

AGENT BASED MODELING FOR
SUPPLY CHAIN MANAGEMENT:
EXAMINING THE IMPACT OF INFORMATION SHARING

A dissertation submitted to:
Kent State University Graduate School of Management
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

by

Xiaozhou (David) Zhu

December, 2008

Dissertation written by

Xiaozhou (David) Zhu

B.S., Shanghai Institute of Mechanical Engineering, 1989

M.B.A., Kent State University, 1999

M.A. Computer Science, Kent State University, 2002

Ph.D., Kent State University, 2008

Approved by:

Chair, Doctoral Dissertation Committee

Member, Doctoral Dissertation Committee

Member, Doctoral Dissertation Committee

Member, Doctoral Dissertation Committee

Accepted by:

Doctoral Director, Graduate School of
Management

Dean, Graduate School of Management

Acknowledgements

Although only my name appears on the cover of this dissertation, a large number of important people in my life have contributed to its production and completion, to whom I own my great gratitude, and because of whom my graduate experience has become cherishable forever.

I would like to gratefully and sincerely thank Dr. Marvin Troutt for his guidance, understanding, patience, and most importantly, his friendship during my graduate studies at Kent State University. I have been amazingly fortunate to have an advisor who gave me the freedom to explore on my own and at the same time the guidance to recover when my steps faltered. Marvin taught me how to question thoughts and express ideas. His patience and support helped me overcome many crisis situations and finish this dissertation. I am also thankful to him for encouraging the use of correct grammar and consistent notation in my writings and for carefully reading and commenting on countless revisions of this manuscript.

Dr. Murali Shanker, has been always there to listen and give advice. I am deeply grateful to him for his assistance and guidance in getting my graduate career started on the right foot. I am also thankful for the long discussions that helped me sort out the technical details of my work.

Dr. Michael Hu's insightful comments and constructive criticisms at different stages of my research were thought-provoking and they helped me focus my ideas. I am grateful to him for holding me to a high research standard and enforcing strict validations for each research result, and thus teaching me how to do research. Michael has also been a close friend and I did enjoy being invited to eat out with him and having him pay for the meals.

I am also grateful to the former or current staff at Kent State University, for their various forms of support during my graduate study. My special thanks go to Ms. Pam Silliman, who has been so helpful in facilitating my study and teaching in the department, as well as so cheerful that I did enjoy the six years in the program.

Many friends have helped me stay sane through these difficult years. Their support and care helped me overcome setbacks and stay focused on my graduate study. I greatly value their friendship and I deeply appreciate their belief in me.

Most importantly, none of this would have been possible without the love and patience of my family. My family, to whom this dissertation is dedicated to, has been a constant source of love, concern, support and strength through all these years. I would like to express my heart-felt gratitude to my wife Dr. Sijing Zong, my dearest daughters Ruobing and Renee, my parents Xiping Zhu and Aihua Yu, my parents-in-law Shouyin Zong and Tianzhi Shi, as well as many others.

Table of Contents

Chapter 1 Introduction and Background	8
1.1 Introduction	8
1.2 Literature Review	16
1.2.1 Models in SCM Research	17
1.2.1.1 Define the Supply Chain	17
1.2.1.2 SCM Defined	19
1.2.1.3 Categories of SCM Models	22
1.2.1.3.1 Models of Contracting Relationships	26
1.2.1.3.2 Models of Information	31
1.2.1.3.2.1 Models of Information Sharing	31
1.2.1.3.2.2 Information Distortion: the Bullwhip Effect	36
1.2.1.3.3 Models of Operational Relationships	39
1.2.1.3.3.1 Pricing Models	39
1.2.1.3.3.2 Inventory Models	41
1.2.1.3.3.3 Capacity Models	43
1.2.1.3.4 Summary of SCM Model Research	44
1.3 Information Sharing Research in SCM	45
Chapter 2 Methodology	50
2.1 Agent Based Modeling	50
2.1.1 Agents	52
2.1.2 Multi-Agent Based Simulation (MABS)	56
2.1.3 Agent Structure Design	62
2.1.4 Agent Based Modeling in SCM	63
Chapter 3 The Model	72

3.1 Conceptual Model	72
3.1.1 Components and Connections	74
3.1.2 Modeling Structure	76
3.2 General consideration of model design architecture	86
3.2.1 Environment	87
3.2.2 Agent objectives and behaviors	88
3.2.2.1 Agent objectives	89
3.2.2.2 Agent behaviors	90
3.2.3 Important Parameters	92
3.2.4 The Architecture	93
3.3 Model implementation	98
 Chapter 4 Experimental Design and Analysis	104
4.1 External Validation	104
4.1.1 The Results	119
4.2 Experimental Design (Internal Validation)	128
4.2.1 The Results	134
 Chapter 5 Conclusions and Future Research	150
5.1 Summary and Conclusions	153
5.2 Future Research	157
5.3 Software Packages for Building and Running the Agent-Based Models	159
 Bibliography	164
 Appendix The Java Code for the Model Implementation	180

Chapter 1 Introduction and Background

1.1 Introduction

A supply chain is a complex dynamic system. Every member (or agent) within the system engages repeatedly in local interactions, giving rise to many flows such as information, material, orders, money, personnel, and capital as defined by Forrester (1959). These flows in turn feed back into the determination of local interactions. The result is an intricate system of interdependent feedback loops connecting micro behaviors, interaction patterns, and flows of different types.

Researchers have grappled with the modeling of supply chain management systems in the past two decades. Nevertheless, the industrial dynamic systems devised by Jay W. Forrester in the late 1950's, in which he systematically defines the levels and flows in a industrial dynamic system, still remains the fundamental paradigm that frame the way many researchers in this field construct models for their research. As detailed in his later work, Forrester (1968) asserts that industrial dynamics, described as the application of feedback to social systems, "is evolving toward a theory of structure in systems ... In high-order, nonlinear systems, with multiple loops and both positive and negative feedbacks, are found the modes of behavior which have been so puzzling in management ..."

The most salient structural characteristic of Forrester's dynamic system model is its strong dependence on the information flow over other flows in the system. All agent interactions are mediated through information feedback mechanisms. However, when researchers study the problems of information exchange (sharing) in a supply chain system as discussed in the chapter that follows, the interactions among the agents are simply and intentionally ignored. This negligence is broadly evidenced in the most common (prevailing) and traditional type of research published in such prestigious journals as Management Science and Operations Research. In these widely-read published studies, every member in a system only takes available information associated with cost (e.g., order quantity, demand forecasting, etc.) as given aspects of their decision problems. Thus, outside of their control, their decision problems reduce to simple optimization problems with no perceived dependence on the action of other agents. This observation holds for the decision problems faced by both suppliers and retailers. Most such research tries to establish models to find the equilibrium (with or without certain mathematical boundaries) and the equilibrium values for the linking cost variables are determined by inventory levels, demand forecasting, and other factors; they are not determined by the actions of, and interactions among suppliers and retailers, or any other agency supposed to actually reside within the system. These mathematically generated equilibriums do not address, and were not meant to address, how flows in a supply chain actually take place in real-world industrial dynamic systems through various forms of information sharing. What Forrester (1968) says still remains mostly true: "... the mathematical orientation of management science, the concentration on analytical solutions, and the optimization objectives could cope only with rather simple situations.

They excluded treatment of the more complex management relationships and also force neglect of most nonlinear phenomena.” In an era that research in many research fields (take behavioral finance as an excellent example) believe that the organizational and human individual behaviors should be taken into considerations in the studies involving organizational and/or human interactions. A famous study is the virtual stock market using the agent based modeling concepts in Santa Fe Institute (see LeBaron, 1998, 2000, and 2001). It seems to the author that the studies of how such behaviors may have impact on a supply chain management system are becoming inevitable.

What, specifically, is meant by “information sharing” in a supply chain? Thumbing through the papers collected on the topics that are related to information sharing, we surprisingly find that there is not a single author that has ever given the convincing and widely agreeable definition. Researchers simply take the magical phrase and apply it to their topical area with the perception that the meaning of it is naturally clear and specific. As a result, with no surprise, the scope of information sharing varies widely in the collected literature. This dissertation does not intend to supply such a characterization that is globally accepted, but a working definition is necessary to draw a line that confines the research scope of the study.

Literally, “information sharing” consists of two meaningful components: information and sharing, which propose three questions: “what information is shared?”, “how does sharing take place?”, and, putting the two components together, “how (and how much) information is shared?” The answer to the first question of the type of information can be collectively answered by authors in the field, which covers broadly the (advanced)

demand (forecasting) (Thonemann, 2002; Bourland et al., 1996; Fisher and Raman, 1996; Gurnani and Tang, 1999; Donohue, 2000), inventory (Bourland et al., 1996; and many papers in other areas that consider inventory), capacity (replenishment policy) (Gavirneni, et al., 1999; Gallego and Ozer, 2003; Graves, 1999; Cachon and Fisher, 2000,), general information costs (Lee et al., 2000; Raghunathan, 2001; Chen et al., 2000), and external information feed (competition, cooperation, etc.) (Li, 2002; Padmanabhan and Png, 1997; Anand and Mendelson, 1997). To answer the second question of how sharing takes place, we may look into the interactions between the parties involved, which is the area that has not been well studied in literature. Some authors look into the research that the third question is to examine and try to find out how much information is shared. Our study defines four different modes of information sharing in an attempt to pave a way for future research to possibly extract a general model that uses agents to simulate the behaviors of each member in a supply chain and scrutinize how interactions among these members may affect the benefits of information sharing.

As noted above, information sharing as an important field of study in supply chain management is typically studied using standard mathematic approaches that focus on the equilibrium states of a system. Apart from many significantly important advantages of the traditional approaches, one major drawback of them is the lack of flexibility to accommodate dynamic interactive activities in the systems and thus, due to such interactions, the dynamics and commonly computationally intractable nature of the systems is partially or totally ignored. Fortunately, over time, increasingly sophisticated tools are developed to permit researchers to investigate supply chain management

research in increasingly compelling ways. Some of these tools involve advances in computational power and artificial intelligence (AI) that uses software agents.

From the AI or general computational point of view, multi-agent systems (also known as the agent based modeling, i.e., in short the ABM) and autonomous agents represent a new way of analyzing, designing, and implementing complex software systems. A supply chain, as we have noted earlier, is a complex system and may be represented in a software environment (i.e. simulation), which is obvious that we may apply autonomous agents and ABM to it. The agent-base view offers a powerful repertoire of tools, techniques, and metaphors that have the potential to considerably improve the way in which people conceptualize and implement many types of research. One important reason for the application of ABM in a supply chain system is the capability of modeling the sophisticated patterns of information flow interactions. Examples of common types of interactions include, as carefully detailed by Jennings et al. (1998):

- cooperation (working together towards a common goal, such as to minimize inventory costs);
- coordination (organizing problem solving activity so that harmful interactions are avoided or beneficial interactions are exploited, such as timing in production and transportation); and
- negotiation (coming to an agreement which is acceptable to all the parties involved, such as when the capacity is constrained, a retailer may need to negotiate with the supplier or other retailers about the quantity they may be entitled to or request for).

It is the flexibility and high-level nature of these interactions that distinguish MAS from other approaches and provide the underlying power of the paradigm.

With such understanding as the important role that information sharing may play in a supply chain, the critical impact of interactions among agents in the system on the information sharing, and the clear scope of study about information sharing, this dissertation attempts to answer the following questions:

- How can ABM help us in SCM modeling when examining the impact of information sharing? And
- How much information should be shared to achieve optimal supply chain performance?

To answer these questions as well as to explore the problem structure in more depth, we first investigate the current literature on SCM modeling and identify a way of categorization; we then examine the literature of ABM development and extract the ABM modeling methods in computer science discipline; and thirdly we explore the connections between ABM and SCM modeling to examine whether the ABM approach is an improved alternative to traditional numerical analysis methods. We develop a multi-agent based simulation modeling system for a two-stage supply chain that consists of a manufacturer (supplier) and three retailers and analyze the benefit of information sharing to the chain. We model the members in this supply chain using multi agents, taking into account the interactions between / among these members, to evaluate and compare the benefits of information sharing to those from the existing literature.

The intended contribution of this research is described below. On the one hand, existing literature suggests that the dominating methodology for modeling a supply chain is to develop a mathematically sound theory and to use numerical examples to provide artificial evidence that supports the theory. This approach may be pragmatically of less usefulness for the managers due to its lack of the ability of capturing what is happening in the ever-changing market place. As Cachon and Fisher (2000) listed, the traditional approach, when examining the value of shared information, has many limitations such as 1, the demand is known to the retailers; 2, the retailers are identical; 3, there exists only a single source for inventory; 4, there are no capacity constraints; 5, there are no incentive conflicts among the supply chain's firms; and 6, firms choose rational ordering policies. Although some of the assumptions are still used in our model design and development for our particular case, ABM in fact is capable of relax all the assumptions in the future research. This approach developed in this dissertation encompasses the intelligent agents found in computer science, which are capable of (at least in theory, though many concepts are yet to be implemented due to the limitations of technology advancement nowadays) modeling the interactions among members in a supply chain. Thus, using the mathematically sound theory with the pragmatically useful techniques, a more realistic analytical environment is set up for better decision making.

On the other hand, only a very small number of papers exist using multi-agent simulation approach and they simply model the one-to-one control type of relationships between supplier and retailers. This approach not only does not fully take advantage of the rich capabilities of the agents; but also it fails to consider that there exist interactions among agents (i.e., the members of the supply chain) and as stated previously does not give a

complete, if not incorrect, picture of the subjects being studied. This dissertation explores the connections between multi-agent system approach widely found in computer science and the mathematical modeling method in SCM and shows that agent based modeling is a good alternative.

The third intended contribution is that although an ample number of studies have studied the impact of information sharing in a supply chain system, these studies all look at the supply chain as a vertical structure in which the information and material flows solely vertically. We claim that the supply chain should not be as simple as the most commonly used the two echelon systems in which one supplier interacts with each individual retailer and the retailers do not exchange information and materials at all. The horizontal interaction / information sharing may play an important role in supply chain performance improvement. The new dimension we add to the system makes the supply chain not flat any more, it becomes a 3-D type of structure. Our study opens the opportunities for examining a supply chain system that is closer to the real world, the reality.

Human and organizational behaviors are very important factors that have great impacts on the performance of a system (like SCM) to be modeled. Any models (theoretical) without considering these factors are prone to imperfection, even failure, when applied to a real situation. Multi-agent simulation provides us a powerful and flexible instrument to accomplish such tasks that mathematical models and numerical analysis fail to do. Our method in the dissertation is to apply the simple but generic information-feedback systems theory found in other disciplines (mostly in science and applied science) and the intelligent agents borrowed from the contemporary artificial intelligence research and

applications to model the inter-relationships and interactions among the subjects (firms in a supply chain) studied. Each of such subjects is modeled by an agent cluster. A modular design approach is applied to the architecture to ensure full flexibility and expandability.

The structure of the dissertation is laid out as follows. We first review the current literature. The literature review in this chapter covers models in SCM, multi-agent simulation in SCM, the research in multi-agent simulation, and information-feedback systems theory. We then describe our methodology in chapter 2. In this chapter, introduce the agent based modeling. A conceptual model is illustrated afterwards in Chapter 3, in which we develop a conceptual agent based model for our experiment architecture; and in Chapter 4, we compare our results from the agent based model with those from the literature as the external validation; and the experimental design is outlined in detail in this chapter. The implementation of the model outlined uses Java agents running in Repast (an open-source agent model implementation environment). The analysis and discussion complete this chapter. Chapter 5 concludes the dissertation. Future research directions are discussed.

1.2 Literature Review

As Ganeshan et al. (1999) point out, efforts to describe and explain supply chain management (SCM) have recently led to a plethora of research and writing in this field. At the same time, the level of attention to SCM received in business practices nowadays also heavily influences the growing interest in SCM research. Several researchers have attempted to provide some taxonomies and frameworks to present both practitioners and

academics cohesive information that explains the SCM concept and emphasizes the variety of research work being accomplished in this area (See Bowersox, 1969; Laugley, 1992; Bechtel and Jayaram, 1997; Ganeshan, et al., 1999; and Tan, 2000). These authors, among others, have reviewed relevant streams of thoughts in SCM research, provided integrative frameworks to help design and manage supply chains, or categorized the existing research on SMC to offer organized information for other researchers.

Despite the efforts of previous authors, we believe that the growing and rich literature in SCM warrants a close examination of SCM models published in major operations research and management journals. By doing so, we will be able to better understand what the areas are studied and how these areas of study are modeled, and synthesize future directions of SCM research. We classify these models found in major academic journals into three categories, namely, contracting relationship models, information models, operational relationship models.

1.2.1 Models in SCM Research

1.2.1.1 Define the Supply Chain

Supply chains have existed ever since business has been organized to bring products and services to customers (Kumar, 2001). Many variations are found in literature on the same theme when defining a supply chain. There are two major views of supply chain in the literature.

One school takes the system view which can be found in Houlihan (1985), Jones and Riley (1984), Stevens (1989), Scott and Westbrook (1991), and Lamming (1996). This

theme of thought believes that supply chain is a system of suppliers, manufacturers, distributors, retailers, and customers where materials flow downstream from suppliers to customers and information flows in both directions. As Mentzer et al. (2001) put it: a supply chain is a set of entities (organizations or individuals) directly involved in the upstream and downstream flows of products, services, finances, and/or information from a source to a customer.

Other authors view supply chain as a network of organizations and their associated activities that work together, usually in a sequential manner, to produce value for the consumer (Kumar, 2001). Swaminathan et al. (1998) completes this network view by defining a supply chain as a network of autonomous or semiautonomous business entities collectively responsible for procurement, manufacturing and distribution activities associated with one or more families of related products.

Both views have accurately described the entities, activities, and missions of a supply chain from different perspectives and each has its own emphasis. For the system view, it focuses on the processes of making from raw material to final products and how these products are handed to customers in an effective and efficient way, as well as how information is passed within this system to support those processes. While the network view aims to explain the supply chain through the inter-relations and inter-actions between each entities involved. These entities are highly interdependent when it comes to improving performance of the supply chain in terms of objectives such as on-time delivery, quality assurance, and cost minimization (Swaminathan et al., 1998). Authors with the latter view identify different functions (groups of entities working closely) in a

supply chain such as, procurement of material, transformation of material to intermediate and finished products, and distribution of finished products to customers, etc. (Lee and Billington, 1993).

1.2.1.2 SCM Defined

The term SCM appears to have originated in the early 1980s when Oliver and Webber (1982) discuss the potential benefits of strategically integrating the internal business functions of purchasing, manufacturing, sales and distribution (Harland, 1996). The idea of SCM emerges from logistics management integration, which shifts the focus from materials management to the movement of material throughout the firm in an organic and systemic way to greatly improve the effectiveness and the efficiency of the operation (La Londe ad Masters, 1994).

SCM is different from logistics management, as often confused by practitioners and academics. According to the definition given by the Council of Supply Chain Management Professionals (CSCMP), “Logistics management is that part of supply chain management that plans, implements, and controls the efficient, effective forward and reverse flow and storage of goods, services, and related information between the point of origin and the point of consumption in order to meet customers’ requirements.” (CSCMP, 2005) The transition of logistics (materials) management to SCM is a result that what were hitherto considered mere logistics problems have emerged as much more significant issues of strategic management. Houlihan (1985) studies firms in a variety of industries in the USA, Japan and Western Europe and finds that the traditional approach of seeking trade-offs among the various conflicting objectives of key functions such as purchasing,

production, distribution and sales, along the supply chain no longer worked very well and calls for a new approach, which is SCM.

SCM consists of a decision support system, which is concerned with determining supply, production and stock levels in raw materials, subassemblies at different levels of the given Bills of Material (BoM), end products and information exchange through (possibly) a set of factories, depots and dealer centers of a given production and service network to meet fluctuating demand requirements (Escudero et al., 1999). The necessity of managing the supply chain is mainly contributed by three factors, as indicated by Kumar (2001). These factors include: first, customers become more cost and value conscious and demand more, varied, often individualized value from the supply chain; second, the modern information and communication technologies enable the firms to obtain an overview of the entire supply chain so that they can redesign and manage it to meet this demand; finally, the emergence of global markets and global sourcing have stretched these supply chains over inter continental distances. As a result, the accumulated demand variety, uncertainty, costs, distances, and time lags on a global scale make it even more imperative that these long chains be managed efficiently and effectively. Consequently the focus shifted from the competitive advantage of firms to competitive advantages of entire supply chains.

The definitions of SCM differ across authors. For examples, Monczka et al. believe that SCM is a concept “whose primary object is to integrate and manage the sourcing, flow, and control of materials using a total systems perspective across multiple functions and multiple tiers of suppliers.” Jones and Riley (1995) state that SCM deals with the total

flow of material from suppliers through end users. Cooper et al. (1997) define SCM as an integrative philosophy to manage the total flow of a distribution channel from supplier to the ultimate user. Other examples of definitions can be found in La Londe and Masters (1994), Stevens (1998), and Houlihan (1985). These definitions can be classified into three categories: a management philosophy, implementation of a management philosophy, and a set of management processes. Put them together, Mentzer et al. (2001) give a definition: SCM is “the systemic, strategic coordination of the traditional business functions and the tactics across these business functions within a particular company and across business within the supply chain, for the purposes of improving the long-term performance for the individual companies and the supply chain as a whole.”

It is worthwhile to mention that owing to the rapid advances of information and communication technology, the most recent development in the concept of SCM includes the Internet as one of the key influential factors. The e-commerce, business-to-business (B2B) and business-to-customers (B2C) through the Internet, is transforming organizations and organizational processes and creating new opportunities and challenges for domestic and international companies and their supply chains as the Internet is enabling greater integration of businesses and a blurring of traditional organizational boundaries (Bakos, 1998, Hitt et al., 1999, Lancioni et al., 2000, Overby and Min 2001, Johnson and Whang, 2002). This transformation dramatically changes the relationships between the entities in a supply chain and the perception of the traditional SCM. There is at this moment not a definition of this new SCM from the literature available. As noticed by Lancioni et al. (2000), to date, there have been few academic studies examining the development and use of the Internet in SCM.

For the purpose of this dissertation, we give a working definition of SCM which is built on the Mentzer et al. (2001)'s as well as others definition.

SCM is the system that aims to coordinate the interrelations and interactions among networked business functions, to manage the information flows between these business functions, and to provide strategic and tactical decisions in an effective and efficient way with the help of available information and communication technologies within a particular company and across business within the supply chain, for the purposes of improving the long-term performance for the individual companies and the supply chain as a whole.

1.2.1.3 Categories of SCM Models

The objective of classifying SCM models is to provide a clear overview of the types of models that have been researched in the current major literature organized in the way that we can decide and formulate a model for the dissertation to work from.

There are only a small number of papers that classify the research in SCM. Meixell and Gargeya (2005) review decision support models for the design of global supply chains and assess the fit between the research literature in this area and the practical issue of global supply chain design. Their review is thorough and well organized. However, it does not fit the purpose of our review for the following reasons:

- 1) The review focuses on models for designing a supply chain rather than the management of the supply chains.

2) It investigates the models based on the years that they are developed, i.e., period prior to 1990, between 1991 and 1995, from 1996 to 2000, and years after 2000.

This would not give us a clear view of the focuses of the models.

3) It looks into the models by using four dimensions – decision variables, performance measurements, supply chain integration, and globalization considerations.

4) It does not scrutinize the models based on the problems that each model is trying to solve.

Ganeshan et al. (1999) classify the SCM research into three broad perspectives; competitive strategy, firm focused tactics, and operational efficiencies. These perspectives are further divided into many subcategories. The authors try to give a broad review of updated chart of the historical development of SCM and their focus is on much broader field in SCM research and has not specifically synthesized the models in SCM. Therefore it will not fit the purpose of this paper, either.

Many models have been developed in the SCM literature. Authors address various issues from different aspects of SCM. In this review, we group their focuses into three categories, which are information models, contracting relationship models, and operational relationship models. This categorization is broad and based on the following questions that each categorical model needs to answer:

- What is the focus that the target model is trying to place on? Is it on information or relationships (contracting or operational) issues? Each type of modeling has its own specific questions as described in the following explanations.

➤ Information models:

- ✓ What type of information is the focus of the study? One of the common information often addressed in the literature, for example, is in a typical two-echelon supply chain model the demand information to examine the bullwhip effect (Lee et al. 1997). Other information may include the sales data (real time or batch mode), inventory level information, delivery information, etc. Readers who are interested in more detailed description of the types of information may be referred to Lee and Whang (2000).
- ✓ How information is handled in SCM? Such studies include the use of traditional or electronic data exchange technology to make the information available when needed. Some of common technology include EDI (Electronic Data Interchange), Intranet, Internet, VPN, and many others.
- ✓ What are the roles of information in SCM modeling? Information of different types may play different roles and the sharing of different information may have different impacts to the supply chain performance in magnitude as well as levels.
- ✓ How does the model deal with information distortion, information asymmetry, information sharing, and other phenomenon found in SCM? What are the impacts and how does the model manage to cope with them?

➤ Contracting Relationship models:

- ✓ What are the relationships that SCM modeling is interested in – manufacturer/supplier-retailer, retailer-customer, or others?

- ✓ How does the contract affect the relationship between the entities in SCM and how does a model map this relationship and find the solution to the problem if there is any?
 - ✓ How does a model deal with the supplier selection issues?
- Operational Relationship models
- ✓ What are the capacity, pricing, or inventory policies and how can a model find the optimal one?
 - ✓ What are the variables that a model identifies to work with the capacity, pricing, or inventory problems?
 - ✓ How does the uncertainty in demand impact the decisions on capacity choice and allocation, pricing, and inventory policies? What are the models examining them and how do these models work?

This review focuses on the model-based literature and we conducted a search using library databases covering the major journals in management science and operations management, such as *Management Science*, *Operations Research*, *European Journal of Operational Research*, *Decision Sciences*, *the Journal of Supply Chain Management*, *Decision Support Systems*, etc. Research papers found are grouped into four categories, each of which has subcategories. Figure 1 provides a guideline of how this classification is structured in this paper. The following section gives detailed descriptions of each category.

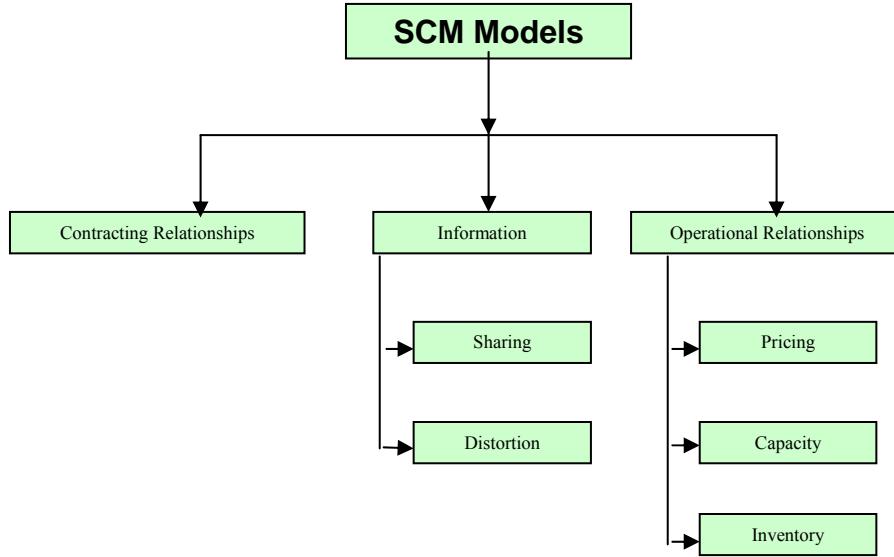


Figure 1: SCM Models Categorization Structure

1.2.1.3.1 Models on Contracting Relationships

Harland (1996) views SCM as the management of supply relationships, i.e., “an intermediate type of relationship within a spectrum ranging from integrated hierarchy (vertical integration) to pure market.” This perspective of SCM has as its foundations an industrial organization and contract view of the firm as a nexus of contracts (Aoki, 1990). Models are found to address the relationship problems related to contracting, incentives, and supplier selection.

Hui and Beath (2001) point out that the natural artifact to analyze the inter-organizational relationship in modern industrial organization is the contract. Contracts not only serve as simply a list of rules, but also define the tone and nature of the relationship. The contract has become an exemplary subject for studying SCM because it is the written artifact that results from firms in a supply chain trying to understand the nature of the relationship

(Walden, 2002). The models that study contracts in supply chain are summarized in

Table 1.

Corbett and DeCroix (2001) study how contracts affect consumption of indirect materials, which are consumed during the production process but do not become part of the final product, by influencing the amount of effort supplier and customer exert to reduce that consumption. The authors create a mathematical model to analyze the impact of shared-savings contracts on channel profits and material consumption assuming that the variable component is linear in the quantity of indirect material used. While such linear contracts can yield higher profits as well as possibly lower consumption, the resulting equilibrium effort levels are generally not channel-optimal outcome since such contracts do not necessarily lead to lower consumption.

Corbett et al. (2005) further the study in the stream by addressing the questions of determining the optimal shared-savings contract from the supplier's perspective, other than the channel perspective adopted in Corbett and DeCroix (2001), in a more general setting. The authors use the double moral hazard framework, in which both parties (consumer and supplier) decide how much effort to exert by trading off the cost of their effort against the benefits that they will obtain from reduced consumption. The model they use extends the double moral hazard literature to allow for a broader class of cost-of-effort functions, including the linear functions found in practice, and shows that the supplier's optimal contract still consists of a fixed part and a variable part which is linear in consumption.

Similarly, the research of Tsay (1999) considers a supply chain consisting of two independent agents, a supplier (e.g., a manufacturer) and its customer (e.g., a retailer). He finds that the retailer, who serves an uncertain market demand and typically provides a planning forecast of its intended purchase, has incentive to initially over-forecast before eventually purchasing a lesser quantity. The supplier must in turn anticipate such behavior in its production quantity decision. This individually rational behavior results in an inefficient supply chain. The author applies the quantity flexibility (QF) contract which couples the retailer's commitment to purchase no less than a certain percentage below the forecast with the supplier's guarantee to deliver up to a certain percentage above. An equilibrium solution is given and the efficiency of QF contract is evaluated through numerical analysis. Under certain conditions, this method can allocate the costs of market demand uncertainty so as to lead the individually motivated supplier and customer (retailer) to the system-wide optimal outcome.

In the same line of research, instead of focusing on using the quantity flexibility contracts, Eppen and Iyer (1997) study the backup agreements (contracts) between a catalog company and manufacturers – a scheme to provide upstream sourcing flexibility for fashion merchandise. As authors define, a backup agreement states that if the catalog company commits to a number of units for the season, the manufacturer holds back a constant fraction of the commitment and delivers the remaining units before the start of the fashion season. After observing early demands, the catalog company can order up this backup quantity for the original purchase cost and receive quick delivery but will pay a penalty cost for any of the backup units it does not buy. They use a dynamic programming model to derive the form of the optimal policy. The model is evaluated by

performing a retrospective parallel test on the data obtained from a catalog company and they conclude that backup agreements can benefit both the retailer and the manufacturer and the adjusting the order commitment in response to the offered percentage to hold by the retailer can have a significant impact on expected profit.

Cachon and Lariviere (2001) compare revenue sharing contracts with other types of contracts we have mentioned above, among others. They found the revenue sharing contracts are at least equivalent with, and most times superior to other types of contracts. Their base model has a supplier selling to a single retailer. The retailer makes two decisions that determine the total revenue generated over a single selling period: the number of units to purchase form a supplier and the retail price. The functions derived show that revenue sharing coordinates this supply chain; i.e., the retailer chooses supply chain optimal actions (quantity and price) and the supply chain's profit can be arbitrarily divided between the firms. Further, a single revenue sharing contract can coordinate a supply chain with multiple non-competing retailers even if the retailers have different revenue functions. However, revenue sharing generally does not coordinate competing retailers when each retailer's revenue depends on its quantity, its price, and the actions of the other retailers.

Cachon (2004) looks into the above issues from inventory risk management perspective. He considers, as others, a supply chain with one risk-neutral supplier buying a product one risk-neutral retailer and demand for the product being stochastic over a single selling season, and studies how the allocation of inventory risk through various types of contracts influences a supply chain's performance and its division of profit. Three types

of wholesale price contracts are modeled. With the push contract, the retailer must pre-book inventory and the supply only produces the retailer's pre-book quantity, therefore, all inventory risk is pushed onto the retailer. In contrast, with the pull contract, the retailer pulls inventory from the supplier with at-once orders, thereby leaving the supplier with all inventory risk. The third type is the advance-purchase discounts, which blends both push and pull by having two wholesale prices. The retailer may pre-book some inventory at a lower price than at-once wholesale price to bear the risk on that inventory; and the supplier may produce additional inventory in anticipation of at-once orders to bear the risk on that additional production. Cachon (2004)'s research shows that the allocation of inventory risk matters for supply chain efficiency even if firms are risk neutral. It is also shown that without changing the wholesale price, merely shifting the inventory risk from one firm to another can improve supply chain efficiency and increase profit at both firms. In addition, if the firms are willing to share inventory risk via advance-purchase discounts, then supply chain coordination is achievable with any division of the supply chain's profit.

A more recent study on shared demand forecasts in a supply chain is conducted by Cachon and Lariviere (2001). The authors identify three key components of a software development contract – product definition, intellectual property protection and payment. They develop a game-theoretic model to incorporate incentive and information issues associated with the payment structure in software contracting. They derive the structure of a viable contract that aligns the incentives of the contracting parties and produces the same efficient equilibrium outcome as in in-house development. However this model has several drawbacks because of some unrealistic assumptions, such as players in the game

are risk-neutral; the design will not change in the middle of the development; and it assumes that the value of a design is exactly determined based on the prototype.

Baiman et al. (2001) model the contracting relationship between a supplier and a buyer, which not necessary is a retailer. However, their focus is on the buyer outsourcing the production of some part to the supplier. They study the interactive effect of the performance metric chosen and the architecture of the product being manufactured on the incentive efficiency of the supply chain. The performance matrices studied are the occurrence of an external (final product fails after sale) and /or internal failure (defective product supplied by the supplier).

Contracting relationship is one of the most important artifacts in SCM. Number of literature can be found and we pick several representative ones to demonstrate the main stream models that have been studied to date.

1.2.1.3.2 Models on Information

SCM is concerned with coordination of independent enterprises in order to improve the performance through the whole supply chain by considering their individual needs. One of the important issues of the coordination is to manage the product and production information among them (Lau et al., 2004). The models that study the management of information in supply chain deal mainly with the issues as related to information sharing and information distortion.

1.2.1.3.2.1 Models on Information Sharing

Research on SCM information modeling has different focus as stated above. However, literature shows that the emphasis of this theme of research heavily leans to information sharing. The most commonly asked questions are, “What is the value of information and information sharing?” “What and how information are shared in supply chain?” and “How do we model the information sharing in supply chain?” The answers to these questions are the models as summarized in the following text.

Gavirneni et al. (1999), Lee et al. (2000), and Raghunathan (2001) model the value of information and information sharing by incorporating information flow between a supplier and a retailer in a typical two-echelon supply chain. Gavirneni et al. (1999)'s model captures the capacitated setting of a supply chain. They consider three situations with three models: 1) a traditional model where there is no information to the supplier prior to a demand to him except for past data; 2) the supplier knows the (s, S) policy (i.e., at the supplier-retailer interface there is an implicit fixed ordering cost) used by the retailer as well as the end-item demand distribution; and 3) the supplier has full information about the state of the retailer. The authors' computational results of these models show that information is always beneficial but the degrees may vary. That is, in the case that end-demand variance is high, or the value of $\Delta = S - s$ is very high or very low, the benefits of additional information is not as great as in the other case when end-item variance is moderate and the value of Δ is not extreme.

While the research by Gavirneni et al.. (1999) is based on demand processes that are independent and identically distributed over time; thus the benefit of information sharing lies in the supply's capability to react to the retailer's needs via the knowledge of the

retailer's inventory levels to help reduce uncertainties in the demand process faced by the supplier, Lee et al. (2000) examine a different situation in which the underlying demand process is a simple auto correlated AR(1) process. A supplier can benefit from obtaining information about the demand from the retailer because it would enable the supplier to derive a more accurate forecast of future orders placed by the retailer. The analytical and mathematical analyses show that the supplier can benefit from inventory reduction and cost reduction with information sharing (of demand). In addition, the authors find that under the conditions that underlying demand is highly correlated over time, highly variable, or when the lead time is long, the supplier obtains larger benefits.

However, the subsequent research by Raghunathan (2001) reaches a different result. He finds that the supplier's benefit is insignificant when the parameters of the AR(1) process are known to both parties, as in Lee et al. (2000). The key reason for the discrepancy is that Lee et al. (2000) assume that the supplier also uses an AR(1) process to forecast the retailer order quantity. Nonetheless, the supplier can reduce the variance of its forecast further by using entire order history to which it has access. Thus, when intelligent use of the already available internal information (order history) suffices, there is no need to invest in inter-organizational systems for information sharing. However, the AR(1) demand process of Lee et al. (2000) may be too simplified. Some retailer actions such as promotion, price reduction, and advertising taken during next period may change the AR(1) demand process over time. In this case, if the retailer actions are independent of each other, the information sharing will still be valuable because the retailer actions can not be inferred by the supplier using order history. In a more general case, the normality assumption of the AR(1) demand model of Lee et al. (2000) may also be over-simplified.

A slightly different research from Gavirneni et al. (1999) and Lee et al. (2000) is by Cachon and Fisher (2000), who study the value of sharing demand (received by retailers) and inventory data (inventory replenishment policy, as well as the current inventory positions) in a model with one supplier, N identical retailers, and stationary stochastic consumer demand. Cachon and Fisher (2000) compare a traditional information policy that does not use shared information with a full information policy that does exploit shared information to allocate the supply chain inventory based on the retailers' actual inventory positions rather than the number of batches they order. Both numerical and simulation based analysis show that the latter case offers cost savings. In addition to the contributions offered by the other authors, Cachon and Fisher (2000) developed a lower bound over all feasible policies. They find that the cost difference between the traditional information policy and the lower bound is an upper bound on the value of information sharing. The more interesting result is that by contrasting the value of information sharing with the benefits of information technology (IT), i.e., faster and cheaper order processing, the authors conclude that implementing IT to accelerate and smooth the physical flow of goods through a supply chain is significantly more valuable than using IT to expand the flow of information. This view is supported by the research of Srinivasan et al. (1994), who use baseline logit model to explore the relationship between shipping discrepancies and two factors (JIT schedules and EDI integration) that capture the level of vertical information integration between a firm and its suppliers. This study suggests that gains from JIT systems can be substantially increased by modern information technology support.

Although information sharing has the potential to dramatically improve supply chain performance, some research find this may not always be true. A recent paper by Terwiesch et al. (2005) studies information sharing from the demand forecast perspective. The authors find that the retailer's forecasting behavior can be characterized by the frequency and magnitude of forecast revisions it requests (forecast volatility) as well as by the fraction of orders that are forecasted but never actually purchased (forecast inflation). They model the evolution of a soft order to a firm order and ultimately to a delivered piece of equipment in the form of a two-stage process. The first state captures the fact that soft orders can either end up as firm orders, that is, the buyer places an order, or be cancelled. In the second state, a firm order will see a delivery time. They find that forecast sharing does not provide benefits to the performance of the supply chain. This may be explained from two perspectives. As for the supplier, the forces that prevent effective forecast sharing are forecast volatility and forecast inflation. The supplier is not willing to allocate production capacity to the soft order that has changed multiple times and penalizes the retailer for inflated forecasts through longer delivery times. On the other hand, retailer will provide more aggressive forecasts to those suppliers that have failed to deliver previous orders on time. This follows the logic of the repeated prisoner's dilemma game and establishes that both retailer and supplier apply a tit-for-tat strategy. The authors call for research to analyze supply chain coordination in repeated game settings and to overcome the forecast volatility problem.

