.last + "  ")
    println(this + " privateLender: " + _privateLender + "  ")
    println(this + " prodTarget: " + _productionTarget.last + "  ")
    println(this + " producedGoods: " + _producedGoods.last + "  ")
    println(this + " amountOfInventory: " + _amountOfInventory.last + "  ")
    println(this + " queuedEmployees: " + _queuedEmployees + "  ")
    println(this + " employees: " + _employees + "  ")
    println(this + " numberOfEmployees: " + _numberOfEmployees.last + "  ")
    println(this + " price: " + _price.last + "  ")
    println(this + " sales: " + _sales.last + "  ")
    println(this + " valuedInventory: " + _valuedInventory + "  ")
  }
}






/**
 *
 *  After a default of a firm agent, there is a possibility that a new firm enters the market (from a technical point of view, the entirely cleaned but still existing bank object is reactivated) if there are
 *  enough HH that provide sufficient liquidity to found a new firm.
 *
 *    */
def reactivateFirm (t:Int) {time({
  sim.p(t, s"before reactivating $this")
  println(s"Reactivating $this/$houseBank: ${sim.bankList.map{
    bank =>
      bank ->
        (bank.active,
         bank.retailDeposits.last,
         bank.businessClients.map(_.bankDeposits.last).sum,
           bank.retailClients.map(_.bankDeposits.last).sum,
             bank.MMMFClients.map(_.bankDeposits.last).sum,
               bank.BDClients.map(_.bankDeposits.last).sum)}}"
  )
  // renew owners
  if(sim.test) require(owners.isEmpty, {if(sim.pln) println(owners);  sys.error("new activated firm should not have any owners yet")})
  _age = 0                                                              // reset firm age
  val investment = 2500.0
  val newOwners = if(tradBanks){
    random.shuffle(sim.hhList.filter(_.bankDeposits.last >= investment))
  } else {
    random.shuffle(sim.hhList.filter(_.cash.last >= investment)).take(random.nextInt(sim.numberOfHH/sim.numberOfFirms/2))
  }
```

```
    println(s"newOwners: ${newOwners.filter{_.bankDeposits.last < investment}}")
    println(s"newOwners of reactivated $this: $newOwners")
    val newOwnersContribution = newOwners.map(no => no -> investment).toMap
    println(s"NOC: $newOwnersContribution to found $this")
    println(s"NOC_HB: ${newOwners.map { no => no -> (no.houseBank, no.bankDeposits.last) }}")
    sim.bankList.filter(_.active).foreach {
      bank =>
        require(
            newOwners.filter(_.houseBank == bank).size * 2500 <= bank.retailDeposits.last,
            s"$bank has not enough rD to transfer all the fund of the owners: ${newOwners.filter(_.houseBank == bank).size * 2500} / ${bank.retailDeposits.last}"
        )
    }
    println(s"newOwnersContribution of reactivated $this: $newOwnersContribution")
    println(s"newOwners.size (${newOwners.size}) / initialCapital/investment (${(sim.initialCapital/investment).toInt})")
    if(newOwners.nonEmpty) {
      _active = true
      newOwners.foreach{
        hh =>
          owners                  += hh                                                         // relationship on corp-side
          hh.foundedCorporations += this                                                        // relationship on   hh-side
          hh.shareOfCorporations += this -> (newOwnersContribution(hh) / newOwnersContribution.values.sum)
          if(sim.pln) println(s"$hh founded $this with a share of ${newOwnersContribution(hh) / newOwnersContribution.values.sum}")
      }
      sim.p(t, s" reactivating $this")
      require(newOwners == owners, s"Owners wrong...")
      if(sim.test) require(rounded(owners.map(_.shareOfCorporations(this)).sum) == 1, s"${owners.map(_.shareOfCorporations(this)).sum}")
      if(tradBanks) {
        owners.foreach{
          owner =>
            sim.p(t, s"before transfering investment from $owner to $this")
            println(s"transfering $investment from $owner (${owner.bankDeposits.last}) to $this: owner_HB_rD [${owner.houseBank}]: ${owner.houseBank.retailDeposits.last} / 
HB_rd: ${houseBank.retailDeposits.last}")
            sim.bankList.foreach{ bank => println(bank.retailClients.map{ hh => hh -> hh.bankDeposits.last }) }
            require(
                owner.bankDeposits.last >= investment && owner.houseBank.retailDeposits.last >= investment,
                s"bD of $owner are too low: ${owner.bankDeposits.last} || rD of owners houseBank [${owner.houseBank}] is too low: ${owner.houseBank.retailDeposits.last}"
            )
            transferMoney(owner, this, newOwnersContribution(owner), "reactivateFirm1", sim, t)
            sim.p(t, s"after tranfering investment from $owner to $this")
          }
      } else owners.foreach(owner => transferMoney(owner, this, 100, "reactivateFirm0", sim, t))
    } else {
      _periodOfReactivation = t + 4
      if(sim.pln) println("Currently no entrepreneurs around here to reactivate " + this)
    }
    printFirmData
    sim.p(t, s"after reactivating $this")
  }, "firm_reactivateFirm", sim)
 }



 /**
```

```scala
 *
 *  Firms have to pay a fee to their house bank, since they use the payment system through their bank account.
 *
 *    */
def payBankAccountFee (t:Int) = if(_bankDeposits.last >= 50) transferMoney(this, houseBank, 50, "payBankAccountFee", sim, t)




val _aLaborCosts   = ArrayBuffer[Double]()
val _interestCosts = ArrayBuffer[Double]()

/**
 *
 *    Cost of goods sold of the current period.
 *
 *    */
def determineCOGS (t:Int) = {
  val fee           = 50.0 / sim.updateFrequency
  val laborCosts    = actualLaborSkillCostsWeekly
  val interestCosts = if(houseBank.listOfDebtors.contains(this)) houseBank.listOfDebtors(this).map( _.interestPayments.filterKeys(_ == t).values.sum).sum else 0.0
  _costOfGoodsSold += fee + laborCosts + interestCosts
  _aLaborCosts     += laborCosts
  _interestCosts   += interestCosts
}





/**
 *
 *   Profit and loss account for firms.
 *
 *    */
def determineProfit (t:Int) = {
  _revenues += rounded( _sales.last * _price.last )
  determineCOGS(t)
  if(_costOfGoodsSold.isEmpty) profit += _revenues.last else profit += rounded( _revenues.last - _costOfGoodsSold.last )
}




private var _lossCarriedForward = 0.0
private var _deferredTax        = 0.0




/**
 *
 *   Firm agents have to pay taxes on the revenue they made during the course of the fiscal year.
```

```
   *
   *    */
  def payTaxes (t:Int, taxableResultOfCurrentPeriod:Double = sumOfNPastPeriods(profit, 48), cause:String = if(tradBanks) "corporateTax1" else "corporateTax0") = {time({
    // determine tax liability for current period
    val taxForCurrentPeriod:Double = taxableResultOfCurrentPeriod match {
      case taxableResultOfCurrentPeriod:Double if taxableResultOfCurrentPeriod <= 0 =>
        _lossCarriedForward += taxableResultOfCurrentPeriod
        0.0
      case taxableResultOfCurrentPeriod:Double if taxableResultOfCurrentPeriod >  0 =>
        {taxableResultOfCurrentPeriod - _lossCarriedForward} match {
        case netTaxableResult:Double if netTaxableResult <= 0 =>
          _lossCarriedForward -= taxableResultOfCurrentPeriod
          0.0
        case netTaxableResult:Double if netTaxableResult >  0 =>
          val residual = taxableResultOfCurrentPeriod - _lossCarriedForward
          _lossCarriedForward -= _lossCarriedForward
          residual * government.corporateTax.last
        case _ => 0.0
      }
      case _ => 0.0
    }
    // pay taxForCurrentPeriod and incorporate possible deferred taxes from the past periods
    _deferredTax += taxForCurrentPeriod
    if(_deferredTax > 0){
      {_bankDeposits.last - _deferredTax} match {
        case r:Double if r >= 0 =>
          transferMoney(this, government, _deferredTax, cause, sim, t)
          _deferredTax -= _deferredTax
        case r:Double if r <  0 =>
          _deferredTax -= _bankDeposits.last
          transferMoney(this, government, _bankDeposits.last, cause, sim, t)
      }
    }

    }, "firm_payTaxes", sim)
  }





  /**
   *
   *   If there is a positive profit after tax, firms distribute a part of the rest among their equityholders.
   *
   *    */
  def payOutDividends2Owners (t:Int, profitAfterTax:Double = sumOfNPastPeriods(profit,48) * (1 - government.corporateTax.last), cause:String = if(tradBanks) "dividends1" else
"dividends0") {
    if(sim.test) require(owners.nonEmpty, this + " has no owners to pay out dividends!")
    if(sim.pln) println(s"$this has deposits of ${_bankDeposits.last} and, thus, can distribute $retainedEarningsParameter of its profitAfterTax ($profitAfterTax) to its
owners.")
    if(profitAfterTax > 0) owners.foreach(hh => transferMoney(this, hh, (retainedEarningsParameter * math.min(_bankDeposits.last, profitAfterTax)) *
hh.shareOfCorporations(this), cause, sim, t))
```

```scala
  }




  /**
   *
   *  At the end of each fiscal year, the firm agent makes an annual report to update its balance sheets statements in order to check its solvency and financial soundness.
   *
   *     */
  def makeAnnualReport (t:Int) {time({
    profitabilityOfOperatingBusiness
    determineProfit(t)
    if(_active){
      determineUtilGapOfTick
      determineEmployGapOfTick
      if(t>1 && t % 48 == 0) payTaxes(t)                                                                                          // pay taxes to government
      if(t>1 && t % 48 == 0) payOutDividends2Owners(t)                                                                            // pay dividends

      // inventory
      if(producedGoods.last - sales.last > 0) valuedInventory += t -> (producedGoods.last - sales.last, price.last)                // determines BSP inventory
      inventory update(inventory.length-1, rounded(valuedInventory.values.toList.map{ case(quantity, price) => quantity * price}.sum) )              // value the unsold goods -> inventory (BSP)
      if(_amountOfInventory.last.toInt - rounded(initialInventory + _producedGoods.sum - _sales.sum).toInt > 1 || _amountOfInventory.last.toInt - rounded(initialInventory + _producedGoods.sum - _sales.sum).toInt < -1){
        if(sim.pln){
          println(this + " Amount of unsold goods in NOT correct: " + rounded(initialInventory + _producedGoods.sum - _sales.sum).toInt + "/" + _amountOfInventory.last.toInt)
        }
      }

      // AR
      val TA = rounded( Seq(inventory.last, bankDeposits.last, cash.last).sum )
      val TL = rounded( Seq(debtCapital.last, interestOnDebt.last).sum )
      totalAssets += TA
      if(sim.pln) println("Total assets of " + this + ": " + inventory.last + " + " + bankDeposits.last + " + " + cash.last + " = " + totalAssets.last)
      equity      += rounded( TA - TL )
      if(sim.pln) println("Equity of " + this + ": " + totalAssets.last + " - (" + debtCapital.last + " + " + interestOnDebt.last + ") = " + equity.last)

      // test whether equity is correctly calculated
      if(sim.test) require( SE(TA, TL + equity.last), s"Annual Report of $this is not correct: (A) $TA / (L) ${rounded( TL + equity.last )}")
      if(equity.last < 0){
        printlnBSP
        if(sim.pln) println(s"$this inventory ${inventory.last}")
        shutDownFirm(t, "negative equity")
      }
    } else {
      _utilizationGap += 0.0
      _employmentGap  += 1.0
      _totalAssets    += 0.0
      _equity         += 0.0
```

```
  }
}, "firm_makeAnnualReport", sim)}




/**
 *
 *  This method prints the firm agent's current balance sheet.
 *
 *    */
def printBSP = {
  println(f"""
           A                 $this               P
         ---------------------------------------------
         inve ${_inventory.last}%15.2f  | debt ${_debtCapital.last}%15.2f
         bd   ${_bankDeposits.last}%15.2f  | int  ${_interestOnDebt.last}%15.2f
         cash ${_cash.last}%15.2f  | eq.  ${if(_equity.nonEmpty) f"${_equity.last}%15.2f" else "NaN"}
         ---------------------------------------------
         TA   ${if(_totalAssets.nonEmpty) f"${_totalAssets.last}%15.2f" else "NaN"}  |
                                                                                                          """)
}



/**
 *
 *  These values are jsut for data saving purposes.
 *
 *    */
 val firmEndOfTickData     = Map("vacancies"  -> firm.vacancies,
                                 "employees"        -> employees
                                 "queuedEmployees" -> queuedEmployees
                                                                        )

 val firmEndOfSimulationData = Map(
     "productionTarget"          -> _productionTarget,              // LB[Int]
     "producedGoods"             -> _producedGoods,                 // LB[Int]
     "amountOfInventory"         -> _amountOfInventory,             // LB[Int]
     "valuedInventory"           -> _valuedInventory.toList,        // LinkedHashMap.toList -> List[Int, (Int, Double)]
     "offeredWages"              -> _offeredWages,                  // LB[Double]
     "vacancies"                 -> _vacancies,                     // LB[(Double, Double)]
     "needForExternalFinancing"  -> _needForExternalFinancing,      // LB[Double]
     "numberOfEmployees"         -> _numberOfEmployees,             // LB[Int]
     "price"                     -> _price,                         // LB[Double]
     "sales"                     -> _sales,                         // LB[Int]
     "COGS"                      -> _costOfGoodsSold,               // LB{Double]
     "revenues"                  -> _revenues,                      // LB{Double]
     "profit"                    -> profit,                         // LB[Double]
     "owners"                    -> owners,                         // LB[HH]
     "privateLender"             -> _privateLender,                 // LB[HH]
     "totalAssets"               -> _totalAssets,                   // LB[Double]
```