Most of the research as reviewed above study the models that apply to information sharing for inter-organizational supply chain management and very few has looked into the intra-organizational supply chain. Unlike others, Chen considers a supply chain

whose members are divisions of the same firm. Under the assumption that he division managers share a common goal to optimize the overall performance of the supply chain, which is difficult to achieve in an inter-organizational supply chain, the author develops a team solution that reveals that information lead time in determining the optimal replenishment strategies offer cost savings.

1.2.1.3.2.2 Information Distortion: the Bullwhip Effect

As noted in the previous text, one important mechanism for coordination in a supply chain is the information flows among members of the supply chain. These information flows have a direct impact on the production scheduling, inventory control and delivery plans of individual members in the supply chain. Several researchers (Forrester, 1959, Sterman, 1989, Lee et al. 1997, etc.) in management science notice that there exists systematic distortion in demand information as it is passed along the supply chain in the form of orders. This information distortion is called bullwhip effect.

The “bullwhip effect” was first coined by Lee et al. (1997) in their earlier working paper on the same topic. This is a phenomenon where orders to the supplier tend to have larger variance than sales to the buyer (i.e. demand distortion), and the distortion propagates upstream in a amplified form (i.e., variance amplification). Evidence of the bullwhip effect is reported by Sterman (1989) in an inventory management experimental context called “Beer Distribution Game.” This experiment, under the linear cost structure, shows that the variances of orders amplify as one moves up in the supply chain. The author interprets the phenomenon as a consequence of players’ systematic irrational behavior, or “misperceptions of feedback.” Another evidence is reported by Kahn (1987), who

models inventory behavior that incorporates stockouts, backlogs, and serially correlated demand in a supply chain. He reports that the variance of production will exceed the variance of sales even in the absence of movements in productivity or costs. Lee et al. (1997) in their study develop mathematical models of supply chain that capture essential aspects of the institutional structure and optimizing behaviors of members who are assumed to be rational and optimizing. These assumptions are important because the authors employ mathematical models to explain the outcome of rational decision making (for example, the supplier may allocate the supply in proportion to the retailer's share of total sales of the product), as opposed to deriving an optimal decision rule for managers, which may cause the nonproductive gaming (for example, the manager of a retailer who assesses the possibility of being placed on allocation by the manufacturer due to insufficient supply. He may exaggerate the actual demand to get the greater allocation).

There are four causes of bullwhip effect, namely, demand signal processing, the rationing game, order batching, and price variations. Lee et al. (1997) use four models to investigate the effects of the four causes that lead to systematic distortions of information in the order-replenishment transactions of a standard supply chain. Table 3 lays out the mathematical models from their research in addition to the research by others in this field.

At the same time, Metters (1997) uses a classical approach to determine the optimal policies by dynamic programming as the model used by Zipkin (1989). He concludes that a lack of inter-organizational communication combined with large time lags between receipt and transmittal of information are at the root of the problem.

Cachon (1999) models the supply chain demand variability using the same settings as Lee et al. (1997) with one supplier, N retailers, and stochastic demand. In addition to the findings of lee et al. (1997), the author shows that the supplier's demand variance will generally decline as the retailers' order interval is lengthened or as their batch size is increased. By reducing supplier demand variance with scheduled ordering policies, the total supply chain costs can be lowered.

Another important finding about bullwhip effect is reported in Chen et al. (2000). Beginning with a simple supply chain model with one retailer and one supplier, the authors quantify the bullwhip effect by considering two of the factors: demand forecasting and order lead times. They show that if a retailer periodically updates the mean and variance of demand based on observed customer demand data, then the variance of the orders placed by the retailer will be greater than the variance of demand. The authors then extend the results to multiple-stage supply chains and find that providing each stage of the supply chain with complete access to customer (centralized) demand information can significantly reduce this increase in variability. However, the bullwhip effect can not be eliminated fully as noticed by other authors (Metters, 1996; Cachon, 1999)

In summary, the bullwhip effect phenomenon has been described in the literature over many years; however, it is only in the past decade that the full extent of the problem has been recognized, which has stimulated the interest of a number of researchers. It is noticeable that better information sharing between the members in a supply chain may help mitigate the damages of bullwhip effect. The advances in information technology

may offer better and more efficient channels and methods for information sharing. Some future research may investigate the impact of bullwhip effect in the information age and define new models to alleviate deficiency caused by this phenomenon.

1.2.1.3.3 Models on Operational Relationships

Operational relationships in SCM are widely researched and they mainly cover the models in pricing, inventory, and capacity management. The following sections review the literature in these three sub-categories.

1.2.1.3.3.1 Pricing Models

Not all supply chain models are directly related to information sharing. However, to make the categorization complete, we include the pricing models here briefly.

Research in pricing models are mostly related to “timing” for limited capacity product, that is, prices are determined based on the time priorities (the user needs the product now – higher priority, or he can wait until the product is available after a delay – lower priority), and the prices are set based on the priorities. Several papers study the internal pricing for service facilities (Mendelson and Whang, 1990; and Dewan and Mendelson, 1990), others study the pricing policies in the competing firms in a supply chain (Lederer and Li, 1997). Also others study the pricing for optimal bundling strategies for information goods.

Dewan and Mendelson (1990) study optimal pricing and capacity decisions for a service facility in an environment where users’ delay cost is important. Their model assumes a

general nonlinear delay cost structure and incorporates the tradeoff between the delay cost and capacity cost. A queuing model is defined by considering a flow of service requests generated by a continuum of atomistic individual users to the system. Each job is set a price by the service department and each user makes individual decision on whether or not to submit his jobs for service. Under certain restrict assumptions, the job arrival and service times are modeled in a queue. Based on the short-run and long-run situations, the authors develop the optimal pricing scheme that would lead to an optimal utilization of the available capacity.

Also using the queuing theory, Mendelson and Whang (1990) conduct a similar research and derive a pricing mechanism which is optimal and incentive-compatible in the sense that the arrival rates and execution priorities jointly maximize the expected net value of the system while being determined, on a decentralized basis, by individual users. The resulting price structure from their model reveals how the factors of job length and priority each contribute to the overall costs inflicted by a job on the rest of the system.

The queuing theory is also used by the research by Lederer and Li (1997), when they study how delay costs affects prices, operating policies, sales, and firm profits in a competitive environment. This paper assumes that firm capacity, processing variability and cost function are all fixed. The prices are determined by the competitive equilibrium developed in the paper.

One type of the products in a supply chain is information goods – software or other copyrighted materials that are distributed online (almost costless via data networks such as Internet) or in stores. The existing traditional theory and practice in SCM are not

suitable for providing clear guidance on how digital information goods should be packaged, priced, and sold. Bakos and Brynjolfsson (1999) analyze the strategy of bundling a large number of information goods and selling the bundle for a fixed price. The authors use statistical techniques to provide strong asymptotic results and bounds on profits for bundles of any arbitrary size. It is shown that the model can be used to analyze the bundling of complements and substitutes, bundling in the presence of budget constraints, and bundling of goods with various types of correlations and how each of these conditions can lead to limits on optimal bundle size.

1.2.1.3.3.2 Inventory Models

Inventory management may be one of the most important fields and takes a major portion of the SCM models researched in the category of operational relationships.

An early paper by Topkis (1968) considers the problems associated with an inventory system in which demands for stock are prioritized based on its classes of importance. The author investigates the conditions that will satisfy the optimal rationing policy. A later research by Nahmias and Demmy (1981) continues the study and model an inventory system which maintains stock to meet both high and low priority demands. They analyze the following type of control policy: there is a support level, say $K > 0$, such that when the level of on hand stock reaches K , all low priority demands are backordered while high priority demands continue to be filled. Both continuous review and periodic review systems are considered. They compare fill rates when there is rationing and when there is no rationing for specified values of the reorder point, order quantity and support level.

Along the same line, Axsater (1993) by comparing the one-for-one replenishments at the retailers, studies the exact and approximate evaluation of general installation stock policies where both the retailers and the warehouse order in batches. Similar to most others, the author assumes an inventory system with N identical retailers. Other assumptions are stationary and independent poison demand, significant order costs, and constant lead time, etc. Such strict assumptions make the mode rigorous but the relevance of it to the practices is questionable.

A different inventory system is considered by Ha (1997). The author investigates the inventory rationing in a make-to-stock production system with several demand classes and lost sales. The simple queuing model and numerical analysis show that the optimal policy can be characterized by a sequence of monotone stock rationing levels. For each demand class, there exists a stock rationing level at or below which it is optimal to start rejecting the demand or this class in anticipating of future arrival of higher priority demands.

Rather than studying the inventory rationing, Cachon and Zipkin (1999) consider two games. In both, the supply chain stages independently choose base stock policies to minimize their costs. The games differ in how the firms track their inventory levels (in one, the firms are committed to tracking echelon inventory; in the other they track local inventory). The authors compare the policies chosen under this competitive regime to those selected to minimize total supply chain costs, i.e., the optimal solution. The analysis shows that the games nearly always have a unique Nash equilibrium, and it differs from the optimal solution, which results in that competition reduces efficiency.

Furthermore, the two games' equilibriums are different, so the tracking method influences strategic behavior. The authors show that the system optimal solution can be achieved as a Nash equilibrium using simple linear transfer payments.

Also use the game theory, Cachon (2001) continues the research and examines the supply chain inventory game, in which the firms manage inventory with reorder point policies; competition leads to a pure strategy Nash equilibrium in reorder points, which is a set of reorder points such that no player can lower its cost by deviating from the equilibrium, assuming the other players play their equilibrium strategies; there are no profitable unilateral deviations. The model investigates the competitive behavior in the supply chain inventory game and shows that Nash equilibria exist in reorder point policies. The author also suggests cooperation strategies available to the firms to help improve supply chain performance.

1.2.1.3.3 Capacity Models

The research in capacity for the supplier in a supply chain are very closely linked to and mostly embedded in the studies of other areas such as inventory and pricing, as well as contracting relationships. Only a few models specialize in discussing the particular topics for this type of relationship as how to find the optimal solutions for choosing and allocating capacity instead of through price mechanisms. The general assumptions for this type of models are that supplier has limited capacity and retailer orders exceed available capacity.

Cachon and Lariviere (1999) discuss the allocation game and find the Bayesian equilibrium to investigate the allocation mechanisms that are manipulable and induce retailers to misrepresent their needs (bullwhip effect) and those that are truth-inducing and lead to truthful reporting of retailer information. The authors also discuss the benefits for the retailers from restricting the supply chain to truth-inducing mechanisms. Finally they show how the chosen allocation mechanism influences how much capacity the supplier elects to build. In general, the authors focus is on the impact of the quality of information (truthfulness) and the mechanisms to induce them.

Linear programming models, incorporating the concept of planned lead times with multi-period capacity consumption are used by Spitter et al. (2005) to solve the general capacitated assembly problem. They propose two linear program formulations to find the optimal solution. One is that the capacity restrictions are incorporated using cumulative inequalities. The decision variables involved relate to quantities released at the start of a period and quantities processed in a period. The other one is that the cumulative inequalities are replaced by more detailed equalities. The decision variables involved in this formulation relate to quantities released in a particular period and processed in another (later) period.

1.2.1.3.4 Summary of SCM Model Research

The models presented above are established on very sound mathematical and logical analysis; however, most of them have strict assumptions. For example, in a two-echelon supply, as used in most models, the supplier and retailer are all risk-neutral.

Theoretically, this is not a problem, but in practice, it is very hard to find a risk-neutral

firm. Another example of restrictions is that demand is stochastic, which may not hold in practice. These questionable assumptions lead to a call for rigorous inter-disciplinary research to establish a framework that apply the theoretically strong models to a practical situation with the consideration of firms' behaviors that may violate some or all of the assumptions in those models one way or the other.

As part of the literature review for the dissertation, this section looks into the models for SCM in the major literature in the management and decision science journals and classifies them into three categories. The sole purpose of the categorization is to provide a clear and precise view of the SCM models literature, based on which we can find the gaps in the current research.

1.3 Information Sharing Research in SCM

Theoretical research on information sharing was seen in economics literature (Novshek and Sonnenschein 1982, Clarke 1983, and Vives 1988). Raith (1996) presents a general model which encompasses the existing economic models on information sharing as special cases. He shows the incentives for information sharing is highly related to the cost, i.e., firms exchange information when an improvement of the information about market conditions is valid only as far as information about own demand or cost is concerned, i.e., reduced. Some mixed analysis is presented and the benefit of information sharing is considered conditional.

Only in the middle of 1990's, researchers in operations management started to recognize that the value of information is a central issue when they studied the inventory

management in a supply chain and they substituted for one another (Chen 1998). For example, Hariharan and Zipkin (1995) show that advanced warnings from customers of their orders reduce inventory; and the additional information obtained through various marketing means will do the same (Milgrom and Roberts 1998). These examples mainly focus on the value of better communication between the customer and the supplier as described above. Similarly, it also creates value to improve the communication between supply-chain members. For example, Bourland et al. (1996) investigate the operational side of the tradeoff between communication and inventory and demonstrate that new technology such as EDI (Electronic Data Interchange) enables the supplier to obtain more accurate and timely information from the point of sale in the supply chain, and in turn the retailer could reduce its inventories also. Therefore, as Cachon and Fisher (2000) point out, it is not the question of whether information sharing improves supply chain performance, but how.

A comprehensive literature search shows the research of the value information sharing mainly focuses on two areas: inventory management and demand information. These two areas are not exclusive and in fact most literature emphasizing one have to take into account for the other. Some studies bring in the external factors such as retail competition (Padmanabhan and Png 1997, and Li 2002) and horizontal multimarket coordination (Arnand and Mendelson 1997). There are also researchers who focus on a particular supply chain, such as Wal-Mart, in which centralized demand information is shared (Chen, 1998), but this dissertation does not consider this type of serial inventory system.

Inventory models, as one of the most studied areas in supply chain management, are an important tool for evaluating the benefit of information sharing. Bourland et al. (1996) compare warehouse inventory costs with and without timely demand information and analyze the cost differential due to timely information sharing. They document that more timely shared demand information produces better results in terms of inventory reduction with more inventory benefits produced in a setting of higher demand variability (seemingly a positive way of mitigating the bull-whip effect). Cachon and Fisher (2000) confirm that “capturing and sharing real-time demand information is the key to improved supply chain performance”. These authors focus on sharing demand and inventory data, and they assert that the difference between supply chain costs under traditional (partial) and full information is one measure of the value of shared information.

In contrast to Lee et al. (2000) and Gavirneni et al. (1999), who assume there exists a perfectly reliable exogenous source of inventory and information sharing therefore has no impact on the retailer because its orders are always received in full after a fixed number of periods, Cachon and Fisher (2000) assume that the supplier is the only source of inventory and so information sharing may impact the retailers by changing the supplier’s order quantities or allocations.

However, some author doubt that information sharing has positive impact on the supply chain. For example, Graves (1999) studies a similar model to Cachon and Fisher (2000)’s, with the assumption that there is no outside inventory source, and concludes that information sharing provides no benefit to the supply chain.

Many research put emphasis on modeling the effects of demand. Lee et al. (1997) find that sharing retailer demand and inventory policy information reduces the supplier's demand variance (the bullwhip effect), which should benefit the supply chain, but they do not quantitatively measure this benefit. Chen et al. (2000) continue this study and quantify this effect for simple, two-echelon supply chains consisting of a single retailer and a single supplier. They include two factors that are commonly assumed to cause the bullwhip effect: demand forecasting and order lead times. Demand forecasting information is important both for the supplier as well as the retailer. Gurnani and Tang (1999) present a nested newsvendor model for determining the optimal order quantity with uncertain cost and demand forecast updating. Donohue (2000) suggests that including demand forecast updating in the supply contracts can coordinate the supplier and retailer to act in the best interest of the channel, which is to reduce the overall cost and increase the efficiency of the supply chain. Thonemann (2002) confirms that sharing forecasting information (advance demand information) can improve supply-chain performance, and he shows the benefits of such shared information to both the supplier and retailers, however it increases the bullwhip effect. Gallego and Ozer (2003) find the optimal replenishment policies for multi-echelon inventory problems under advance demand information.

Lee et al. (2000) suggest that when customer demands are highly correlated over time, the benefit of sharing demand information with the supplier is high. Raghunathan (2000) argues that Lee et al.'s results are based on an impractical assumption that supplier uses only the most recent order information to forecast the future orders. In reality, however, as the author indicates, the supplier may make the prediction of future demand basing on

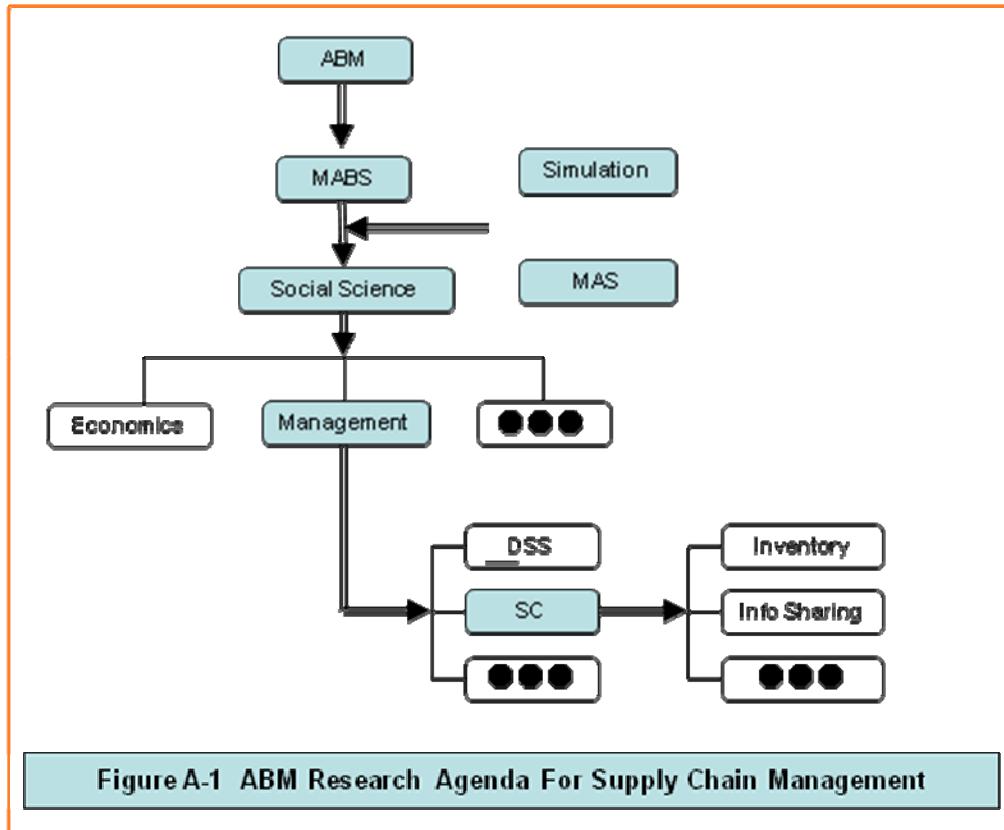
the archived historical order information. Nevertheless, the author's model and proofs are also problematic. For example, the author observes that the order quantities over time, ... $t-2, t-1, t$, are used to forecast the order quantities of next period ($t+1$), the benefit of information sharing approaches 0 as $t \rightarrow \infty$. The proof follows the assumption that the variance is constant – and this assumption may prove to be very unrealistic and perhaps dangerous when applied in the ever-changing competitive market place.

Chapter 2 Methodology

Instead of using the above mentioned traditional modeling methods, an emerging alternative to the supply chain research field is the new modeling approach using agents, the independent small computer programs may be used to represent the individual entities in the simulated world and interact with each other to reveal and help explain the phenomenon that are difficult or impossible to model using the traditional modeling methods. The underling scientific method is simulation. This thesis uses agent-based modeling as the methodology.

2.1 Agent Based Modeling

Figure A-1 shows the logic that links the different parts of ABM research in supply chain management. We start with ABM by combining the concept of multi-agent based simulation (MABS), which have been widely studied in the computer science and engineering fields; the research method of simulation, which is regarded one of the important ways of doing science; and then the agent-based modeling that is based on the theoretical application of MABS and simulation to social science. Then our center of attention turns to supply chain (SC), in particular to information sharing. We define four different modes of information sharing as the conceptual models and the experimental design will be based on these modes.



ABM was coined by Axelrod (2006) in his earlier papers. The purpose of ABM is suitable for the social science objective that is to “understand not only how individuals behave but also how the interaction of many individuals leads to large-scale outcomes. Understanding a political or economic system requires more than an understanding of the individuals that comprise the system. It also requires understanding how the individuals interact with each other, and how the results can be more than the sum of the parts.” By applying this concept to a supply chain, ABM is also very well suitable for modeling and understanding how individual members (organizations) behave, and how the interaction of these members can lead to improved supply chain performance. The system view of ABM method can easily be integrated into the conceptual model of this study.

ABM uses multi-agent based simulation as its essential research method. It is important to understand what it intends to study, and how simulation works. This section will review the literature on these research fields and discuss how ABM can be applied to supply chain research when evaluating the value of information sharing.

Simulations with intelligence were made possible by Artificial Intelligence (AI). AI, first coined by McCarthy (1959), is defined as a subfield of computer science that aims to construct agents that exhibit aspects of intelligent behavior and the notion of “agent” is thus central to AI (Wooldridge and Jennings, 1995). The theories of AI have been advancing along with the development of information technologies including computer hardware and software, computing theories, database, networking technologies, etc. One of the most important applications of AI, among others, is multi-agent based simulation. There has been an intense flowering of interest in the subject in many different fields such as the studies of social and ecological systems (Tsvetovat and Carley, 2004; Drougoul and Ferber, 1992; and Barr, 2004), transportation and geographies (Balmer, et al. 2004), marketing (Janssen and Jager, 2003; Lopez-Sanchez, et al., 2004), and many others. Several studies are also found in SCM model research.

2.1.1 Agents

Although agent research was started in the early 1980’s (for example, Rosenschein and Genesereth (1985) presented rational agents in an Artificial Intelligence conference), it became popular only around 1994 when several key papers appeared. The special issue of Communications of the ACM in the year published such papers as Maes’ now-classic paper on “Agents that reduce work and information overload” (Maes, 1994), Norman’s

conjectures on “How might people interact with software agents” (Norman, 1994), as well as the “Software agents” by Genesereth and Ketchpel (1994). In this period, agents are application programs in the software development (so-called agent-based software engineering) that aims to facilitate the creation of applications able to interoperate in the heterogeneous and dynamic software environment (Genesereth and Ketchpel, 1994).

The milestone of agent theory development was the “Intelligent Agents: Theory and Practice” authored by Wooldridge and Jennings (1995). The authors summarized and synthesized the research in the field. Their work provides an insight into what an agent is, how the notion of an agent can be formalized, how appropriate agent architectures can be designed and implemented, how agents can be programmed, and the types of applications for which agent-based solutions have been proposed. After thorough investigation, they note that although the term “agent” is widely used, there does not exist a universally accepted definition. There are in general two notions of agency, namely, weak and strong notions.

For the weak notion of agency, the authors define the term agent as a hardware or (more usually) software-based computer system that enjoys such properties as being autonomous, social, reactive, and pro-active. The strong notion of agency states that an agent is a computer system that, in addition to having the properties identified above, is either conceptualized or implemented using concepts that are more usually applied to human. Some of the added properties include mentalistic notions (e.g., knowledge, belief, intention, and obligation), emotions, mobility, veracity, and rationality.

There exist some other definitions. An example is Franklin and Graesser (1997) who, in their taxonomy, list many different definitions from various sources in academia and industry – including the one mentioned above – and propose the definition: “An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.”

The varieties of agent design can be found in the computer science literature like Earl et al. (2005), Kargl et al. (1999), Potok, et al. (2003), etc. The applications of agents, in addition to simulations, covers many areas in AI, namely, game playing, speech recognition, understanding natural language, computer vision, expert systems, and heuristic classification.

Certain points may need to be clarified as it comes to the understanding of an agent:

- There is no such thing called a general/all-purpose agent. Unlike many other concepts such as knowledge management for example, agent has been well defined though several versions exist. An agent, fundamentally, is the stand alone piece of software (program) that can accomplish a certain set of tasks. Many properties have been presented in Wooldridge and Jennings (1995).
- An agent is used to present an individual entity in the system. Such anti-team may be a human and animal and organization, and so forth. A system may be a financial market or a supply chain. An agent in the system is governed by behavioral rules. These rules are usually very simple but well defined. Some examples include what action to take when price of a stock goes below a

threshold by a stockholder agent, or whether a new order is triggered when a demand arrives from the customer by a retailer agent.

- The decisions of the agents are coded in an agent based model on the basis of the action of other agents. They may also be based on the results of its own previous decision or decisions, or based on the results of the previous decisions of other agents. We may use a decision tree or decision table to model the decision process of the entity that an agent tries to represent. Sometimes these decisions are also called the agent rules and behaviors. Agents rules are the logic that governs the behaviors of an agent which include a set of logical operators and “if...then” statements. Agent behaviors are model specific, which are a set of rules that tell an agent what to do under certain circumstances. In the implementation, rules and behaviors are the logic in the form of functions, the model specific functions that tell how the agents are interacting with or related to each other; what results may be generated by certain agent actions; how the results affect the behaviors of the agents.
- The design of an agent has its limitations. It's almost impossible to create an agent that can perform everything a human can, however we are still far from knowing what and how the human brain works, because we don't know ourselves. Only when a human behavior is understood and can be described in detail clearly, it is possible modeled by an agent. Just as J. von Neumann indicated in a talk on computers given in Princeton in 1948, the only real limitations on making “machines which think” (in fact the programs that do the job) are our limitations in not knowing exactly what “thinking” consists of. “But of course, a mere

machine can't really think, can it?" von Newmann said, "You insist that there is something a machine cannot do. If you will tell me precisely what it is that a machine cannot do, then I can always make a machine which will do just that!"

Agents are never designed to "replace" human, and agents can never do that.

Nevertheless, the improvements in technology and artificial intelligence make us slowly but steadily closer in making agents "think" the way that human do. As knowledge advances, we are able to define better agents and their behaviors to help use reproduce more and more features of the real world, more and more accurately. No one knows whether there is some natural end to this process, or whether it will go on indefinitely.

2.1.2 Multi-Agent Based Simulation (MABS)

MABS was naturally proposed as the software environment in which agents are organized to interact with each other, which was also once named "federated system" (Genesereth and Ketchpel, 1994). The goal of MABS is to create a system that interconnects separately developed agents, thus enable the ensemble to function beyond the capabilities of any singular agent in the set-up (Nwana and Ndumu, 1999). In their classic work, the latter authors pose major challenges in agent research, and hence MABS research, such as information discovery problem, communication problem, ontology, legacy software integration problem, the reasoning and coordination problem, and the monitoring problem. To date, none of the above problems are close to being solved. Due to the lack of true understanding of what MABS is and what MABS can offer, when

applying this method, many researchers attempt to provide multi-agent solutions to the wrong problems, or even worse, many of them do not really “own” real MABS problems.

Simulation is regarded as the third way of doing science. Axelrod (2006) argues that it “starts with a rigorously specified set of assumptions regarding an actual or proposed system of interest” like deduction, but “it does not prove theorems with generality.” Instead, a simulation generates data that can be analyzed inductively. Unlike typical induction, however, the simulated data comes from a controlled computational experiments rather than direct measurement of the real world. While induction can be used to find patterns in data, and deduction can be used to find consequences of assumptions, simulation modeling can be used as an aid intuition. Simulation as a tool has been widely used in management (social science) field. It is particularly useful for evaluating supply chain reengineering efforts which may have the potential to impact performance in a big way. As explained by Swaminathan et al. (1998), simulation is “the only viable platform for detailed analysis for alternative solutions” in most organizations when making organizational reengineering decisions.

Multi-Agent-Based Simulation is the integration of multi-agent technology and simulation that creates an improved tool for doing science. Exploiting the growing capabilities of computers and advancement in artificial intelligence research that focuses primarily on multi-agent systems, MABS enrich the tool set for attacking the problems in social science that simply applying mathematical analytical modeling approaches may not be adequate because simulations using numerical analysis lack of capabilities of responding to changes in the testing environment.

It is important to note that when applying the concepts and tools from pure computer science and engineering fields that look at the world in an objective and physical way to social science (including management) that views the world with more subtleness and subjectivity, some variations and modifications may be necessary. For example, the agents, which are referred to as bundled data and behavioral methods representing an entity constituting in part a computational constructed world as described in Tesfatsion (2006), can range from active data-gathering decision-makers with sophisticated learning (adaptive) capabilities to passive world features with no cognitive functioning. Moreover, agents can be composed of other agents, thus permitting hierarchical constructions (agent clusters).

The objectives of MABS modeling take four forms: empirical, normative, heuristic, and methodological, as detailed in Axelrod (2006). The primary objective is empirical understanding which asks the question of “why have particular large-scale regularities evolved and persisted, even when there is little top-down control?” We seek causal explanations grounded in the repeated interactions of agents operating in realistically rendered worlds. Ideally, the agents should have the same flexibility of action in their worlds as their corresponding entities have in the real world. In particular, the cognitive agents should be free to behave in accordance with their own beliefs, preferences, institutions, and physical circumstances without the external imposition of equilibrium conditions. The key issue is whether particular types of observed global regularities can be reliably generated from particular types of agent-based worlds.

A second objective is normative understanding: How can agent-based models be used as laboratories for the discovery of good designs? We are interested in evaluating whether designs proposed from processes such as information sharing and member interactions will result in desirable supply chain performance over time. The general approach is akin to filling a bucket with water to determine if it leaks. An agent-based world is constructed that captures the salient aspects of a social systems such as a supply chain operating under the design. The world is then populated with privately motivated agents with learning capabilities and allowed to develop over time. The key issue is the extent to which the resulting world outcomes are efficient, fair, and orderly, despite attempts by agents to gain individual advantage through strategic behavior.

A third objective is heuristic: How can greater insight be attained about the fundamental causal mechanisms in social systems? That is, how can we understand fully a social system (e.g., a supply chain) through systematic examination of their potential dynamical behaviors under alternatively specified initial conditions? Even if the assumptions used to model a social system are simple, the consequences can be far from obvious if the system is composed of many interacting agents. The key issue is the extent to which coordination of supply chain activities emerges and persists as the members collectively learn how to make their production and pricing decisions.

A fourth objective is methodological advancement: How best to provide MABS modeling researchers with the methods and tools they need to undertake the rigorous study of social systems through controlled computational experiments, and to examine the compatibility of experimentally-generated theories with real-world data? To produce

compelling analyses, the researchers need to model the salient structural, institutional, and behavioral characteristics of social systems. They need to formulate interesting theoretical propositions about their models, evaluate the logical validity of these propositions by means of carefully crafted experimental designs, and condense the reported information from their experiments in a clear and compelling manner. Finally, they need to test their experimentally-generated theories against real-world data.

MABS has been applied to social science research. One of the examples is found in Janssen and Jager (2003). The authors use agents to simulate the consumer behaviors to systematically investigate the effects of different network structures. They model the utility of products, cognitive processing by consumers, and product characteristics; and put them in the social networks. Their experiment indicates that besides psychological needs and decision processes, the size and shape of the network involved in consumer decision making have an important influence on how the market organizes itself. A small proportion of consumers may have an exceptional influence on the consumptive behavior of others. Market dynamics is a self-organized property depending on the interaction between the agents' decision-making process (heuristics), the product characteristics, and the structure of interactions between agents. Lopez-Sanchez et al. (2004) show that MABS can be used for strategic decision-making. Their focus is on modeling the structure and behavior of the on-line music B2C (Business to Consumer) market and their constituents or stakeholders. Agents are used to model the product providers and consumers and several behaviors are specified in the simulation, such as those of buy best offers, buy cheapest offers, loyalty, follower, satisfaction, etc. Agents act autonomously according to their interests and interact with other agents inside the market environment.

An important application of MABS is in social systems as an effective tool for reasoning about human and group behavior. Tsvetovat and Carley (2004) believe that the effectiveness is enhanced when the algorithms lead the simulated agents to behave as humans behave, rather than doing what is optimal for the task. More effectiveness can be achieved when the model's inputs are real data and the generated outputs are comparable to actual data files in the real world. Their simulations show that MABS has the potential power for addressing real world issues and can enable a more realistic and extendable architecture for addressing policy issues in a manner comparable to human behavior.

Barr (2004) uses MABS to investigate the role of common knowledge in establishing conventional communication systems. His agents are “egocentric” in how they produce and interpreted signals; and use the signaling game theory to model the behaviors of agents. He shows that large populations of agents can establish and sustain conventional signaling systems without common knowledge. His study indicates that MABS can be used for not only pragmatic decision making as in marketing research, but also building theories that contribute to the development of a general understanding of social systems

Ecological systems are often used to study the process of emergence, which in turn can be applied to sociological systems. The behaviors in ant colonies have drawn interests from many researchers. Drogoul and Ferber (1992) used MABS as a tool for modeling such societies. Agents (ants) are defined with simple tasks, e.g., queen for laying eggs, spy ants for getting outside information, brood agents that represent the three steps of growth (the eggs, the larvae, and the cocoons), etc. Different agent behaviors are defined and a reactive simulation environment is created to simulate an ant nest. Their work provides a tool for modeling societies of simple agents.

MABS also appears to be useful in other areas such as transportation. Balmer et al. (2004) apply MABS to intelligent transportation systems (ITS) to predict travelers' behavior. Agents are used to simulate individual travelers directly and they will react to the system based on certain behavioral rules (e.g., a traveler will take the route that costs her the least time). The agents have the ability of learning but only have local knowledge about the system. Thus optimal solutions for individuals cannot always be computed and the simulation uses heuristics to find good but non-optimal solutions. Also, all agents learn simultaneously, which is called co-evolution. This implies that the traditional simulation theory (one agent is learning while all other agents have fixed strategies) is not a suitable strategy. The scope of the simulation is large which generates approximately five million trips. The authors show that the MABS is a perfect technology to evaluate ITS since it is possible to implement individual behavioral rules for each individual traveler and thus allowing for a differentiated and segmented response of the traveler population.

Other areas of application include, as Jennings et al. (1998) indicate, process control, telecommunications, air traffic control, information management, E-commerce, Games, Healthcare, etc.

2.1.3 Agent Structure Design

The structural design of the agents deals with the specifications of how the agent can be decomposed into the constructions of a set of component modules and how these modules should be made to interact based on certain behavioral rules. Wooldridge and Jennings (1995) thoroughly investigate the agent architectures available to the time and conclude that the classic deliberative, symbolic paradigm is the dominant approach in AI

and this trend goes on. However, this may give way to reactive architectures, which have two key properties: “situatedness” and “embodiment” (real intelligence is situated in the world, not in disembodied systems); and intelligence and emergence (intelligent behavior arises as a result of an agent’s interaction with its environment; also, intelligence is not an innate, isolated property) (Brooks, 1991a, 1991b).

An example of the reactive agent architecture is by Drogoul and Ferber (1992), who program each agent as an object and put these agents in a basic environment called space. Basic communication mechanism is implemented through stimulus-reaction scheme. An agent is seen as consisting of a set of behaviors (tasks). Agents in the simulation exhibit flexible mechanisms of behavior selection and they can take former experiences of interactions with their environment into account when choosing their future behavior. The activation of a behavior also integrates non-environmental conditions such as motivations.

Nwana (1996) studies the software agents – agents in general are the same as those we have discussed in the preceding text – and classified the agent architectures into six types: collaborative (deliberative, symbolic), interface, mobile, internet, reactive, and hybrid. Each type has its own strengths and deficiencies. The architecture that is proposed in our study tries to maximize the strengths and minimizes the deficiencies of these architectures. To this end the hybrid approach appears to be more appropriate.

2.1.4 Agent Based Modeling in SCM

Other than mathematically analyze the model using numerical test, empirical study and simulation are also important methodologies in SCM research. Simulation has in the recent years been adopted in management and social science research and has been regarded as the third way of doing science (Axelrod, 2005). Agent-based simulation which encompasses the simulation technique with the advances in artificial intelligence is characterized by the existence of many agents who interact with each other with little or no central direction, nor human interference. This type of simulation only appears in the academic literature in last decade.

Only a small number of research papers could be found that adopt agent-based simulation as a methodology in supply chain management literature. One of the important works is by Swaminathan et al. (1998), who find that supply chain reengineering (improvement) is critical to the companies exposed to global economy and striving to meet customer expectations regarding cost and service. As the reengineering process is a strategic move, it requires detailed risk analysis. Since quantitative analysis provide insights into current trends but not prescriptive, simulation becomes the only viable platform for detailed analysis for alternative solutions. The authors design a multi-agent framework in which different agents are specified and different control mechanisms are defined. The purpose of this framework is to provide members in the supply chain a customizable decision support tool that can help managers to understand the costs, benefits, and risks associated with various alternatives.

Garcia-Flores and Wang (2002) propose a multi-agent system to model the three flows (money, information, and material) in a chemical supply chain. Their design is very

specific for use in the chemical supply chain. The main emphasis of this design is on the processes of paints and coatings production in a plant in addition to certain simplified relationships up- and down-stream. The major addition in the agent design to those by Swaminathan et al. (1998) is that they use one of the common agent communication languages (ACL) to specify in detail the mechanism of communications between agents.

A shift of interests is found to move from traditional to net-enabled supply chain (using the Internet or other types of electronic telecommunication media) research. Some researchers name this as e-supply chain (Poirier and Bauer, 2000) or e-chain (Singh et al., 2005). The later authors present an agent-enabled architecture that exhibits information transparency (the availability of information through out the supply chain in an unambiguously interpretable format) and enable enhanced interaction among participants in an e-chain. The focus of the system is largely on the supplier-buyer relationships and processes. The authors describe the process of how a discovery agent matches the buyer and supplier based on the buyer's demand, supplier capacity and reputation, etc.; how transaction is promoted by a transaction agent; and the control mechanism facilitated by the monitoring agent. This platform can be applied to multi-buyer multi-supplier supply chain environment

Some researchers use agent-based simulation to study the traditional SCM topics in a new way. Such example is Lin et al. (2005), who examine the effects of trust mechanisms on supply-chain performance in an e-commerce environment. Their research framework has particular focus of exploring the trust mechanism, based on the integrated view of Mayer et al. (1995), in facilitating information flows and transactions

within a supply chain. The authors have not described how agents and agent functions are defined and they implement their study on the Swarm platform, one of the agent-based simulation software packages available.

So the question is: “What makes agent-based modeling a good choice in SCM research?”

Mathematical models in the traditional SCM research use mathematical analysis. Due to the difficulties of obtaining empirical data for validation, simulation techniques are used. One of the popular options, among many stochastic procedures, is to use Monte Carlo simulation, which is a great technique that relies on repeated computation and random or pseudo-random number and is used when it is infeasible or impossible to compute an exact result with a deterministic algorithm. Although Monte Carlo method has wide uses in applied sciences and mathematics, as well as in some business applications such as the calculation of risk in business, evaluation of investment projects, etc., and shares many benefits of agent-based modeling (ABM), it does not work as well in many other areas.

ABM is a totally different and new approach that models a system as a collection of autonomous decision-making entities called agents. Therefore, ABM has become increasingly popular as a modeling approach in the social sciences because it enables one to build models where individual entities and their interactions are directly represented. The modeler has absolute and total control, if desired, of each single agent, and as in a physics or chemical experiment, he/she also has complete control of simulation environment. In comparison with variable-based approaches using structural equations, or system-based approaches using differential equations, agent-based simulation offers the possibility of modeling individual heterogeneity, representing explicitly agents’

decision rules, and situating agents in a geographical or another type of space. It allows modelers to represent in a natural way multiple scales of analysis, the emergence of structures at the macro or societal level from individual action, and various kinds of adaptation and learning, none of which is easy to do with other modeling approaches.

There are many characteristics that make ABM a powerful modeling technique, which may be first seen through the definition of it. Though many different descriptions of ABM may be found, we believe the following is a good working definition:

“...Agent-based modeling is a computational method that enables a researcher to create, analyze, and experiment with models composed of agents that interact within an environment.” (Gilbert, 2007)

This definition has several terms that worth further exploration. Using an example may get us better idea. A conceptual implementation of Sterman (1989) Beer Game supply chain will serve the purpose.

First, ABM is a form of *computational* social science. That is, it involves building models using computer programs. Similar to any other programs, the computational model takes one or more inputs (just as the independent variables if use regression model), and some outputs (similar to the dependent variables). The program itself presents the processes that transfer the inputs to outputs. In Beer Game, the program includes all the participants, i.e., the agents; the inputs are the orders and shipments passing through the chain; and the outputs are the costs to evaluate the significance of

individual decisions, while these individual decisions (or decision rules) plus the embedded system variables are the processes.

Second, in physics, chemistry and some parts of biology, or other so-called hard science, the main and standard method of research in proving the theory or exploring the unknown is to conduct *experiments*, the same process can be repetitively tested many times.

Conducting experiments in most of the social science is impossible or undesirable because the subjects are human beings. Even an experiment is desired and proceeded, the tester does not have control on the heterogeneity of different subjects as one does in, for example, a physics experiment, where one can apply some treatment to an isolated system by controlling the quality, quantity of the subjects, the temperature and humidity of the environment, as well as the duration that can be as exact as to millisecond. Such isolation in the social systems is generally impossible, if not unethical.

ABM simulation provides such a tool for researchers of social science to have the luxury of doing the experiments as those of hard science. The experiments may be repeated many times, using a range of parameters or allowing some factors to vary randomly. The human subjects are replaced by the computer programs, i.e., the agents. The environment is virtually under full control of the researchers as it is programmed in the software. The behaviors of the agents are also under control, though to date many human behaviors are still not replicable due to the constraints of limited understanding of ourselves. We do not understand why, for example, among countless many others, one reacts to one thing that is different from another one's reaction. Thus it's impossible to program these unknown behaviors. It's the hope that the progress in the research of human psychology,

artificial intelligence, etc. will make us more understand our behaviors and the ABM will be more precise in modeling the phenomena that involve human interactions.

However, although each individual is different from each other, in some cases there may be finite categories of possible reactions. For example, in the stock market when investors hear the news of higher unemployment rate in the previous quarter, some may be pessimistic and believe that the economy will experience a recession and the stock market may crash soon. Thus the reaction is to “sell”. On the other hand, some other investors may believe this is temporary and optimistically believe that economy is in good shape and the unemployment rate will soon drop to the normal level and thus they will buy stocks at the perceived low price. There might be other reactions such as wait-and-see, etc. This makes it possible for the researchers to categorize their reactions at the high (aggregate) level, and program the agents to see how the change of one factor may affect the results of the experiment, as one does in physics (hard science).

Getting back to the Beer Game example, the game has been designed simple enough and has been experimented on many groups of individuals of different kinds. However, by reading the results of Sterman (1989), we find that because of the lack of control of individual participants in addition to human errors, the useful samples after four years of work is only 11 out of 48 (i.e., 44 out of 192 subjects). This causes great waste of time and money. With the help of ABM, we may set up an experiment and repeat it as much as desired and change any parameters.

The third term is the *Agents*. Agent based models consist of agents that interact within an environment such as the supply chain. Agents are either separate computer programs or,

more commonly, distinct parts of a program that are used to present social actors – individual people (customers, managers, employees, etc.), organizations such as firms (suppliers, retailers, etc.), or bodies such as nations or states. They are programmed to react to the computational environment in which they are located, where this environment is a model of a real environment in which the social actors operate.

As will be seen in our model implementation, a crucial feature of agent-based models is that the agents can interact, that is, they can pass information or messages (such as the order quantity, demand pattern, i.e., distribution and parameters) to each other and act on basis of what they learned from these messages. The messages may represent spoken dialogue between people in some cases, or more indirect means of information flow, such as the observation of another agent all the detection of the effects of another agent's actions. The possibility of modeling such agent-to-agent interactions is the main way in which agent-based modeling differs from other types of computational models.

The last term is the *environment*. The environment is the virtual world in which the agents act, which is as important as the agents themselves. The environment may be in different forms or of different types. It may be an entirely neutral medium with little or no effect on the agents, or in other models, the environments may be as carefully crafted as the agents themselves. Commonly, environments represent geographical spaces, for example, eating models concerning residential segmentation, where the environment simulates some of the physical features of a city, and in models of international relations, where the environment maps states and nations. This type of environment is often seen in the research of GIS (Geographical Information Systems) field. Models in which

the environment presents a geographical space are sometimes called spatial explicit. In other models, the environment could be a space, but one represents not geography but some other features. For example, scientists can be modeled in knowledge space. This type of agent-based modeling may be used to create expert systems. In these spatial models, the agents have coordinates to indicate their location. Another option is to have no representation at all but to link agents together into a network in which the only indication of an agent's relationship to other agents is the list of agents to which it is connected by network links.

Some prevailing agent-based modeling environments include Repast (Recursive Porous Agent Simulation Toolkit), Swarm, Ascape, all of which support pure java programs and Repast supports Python and Microsoft .Net, and Netlogo, which is a cross-platform multi-agent programmable modeling environment and support click and drop model creation. In our model implementation, we select Repast and the agents are coded using Java programming language.

Chapter 3 The Model

The original intension of the author was to establish a general agent based model structure (or framework) that can be customized to solve a wide range of complex system problems by following the infamous “Make everything as simple as possible, but not simpler.” by Albert Einstein. However, the other part of the slogan that states “but not simpler” proves true as it was later found that agent based modeling has its limitations, one of which is that, common to all modeling techniques, as Bonabeau (2002) puts it, “a model has to serve a purpose”. A general purpose model is not sufficient to define and deliver such a purpose. The model has to be domain-specific and with just the right amount of details to serve its purpose. How much is “right” remains an art more than a science.

3.1 Conceptual Model

Most papers in the literature that we have examined show that the models investigating the value of information sharing look into a simple 2-echelon supply chain structure. Although we may model the entire supply chain from raw material to production to retailers and then to the final customers, we for simplicity, will look into the commonly used 2-echelon approach. When the simplified model is established, it will be relatively easy to expand it to any levels of the supply chain of different types of structure.

Also, it is well known that there are many types of flows within a supply chain, such as, materials, orders, money, personal, capital equipment, and information, as indicated by Forrester (1959), and information sharing may have different levels of impact on these

flows to certain degrees. Again for simplicity and without losing generality, we assume that this type of impact will hold constant throughout the simulation and the focus of the study will lie on those related to the supplier in terms of operational cost such as inventory and returns; and those related to the retailers such as backlog and inventory.

It is necessary to point out that most literature consider the cost savings at the supplier side, and a few others included the retailers when they model the value of information sharing. Only some consider both sides. I find it inadequate to consider either side alone because the interactions between the supplier and retailers may have effects on how the value is maximized. However, it is understandable that these types of interactions may be very difficult to capture using the mathematical modeling approach because there is not a generic pattern for the interactions for various types of supply chain, and the mathematical models cannot adapt to the dynamics in the process of information exchange. Multi-agent approach may be an appropriate method to overcome these shortcomings.

There exist a small number of research results that start to consider the agent based modeling approach. However, these papers, although they claim that they use multi-agents in the simulation, use the agents in a very simple way, which simply perform the roles of gate-keepers type of controlling units in the simulation. For example, in Swaminathan et al. (1998), the agents may take the inputs from the downstream and pass them to the upstream. They may look up the current inventory level and decide whether or not to order more. The agents communicate in a virtual hierarchical structure and

perform very simple tasks based on the simple yes/no logic. They do not preserve their own local memory, thus possess no learning capacities.

The agent based model we design attempts to provide a simple view of a complex system, i.e., we consider the supply chain studied as a complex system, in which members are not isolated and they interact with each other. Members are represented by agents, the computerized representation of the entities in a supply chain who perform specific tasks. The design and modeling of the agents, as well as interactions between / among the agents are described.

For the purpose of this dissertation, that is to develop a multi-agent system that simulates the information sharing process in a supply chain so that the benefits of information sharing can be evaluated more precisely and realistically, I find there are several important components in the system and there exist connections between them.

3.1.1 Components and Connections

As was mentioned earlier, from a system view point, everything in a system is connected to each other in a certain way. The components in the system are briefly discussed below, and the connections between these components follow. More details will be given in the later part of this chapter when explanations of the model are described.

Information: This study lays its focus on information sharing within a supply chain. Information is the key as it is the center of the diagram. Information flow can be viewed as a two-way process. Supplier and retailers share the information both ways, the type of information and degree (in our experimental design as

shown in the next chapter, the degree is defined as three levels: high, medium, and low, respectively) of such sharing may vary. We categorized them into four modes as further discussed later (Section 3.1.2).

- o Impact of Information Sharing

Many papers study the impact of information sharing based on the values or costs. This research follows the same route. Multi-agent simulation is used to identify the total cost saving that information sharing brings to the supply chain.

- o Supplier / Retailers

These are two commonly used terms in supply chain literature. No additional meaning is added to them in this study.

- o Relationships / Interactions

Relationships are defined as the results of the interactions between supplier and retailers (vertical interactions), as well as between retailers (horizontal interactions). Current literature mostly examines the vertical information sharing for a good reason: the traditional definition of a supply chain is the collection of upstream and downstream type of flows (Forrester, 1959), thus it is natural to investigate the effect of information sharing in the same fashion. Since nothing is isolated and everything is interrelated to each other in certain way, it becomes natural to ask the question, “How will the interactions between the retailers

impact the information sharing in the supply chain?" A few researchers have partially addressed this problem, but no overall picture has been drawn.

- o Agents

Software and hardware components that can accomplish certain tasks with certain characteristics that traditional software and hardware do not have are called agents. A discussion of the definition of an intelligent agent can be found in Wooldridge and Jennings (1995) and a discussion in more detail is given in the previous chapter.

3.1.2 Modeling Structure

The information flow may be modeled differently in different modes (or types). As most literature suggests, in the first three modes presented below, the information flows both directions in vertical directions with focus on evaluating the benefit of information sharing from the supplier's perspective. The information sharing in these three modes has direct effect on the performance of a supply chain (Li 2002). Although it is well documented that information indeed flows both ways between the supplier and retailers, most authors (some exceptions such as Cachon and Fisher 2000 who believe that information sharing may have impact on the retailers; and Gurnani and Tang 1999 who study the impact of demand forecasting on retailer's optimal ordering policy) when they evaluate the values of information sharing pay most attention to the flows from downstream (retailers) and ignore the other half from the upstream (supplier). Another aspect that the current literature is missing is that members in a system are not isolated

from each other and there exist interactions. These interactions may have direct or indirect impact on the information sharing studied.

This dissertation extends the views of information sharing from purely half vertical (one way) to full vertical (two ways) with horizontal interactions. External factors are also considered. I may use the term indirect effect (Li, 2002) to address the impact of information sharing in this new mode. I hereby summarize four such modes as described below.

Mode 1: This mode is called “No Information Sharing” by following the model description of existing literature. “No Information” in fact is not a very accurate term. As generally understood, in a system, there is always certain information exchange to maintain the links between components. This is the lowest level at which the least information is shared. Only the necessary information flows between the supplier and retailers so that orders are placed and delivered. Lee et al. (2000), Raghunathan (2000), and Gavirneni et al. (1999) model this as “no information sharing” as the base case for comparison. This is similarly considered as “traditional information sharing” (Cachon and Fisher, 2000) that supplier only observes the retailers’ orders. The information from the supplier to retailers is minimal and is not considered in the research found. See Figure 1 for the graphic representation of this mode. It is noted that this mode is close to the “closed system” notion in the system theory (Bertalanffy, 1968). However, since everything is connected one way or the other, it is not possible to find a real closed system, thus no “no information sharing” mode exists. The presentation of this mode is solely for comparison

purpose. The mathematical analysis such as that in Lee et al. (2000) shall be sufficient to serve for this purpose.

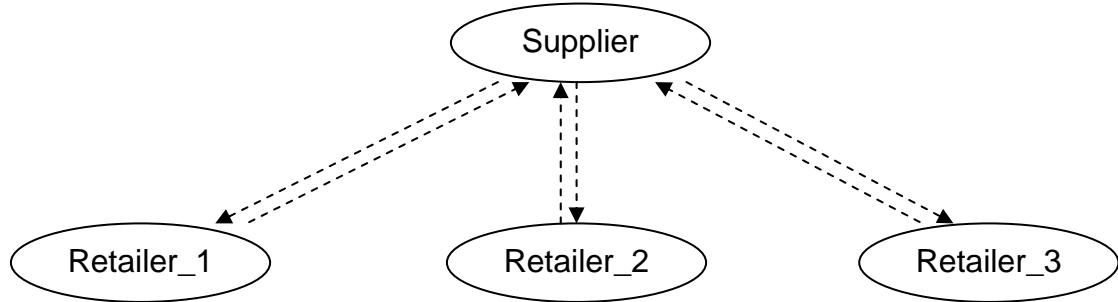


Figure 1 – Mode 1: No Information Sharing

Mode 2: Thonemann (2002) models this as aggregate information sharing, at which retailers share with supplier information about whether they will place an order of some product in the next time period, but do not share information about which product they will order and which of several potential suppliers will receive the order. Gavirneni et al. (1999) name it as partial information sharing, where supplier knows the (s, S) , i.e., order up-to, policies used by the retailers as well. It is assumed the future demands for the supplier are independent of past demand. This assumption may be relaxed using processes such as autoregressive, AR(1) as described in Lee et al. (2000), or ARMA (autoregressive with moving average) described in Graves (1999) and Thompson (1975). Figure 3 shows this mode. Note that the diagram differs from that of mode 1.

- The type of arrows. This shows that more information is shared in this mode. Such information may be as the details of the s-S policy from the retailers (solid lines), or the inventory levels and capacity information from the supplier (dotted lines).
- Demand information flows into the retailers. However, this information does not go further upstream because in this mode, the supplier does not get this information from the retailers, while some authors such as Gavirneni et al. (1999) include the information of end-item demand distribution into this mode. For this reason, the lines related to demand information are all dotted.
- Demand can be stationary or non-stationary, depending on how the model is specified. Many literature studies the stationary demand (Thonemann 2000, Cachon and Fisher, 2000, and Gallego and Ozer 2003), others investigate the non-stationary demand (Lee et al. 2000, Hu 2003, and Graves 1999).

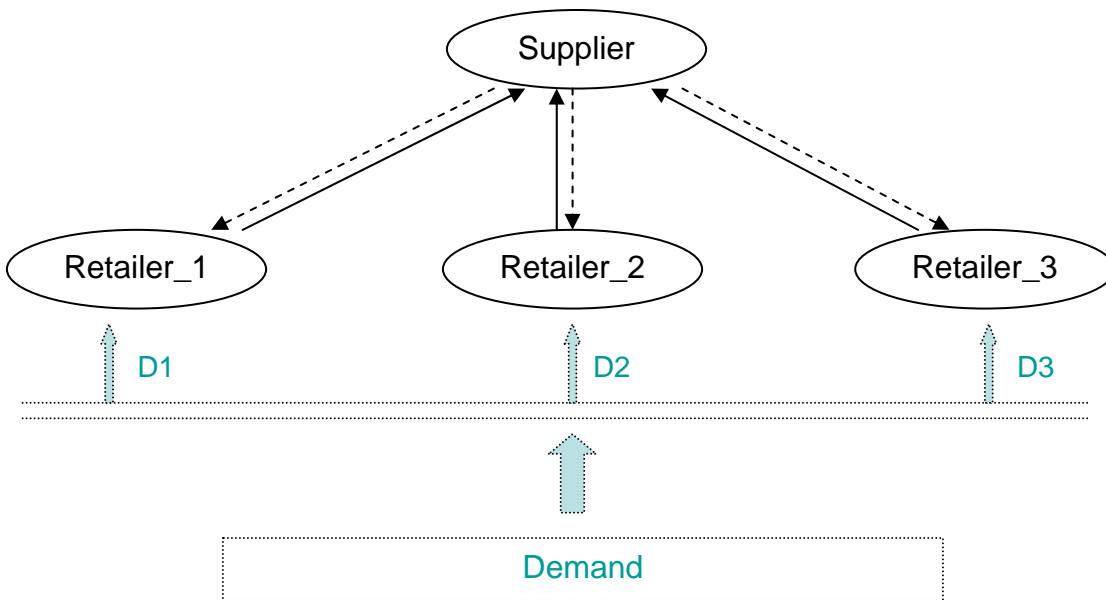


Figure 2 – Mode 2: Partial Information Sharing

Mode 3: full information sharing between supplier and retailers, as modeled by Lee et al. (2000) and Raghunathan (2000) where the supplier has full information about the state of the retailer (Gavirneni et al. 1999), or the “detailed” information sharing as described by Thonemann (2002) where supplier gets the detailed information of the orders to be placed. Again, the current literature focuses on the benefits of information sharing from the supplier side and few consider the retailer side. For example, as Lee et al. (2000) suggest, information sharing by the retailers provides significant inventory reduction and cost savings to the supplier, and the retailers can, on the basis of the potential benefits of such savings to the supplier, negotiate arrangements with the supplier to reduce their own overhead and processing costs.

As the phrase “full information” suggests, the supplier and retailers have established communication channel and trust so that information can flow both way flawlessly. It is assumed that the information flow shall be real time and decisions made without delay. (However due to many reasons such as limitations in information technology, there exists information delay and thus decision delay. This delay will add costs to the supply chain due to, for example, delayed delivery, and the existing literature fails to notice and inform this. It is not the goal of this study to address this problem.) The differences between this mode and the other two are as follows.

- Solid arrows, which represent the assumed solid relationships between the parties. Solid here means the trust, goal congruent, etc., which are the dimensions that can

be found in a large number of research papers and are out of the scope of this study.

- Demand plays an important role in this mode. Demand information is shared between supplier and retailers and it does have impact on the supply chain. Individual demands are assumed to be identical as in majority of the literature, i.e., $D_1 = D_2 = D_3$.

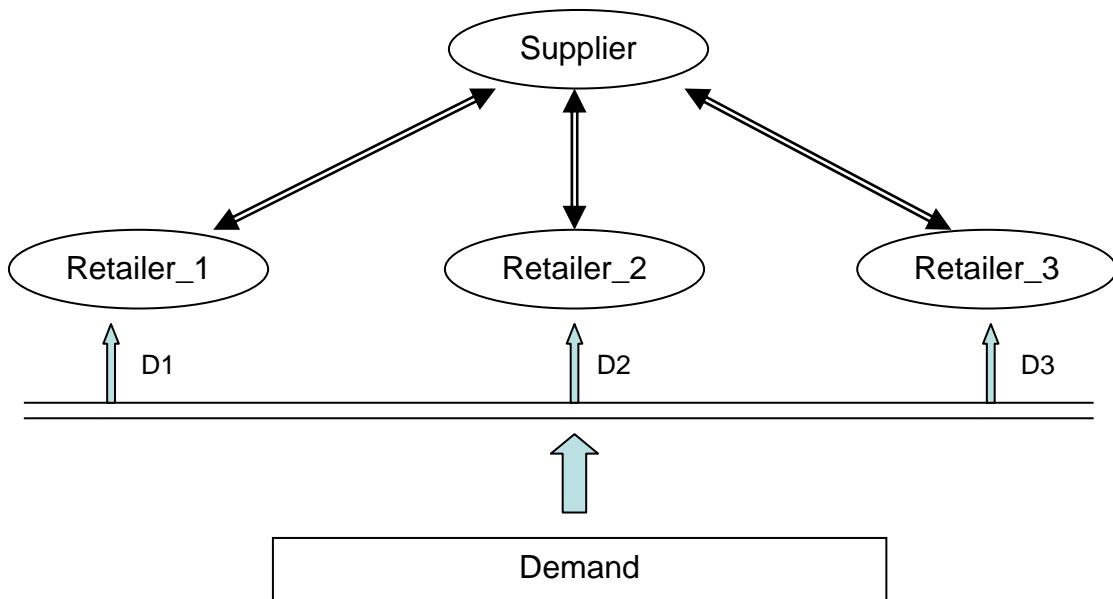


Figure 3 – Mode 3: Full Information Sharing

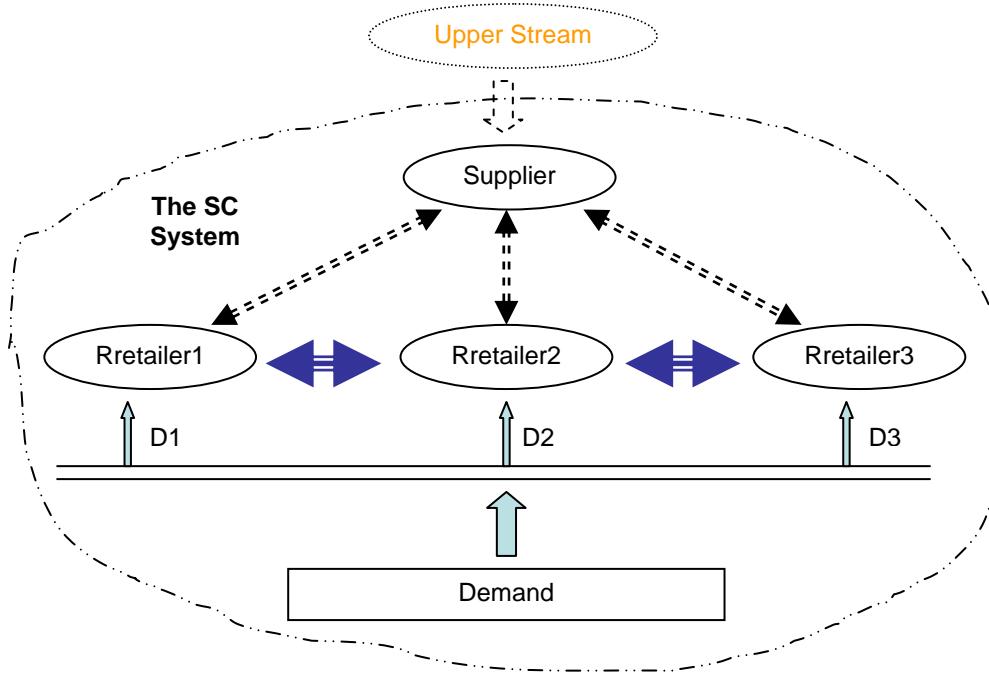
Mode 4: comprehensive information sharing. This level of information sharing has only been partially modeled in the existing literature. For instance, Li (2002) considers competitions among retailers when they model the information sharing in a supply chain.

Different from the other three modes, which consider only the vertical information flows, the comprehensive information sharing mode (Figure 4) factors in the horizontal

interactions as well as the external forces. This mode applies the systems thinking¹ approach: it considers the supply chain as a complete open system. Based on this approach, each part (the supplier, retailers, external demand, etc.) are all considered as the components of the supply chain system and they interact with each other. The difference is obvious: it is not the traditional vertical hierarchy any more, rather, it may be viewed as a networked system. The horizontal interactions appear to be a new area of research. The information, along with other elements such as material/products, flows among different parts that compose the system. Figure 4 indicates these relationships.

Multi-agent simulation is applied to evaluate the impact of comprehensive information sharing. The approach has been shown to be effective for investigating information sharing in supply chains in the literature (Lau et al. 2004, Swaminathan et al. 1998).

¹ An extensive reference list may be found at the web page maintained by W. Huitt at:
<http://chiron.valdosta.edu/whuitt/materials/sysphil.html>



**Figure 4 – Mode 4: Comprehensive Information Sharing
The System View**

As shown in the figure above, the retailers are no longer information isolated from each other. The interactions among them give them the opportunity to exchange information at different levels, depending on the relationships established between them.

The information sharing between retailers is considered as a form of interaction. Interactions are the fundamental research area in the organizational research and may have many different forms (such as, cooperation, negotiation, competition, trust, etc.) They may be between individuals or groups of human beings or organizations. However, our study focuses on the consequences of organizational interactions, i.e., the relationships established and enhanced due to the interactions, but not the forms of different interactions.

The buyer-seller relationships between the seller (supplier) and buyer (retailers), also known as the channel-relationships, as well as strategic alliances literature mainly focus on the issues in a vertical structure. This vertical structure view looks at the system in an incomplete way. It overlooks the fact that the nature of the activities at each level of the vertical structure may have impact on the dynamics of the whole system: organizations at the same level compete for resources or customers of the lower level, and seek possible interorganizational cooperation to, for example, minimize the transaction costs. As Cravens and Shipp (1993) put it, independent organizations collaborate to increase the competitive advantage of each organization, and these cooperative relationships are of escalating importance. Rather than competing independently against other organizations in the market-place, organizations are forming cooperative interorganizational relationships to take advantage of dissimilar strengths of different organizations (Borys and Jemison, 1989). For example, as the authors indicated, over time, “many firms develop more sophisticated supplier arrangements in which reciprocal interdependence becomes strategically important, purpose broadens, and stability mechanisms become institutionally based as well as contractual.”.

The relationships between these organizations as the result of such interactions of cooperation are the focus of the thesis. In particular, we examine the impact of information sharing (the cooperation side) on the costs of the buyers (retailers), and in turn on those of the buyer (supplier). The theoretical support for such examination may be found in Barringer and Harrison (2000) which tries to investigate the theoretical paradigms that explain inter-organizational relationship formation. The authors summarize the six widely used paradigms, including transaction costs, resource

dependence, strategic choices, stakeholder theory, learning theory, and institutional theory, as shown in Table X1. A brief discussion of the relevance of these paradigms to this thesis follows.

Perhaps Transaction Costs theory, the first paradigm of Barringer and Harrison (2000), is the best fit for the inter-organizational relationships due to information sharing. Using the simple problem settings (or “domain” as we use in this thesis) for examining the impact of information sharing on inter-organizational relationships of a supply chain, following the theoretical conclusions of many previous research works published, we find one of the ultimate motivations for the organizations (or firms) to share information is to lower the transaction costs. The previous findings mostly have pointed out that information sharing has positive impact on an organization’s transaction costs, thus an improved (closer) relationship between the buyer and retailer. It is natural to apply this theoretical paradigm to state that other than the impact on the vertical structure of the supply chain, information sharing between retailers at the horizontal level also plays a role in reducing the transaction costs of organizations.

Clearly, Information sharing can as well be rightfully categorized into the Resource Dependence paradigm. This theory is rooted in an open system framework, which argues that organizations must engage in exchanges with their environment to obtain resources (Scott, 1987). This theory is different from the resource-based view (RBV) (Mahoney and Pandian, 1992, Barney, 1991) which states that rare and difficult to imitate internal firm resources are key to the firm’s acquisition and maintenance of sustainable, competitive advantage, thus RBV’s focus is more internal, although some of the

important internal sources may be obtained from external sources. The resource dependence theory states, otherwise, that the resources must be obtained from external sources for an organization to survive or prosper.

Information is well regarded as a resource (Wade and Hulland, 2004) and its value (Howard, 1966) or “usefulness” may be evaluated based on certain determinants (or factors, namely, flexibility, payoff function, degree of uncertainty, and the nature of information system) (Hilton, 1981). Some information may be obtained internally, which complies with the definition of RBV, while other information may only be acquired externally from other sources or organizations through exchange or sharing. The latter type of information fits the characteristics of resource dependence paradigm.

Information sharing may not be applied in other paradigms in forming the inter-organizational relationships. For example, the Strategic Choice theory² studies the factors that provide opportunities for firms to increase in competitiveness or market power. The profit and growth are typically the major firm objectives that drive strategic behavior. The strategic alliances may be a better application of such paradigm.

3.2 General consideration of model design architecture

As we mentioned in the beginning of the chapter, we need to specify the domain or purpose of our agent based model. Simply put, the purpose of the model is to examine the interactions due to information sharing between the retailers in a simple 2 level

² The strategic Choice Theory was developed when industrial relations in the U.S. were changing rapidly and the then popular conventional theories, such as the Dunlop's systems model, were overly static and could not accommodate interactions. The theory was first found published by Kochan, T. A., et al. (1984). "Strategic Choice and Industrial Relations Theory," *Industrial Relations*, 23(1), 16-39.

supply chain. However, a full scale application with multiple suppliers and retailers may be implemented based on the results of this essay. That said, our implementation takes a small representative portion of a supply chain, as most current SCM studies do, and supposes a supply chain with one supplier and three retailers with one product. More retailers and more products may be added in the future research based on the similar principles as described below to examine impact of the multiple interactions among retailer agents to the system. Instead of assuming that all the retailers are identical as seen in most SCM literature, we allow each retailer to have its own decision process using a probability procedure (as later shown in the implementation, this procedure is a decision logic using the input from the simulation and deciding certain actions to be taken).

3.2.1 Environment

All agents are put into an environment (or space) where they can interact with others based on certain behavioral rules and design objectives. This environment provides the channel of communication between agents and may also include nonreactive objects (for example the input such as the demand. To keep the implementation simple and without losing the generality, the demand, as in the approaches found in previous research (including all that are cited in this essay), is regarded as independent and not reacting to the actions of the retailers and/or the suppliers.). It is convenient to route all communication between agents through the environment, not only because this is the natural way to do it, corresponding to the role of the environment in human affairs, but also because it makes monitoring the agents easier in implementation. It also means that

messages from one agent to another can be temporarily stored in the environment at runtime, reducing the likelihood that the results of the simulation will depend on the accidents of the order in which agent code is executed. This ensures that the messages are delivered to the recipients in the order as expected. And in fact this brings up the necessary and important component in the simulation: the time.

The only time unit (steady and uniform) is one period and each agent does one set of tasks at each period. The simulation proceeds as though orchestrated by a clock, tick by tick, while each period may include multiple ticks. At each period, all the agents are given a turn. Thus, time is modeled in discrete time steps.

To make it simpler without affecting the results of the simulation, we assume that agents all work full time and there are no breaks such as weekends and holidays. The simulation software environment is fully synchronized. This is very important because without synchronicity, the simulation may not run reliably.

3.2.2 Agent objectives and behaviors

The objectives of an agent may vary depending on the types of agents to be modeled. For example, the objective of a supplier is to minimize its inventory cost and deliver the products to retailers on time. However, the objective of a retailer may be different in a sense that it seeks not to lose sales by ordering excessive inventory. A real life example may be the currently hard to find Wii game console designed and manufactured by Japanese company Nintendo. The supplier has limited production capacity while the demand on the market is high. The supplier needs to carefully evaluate the true market

demand before it dispatch the products. The reason is simply for retailers, they have the incentive to exaggerate the demand and get a bigger shipment. In the most recent trip to the Target Store, I found the store had a big inventory of the console, while I found online the buyers complain about the delivery delays for the console.

The agent behaviors that we will model are in the form of rules. Based on these rules, an agent knows when to initiate an action and how to react when a request (an inquiry or a response) is received from another agent.

3.2.2.1 Agent objectives

The supplier and retailers may have very different design objectives due to their unique positions in the business relationship. The supplier holds the product, but it needs information from the retailers to know what the future orders are. Retailers order product from the supplier. However the shipment from the supplier may need a prolonged period to arrive. So the retailers may coordinate with each other their stock levels to reduce the chance of losing business due to insufficient stocks or to lower the inventory cost due to excessive product in stock. So the ultimate design objectives for these agents are:

- Supplier:
 - o Reduce the inventory cost (including overstock and backlog) by ordering carefully calculated quantity of inventory. This requires good communications with the retailers to obtain their (s, S) parameters.
 - o Deliver the product to retailers without delay
- Retailer:

- Reduce the inventory cost (including overstock and backlog). This can be achieved by ordering the inventory according to the forecast.
- When the stock reduces to a level too low, it should seek to get immediate help from other retailers at a premium to increase the stock.
- When the stock is too high, it may ship some of its stock to other retailers to reduce the stock.

(In the future research, we may add more dynamics to the system: when there is not sufficient inventory for all retailers, the supplier has the negotiation power to decide the price and quantity for each retailer. However, this situation changes when the demand is lower than expected and there are excessive inventory of product. The retailers may have more power in negotiating the price and quantities of the product to buy.)

The differences in the design objectives between the supplier and retailers determine the discretions in the interactions they engage in with each other.

3.2.2.2 Agent behaviors

- Supplier: as defined, a supplier provides products for the retailers to sell. We assume that these products can be made by the supplier without purchase of any components and, as a result, the supply chain under consideration ends there.
Also we assume that the supplier has unlimited capacity
 - The supplier receives orders from the retailers when their inventory reaches reorder level

- The supplier needs a fixed amount of time T_{mk} to make the product (any units) (future research may add the production capacity constraints to the supplier). The value of T_{mk} may be customized by a user at runtime
 - The supplier needs a fixed amount of time T_{tr} to transport the orders to retailers. The value of T_{tr} may be customized by a user at runtime
 - The supplier may ask the retailer to provide their (s, S) information and they may get this information in a full information sharing case. The supplier uses this information together with the demand information received from the retailers, by using the approach described in Lee et al. (2000), prepare the demand forecast and order the inventory for next period.
 - The supplier has its own demand forecasting model if the retailers (s, S) information is not available and the supplier will calculate its own (q, Q) values in each 30 day period based either on the (s, S) values from the retailers, or when that information is not available, on the forecast values it obtains. New production will start when the inventory is below q and stop when it reaches Q .
 - When inventory available is not sufficient to satisfy the orders placed by multiple retailers, the supplier needs to decide the quantity each retailer may receive. A simple supplier decision process is used.
- Retailer:

- Each retailer every day receives demand at the beginning of the day and it checks if it has sufficient stock. If it does, it will reduce the stock level by subtracting the demand amount.
- At the end of the day, it will check if the stock level is below a preset reorder-level value “ s ”, if not, it will do nothing, otherwise it will order more stock to order-up-to level “ S ” from the supplier.
- In addition to this reordering process, the retailer needs to have a long term forecasting of the average demand in the next 30 days. On the first day each 30 day period, it will adjust the values of “ s ” and “ S ” for that period based on that forecast.
- In addition to these activities, each retailer agent will communicate through messages with its peers when it is at lower-than- s stock level so that an immediate replenishment (from its peers) may achieve to avoid falling into the situation that it has insufficient inventory before the supplier’s shipment arrives, which may take several days.
- During the above communication, a retailer may respond with yes or no to the request from other retailers. The decision is made based on the retailer’s decision process. More details of this process may be found in the explanation of the implementation details.

3.2.3 Important Parameters

- Demand distribution: Demands arrive every day, which lower retailers’ inventory levels and increase profit account. Demand arrival can be of any distributions as

long as it is coded in the simulation implementation. Some commonly used distributions may include normal, uniform, and exponential distributions (Gavirneni, et al., 1999). Demand distribution parameters may be set as changeable so that a user can change the parameter values to compare the results under different demand distributions.

- Decision Process for Supplier / Retailer: Decision processes are used by every agent for making such decisions as whether to accept or reject a proposal from another agent. The supplier and retailers may use different decision processes because of their different positions in the simulation. More details may be found in the explanation of the implementation details.
- Demand Forecasting Process: Our implementation uses the Java engine to generate normally distributed random variables as the demand for each retailer. Then retailers than use the Lee et al. (1997) AR1 process to generate the demand forecasting for next period. However, for future research, there are some other forecasting models available, such as moving averages, exponential smooth function, etc.

3.2.4 The Architecture

A simplest and most basic type of interaction involves two parties, who follow the basic behaviors in communications: one party sends a request and the other sends a response back. Based on the information system IOP model (Zachman, 1987), the request from P1 (short for Party 1) is regarded as the input to the system (or sub-system) of such two players, and the input is processed by the request-receiver P2 (short for Party 2). P2

generates the response, which is regarded as the output. The output is sent back to P1 (this output is named as feedback in this essay following the information feedback model of Forrester 1959), who takes it as the input and decide whether further communication is necessary. When this process continues, the interactions between P1 and P2 enters a loop until one of them decides to interrupt the communication because a satisfactory result is generated. There is another possibility, which is that the P2's response is used as the output of the system (or sub-system) and becomes the input of another system (or sub-system). We show this interaction process in Figure 5. It is noted that we replace the P1 and P2 with X and Y because in the later Figures, we use X, Y, and Z to represent three retailers in a supply chain system. Because the above described process is common across the supply chain (system), and many other systems, we simplify it as an interaction process with one arrow (input) in and two arrows (feedback and output) out. As shown in Figure 5. With this simplified symbol the used-to-be complex flowchart of system processes become much more readable and possible to fit on one page as in Figure 6. Note that the interaction processors are created for flowcharting and programming purposes (in a program, the interaction processor can be implemented as a method in Java or function call in C and other languages, which makes the implemented program modularized and easier). Each interaction processor may be different in itself because, for example, the process (such as individual decision process and consequent actions taken) within may be very different and agent specific.

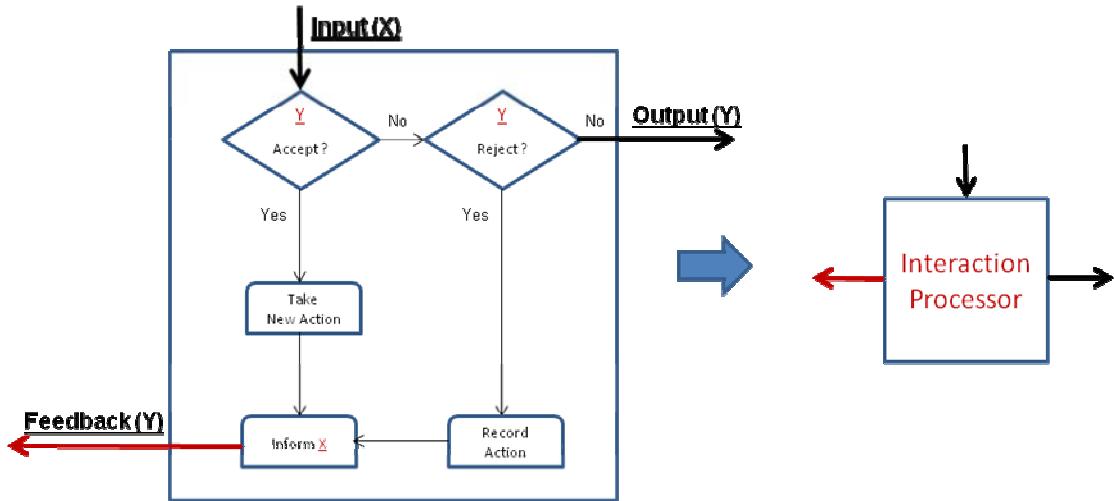


Figure 5 Interaction Processor

We may use the pseudo code to represent the logic of the interaction processor with explanation the follows:

```

//Given two agents x and y, and x initiates a proposal
function Interaction(x,y)
{
    proposal(x) accepted?
    If no, proposal rejected, than
        y prepares counter offer (if any) and send message to x: output(y)
    If yes, proposal accepted, than
        y takes new action, and send message to x: feedback(y)
}

```

The logic of the processor is very simple: to take the input from, say, agent x, and y processes it with certain output. Note that in the pseudo code, we have not included the necessary supporting functions to implement the decisions of y, nor have we provided the description of the functions `feedback(y)` and `output(y)`. This processor may be coded as a Java public method module in the implementation package that can be called by any agent. A common usage of the processor method is to put it in a loop: x sends a proposal to y and y (processor is called once here); and y sends the counter offer (the processor is

called again, but the input will be from y and x makes the decision); once x receives the counter offer, it decides, using the processor, its action, and if a new offer is formed, the loop continues. The loop may continue as long as necessary. However, it is not practical to do so for the following reasons. First, for synchronization consideration, since each period has a fixed set of ticks (time units, say miliseconds) and the number of ticks should be within reasonable limit so that the simulation will not run too slow, the interactions between any agents are bounded to the number ticks in a period. Second, in the real world, there is also a time limit as of one project should be completed with a limited time – the pressure of the timing may be much bigger than the simulation itself. However, we can not undermine the use of the interactiton processor. This method may be called many times and it will consume a significant amount of computing resource if it is not coded carefully. This may be briefly noticed in the flowchart in Figure 6.