```
    "equity"                    -> _equity,                    // LB[Double]
    "insolvencies"              -> _insolvencies,              // LB[Int]
    "creditRationed"            -> _creditRationed,            // AB[Int]
    "debtToEquityTarget"        -> _debtToEquityTarget,        // AB[Double]
    "ptDecision"                -> _ptDecision,
    "pastInventory"             -> _pastInv,
    "pastProduction"            -> _pastProd,
    "debtFinancing"             -> debtFinancing,
    "currentProdCap"            -> _currentProdCap,
    "COGS_interestCosts"        -> _interestCosts,
    "COGS_aLaborCosts"          -> _aLaborCosts,
    "interestOfferedOnBankLoan" -> _interestOfferedOnBankLoan
    "profitabilityOfOB"         -> _profitabilityOfOB,
    "utilizationGap"            -> _utilizationGap,
    "employmentGap"             -> _employmentGap,
    "utilizationGapWeighted"    -> _utilizationGapWeighted,
    "employmentGapWeighted"     -> _employmentGapWeighted

}
```

A.6 Public Sector

A.6.1 Government Class

Government.scala

```scala
/**
 *
 */

package monEcon.publicSector

import monEcon.Agent
import monEcon.financialSector._
import monEcon.realSector._
import monEcon.Simulation
import monEcon.Markets._
import monEcon.bonds

import collection.mutable._
import math._
import util.Random

/**
 * @author Krugman
 *
 */


// ---------- Class for Government-Object ----------
case class Government (tradBanks                 :Boolean,          //
                       initialMoney              :Double,           //
                       CB                        :CentralBank,      //
                       goodsMarket               :GoodsMarket,      //
                       laborMarket               :LaborMarket,      //
                       interbankMarket           :InterbankMarket,  //
                       initialUnemploymentBenefit:Double,           //
                       initialVAT                :Double,           //
                       initialCorporateTax       :Double,           //
                       initialTaxOnCapitalGains  :Double,           //
                       subsidyFraction           :Double,           //
                       nbcParameter              :Double,           //
                       sim                       :Simulation        //
                                                                        ) extends Agent with bonds {

  val name              = "Government"
  override def toString = name


  /*  ----------------------------------------  government balance sheet positions  ----------------------------------------  */
  // Asset Side
  private val _bankDeposits = ArrayBuffer[Double](0.0)              //
  private val _cbDeposits   = ArrayBuffer[Double](0.0)              //
  private val _cash         = ArrayBuffer[Double](initialMoney)     //
  // -------------------------------------------------------------
  private val _totalAssets  = ArrayBuffer[Double]()                 // sum of all assets

  // Liabilities Side
  //  private val  bonds      = ArrayBuffer[Double](0.0)            // i.e. publicDebt
```

```scala
private val _equity       = ArrayBuffer[Double](0.0)              // government deficit


/**
 *
 *  This is just to save balance sheet data.
 *
 *    */
val governmentBSP = Map("bankDeposits"  ->  _bankDeposits,
                        "cbDeposits"    ->  _cbDeposits,
                        "cash"          ->  _cash,
                        "bonds"         ->  bonds,
                        "totalAssets"   ->  _totalAssets,
                        "equity"        ->  _equity
                                                                  )


// other data
private val _VAT                       = ArrayBuffer(initialVAT)                   //
private val _VATrevenue                = ArrayBuffer(0.0)                          //
private val _corporateTax              = ArrayBuffer(initialCorporateTax)          //
private val _corporateTaxRevenue       = ArrayBuffer(0.0)                          //
private val _capitalGainsTax           = ArrayBuffer(initialTaxOnCapitalGains)     //
private val _capitalGainsTaxRevenue    = ArrayBuffer(0.0)                          //
private val _incomeTaxRevenue          = ArrayBuffer(0.0)                          //
private val _taxRevenues               = ArrayBuffer(0.0)                          //
private val _govSpending               = ArrayBuffer(0.0)                          //
private val _deficit                   = ArrayBuffer(0.0)                          //
private val _unemploymentBenefit       = ArrayBuffer(initialUnemploymentBenefit)  //
private val _benefitPayed              = new ArrayBuffer[Double]                   //

private val _offeredGovDebt            = ArrayBuffer[stackOfBonds]()               //
private val _govLOB                    = ArrayBuffer[stackOfBonds]()
private val _numberOfExistingBonds     = ArrayBuffer[Int]()
private val _coupon2Pay                = Map[Int, Map[Bank, Double]]()             //
private val _coupon2PayBD              = Map[Int, Map[BrokerDealer, Double]]()     //
private val _coupon2PayCB              = Map[Int, Double]()                        //
private val _dueDebt                   = Map[Int, Map[Bank, Double]]()             //
private val _dueDebtBD                 = Map[Int, Map[BrokerDealer, Double]]()     //
private val _dueDebtCB                 = Map[Int, Double]()                        //


private val _M0                        = ArrayBuffer[Double]()                     // monetary base/CB money/outside money/high-powered money --> CB reserves + currency (notes
+ coins)
private val _M1                        = ArrayBuffer[Double]()                     // M1 or MZM [money of zero maturity] --> currency + deposits held by non-bank private
sector
private val _M3                        = ArrayBuffer[Double]()                     // broad money or M3 = M1/M2 + RePos]
private val _GDP                       = ArrayBuffer[Double](0.0)                  // nominal GDP
private val _realGDP                   = ArrayBuffer[Double](0.0)                  // real GDP
private val _GDPdeflator               = ArrayBuffer[Double]()                     //
private val _GDPdeflatorMP             = ArrayBuffer[Double]()                     //
private val _nomEconGrowth             = ArrayBuffer[Double]()                     //
private val _nomEconGrowthLog          = ArrayBuffer[Double]()                     //
```

Government.scala

```scala
private val _realEconGrowth           = ArrayBuffer[Double]()                //
private val _realEconGrowthLog        = ArrayBuffer[Double]()                //
private val _productionOfTick         = ArrayBuffer[Double]()                //
private var _NBC                      = 0.0                                  // New Borrowing Criterion (like contract of Maastricht)
private val _lossFromBailOut          = ArrayBuffer[Double](0.0)


// getter
def bankDeposits                   = _bankDeposits
def cbDeposits                     = _cbDeposits
def cash                           = _cash
def totalAssets                    = _totalAssets
def equity                         = _equity
def VAT                            = _VAT
def VATrevenue                     = _VATrevenue
def corporateTax                   = _corporateTax
def corporateTaxRevenue            = _corporateTaxRevenue
def capitalGainsTax                = _capitalGainsTax
def capitalGainsTaxRevenue         = _capitalGainsTaxRevenue
def incomeTaxRevenue               = _incomeTaxRevenue
def taxRevenues                    = _taxRevenues
def govSpending                    = _govSpending
def deficit                        = _deficit
def unemploymentBenefit            = _unemploymentBenefit
def benefitPayed                   = _benefitPayed
def offeredGovDebt                 = _offeredGovDebt
def govLOB                         =  _govLOB
def GDP                            = _GDP
def realGDP                        = _realGDP
def GDPdeflator                    = _GDPdeflator
def GDPdeflatorMP                  = _GDPdeflatorMP
def nomEconGrowth                  = _nomEconGrowth
def nomEconGrowthLog               = _nomEconGrowthLog
def realEconGrowth                 = _realEconGrowth
def realEconGrowthLog               = _realEconGrowthLog
def productionOfTick               = _productionOfTick
def NBC                            = _NBC
def M0                             = _M0
def M1                             = _M1
def M3                             = _M3
def lossFromBailOut                = _lossFromBailOut
def coupon2Pay                     = _coupon2Pay
def coupon2PayBD                   = _coupon2PayBD
def coupon2PayCB                   = _coupon2PayCB
def dueDebt                        = _dueDebt
def dueDebtBD                      = _dueDebtBD
def dueDebtCB                      = _dueDebtCB
def numberOfExistingBonds          = _numberOfExistingBonds
```

```scala
/**
 *
 *  Determines the nominal GDP, i.e. the total value of all produced goods meaning the production of the current fical year in current prices.
 *
 *    */
def determineNominalGDP {time({
  sim.firmList.foreach{
    firm =>
      _GDP(_GDP.size - 1) += firm.producedGoods.last * firm.price.last
      _productionOfTick(_productionOfTick.size-1) += firm.producedGoods.last
  }
 }, "gov_nominalGDP", sim)
}



val wapOFyear = ArrayBuffer[Double]()


/**
 *
 *   Calculates the whole real GDP-time series based on the underlying time series of nominal GDP relative to the current base year, i.e.
 *   the total value of all produced goods or the production of the year in prices of the current base year. To adjust for price changes,
 *   real GDP is calculated using prices from the base year. This allows real GDP to accurately measure changes in output separate from changes in prices.
 *
 *    */
def determineRealGDP (t:Int) {time({
  if(sim.test) require(_productionOfTick.grouped(48).toList.last.size == 48, "governments _productionOfTick has not enough values to calc productionOfYear: " + _productionOfTick.grouped(48).toList.last.size + "/48")
  if(_realGDP.nonEmpty) _realGDP.clear
  val pIndex = if(t % 48 != 0) t % 48 - 1 else 47
  val p = sim.goodsMarket.weightedAvgPriceOfTick.grouped(48).toBuffer(sim.centralBank.baseYear)(pIndex)
  wapOFyear += p
  _productionOfTick.foreach(productionOfTick => _realGDP += productionOfTick * p)
 }, "gov_realGDP", sim)
}


/**
 *
 *   Measure of the level of prices of all new, domestically produced, final goods and services in an economy;
 *   Like the Consumer Price Index (CPI), the GDP deflator is a measure of price inflation/deflation with respect to a specific base year.
 *   (yearly)
 *
 *    */
def calcGDPdeflator {
  val yearlyNominalGDP = _GDP.grouped(48).toList.map(_.sum)
  if(sim.test) require(_GDP.grouped(48).toList.last.size == 48, "GDP has not enough values to calc value of the year")
  if(sim.test) require(yearlyNominalGDP.size == _realGDP.grouped(48).toList.size, "nominal and real GDP differ in size")
  if(_GDPdeflator.nonEmpty) _GDPdeflator.clear
  yearlyNominalGDP.zip(_realGDP.grouped(48).toList.map(_.sum)).foreach(pairOfGDP => _GDPdeflator += (pairOfGDP._1 / pairOfGDP._2) * 100.0 )
}

```

```scala

/**
 *
 *   Measure of the level of prices of all new, domestically produced, final goods and services in an economy;
 *   Like the Consumer Price Index (CPI), the GDP deflator is a measure of price inflation/deflation with respect to a specific base year.
 *   (during the year)
 *
 *    */
def calcGDPdeflatorMP {
  val maintenancePeriod = 6
  val maintenancePeriodNominalGDP = _GDP.grouped(maintenancePeriod).toList.map(_.sum)
  if(sim.test) require(_GDP.grouped(maintenancePeriod).toList.last.size == maintenancePeriod, "GDP has not enough values to calc value of the year")
  if(sim.test) require(maintenancePeriodNominalGDP.size == _realGDP.grouped(maintenancePeriod).toList.size, "nominal and real GDP differ in size")
  if(_GDPdeflatorMP.nonEmpty) _GDPdeflatorMP.clear
  maintenancePeriodNominalGDP.zip(_realGDP.grouped(maintenancePeriod).toList.map(_.sum)).foreach(pairOfGDP => _GDPdeflatorMP += (pairOfGDP._1 / pairOfGDP._2) * 100.0 )
}




/*  ---------------------------------------------------------------------------------------------------------------------------------------------------------
 *  ---------------------------------------------------- Government Bonds ------------------------------------------------------------------------------------
 *  ---------------------------------------------------------------------------------------------------------------------------------------------------------  */

/**
 *
 *  The government bonds are organized in stacks since every stack has an individual ID and it would be do RAM-intensive to assign an ID to every single bond.
 *
 *    */
case class stackOfBonds (amountOfBondsInStack:Int, t:Int = 1) {
  val bond:govBond = govBond(t)
  val id:Long      = setID
}


/**
 *   The government bond class contains all relevant data concerning the counterparties, the interest as well as the coupon.
 *   The couponPayment happens once a year.
 *
 *    */
case class govBond (tickOfSettlement:Int, faceValue:Double = sim.faceValueOfBonds, duration:Int = 240, DIC:Int = 48, tradBanks:Boolean = sim.tradBanks) {
  val couponRate           = if(tradBanks) sim.centralBank.targetFFR.last + 0.015 else 0.05
  val coupon               = faceValue * couponRate
  val accumulatedCoupon    = coupon * (duration / DIC)
  val ticksOfCouponPayment = Vector.tabulate(duration/DIC)(n => tickOfSettlement - 1 + (n+1) * DIC)
  val maturity:Int         = tickOfSettlement - 1 + duration
}






/**
```

```
 *
 *   Government issues an initial amount of bonds and sells it to the commercial/traditional banks. The banks now have the ability
 *   to place them as collateral with the CB to get the required reserves. At period of issuing, govBond is exactly worth the faceValue
 *   (since the market interest rate hasn't change).
 *
 *    */
def issueInitialGovBonds (amount:Double, t:Int = 1) {time({
  tradBanks match {
    case false =>
      _offeredGovDebt += {( (roundUpTo1000(amount)/1000).toLong, govBond(t) )}

    case true  =>
      sim.bankList.foreach{bank => createBondRelationship(bank, amount, "buyInitialGovBonds", t) }

      // CB
      createBondRelationship(CB, amount, "buyInitialGovBonds", t)
  }// match
 }, "Gov_issueInitialGovBonds", sim)
}




/**
 *
 *  This method issues the govBonds in case of a market entry of a newly founded (traditional) bank.
 *
 *    */
def issueGovBondsAtReactivatedBank (reactivatedBank:Bank, amount:Double, t:Int) {time({
  tradBanks match {
    case false =>

    case true  => createBondRelationship(reactivatedBank, amount, "recapitalizeBank", t)

  }// match
 }, "Gov_issueGovBondsAtReactivatedBank", sim)
}




/**
 *
 *  Every time a government bond is created, the government has to add the face value to its debt obligations, i.e.
 *  to the outstanding public debt in order to repay it once the debt becomes due.
 *
 *    */
def addPublicDebt4Repayment (agent:Agent, newSoB:stackOfBonds) {time({
  agent match {
    case bank:Bank =>
      bank.updateBondsAddedWithRelationship(newSoB.amountOfBondsInStack)
      newSoB.bond.ticksOfCouponPayment.foreach{
        tick =>
```

```
          if(_coupon2Pay.contains(tick)) {
            if(_coupon2Pay(tick).contains(bank)){
              _coupon2Pay(tick)(bank) += newSoB.bond.coupon * newSoB.amountOfBondsInStack
            } else {
              _coupon2Pay(tick) += bank -> newSoB.bond.coupon * newSoB.amountOfBondsInStack
            }
          } else {
            _coupon2Pay += tick -> Map(bank -> newSoB.bond.coupon * newSoB.amountOfBondsInStack)
          }
      }
      if(_dueDebt.contains(newSoB.bond.maturity)) {
        if(_dueDebt(newSoB.bond.maturity).contains(bank)){
          _dueDebt(newSoB.bond.maturity)(bank) += newSoB.bond.faceValue * newSoB.amountOfBondsInStack
        } else {
          _dueDebt(newSoB.bond.maturity) += bank -> newSoB.bond.faceValue * newSoB.amountOfBondsInStack
        }
      } else {
        _dueDebt += newSoB.bond.maturity -> Map(bank -> newSoB.bond.faceValue * newSoB.amountOfBondsInStack)
      }

    case bd:BrokerDealer =>
      bd.updateBondsAddedWithRelationship(newSoB.amountOfBondsInStack)
      newSoB.bond.ticksOfCouponPayment.foreach{
        tick =>
          if(_coupon2PayBD.contains(tick)) {
            if(_coupon2PayBD(tick).contains(bd)){
              _coupon2PayBD(tick)(bd) += newSoB.bond.coupon * newSoB.amountOfBondsInStack
            } else {
              _coupon2PayBD(tick) += bd -> newSoB.bond.coupon * newSoB.amountOfBondsInStack
            }
          } else {
            _coupon2PayBD += tick -> Map(bd -> newSoB.bond.coupon * newSoB.amountOfBondsInStack)
          }
      }
      if(_dueDebtBD.contains(newSoB.bond.maturity)) {
        if(_dueDebtBD(newSoB.bond.maturity).contains(bd)){
          _dueDebtBD(newSoB.bond.maturity)(bd) += newSoB.bond.faceValue * newSoB.amountOfBondsInStack
        } else {
          _dueDebtBD(newSoB.bond.maturity) += bd -> newSoB.bond.faceValue * newSoB.amountOfBondsInStack
        }
      } else {
        _dueDebtBD += newSoB.bond.maturity -> Map(bd -> newSoB.bond.faceValue * newSoB.amountOfBondsInStack)
      }

    case cb:CentralBank =>
      newSoB.bond.ticksOfCouponPayment.foreach(
          tick =>
            if(_coupon2PayCB.contains(tick)){
              _coupon2PayCB(tick) += newSoB.bond.coupon * newSoB.amountOfBondsInStack
            } else {
              _coupon2PayCB += tick -> newSoB.bond.coupon * newSoB.amountOfBondsInStack)
            }
      if(_dueDebtCB.contains(newSoB.bond.maturity)) {
```

```scala
        _dueDebtCB(newSoB.bond.maturity) += newSoB.bond.faceValue * newSoB.amountOfBondsInStack
      } else {
        _dueDebtCB += newSoB.bond.maturity -> newSoB.bond.faceValue * newSoB.amountOfBondsInStack
      }
  }
 }, "Gov_addPublicDebt4Repayment", sim)
}




/**
 *
 *  This method is mainly for testing purposes. Issuing government bonds and selling them to a bank means the creation of mutual relationship
 *  that has to be consitent during the course of the simulation.
 *
 *    */
def createBondRelationship (agent:Agent, amount:Double, cause:String, t:Int, test:Boolean = true) {time({

  val amountOfBonds = agent match {
    case agent:Bank         => (  roundUpXk(amount,sim.faceValueOfBonds)/sim.faceValueOfBonds).toInt
    case agent:CentralBank  => (  roundUpXk(amount,sim.faceValueOfBonds)/sim.faceValueOfBonds).toInt
    case agent:BrokerDealer => (roundDownXk(amount,sim.faceValueOfBonds)/sim.faceValueOfBonds).toInt
  }
  if(sim.pln) println(s"$agent has to transfer $amount of gD and buys $amountOfBonds bonds from Gov")
  val priorAmountOfIDs = if(sim.test) agent match {
    case a:Bank         => a.listOfBonds.map{ case(id, fraction) => sim.government.findStackOfBondsByID(id).amountOfBondsInStack }.sum
    case a:CentralBank  => a.listOfBonds.map{ case(id, fraction) => sim.government.findStackOfBondsByID(id).amountOfBondsInStack }.sum
    case a:BrokerDealer => a.listOfBonds.map{ case(id, fraction) => sim.government.findStackOfBondsByID(id).amountOfBondsInStack }.sum
  } else 0
  val newStackOfBonds = stackOfBonds(amountOfBonds, t)
  _govLOB += newStackOfBonds
  agent match {
    case agent:Bank =>
      agent.listOfBonds += newStackOfBonds.id -> 1.0
      addPublicDebt4Repayment(agent, newStackOfBonds)
    case agent:BrokerDealer =>
    agent.listOfBonds += newStackOfBonds.id -> 1.0
    addPublicDebt4Repayment(agent, newStackOfBonds)
    case agent:CentralBank =>
      agent.listOfBonds += newStackOfBonds.id -> 1.0
      addPublicDebt4Repayment(agent, newStackOfBonds)
    case _ => sys.error("agents must be either Bank or CB")
  }// match
  if(sim.test){
    agent match {
      case a:Bank =>
        val currentAmountOfIDs = a.listOfBonds.map{ case(id, fraction) => sim.government.findStackOfBondsByID(id).amountOfBondsInStack }.sum
        require( currentAmountOfIDs == priorAmountOfIDs + amountOfBonds, s"createBondRelationship failed, wrong amountOfIDs for $a: $currentAmountOfIDs != $priorAmountOfIDs + $amountOfBonds")
        testBankBondPayments(a, t, test)
      case a:CentralBank =>
        val currentAmountOfIDs = a.listOfBonds.map{ case(id, fraction) => sim.government.findStackOfBondsByID(id).amountOfBondsInStack }.sum
```

```
        require( currentAmountOfIDs == priorAmountOfIDs + amountOfBonds, s"createBondRelationship failed, wrong amountOdIDs for $a: $currentAmountOfIDs != $priorAmountOfIDs + $amountOfBonds")
        testCBBondPayments(t, test)
      }
    }
    agent match {
      case agent:Bank         => transferMoney(agent, this,   roundUpXk(amount, sim.faceValueOfBonds), cause, sim, t)
      case agent:CentralBank  => transferMoney(agent, this,   roundUpXk(amount, sim.faceValueOfBonds), cause, sim, t)
      case agent:BrokerDealer => transferMoney(agent, this, roundDownXk(amount, sim.faceValueOfBonds), cause, sim, t)
    }
   }, "Gov_createBondRelationship", sim)
  }




  def amountOfBondPayments (s:Map[Long, Double], t:Int, str:String, includeCurrentTick:Boolean = true) = {

    includeCurrentTick match {
      case true =>
        s.map{
          case(id:Long, fraction:Double) =>
            val SoB = findStackOfBondsByID(id)
            if(str == "c"){
              (SoB.bond.ticksOfCouponPayment.filter(_ >= t).size * SoB.bond.coupon * SoB.amountOfBondsInStack) * fraction
            } else if(str == "FV"){
              (SoB.bond.faceValue * SoB.amountOfBondsInStack) * fraction
            } else {
              sys.error(s"str is wrong")
            }
        }.sum

      case false =>
        s.map{
          case(id:Long, fraction:Double) =>
            val SoB = findStackOfBondsByID(id)
            if(str == "c"){
              (SoB.bond.ticksOfCouponPayment.filter(_ > t).size * SoB.bond.coupon * SoB.amountOfBondsInStack) * fraction
            } else if(str == "FV"){
              (SoB.bond.faceValue * SoB.amountOfBondsInStack) * fraction
            } else {
              sys.error(s"str is wrong")
            }
        }.sum
    }
  }




  /**
   *
```

```scala
 *  Method for testing of financial claims between the government and traditional banks related to government bonds.
 *
 *    */
def testBankBondPayments (bank:Bank, t:Int, includeCurrentTick:Boolean = true) = {
  val couponBank = rounded(
      amountOfBondPayments(bank.listOfBonds, t, "c", includeCurrentTick) +
      amountOfBondPayments(bank.bondsPledgedAsCollateralForOMO, t, "c", includeCurrentTick) +
      amountOfBondPayments(bank.bondsPledgedAsCollateralForOSLF, t, "c", includeCurrentTick) +
      amountOfBondPayments(bank.bondsPledgedAsCollateralForIDL, t, "c", includeCurrentTick)
      )
  val couponGov  = if(includeCurrentTick){
    rounded(_coupon2Pay.filterKeys(_ >= t).filter(_._2.contains(bank)).map(_._2(bank)).sum)
  } else {
    rounded(_coupon2Pay.filterKeys(_ > t).filter(_._2.contains(bank)).map(_._2(bank)).sum)
  }
  require( couponBank == couponGov, s"Amount of coupon2Pay to $bank is not correct: $couponBank / $couponGov")
  val FVbank = rounded(
      amountOfBondPayments(bank.listOfBonds, t, "FV", includeCurrentTick) +
      amountOfBondPayments(bank.bondsPledgedAsCollateralForOMO, t, "FV", includeCurrentTick) +
      amountOfBondPayments(bank.bondsPledgedAsCollateralForOSLF, t, "FV", includeCurrentTick) +
      amountOfBondPayments(bank.bondsPledgedAsCollateralForIDL, t, "FV", includeCurrentTick)
      )
  val FVgov  = if(includeCurrentTick){
    rounded(_dueDebt.filterKeys(_ >= t).filter(_._2.contains(bank)).map(_._2(bank)).sum)
  } else {
    rounded(_dueDebt.filterKeys(_ > t).filter(_._2.contains(bank)).map(_._2(bank)).sum)
  }
  require( FVbank == FVgov, s"Amount of dueDebt to repay to $bank is not correct: $FVbank / $FVgov")
}



/**
 *
 *  Method for testing of financial claims between the government and the CB related to government bonds.
 *
 *    */
def testCBBondPayments (t:Int, includeCurrentTick:Boolean = true) = {
  val couponCB  = rounded( amountOfBondPayments(CB.listOfBonds, t, "c", includeCurrentTick) )
  val couponGov = if(includeCurrentTick) rounded(_coupon2PayCB.filterKeys(_ >= t).values.sum) else rounded(_coupon2PayCB.filterKeys(_ > t).values.sum)
  require( couponCB == couponGov, s"Amount of coupon2Pay to $CB is not correct: $couponCB / $couponGov")
  val FVbank = rounded( amountOfBondPayments(CB.listOfBonds, t, "FV", includeCurrentTick) )
  val FVgov  = if(includeCurrentTick) rounded( _dueDebtCB.filterKeys(_ >= t).values.sum ) else rounded( _dueDebtCB.filterKeys(_ > t).values.sum )
  require( FVbank == FVgov, s"Amount of dueDebt to repay to $CB is not correct: $FVbank / $FVgov")
}





/**
 *
 *  Method searches for a specific bond using its individual ID. It is useful to ensure stock flow consistency.
```

```scala
   *
   *    */
  def findStackOfBondsIndexByID (ID:Long):Int = _govLOB.indexWhere(_.id == ID)





  /**
   *
   *  Method searches for a specific stackOfBonds using its individual ID. It is useful to ensure stock flow consistency.
   *
   *    */
  def findStackOfBondsByID (ID:Long) = {
    if(sim.test) require(_govLOB.map(_.ID).contains(ID), s"bond ID not found")
    if(sim.test) require(_govLOB.filter(_.ID == ID).size == 1, s"ID of bond is not unique $ID / ${_govLOB.filter(_.ID == ID)}.")
    _govLOB.find(_.id == ID) match {
      case Some(sob) => if(sim.test) {if(sob.id == ID) sob else sys.error("findBondByID is not correct.")} else sob
      case None      =>
        sim.BrokerDealerList.foreach {
          BD =>
            if(BD.listOfBonds.contains(ID)) println(s"Missing ID $ID exists in listOfBonds of $BD: ${BD.listOfBonds}")
            if(BD.bondsPledgedAsCollateralForRepo.contains(ID)) println(s"Missing ID $ID exists in bondsPledgedAsCollateralForRepo of $BD (${BD.active}/age=${BD.age}): ${BD.bondsPledgedAsCollateralForRepo}")
        }
        sim.MMMFList.foreach {
          mmmf =>
            if(mmmf.listOfBonds.contains(ID)) println(s"Missing ID $ID exists in listOfBonds of $mmmf: ${mmmf.listOfBonds}")
            if(mmmf.bondsPledgedAsCollateralForRepo.contains(ID)) println(s"Missing ID $ID exists in bondsPledgedAsCollateralForRepo of $mmmf (${mmmf.active}/age=${mmmf.age}): ${mmmf.bondsPledgedAsCollateralForRepo}")
        }
        sim.bankList.foreach {
          bank =>
            if(bank.listOfBonds.contains(ID)) println(s"Missing ID $ID exists in listOfBonds of $bank: ${bank.listOfBonds}")
            if(bank.bondsPledgedAsCollateralForRepo.contains(ID)) println(s"Missing ID $ID exists in bondsPledgedAsCollateralForRepo of $bank(${bank.active}/age=${bank.age}): ${bank.bondsPledgedAsCollateralForRepo}")
        }
        sys.error(s"ID $ID of bond does not exist in _govLOB")
    }
  }