One thing worth mention about the interaction processor is that this is a logical implementation and the simple interaction between two agents. Simply put, one agent takes input from another agent, processes the input, and outputs the result. This is a very typical IPO model in the information system literature. The output result is again used as the input and it is process and output to next... The implementation puts this in a loop. Depending on the implementation logic, the number of loops may vary. There exist numerous such interactions in a say, 3-agent environment as shown in Figure 6.

The flowchart in Figure 6 uses the interaction processors to simplify otherwise very busy interactions among three agents. In the flowchart, we assume there are three agents interacting with each other. Retailer X sends a request (proposal) of, for example

exchanging the overstock at certain price to retailers Y and Z. Y receives the request and will accept/reject it using the (implementation specific) decision rules in the interaction processor. Y may contact Z for additional stock. Same story happens on the Z side. We may find that loops with interaction processors are used in the flowchart. This chart may be expanded when more agents are included in the interaction.

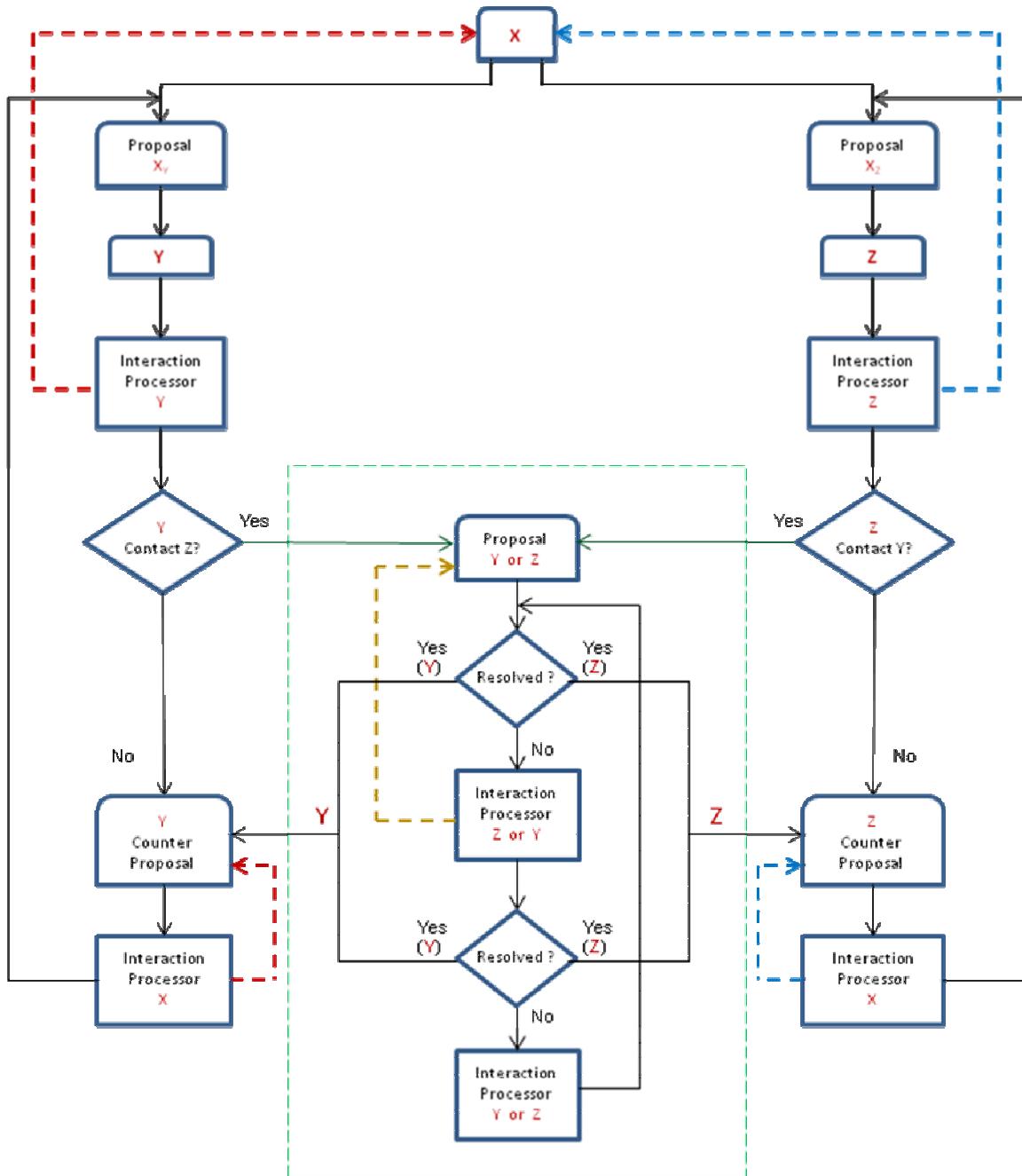


Figure 6 Flowchart of Interactions between retailers

3.3 Model implementation

Our model is implemented using language in Repast. Java is an objected oriented computer programming language that is suitable for modeling the individual entities. Details of the programming language features may be found in many textbooks and are not listed here. The essential characteristics of the object-oriented paradigm that are especially suitable for agent based modeling are embedded in each agent. For example, an agent is considered a class object and it possesses many attributes about itself such as inventory level, cost structure/variables, product library, etc. This agent also has many predefined behavioral rules for guiding the actions that the agent may take. Such behavioral rules may be, what to do when a customer demand arrives (the triggering events and the triggered behaviors), how to decide the next order level (the decision rules), when and from whom to replenish the inventory (the interaction and communication logic), etc. The implemented behavioral rules are also called functions or methods in the computer programming terms.

After the agents are coded in computer programs, they are placed in an environment to interact with each other. The publicly available simulation package Repast (version name, Simphony) is one of such implementation environments, which is also written in Java programming language and enables the processes of the agent based model. Details of Repast framework may be found from its own website (www.repast.com). What is worth of mentioning is the major benefits of using the Repast in this essay, which are that our programs written in Java are of fully compatible with the Repast framework (because Repast is in fact written in Java) and can be plausibly easily debugged and tested. We

also use the Java program development interface Eclipse, which has also been integrated with Repast. The combination of these tools gives us the possibility to develop the agent based model from scratch within a sensibly short period of time with the requirement of author's Java programming capability as well as the knowledge of agent based modeling.

The coded model runs within a specific domain or scope, that is,

- There is one product in the system
- There are three retailers and one supplier in the system. It is possible to add more retailers and suppliers to the system in the future implementation.
- Retailers are all equally distanced from the supplier, thus the transaction costs and time of transfer are the same for each
- Retailers are in different independent market segments so that they are not competing, though they sell the same product
- Demands in each location is independent and demand over time fall in normal distribution
- Retailers are not so far apart, thus the transaction time between them is negligible.

This assumption may be relaxed in future implementation.

- Information sharing between retailers and supplier is the demand information, which has been extensively examined in many previous research. This essay does not focus on how the information is shared, but the results. We use the approach in one of the previous research, in particular, Lee et al. (2000), for the external validation purpose. Information sharing between retailers is the stock (inventory) data including the overstock and backorders.

- When stock movement between retailers is necessary and there is a chance of its happening, the possibility of such stock sharing is calculated based on the relationship (high, medium, low) between the retailers.
- We assume that information sharing and relationship positively relate to each other, and to simplify, the links between them is linear. (It is possible to assume the exponential function will work too, as used in the friendship model in social study.)
- The relationship may also be predetermined by the contracts and negotiations between retailers which detail the terms and conditions of such relationship.
- The overhead of stock movement, including any transaction and transportation costs, at this moment, is set to a fixed and small (compared to the costs of overstock or under-stock) value for each time. Both sides (overstock and backorder) have the incentives to share the costs due to market fluctuations.
- In future implementation, such overhead may either be fixed by contracts; or vary based on the degree of the relationships.

There are four types of agents, namely, the demand agent, retailer agent, moderator agent, and supply agent. Each type of agent inherits SC agent³, which is the super (parent) class (a class which gives a method or methods to a Java subclass). The inheritance hierarchical structure is shown in Figure X1. The four types of agents are arranged in a framework as shown in Figure X2. The arrows indicate the inheritance relationships

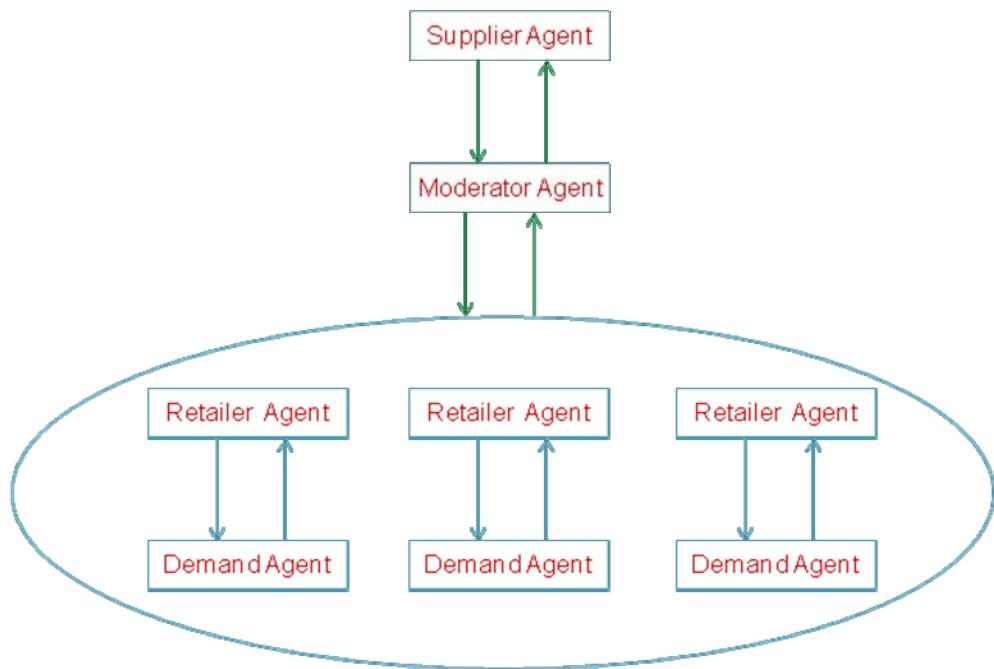
³ It is named as SC agent because this is the super class agent and it is the parent of all supply chain sub-class agents. The modeling approach follows the inheritance structure of object oriented programming language, which is considered one of the best fit for agent based modeling. More details about object oriented programming as well as Java programming language may be found at www.sun.com.

between the superclass SC agent and the subclass agents (A Java superclass is a class which gives a method or methods to a Java subclass.).



* The arrows show the inheritance relationships of the class objects

Table x1 Agent Class Objects of the Model



* The arrows show the direction of information and stock flow

Table x2 the Model

The demand agent is the simplest. The only task of this agent is to generate demand flow to feed into the retailer agent. There are three demand agents, one for each retailer. We consider external (customer) demand for a single product occurs at each retailer, where

the underlying demand process faced by each retailer is a simple auto-correlated AR(1) process. This approach may also be found in Lee et al. (2000), Kahn (1987, and Miller (1986). Let D_t be the AR(1) demand process at a retailer, where

$$D_t = d + \rho D_{t-1} + \varepsilon_t$$

Where D_t is the demand forecast at time t , D_{t-1} is the demand at $t-1$, d is a constant in AR(1), $d > 0$, ρ is the autocorrelation coefficient, $-1 < \rho < 1$, and ε_t is *i.i.d.* normally distributed with mean zero and constant variance.

The retailer agent is the seller to the customer. It takes the orders from the customer and reloads its inventory from the supplier. Since the retailer faces the customer and we assume there is no delay in delivering the product to the customer. The optimal order decision of the retailer follows the formula in Lee et al. (2000):

$$Y_t = D_t + \frac{\rho(1 - \rho^{L+1})}{1 - \rho} (D_t - D_{t-1})$$

Where Y_t is the retailer order quantity at time t , D_t is the demand forecast at time t , D_{t-1} is the demand at $t-1$, d is a constant in AR(1), $d > 0$, ρ is the autocorrelation coefficient, $-1 < \rho < 1$, L is the delivery delays (lead time) from supplier.

The full implementation programming code may be found in the appendix.

Chapter 4 Experimental Design and Analysis

This chapter implements the model as described in the previous chapter and analyze the results. There are two parts in the implementation. First we conduct an external validation by comparing the results of the model with those from the leading management journal (Management Science, namely). Then we conduct the experimental design and check the internal validity of the model.

4.1 External Validation

The encoded model needs validation. The task of validation, i.e., to show that the model is fit-for-purpose, is a complex one because the model is based on a gross simplification of the real situation and the experimental conditions are necessarily far removed from severe market or economic conditions. For example, the model does not allow for the complex behaviors of a real manager of the supplier who may stop or reduce supplying products to one of the retailers because of some social (for example, boycott) or political (for example, on different sides of the political agenda) issues between them. Also the experiments are necessarily at a much smaller scale than the real supply chain, as we mentioned earlier, where in a real market one supplier may have more retailers and one retailer may find multiple suppliers for an identical product under different brand names, involving one product rather than hundreds and thousands of products.

The model requires numerical schemes and contains uncertain market conditions. Thus verification, to check that the model has been correctly coded, and validation, to ensure

that the model is an adequate representation of reality, is required before great reliance can be placed on the results of the experiments. Verification is a relatively straightforward task, and our model has been carefully checked and compared to the other published (and thus assumed validated) model that we use for validation as we will explain in more details below.

Our verification and validation process uses the model descriptions and implementation of Lee et al. (2000), which shows within the context of a two-level supply chain consisting of a manufacturer (supplier) and a retailer and a non-stationary AR(1) end demand that the supplier would experience great savings when the retailer shared its demand information. Our model is coded based on the following setups and descriptions, which are very well summarized in Raghunathan (2001) and we borrow from it here.

1. Setup: A simple two-level supply chain model is considered, which consists of one retailer and one manufacturer. The external demand for a single item occurs at the retailer, where the underlying process faced by the retailer is a simple AR(1) process. Some of the notation and assumptions follow.
2. Model Notation
 - a. t : Time Period, $t = 0, 1, 2, 3, \dots$
 - b. D_t : Demand faced by retailer at time period t
 - c. Y_t : Retailer's order quantity at the end of time period t
 - d. L : Replenishment lead-time for the supplier and supplier
 - e. H : Unit holding cost per time period
 - f. P : Unit storage cost per time period

- g. ρ : The coefficient in the AR(1) demand function
 - h. d : The constant in the AR(1) demand function
3. Assumptions
- a. The AR(1) demand model: $D_t = d + \rho D_{t-1} + \epsilon_t$
 - b. ϵ_t is normally distributed with mean 0 and variance σ^2
 - c. $\sigma \ll d$
 - d. $0 < \rho < 1$
 - e. All shortages are backordered (backorder do not get back to the ordering process but are used for calculation the total costs)
 - f. The demand process is common knowledge to all the supplier and retailers
4. Additional assumptions when the model is used to validate the bullwhip effect
 (these conditions are proposed in Lee et al. (1997))
- a. Past demands are not used for forecasting⁴
 - b. Resupply is infinite with a fixed lead time⁵
 - c. There is no fixed order cost⁶
 - d. Purchase cost of the product is stationary over time⁷
5. Information structure

⁴ Raghunathan (2001) argues that supplier may reduce the variance of its forecast further by using the entire order history to which it has access, thus information sharing may not have significant impact on the supplier's stock and costs. A further investigation may be made in future research.

⁵ the capacitated supply chain case may be introduced to the model in the future implementation following the Gavirneni et al. (1999) paper

⁶ this assumption may be relaxed because when it is very small, it does not make significant impact to the total cost

⁷ also, this assumption may be tested in the future implementation

- a. When there is no information sharing, the supplier receives only Y_t at the end of time period t from the retailer
- b. When there is information sharing, the supplier receives Y_t and D_t at the end of time period t from the retailer

The ordering decision of the retailer is

$$Y_t = D_t + \frac{\rho(1-\rho^{(l+1)})}{1-\rho} (D_t - D_{t-1})$$

Supplier's anticipated retailer order quantity for period $(t+1)$ is

$$Y_{t+1} = d + \rho Y_t + \frac{1-\rho^{(l+2)}}{1-\rho} \varepsilon_{t+1} + \frac{\rho(1-\rho^{(l+1)})}{1-\rho} \varepsilon_t$$

It is assumed that the supplier is aware of the fact that the demand process D_t follows an AR(1) process, with known parameter d , ρ , and σ . This assumption is considered reasonable by the authors (Lee et al. 2000) because such information about the underlying demand process can be communicated to the supplier through periodic discussion with the retailer, or the supplier can be provided with historic demand data from which such information can be readily deduced with sufficient accuracy. The authors implied that this is already the information sharing. Indeed, without the knowledge of such parameters, as shown in Gavirneni et al. (1999), the costs would be higher under information sharing when the supplier has no or partial knowledge about the underlying process. The latter paper has considered, as one of the three situations, where absolutely no information flows from the retailer to the supplier prior to a demand to him

except for past data. The author concludes that in this situation, the supplier suffers the highest costs, comparing to other two, namely, partial information sharing and full information sharing.

In the case of information sharing, the supplier knows both the retailer's order quantity Y_t , and the error term ϵ_t (through the sharing of information of D_t) when he determines the order for next period.

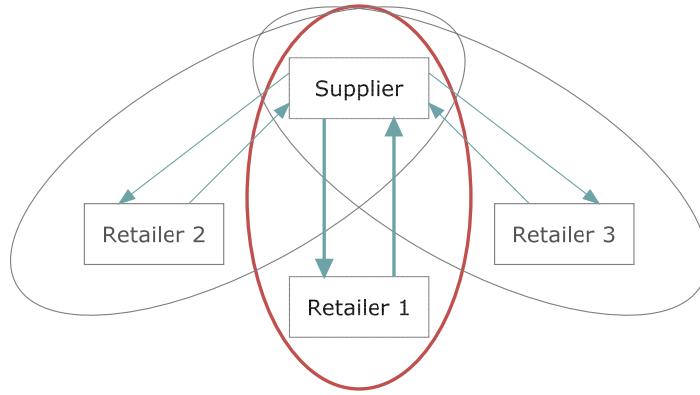


Figure 1 One Supplier Three Retailers: Three Pairs

Our validation model is illustrated in Figure 1. The underlying input is the independent demand for each retailer (not shown in the figure). The Lee et al. (2000) only consider one supplier and one retailer. In our model, we add another two retailers. This does not change the model dynamics since the supplier interacts with each retailer individually and the decisions of the supplier is made based on the individual interaction. In another words, our model is simply three Lee et al. (2000) running once at a time. In the figure, we use the oval to make the separation clear. It is assumed that each individual keeps his information locally and no global information is available. When information sharing

occurs, one individual shares his partial or all local information with another. This assumption is implied in Lee et al. (2000) and is important to be clarified for the implementation (programming) purpose.

Since the three groups work identically (except the parameters may be initialized with different values) and there is no need to examine all separately, we only look into one of them as shown in Figure 2.

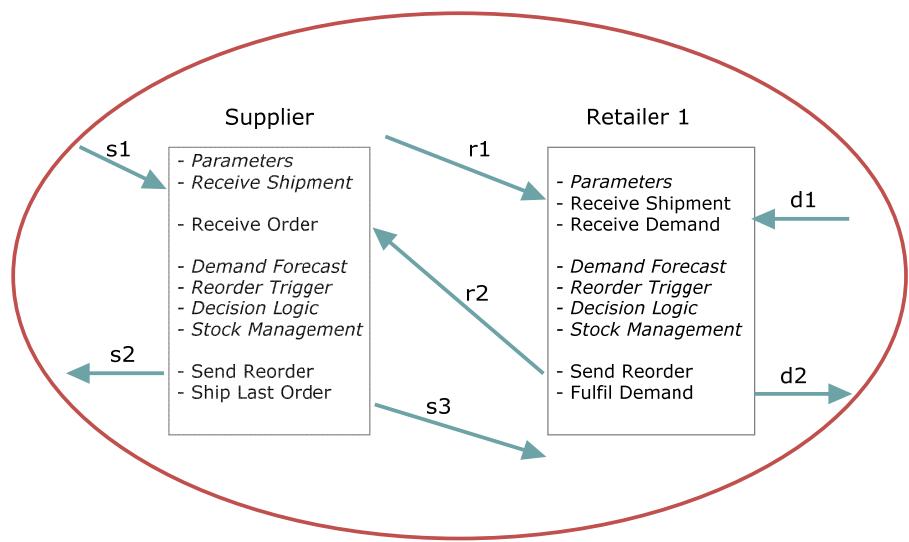


Figure 2 One Supplier and One Retailer: One Pair

Figure 2 is the model as described in Lee et al. (2000). Detailed explanations of the symbols and processes follow. The essential interactions between the two entities and the processes within each entity are similar for those with or without information sharing (readers are referred to the explanation for Figure 5 - Interaction Processor in Chapter 3).

d1: Demand arrives at the retailer. It is assumed the demand follows the AR(1) process.

All the historical and current order information is stored in the memory of the demand agent. This information is local to the demand agent only.

r1: Initialization. Retailer initializes the model run time parameters for retailer from the design, receives the demand for the period from the demand agent, and receives the shipment of his order of last period (It is assumed in our model implementation that lead time $L = 1$). All the information (parameters, demand, shipment) are stored in the local memory of the retailer agent.

s1: Initialization. Supplier initializes the model run time parameters for supplier from the design, and receives the shipment of his last order from the upper stream. It is assumed in our model implementation that lead time $L = 1$. It is also assumed that the upper stream has unlimited capacity. (In fact, the value of information in capacitated supply chain may also be modeled in the future, such as the one that is described in Gavirneni et al. (1999)). All the information (parameters, demand, shipment) are stored in the local memory of the supplier agent.

r2: After the retailer sets up the initial parameters, receives the demand and shipment, it activates the module to check if the current stock is sufficient for the demand as well as the predicted next demand (the forecast algorithm also follows the AR(1) process). The result may trigger another module to decide the quantity of the order for next period. All these modules are in the stock management sub-system of the retailer's decision logics. If no order will be placed, a zero order will be recorded. It is assumed that no negative order is allowed. Again, the retailer keeps all the information received and computed in

its own local memory. The cost calculation module reports the total costs for the current period, including the overstock (stock holding cost), under-stock (i.e., cost of backorders), and stock purchase costs. When everything is completed, the retailer sends the reordering information to the supplier and delivers the product to fulfill the demand. In the experiment of information sharing, the retailer also sends the demand information to the supplier.

d2: The demand agent receives the delivered product and saves the information to its memory.

s2: This is the continuation of S1. The supplier receives the order just placed by the retailer and uses its own stock management sub-system to check the current stock level for this and next period (forecasted) demand from the retailer. The forecast algorithm theoretically can be of anything, such as moving average, exponential forecasting, economic ordering quantity method, etc., but for the validation purpose, following the description of Lee et al. (2000), we use the given formula described in Chapter 3. The results from the stock management process may trigger the reorder process and the decision logic will decide the quantity of the reorder for next period. A no order will be recorded as zero for book-keeping purpose. Again, the supplier keeps all the information received and computed in its own local memory. The cost calculation module reports the total costs for the current period, including the overstock (stock holding cost), under-stock (i.e., cost of backorders), and stock purchase cost. A fixed ordering cost is added if an order is placed. When everything is completed, the supplier sends the reordering information to the upper stream.

s3. Supplier ships the order to the retailer. The retailer will receive this shipment in the next period and the process described above repeats. As we assume that lead time for both the supplier and retailer is $L=1$, the retailer usually receives his order placed two periods ago. When the lead time is larger, which may be true in the real situations, the retailer's delivery schedule may be stretched much longer.

The demand agent is the simplest. The only purpose of the agent is to generate demand for each period. It takes the initial parameters including the initial demand and the input and use the AR(1) process to calculate the output, i.e., the demand value for the current period. It is noticeable that the demand generation may be customized to fit the specific implementation if necessary. Figure 3 shows the simple flowchart for the demand agent. Once the demand is generated, the retailer takes it as the input. And the flow of information and product moves on to that as shown in Figure 4, the detailed implementation of the retailer in the model.

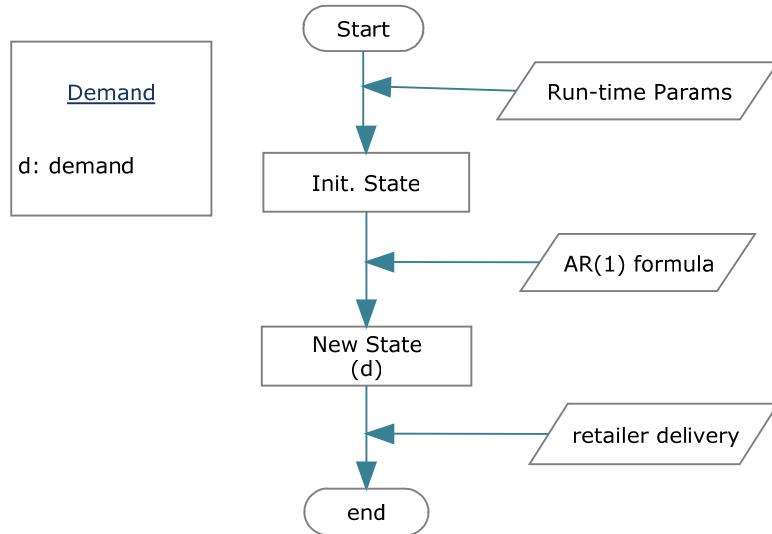


Figure 3 Demand Agent Process Flowchart

There are three retailer agents and they all run at the same time with the same pace (synchronized). We will only use one of the three to make the explanation because they are identical runs except they take different parameters and other data inputs. The retailer agent first, as the demand agent, receives the run-time parameters to initialize to the start state with the current stock and backorder information retrieved from his local memory. When the initial state is established, the agent updates its state to state A after the demand of the period from the demand agent and shipment (shipped last period) from the supplier is received. The shipment is added to the stock and all other information (run-time parameters, backorder, demand) is kept as the state variables to be used in the stock management sub-system. The stock updating logic is very simple in this validation process, following the description of Lee et al. (2000), which is shown in the code below. Decision logic determines the stock level and the backorder level, based on the values of the demand received.

```

stock -= totalRcvdOrder;

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

    supplied = totalRcvdOrder - stock;

}

else if (stock >= 0){

```

```
backOrder = 0;

supplied = totalRcvdOrder;

}
```

Then the new state B is established with the newly updated information. The agent now delivers the product to the demand agent (the customer) at the quantity determined from the previous state, and enters state C by computing the quantity of the current reorder to be sent to the supplier. If the calculated value is zero, no order will be sent and a book-keeping will be made to keep all the historical records.

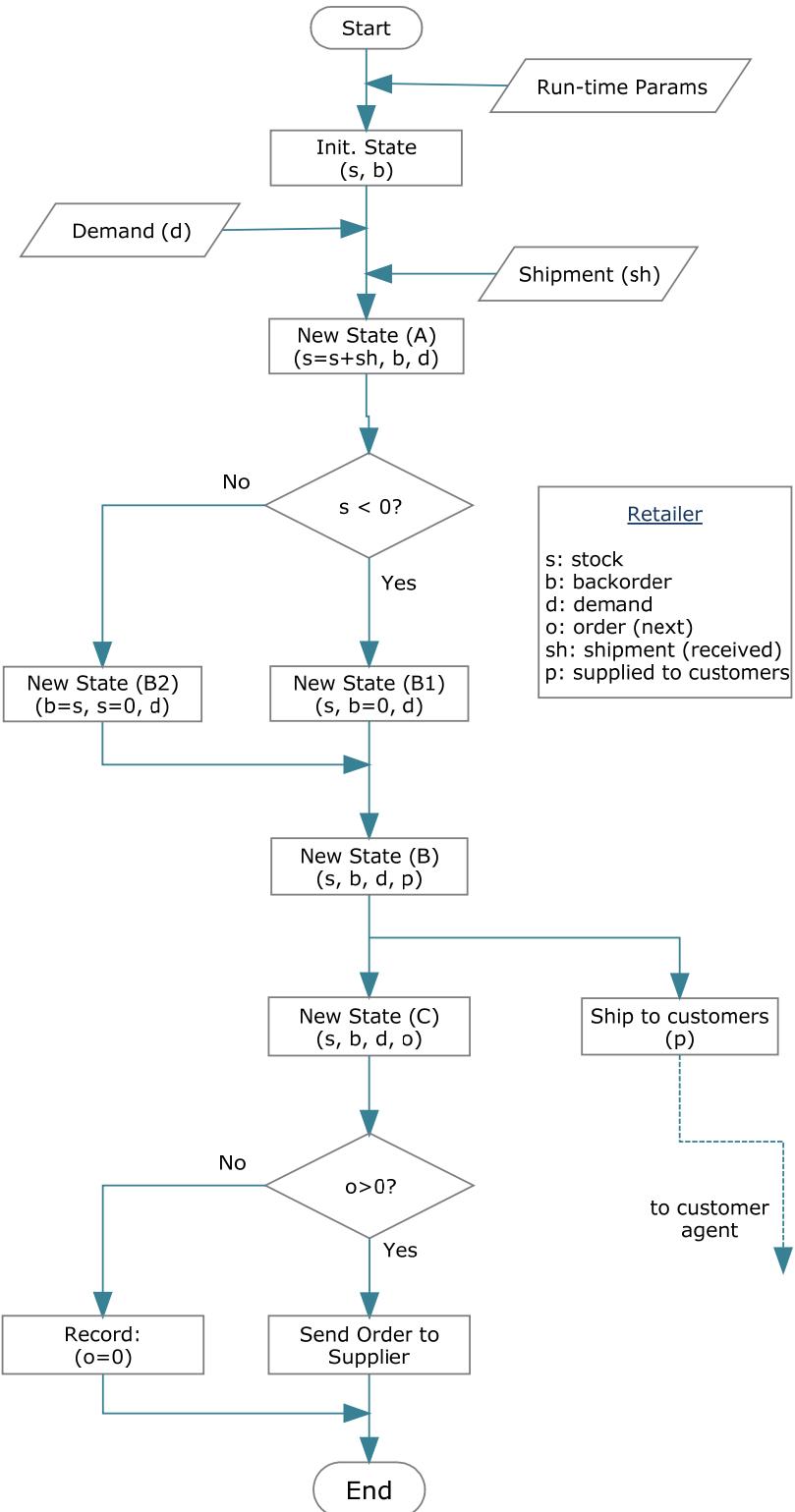


Figure 4 Retailer Agent Process Flowchart

When the period starts, the supplier starts at the same time as the retailers and it will end the same time as the retailers, too. The Repast's built-in JAVA multi-thread multi-task synchronization mechanism ensures that the periods start and end at the exact same time and commands are executed when expected. This synchronization is critically important for the precise and full control of every detail of the processes running at the run-time. If synchronization is not warranted, the retailers may, for example, not know the order received from the demand agent is from which period, or the supplier may, for another example, delay the shipment to the retailers certain periods. The fallouts of such abnormal behaviors of the model executing environment will totally void the purpose of the agent base modeling because the user would not have any control of the program run-time and could not justify the validity of the results from the model, no matter how many times he runs and how consistent (or reliable) the results may be. Figure 5 draws the flowchart for the supplier. From the flowchart perspective, we may find that Figures 4 and 5 are similar; however, the implementation code is different for each because the underlying logic for interaction process as well as the decision logic may be very different.

Similar to other agents, the supplier agent first picks up the run-time parameters and retrieves the stored stock and backorders (including the backorder for individual retailers) information from its local memory before it enters to the initial state. When the initial state is confirmed, the supplier receives the current (individual) orders from the retailers and the shipment for the order placed in last period from the upper stream. As we have assumed that there is no production or inventory capacity limit at the upper stream, the supplier receives the quantity exactly as ordered.

When the new state A is entered, the supplier has all the information for managing the stock. The individual orders are aggregated and the same procedure as the retailer is used to determine the current stock level as well as the possible backorder quantities. The newly computed information brings the supplier to the state B in which the supplier is armed with all the information for computing the order for next period, as well as the available stock for delivering to the retailers. When the stock is sufficient, all retailers will get what they have ordered. However, when the stock is low due to the variation of the demand and fluctuation in the calculated demand, the retailers are shipped the quantity proportionally of the available stock using the calculated fulfill rate. The model supplies the Lee et al. (2000) functions (as shown earlier) for calculating the reordering amount. The new state C is the final book-keeping and reorder finalizing state. The supplier now saves all the state information to the memory and sends the order to the upper stream. This completes the period and the new period repeats from the next tick of the clock.

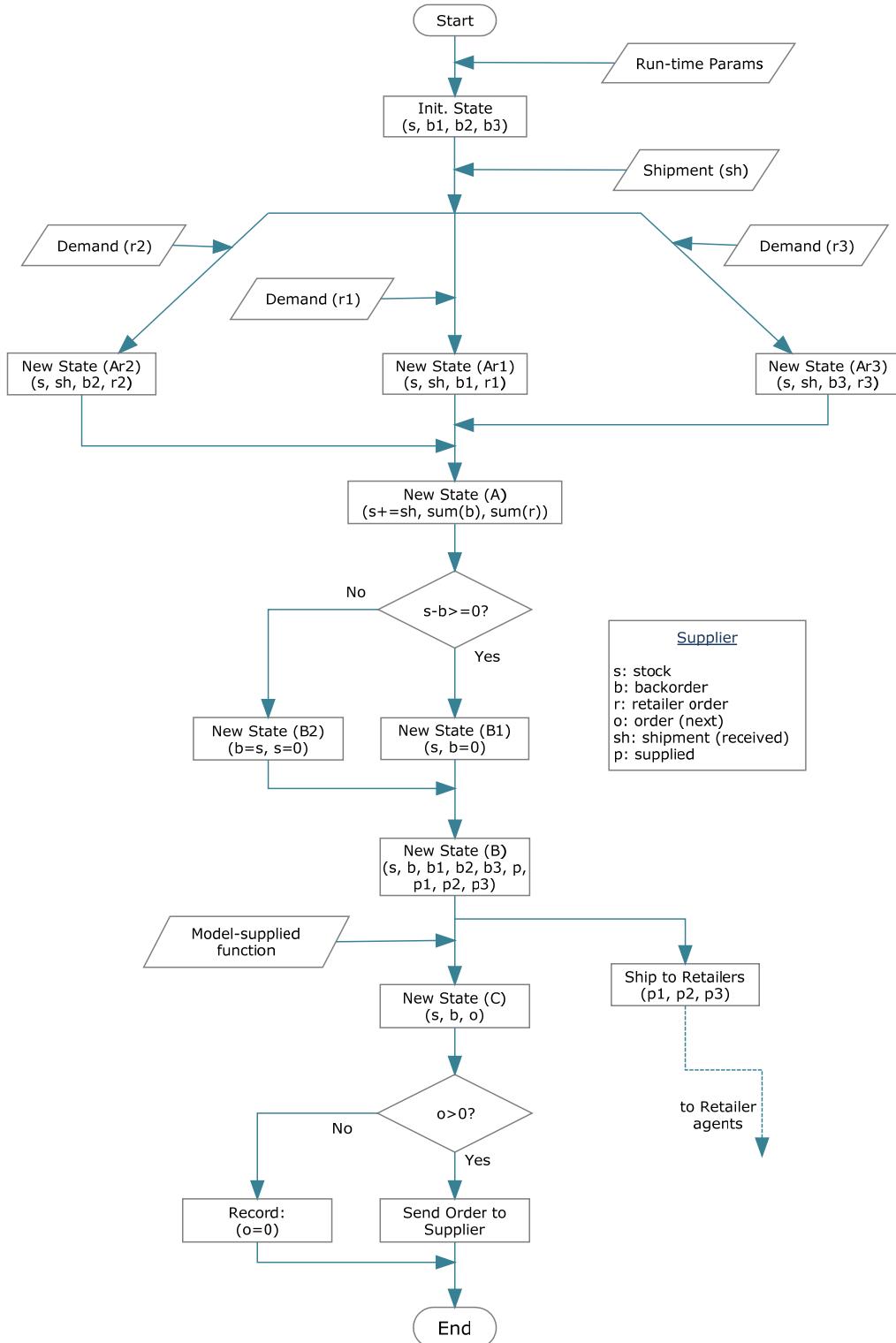


Figure 5 Supplier Agent Process Flowchart

4.1.1 The Results

Following the process in Lee et al. (2000), we vary the AR(1) demand process coefficient ρ from 0 to 0.9 to examine the impact of ρ . With the same parameters given in the paper, we generate the random demand for 2000 consecutive time periods and we compute the simulated average actual (on-hand) stock levels for the supplier. The intention is to analyze the impact of information sharing on inventory levels, inventory reduction, and inventory costs.

Figure 1 Impact of ρ on Average Manufacturer's On-hand Inventory

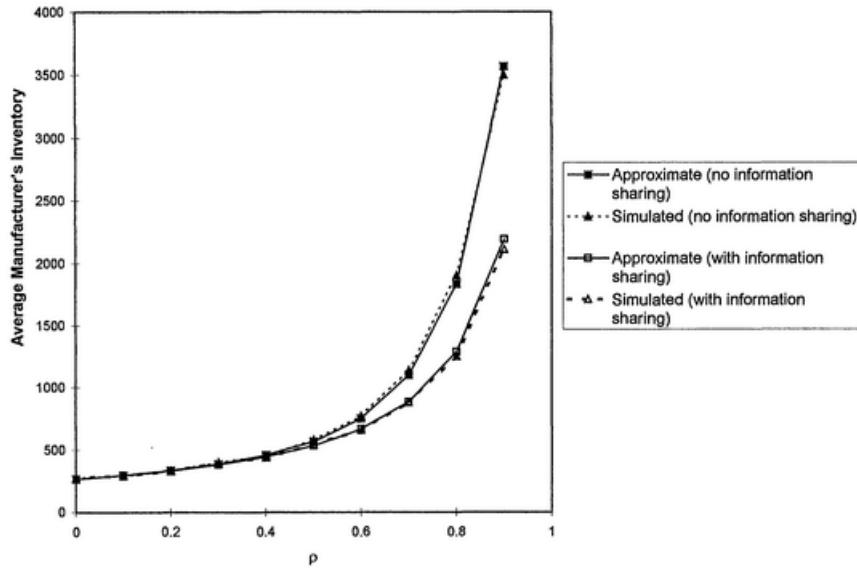


Figure 6 Impact of ρ on Average Supplier's Stock, Adapted from Lee et al. (2000)

Figure 6 is adapted from Lee et al. (2000). It reports the approximated and simulated average supplier's stock when ρ varies from 0 to 0.9, with and without information sharing. It is observed that the average supplier's inventory increases as ρ increases. The results that we have obtained from the agent simulation shown in Figure 7 is almost identical to those of Lee et al. (2000)'s. We easily can observe the upward trend of the

inventory levels as the value of ρ increase, and the average level of stock without information sharing is higher than that with information sharing. This figure shows the impact of ρ on the aggregate average stock level at the supplier side. The Lee et al. (2000) result is based on the interactions between one supplier and one retailer, while we have three retailers. We also check the impact to ρ on the stock specifically reserved by the supplier for each individual retailer. The figures 8 through 10 that follow show such investigation. Very similar, if not identical, to the results from Lee et al. (2000), we find the pattern we just described: upward trend of the average stock levels when ρ varies from 0 to 0.9; and information sharing does have impact on the stock for each individual retailer that reduces the average stock level. The consistency may be further found when we examine the percentage stock reduction and stock costs.