  /**
   *
   *
   *    */
  def issueNewGovBonds   (bank:Bank,         amount:Double, t:Int, test:Boolean = true) = time({createBondRelationship(bank, amount, "buyGovBonds", t, test)}, "Gov_issueNewGovBonds", sim)
  def issueNewGovBondsBD (  BD:BrokerDealer, amount:Double, t:Int, test:Boolean = true) = time({createBondRelationship(BD  , amount, "buyGovBonds", t, test)},
```

```
"Gov_issueNewGovBonds", sim)





 /**
  *
  *  The government pays the yearly coupon on the outstanding government bonds. It also repays the face value at maturity.
  *
  *    */
  def payCoupon (t:Int) {time({

    if(sim.testSB){
      sim.bankList.filter(_.active).foreach(_.checkExistenceOfIDs("BEFORE", "start of payCoupon"))
      sim.BrokerDealerList.filter(_.active).foreach(_.checkExistenceOfIDs("BEFORE", "start of payCoupon"))
      sim.testAmountOfOutstandingBonds(t)
    }
    val initialPV = if(sim.test) sim.bankList.filter(_.active).map(_.currentPVofSoBs(t)) else Seq[Double]()



    def testPVOfBonds = {
      if(sim.pln){
        println(s"_govLOB: ${_govLOB.map(_.amountOfBondsInStack ).sum}; PV of single bond: ${this.PVofSoB(sim.government._govLOB.head, t)} / ${sim.government._govLOB.head.amountOfBondsInStack}")
        sim.bankList.foreach{
          bank =>
            println(s"$bank LOB:  " + bank.listOfBonds.map{ case(id, fraction) => findStackOfBondsByID(id).amountOfBondsInStack * fraction}.sum)
            println(s"$bank OMO:  " + bank.bondsPledgedAsCollateralForOMO.map{ case(id, fraction) => findStackOfBondsByID(id).amountOfBondsInStack * fraction}.sum)
            println(s"$bank OSLF: " + bank.bondsPledgedAsCollateralForOSLF.map{case(id, fraction) => findStackOfBondsByID(id).amountOfBondsInStack * fraction}.sum)
            println(s"$bank IDL:  " + bank.bondsPledgedAsCollateralForIDL.map{ case(id, fraction) => findStackOfBondsByID(id).amountOfBondsInStack * fraction}.sum)
            println(s"Sum for $bank: ${
              bank.listOfBonds.map{
                case(id, fraction) =>
                  findStackOfBondsByID(id).amountOfBondsInStack * fraction
              }.sum +
              bank.bondsPledgedAsCollateralForOMO.map{
                case(id, fraction) =>
                  findStackOfBondsByID(id).amountOfBondsInStack * fraction
              }.sum +
              bank.bondsPledgedAsCollateralForOSLF.map{
                case(id, fraction) =>
                  findStackOfBondsByID(id).amountOfBondsInStack * fraction
              }.sum +
              bank.bondsPledgedAsCollateralForIDL.map{
                case(id, fraction) =>
                  findStackOfBondsByID(id).amountOfBondsInStack * fraction
              }.sum}"
            )
        }
        println(s"CB LOB: ${CB.listOfBonds.map{ case(id, fraction) => findStackOfBondsByID(id).amountOfBondsInStack * fraction}.sum}")
      }
```

```scala
      val govSide  = rounded(PVofOutstandingBonds(t))
      val bankSide = rounded(         sim.bankList.filter(_.active).map(bank => bank.currentPVofSoBs(t)).sum + CB.currentPVofSoBs(t) )
      val BDSide   = rounded( sim.BrokerDealerList.filter(_.active).map(  bd =>   bd.currentPVofSoBsBD(t)).sum)
      require(SEc(govSide, bankSide + BDSide, 5), s"Gov: $govSide != Bank/CB/BD: $bankSide/$BDSide; difference is G-B/CB: ${rounded(govSide-bankSide-BDSide)}" )
    }



    if(sim.test) testPVOfBonds
    tradBanks match {

      case true =>
        val thereAreCoupons2Pay      = if(_coupon2Pay.contains(t)) true else false
        val thereAreFaceValues2Repay = if(_dueDebt.contains(t))    true else false
        if(thereAreCoupons2Pay || thereAreFaceValues2Repay){
          // Coupon of bonds hold by trad. Banks
          for(bank <- sim.bankList.filter(_.active)) {
            if(sim.pln)  println(s"Gov starts to payCoupon for the bonds of $bank")
            if(sim.test) require(!bank.listOfBonds.contains(null), s"listOfBonds of $bank contains null.")

            val PVofBondsPledgedAsCollateralBeforeCouponPay:Double = if(sim.test) bank.PV_OMO(t) + bank.PV_OSLF(t) + bank.PV_IDL(t) else 0.0
            val initialPVofOMO  = bank.PV_OMO(t)
            val initialPVofOSLF = bank.PV_OSLF(t)
            val initialPVofIDL  = bank.PV_IDL(t)

            // pay all coupons & faceValues
            if(sim.pln && thereAreCoupons2Pay && _coupon2Pay(t).contains(bank)){
              println(s"_coupon2Pay of $t: ${_coupon2Pay(t)}")
              println(s"_bondsAddedWithBondRelationship of $bank: ${bank.bondsAddedWithBondRelationship}")
              val couponBank = rounded(
                  amountOfBondPayments(bank.listOfBonds, t, "c") +
                  amountOfBondPayments(bank.bondsPledgedAsCollateralForOMO, t, "c") +
                  amountOfBondPayments(bank.bondsPledgedAsCollateralForOSLF, t, "c") +
                  amountOfBondPayments(bank.bondsPledgedAsCollateralForIDL, t, "c")
                  )
              val couponGov  = rounded(_coupon2Pay.filterKeys(_ >= t).filter(_._2.contains(bank)).map(_._2(bank)).sum)
              println(s"Aggregate amount of coupon2Pay to $bank BEFORE paying coupon of $t is: ${couponBank} / ${couponGov}")
              println(s"Gov is gonna pay coupons in $t to $bank of ${_coupon2Pay(t)(bank)}")
            }
            if(sim.test) bank.checkExistenceOfIDs("BEFORE", "payment of Coupons")
            if(thereAreCoupons2Pay      && _coupon2Pay(t).contains(bank)){
              if(sim.pln){
                println(s"_coupon2Pay of $t: ${_coupon2Pay(t)}")
                bank.printBSP
                if(bank.govDeposits.last < _coupon2Pay(t)(bank)) println(s"Gov has not enough deposits at $bank to pay the coupons of ${_coupon2Pay(t)(bank)}")
              }
              transferMoney(this, bank, _coupon2Pay(t)(bank), "payCoupon", sim, t)
              if(sim.pln) println(s"coupon payment to $bank done...")
            }
            if(sim.pln && thereAreCoupons2Pay && _coupon2Pay(t).contains(bank)){
              val couponBank = rounded(
                  amountOfBondPayments(bank.listOfBonds, t, "c") +
                  amountOfBondPayments(bank.bondsPledgedAsCollateralForOMO, t, "c") +
```

```
      amountOfBondPayments(bank.bondsPledgedAsCollateralForOSLF, t, "c") +
      amountOfBondPayments(bank.bondsPledgedAsCollateralForIDL, t, "c")
      )
  val couponGov  = rounded(_coupon2Pay.filterKeys(_ >= t).filter(_._2.contains(bank)).map(_._2(bank)).sum)
  println(s"Aggregate amount of coupon2Pay to $bank AFTER paying coupon of $t is: ${couponBank} / ${couponGov}")
}
if(sim.pln && thereAreFaceValues2Repay && _dueDebt(t).contains(bank)) println(s"Gov has paid coupons to $bank and is gonna repay FV of ${_dueDebt(t)(bank)}")

if(thereAreFaceValues2Repay &&    _dueDebt(t).contains(bank)){
  if(sim.pln){
    println(s"_dueDebt of $t: ${_dueDebt(t)}")
    bank.printBSP
    if(bank.govDeposits.last < _dueDebt(t)(bank)) println(s"Gov has not enough deposits at $bank to pay the coupons of ${_dueDebt(t)(bank)}")
    val FVbank = rounded(
        amountOfBondPayments(bank.listOfBonds, t, "FV") +
        amountOfBondPayments(bank.bondsPledgedAsCollateralForOMO, t, "FV") +
        amountOfBondPayments(bank.bondsPledgedAsCollateralForOSLF, t, "FV") +
        amountOfBondPayments(bank.bondsPledgedAsCollateralForIDL, t, "FV")
        )
    val FVgov  = rounded( _dueDebt.filterKeys(_ >= t).filter(_._2.contains(bank)).map(_._2(bank)).sum )
    println(s"Aggregate amount of dueDebt to repay to $bank BEFORE repay of FV due in $t: ${FVbank} / ${FVgov}")
    println(s"$bank has ${
      bank.listOfBonds.size +
      bank.bondsPledgedAsCollateralForOMO.size +
      bank.bondsPledgedAsCollateralForOSLF.size +
      bank.bondsPledgedAsCollateralForIDL.size} bonds (BEFORE repayment of due debt). ${_dueDebt(t)(bank)/1000} are due.")
  }
  transferMoney(this, bank, _dueDebt(t)(bank), "repayDuePublicDebt1", sim, t)
  if(sim.test) bank.checkExistenceOfIDs("BEFORE", s"removing ID from $bank")
  bank.listOfBonds                     --= bank.listOfBonds.filterKeys(          id => findStackOfBondsByID(id).bond.maturity == t).keys
  bank.bondsPledgedAsCollateralForOMO  --= bank.bondsPledgedAsCollateralForOMO.filterKeys( id => findStackOfBondsByID(id).bond.maturity == t).keys
  bank.bondsPledgedAsCollateralForOSLF --= bank.bondsPledgedAsCollateralForOSLF.filterKeys(id => findStackOfBondsByID(id).bond.maturity == t).keys
  bank.bondsPledgedAsCollateralForIDL  --= bank.bondsPledgedAsCollateralForIDL.filterKeys( id => findStackOfBondsByID(id).bond.maturity == t).keys
  if(sim.test) bank.checkExistenceOfIDs("AFTER", s"removing ID from $bank")
  if(sim.pln){
    val FVbank = rounded(
        amountOfBondPayments(bank.listOfBonds, t, "FV") +
        amountOfBondPayments(bank.bondsPledgedAsCollateralForOMO, t, "FV") +
        amountOfBondPayments(bank.bondsPledgedAsCollateralForOSLF, t, "FV") +
        amountOfBondPayments(bank.bondsPledgedAsCollateralForIDL, t, "FV")
      )
    val FVgov  = rounded( _dueDebt.filterKeys(_ >= t).filter(_._2.contains(bank)).map(_._2(bank)).sum )
    println(s"Aggregate amount of dueDebt to repay to $bank AFTER repay of FV due in $t: ${FVbank} / ${FVgov}")
    println(s"$bank has now ${
      bank.listOfBonds.size +
      bank.bondsPledgedAsCollateralForOMO.size +
      bank.bondsPledgedAsCollateralForOSLF.size +
      bank.bondsPledgedAsCollateralForIDL.size} bonds (AFTER repayment of due debt).")
  }

  if(sim.test){
    bank.listOfBonds.foreach{
      case(id, fraction) =>
```

Government.scala

```scala
                    if(t >= findStackOfBondsByID(id).bond.maturity) sys.error(s"maturity of bond in listOfBonds of $bank is already over-due: ${findStackOfBondsByID(id).bond.maturity} / $t")}
                bank.bondsPledgedAsCollateralForOMO.foreach{
                  case(id, fraction) =>
                    if(t >= findStackOfBondsByID(id).bond.maturity) sys.error(s"maturity of bond in listOfBonds of $bank is already over-due: ${findStackOfBondsByID(id).bond.maturity} / $t")}
                bank.bondsPledgedAsCollateralForOSLF.foreach{
                  case(id, fraction) =>
                    if(t >= findStackOfBondsByID(id).bond.maturity) sys.error(s"maturity of bond in listOfBonds of $bank is already over-due: ${findStackOfBondsByID(id).bond.maturity} / $t")}
                bank.bondsPledgedAsCollateralForIDL.foreach{
                  case(id, fraction) =>
                    if(t >= findStackOfBondsByID(id).bond.maturity) sys.error(s"maturity of bond in listOfBonds of $bank is already over-due: ${findStackOfBondsByID(id).bond.maturity} / $t")}
              }
              if(sim.pln) println(s"restoring due bonds pledged as collateral for OMO ($bank)")
              val amount2Restore_OMO = initialPVofOMO - bank.PV_OMO(t)
              if(amount2Restore_OMO > 0){
                try{
                  bank.pledgeCollateral(bank.bondsPledgedAsCollateralForOMO, amount2Restore_OMO, t)
                } catch {
                  case e:NullPointerException =>
                    println(s"$e: initialPVofOMO: $initialPVofOMO / amount2Restore: $amount2Restore_OMO /  OMO of $bank: ${bank.bondsPledgedAsCollateralForOMO} / LOB: ${bank.listOfBonds} / active: ${bank.active} / age: ${bank.age}") }
              }
              if(sim.pln) println(s"restoring due bonds pledged as collateral for OSLF ($bank)")
              val amount2Restore_OSLF = initialPVofOSLF - bank.PV_OSLF(t)
              if(amount2Restore_OSLF > 0) bank.pledgeCollateral(bank.bondsPledgedAsCollateralForOSLF, amount2Restore_OSLF, t)
              if(sim.pln) println(s"restoring due bonds pledged as collateral for IDL ($bank)")
              val amount2Restore_IDL = initialPVofIDL - bank.PV_IDL(t)
              if(amount2Restore_IDL > 0) bank.pledgeCollateral(bank.bondsPledgedAsCollateralForIDL, amount2Restore_IDL, t)
              if(sim.test){
                require(bank.PV_OMO(t)  >= initialPVofOMO,  s"restorePVofCollateral (OMO) was not succesful: now ${bank.PV_OMO(t)} / before $initialPVofOMO")
                require(bank.PV_OSLF(t) >= initialPVofOSLF, s"restorePVofCollateral (OSLF) was not succesful: now ${bank.PV_OSLF(t)} / before $initialPVofOSLF")
                require(bank.PV_IDL(t)  >= initialPVofIDL,  s"restorePVofCollateral (IDL) was not succesful: now ${bank.PV_IDL(t)} / before $initialPVofIDL")
                require(
                    PVofBondsPledgedAsCollateralBeforeCouponPay <= bank.PV_OMO(t) + bank.PV_OSLF(t) + bank.PV_IDL(t),
                    s"Transfer of due collateral of $this is wrong: Before $PVofBondsPledgedAsCollateralBeforeCouponPay / After ${bank.PV_OMO(t) + bank.PV_OSLF(t) + bank.PV_IDL(t)}"
                    )
              }
              if(sim.pln) println(s"FV payment to $bank done...")
            }// if dueDebt
          }// foreach
        }
        if(thereAreCoupons2Pay)      _coupon2Pay -= t
        if(thereAreFaceValues2Repay) _dueDebt    -= t


        // +++++++++++++++++++++++++ BrokerDealer +++++++++++++++++++++++++
        val thereAreCoupons2PayBD      = if(_coupon2PayBD.contains(t)) true else false
        val thereAreFaceValues2RepayBD = if(   _dueDebtBD.contains(t)) true else false
        if(thereAreCoupons2PayBD) println(s"thereAreCoupons2PayBD: ${_coupon2PayBD(t)}") else println(s"No couponPayments to BDs this in t=$t")
```

```
    if(thereAreFaceValues2RepayBD) println(s"thereAreFaceValues2RepayBD: ${_dueDebtBD(t)}") else println(s"No FV payments to BDs this in t=$t")
    if(thereAreCoupons2PayBD || thereAreFaceValues2RepayBD){
      // Coupon of bonds hold by BrokerDealer
      for(BD <- sim.BrokerDealerList.filter(_.active)) {
        println(s"++++++++++++++++++++++++ Gov starts to payCoupon for the bonds of $BD ++++++++++++++++++++++++")
        println(s"Due bonds of $BD: ${
          BD.listOfBonds.filterKeys(
              id =>
                findStackOfBondsByID(id).bond.maturity == t).keys
          } / ${BD.bondsPledgedAsCollateralForRepo.filterKeys(id => findStackOfBondsByID(id).bond.maturity == t).keys}")
        val PVofBondsPledgedAsCollateralBeforeCouponPayBD:Double = if(sim.test) BD.PV_Repo(t) else 0.0
        val initialPVofRepo = BD.PV_Repo(t)
        val bDbefore        = BD.bankDeposits.last

        // pay all coupons
        if(sim.testSB) BD.checkExistenceOfIDs("BEFORE", "payment of Coupons")
        if(thereAreCoupons2PayBD && _coupon2PayBD(t).contains(BD)){
          if(sim.pln){
            println(s"_coupon2PayBD of $t: ${_coupon2PayBD(t)}")
            if(BD.houseBank.govDeposits.last < _coupon2PayBD(t)(BD)) println(s"Gov has not enough deposits at $BD to pay the coupons of ${_coupon2PayBD(t)(BD)}")
          }
          transferMoney(this, BD, _coupon2PayBD(t)(BD), "payCoupon", sim, t)
          if(sim.pln) println(s"coupon payment to $BD done...")
        }

        // pay all faceValues
        if(thereAreFaceValues2RepayBD && _dueDebtBD(t).contains(BD)){
          if(sim.pln){
            println(s"_dueDebtBD of $t: ${_dueDebtBD(t)}")
            if(BD.houseBank.govDeposits.last < _dueDebtBD(t)(BD)) println(s"Gov has not enough deposits at $BD to pay the coupons of ${_dueDebtBD(t)(BD)}")
          }
          transferMoney(this, BD, _dueDebtBD(t)(BD), "repayDuePublicDebt1", sim, t)
          if(sim.testSB) BD.checkExistenceOfIDs("BEFORE", s"removing ID from $BD")
          // remove due SoBs
          BD.listOfBonds --= BD.listOfBonds.filterKeys(id => findStackOfBondsByID(id).bond.maturity == t).keys
          // welche ID -> fraction combi is past due?
          println(s"$BD --> PV of pledged bonds for repos which are due: ${
            BD.bondsPledgedAsCollateralForRepo.filterKeys{
              id =>
                findStackOfBondsByID(id).bond.maturity == t }.map{
                  case(id, fraction) =>
                    BD.PVofSoB(BD.sim.government.findStackOfBondsByID(id), t) * fraction }.sum }"
          )
          val IDsOfPastDueSoBs       = BD.bondsPledgedAsCollateralForRepo.filterKeys(id => findStackOfBondsByID(id).bond.maturity == t).keys.toList
          val mapRepo2IDofPastDueSoB = IDsOfPastDueSoBs.map{ ID => ID -> BD.outstandingRepos.toList.filter{ _.linkedBondIDs.contains(ID) } }.toMap
          val reposWithDueBondAsCollateral = mapRepo2IDofPastDueSoB.values.toList.flatten.toSet.toList
          val PVofPastDueCollateral        = reposWithDueBondAsCollateral.map {
            repo =>
              repo -> repo.linkedBondIDs.map {
                case (id, fraction) =>
                  if(IDsOfPastDueSoBs.contains(id)) BD.PVofSoB(BD.sim.government.findStackOfBondsByID(id), t) * fraction else 0.0
              }.sum
          }.toMap
```

```
          BD.outstandingRepos.foreach { repo => repo.linkedBondIDs.keys.foreach(id => if(IDsOfPastDueSoBs.contains(id)) repo.linkedBondIDs -= id) }
          BD.bondsPledgedAsCollateralForRepo --= BD.bondsPledgedAsCollateralForRepo.filterKeys(id => findStackOfBondsByID(id).bond.maturity == t).keys
          if(sim.testSB) BD.checkExistenceOfIDs("AFTER", s"removing ID from $BD")
          if(sim.testSB){
            BD.listOfBonds.foreach{
              case(id, fraction) =>
                if(t >= findStackOfBondsByID(id).bond.maturity){
                  sys.error(s"maturity of bond in listOfBonds of $BD is already over-due: ${findStackOfBondsByID(id).bond.maturity} / $t")
                }
            }
            BD.bondsPledgedAsCollateralForRepo.foreach{
              case(id, fraction) =>
                if(t >= findStackOfBondsByID(id).bond.maturity){
                  sys.error(s"maturity of bond in listOfBonds of $BD is already over-due: ${findStackOfBondsByID(id).bond.maturity} / $t")}
                }
          }
          // restore due bonds
          if(sim.pln) println(s"restoring due bonds pledged as collateral for Repo ($BD)")
          if(sim.testSB) require(bDbefore + _coupon2PayBD(t)(BD) + _dueDebtBD(t)(BD) == BD.bankDeposits.last, s"$bDbefore + ${_coupon2PayBD(t)(BD)} + ${_dueDebtBD(t)(BD)} 
== ${BD.bankDeposits.last}")
          if(PVofPastDueCollateral.nonEmpty){
            PVofPastDueCollateral.foreach{
              case (repoWithDueCollateral, amount2Restore) =>
                if(BD.active){
                  BD.pledgeCollateral(BD.bondsPledgedAsCollateralForRepo, amount2Restore, repoWithDueCollateral, t)
                }
            }
          }
          if(sim.testSB && BD.active){
            require(BD.PV_Repo(t) >= initialPVofRepo,  s"restorePVofCollateral (Repo) was not succesful: now ${BD.PV_Repo(t)} / before $initialPVofRepo")
            require(
                PVofBondsPledgedAsCollateralBeforeCouponPayBD <= BD.PV_Repo(t),
                s"Transfer of due collateral of $this is wrong: Before $PVofBondsPledgedAsCollateralBeforeCouponPayBD / After ${BD.PV_Repo(t)}")
          }
          if(sim.pln) println(s"FV payment to $BD done...")
        }// if dueDebt
      }// foreach
    }
    if(thereAreCoupons2PayBD)      _coupon2PayBD -= t
    if(thereAreFaceValues2RepayBD) _dueDebtBD    -= t



    // Coupon of bonds hold by the CB
    if(sim.pln) println(s"Now paying the coupon of the CB")
    val thereAreCoupons2PayCB      = if(_coupon2PayCB.contains(t)) true else false
    val thereAreFaceValues2RepayCB = if(_dueDebtCB.contains(t))    true else false
    if(sim.pln) if(thereAreCoupons2PayCB) println(s"_coupon2PayCB of $t: ${_coupon2PayCB(t)}") else println(s"No coupon payments to CB due...")
    if(sim.pln) if(thereAreFaceValues2RepayCB) println(s"_dueDebtCB of $t: ${_dueDebtCB(t)}") else println(s"No FV payments to CB due...")
    if(thereAreCoupons2PayCB) transferMoney(this, CB, _coupon2PayCB(t), "payCoupon", sim, t)
    if(sim.pln) println(s"coupon payment to CB done...")
    if(thereAreFaceValues2RepayCB){
      transferMoney(this, CB, _dueDebtCB(t), "repayDuePublicDebt1", sim, t)
```

```
        CB.listOfBonds --= CB.listOfBonds.keys.filter(ID => findStackOfBondsByID(ID).bond.maturity == t)
        if(sim.test){
          CB.listOfBonds.keys.foreach{
            ID =>
              if(t >= findStackOfBondsByID(ID).bond.maturity){
                sys.error(s"maturity of bond in listOfBonds of $CB is already over-due: ${findStackOfBondsByID(ID).bond.maturity} / $t")
              }
          }// foreach
        }
      }
      if(sim.pln) println(s"FV payment to CB done...")
      if(thereAreCoupons2PayCB)      _coupon2PayCB -= t
      if(thereAreFaceValues2RepayCB) _dueDebtCB    -= t



      if(sim.testSB) {
                sim.bankList.filter(_.active).foreach(_.checkExistenceOfIDs("BEFORE", "removing bonds from _govLOB"))
        sim.BrokerDealerList.filter(_.active).foreach(_.checkExistenceOfIDs("BEFORE", "removing bonds from _govLOB"))
      }
      println(s"IDs of due SoBs in $t (removed from gov_LOB): ${_govLOB.filter( SoB => SoB.bond.maturity == t).map(_.id)}")
      _govLOB --= _govLOB.filter( SoB => SoB.bond.maturity == t)
      if(sim.testSB) _govLOB.foreach(SoB => require( SoB.bond.maturity > t ) )
      if(sim.testSB) {
        sim.BrokerDealerList.foreach{
          BD =>
            BD.listOfBonds.keys.foreach{
              id =>
                require(!_govLOB.filter( SoB => SoB.bond.maturity == t).map(_.id).contains(id), s"listOfBonds of $BD contains ID $id of due bond in t=$t: ${BD.listOfBonds}")
            }
            BD.bondsPledgedAsCollateralForRepo.keys.foreach{
              id =>
                require(!_govLOB.filter( SoB => SoB.bond.maturity == t).map(_.id).contains(id), s"bondsPledgedAsCollateralForRepo of $BD contains ID $id of due bond in t=$t: ${BD.bondsPledgedAsCollateralForRepo}")
            }
        }
      }
      if(sim.testSB) {
                sim.bankList.filter(_.active).foreach(_.checkExistenceOfIDs("BEFORE", "removing bonds from _govLOB"))
        sim.BrokerDealerList.filter(_.active).foreach(_.checkExistenceOfIDs("BEFORE", "removing bonds from _govLOB"))
      }

      if(sim.test){
        sim.bankList.filter(_.active).foreach(_.checkExistenceOfIDs("AFTER", "removing bonds from _govLOB"))
        testPVOfBonds
        sim.bankList.foreach(bank => testBankBondPayments(bank, t, false))
        testCBBondPayments(t, false)
      }


    case false =>
      sim.hhList.foreach{
```

```
          hh =>
            val IDsOfDueCollateral_HH = ArrayBuffer[Long]()
            hh.listOfBonds.foreach{
              ID =>
                val bond = findStackOfBondsByID(ID)
                if(bond.ticksOfCouponPayment.contains(t)) transferMoney(this, hh, bond.coupon, "payCoupon", sim, t)
                if(bond.maturity == t){
                  transferMoney(this, hh, bond.faceValue, "repayDuePublicDebt0", sim, t)
                  removeBondFromGovLOB(ID)
                  IDsOfDueCollateral_HH += ID
                }
            }// foreach
            IDsOfDueCollateral_HH.foreach(ID => hh.listOfBonds -= ID)
            hh.listOfBonds.foreach{ID => if(t >= findStackOfBondsByID(ID).maturity) sys.error(s"maturity of bond in listOfBonds of $hh is already over-due: ${findStackOfBondsByID(ID).maturity} / $t")}
        }// foreach
    }// match
   }, "Gov_payCoupon", sim)
  }






 /**
  *
  *   If a payment by the government must be made to a real sector agents that is a customer of bank at which the government has no sufficient amount of deposits,
  *   the government can try to transfer the missing amount from another bank account. If that is not possible, it issues new debt.
  *
  *    */
  def getGovDeposits (bankWithInsufficientDeposits:Bank, amount:Double, t:Int, includeCurrentTick:Boolean = true) {time({
    val banksWithSufficientGovDeposits = sim.bankList.filter(bank => bank.active && bank.govDeposits.last > amount)
    if(t > 1 && banksWithSufficientGovDeposits.nonEmpty){
      val peerWithGovDeposits = banksWithSufficientGovDeposits.maxBy(_.govDeposits.last)
      if(sim.pln) println(s"$bankWithInsufficientDeposits gets govDeposits of $amount from $peerWithGovDeposits (gD: ${peerWithGovDeposits.govDeposits.last} / cbR: ${peerWithGovDeposits.cbReserves.last})")
      transferMoney(peerWithGovDeposits, bankWithInsufficientDeposits, amount, "transferGovDeposits", sim, t)
    } else issueNewGovBonds(bankWithInsufficientDeposits, amount, t, includeCurrentTick)
   }, "Gov_getGovDeposits", sim)
  }