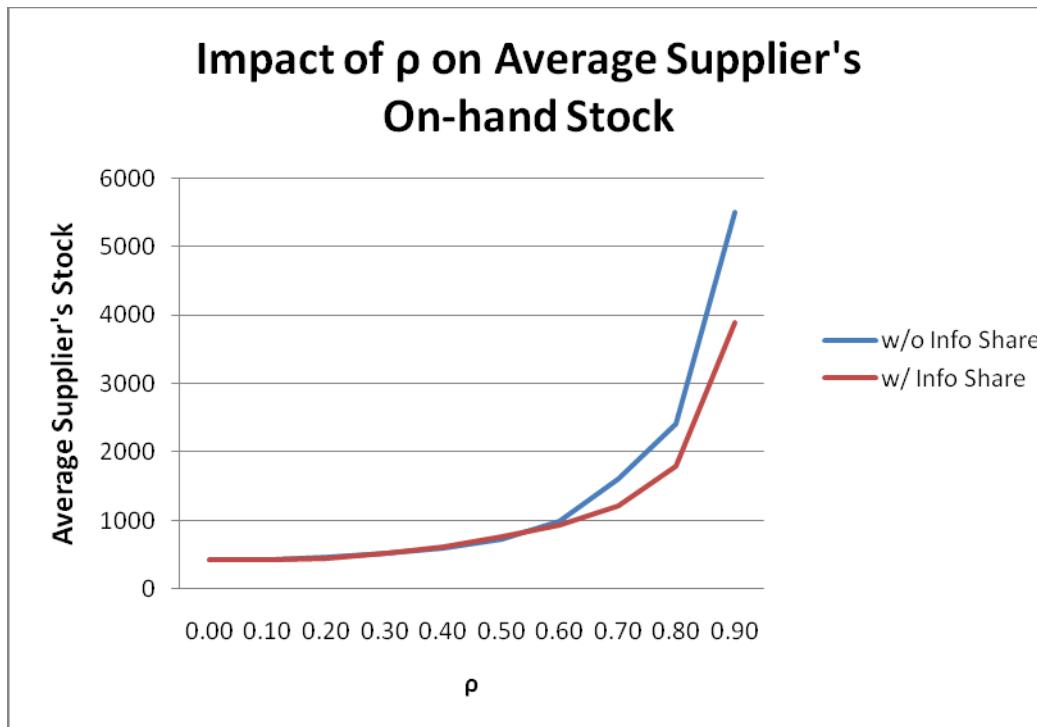


Figure 7 Impact of ρ on Average Supplier's On-hand Stock

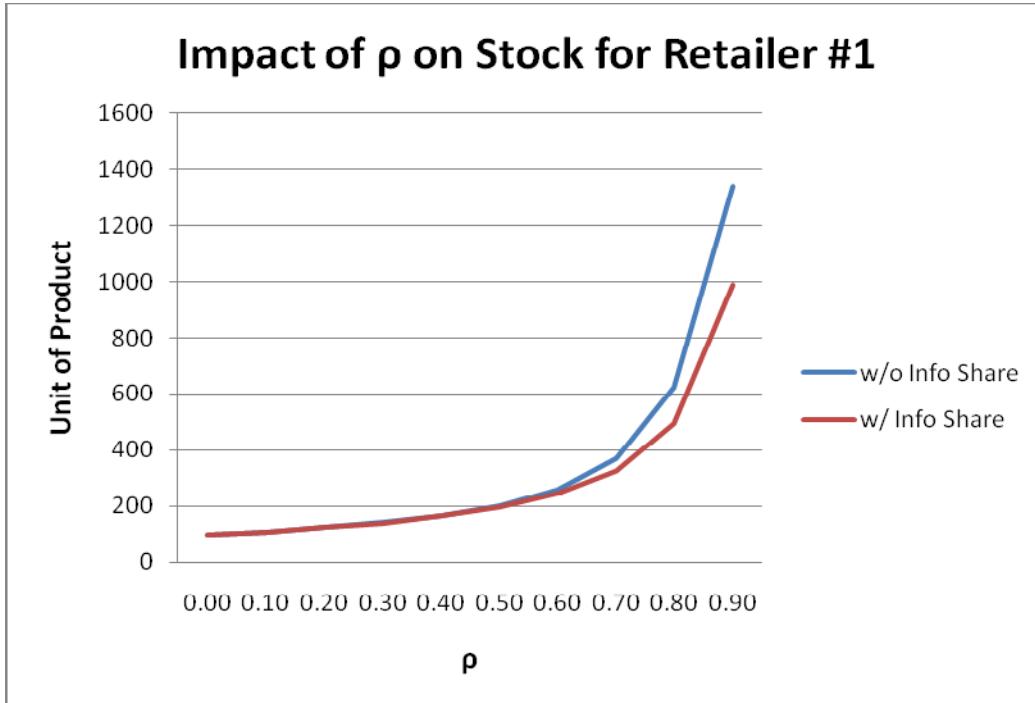


Figure 8 Impact of ρ on Stock for Retailer #1

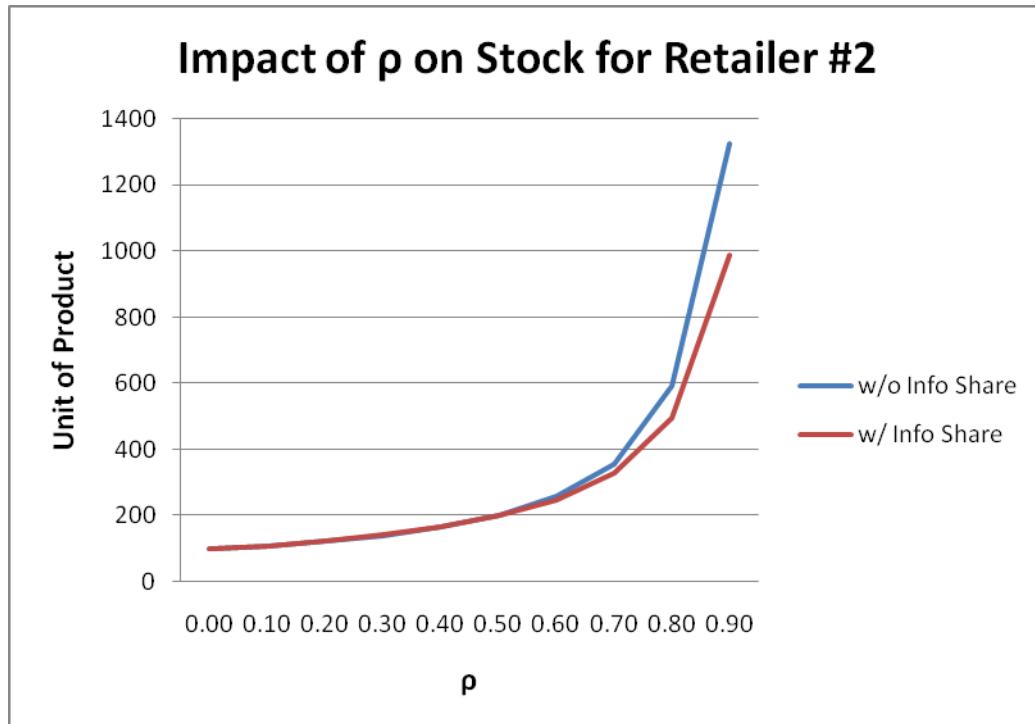


Figure 9 Impact of ρ on Stock for Retailer #2

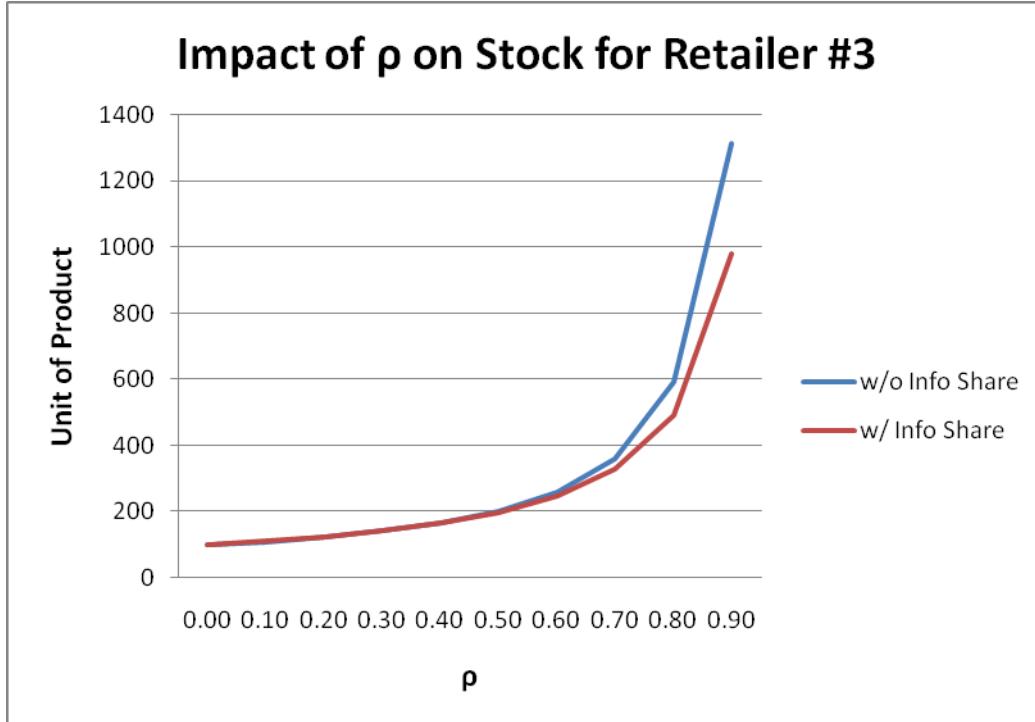


Figure 10 Impact of ρ on Stock for Retailer #3

Figure 2 Impact of ρ on Average Cost

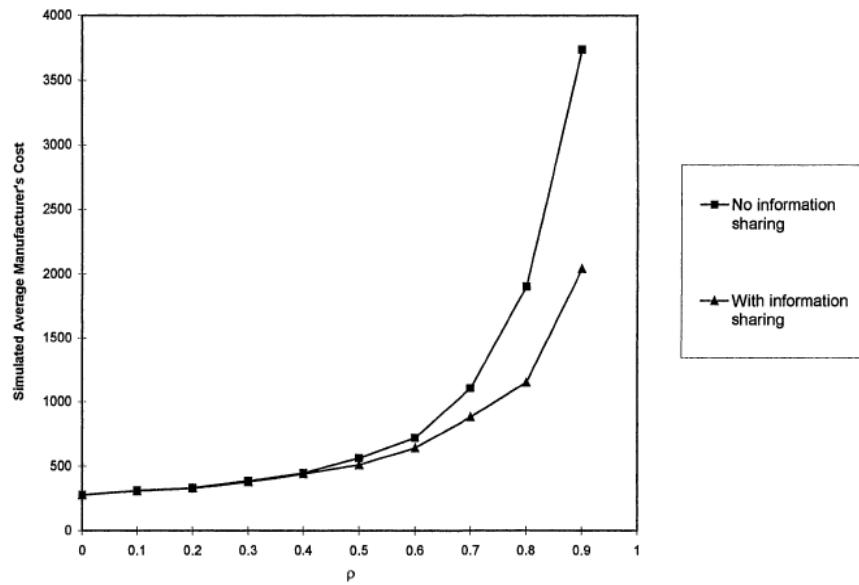


Figure 11 Impact of ρ on Average Cost

Figure 11 is adapted again from Lee et al. (2000), which shows the similar to Figure 6 observation for the impact of ρ on the average cost with and without information sharing. Figure 12 is the result from the agent based model. We once more see the consistent pattern of the impact of ρ as well as that of the information sharing.

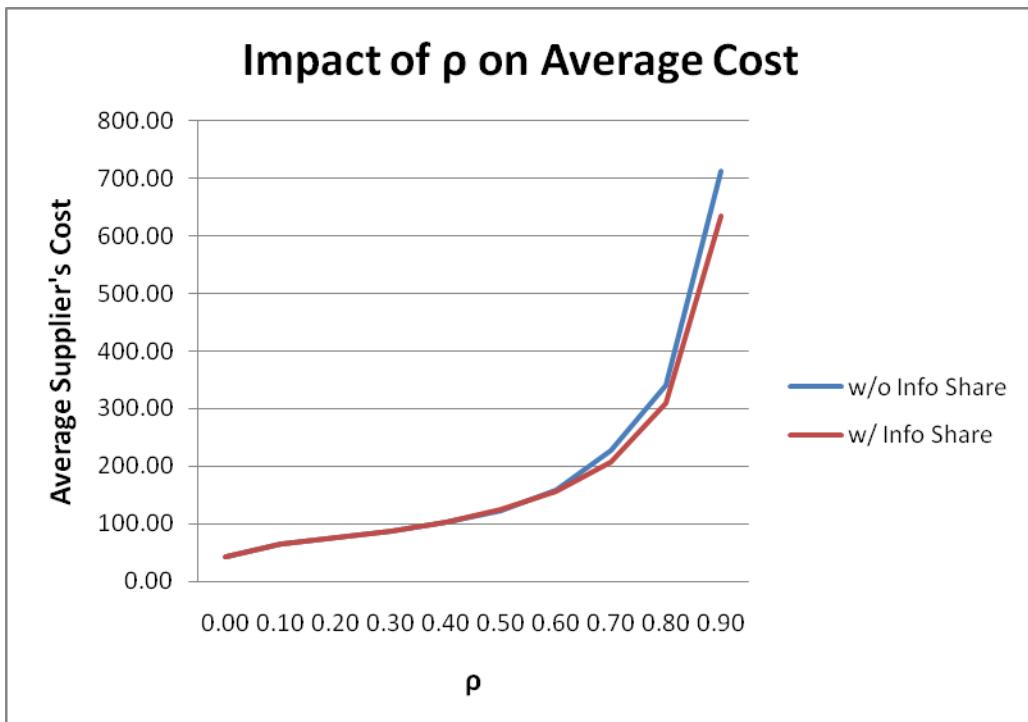


Figure 12 Impact of ρ on Cost Reduction

Figure 13 is adapted again from the same paper, which reports the percentage of stock reduction from information sharing. Figures 6, 12 and 13 all suggest that information sharing enables the supplier to reduce average stock, and this stock reduction, both in absolute terms or as a percentage of stock, is greater when ρ is greater. In fact these phenomena are consistent with the analytical findings as presented by the authors. Our results in figures 7, 12 and 13, as well as figures 8 through 10 are consistent with the

authors' findings. Thus we are confident that the model coded produces meaningful results.

Figure 3 Impact of ρ on Percentage of Inventory Reduction from Information Sharing

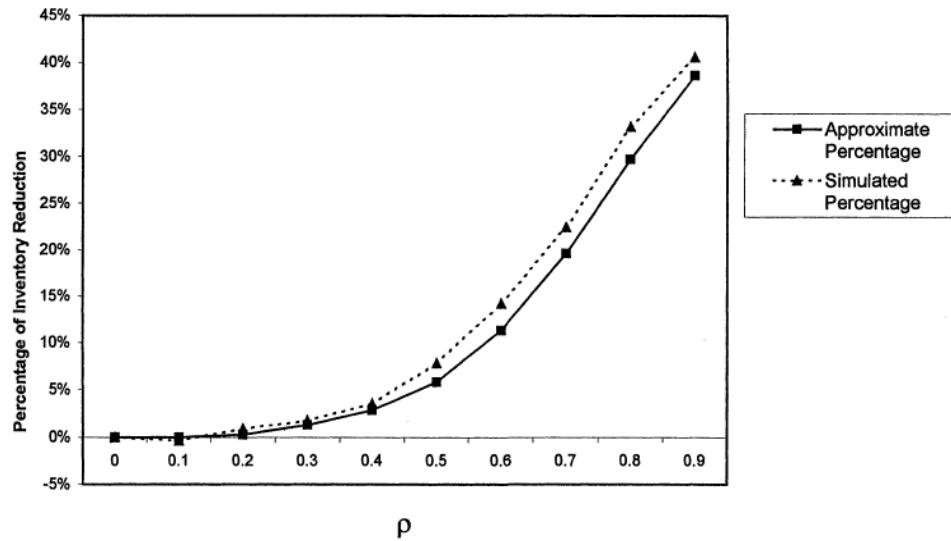


Figure 13 Impact of ρ on Percentage of Stock Reduction from Information Sharing

Lee et al. (1997) study the bullwhip effect phenomenon, i.e., the variance of order may be larger than that of sales, and the distortion tends to increase as one moves upstream. Four sources of bullwhip effect are analyzed, namely, demand signal processing, rationing game, order batching, and price variations. To make the case simple, we only use the agents to examine the impact of demand signal processing to the bullwhip effect. That is, the supplier receives true demand information from the retailer when information is shared. We will compare the results against the findings of Lee et al. (1997).

Figure 1 Orders vs. Sales

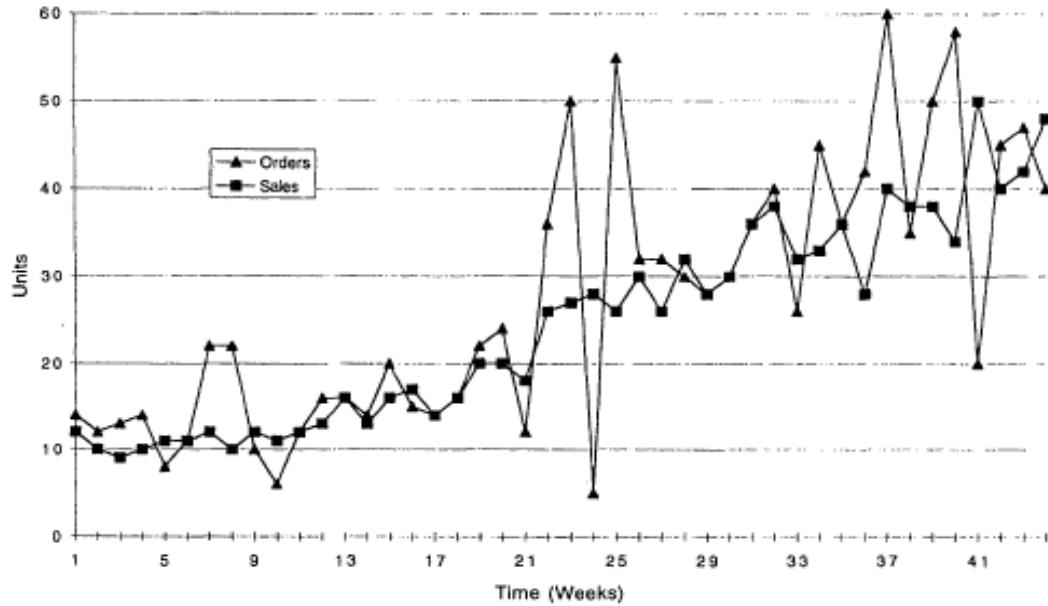


Figure 14 Bullwhip Effect: Orders vs. Sales, Adapted from Lee et al., 1997

Figure 14 depicts the authors' exemplary demonstration of the bullwhip effect in the market place based on real data but manipulated to maintain confidentiality, which shows a retail store's sales of a product, alongside the retailer's orders issued to the manufacturer. The figure clearly highlights the distortion in demand information. The retailer's orders do not coincide with the actual retail sales. In the real case as depicted in the figure, we cannot "assume" that the retailer knows the demand distribution and process (such as the AR(1) process in our case) and all the parameters that we use are not known to the retailer, the information distortion is obvious from the very end of the supply chain (the customers demand vs. the retailer's sales forecast). However, this is not the case in our agent base model settings, which follows the settings in Lee et al. (2000). In our case, the retailer knows the demand process that is AR(1) and it has all the

information about the parameters. So, to make the cases comparable, we move the supply chain up and try to find the information distortion at the next level between the supplier and retailers.

The results are depicted in Figures 15 and 16.

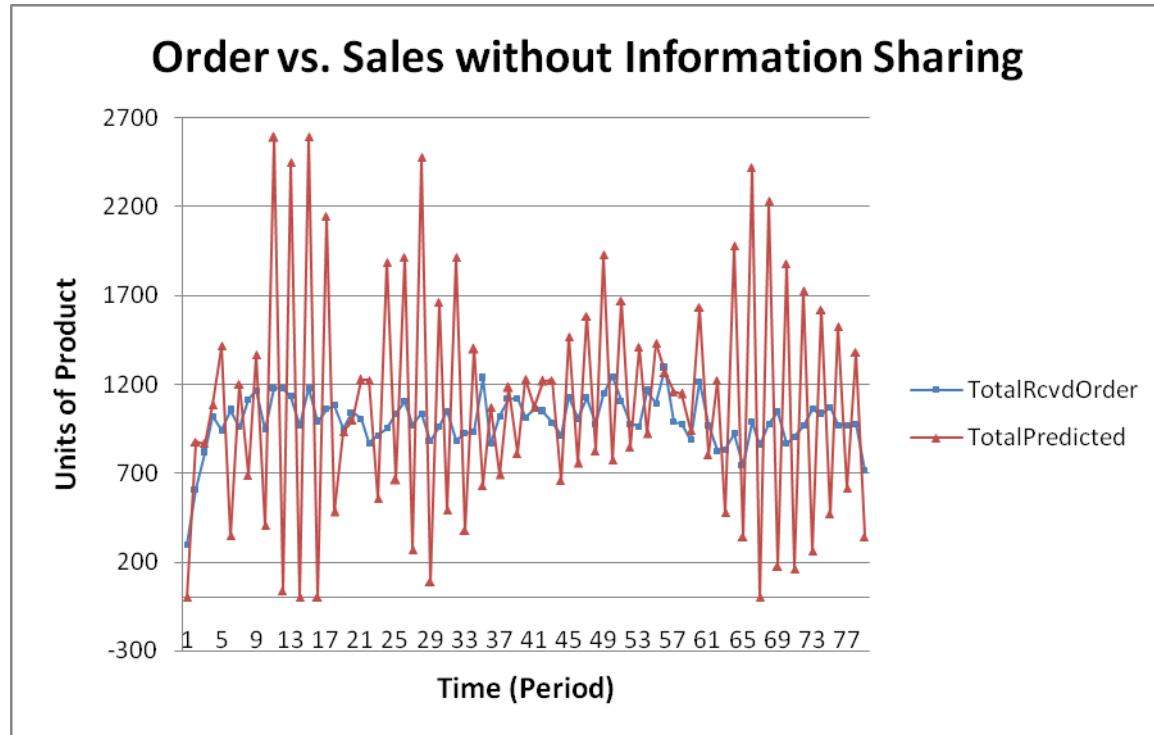


Figure 15 Order vs. Sales without Information Sharing

Figure 15 simply recreates the bullwhip effect very nicely. It is obvious that the blue demand from the retailers is fluctuate much less than the forecasted quantities by the supplier. The only difference between this figure and the Lee et al. (1997)'s is that the retailer order's trend does not go up as the demand from the customers does not, per the settings of the model parameters. The reproduction of the bullwhip effect gives us the

confidence that the agent base model that we have created is producing the appropriate results.

As Lee and Whang (2000) suggest that information sharing is one of the solutions that has positive impact on the bullwhip effect, i.e., to reduce the variations in the forecast. Figure 16 shows the support to their suggestion.

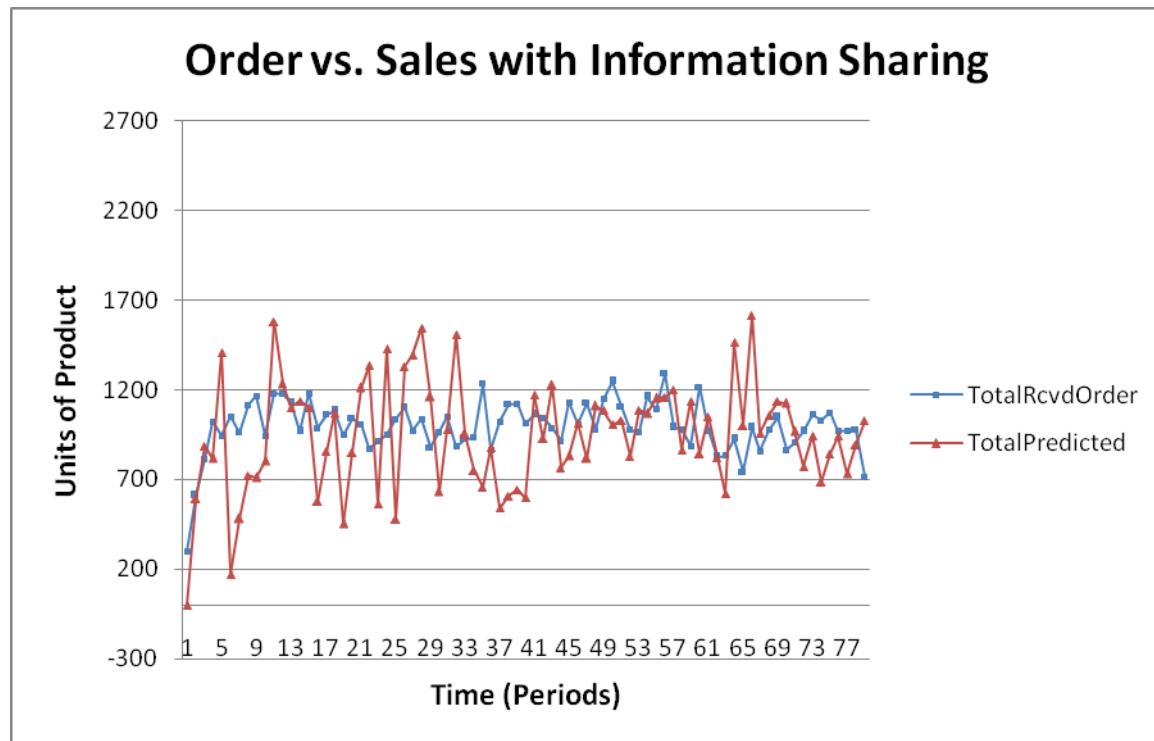


Figure 16 Order vs. Sales with Information Sharing

In this figure, the total orders from the retailers are the same since we repeated the experiment based on the exact same parameters⁸. An obvious improvement is found as the variability of the forecasts is much smaller comparing to the line in Figure 15.

⁸ This is exactly one of the benefits of using agent based modeling method as mentioned earlier as we can recreate the same scenario precisely as many times as needed.

The above experiments concludes the validation process and from the careful experiments and comparison, we conclude that the agent based model that we have created is sound and works as expected to create valid and reliable results.

4.2 Experimental Design (Internal Validation)

Now that we have inspected the validity of using agent based modeling to examine the impact of information sharing to the performance of a supply chain, we make some modifications to our model to investigate the effects of the interactions between the retailers to the supply chain. The external validation process explained above is simply a special case in our interaction experiments, i.e., the interaction coefficients between the retailers are zero, and all the information and material flows between the retailers and supplier pass through the moderator, thus we only study the one to one relationship between supplier and retailers. With interactions, using the moderator, the information flows are managed by the moderator and the supplier no longer interacts directly with the individual retailers, but the moderator. This is shown in Figure IM-1, which uses the dotted lines and faded color to differ from the model in the originally proposed (Table x2). We use the moderator in our internal experimental design, i.e., the validation process.

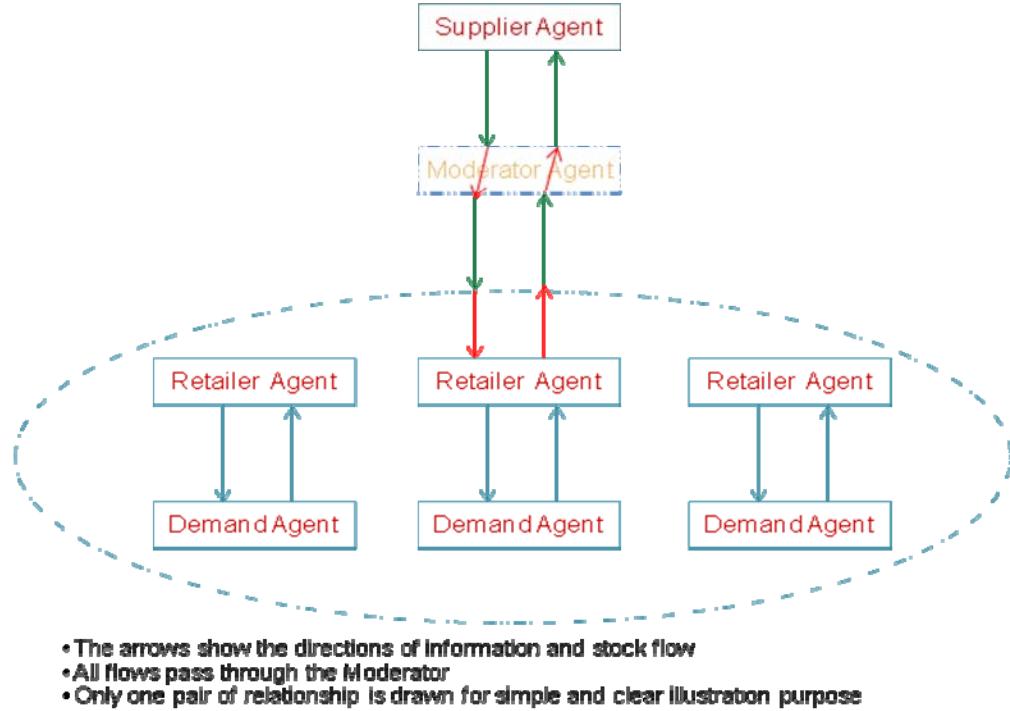


Figure IM-1 The Pass-through Model

The supply chain is a demand driven model and our agent based model follows the same logic. The volatility level of the demand (the variance, in fact we use the standard deviation σ as used in Lee et al. 2000) from the demand agent will have different impacts on the other end of the supply chain, i.e., the supplier agent. As the demand process is an AR(1) process in our model (demand processes of this form have been assumed by several authors (e.g., Chen et al., 2000, Lee et al., 1997), another factor may have effects on the stock and total cost of the supplier agent, which is the coefficient ρ in the demand generation function. Finally, as the agent based model has encoded the relationships between the retailer agents, the experiment design will also test what effects of the levels of relationships may have on the supplier performance.

Like the literature we have examined in the design of the experiment, we also assume that all levels in the supply chain (namely, the supplier and retailers) use the same demand

data, the same inventory policy and the same forecasting technique, which allows us to determine the impact of just the demand forecasting, as well as the interaction/relationships, without considering the impact of different forecasting techniques or different inventory policies across the levels. However, in the future research, we may examine the impact of different forecasting techniques, as well as the different inventory policies. Thanks to the modularizing features of agent based modeling, the new experiments may be designed in a much easier manner by simply adding more functional modules (methods) to the agents, or simply add some more agent types, comparing to the traditional procedural programming and mathematical programming which may require the re-writing of the whole implementation.

The experimental design uses factorial design of a 3x3 matrix that has three levels for the standard deviation σ and three levels for coefficient ρ as shown below. The dependent variable is the total inventory costs of the supplier for each period.

		σ		
		H	M	L
P	H	Test result	Test result	Test result
	M	Test result	Test result	Test result
	L	Test result	Test result	Test result

Where

H: High

M: Medium

L: Low

In the experiment, we setup three levels for each of σ and ρ . The values of these variables are:

		Retailer 1	Retailer 2	Retailer 3	Supplier
σ	H	180	140	100	140
	M	80	60	40	60
	L	30	20	10	20
ρ	H	0.80	0.80	0.80	0.80
	M	0.50	0.50	0.50	0.50
	L	0.20	0.20	0.20	0.20

Also the level of the relationships cor has three levels as show below (in fact we may treat this as the third dimension to the 3x3 factorial design matrix. However, it is difficult to visualize the 3D effects, so we break them down to sets of 2D tables with one parameter at fixed level):

		Retailer 12	Retailer 13	Retailer 23	Medium
cor	H	0.85	0.90	0.95	0.90
	M	0.55	0.60	0.65	0.60
	L	0.25	0.30	0.35	0.30

Varying the relationship levels (H, M, L) reflect different business scenarios as described below. The assumptions of the scenarios are the same as the description in the model implementation design, which are that in the market there only exist one supplier, one product, and three independent retailers. The demand for each retailer is independent of others, thus these retailers are not competing with each other. As argued in the well-regarded paper Axelrod and Hamilton (1981) published in *Science* and the later book “The Evolution of Cooperation” (Axelrod, 1984 and 2006), in a situation where there exist interactions, cooperative relationship will evolve. However Axelrod and Hamilton have not discussed the levels of such cooperative relationship, which define them in our implementation as high, medium, and low. In another words, when the relationship level is high, the chance (or probability) of helping each other is high; and when it is medium or low, the chances are medium or low.

- Scenario 1: This is the situation that the value of relationship may be low when the retailers do not often interact with each other for help or offering help. In such situation, without active interactions and more specifically, communications, information sharing is at minimum and the chance of exchanging stock information is low. Thus we set the value to 0.3 or lower, meaning, when needs rise, the agents only have 30 percent of chance to cooperate. It is possible when one retailer finds that getting help from another retailer may be more economically plausible, his request may be rejected due to the lack of communications and thus the adequate cooperative relationship between each other. This situation may be improved when interactions and exchange of information between retails increase, and cooperation will evolve.

- Scenario 2: The value of relationship is set to the medium range. In this situation, a concrete cooperative relationship has not been established. For example, the retailers have not setup the contractual and formal cooperative relationship, and a retailer is willing to help others because he gets help for the others too. This follows the Tit for Tat strategy of Axelrod and Hamilton (1981). The mutually informally agreed relationship is not guaranteed and the information sharing is volunteered. Thus we set the average value of the relationship to 0.6, meaning there is an average 60% of chance to cooperate.
- Scenario 3: The value of relationship is set to high. A concrete cooperative arrangement is found in this situation. For example, the retailers may have certain type of contractual relationship, such as strategic alliances, or other type of cooperative contracts. With the formal relationship, the interactions between the retailers are on the regular basis and information sharing is a usual matter. Thus when needs rises, the retailers will very likely help each other. So we set the value to 0.9, meaning, there is 90% of chance to cooperate. Note that even at the high relationship level, the chance of cooperation is not 100%. There are many explanations for this setting, one of which is that each retailer is assumed to be an egoist and to act rationally. For the details of

The data for the experimental design implementation will be filled in the table following as shown below:

$cor (0.30, 0.60, 0.90)$				
		σ		
		H	M	L
ρ	H	Test result	Test result	Test result
	M	Test result	Test result	Test result
	L	Test result	Test result	Test result

The test results in the table will be replaced with the data as the supplier agent average total costs. These values are the average values from the runs of the model for 150 periods. One table is created for each relationship cor value.

The runtime parameters include, in addition above values, stock cost per unit = 0.2, back order cost per unit = 0.2, purchase cost = 0.4, fixed order cost = 0.0.

There are changes to the program code for the implementation. Since the interaction will cause the changes to the back order level and in turn affect the overall cost at the supplier side, in our program we add the logic of handling the backorders. All the backorders at the retailers and suppliers are recorded and shall be fulfilled at the coming periods.

4.2.1 The results

Since we have implemented our factorial design at three dimensions: demand standard deviation, demand AR(1) process coefficient, and the relationships *cor*, we get nine matrices plus one without retailer agents' interaction matrix for comparison purpose.

We use the three relationship levels for the three scenarios as explained above. At each level of the relationships, we run nine sets of experiments. For example, at the low relationships, we have three levels of standard deviations σ and three levels of AR(1) coefficient ρ , which makes 3×3 sets. Each set of the experiments needs to run 200 times, from which we find the average total costs for the supplier. After all three levels of relationships are completed, we will have 3×3 tables for data analysis, one table for each level of relationships. Also for the comparison purpose, we run the base set of experiments that does not involve the relationship, i.e., the information flows pass through the moderator between the retailers and the supplier, as stated earlier. This is the base case of average costs when there is no interaction, which makes the four cases for shown in the charts that follow.

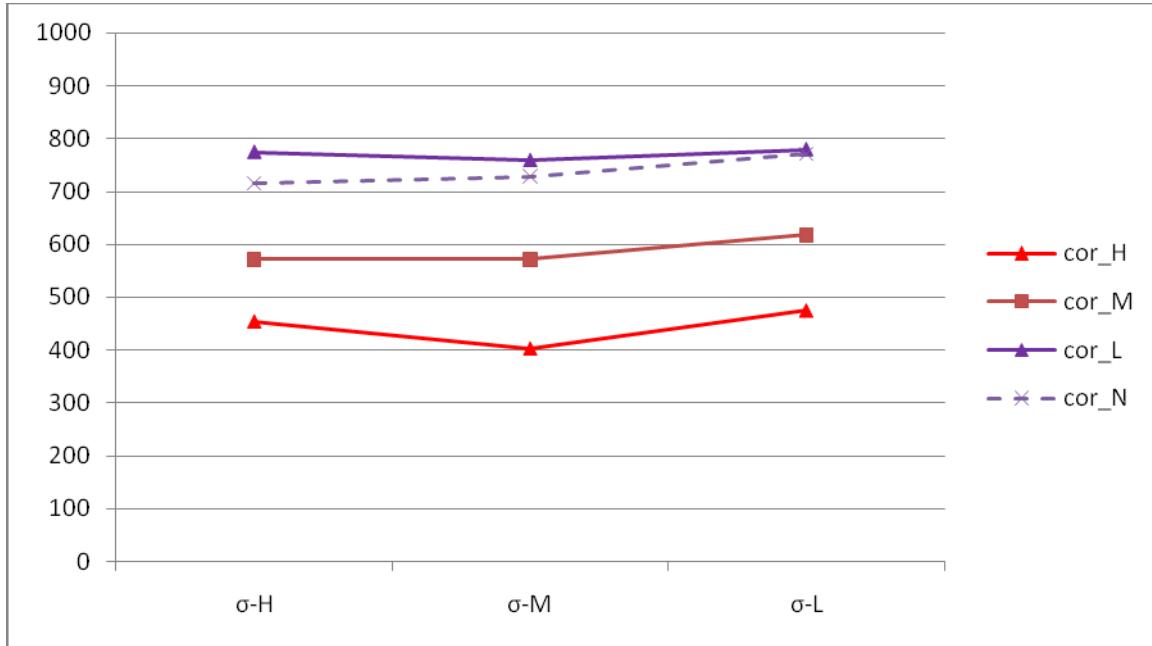


Figure 17 Effect of Information Sharing with Respect to Relationships and σ

The results are aggregated into one chart as shown in Figure 17. In this chart, at an average level of ρ (there are three levels of ρ and we average the results from these three levels and plot them there), the general conclusion is that the higher the relationship level between the retailers, the better cost savings we get. The lowest blue curve with diamond marks shows the significant savings at a high relationship level. The next lower level is the results from medium relationship, which ranges from about 550 to 650 and is much lower than the curve for the results from without relationship (the curve with x marks). The interesting curve is that from the low relationship level. It indicates the slightly higher total costs when retailers have only some interactions. This is mainly due to the results from the experiments that use low ρ and medium ρ , as shown in the charts that follow. We find that when relationship level is high between the retailers, the total costs are significantly lower than those without relationships.

The breakdowns of the impact of different level of ρ is showing in figures 19 to 21 When we revisit the Figure 11 adapted from Lee et al. (1997) which is shown in the external validation part above, we notice that when ρ is at lower ($0 \leq \rho \leq 0.4$) level, the average cost at the supplier does not vary very much and the curve is almost flat. We confirm this trend in our experiment shown in Figure 12. This indicates when the value of ρ is not very big the impact of ρ on the demand is relatively small (or none if $\rho=0$), thus the variation of the demand will be minimum to not significant. Therefore we may expect that the small variance at the demand end due to the nature of the AR(1) process will cause relatively small fluctuations at the supplier side. Since the two retailers will share the stock only at low or not so high levels as the possibility of such interaction is decided by the “IsShared()” test, when interaction is added, this may in fact cause the higher fluctuation of the demand seen from the supplier side.

“IsShared()” is a function embedded in the implementation that will decide whether the pair of retailers do interact given the relationship levels (H, M, and L) as well as the levels of current stock levels at both side. The yield possibility of interaction may be low when demand variation is low. More details may be found in the code in the appendix.