 /**
  *
  *    */
 def GovDeficitLimit (NBCparameter:Double = nbcParameter) = {
   _NBC = NBCparameter * _realGDP.grouped(48).toList.map(_.sum).last
 }

```

```scala




  /**
   *
   *  HH are burdened with an income tax according to the german tax rates.
   *
   *    */
  def incomeTax (wage:Double) = {
    0.3
//    wage match {
//        case wage:Double if wage <      0 => sys.error("income tax on negative wage is not possible")
//        case wage:Double if wage <=   800 => 0.0
//        case wage:Double if wage <=  1125 => 912.17 * math.pow(10,-8) * math.pow(wage -  800,2) + 0.14   * (wage -  800)                                   //
//        case wage:Double if wage <=  4400 => 228.74 * math.pow(10,-8) * math.pow(wage - 1125,2) + 0.2397 * (wage - 1125) +   86.5                           //
//        case wage:Double if wage <= 20900 =>                                                       0.42   *  wage         -  681                            //
//        case _                            =>                                                       0.45   *  wage         - 1308                            //
//      }
    }





  /**
   *
   *   From time to time, the government decides to spend some money to stimulate the economic activity (keynesian policy tool).
   *
   *    */
    def governmentSpending (t:Int, cause:String = if(tradBanks) "govConsumption1" else "govConsumption0") {time({
      var amountToSpend = if(tradBanks) _bankDeposits.last else _cash.last
      if(amountToSpend > 0){
        val unemploymentRate = (sim.numberOfHH - sim.firmList.map(_.numberOfEmployees.last).sum) / sim.numberOfHH.toDouble
        if(unemploymentRate > 0.1 && t > 100){
          sim.random.shuffle(sim.firmList).filter(firm => goodsMarket.currentOffers(firm).quantity > 0).foreach{firm =>
            if(amountToSpend > 0){
              val affordableQuantity = math.min( unemploymentRate * goodsMarket.currentOffers(firm).quantity, amountToSpend / goodsMarket.currentOffers(firm).price )
              if(affordableQuantity >= 0.1){
                if(sim.pln) {
                  println(
                        this + " has " + amountToSpend + " to spend and buys " + affordableQuantity + " / " + goodsMarket.currentOffers(firm).quantity + " at a price of " +
goodsMarket.currentOffers(firm).price + " from " + firm
                        )
                }
                deposit( firm.sales,       affordableQuantity, t, sim)
                withdraw(firm.amountOfInventory, affordableQuantity, t, sim)
                transferMoney(this, firm, affordableQuantity * goodsMarket.currentOffers(firm).price, cause, sim, t)
                amountToSpend = rounded( amountToSpend - affordableQuantity * goodsMarket.currentOffers(firm).price )
              }// if
            }// if
          }// foreach
      }// if
```

```scala
    }// if

  }, "Gov_governmentSpending", sim)}



  def avgLS = if(sim.laborSkillUpdateParameter > 0) average( sim.hhList.map { _.laborSkillFactor.last } ) else sim.avgInitialLS



  /**
   *
   *  The government updates the level of the unemployment benefit paid to unemployed HH according to the current price level of the goods bundle.
   *
   *    */
  def updateUnemploymentBenefit = time(unemploymentBenefit += math.max(1000, 4 * avgLS * math.exp( 0.012 / (48 / sim.updateFrequency) ) *
goodsMarket.weightedAvgPriceOfYear.last), "gov_updateUB", sim)









  /**
   *
   *  If a HH is unemployed, it receives unemployment benefit from the government until it finds a new job.
   *
   *    */
  def payUnemploymentBenefit2HH (t:Int = 1, cause:String = if(tradBanks) "unemploymentBenefit1" else "unemploymentBenefit0") {time({
    tradBanks match {

      case true =>
        sim.hhList.foreach{
          hh =>
            if(hh.currentEmployer == sim.arge){
              if(t == 1){
                transferMoney(this, hh, math.max(hh.laborSkillFactor.last * sim.initialWage * subsidyFraction * 20, unemploymentBenefit.last), cause, sim, t)
              } else {
                transferMoney(this, hh, math.max(hh.laborSkillFactor.last * unemploymentBenefit.last, unemploymentBenefit.last), cause, sim, t)
              }
            }// if
        }// foreach

      case false =>
        sim.hhList.foreach{
          hh =>
            if(hh.unemployed.last){
              if(hh.cash.last < (sim.random.nextInt(20) + 8) * goodsMarket.weightedAvgPriceOfYear.last){
                if(hh.periodsOfUnemployment.last >= 24){
                  transferMoney(this, hh, math.min(unemploymentBenefit.last, cash.last), cause, sim, t)
```

```
              } else transferMoney(this, hh, math.min(math.max(hh.laborSkillFactor.last * sim.initialWage * subsidyFraction, unemploymentBenefit.last), cash.last), cause, sim, t)
            }// if
          }// if
      }// foreach

    }// match
   }, "Gov_payUnemploymentBenefit2HH", sim)
  }

  /**
   *
   *  Since the government is a customer of traditional/commercial banks, it has to pay a small fee to use the bank account.
   *
   *    */
  def payBankAccountFee (t:Int) = if(_bankDeposits.last >= 100 * sim.numberOfBanks) sim.bankList.filter(_.active).foreach( bank => transferMoney(this, bank, 100, "payBankAccountFee", sim, t) )

  /**
   *
   *  Once a year, the government calculates both nominal and real GDP and stores it to produce an appropriate time series.
   *
   *    */
  def determineEconomicGrowth {
    // in nominal terms
    val annualGDP = _GDP.grouped(48).toList.map(_.sum).toBuffer
    if(sim.test) require(_GDP.grouped(48).toList.last.size == 48, "to determine nomEconGrowth, grouped GDP has to have 48 values but the last has not!")
    if(_GDP.grouped(48).toList.last.size != 48) annualGDP -= annualGDP.last
    for(i <- 0 until annualGDP.size-1) _nomEconGrowth     += rounded( ((annualGDP(i+1) - annualGDP(i)) / annualGDP(i)) * 100 )
    for(i <- 0 until annualGDP.size-1) _nomEconGrowthLog  += rounded( math.log(annualGDP(i+1)) - math.log(annualGDP(i)) )

    // in real terms
    val rGDP = _realGDP.grouped(48).toList.map(_.sum).toBuffer
    for(i <- 0 until rGDP.size-1) _realEconGrowth    += rounded( ((rGDP(i+1) - rGDP(i))   / rGDP(i))  * 100 )
    for(i <- 0 until rGDP.size-1) _realEconGrowthLog += rounded( math.log(rGDP(i+1)) - math.log(rGDP(i)) )
  }

  /**
   *
   *  In the case of a default of a systemically important bank (measured by current market share), the government is forced to bail out the institution in order to ensure the
   *  functioning of the payment system. Thus, the bad debt of the bank is partly taken by the government as well as by the equityholders of the financial institution.
   *
   *    */
  def bailOutLastBank (bank2BailOut:Bank, t:Int) {time({
```

```
  // 1.
  val E  = abs(bank2BailOut.equity.last)
  val gD = bank2BailOut.govDeposits.last
  val missingAmount = E-gD
  if(missingAmount > 0) issueNewGovBonds(bank2BailOut, 1.5 * missingAmount, t)
  withdraw(_bankDeposits,            E, t, sim)
  withdraw(bank2BailOut.govDeposits, E, t, sim)
  deposit(          lossFromBailOut, E, t, sim)

  // 2. owner provide new equity (owner buy govBonds for the bank to increase assets)
  var amountOfNewIssuedBonds = 0
  bank2BailOut.owners.foreach{
      hh =>
        val newInvestment   = math.max(1, hh.bankDeposits.last * 0.5)
        val amountOfBonds   = (roundUpXk(newInvestment,sim.faceValueOfBonds)/sim.faceValueOfBonds - 1).toInt
        val newStackOfBonds = stackOfBonds(amountOfBonds, t)
        _govLOB += newStackOfBonds
        bank2BailOut.listOfBonds += newStackOfBonds.id -> 1.0
        addPublicDebt4Repayment(bank2BailOut, newStackOfBonds)
        amountOfNewIssuedBonds += amountOfBonds
      if(sim.test) testBankBondPayments(bank2BailOut, t, false)
      transferMoney(hh, hh.houseBank, roundUpXk(newInvestment,sim.faceValueOfBonds) - sim.faceValueOfBonds, "initialInvestmentB", sim, t)
    }
    bank2BailOut.updatePVofSoBs(t)
    val TA = rounded( Seq(bank2BailOut.businessLoans.last, bank2BailOut.interbankLoans.last, bank2BailOut.bonds.last, bank2BailOut.interestReceivables.last,
bank2BailOut.OSDF.last, bank2BailOut.cbReserves.last).sum )
    val TL = rounded( Seq(bank2BailOut.retailDeposits.last, bank2BailOut.govDeposits.last, bank2BailOut.cbLiabilities.last, bank2BailOut.interbankLiabilities.last).sum)
    if(sim.test) require(rounded( TA - TL ) >= 0, s"Bank in distress has not enough equity after bail out: (TA) $TA - (L) $TL = ${rounded( TA - TL )}")

    if(sim.pln) println(s"BS of $bank2BailOut after govBailOut and the issuing of $amountOfNewIssuedBonds newly issued bonds.")
    if(sim.pln) bank2BailOut.printBSP

 }, "Gov_bailOutLastBank", sim)
}




/**
 *
 *  At the end of each fiscal year, the government agent makes an annual report to update its balance sheets statements.
 *
 *    */
def makeAnnualReport (t:Int) {
  _numberOfExistingBonds += _govLOB.size
}




/**
 *
```

```scala
 *  These values are jsut for data saving purposes.
 *
 *    */
val governmentEndOfSimulationData = Map(
      "taxVAT"                  -> _VAT,                    // AB[Double]
      "VATrevenue"              -> _VATrevenue,             // AB[Double]
      "taxCorporate"            -> _corporateTax,           // AB[Double]
      "corporateTaxRevenue"     -> _corporateTaxRevenue,    // AB[Double]
      "taxDividends"            -> _capitalGainsTax,        // AB[Double]
      "capitalGainsTaxRevenue"  -> _capitalGainsTaxRevenue, // AB[Double]
      "incomeTaxRevenue"        -> _incomeTaxRevenue,       // AB[Double]
      "deficit"                 -> _deficit,                // AB[Double]
      "unemploymentBenefit"     -> _unemploymentBenefit,    // AB[Double]
      "benefitPayed"            -> _benefitPayed,           // AB{Double]
      "GDP"                     -> _GDP,                    // AB[Double]
      "realGDP"                 -> _realGDP,                // AB[Double]
      "GDPdeflator"             -> _GDPdeflator,            // AB[Double]
      "GDPdeflatorMP"           -> _GDPdeflatorMP,          // AB[Double]
      "govSpending"             -> _govSpending,            // AB[Double]
      "econGrowthNominal"       -> _nomEconGrowth,          // AB[Double]
      "econGrowthNominalLog"    -> _nomEconGrowthLog,       // AB[Double]
      "econGrowthReal"          -> _realEconGrowth,         // AB[Double]
      "econGrowthRealLog"       -> _realEconGrowthLog,      // AB[Double]
      "productionOfTick"        -> _productionOfTick,       // AB[Double]
      "M0"                      -> _M0,                     // AB[Double]
      "M1"                      -> _M1,                     // AB[Double]
      "M3"                      -> _M3,                     // AB[Double]
      "lossFromBailOut"         -> _lossFromBailOut,        // AB[Double]
      "wapOFyear"               -> wapOFyear,               // AB[Double]
      "numberOfExistingBonds"   -> _numberOfExistingBonds   // AB[Double]
  )


}// end of class government
```

A.6.2 Central Bank Class

centralBank.scala

```
/**
 *
 */

package monEcon.publicSector

import monEcon.Agent
import monEcon.financialSector.Bank
import monEcon.financialSector.BrokerDealer
import monEcon.Simulation
import monEcon.bonds
import monEcon.hpFilter
import collection.mutable._
import scala.sys.process._


/**
 * @author Sebastian Krug
 *
 */


// Central Bank-Class
case class CentralBank (initialTargetRate          :Double,
                        maxTargetRate              :Double,
                        minTargetRate              :Double,
                        initialLendingFacilityRate:Double,
                        initialDepositFacilityRate:Double,
                        initialReserveReq          :Double,
                        delta_pi                   :Double,
                        delta_x                    :Double,
                        delta_s                    :Double,
                        inflationTarget            :Double,
                        yearsOfInactiveMP          :Double,
                        years2TakeIntoAccountInTR  :Int,
                        sim                        :Simulation,
                        taylorRule                 :Boolean,
                        TRpathdependence           :Boolean,
                        CCycB                      :Boolean,
                        CFSItarget                 :Double,
                        creditToGDPratioinTR       :Boolean
                                                              ) extends Agent with bonds with hpFilter {

  val name = "centralBank"
  override def toString = s"$name"


  // Monetary Policy Rates
  private val _lendingFacilityRate  = ArrayBuffer[Double](initialLendingFacilityRate)   // OSLF or LFR
  private val _depositFacilityRate  = ArrayBuffer[Double](initialDepositFacilityRate)   // OSDF or DFR
  private val _RePoRate             = ArrayBuffer[Double](sim.initialTargetRate)        //
  private val _effectiveFFR         = ArrayBuffer[Double](initialTargetRate)            //
  private val _targetFFR            = ArrayBuffer[Double](initialTargetRate)            // set according to Taylor Rule
  private val _TR                   = ArrayBuffer[Double]()                             //
```

centralBank.scala

```scala
private val _outputGap           = ArrayBuffer[Double]()                            //

private val _inflationCPI        = ArrayBuffer[Double]()                            //
private val _inflationDeflator   = ArrayBuffer[Double]()                            //
private val _inflationDeflatorMP = ArrayBuffer[Double]()                            //






/*  ---------------------------------------  central bank balance sheet positions  ---------------------------------------  */
// Assets Side
private val _loans2CommercialBanks   = ArrayBuffer(0.0)
//  bonds                  = ArrayBuffer(0.0) //

// Liabilities Side
private val _reserves            = ArrayBuffer(0.0)   //
private val _OSDF                = ArrayBuffer(0.0)   //
private val _governmentsAccount  = ArrayBuffer(0.0)   //
private val equity               = ArrayBuffer(0.0)   //



/**
 *
 *  This is just to save balance sheet data.
 *
 *    */
val centralBankBSP = scala.collection.mutable.Map("loans2CommercialBanks" -> _loans2CommercialBanks,
                                                  "bonds"                 -> bonds,
                                                  "reserves"              -> _reserves,
                                                  "OSDF"                  -> _OSDF,
                                                  "governmentsAccount"    -> _governmentsAccount,
                                                  "equity"                -> equity                   )


// other data
private val _avgReserves                   = Map[Bank, ArrayBuffer[Double]]()                //
private val _deficitReserves               = Map[Bank, ArrayBuffer[Double]]()                //
private val _excessReserves                = Map[Bank, ArrayBuffer[Double]]()                //
private val _minReserveRequirement         = ArrayBuffer(initialReserveReq)                  //
private val _credit2privateSector          = ArrayBuffer[Double]()                           //
private val _credit2GDPratio               = ArrayBuffer[Double]()                           //
private val _credit2GDPratioTR             = ArrayBuffer[Double]()                           //
private val _credit2GDPtrend               = ArrayBuffer[Double]()                           //
private val _credit2GDPgap                 = ArrayBuffer[Double]()                           //
private val _credit2GDPgapTR               = ArrayBuffer[Double]()                           //
private val _outstandingPrivateSectorDebt  = ArrayBuffer[Double]()                           //
private val _CFSI                          = ArrayBuffer[Double]()                           //
private val _CFSIHP                        = ArrayBuffer[Double]()                           //
private val _CFSIgap                       = ArrayBuffer[Double]()                           //

private val _aggBankBailOuts               = ArrayBuffer[Double]()
```

```
private val _avgFirmDEratio               = ArrayBuffer[Double](1.0)
private val _aggBankInsolvencies          = ArrayBuffer[Double]()
private val _avgBankLR                    = ArrayBuffer[Double]()
private val _avgBankCCQ                   = ArrayBuffer[Double](1.0)
private val _numbOfActiveFirms            = ArrayBuffer[Double]()
private val _avgBankDEratio               = ArrayBuffer[Double](1.0)
private val _aggFirmInsolvencies          = ArrayBuffer[Double]()
private val _liquidityInsuranceDebtBD     = Map[BrokerDealer, Double]()



/**
 *
 *  The ReserveTarget class defines the 1%-range around the bank agents' reserve target in each maintenance period.
 *
 *    */
case class ReserveTarget (reserveTargetBalance:Double) {
  if(sim.test) require(reserveTargetBalance % 10000 <= reserveTargetBalance * 0.000000001, s"This reserve target has not the correct value: $reserveTargetBalance")
  val lowerBound = reserveTargetBalance * 0.99
  val upperBound = reserveTargetBalance * 1.01
}


private val _reserveTargetBalances      = Map[Bank, ReserveTarget]()            //
private val _outstandingOMOreceivabels  = Map[Bank, OMO]()                      //
private val _reservesLendOvernightOnIBM = Map[Int, ArrayBuffer[IBMloan]]()      //
private val _outstandingOSLFreceivables = Map[Bank, OvernightOSLFloan]()        //
private val _intraDayLiquidity          = Map[Bank, Double]()                   //



// getter
def lendingFacilityRate          = _lendingFacilityRate
def depositFacilityRate          = _depositFacilityRate

def inflationCPI                 = _inflationCPI
def inflationDeflator            = _inflationDeflator
def inflationDeflatorMP          = _inflationDeflatorMP
def loans2CommercialBanks        = _loans2CommercialBanks
def reserves                     = _reserves
def OSDF                         = _OSDF
def governmentsAccount           = _governmentsAccount
def avgReserves                  = _avgReserves
def deficitReserves              = _deficitReserves
def excessReserves               = _excessReserves
def minReserveRequirement        = _minReserveRequirement
def reserveTargetBalances        = _reserveTargetBalances
def intraDayLiquidity            = _intraDayLiquidity
def outstandingOSLFreceivables   = _outstandingOSLFreceivables
def outstandingOMOreceivabels    = _outstandingOMOreceivabels
def reservesLendOvernightOnIBM   = _reservesLendOvernightOnIBM
def RePoRate                     = _RePoRate
def effectiveFFR                 = _effectiveFFR
def targetFFR                    = _targetFFR
```

```scala
  def TR                           = _TR
  def outputGap                    = _outputGap
  def baseYear                     = _baseYear
  def currentOutstandingReserves   = sim.bankList.filter(_.active == true).map(_.cbReserves.last).sum
  def credit2privateSector         = _credit2privateSector
  def credit2GDPratio              = _credit2GDPratio
  def credit2GDPratioTR            = _credit2GDPratioTR
  def credit2GDPtrend              = _credit2GDPtrend
  def credit2GDPgap                = _credit2GDPgap
  def credit2GDPgapTR              = _credit2GDPgapTR
  def outstandingPrivateSectorDebt = _outstandingPrivateSectorDebt
  def CFSI                         = _CFSI
  def CFSIHP                       = _CFSIHP
  def CFSIgap                      = _CFSIgap
  def liquidityInsuranceDebtBD     = _liquidityInsuranceDebtBD


  // setter
  def discountRateCB_+= (value:Double):Unit = _discountRateCB += value
  def depositRateCB_+=  (value:Double):Unit = _depositRateCB  += value






  /*  ------------------------------------------------------------  Central Bank short-term interest rates for monetary policy  ------------------------------------------------------------  */


  /**
   *
   *    This method computes a composite financial stability measure or indicator.
   *    We do not incorporate GDPgrowth here, since the output gap concerning this is already captured in the standard TR.
   *    We do not incorporate inflation here, since a gap concerning this is already captured in the standard TR.
   *
   *    */
  def determineCFSI (t:Int, frequency:Int = 6) = {
    if(t >= frequency){
      val b                         = sim.firmList.filter(firm => firm.active && firm.equity.last > 0).map(firm => firm.debt2EquityRatio * firm.determineCurrentMarketShareCFSI).sum
      val avgDEratioOfFirmSector    = if(b > 1.0/5) math.log(5 * b) else math.max(0, _avgFirmDEratio.last)
      _avgFirmDEratio += avgDEratioOfFirmSector

      val g                         = sim.bankList.filter(bank => bank.active && bank.equity.last > 0).map(bank => math.min(1, bank._currentEquityOfRWA(t) * bank.determineCurrentMarketShareCFSI )).sum
      val avgBankCCQ                = if(1.0/g > 1) math.log(1/g) else math.max(0, _avgBankCCQ.last)
      _avgBankCCQ += avgBankCCQ

      val h                         = sim.bankList.filter(bank => bank.active && bank.equity.last > 0).map(bank => bank.debt2EquityRatio * bank.determineCurrentMarketShareCFSI).sum
      val avgDEratioOfBankSector    = if(h > 1) math.log(h) else math.max(0, _avgBankDEratio.last)
      _avgBankDEratio += avgDEratioOfBankSector

      _CFSI += avgDEratioOfBankSector + avgDEratioOfFirmSector

```

```
    if(sim.CFSIbackstop) _CFSIgap += math.max(_CFSI.last - CFSItarget, 0) / 100.0 else _CFSIgap += (_CFSI.last - CFSItarget) / 100.0
  }
}

/**
 *
 *  Since we are interested in the deviation from the long-term trend of the Credit-to-GDP ratio, we have to calculate
 *  the trend and the current ratio.
 *
 *    */
def determineCreditToGDPgap (t:Int, frequency:Int = 6) = {
  if(t >= frequency){
    val credit            =  _credit2privateSector.takeRight(frequency).sum
    val realGDP           = sim.government.realGDP.takeRight(frequency).sum
    _credit2GDPratioTR   += (credit / realGDP)
    val credit2GDPtrendTR = HPfilterData(_credit2GDPratioTR, 6.25)
    _credit2GDPgapTR     += 0.2 * (_credit2GDPratioTR.last - credit2GDPtrendTR.last)
  }
}

/**
 *
 *  Sets the target short-term nominal interest rate (FFR or base rate) according to a standard interest rate rule of the Taylor type [Taylor, J.B. (1993)].
 *  For potential output in the output gap (log of the HP-filtered trend in real GDP) is used.
 *
 *    */
private def setTargetRate (t:Int, realInterestRate:Double = 0.02, delta_pi:Double = delta_pi, delta_x:Double = delta_x, delta_s:Double = delta_s) {time({
  // every 6 ticks -> 8 meeting a year -> 48/8 = 6
  taylorRule match {

    case true =>
      if(_inflationDeflatorMP.nonEmpty && sim.government.realGDP.nonEmpty){
        if(creditToGDPratioinTR) determineCreditToGDPgap(t) else determineCFSI(t)
        val tickOfCurrentFiscalYear = t % 48
        val macroData =
          {
            val rGDPofMP = ArrayBuffer[Double]()
            rGDPofMP   ++= sim.government.realGDP.grouped(6).toList.map(_.sum)
            val rGDPHP   = HPfilterData(rGDPofMP, 6.25)
            _outputGap  += 0.25 * ( math.log(rGDPofMP.last) - math.log(rGDPHP.last) )
            (0.05 * _inflationDeflatorMP.last/100, _outputGap.last)
          }
        if(sim.test) require(macroData._2.size == macroData._3.size, s"realGDP data and HP filtered realGDP data have unequal size: ${macroData._2.size} / ${macroData._3.size}")
        val i = if(creditToGDPratioinTR){
```

```
            realInterestRate + 0.2 * sim.expPi.last + delta_pi * (macroData._1 - inflationTarget) + delta_x * ( macroData._2 ) + delta_s * _credit2GDPgapTR.last   // target rate according to TR
          } else {
            realInterestRate + 0.2 * sim.expPi.last + delta_pi * (macroData._1 - inflationTarget) + delta_x * ( macroData._2 ) + delta_s * _CFSIgap.last           // target rate according to TR
          }
          _TR += i
          val newTarget = TRpathdependence match {

            case true =>
              i match {
                case i:Double if i < -0.0100 => math.max( _targetFFR.last - 0.0100, minTargetRate ) // -100 bp
                case i:Double if i < -0.0075 => math.max( _targetFFR.last - 0.0075, minTargetRate ) // - 75 bp
                case i:Double if i < -0.0050 => math.max( _targetFFR.last - 0.0050, minTargetRate ) // - 50 bp
                case i:Double if i < -0.0025 => math.max( _targetFFR.last - 0.0025, minTargetRate ) // - 25 bp
                case i:Double if i >  0.0100 => math.min( _targetFFR.last + 0.0100, maxTargetRate ) // +100 bp
                case i:Double if i >  0.0075 => math.min( _targetFFR.last + 0.0075, maxTargetRate ) // + 75 bp
                case i:Double if i >  0.0050 => math.min( _targetFFR.last + 0.0050, maxTargetRate ) // + 50 bp
                case i:Double if i >  0.0025 => math.min( _targetFFR.last + 0.0025, maxTargetRate ) // + 25 bp
                case _                       =>           _targetFFR.last
              }

            case false =>
              i match {
                case i:Double if i * 100 % 1 < 0.125  => math.min( (i*100).toInt.toDouble/100,          maxTargetRate)
                case i:Double if i * 100 % 1 < 0.375  => math.min( (i*100).toInt.toDouble/100 + 0.0025, maxTargetRate)
                case i:Double if i * 100 % 1 < 0.625  => math.min( (i*100).toInt.toDouble/100 + 0.0050, maxTargetRate)
                case _                                => math.min( (i*100).toInt.toDouble/100 + 0.0075, maxTargetRate)
              }

          }// match
          _targetFFR += roundTo4Digits(newTarget)
          if(sim.pln){
            println( s"setting targetRate for ${t}+ (tickOfCurrentFiscalYear: $tickOfCurrentFiscalYear) to: $realInterestRate + $inflationTarget + ${delta_pi * (macroData._1/100 - inflationTarget)} + ${delta_x * ( macroData._2)} = $i -> $newTarget / ${_targetFFR.last}")
          }
        } else _targetFFR += _targetFFR.last


      case false => _targetFFR += _targetFFR.last

    }
  }, "CB_setTargetRate", sim)
 }





 /**
  *
  *   The effective money market rate (not the target rate!!):
  *   - represents a time-varying distribution of rates charged by lenders with excess reserves on borrowers with reserve deficit;
  *   - Afonso/Lagos (2014, p.13): "there is no such thing as the federal funds rate (FFR) --> there is the _targetFFR of the CB and the effective federal funds rate (eFFR)
```

```
  *     which is rather a time-varying distribution of rates depending on individual negotiations on the decentralized OTC-interbank market."
  *   - determine eFFR as a value/volume-weighted daily avg FFR from information of the interbank market (IBM)
  *   - CB receives data at end of the day and publishes the eFFR for the previous tick
  *
  *    */
def determineEffectiveFFR (t:Int) = {time({
  if(_reservesLendOvernightOnIBM.contains(t)){
    _effectiveFFR += roundTo3Digits( _reservesLendOvernightOnIBM(t).map(IBMloan => IBMloan.amountOfReserves * IBMloan.interest).sum / 
_reservesLendOvernightOnIBM(t).map(_.amountOfReserves).sum )
    if(t == 1) _effectiveFFR -= _effectiveFFR.head
  } else{
    _effectiveFFR += _effectiveFFR.last
    if(t == 1) _effectiveFFR -= _effectiveFFR.head
  }
  if(sim.test) require(_effectiveFFR.size == t, s"CB's _effectiveFFR has not the appropriate amount of values (unequal t): ${_effectiveFFR.size} / $t")
  determineRepoRates
 }, "CB_determineEffectiveFFR", sim)
}