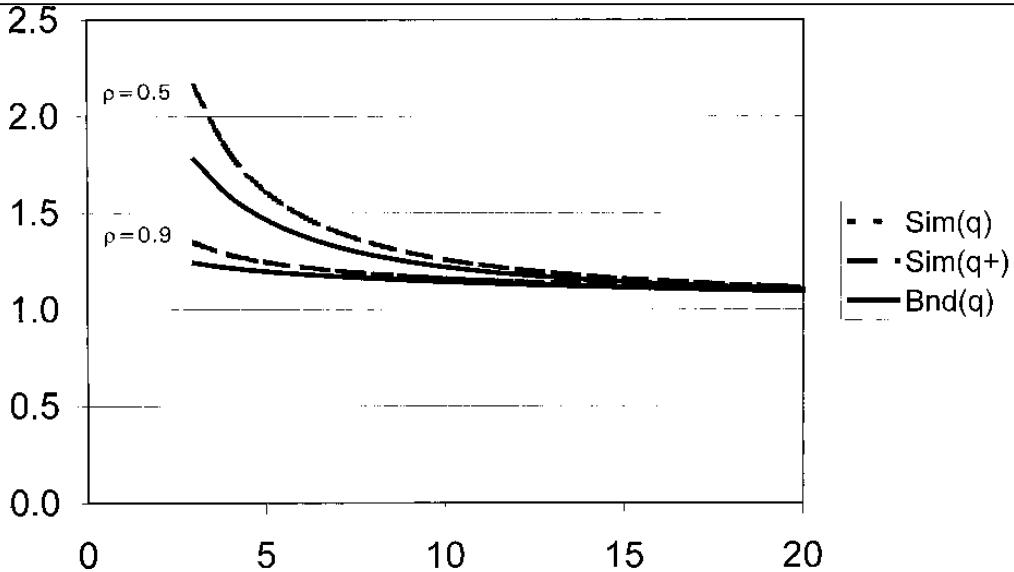


Figure 18 (Adapted from Chen et al. 1997) $\text{Var}(q)/\text{Var}(D)$ for $p = 0.5$ and 0.9

The above statement (without interactions at the retailer side) is mathematically proved in Chen et al.(1997) and Figure 18 depicts the difference between the variance at the supplier as well as the retailer sides. In this chart, the Y-axis is $\text{Var}(q)/\text{Var}(D)$, where $\text{Var}(q)$ is the variance of order placed by the supplier and $\text{Var}(D)$ is the variance of the demand received by the retailer. Chen et al. uses the same AR(1) demand process. The X-axis is the number of periods the supplier uses to find the moving average of the order received from retailers. This is to simulate the situations from minimum information to sufficient information at the supplier side. It is clear that the lower the value of ρ , the higher fluctuations at the supplier side (the value of $\text{Var}(q)/\text{Var}(D)$ get greater). With higher volatility, the cost of accommodating such volatility gets higher.

When interactions between the retailers are involved, which always happens when necessary. That is, when the value of relationship is high, the retailers share the overstock and backorders, which will reduce the variance of their orders to the supplier.

In turn, therefore, the supplier side will show less volatility and thus smaller average total costs. This is shown in Figure 19. The curve of *cor_H* is line at the bottom. The dotted line is from the base case without interactions. We notice that the curves for lower and medium relationship levels show higher than base case costs.

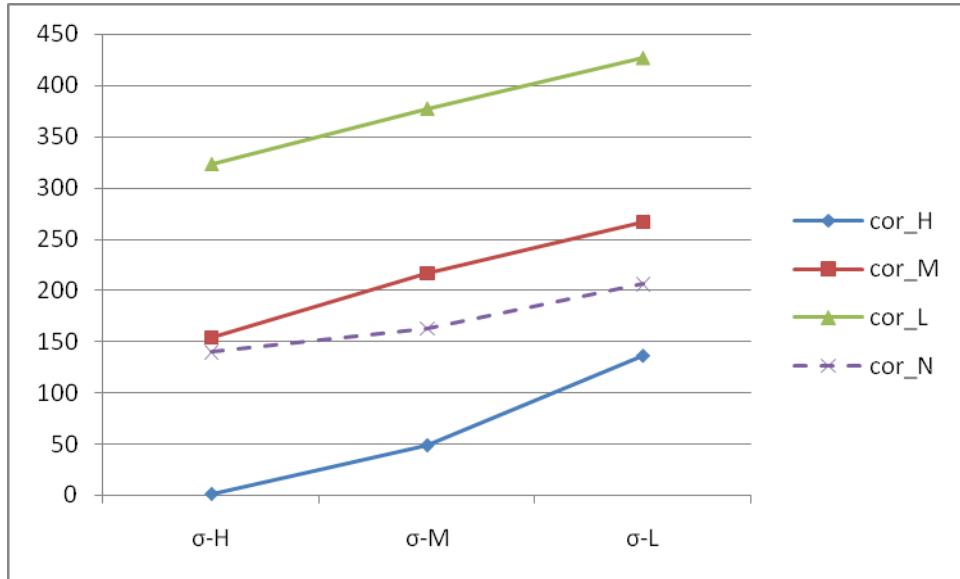


Figure 19 The Levels of Costs When ρ is Low

We may use an example to explain this abnormal result. To make the case simpler, without losing generality, we only examine two retailers with independent demand sources the same as in our implementation, and one supplier in the supply chain. Adding more retailers will not change the dynamics but add complexity, which thus is not necessary. Also we suppose one retailer (A) has relatively small variations in demand and the other's (B) relatively bigger. When the transactions start, A produces somewhat steady ordering flow up to the supplier, while the reordering quantities from B change with bigger deviations. If no interactions are involved, the commonly documented bullwhip effect will be observed as many researchers have proved. Suppose the

relationships between the retailers are at the low or medium levels, which means that the possibility of sharing information and stock is at a level that they tend not to happen. This may result in a distorted demand pattern in the eyes of the supplier because the variations of the demand may seem higher. Therefore, the supplier, without the knowledge of whether the two retailers have shared stock or not, may be “fooled” by the resulting reordering quantities received from the retailers and believe the quantity of order received is the true quantity and computes the underlying demand patterns accordingly. The supplier thus adjusts his own ordering quantity in favor of the adjusted demand. As proved in the bullwhip effect theory, the misperception results in an exaggerated variance of demand along the supply chain. The consequence is shown in our Figure 19 where the two curves of low and medium relationships at low ρ are above the base case line.

When the value of ρ reaches the medium level, the results start to show the advantages of interactions between the retailers. Figure 20 is the cost levels for medium ρ at the supplier side.

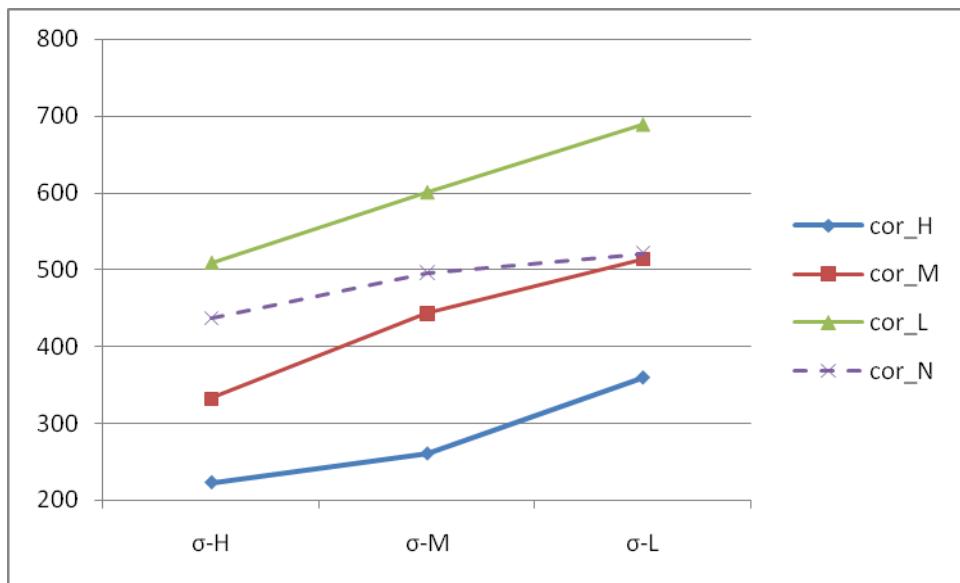


Figure 20 The Levels of Costs When ρ is Medium

When ρ is increased from low to high, by comparing this chart with the previous one, we notice that the *cor_N* (no relationship) curve range increases from about 150-200 to around 450-520, an increase of about two times the original; while the *cor_H* (high relationship) increases from about 10-180 to 230-360, about two times of the original. The significant change to the picture is *cor_M* (medium relationship) which also increases by about doubled from 150-270 to 330 -520. Although *cor_L* (low relationship) still shows higher costs at range of 500-700, an increase of only 1.5 times from about 350-450. It shows the trend of slower increase in costs for the cases with interactions when ρ goes from low to medium than that without interaction. Another observation from both figure 19 and 20 is that the curve goes upwards as the average standard deviation (or variance if the values are squared) of the demands for each retailer. It is quite easy to explain because the higher standard deviation values at the demand side, the more possible that one or more of the retailers will run under/over stock and raise the

need for sharing stock. Using the *IsShared()* test (the programming logic that tests whether the two agents may share the information or not), it is more likely the interaction is granted. Thus, the costs will be lower at the higher standard deviation level.

The last case to examine is when the value of ρ is at the highest level (Figure 21). This case needs more explanation since the pattern of the curve has changed, or the dynamics of the system show significant difference. Firstly, we notice the range costs of *cor_N* (no relationship) keeps its growth trend with values more than tripled from those at the medium ρ level at about 1580-1590. This brings the *cor_N* curve to the very top for all the cases investigated.

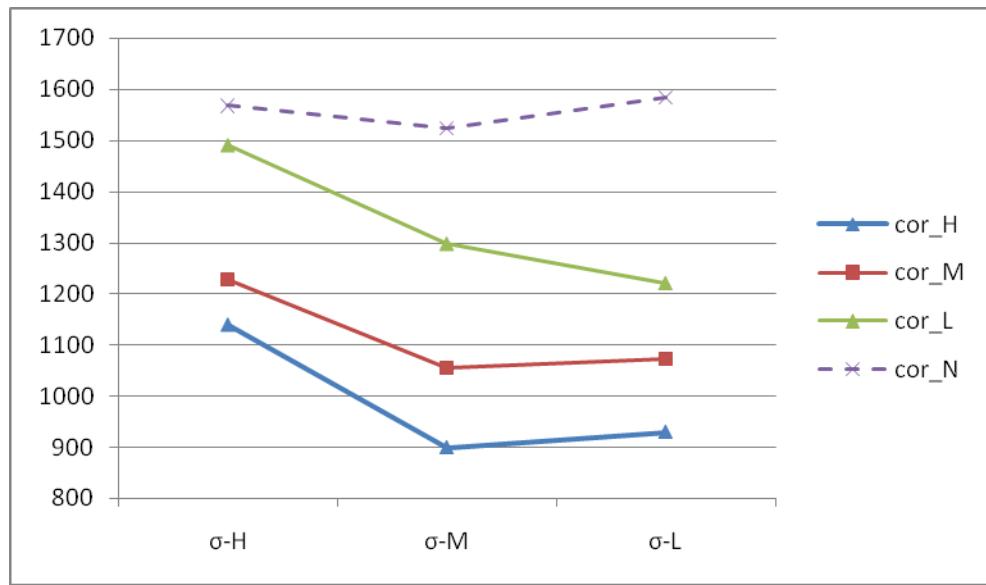


Figure 21 The Levels of Costs When ρ is High

Secondly, the curve patterns have changed. In the other cases, we notice that the curve increase continuously when ρ is of low and medium values. This is because when ρ gets greater and close to one, for the AR(1) demand process, the variance of the demand:

$$D_t = d + \rho D_{t-1} + \varepsilon_t,$$

can easily be derived as

$$Var(D_t) = \sigma^2 / (1 - \rho^2)$$

When ρ gets greater, it is obvious $Var(D_t)$ will increase to a very large value because $(1 - \rho^2)$ gets smaller. In fact

$$\lim_{\rho \rightarrow 1} Var(D_t) = \infty$$

While

$$\lim_{\rho \rightarrow 0} Var(D_t) = \sigma^2$$

At very high variances, the benefits from the interactions may not offset the volatility of the demand processes at the retailer level. Thus due to the high variance, though all are reduced to the levels lower or much lower than the one without interaction, the average costs for the high average deviation values are still very high as shown in the chart.

However, when the average demand deviation is lowered to medium and low levels, the curves resume the original patterns.

As we mentioned in the model description, we have three (relationships) 3x3 factorial design, which give us a three-dimension type of experimental design. We will get nine two-dimension tables as the results. So far we have examined three of them, which are

those that look at the impact of ρ . Now we turn our attention to the other two parameters, or independent variables that may affect the total average costs at the supplier side.

First we examine the relations between the ρ and σ at the fixed retailer relationship levels. The sets of experiments follow precisely the same procedures as described earlier, except we change the parameter relationship values rather than the values of ρ . Again we first look at the aggregate average costs at each level in one chart.

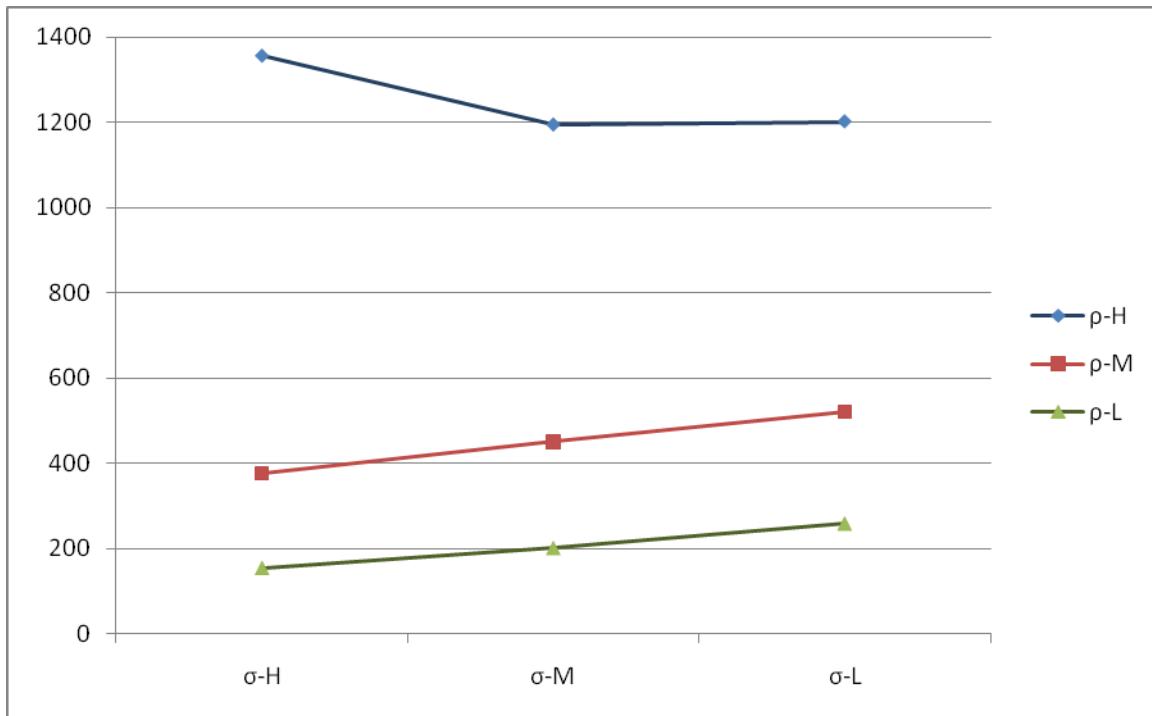


Figure 22 Effect of Information Sharing with Respect to σ and ρ

Figure 22 is a very consistent result with those shown in Figure 17. First, we see that the cost savings are significantly lower at the high and medium ρ levels, in addition to their obvious advantages to the non-relationship case as marked with purple (viewable if printed in color) x of dotted line above. The savings do not present when ρ level is low.

We have used an example earlier to explain reason for the latter case. Again we notice that the trend of the savings along the X-axis (standard deviation) is not continuously increasing. The best scenario according to our experiment is that the cost savings are maximized when ρ level is high and standard deviation of the demand process is at the medium level when relationship in general is in place.

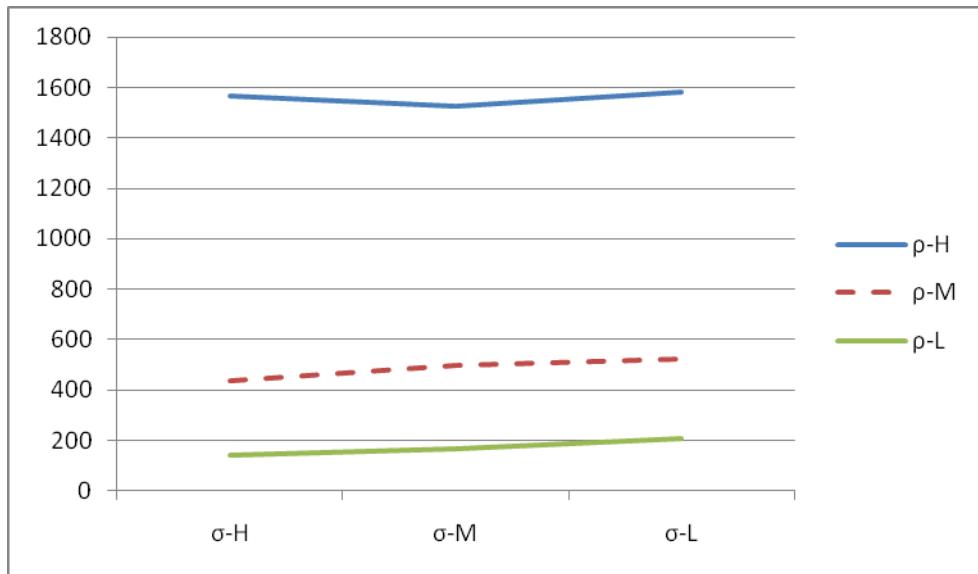


Figure 23 The Levels of Costs without no Interactions

Let us look now at the individual levels of relationship. The base case is the one without relationship as shown in Figure 23 above. This is the same pattern as found in Lee et al. (2000): at the fixed standard deviation level, the total average cost is larger when ρ level is higher. We show again this in Figure 5-8. The authors data are generated by fixing the standard deviation level to 0.7 and varying the value of ρ from zero to 0.9 (as noted early, if the value of ρ is too close to 1, the stock cost will be extremely high.). Clearly, increasing the value of ρ at a fixed σ level will increase the average stock cost. Our experiments exactly follow such a pattern. Again as our external validation has proved,

the agent based model that we have designed and implemented is capable of producing meaningful results as those previously observed and presented in the literature. However, the previous research has a large gap. They only examine the single point scenario and have not investigated multiple points scenarios in order to find the trend when standard deviation varies value from high to low. Our factorial design fills this gap.

Figure 1 Impact of ρ on Average Manufacturer's On-hand Inventory

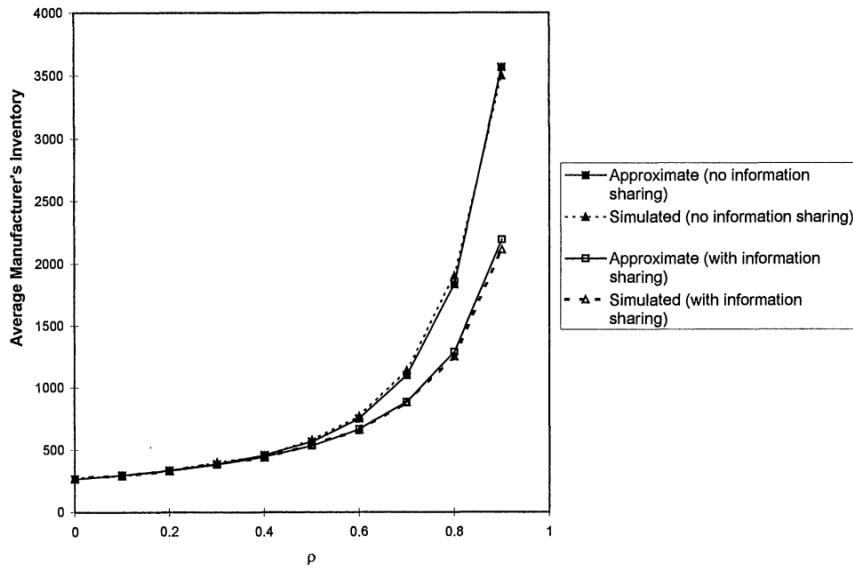


Figure 24 Impact of ρ on Average Stock Costs, Adapted from Lee et al. (2000)

Next we look at the cases with different levels of relationships. We list them in Figure 25 to 27. It is not a surprise that at all relationship values, the general pattern does not change: when level of standard deviation changes from high to low, at the low and medium levels of ρ , the average stock total costs go upward. This is because the benefits of the interaction are more significant at the experiments with higher volatility than otherwise. The difference is found for the curve of higher ρ . Though the general pattern does not change, i.e. the higher ρ produces higher average total costs, the trend however

goes straight downwards at low relationship level, goes downwards from σ_H to σ_M and is flat to σ_L at medium relationship level, and goes back to normal at high relationship level from σ_M to σ_L . As the reasons given in the previous discussion, when ρ is at its high level and standard deviation is high, the benefits of the relationship may not offset the higher costs due to higher volatility from the demand process, and thus consistently we see higher average cost output from the experiments. Nevertheless, when standard deviation goes lower, the interactions start to show the advantage and correct the curve back to normal.

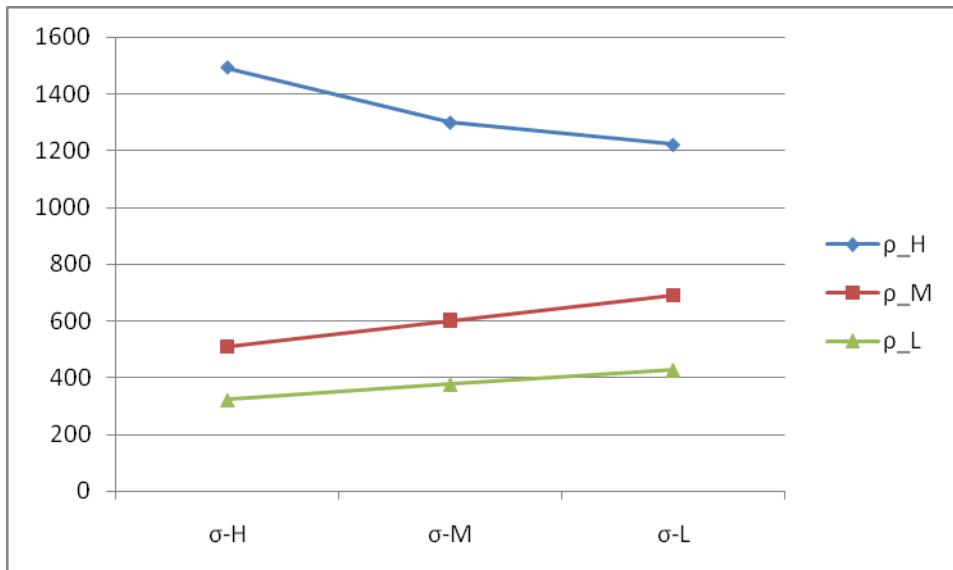


Figure 25 The Average Costs Curve when Relationship is Low

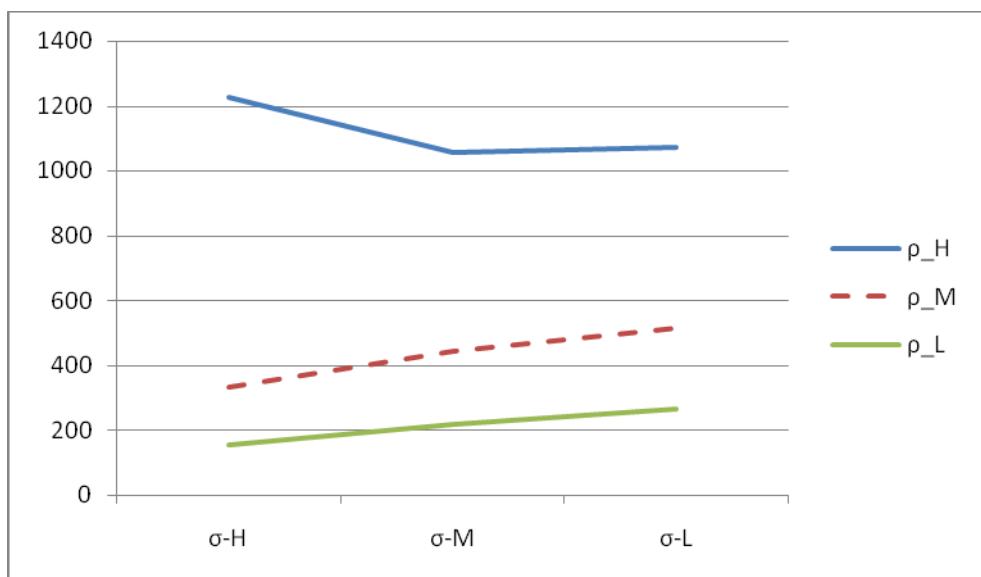


Figure 26 The Average Costs Curve when Relationship is Medium

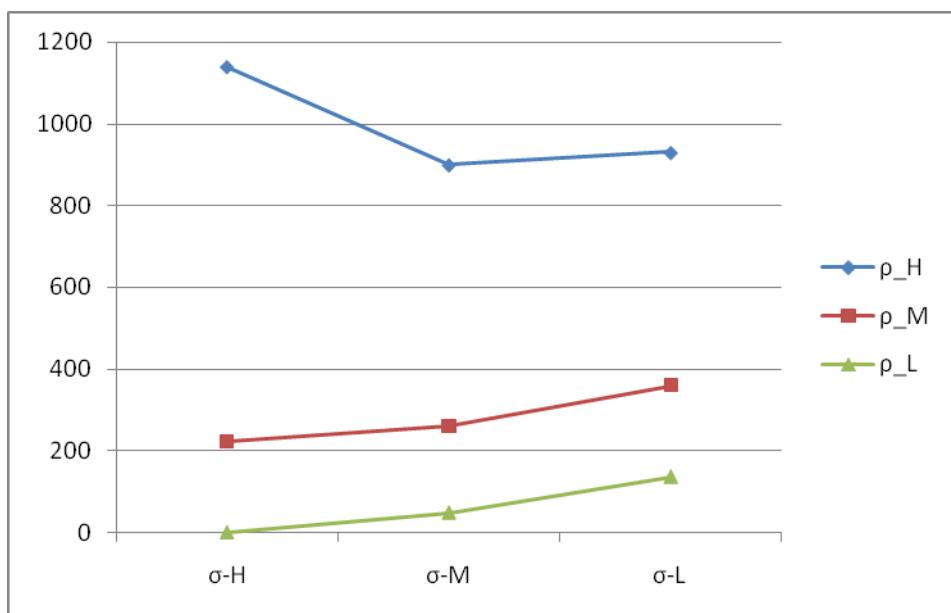


Figure 27 The Average Costs Curve when Relationship is High

The last set of runs is to fix the values of standard deviation and examine the relations between the interactions and the value of ρ . In fact this set of experiments just verified our observations and analysis above.

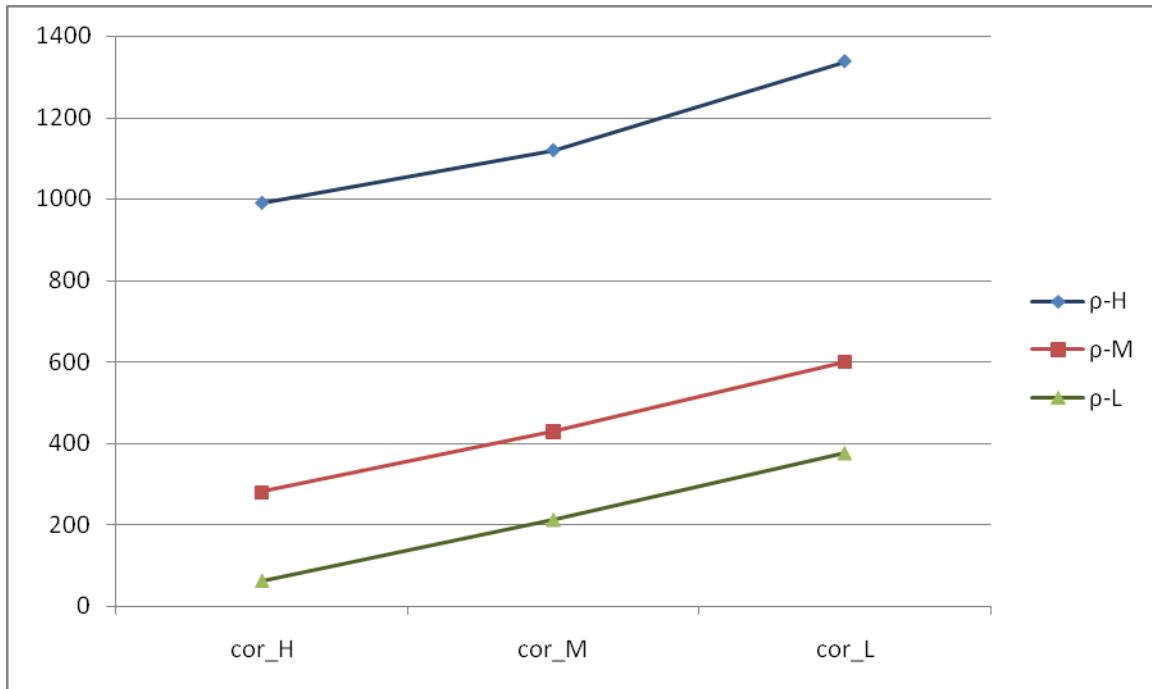


Figure 28 Effect of Information Sharing with Respect to Relationships and ρ

Figure 28 is the aggregate average costs chart that shows the relations between ρ and interactions. Three levels of standard deviation are used in the experiments and they are averaged to create this chart. Recall Figure 24 adapted from Lee et al. (2000) that at fixed demand standard variance or standard deviation, the total inventory costs goes up when the values of ρ increases, and the larger the value of ρ , after it passes about medium, the much faster the costs grow. We can see this pattern here. When we look at the distance between each curve, from smaller value of ρ to the higher ρ , at any points of relationship, including the one without relationship, the curves line one above the other

corresponding to the values of ρ . A noticeable much bigger gap is found between the medium ρ curve and the high ρ curve, indicating the fact of the faster growth of costs when the medium point is passed, exactly as shown in Lee et al. (2000). The curves also very nicely lay out the fact that at a fixed level of standard deviation, the lower the value of relationships, the higher the total average costs at the supplier side.

Figures 29 to 31 are the breakdowns of this aggregate chart. These charts are very similar in telling the stated analysis above validate the agent based model which follows the general theory in supply chain management as well as display the improvement in cost saving when interactions are put into action.

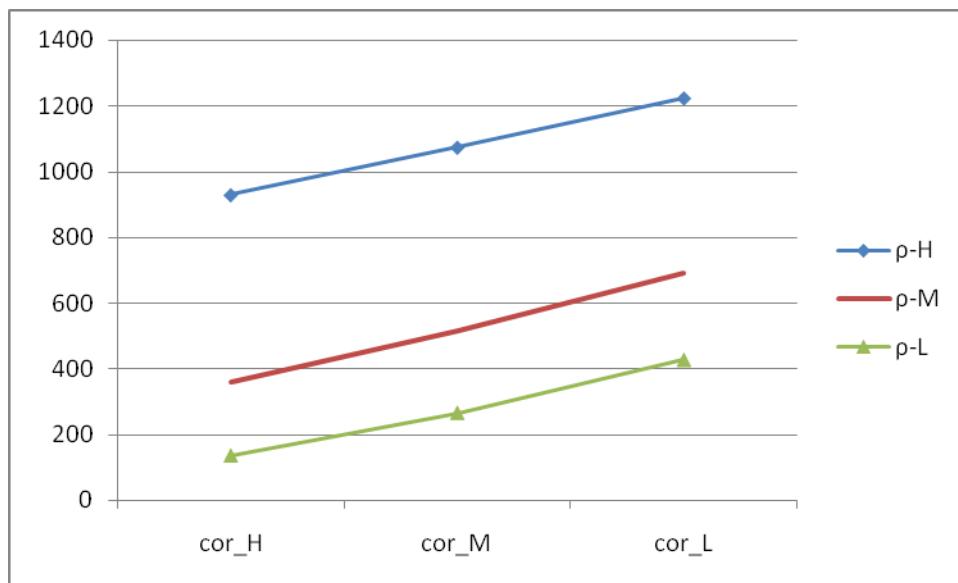


Figure 29 The Average Cost Curve with Low Standard Deviation

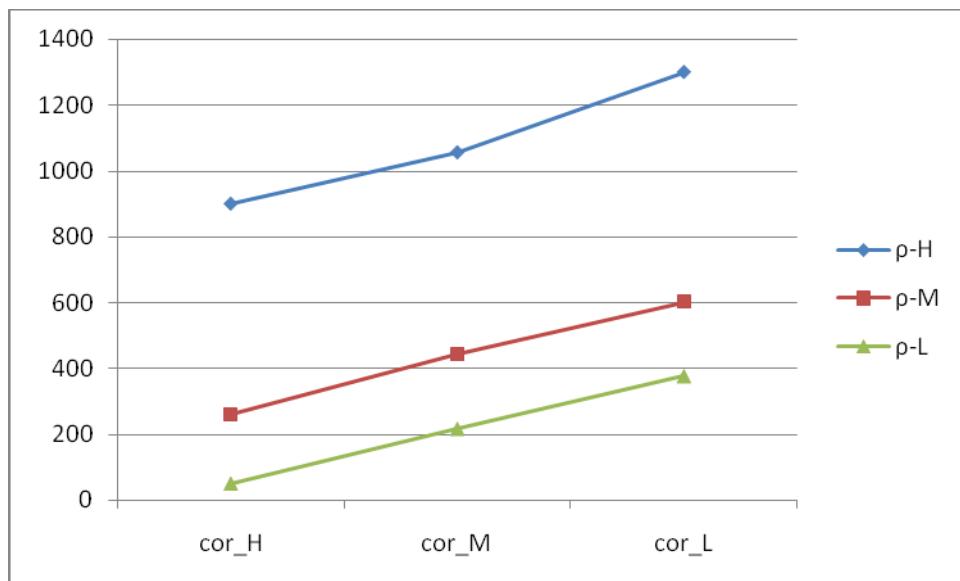


Figure 30 The Average Cost Curve with Medium Standard Deviation

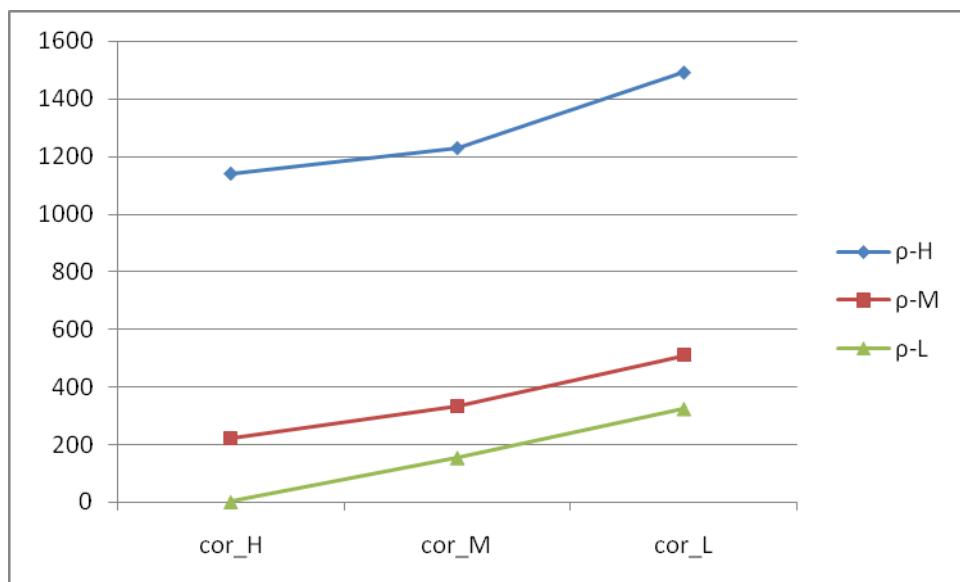


Figure 31 The Average Cost Curve with high Standard Deviation

This concludes the experimental design. Our factorial design has meaningfully produced the results that follow the theories in previous research and serves its purpose as the tool for internal validation.

Chapter 5 Conclusions and Future Research

Topics in supply chain management have been widely researched and the issues of information sharing are getting more attention in the age when information technology reaches such a height at the speed that has never been fully and readily taken advantage of. The information technology nowadays is able to and makes it possible to have the information or data transferred or communicated fast as well as accurate. However, the advancement of the technology does not guarantee the applications. Research is necessary to examine the impact of the information technology as well as the effects of information sharing in more detail, if not in more renovating ways to fully utilize the information technology that researchers twenty years ago were dying for. Artificial intelligence (AI in short and more commonly) is one of the areas that benefit from information technology. One of the major ideas of AI, from software engineering perspective only, is to create the applications that are embedded with advanced logic and algorithms to "think" like a human does. This has, still, been a far-reaching goal for hundreds and thousands of scientists and researchers since the term first coined by John McCarthy in 1956. The concept of artificial intelligence may be traced back to the time in the Greek myths of Hephaestus and Pygmalion which incorporated the idea of intelligent robots (such as Talos) and artificial beings (such as Galatea and Pandora)⁹. Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an intelligent agent is a system that perceives its environment and takes actions which maximize its chances of success. However, agents are not very often, yet,

⁹ See McCorduck, Pamela (2004), *Machines Who Think* (2nd ed.), Natick, MA: A. K. Peters, Ltd., pp. 4-5. Of course, the modern research only appeared in last century. The first working AI programs were written in 1951 to run on the Ferranti Mark I machine of the University of Manchester: a checkers-playing program written by Christopher Strachey and a chess-playing program written by Dietrich Prinz.

used in social science studies. Nevertheless, the literature in the recent years find the dramatic growth in papers published using agent based modeling and agent based simulation.

This thesis attempts to be part of the recent trend and push the research of applying intelligent agents in social science, in particular the supply chain management.

5.1 Summary and Conclusions

In this essay we have demonstrated that interactions between retailers do have plausible consequences on the supply chain performance. In particular, we look at the total average costs at the supplier side and find significant improvement in cost reduction as the interactions among the retailers increase. We applied the agent based model as the research methodology and through the external validation, We confirm that the new method produces the exact results as those from the previous research. We use agents to represent the firms in the supply chain and program the behaviors for each agent by applying the existing mathematically proved functions. We also use the agents to demonstrate the bullwhip effect. We have shown that without information sharing, the supplier needs to periodically update the mean and variance of demand based on observed demand data through orders received from retailers, then the variance of the orders placed by the supplier will be greater than the variance of the demand. More importantly, providing sufficient information of the underlying demand to the supplier can significantly reduce this increase in variability. However, we have also shown that the bullwhip effect will exist even when demand information is shared by all and all use the same forecasting technique and inventory policy.

To further explain this point, consider the retailers. Notice that if the retailers knew the mean and the variance of customer demand, then the orders placed by the retailers would be exactly equal to the customer demand and there would be no increase in variability.

However, in the model we implemented, even though the retailers have complete knowledge of the observed customer demands, they must still estimate the mean and variance of demand. As a result, the supplier sees an increase in variability.

When interactions are considered, the dynamics of the supply chain system change. In our model implementation, the information sharing between the retailers is considered as a type of interaction through communications. Such interactions will affect the relationships between the retailers. Therefore we use the values of the relationships as the proxy to examine how information sharing, thus interactions, may have impact on the supply chain performance. The model reveals the positive relationships between the two variables: interactions and performance.

Starting from the review of literature that develops different theories and models in supply chain management, we explore the different categories in model development. The supply chain management is mainly researched in four different categories, namely, information models, operational relationships models, and the contracting relationships models. By looking into each category, we summarize the theoretical development in the research in each area and find that, though agent based modeling has been well emphasized in many other fields of social sciences, very few are found in dealing with the problems in supply chain management.

Our model is coded using Java programming language, which models everything in the world as abstract objects and classes (groups of objects of the same nature, indeed; and they are objects themselves). An object in Java is like the real object in life. For example, we may define a car with wheels, body, engine, color, etc. in life, and in Java we define the exact same things. Another very important feature of Java, a Object Oriented Programming language as often called, is that each object may have behaviors coded as methods (similar to the functions and procedures in other programming languages). The model developer may design all types of logic and algorithms, as long as he may find and develop, and then the objects are able to “think” as close to our “thinking” as the program permits. This gives a great advantage to the model research world as the new tool to mimic the real world and by using the information technology and computing power to possibly create the models so real that other methods are not capable of doing. The similarity of the programming language to the real life concepts makes it a perfect candidate for implementing the models that the individual actors are important rather than just to look at the general outputs of hypothetical groups.

In addition to the external validation, we have also the experimental design. It is a 3 3x3 factorial design. The variables include the average total costs of the supplier, as well as the AR(1) demand process coefficient, the variance of the demand process, and the value of the relationships between retailers. The last variable is the proxy for the interaction, i.e., the information sharing. The results of the experimental design mainly follow the theoretical conclusions on the effects of the demand process coefficient as well as the variance of the demand process. However the improvement in the cost saving is

noticeable and we conclude that the information sharing between the retailers (horizontally) does have positive impact on the supply chain performance.

One of the major contributions of this essay is that unlike the previous researchers who examine the supply chain in a vertical way, we look at it in both ways, vertical and horizontal. This gives us a more complete picture of the system, though still not fully complete, as mentioned later in this chapter. Our approach to the system opens the possibilities of not just investigating the simple point to point interactions, but point to group of interacting points.

This essay would be incomplete if we did not mention several important limitations of our model and results. The main drawback of our model is the assumption that information sharing / interaction has linear relations with the relationships between the retailers. This is because of the lack of theory and empirical evidence. However, due to the scalability of the agent based model that we have developed, it is relatively easy if some new connections between the interaction and relationships is defined and coding into different software module is relatively easy and adding the new module to the model will be of an easy task comparing to the task of defining such links. Nevertheless, the change of the functions that define the links between the interaction and relationships may not change the model dynamic much as we can expect that the smoothed reordering curve due to the interactions may subsequently reduce the variance at the supplier side and thus provide cost reduction benefits. This is yet to be tested. Another limitation of our model is the fact that we are using the optimal order-up-to policy, as defined in Lee et al. (2000), but rather the policy defined in Chen et al. (2000). The order-up-to policy is

widely used in literature and made the assumption that the retailer has all the information about the underlying demand, including the values of the ρ and σ . These assumptions are noticeably not very practical as to the difficulties of obtaining such information. Chen et al. (2000) use the moving average to compute the mean lead time demand and standard deviation of the lead time forecast error. It is important to point out that policies of the latter form are frequently used in practice and the moving average is one of the most commonly used forecasting techniques in practice. In the future research, we may use this function to test our model performance.

Finally the assumption of one supplier and three retailers could not clearly capture many of the complexities involved in real-world supply chain. Although our model has been much more complex than most of the models that are developed in literature, which most commonly assume one supplier and one (or two) retailers because it is not necessary to add more retailers for the interactions between them are ignored. Although still having not examined the impact of more than one supplier (more than one supplier will introduce a much complex pattern to the dynamics of the supply chain as all the buy-seller relationships will start to take effect), our model has successfully experimented and demonstrated the significant improvement in supply chain performance due to the interactions between retailers.

5.2 Future Research

Chen et al. (2000) claim that the overstock will not affect the variability of the retailer or supplier. They verified their claim by a simulation. However, the authors have not shown how the stocks are handled if not sent directly back to the upstream without

charge. I have some doubts of their findings and propose that the variability may vary if the stocks are handled differently.

The literature has not examined the impact of different demand (or sales) forecasting functions to the order variability, inventory level, as well as the total costs. Some empirical tests may be useful to find the best forecast functions so that the agent based model created may produce more realistic results. This is the advantage of the use of intelligent agents in the model that give the model the adaptability to changes and learn from the different sources as external inputs, in addition to the past errors, just as a real firm does (at least as close to this goal as possible).

The literature, in order to make the analysis simple and mathematically traceable, totally ignored the possible impact of the backorders to the total costs at each level of the supply chain; and of course they have not examined the approaches of how the backorders are handled and what they may affect the inventory level as well as the total costs. Our thesis has noticed this gap and, other than unrealistically ignoring them, has investigated the simplest approach of handling the overstock by fulfilling them in the next period and additional quantity is added to the forecasted reorder amount.

Last but not the least, the literature on information sharing in the supply chain may require great efforts to fill the gap of examining what types of information should be shared. There is a great momentum behind the growth in supply chain model research. Stepping back by taking a broad view of the literature on the said research, however, causes a degree of discomfort. The vast majority of articles focus on finding the optimal ordering quantities, upper and lower bounds of certain algorithms, or discuss the

equilibrium points, while little attention is devoted to the types of information to be shared and how different types of information may impact the supply chain performance and in what degree.

5.3 Software Packages for Building and Running the Agent-Based Models

REPAST (Recursive Porous Agent Simulation Toolkit) (<http://repast.sourceforge.net/>), originally developed at University of Chicago, now maintained in the Argonne National Laboratory, is a widely used, free, and open-source, agent-based modeling and simulation toolkit. Three Repast platforms are currently available, each of which has the same core features but a different environment for these features. Repast Simphony (Repast S) extends the Repast portfolio by offering a new approach to simulation development and execution. Repast is one of the most used agent based simulation frameworks and is under very active updates. It has the root of the SWARM toolkit as introduced below. The current version is Repast Simphony 1.0. The Repast software is available to the general public under GNU licensing terms.

SWARM (http://www.swarm.org/wiki/Main_Page) is a software package for multi-agent simulation of complex systems, originally developed at the Santa Fe Institute (<http://www.santafe.edu>). Swarm is intended to be a useful tool for researchers in the study of agent based models. Swarm software comprises a set of code libraries which enable simulations of agent based models to be written in the Objective-C or Java computer languages. These libraries will work on a very wide range of computer platforms. The basic architecture of Swarm is the simulation of collections of concurrently interacting agents: with this architecture, we can implement a large variety

of agent based models. The Swarm software is available to the general public under GNU licensing terms. Every year SWARM hosts (together with Repast) the Swarm Fest conference in Chicago.

Netlogo (<http://ccl.northwestern.edu/netlogo/>) is less popular than the above two. It still has its place in the applications. NetLogo is a multi-agent programming language and integrated modeling environment. NetLogo was designed in the spirit of the Logo programming language to be "low threshold and no ceiling," that is to enable easy entry by novices and yet meet the needs of high powered users. The NetLogo environment enables exploration of emergent phenomena. It is particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of independent "agents" all operating concurrently. This makes it possible to explore the connection between the micro-level behavior of individuals and the macro-level patterns that emerge from the interaction of many individuals. NetLogo was developed at Northwestern University and has been funded by the National Science Foundation and other foundations.

Bibliography

Anand, Krishnan and Mendelson, Haim (1997), "Information and Organization for Horizontal Multimarket Coordination", *Management Science*, Vol. 43, No. 12, pp. 1609-1627.

Aoki, Masahiko; Gustafsson, Bo; and Williamson, Oliver (1990) (Editors), "The Firm as a Nexus of Treaties", Swedish Collegium for Advanced Study in the Social Sciences series, Sage, London, UK.

Axelrod, Robert; and Hamilton, William D. (1981), "The Evolution of Cooperation", *Science*, No. 211, pp. 1390-1296.

Axelrod, Robert (1984 and 2006), "The Evolution of Cooperation", Basic Book, New York.

Axelrod, Robert (2005), "Advancing the Art of Simulation in the Social Sciences", in Handbook of Research on Nature Inspired Computing for Economy and Management, Jean-Philippe Rennard (Ed.), Hersey, PA: Idea Group.

Baiman, Stanley; Fischer, Paul E.; Rajan, madhav V. (2001), "Performance Measurement and Design in Supply Chains", *Management Science*, Vol. 47, No. 1, pp. 173-188.

Bakos, Yannis (1998), "The Emerging role of electronic marketplaces on the Internet", *Communications of the ACM*, Vol. 41, August, pp. 35-42.

Bakos, Yannis, and Brynjolfsson, Erik (1999), "Bundling Information Goods: Pricing, Profits, and Efficiency", *Management Science*, Vol. 45, No. 12, pp. 1613-1630.

Balmer, Michael; Nagel, Kai; and Raney, Bryan (2004), "Large-Scale Multi-Agent Simulations for Transportation Applications", *Intelligent Transportation Systems*, Vol. 8, pp. 205-221.

Barney, Jay (1991), "Firm Resources and Sustained Competitive Advantage", *Journal of Management*, Vol. 17, No. 1, pp. 99-120.

Banker, Rajiv D.; Kalvenes, Joakim; Patterson, Raymond A. (2000), ""information technology, Contract Completeness, and Buyer-Supplier Relationships", the 21st International Conference on Information Systems (ICIS-00), Brisbane, Australia, 2000

Barr, Dale J. (2004), "Establishing Conventional communication Systems: Is Common Knowledge Necessary?", *Cognitive Science*, Vol. 28, pp. 937-962.

Barringer, Bruce; and Harrison, Jeffrey (2000), "Walking a Tightrope: Creating Value through Inter-organizational Relationships", *Journal of Management*, Vol. 26, No. 3, pp. 367-403.

Batty M.; Dodge M.; Jiang B.; and Smith A. (1999), "Geographical information systems and urban design", In: Stillwell J., Geertman S. and Openshaw S. (eds), *Geographical Information and Planning*, Springer, Heidelberg, pp. 43-65.

Bechtel, Christian; and Jayaram, Jayanth (1997), "Supply Chain Management: A Strategic Perspective", International Journal of Logistics Management, Vol 8 No 1, 1997, p. 15-34.

Bernstein, Fernando, and Federgruen, Awi (2005), "Decentralized Supply Chains with Competing Retailers Under Demand Uncertainty", Management Science, Vol. 51, No. 1, pp. 18-29.

Bertalanffy, Ludwig Von (1968), General System Theory: Foundations, Development, Applications.

Borys, Bryan; and Jemison, David (1989), "Hybrid Arrangements as Strategic Alliances: Theoretical issues in organizational combinations", Academy of Management Review, Vol. 14, No. 2, pp. 234.

Bowersox, D. J. (1969), "Readings in Physical Distribution Management: the Logistics of Marketing". Eds. Bowersox, D.J.; la Londe, B.J.; and Smykay, E.W.; New York: MacMillan

Bourland, Karla E.; Powell, Stephen G.; and Pyke, David F. (1996), "Theory and Methodology: Exploiting Timely Demand Information to Reduce Inventories", European Journal of Operational Research, Vol. 92, pp. 239-253.

Bowersox, D. J. (1969), "Readings in Physical Distribution Management: the Logistics of Marketing". Eds. Bowersox, D.J.; la Londe, B.J.; and Smykay, E.W.; New York: MacMillan.

Brereton, Pearl (2004), "The Software Customer/Supplier Relationship",
Communications of the ACM, Vol. 47, No. 2, pp. 77-81.

Brooks, R. A. (1991a), "Intelligence without reason", in: Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, Australia, pp. 569-595.

Cachon, Gerard P. (1999), "Managing Supply Chain Demand Variability with Scheduled Ordering Policies", Management Science, Vol. 45, No. 6, pp. 843-856.

Cachon, Gerard P. and Fisher, Marshall (2000), "Supply Chain Inventory Management and the Value of Shared Information", Management Science, Vol. 46, No. 8, pp. 1032-1048.

Cachon, Gerard P., and Zipkin, Paul H. (1999), "Competitive and Cooperative Inventory Policies in a Two-State Supply Chain", Management Science, Vol. 45, No. 7, pp. 936-953.

Cachon, Gerard P. (2001), "Stock Wars: Inventory Competition in a Two-Echelon Supply Chain with Multiple Retailers", Management Science, Vol. 49, No. 5, pp. 658-674.

Cachon, Gerard P. (2004), "The Allocation of Inventory Risk in a Supply Chain: Push, Pull, and Advance-Purchase Discount Contracts", Management Science, Vol. 50, No. 2, pp. 222-238

Cachon, Gerard P., and Lariviere, Martin A. (2001), "Capacity Choice and Allocation: Strategic Behavior and Supply Chain Performance", Management Science, Vol. 45, No. 8, pp. 1091-1108.

Cachon, Gerard P., and Lariviere, Martin A. (2001), "Contracting to Assure Supply: How to share Demand Forecasts in a Supply Chain", Management Science, Vol. 47, No. 5, pp. 627-646.

Cachon, Gerard P., and Lariviere, Martin A. (2005), "Supply Chain Coordination with Revenue-Sharing Contracts: Strengths and Limitations", Management Science, Vol. 51, No. 1, pp. 30-44.

Chen, Fangruo (1998), "Echelon Reorder Points, Installation Reorder Points, and the Value of Centralized Demand Information", Management Science, Vol. 44, No. 12, Part 2 of 2, pp. S221-S234.

Chen, Frank; Drezner, Zvi; Ryan, Jennifer K.; and Simchi-Levi, David (2000), "Quantifying the Bullwhip Effect in a Simple Supply Chain; The Impact of Forecasting, Lead Times, and Information," Management Science, Vol. 46, No.3, pp. 436-443.

Chetty, Sylvie; and Holm, Desiree Blankenburg (2000), "Internationalisation of small to medium-sized manufacturing firms: a network approach", International Business Review, Vol. 9, pp. 77-93

Clarke, Richard (1983), "Collusion and the Incentives for Information Sharing", The Bell Journal of Economics, Vol. 14, pp. 383-394.

Cooper, Martha C.; Lambert, Douglas M.; and Pagh, Janus D. (1997), "Supply Chain Management: More Than a New Name for Logistics", International Journal of Logistics Management, Vol 8 No 1, 1997, p. 1-14.

Corbett, Charles J.; DeCroix, Gregory A. (2001), "Shared-Savings Contracts for Indirect materials in Supply Chains: Channel Profits and Environmental Impacts", Management Science, Vol. 47, No. 7, pp. 881-893.

Corbett, Charles J.; DeCroix, Gregory A.; Ha, Altert Y. (2005), "Optimal Shared-Savings contracts in Supply Chains: Linear Contracts and Double Moral hazard", European Journal of Operational Research, Vol. 163, pp. 653-667.

Cravens, David; and Shipp, Shannon (1993), "Analysis of Co-operative Interorganizational Relationships, Strategic Alliance Formation, and Strategic Alliance Effectiveness", Journal of Strategic Marketing, Vol. 1, No. 1, pp. 55-70.

Craves, Stephen (1999), "A Single-Item Inventory Model for a Nonstationary Demand Process", Manufacturing & Service Operations Management, Vol. 1, No. 1, pp. 50-61.

CSCMP (2005), <http://www.cscmp.org/Website/Resources/Terms.asp>

Darl, Charles; Remolina, Emilio; Ong, Jim (2005), "Distributed Troubleshooting Agents", in: Proceedings of the 2nd International Conference on Autonomic computing (ICAC05).

Dewan, Sanjeev, and Mendelson, Haim (1990), "User Delay Costs and Internal Pricing for a Service Facility", Management Science, Vol. 36, No. 12, pp. 1502-1517.

Donohue, Karen L. (2000), "Efficient Supply Contracts for Fashion Goods with Forecast Updating and Two Production Modes", *Management Science*, Vol. 46, No. 11, pp. 1397-1411.

Drogoul, Alexis, and Ferber, Jacques (1992), "Multi-Agent Simulation as a Tool for Modeling Societies: Application to Social Differentiation in Ant colonies", In: Proceedings of MAAMAW92: 4th European Workshop on Modeling Autonomous Agents in a Multi-Agent World, Italy.

El-Ansary, Adel; Stern, Louis (1972), "Power Measurement in the Distribution Channel", *Journal of Marketing Research*, Vol. 9, February, pp. 47-52.

Eppen, Gary d., and Iyer, Ananth V. (1997), "Backup Agreements in Fashion Buying – The Value of Upstream Flexibility", *Management Science*, Vol. 43, No. 11, pp. 1469-1484

Escudero, L. F.; Galindo, E.; Garcia, G.; Gomez, E.; and Sabau, V. (1999), "Schumann, a Modeling Framework for Supply Chain Management", *European Journal of Operational Research*, Vol. 119, pp. 14-34.

Ford, David (1980), "The Development of Buyer-Seller Relationships in Industrial Markets", *European Journal of Marketing*, Vol. 14, No. 5/6, pp. 339-353.

Forrester, Jay W. (1969), *Industrial Dynamics*, The MIT Press, Cambridge, Massachusetts.

Franklin, Stan, and Graesser, Art (1997), “Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents”, in: Muller, J.P., Wooldridge, M.J., and Jennings, N.R. (Eds.) Intelligent Agents III: Agent Theories, Architectures, and languages, Springer-Verlag, Berlin, pp. 21-35.

Gallego, Guillermo and Ozer, Ozalp (2003), “Optimal Replenishment Policies for Multiechelon Inventory Problems under Advance Demand Information”, Manufacturing & Service Operations Management, Vol. 5, No. 2, pp. 157-175.

Ganeshan, Ram; Jack, Eric; Magazine, M. J.; and Stephens, Paul (1999), “A Taxonomic Review of Supply Chain Management Research”, Quantitative Models for Supply Chain Management, Eds. Tayur, Sridhar; Ganeshan, Ram; and Magazine Michael, Boston: Kluwer Academic Publishers, pp. 839.

Garcia-Flores, Rodolfo, and Wang, Xue Zhong (2002), “A Multi-Agent System for Chemical Supply Chain Simulation and Management Support”, OR Spectrum, No. 24, pp. 243-370.

Gavirneni, Srinagesh; Kapuscinski, Roman; and Tayur, Sridhar (1999), “Value of Information in Capacitated Supply Chains”, Management Science, Vol. 45, No. 1, pp. 16-24.

Gilbert, Nigel (2007), “Agent-Based Modeling, Series: Quantitative Applications in the Social Sciences”, SAG Publications.

Gurnani, Haresh and Tang, Christopher S. (1999), "Optimal Ordering Decisions with Uncertain Cost and Demand Forecast Updating", Management Science, Vol. 45, No. 10, pp. 1456-1462.

Harland, C. M. (1996), "Supply Chain Management: Relationships, Chains and Networks", British Journal of Management, vol. 7, Special Issue, pp. S63-S80

Heide, Jan (1994), "Interorganizational Governance in Marketing Channels", Journal of Marketing, Vol. 58, No. 1, pp. 71-85.

Heide, Jan; and John, George (1990), "Alliances in Industrial Purchasing: The Determinants of Joint Action in Buyer-Supplier Relationships", Journal of Marketing Research, Vol. 27, February, pp. 24-36.

Hilton, Ronald (1981)"The Determinants of Information Value: Synthesizing Some General Results", Management Science, Vol. 27, No. 1, pp. 57-64.

Hitt, M. A.; Ireland, D. R.; and Hoskisson, R. E. (1999), Strategic Management, Cincinnati, OH: South-Western College Publishing, Chapter 1.

Houlihan, John B. (1985), "International Supply Chain Management", International Journal of Physical Distribution & Materials Management, Vol. 15, pp. 22-38.

Howard R.A. (1966), "Information Value Theory", Systems Science and Cybernetics, IEEE Transactions, Vol. 2, No. 1, pp. 22-26.

Hui, Pamsy P., and Beath, Cynthia M. (2001), "The IT Sourcing Process: A Research Framework", The Annual Meeting of the Academy of Management, Washington, DC.

Janssen, Marco A., Jager, Wander (2003), "Simulating market dynamics: Interactions between Consumer Psychology and Social Networks", Artificial life, Vol. 9, 343-356.

Jennings, Nicholoas R., Sycara, Katia, and Wooldridge Michael (1998), "A Roadmap of Agent Research and Development", Autonomous Agents and Multi-Agent Systems, Vol. 1, No. 1, pp. 7-38.

Johns, T. C., and Riley, D. W. (1984), "Using Inventory for Competitive Advantage through Supply Chain Management", International Journal of Physical Distribution & Materials Management, Vol. 15, pp. 16-26.

Johnson, M. Eric, and Whang, Seungjin (2002), "E-Business and Supply Chain Management: An Overview and Framework", Production and Operations management, Vol. 11, No. 4, pp. 413-423.

Kahn, James A. (1987), "Inventories and the Volatility of Production", The American Economic Review, Vol. 77, No. 4, pp. 667-679.

Kargl, Frank; Illmann, Torsten; and Weber Michael (1999), CIA – a Collaboration and Coordination Infrastructure for Personal Agents", DAIS 99, Helsinki, Finland.

Kenney, J. F. and Keeping, E. S. (1951) Mathematics of Statistics, Pt. 2, 2nd ed. Princeton, NJ: Van Nostrand.

- Kumar, Kuldeep (2001), "Technology for Supporting Supply Chain Management: Introduction", Communications of the ACM, Vol. 44, No. 6, pp. 58-61.
- La Londe, Bernard J., and Masters, James (1994), "Emerging Logistics Strategies: Blueprints for the next Century", International Journal of Logistics Management, Vol. 24, No. 7, pp. 35-47
- LeBaron, Blake (1998), "Technical Trading Rules and Regime Shifts in Foreign Exchange", In: Acar, E., Satchell, S. (Eds.), Advanced Trading Rules, Butterworth-Heinemann, London, pp. 5-40.
- LeBaron, Blake (2000), "Agent-Based Computational Finance: Suggested Readings and Early Research", Journal of Economic Dynamics & Control, Vol. 24, No. 5-7, pp. 679-702.
- LeBaron, Blake (2001), "Evolution and Time Horizons in an Agent-Based Stock Market", Macroeconomic Dynamics, Vol. 5, No. 2, pp. 225-254.
- Lancioni, R. A.; Smith, M. F.; and Oliva, T. A. (2000), "The Role of the Internet in Supply Chain Management", Industrial Marketing Management, Vol. 29, No. 1, pp. 45-56.
- Lau Jason S. K.; Huang, George Q.; and Mak, K. L. (2004), "Impact of Information Sharing on Inventory replenishment in Divergent Supply Chain", International Journal of Production Research, Vol. 42, No. 5, 919-941.

Laugley, C. J. (1992), "The Evolution of the Logistics Concept, in Logistics: The Strategic Issues, in Christopher, Martin (1992), Logistics: The Strategic Issues, Chapman and Hall, London, UK.

Lederer, Phillip J., and Li, Lode (1997), "Pricing, Production, Scheduling, and Delivery-Time Competition", Operations Research, Vol. 45, No. 3, pp. 407-420.

Lee, Hau L., and Billington, Corey. (1993), "Material Management in Decentralized Supply Chains", Operations Research, Vol. 41, No. 5, pp. 835-847.

Lee, Hau, Padmanabhan, V, and Whang, Seungjing (1997), "Information Distortion in a Supply Chain: the Bullwhip Effect", Management Science, Vol. 43, No. 4, pp. 546-558.

Lee, Hau, and Whang, Seungjing (2000), "Information Sharing in a Supply Chain", International Journal of Manufacturing Technology and Management, Vol. 1, No. 1, pp. 79-93.

Lee, Hau L.; So, Kut C.; and Tang Christopher S. (2000), "The Value of Information Sharing in a Two-Level Supply Chain", Management Science, Vol. 46, No. 5, pp. 626-643.

Lin, Fu-ren, Sung, Yu-wei, and Lo, Yi-pong (2005), "Effects of Trust Mechanisms on Supply-Chain Performance: A Multi-Agent Simulation Study", International Journal of Electronic Commerce, Vol. 9, No. 4, pp. 91-112.

Lopez-Sanchez, Maite; Noria, Xavier; Rodriguez, Juan A.; Gilbert, N.; and Schuster, S. (2004), “Multi-Agent Simulation Applied to On-Line Music Distribution market”, in Proceedings of the 4th International Conference on Web Delivering of Music.

Mahoney, Joseph; and Pandian, Rajendran (1992), “The Resource-Based View Within the Conversation of Strategic Management”, Strategic Management Journal, Vol. 13 No. 5, pp. 363.

Mayer, R. C., Davis, J. H., and Schoorman, F. D. (1995), “An Integrative Model of Organizational Trust”, Academy of Management Review, Vol. 20, No. 3, pp. 709-734.

Meixell, Mary J., and Gargeya, Vidyaranya (2005), Global Supply Chain Design: A Literature Review and Critique, Transportation Research Part E, Vol 41, pp. 531-550.

Mendelson, Haim, and Whang Seungjin (1990), “Optimal Incentive-Compatible Priority Pricing for the M/M/1 Queue”, Operations Research, Vol. 38, No. 5, pp. 870-883.

Mentzer, John T.; DeWitt, William; Keebler, James S.; Min, Soonhon; Nix Nancy W.; Smith, Carlo D.; and Zacharia, Zach G. (2001), “Defining Supply Chain Management”, Journal of Business Logistics, Vol. 22, No. 2, pp. 1-25

Metters, Richard (1987), “Quantifying the Bullwhip Effect in Supply Chain”, Journal of Operations Management, Vol. 15, pp. 89-100

Milgrom, Paul, and Roberts, John (1988), “Communication and Inventory as Substitutes in Organizing Production”, Scandinavian Journal of Economics, Vol. 90, No. 3, pp. 275-289.

Monczka, Robert; Trent, Robert; and Handfield, Robert (1998), "Purchasing and Supply Chain Management", Cincinnati, OH: South-Western College Publishing, Chapter 8.

Nahmias, Steven, Demmy, Steven W. (1981), "Operating Characteristics of an Inventory System with Rationing", *Management Science*, Vol. 27, No. 11, 1236-1245.

Nilsson, Nils (1998), *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann Publishers.

Novshek, William and Sonnenschein, Hugo (1982), "Fulfilled Expectations Cournot Duopoly with Information Acquisition and Release", *The Bell Journal of Economics*, Vol. 13, pp. 214-218.

Nwana, Hyacinth S (1996), "Software Agents: An Overview", *Knowledge Engineering Review*, Vol. 11, No. 3, pp. 205-244.

Oliver, R. Keith; and Webber, Michael D. (1982), "Supply Chain management: Logistics Catches up with Strategy", in Christopher, Martin (1992), *Logistics: the Strategic Issues*, Chapman and Hall, London, UK, pp. 63-75.

Overby, Jeffrey W., and Min Soonhong (2001), "International Supply Chain Management in an Internet Environment: A Network-Oriented approach to Internationalization", *International Marketing Review*, Vol. 14, No. 4pp. 392-420.

Padmanabhan, V. and Png, I.P.L. (1997), "Manufacturer's Returns Policies and Retail Competition", *Marketing Science*, Vol. 16, No. 1, pp. 81-94.

Parkhe, Arvind (1993), "Strategic Alliance Structuring: A Game Theoretic and Transaction Cost Examination of Interfirm Cooperation", *Academy of Management Journal*, Vol. 36, No. 4, pp. 794-829.

Poirier, C. C., and Bauer, M. (2000), "E-Supply Chain: Using the Internet to Revolutionize Your Business", Berrett-Koehler Publishers.

Poole, David; Mackworth, Alan & Goebel, Randy (1998), *Computational Intelligence: A Logical Approach*, Oxford University Press.

Potok, Thomas E.; Elmore, mark; Reed, Joel; and Sheldon, Frederick T. (2003), "VIPAR: Advanced Information Agents Discovering Knowledge in an Open and Changing Environment", in: *Proceedings of 7th World Multiconference on Systemics, Cybernetics and Informatics Special Session on Agent-Based Computing*, Orlando, FL, pp. 28-33.

Raghunathan, Srinivasan (2001), "Information Sharing in a Supply Chain: A Note on its Value when Demand Is Nonstationary", *Management Science*, Vol. 47, No. 4, pp. 605-610.

Raith, Michael (1996), "A General Model of Information Sharing in Oligopoly", *Journal of Economic Theory*, Vol. 71, pp. 260-288.

Raman, Ananth and Fisher, Marshall (1996), "Reducing the Cost of Demand Uncertainty through Accurate Response to Early Sales", *Operations Research*, Vol. 44, No. 1, Special Issue on New Directions in Operations Management (Jan. – Feb., 1996), pp. 87-99.

Scott, C., and Westbrook, R. (1991), "New Strategic Tools for Supply Chain Management", International Journal of Physical Distribution & Logistics Management, Vol. 21, pp. 22-33.

Russell, Stuart J. & Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, NJ: Prentice Hall.

Siguaw, Judy; Simpson, Penny; and Baker, Thomas (1998), "Effects of Supplier Market Orientation on Distributor Market Orientation and the Channel Relationship: The Channel Relationship: The Distributor Perspective", Journal of Marketing, Vol. 62, July, pp. 99-111.

Singh, Bahul, Salam, A. F., and Iyer, Lakshmi (2005), "Agents in E-Supply Chains: Realizing the Potential of Intelligent Informed intermediary-Based E-Marketplaces", Communications of the ACM, Vol. 48, No. 6, pp. 109-115"

Spitter, J. M.; Hurkens, C. A. J.; de Kok, A. G.; and Lenstra, J. K. (2005), "Linear Programming Models with Planned lead Times for Supply Chain Operations Planning", European Journal of Operational Research, Vol. 163, pp. 706-720.

Srinivasan, Kanna; Kekre, Sunder; Mukhopadhyay, Tridas (1994), " Impact of Electronic Data Interchange Technology on JIT Shipments", Management Science, Vol. 40, No. 10, pp. 1291-1304.

Sterman, John D. (1989), "Modeling Managerial Behavior: Misperceptions of Feedback in a Dynamic Decision Making Experiment", *Management Science*, Vol. 35, No. 3, pp. 321-339.

Stevens, Graham C. (1989), "Integrating the Supply Chains", *International Journal of Physical Distribution & Materials Management*, Vol. 8, No. 8, pp. 3-8.

Swaminathan, Jayashankar; Smith, Stephen F.; and Sadeh, Norman M. (1998), "Modeling Supply Chain Dynamics: A Multiagent Approach", *Decision Sciences Journal*, Vol. 29, No. 3, pp. 607-632.

Tan, Keah Choon (2000), "A Framework of Supply Chain Management Literature", *European Journal of Purchasing and Supply Management*, Vol. 7, pp. 39-48.

Terwiesch, Christian; Ren Z. Justin; Ho, Teck H.; and Cohen, Morris A. (2005), "An Empirical Analysis of Forecast Sharing in the Semiconductor Equipment Supply Chain", *Management Science*, Vol. 51, No. 2, pp. 208-220.

Thonemann, U. W. (2002), "Improving Supply-Chain Performance by Sharing Advance Demand Information", *European Journal of Operational Research*, Vol. 142, pp. 81-107.

Topkis, Donald M. (1968), "Optimal Ordering and Rationing Policies in a Nonstationary Dynamic Inventory Model with n Demand Classes", *Management Science*, Vol. 15, No. 3, pp. 160-176.

Tsay, Andy A. (1999), "The Quantity Flexibility Contract and Supplier-Customer Incentives", *Management Science*, Vol. 45, No. 10, pp. 1339-1358.

Tsvetovat, Maksim, and Carley, Kathleen M. (2004): "Modeling Complex Socio-technical Systems using Multi-Agent Simulation Methods". KI (www.kuenstliche-intelligenz.de), 18(2): 23-28.

Vives, Xavier (1984), "Duopoly Information Equilibrium: Cournot and Bertrand", Journal of Economic Theory, Vol. 34, pp. 71-94.

Wade, Michael; and Hulland, John (2004), "Review: The Resource-Based View and Information Systems Research: Review, Extension, and Suggestions FOR Future Research", MIS Quarterly, Vol. 28, No. 1, pp. 107. Walden, Eric A. (2000), "On the Structure and Function of Outsourcing Contracts: an Integrative Analysis of the Economics Behind Vendor-Client Relationships", Working Paper, MIS Research Center, University of Minnesota.

Whang, Seungjing (1992), "Contracting for Software Development", Management Science, Vol. 38, No. 3, pp. 307-324.

Whittaker, E. T. and Robinson, G. (1967), "Determination of the Constants in a Normal Frequency Distribution with Two Variables" and "The Frequencies of the Variables Taken Singly." §161-162 in The Calculus of Observations: A Treatise on Numerical Mathematics, 4th ed. New York: Dover, pp. 324-328.

Wooldridge, Michael, and Jennings, Nicholas R. (1995), "Intelligent Agents: Theory and Practice", Knowledge Engineering Review, Vol. 10, No. 2, pp. 115-152.

Zachman, J. A. (1987), "A Framework for Information Systems Architecture", IBM Systems Journal, Vol 26, No. 3, pp. 276 -292.

Zhang, Yiyi; Guo, Lei; and Georganas, Nicolas D. (2000), "AGILE: An Architecture for Agent-Based Collaborative and Interactive Virtual Environments", in: Proceedings of the Workshop on Application of Virtual Reality Technologies for Future Telecommunication Systems, IEEE Globecom 2000 Conference, San Francisco.

Zipkin, Paul (1989), "Critical Number Policies for Inventory Models with Periodic Data", Management Science, Vol. 35, No. 1, pp. 71-80.

Appendix The Java Code for the Model Implementation

File 1: model.score

```
<?xml version="1.0" encoding="UTF-8"?>

<score:SContext xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:score="http://scoreabm.org/score" label="sc" ID="sc" pluralLabel="scs">

  <attributes label="initialDemandAmount" ID="initialDemandAmount"
    pluralLabel="initialDemandAmounts" sType="INTEGER" defaultValue="100"
    units="unit"/>

  <attributes label="orderLag" ID="orderLag" pluralLabel="orderLags"
    sType="INTEGER" defaultValue="1" units="tick"/>

  <attributes label="Roo" ID="Roo" pluralLabel="Roos" sType="FLOAT"
    defaultValue="0.8"/>

  <attributes label="dValue" ID="dValue" pluralLabel="dValues" sType="INTEGER"
    defaultValue="100" units="unit"/>

  <attributes label="numAgent" ID="numAgent" pluralLabel="numAgents"
    sType="INTEGER" defaultValue="1"/>
```

```

<attributes label="GrowthRate" ID="GrowthRate" pluralLabel="GrowthRates"
sType="FLOAT" defaultValue="0.001" units="" />

<attributes label="Ratio12" ID="Ratio12" pluralLabel="Ratio12s" sType="FLOAT"
defaultValue="0" />

<attributes label="Ratio13" ID="Ratio13" pluralLabel="Ratio13s" sType="FLOAT"
defaultValue="0" />

<attributes label="Ratio23" ID="Ratio23" pluralLabel="Ratio23s" sType="FLOAT"
defaultValue="0" />

<attributes label="SharingCase" ID="SharingCase" pluralLabel="SharingCases"
sType="INTEGER" defaultValue="0" />

<attributes label="Sdev1" ID="Sdev1" pluralLabel="Sdev1s" sType="FLOAT"
defaultValue="30" />

<attributes label="Sdev2" ID="Sdev2" pluralLabel="Sdev2s" sType="FLOAT"
defaultValue="20" />

<attributes label="Sdev3" ID="Sdev3" pluralLabel="Sdev3s" sType="FLOAT"
defaultValue="10" />

<implementation package="repast.simphony.demo.sc" className="ContextCreator"
basePath="" />

<agents label="RetailerAgent" ID="retailerAgent" pluralLabel="RetailerAgents">

```

```
<implementation package="repast.simphony.demo.sc" className="RetailerAgent"/>

</agents>

<agents label="Moderator" ID="moderator" pluralLabel="Moderators">

<implementation package="repast.simphony.demo.sc" className="Moderator"
basePath="" />

</agents>

<agents label="testAgent2" ID="testAgent2" pluralLabel="testAgent2s">

<implementation className="testAgent2"/>

</agents>

<agents label="testAgent1" ID="testAgent1" pluralLabel="testAgent1s">

<implementation className="testAgent1"/>

</agents>

<agents label="RetailerAgent3" ID="retailerAgent3" pluralLabel="RetailerAgent3s">

<implementation package="repast.simphony.demo.sc" className="RetailerAgent3"
basePath="" />

</agents>
```

```
<agents label="SupplierAgent" ID="supplierAgent" pluralLabel="SupplierAgents">

    <implementation package="repast.simphony.demo.sc" className="SupplierAgent"/>

</agents>

<agents label="RetailerAgent2" ID="retailerAgent2" pluralLabel="RetailerAgent2s">

    <implementation package="repast.simphony.demo.sc" className="RetailerAgent2"/>

</agents>

<agents label="SCagent" ID="SCagent" pluralLabel="SCagents">

    <implementation className="SCagent"/>

</agents>

<projections xsi:type="score:SGrid" label="Grid" ID="grid" pluralLabel="Grids"/>

</score:SContext>
```

File 2: ContextCreator.java

```
package repast.simphony.demo.sc;

import repast.simphony.context.space.grid.GridFactoryFinder;
import repast.simphony.context.Context;
import repast.simphony.dataLoader.ContextBuilder;
import repast.simphony.space.grid.GridBuilderParameters;
import repast.simphony.space.grid.RandomGridAdder;
import repast.simphony.space.grid.WrapAroundBorders;

public class ContextCreator implements ContextBuilder<SCagent> {

    public Context<SCagent> build(Context<SCagent> context) {
```

```

int xdim = 30;

int ydim = 30;

//Change the code to the following to make the values customizable

//Need to add the variables to the score file.

//Parameters p = RunEnvironment.getInstance().getParameters();

//numberAgents = (Integer)p.getValue("numAgent");

//xdim = (Integer)p.getValue("gridWidth");

//ydim = (Integer)p.getValue("gridHeight");

GridFactoryFinder.createGridFactory(null).createGrid("Grid", context,
    new GridBuilderParameters<SCagent>(new
    WrapAroundBorders(),
    new RandomGridAdder<SCagent>(), false, xdim, ydim));

// create the agents and add to the context

```

```
/*
RetailerAgent agentR = null;

for (int r = 0; r < 2; r++)

{

    agentR = new RetailerAgent();

    //agentR.setName("AgentR-"+r);

    context.add(agentR);

}

*/
// for now, we start with one retailer agent and one supplier agent

RetailerAgent agentR1 = null;

agentR1 = new RetailerAgent();

agentR1.setName("AgentR1");

context.add(agentR1);

RetailerAgent2 agentR2 = null;
```

```
agentR2 = new RetailerAgent2();
```

```
agentR2.setName("AgentR2");
```

```
context.add(agentR2);
```

```
RetailerAgent3 agentR3 = null;
```

```
agentR3 = new RetailerAgent3();
```

```
agentR3.setName("AgentR3");
```

```
context.add(agentR3);
```

```
SupplierAgent agentS = null;
```

```
agentS = new SupplierAgent();
```

```
agentS.setName("AgentS");
```

```
context.add(agentS);
```

```
Moderator Mod = null;
```

```
Mod = new Moderator();
```

```
Mod.setName("Mod");

context.add(Mod);

testAgent1 TA1 = null;

TA1 = new testAgent1();

TA1.setName("TA1");

context.add(TA1);

testAgent2 TA2 = null;

TA2 = new testAgent2();

TA2.setName("TA2");

context.add(TA2);

return context;

}
```

File 3: scenario.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<Scenario>

<repast.simphony.dataLoader.engine.ClassNameDataLoaderAction context="sc"
file="repast.simphony.dataLoader.engine.ClassNameDataLoaderAction_0.xml" />

</Scenario>
```

File 4: SCagent.java

```
//This is the base agent class that implements some  
//fundamental methods common to all agents  
  
//Note that many parts of the code are borrowed from the Beer Game  
//implementation by Wolfgang Hoscheck  
  
package repast.simphony.demo.sc;  
  
import java.util.*;  
  
//import repast.engine.environment.RunEnvironment;  
  
import repast.simphony.engine.environment.RunEnvironment;  
  
//import repast.simphony.engine.schedule.ScheduledMethod;  
  
import repast.simphony.parameter.Parameters;  
  