/**
 *
 *  Sets Central Bank short-term interest rates according to current MP goals
 *
 *    */
def setCentralBankInterestRates (t:Int) {time({
  determineEffectiveFFR(t)
  if(t >= yearsOfInactiveMP * 48 && t % 48 % 6 == 0){
    setTargetRate(t)
    // after the calculation of the target rate change, the interest corridor for the standing facilities have to be adjusted
    _targetFFR.last match {
      case i:Double if(i <  0.03) =>    _lendingFacilityRate   +=          i + 0.0025
                                        _depositFacilityRate   += math.max(i - 0.0025,  0.0025)
      case i:Double if(i <= 0.05) =>    _lendingFacilityRate   +=          i + 0.0050
                                        _depositFacilityRate   +=          i - 0.0045
      case i:Double if(i >  0.05) =>    _lendingFacilityRate   +=          i + 0.0100
                                        _depositFacilityRate   +=          i - 0.0075
    }
  } else {
    _targetFFR              += _targetFFR.last
    _lendingFacilityRate    += _lendingFacilityRate.last
    _depositFacilityRate    += _depositFacilityRate.last
  }
 }, "CB_setCentralBankInterestRates", sim)
}
```

centralBank.scala

```scala

  /**
   *
   *    */
    def determineRepoRates = _RePoRate += _effectiveFFR.last



  /**
   *
   *   If bank i's reserves balances lie within target range on average -> pay CB rate on target, if not pay only standing deposit facility rate (CB rate - 1%)
   *
   *    */
  def payInterestOnReserves (t:Int) {time({
    sim.bankList.filter(_.active).foreach{
      bank =>
        val avgReserves = rounded( bank._currentAvgReserves )
        if(avgReserves > reserveTargetBalances(bank).lowerBound && avgReserves < reserveTargetBalances(bank).upperBound){
          if(sim.pln){
            println(s"$bank is within the reserve target range: ${reserveTargetBalances(bank).lowerBound} < $avgReserves (avg) < ${reserveTargetBalances(bank).upperBound} and receives interest on its reserve account of ${rounded(avgReserves * _targetFFR.last)}")
          }
          transferMoney(this, bank, avgReserves * _targetFFR.last, "payInterestOnReserves", sim, t)
        } else if(avgReserves < reserveTargetBalances(bank).lowerBound){
          if(sim.pln) println(s"$bank is below the reserve target range: $avgReserves (avg) < ${reserveTargetBalances(bank).lowerBound}")
          // penalty for too few reserves?
        } else if(avgReserves > reserveTargetBalances(bank).upperBound){
          if(sim.pln){
            println(s"$bank is above the reserve target range: ${reserveTargetBalances(bank).upperBound} < $avgReserves (avg = ${bank._excessReserves}) and receives interest on its reserve account of ${rounded(_reserveTargetBalances(bank).upperBound * math.max(0, _targetFFR.last - 0.01))}")
          }
          if(sim.test) require(avgReserves - reserveTargetBalances(bank).upperBound == bank._excessReserves, "xx")
          transferMoney(this, bank, _reserveTargetBalances(bank).upperBound * math.max(0, _targetFFR.last), "payInterestOnReserves", sim, t)
        }
    }
  }, "CB_payInterestOnReserves", sim)}




  private var _baseYear:Int = 0

  /**
   *
   *  The used base year is updated frequently.
   *
   *    */
  def updateBaseYear (t:Int) {time({
    if(sim.test) require(sim.goodsMarket.weightedAvgPriceOfYear.size == t/48, "sim.goodsMarket.weightedAvgPriceOfYear.size does not have enough entries: " + sim.goodsMarket.weightedAvgPriceOfYear.size + "/" + t/48)
    if(sim.goodsMarket.weightedAvgPriceOfYear.size > 8 && sim.goodsMarket.weightedAvgPriceOfYear.size % 5 == 4) _baseYear += 5      // moving base year
   }, "CB_updateBaseYear", sim)
  }

```

```scala



/**
 *
 *  The inflation rate can either be calculated using the CPI or the inflation deflator.
 *
 *    */
def determineInflation = {time({
  // according to CPI
  val CPI = sim.goodsMarket.weightedAvgPriceOfYear.map(avgPriceLevel => rounded( (avgPriceLevel / sim.goodsMarket.weightedAvgPriceOfYear(_baseYear)) * 100 ) )
  if(_inflationCPI.nonEmpty) _inflationCPI.clear
  for(i <- 0 until CPI.size-1) _inflationCPI += rounded( ((CPI(i+1) - CPI(i)) / CPI(i)) * 100 )

  // according to GDP deflator
  val deflator = sim.government.GDPdeflator
  if(_inflationDeflator.nonEmpty) _inflationDeflator.clear
  for(i <- 0 until deflator.size-1) _inflationDeflator += rounded( ((deflator(i+1) - deflator(i)) / deflator(i)) * 100 )
 }, "CB_determineInflation", sim)
}




def determineInflationMP = {time({
  val deflator = sim.government.GDPdeflatorMP
  if(_inflationDeflatorMP.nonEmpty) _inflationDeflatorMP.clear
  for(i <- 0 until deflator.size-1) _inflationDeflatorMP += rounded( ((deflator(i+1) - deflator(i)) / deflator(i)) * 100 )
 }, "CB_determineInflationMP", sim)
}




/**
 *
 *   Set Countercyclical Buffer (CCycB) according to the BIS rule (Bank of International Settlement)
 *
 *    */
def setCCycB (t:Int) = {
  if(t>1 && t % 12 == 0){
    val credit          = _credit2privateSector.takeRight(12).sum / 1000
    val nominalGDP      =    sim.government.GDP.takeRight(12).sum
    _credit2GDPratio   += (credit / nominalGDP) * 100
  }
  if(t>1 && t % 12 == 0){
    val credit2GDPtrend = HPfilterData(_credit2GDPratio, sim.lambdaCCycB)
    _credit2GDPgap     += _credit2GDPratio.last - credit2GDPtrend.last
    val L = 2
    val H = 10
    val CCycBmax = 2.5
    val guidedBufferAddon = (_credit2GDPgap.last - L) * (CCycBmax / (H - L))
    val buffer = guidedBufferAddon match {
      case buffer:Double if buffer <  0.5  => 0.0
```

```
      case buffer:Double if buffer <  1.0  => 0.005
      case buffer:Double if buffer <  1.5  => 0.01
      case buffer:Double if buffer <  2.0  => 0.015
      case buffer:Double if buffer <  2.5  => 0.02
      case buffer:Double if buffer >= 2.5  => 0.025
      case buffer:Double if buffer.isNaN() => 0.0
      case _                               => 0.0
    }
    sim.supervisor.CCycB += math.max(buffer, sim.supervisor.CCycB.last - 0.005)
  } else {
    sim.supervisor.CCycB += sim.supervisor.CCycB.last
  }
}








/**
 *
 *  At the end of each fiscal year, the CB agent makes an annual report to update its balance sheets statements.
 *
 *    */
def makeAnnualReport (t:Int) {time({
  updatePVofSoBs(t)
  deposit(equity, Seq(_loans2CommercialBanks.last + bonds.last).sum - Seq(_reserves.last + _OSDF.last + _governmentsAccount.last).sum, t, sim)
  }, "CB_makeAnnualReport", sim)
}







/**
 *
 *  These values are jsut for data saving purposes.
 *
 *    */
val centralBankEndOfSimulationData = Map(
  "OSLF"                         -> _lendingFacilityRate,
  "OSDF"                         -> _depositFacilityRate,
  "RePoRate"                     -> _RePoRate,
  "reverseRePoRate"              -> _reverseRePoRate,
  "listOfBonds"                  -> listOfBonds,            // AB[Int, govBond]
  "inflationCPI"                 -> _inflationCPI,
  "inflationDeflator"            -> _inflationDeflator,
  "inflationDeflatorMP"          -> _inflationDeflatorMP,
  "effectiveFFR"                 -> _effectiveFFR,
  "targetFFR"                    -> _targetFFR,
```

```
    "avgReserves"                     -> _avgReserves,          // Map[Bank, ArrayBuffer[Double]]
    "deficitReserves"                 -> _deficitReserves,      // Map[Bank, ArrayBuffer[Double]]
    "excessReserves"                  -> _excessReserves,       // Map[Bank, ArrayBuffer[Double]]
    "TRvalue"                         -> _TR,
    "outputGap"                       -> _outputGap,
    "credit2privateSector"            -> _credit2privateSector,
    "credit2GDPratio"                 -> _credit2GDPratio,
    "credit2GDPratioTR"               -> _credit2GDPratioTR,
    "credit2GDPtrend"                 -> _credit2GDPtrend,
    "credit2GDPgap"                   -> _credit2GDPgap,
    "credit2GDPgapTR"                 -> _credit2GDPgapTR,
    "outstandingPrivateSectorDebt"    -> _outstandingPrivateSectorDebt,
    "CFSI"                            -> _CFSI,
    "CFSIHP"                          -> _CFSIHP,
    "CFSIgap"                         -> _CFSIgap,
    "liquidityInsuranceDebtBD"        -> _liquidityInsuranceDebtBD,
    "CFSI1_aggBankBailOuts"           -> _aggBankBailOuts,
    "CFSI2_avgFirmDEratio"            -> _avgFirmDEratio,
    "CFSI3_aggBankInsolvencies"       -> _aggBankInsolvencies,
    "CFSI4_avgBankLR"                 -> _avgBankLR,
    "CFSI5_avgBankCCQ"                -> _avgBankCCQ,
    "CFSI6_numbOfActiveFirms"         -> _numbOfActiveFirms,
    "CFSI7_avgBankDEratio"            -> _avgBankDEratio
    "CFSI8_aggFirmInsolvencies"       -> _aggFirmInsolvencies
    )


} // End of class CB
```

A.6.3 Financial Supervisor Class

Supervisor.scala

```scala
/**
 * @author Krugman
 *
 */

package monEcon.publicSector

import monEcon.Agent
import monEcon.Corporation
import monEcon.financialSector._
import monEcon.realSector._
import monEcon.Simulation

import collection.mutable._


/**
 *
 *
 *
 *    */
// ---------- Class for Supervisor-Object ----------
case class Supervisor (sim:Simulation, initialCAR:Double, initialLR:Double) extends Agent {

  val name = "Supervisor"
  val initialCConB = if(sim.CConB) 0.025 else 0.0


  /**
   *
   *  The following values for the prevailing regulatory requirements are set according to the current basel III accord.
   *
   *    */
  private val _CAR                        = ArrayBuffer[Double](initialCAR)                                                          // depending on RWA (risk sensitive)
  private val _capitalConservationBuffer = ArrayBuffer[Double](initialCConB)                                                        // depending on RWA (risk sensitive)
  private val _countercyclicalBuffer     = ArrayBuffer[Double](0.0)                                                                 // depending on RWA (risk sensitive)
  private val _surchargesOnSIBs          = Map[Int, Double](1 -> 0.035, 2 -> 0.03, 3 -> 0.025, 4 -> 0.015, 5 -> 0.01, 6 -> 0.0)       // depending on RWA (risk sensitive)
  private val _minLeverageRatio          = ArrayBuffer[Double](initialLR)                                                           // depending on TA  (non-risk sensitive)
  private val _LCR                        = ArrayBuffer[Double](1)

  def CAR                        = _CAR.last
  def CConB                      = _capitalConservationBuffer.last
  def CCycB                      = _countercyclicalBuffer
  def surchargesOnSIBs           = _surchargesOnSIBs
  def minLeverageRatio           = _minLeverageRatio.last
  def LCR                        = _LCR.last


  /** [tested]
   *
   * Returns the risk weight for a loan in dependence of the D/E-ratio of the corporation.
   *
   *
   *  Basel I RW:
```

```
 *    0% - reserves, govBonds
 *   20% - IBM loans
 *   50% - municipal bonds, residential mortgages
 *  100% - loans to HH/Firms
 *
 *  Basel III RW-range:
 *      0% - loans to sovereign entities (government)
 *    150% - risky loans to firms
 *
 *    */
def riskWeightOfGrantedLoan (corp:Corporation) = {
  corp match {

    case corp:Bank  =>  corp.debt2EquityRatio match {
                  case ratio:Double if(ratio <  10.0) => 0.2
                  case ratio:Double if(ratio <  20.0) => 0.4
                  case ratio:Double if(ratio <  35.0) => 0.6
                  case ratio:Double if(ratio <  55.0) => 0.8
                  case ratio:Double if(ratio <  70.0) => 1.0
                  case ratio:Double if(ratio <  80.0) => 1.25
                  case _                              => 1.5
                }


    case corp:Firm  =>  corp.debt2EquityRatio match {
                  case ratio:Double if(ratio <  2.5) => 0.2
                  case ratio:Double if(ratio <  7.5) => 0.4
                  case ratio:Double if(ratio < 12.5) => 0.6
                  case ratio:Double if(ratio < 20.0) => 0.8
                  case ratio:Double if(ratio < 25.0) => 1.0
                  case ratio:Double if(ratio < 30.0) => 1.25
                  case _                             => 1.5
                }

    case _      => error("To calculate the PD, the client must be either a Bank or a Firm!")

  }
}




/**
 *
 *  These values are jsut for data saving purposes.
 *
 *    */
val supervisorEndOfSimulationData = Map(
    "CAR" -> _CAR,    // AB[Double]
    "capitalConservationBuffer" -> _capitalConservationBuffer,
    "countercyclicalBuffer" -> _countercyclicalBuffer
    )
```

```
}
```