//import repast.simphony.random.RandomHelper;
```

```
import repast.simphony.annotate.AgentAnnot;
```

```
@AgentAnnot(displayName = "SC Agent")
```

```
public class SCagent {
```

```
    public SCagent(){
```

```
        //the constructor
```

```
}
```

```
//total cost
```

```
protected double totalCostAvg;
```

```
//the unit cost of stock on hand
```

```
protected double invCost=0.1;
```

```
//the unit cost of stock in shortage
```

```
protected double shortCost=0.2;
```

```
//the cost of placing an order (no matter how big the order is)

protected double fixedOrderCost=0.0;

//the unit cost of goods for each order placed

protected double purchaseCost=0.4;

//the quantity of current order

protected int currentOrder;

//initial demand from the agent of lower level

protected int initialDemandAmount;

//number of retailer agents in the simulation

protected int numberAgent;

// the set of distribution variables

//-----  
protected int normalValue;  
  
protected ArrayList<Integer> normalStream = new ArrayList<Integer>();
```

```
protected int uniformValue;

protected ArrayList<Integer> uniformStream = new ArrayList<Integer>();

//-----
// The agent's name. This is set in the context creator

protected String theName;

protected String theName_Ls = "Supplier";

protected String theName_Lr1 = "Retailer1";

protected String theName_Lr2 = "Retailer2";

protected String theName_Lr3 = "Retailer3";

// List some label names for display purpose only

protected String stock_Ls = "Stock_s";
```

```
protected String stock_Lr1 = "Stock_r1";
```

```
protected String stock_Lr2 = "Stock_r2";
```

```
protected String stock_Lr3 = "Stock_r3";
```

```
protected String backorder_Ls = "backorder_s";
```

```
protected String backorder_Lr1 = "backorder_r1";
```

```
protected String backorder_Lr2 = "backorder_r2";
```

```
protected String backorder_Lr3 = "backorder_r3";
```

```
protected String shipment_Ls = "shipment_s";
```

```
protected String shipment_Lr1 = "shipment_r1";
```

```
protected String shipment_Lr2 = "shipment_r2";
```

```
protected String shipment_Lr3 = "shipment_r3";
```

```
protected String L1 = "lable_1";
```

```
protected String L2 = "lable_2";
```

```
protected String L3 = "lable_3";  
  
//protected double mean;  
  
//**** the unit of cost is in Dollars ****/  
  
// The current tick count, declared here, used by agents  
  
//protected int tickCount;  
  
// The constant for calculating next order value  
  
// This number is read in from the parameter list  
  
protected double Roo;  
  
protected int Elle;  
  
// standard deviation for calculating demand  
  
protected double Sdev1;
```

```
protected double Sdev2;  
  
protected double Sdev3;  
  
protected int dValue;  
  
protected double growthRate;  
  
protected int sharingCase;  
  
protected double Ratio12;  
  
protected double Ratio13;  
  
protected double Ratio23;  
  
// the demand history received from the demand agent  
/*  
//protected ArrayList demandHistory = new ArrayList();
```

* alpha is the parameter from Sterman: it is the stock adjustment to inventory,

* between desired inventory and actual inventory.

* Stock adjustment to inventory = alpha_s * (desired inventory - inventory).

* Note that our definition of alpha is the same as Sterman's alpha_s.

* We do not use an explicit alpha_sl since this can be found from beta.

*/

protected double alpha = 0.30;

/*

* theta is the parameter from Sterman:

* expected demand(t+1) = theta * demand(t) + (1 - theta)*expected demand(t)

*

* $0 \leq \theta \leq 1$

*/

protected double theta = 0.0;

/*

* beta is a parameter from Sterman: it is the relative weight attached to the

* pipeline vs. stock discrepancies from desired levels. beta = alpha_sl / alpha_s

*/

protected double beta = 0.15;

/*

* totalDesiredInventory is Sterman's "Q". It is a measure of the desired inventory

* relative to the desired pipeline.

* totalDesiredInventory = desired inventory + beta * desired pipeline

*/

protected double q = 17.00;

/*

// the inventory units.

protected double stock;

// the orders not fulfilled

protected double backorders;

```
// the inventory cost. In units of dollars per unit item per unit time period.
```

```
protected double inventoryCost = 0.50;
```

```
// the shortage cost. In units of dollars per unit item per unit time period.
```

```
protected double shortageCost = 2.00;
```

```
// the order cost. The cost per order that is fixed, does not include the cost
```

```
// of goods. In units of dollars per order.
```

```
protected double fixedOrderCost = 1.00;
```

```
// the purchase cost. The cost of goods ordered in units of dollars per unit.
```

```
protected double purchaseCost = 2.00;
```

```
// the order rate.
```

```
protected double currentOrder;
```

```
// the actual order that was demanded by the customer
```

```
protected double actualDemand;
```

```
// the forecasted order (demand) that was predicted by the forecaster
```

```
protected double forecastedDemand;
```

```
// the expected order tracker.
```

```
protected double lastExpectedOrder;

// the actual order tracker.

protected double ordersInPipeline;

// the total cost.

protected double totalCost;

// the historical (actual) demand.

protected LinkedList actualDemandHistory;

// the historical (forecasted) demand.

protected LinkedList forecastedDemandHistory;

*/



// method to find the mean of an arrayList of size greater than 0

public double getArrayMean ( LinkedList<Integer> A, int lengthOfMovingAvg){
```

```
int arraySize;  
  
int sum=0;  
  
arraySize = A.size();  
  
double mean;  
  
int theRange=0;  
  
  
  
  
if(arraySize < lengthOfMovingAvg){  
  
    theRange = 0;  
  
}  
  
else if (arraySize >=lengthOfMovingAvg){  
  
    theRange = arraySize - lengthOfMovingAvg;  
  
}  
  
  
  
  
for (int i = (arraySize-1); i>=theRange; i--){  
  
    sum += A.get(i);  
  
}
```

```
//find the mean

mean = sum/(arraySize-theRange);

return mean;

}

// method to find the mean of an arrayList of size greater than 0

// the input variable is the arrayList<Integer>

public double getSdev ( LinkedList<Integer> A, int lengthOfMovingAvg ){

    double Sdev=0.0;

    double variance=0.0;

    double theSquaredSum=0.0;

    int arraySize;

    double mean;

    int theRange=0;

    int sum=0;
```

```
arraySize = A.size();

if(arraySize < lengthOfMovingAvg){

    theRange = 0;

}

else if (arraySize >=lengthOfMovingAvg){

    theRange = arraySize - lengthOfMovingAvg;

}

for (int i = (arraySize-1); i>=theRange; i--){

    sum += A.get(i);

}

//find the mean

mean = sum/(arraySize-theRange);

//sum(Xi-mean)^2
```

```

for (int i = (arraySize-1); i>=theRange; i--){

    theSquaredSum += (A.get(i)-mean)* (A.get(i)-mean);

}

if (arraySize == 0){

    arraySize = 1;

}

//varince = (sum(Xi-mean)^2)/N

variance = (theSquaredSum)/arraySize;

Sdev= (int)Math.abs(Math.sqrt(variance));

return Sdev;

}

```

```
// get the current tick count

public int getTickCount(){

    return (int)

RunEnvironment.getInstance().getCurrentSchedule().getTickCount();

}

// get the initialized parameters

public void getInitParameters(){

    Parameters p = RunEnvironment.getInstance().getParameters();

    dValue = (Integer)p.getValue("dValue");

    Elle= (Integer)p.getValue("orderLag");

    Roo = (Double)p.getValue("Roo");

    growthRate = (Double)p.getValue("GrowthRate");

    sharingCase = (Integer)p.getValue("SharingCase");

    Ratio12 = (Double)p.getValue("Ratio12");

    Ratio13 = (Double)p.getValue("Ratio13");
```

```
Ratio23 = (Double)p.getValue("Ratio23");

Sdev1 = (Double)p.getValue("Sdev1");

Sdev2 = (Double)p.getValue("Sdev2");

Sdev3 = (Double)p.getValue("Sdev3");

}
```

```
public String setName(String name){

    return theName=name;

}
```

```
public String getName(){

    return theName;

}
```

```
public String getTheName_Ls(){
```

```
    return theName_Ls;
```

```
}
```

```
public String getTheName_Lr1(){
```

```
    return theName_Lr1;
```

```
}
```

```
public String getTheName_Lr2(){
```

```
    return theName_Lr2;
```

```
}
```

```
public String getTheName_Lr3(){
```

```
    return theName_Lr3;
```

```
}
```

```
public String getStock_Ls(){
```

```
    return stock_Ls;
```

```
}
```

```
public String getStock_Lr1(){
```

```
        return stock_Lr1;

    }

    public String getStock_Lr2(){

        return stock_Lr2;

    }

    public String getStock_Lr3(){

        return stock_Lr3;

    }

    public String getBackorder_Ls(){

        return backorder_Ls;

    }

    public String getBackorder_Lr1(){

        return backorder_Lr1;

    }

    }

    public String getBackorder_Lr2(){

        return backorder_Lr2;

    }
```

```
}

public String getBackorder_Lr3(){

    return backorder_Lr3;

}

public String getShipment_Ls(){

    return shipment_Ls;

}

public String getShipment_Lr1(){

    return shipment_Lr1;

}

public String getShipment_Lr2(){

    return shipment_Lr2;

}

public String getShipment_Lr3(){

    return shipment_Lr3;

}
```

```
public String getL1(){

    return L1;

}

public String getL2(){

    return L2;

}

public String getL3(){

    return L3;

}

public double getRatio12(){

    return Ratio12;

}
```

}

File 5: SupplierAgent.java

```
package repast.simphony.demo.sc;

import java.util.LinkedList;
//import java.util.ArrayList;
//import org.apache.commons.math.stat.descriptive.moment.StandardDeviation;
// see commons-math archive

import repast.simphony.engine.environment.RunEnvironment;
import repast.simphony.engine.schedule.ScheduledMethod;
import repast.simphony.parameter.Parameters;
import repast.simphony.random.RandomHelper;
import repast.simphony.annotate.AgentAnnot;
```

```
*****
```

/ The following is the working code for validation part of the dissertation.*

** note: to reactivate this code, need to also remove the marks in the middle*

** of the code which marks "continues..."*

```
@AgentAnnot(displayName="Supplier Agent")
```

```
public class SupplierAgent extends SCagent {
```

```
    // total stock (inventory) on hand
```

```
    protected int stock=0;
```

```
    // this is for calculating the average cost only
```

```
    protected int totalStock=0;
```

```
// the memory that keeps the history of stock levels

protected LinkedList<Integer> stockHistory = new LinkedList<Integer>();

//total backordered quantity

protected int backOrder=0;

// the memory that keeps the history of backorder levels

protected LinkedList<Integer> backHistory = new LinkedList<Integer>();

//the current and previous demand received from the retailers

protected int prevDemandS = 0;

protected int currentDemandS = 0;

// received orders from each agent (assume we have 3 agents)

protected int rcvdorder1 = 0;
```

```
protected int recvorder2 = 0;

protected int recvorder3 = 0;

protected int totalRevdOrder = 0;

// The storage for keeping the history of demand

protected LinkedList<Integer> demandStream=new LinkedList<Integer>();

// the current and previous orders placed by supplier to the upper stream

protected int currentOrderS = 0;

protected int totalPredicted = 0;

protected int prevOrderS = 0;

protected int currentOrder1 = 0;

protected int currentOrder2 = 0;

protected int currentOrder3 = 0;
```

```
// the memory for received orders from the retailer agents

protected LinkedList<Integer> rcvOrderHistory1 = new LinkedList<Integer>();

protected LinkedList<Integer> rcvOrderHistory2 = new LinkedList<Integer>();

protected LinkedList<Integer> rcvOrderHistory3 = new LinkedList<Integer>();

protected int YtPredict1;

protected int YtPredict2;

protected int YtPredict3;

protected int YtActual1;

protected int YtActual2;

protected int YtActual3;

protected static LinkedList<Integer> suppliedHistory1 = new  
LinkedList<Integer>();
```

```
protected static LinkedList<Integer> suppliedHistory2 = new  
LinkedList<Integer>();  
  
protected static LinkedList<Integer> suppliedHistory3 = new  
LinkedList<Integer>();  
  
// the memory for the total received orders  
  
protected LinkedList<Integer> totalRcvOrderHistory = new  
LinkedList<Integer>();  
  
protected LinkedList<Integer> totalRcvOrderLine = new LinkedList<Integer>();  
  
// the memory for orders placed by the supplier agent  
  
protected LinkedList<Integer> orderHistoryS = new LinkedList<Integer>();  
  
protected LinkedList<Integer> orderHistory1 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> orderHistory2 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> orderHistory3 = new LinkedList<Integer>();
```

```
// temp memory of the orders placed by the supplier agent

// this storage also serves as the received shipment from upper stream

// (the shipment queue from the upper stream) with a delay of "L"(Elle value).

protected LinkedList<Integer> orderLine = new LinkedList<Integer>(); // total
ordered

// shipment queue is also used to track individual predicted retailer demand

// orderLine = orderLine1+orderLine2+orderLine3

protected LinkedList<Integer> orderLine1 = new LinkedList<Integer>();

protected LinkedList<Integer> orderLine2 = new LinkedList<Integer>();

protected LinkedList<Integer> orderLine3 = new LinkedList<Integer>();

//shipment received by supplier agent from the upper stream

protected int shipment = 0;
```

```
// orders delivered to the retailers

//they are static variables to be accessed directly by the retailers

protected int supplied1 = 0;

protected int supplied2 = 0;

protected int supplied3 = 0;

//private int unfilledOrder1 = 0;

//private int unfilledOrder2 = 0;

//private int unfilledOrder3 = 0;

private double testValue;

protected LinkedList<Integer> unfilledOrderHistory1 = new
LinkedList<Integer>();
```

```
protected LinkedList<Integer> unfilledOrderHistory2 = new  
LinkedList<Integer>();  
  
protected LinkedList<Integer> unfilledOrderHistory3 = new  
LinkedList<Integer>();  
  
// the ratio of supplied (to Retailers) / stock  
  
// for calculating shipment to retailers  
  
// when stock is lower than needed  
  
private double fulfilRate = 0.0;  
  
//private int totalSupplied = 0;  
  
// the mean calculated for generating the error term of t+1  
  
protected double theMean1=0.0;  
  
protected double theMean2=0.0;  
  
protected double theMean3=0.0;
```

```
// the standard deviation of orders received calculated to generate the error term of  
t+1  
  
protected double theSdev1=0.0;  
  
protected double theSdev2=0.0;  
  
protected double theSdev3=0.0;  
  
  
  
  
  
  
// the standard deviation of orders to upper stream calculated for comparison  
  
protected double theSdevS1 = 0.0;  
  
protected double theSdevS2 = 0.0;  
  
protected double theSdevS3 = 0.0;  
  
  
  
  
  
  
  
  
// the error term generated from the normal stream  
  
protected int theErr1;  
  
protected int theErr2;  
  
protected int theErr3;
```

```
// the following are intermediate variables for calculating the next order
```

```
// since the formula is very long, we cut it into three parts
```

```
// part1 + part2 - part3
```

```
// order is placed for each retailer, so there are three sets
```

```
// part1 = d+Roo*YtActual(i)
```

```
// part2 = ((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * error(t+1)
```

```
// part3 = Roo*((1-Math.pow(Roo, (Elle+1)))/(1-Roo)) * error(t)
```

```
// error(t) = YtPredict(i) - YtActual(i)
```

```
protected int part1_1;
```

```
protected int part2_1;
```

```
protected int part3_1;
```

```
protected int part1_2;
```

```
protected int part2_2;
```

```
protected int part3_2;  
  
protected int part1_3;  
  
protected int part2_3;  
  
protected int part3_3;  
  
// if a manufacturer is added, these variables are useful, but not now  
  
//private LinkedList<Integer> shippingLine = new LinkedList<Integer>();  
  
//private LinkedList<Integer> productionLine = new LinkedList<Integer>();  
  
public SupplierAgent(){  
    //the constructor, initialize values  
    /*  
     final Double Sdev=10.0;
```

```
// The seed for generating demand

final int theDemandSeed = 0;

RandomHelper.setSeed(theDemandSeed);

// get the Normal distribution values (zero mean, constant variance)

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //      uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

/*
/* continues...

// get the customizable parameters
```

```

getInitParameters();

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

numberAgent = (Integer)p.getValue("numAgent");

//initial demand from each Retailer agents

rcvdorder1 = this.initialDemandAmount;

rcvdorder2 = this.initialDemandAmount;

rcvdorder3 = this.initialDemandAmount;

// total the retailers' orders

// this should also be the order placed by supplier to upper stream

totalRcvdOrder=rcvdorder1+rcvdorder2+rcvdorder3;

// add to the received total order to memory

totalRcvOrderHistory.addLast(this.totalRcvdOrder);

totalRcvOrderLine.addLast(this.totalRcvdOrder);

```

```
//add to the received order from retailer history memory
```

```
rcvOrderHistory1.addLast(this.rcvdorder1);
```

```
rcvOrderHistory2.addLast(this.rcvdorder2);
```

```
rcvOrderHistory3.addLast(this.rcvdorder3);
```

```
//add to the supplied-to-retail history memory
```

```
suppliedHistory1.addLast(this.initialDemandAmount);
```

```
suppliedHistory2.addLast(this.initialDemandAmount);
```

```
suppliedHistory3.addLast(this.initialDemandAmount);
```

```
unfilledOrderHistory1.addLast(0);
```

```
unfilledOrderHistory2.addLast(0);
```

```
unfilledOrderHistory3.addLast(0);
```

```
// initial state
```

```
stock = 0;

stockHistory.addLast(stock);

backOrder = 0;

backHistory.addLast(backOrder);

// get the initial supplier's order forecasts

// and add them to order history memory

this.currentOrderS=rcvdorder1 + rcvdorder2 + rcvdorder3;

// for the first L=1 period, orders are set

for(int i = 0; i < Elle; i++){

    this.orderLine.addLast(this.currentOrderS);

    this.orderHistoryS.addLast(this.currentOrderS);

    // add to the individual order memory for supplier

    orderLine1.addLast(this.initialDemandAmount);

    orderLine2.addLast(this.initialDemandAmount);
```

```
        orderLine3.addLast(this.initialDemandAmount);

        orderHistory1.addLast(this.initialDemandAmount);

        orderHistory2.addLast(this.initialDemandAmount);

        orderHistory3.addLast(this.initialDemandAmount);

    }

rcvdorder1 = 0;

rcvdorder2 = 0;

rcvdorder3 = 0;

}

@ScheduledMethod(start=1, interval=1)

public void calculateSupplier(){

    if (getTickCount() % 4 == 1){


```

```
// now the supplier will do the job

// 1. check inventory after having received the shipment from
upper stream

shipment= orderLine.getFirst();

// update the order line (the shipment queue from the upper stream)

orderLine.removeFirst();

// add the shipment to the current stock

this.stock += shipment;

totalStock = stock;

}

else if (getTickCount() % 4 == 2){

// do nothing at this tick

}
```

```
else if (getTickCount() % 4 == 3){
```

```
// do nothing at this tick
```

```
}
```

```
else if (getTickCount() % 4 == 0){
```

```
// 2. receive the order(s) from the retailers and update stock
```

```
//current period, this is actual Y(t) from retailer
```

```
//before doing all these, need to get the last retailer ordered
```

```
//to find the (Y(t)predict - Y(t)actual)
```

```
YtPredict1= orderLine1.getFirst();
```

```
orderLine1.removeFirst();
```

```
YtPredict2= orderLine2.getFirst();
```

```
orderLine2.removeFirst();
```

```
YtPredict3= orderLine3.getFirst();
```

```
orderLine3.removeFirst();

rcvdorder1 = RetailerAgent.orderLineR1.getLast();

//RetailerAgent.orderLineR1.removeFirst();

rcvOrderHistory1.addLast(this.rcvdorder1);

rcvdorder2 = RetailerAgent2.orderLineR2.getLast();

//RetailerAgent2.orderLineR2.removeFirst();

rcvOrderHistory2.addLast(this.rcvdorder2);

rcvdorder3 = RetailerAgent3.orderLineR3.getLast();

//RetailerAgent3.orderLineR3.removeFirst();

rcvOrderHistory3.addLast(this.rcvdorder2);

YtActual1 = rcvdorder1;

YtActual2 = rcvdorder2;

YtActual3 = rcvdorder3;
```

```
// total the orders

totalRcvdOrder=rcvdorder1+rcvdorder2+rcvdorder3;

// prepare the variable for calculating the order to upper stream

//prevDemandS = totalRcvOrderLine.getLast();

//currentDemandS = totalRcvdOrder;

// add the received total order to memory

totalRcvOrderHistory.addLast(this.totalRcvdOrder);

totalRcvOrderLine.addLast(this.totalRcvdOrder);

// the "supplied" variable is the current shipment (to-be)

// received by the retailer in next period

supplied1 = 0;

supplied2 = 0;

supplied3 = 0;
```

```
// taking into consideration of backorders of last period  
  
// as well as the scenario of smaller stock available  
  
// note that the backorder is the sum of individual unfilled orders  
  
// (totalRcvdOrder + backOrder) is the total of current received  
  
// order plus back orders  
  
  
  
//unfilledOrder1 = unfilledOrderHistory1.getLast();  
  
//unfilledOrder2 = unfilledOrderHistory2.getLast();  
  
//unfilledOrder3 = unfilledOrderHistory3.getLast();  
  
  
  
  
/* backorder handled differently  
  
if( stock >= (totalRcvdOrder + backOrder) ){  
  
    supplied1 = unfilledOrder1 + rcvdorder1;  
  
    supplied2 = unfilledOrder2 + rcvdorder2;  
  
    supplied3 = unfilledOrder3 + rcvdorder3;
```

```
//add to the supplied-to-retail history memory  
  
suppliedHistory1.addLast(this.supplied1);  
  
suppliedHistory2.addLast(this.supplied2);  
  
suppliedHistory3.addLast(this.supplied3);  
  
  
  
  
this.stock -= (totalRcvdOrder + backOrder);  
  
// fulfill (update) the backorders  
  
backOrder = 0;  
  
unfilledOrder1 = 0;  
  
unfilledOrder2 = 0;  
  
unfilledOrder3 = 0;  
  
  
  
  
unfilledOrderHistory1.addLast(unfilledOrder1);  
  
unfilledOrderHistory2.addLast(unfilledOrder2);  
  
unfilledOrderHistory3.addLast(unfilledOrder3);
```

```
// restore the initial values

supplied1 = 0;

supplied2 = 0;

supplied3 = 0;

}

else if( stock < (totalRcvdOrder + backOrder) ){

    // if stock could not even fulfill the previous order

    // supplied will be only a proportion of previous demand

    from retailers

        if((this.stock - backOrder) < 0){

            if(backOrder == 0){

                fulfilRate = 1;
```

```
    }

    else{

        fulfilRate = Math.abs(stock / backOrder);

    }

    supplied1 = (int)Math.floor(unfilledOrder1 *

fulfilRate);

    supplied2 = (int)Math.floor(unfilledOrder2 *

fulfilRate);

    supplied3 = (int)Math.floor(unfilledOrder3 *

fulfilRate);

//add to the supplied-to-retail history memory

suppliedHistory1.addLast(this.supplied1);

suppliedHistory2.addLast(this.supplied2);

suppliedHistory3.addLast(this.supplied3);

// current received orders go to backorder directly
```

```
unfilledOrder1 += (rcvdorder1 - supplied1);

unfilledOrder2 += (rcvdorder2 - supplied2);

unfilledOrder3 += (rcvdorder3 - supplied3);

unfilledOrderHistory1.addLast(unfilledOrder1);

unfilledOrderHistory2.addLast(unfilledOrder2);

unfilledOrderHistory3.addLast(unfilledOrder3);

backOrder = unfilledOrder1 + unfilledOrder2 +
unfilledOrder3;

stock = 0;

// restore the initial values

supplied1 = 0;

supplied2 = 0;

supplied3 = 0;

}
```

```

// if stock satisfies the previous backorders, but not all
orders from Retailers

else if((this.stock - backOrder) >= 0){

    // this is not possible, but in case there is a bug in
the program

    if(totalRcvdOrder == 0){

        //simply split the stock

        fulfilRate = 0;

    }

    else{

        fulfilRate = Math.abs(stock-backOrder) /
totalRcvdOrder;

    }

}

unfilledOrder1 = unfilledOrderHistory1.getLast();

unfilledOrder2 = unfilledOrderHistory2.getLast();

```

```

unfilledOrder3 = unfilledOrderHistory3.getLast();

supplied1 = (int)Math.floor(rcvdorder1 * fulfilRate)
+ unfilledOrder1;

supplied2 = (int)Math.floor(rcvdorder3 * fulfilRate)
+ unfilledOrder2;

supplied3 = (int)Math.floor(rcvdorder3 * fulfilRate)
+ unfilledOrder3;

//add to the supplied-to-retail history memory

suppliedHistory1.addLast(this.supplied1);

suppliedHistory2.addLast(this.supplied2);

suppliedHistory3.addLast(this.supplied3);

backOrder = totalRcvdOrder + backOrder - stock;

stock = 0;

```

```
        unfilledOrder1 += rcvdorder1 - supplied1;

        unfilledOrder2 += rcvdorder2 - supplied2;

        unfilledOrder3 += rcvdorder3 - supplied3;

        unfilledOrderHistory1.addLast(unfilledOrder1);

        unfilledOrderHistory2.addLast(unfilledOrder2);

        unfilledOrderHistory3.addLast(unfilledOrder3);

        // restore the initial values

        supplied1 = 0;

        supplied2 = 0;

        supplied3 = 0;

    }

    rcvdorder1 = 0;

    rcvdorder2 = 0;
```

```
    rcvdorder3 = 0;  
  
}  
  
*/  
  
/* continues...  
  
stock -= totalRcvdOrder;  
  
// stock < 0? hopefully not, backorder means lost sales  
  
if (stock < 0){  
  
    backOrder = (int)Math.abs(stock);  
  
    stock = 0;  
  
}  
  
else if (stock >= 0){  
  
    backOrder = 0;  
  
}
```

```
stockHistory.addLast(stock);

backHistory.addLast(backOrder);

// 3. calculate the order to the upper stream

// assume all supplier's orders are fulfilled from the upper stream.

// uses the AR1 process described in Lee, et al. (2000)

// need to change to moving average

//mean of past orders received from each retailer

theMean1 =0.0;

theMean2 =0.0;

theMean3 =0.0;
```

```
testValue = getArrayMean(rcvOrderHistory1, 30);

//standard deviation of past orders received from each retailer

theSdev1 = getSdev(rcvOrderHistory1,30);

theSdev2 = getSdev(rcvOrderHistory2,30);

theSdev3 = getSdev(rcvOrderHistory3,30);

//standard deviation of past orders placed with the upper stream

//when info is not shared

theSdevS1 = getSdev(orderHistory1, 30);

theSdevS2 = getSdev(orderHistory2, 30);

theSdevS3 = getSdev(orderHistory3, 30);

//when shared info
```

```
//theSdevS1 = 30;  
  
//theSdevS2 = 40;  
  
//theSdevS3 = 20;  
  
  
  
  
RandomHelper.setSeed(getTickCount());  
  
// get the Normal distribution value: retailer 1  
  
RandomHelper.createNormal(theMean1, theSdev1);  
  
theErr1 = RandomHelper.getNormal().nextInt();  
  
  
  
  
// get the Normal distribution value: retailer 2  
  
RandomHelper.createNormal(theMean2, theSdev2);  
  
theErr2 = RandomHelper.getNormal().nextInt();  
  
  
  
  
// get the Normal distribution value: retailer 3  
  
RandomHelper.createNormal(theMean3, theSdev3);  
  
theErr3 = RandomHelper.getNormal().nextInt();
```

```

// the AR1 formula, the order quantity considers the change in
order-up-to levels

// get each of three parts for the calculation

// retailer 1

part1_1 = dValue + (int)Math.floor(Roo*YtActual1);

part2_1 = (int)((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr1;

// when info is not shared

part3_1 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* (YtPredict1 -
YtActual1)) /(1-Roo));

// when info is shared

//part3_1 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))*
RetailerAgent.getSharedErr1() /(1-Roo)));

// sum them up

currentOrder1 = Math.max(0,part1_1 + part2_1 - part3_1);

```

```

// retailer 2

part1_2 = dValue + (int)Math.floor(Roo*YtActual2);

part2_2 = (int)((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr2;

// when info is not shared

part3_2 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* (YtPredict2 -
YtActual2)) /(1-Roo));

// when info is shared

//part3_2 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))*
RetailerAgent2.getSharedErr2() /(1-Roo)));

// sum them up

currentOrder2 = Math.max(0,part1_2 + part2_2 - part3_2);

// retailer 3

part1_3 = dValue + (int)Math.floor(Roo*YtActual3);

```

```

part2_3 = (int) (((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr3);

// when info is not shared

part3_3 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))*(YtPredict3 -
YtActual3)) /(1-Roo));

// when info is shared

//part3_3 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))*
RetailerAgent3.getSharedErr3()) /(1-Roo));

// sum them up

currentOrder3 = Math.max(0,part1_3 + part2_3 - part3_3);

orderLine1.addLast(currentOrder1);

orderLine2.addLast(currentOrder2);

orderLine3.addLast(currentOrder3);

orderHistory1.addLast(currentOrder1);

```

```
orderHistory2.addLast(currentOrder2);

orderHistory3.addLast(currentOrder3);

// the total order is the sum of individual orders

this.currentOrderS = currentOrder1 + currentOrder2 +
currentOrder3;

// backorders also need to be ordered

//this.currentOrderS += backOrder;

//this.currentOrderS = Math.max(0, currentOrderS+backOrder-
stock);

totalPredicted = currentOrderS;

this.orderHistoryS.addLast(currentOrderS);

// if stock is high, set order to zero
```

```
if(stock-backOrder >= currentOrderS){

    currentOrderS = 0;

}

//if (stock<=0){

//    this.currentOrderS += backOrder;

//}

// add the orders to memory

this.orderLine.addLast(currentOrderS);

// 4. calculate the total cost per unit shipped

// in case total stock is zero

//if (totalStock == 0){

//    totalStock = 1;

//}
```

```
        this.totalCostAvg = (this.invCost * this.stock + this.shortCost *  
        this.backOrder +  
  
        this.fixedOrderCost *  
        ((this.currentOrderS > 0)?1 : 0)+  
  
        this.purchaseCost * this.shipment);  
  
    }  
  
}
```

```
public static int getShipmentToR1(){  
  
    return suppliedHistory1.getLast();  
  
}  
  
public static int getShipmentToR2(){  
  
    return suppliedHistory3.getLast();  
  
}  
  
public static int getShipmentToR3(){
```

```
        return suppliedHistory3.getLast();

    }

public int isSupplier(){

    return 1;

}

public double getTotalCost(){

    return this.totalCostAvg;

}

public int getTested(){

    return this.backOrder - this.stock;

}

public int getStock(){
```

```
    return stock;

}

public int getBackOrder(){

    return backOrder;

}

public int getShipment(){

    return this.shipment;

}

public int getCurrentOrderS(){

    return currentOrderS;

}

public double getFulfilRate(){
```

```
    return fulfilRate;

}

public int getCurrentDemand(){

    return currentDemandS;

}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

public int getTheErr1(){

    return theErr1;

}
```

```
public int getTheErr2(){  
    return theErr2;  
}
```

```
public int getTheErr3(){  
    return theErr3;  
}
```

```
public int getPart1_1(){  
    return part1_1;  
}
```

```
public int getPart2_1(){  
    return part2_1;  
}
```

```
public int getPart3_1(){
```

```
    return part3_1;
```

```
}
```

```
public int getPart1_2(){
```

```
    return part1_1;
```

```
}
```

```
public int getPart2_2(){
```

```
    return part2_1;
```

```
}
```

```
public int getPart3_2(){
```

```
    return part3_1;
```

```
}
```

```
public int getPart1_3(){
```

```
    return part1_1;
```

```
}
```

```
public int getPart2_3(){
```

```
    return part2_1;
```

```
}
```

```
public int getPart3_3(){
```

```
    return part3_1;
```

```
}
```

```
public int getCurrentOrder1(){
```

```
    return currentOrder1;
```

```
}
```

```
public int getCurrentOrder2(){
```

```
        return currentOrder2;

    }

    public int getCurrentOrder3(){

        return currentOrder3;

    }

    public int getRcvdorder1(){

        return rcvdorder1;

    }

    public int getRcvdorder2(){

        return rcvdorder2;

    }

    public int getRcvdorder3(){

        return rcvdorder3;

    }

    public double getTheSdev1(){

        return theSdev1;

    }
```

```
}

public double getTheSdev2(){

    return theSdev2;

}

public double getTheSdev3(){

    return theSdev3;

}

// get the variance

public double getTheVar1(){

    return theSdev1*theSdev1;

}

public double getTheVar2(){

    return theSdev2*theSdev2;

}

public double getTheVar3(){
```

```
    return theSdev3*theSdev3;

}
```

```
public double getMean(){

    return testValue;

}
```

```
public int getYtActual1(){

    return YtActual1;

}
```

```
public int getYtPredict1(){

    return YtPredict1;

}
```

```
public int getYtActual2(){

    return YtActual2;

}
```

```
public int getYtPredict2(){
```

```
        return YtPredict2;

    }

    public int getYtActual3(){

        return YtActual3;

    }

    public int getYtPredict3(){

        return YtPredict3;

    }

}

public double getTheSdevS1(){

    return theSdevS1;

}

public double getTheSdevS2(){

    return theSdevS2;

}

public double getTheSdevS3(){
```

```
    return theSdevS3;

}

// get the variance

public double getTheVarS1(){

    return theSdevS1*theSdevS1;

}

public double getTheVarS2(){

    return theSdevS2*theSdevS2;

}

public double getTheVarS3(){

    return theSdevS3*theSdevS3;

}

public double getRoo(){

    return Roo;

}

public int getTotalRcvdOrder(){
```

```
        return totalRcvdOrder;

    }

    public int getTotalPredicted(){

        return totalPredicted;

    }

}

*/
*****/* The following is the code for testing the interactions
/*

```

*/

@AgentAnnot(displayName="Supplier Agent")

public class SupplierAgent extends SCagent {

// total stock (inventory) on hand

protected int stock=0;

// this is for calculating the average cost only

protected int totalStock=0;

// the memory that keeps the history of stock levels

protected LinkedList<Integer> stockHistory = new LinkedList<Integer>();

//total backordered quantity

```
protected int backOrder=0;

// the memory that keeps the history of backorder levels

protected LinkedList<Integer> backHistory = new LinkedList<Integer>();

//the current and previous demand received from the retailers

protected int prevDemandS = 0;

protected int currentDemandS = 0;

// received orders from each agent (assume we have 3 agents)

protected int rcvdorder1 = 0;

protected int rcvdorder2 = 0;

protected int rcvdorder3 = 0;

protected int totalRcvdOrder = 0;

// The storage for keeping the history of demand
```

```
protected LinkedList<Integer> demandStream=new LinkedList<Integer>();  
  
// the current and previous orders placed by supplier to the upper stream  
  
protected int currentOrderS = 0;  
  
protected int totalPredicted = 0;  
  
protected int prevOrderS = 0;  
  
protected int currentOrder1 = 0;  
  
protected int currentOrder2 = 0;  
  
protected int currentOrder3 = 0;  
  
// the memory for received orders from the retailer agents  
  
protected LinkedList<Integer> rcvOrderHistory1 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> rcvOrderHistory2 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> rcvOrderHistory3 = new LinkedList<Integer>();
```

```
protected int YtPredict1;

protected int YtPredict2;

protected int YtPredict3;

protected int YtActual1;

protected int YtActual2;

protected int YtActual3;

protected static LinkedList<Integer> suppliedHistory1 = new  
LinkedList<Integer>();  
  
protected static LinkedList<Integer> suppliedHistory2 = new  
LinkedList<Integer>();  
  
protected static LinkedList<Integer> suppliedHistory3 = new  
LinkedList<Integer>();  
  
// total shipped to retailers each period  
  
protected LinkedList<Integer> suppliedHistory = new LinkedList<Integer>();
```

```
// the memory for the total received orders

protected LinkedList<Integer> totalRcvOrderHistory = new
LinkedList<Integer>();

protected LinkedList<Integer> totalRcvOrderLine = new LinkedList<Integer>();

// the memory for orders placed by the supplier agent

protected LinkedList<Integer> orderHistoryS = new LinkedList<Integer>();

protected LinkedList<Integer> orderHistory1 = new LinkedList<Integer>();

protected LinkedList<Integer> orderHistory2 = new LinkedList<Integer>();

protected LinkedList<Integer> orderHistory3 = new LinkedList<Integer>();

// temp memory of the orders placed by the supplier agent

// this storage also serves as the received shipment from upper stream

// (the shipment queue from the upper stream) with a delay of "L"(Elle value).
```

```
protected LinkedList<Integer> orderLine = new LinkedList<Integer>(); // total  
ordered  
  
// shipment queue is also used to track individual predicted retailer demand  
  
// orderLine = orderLine1+orderLine2+orderLine3  
  
protected LinkedList<Integer> orderLine1 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> orderLine2 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> orderLine3 = new LinkedList<Integer>();  
  
//shipment received by supplier agent from the upper stream  
  
protected int shipment = 0;  
  
// orders delivered to the retailers  
  
//they are static variables to be accessed directly by the retailers  
  
protected int supplied1 = 0;
```

```
protected int supplied2 = 0;

protected int supplied3 = 0;

private int unfilledOrder1 = 0;

private int unfilledOrder2 = 0;

private int unfilledOrder3 = 0;

private double testValue;

protected LinkedList<Integer> unfilledOrderHistory1 = new
LinkedList<Integer>();

protected LinkedList<Integer> unfilledOrderHistory2 = new
LinkedList<Integer>();

protected LinkedList<Integer> unfilledOrderHistory3 = new
LinkedList<Integer>();

// the ratio of supplied (to Retailers) / stock
```

```
// for calculating shipment to retailers

// when stock is lower than needed

private double fulfilRate = 0.0;

//private int totalSupplied = 0;

// the mean calculated for generating the error term of t+1

protected double theMean1=0.0;

protected double theMean2=0.0;

protected double theMean3=0.0;

// the standard deviation of orders received calculated to generate the error term of

t+1

protected double theSdev1=0.0;

protected double theSdev2=0.0;

protected double theSdev3=0.0;
```

```
// the standard deviation of orders to upper stream calculated for comparison

protected double theSdevS1 = 0.0;

protected double theSdevS2 = 0.0;

protected double theSdevS3 = 0.0;

// the error term generated from the normal stream

protected int theErr1;

protected int theErr2;

protected int theErr3;

// the following are intermediate variables for calculating the next order

// since the formula is very long, we cut it into three parts

// part1 + part2 - part3

// order is placed for each retailer, so there are three sets

// part1 = d+Roo*YtActual(i)
```

```
// part2 = ((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * error(t+1)

// part3 = Roo*((1-Math.pow(Roo, (Elle+1)))/(1-Roo)) * error(t)

// error(t) = YtPredict(i) - YtActual(i)

protected int part1_1;

protected int part2_1;

protected int part3_1;

protected int part1_2;

protected int part2_2;

protected int part3_2;

protected int part1_3;

protected int part2_3;

protected int part3_3;
```

```
// if a manufacturer is added, these variables are useful, but not now

//private LinkedList<Integer> shippingLine = new LinkedList<Integer>();

//private LinkedList<Integer> productionLine = new LinkedList<Integer>();

public SupplierAgent(){

    //the constructor, initialize values

    /*

    final Double Sdev=10.0;

    // The seed for generating demand

    final int theDemandSeed = 0;

    RandomHelper.setSeed(theDemandSeed);

    // get the Normal distribution values (zero mean, constant variance)

    RandomHelper.createNormal(0.0, Sdev);

    //RandomHelper.createUniform(0, 25);
```

```

for(int i=0; i<10000; i++){

    //      uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());


}

/*
// get the customizable parameters

getInitParameters();

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

numberAgent = (Integer)p.getValue("numAgent");

//initial demand from each Retailer agents

rcvdorder1 = this.initialDemandAmount;

rcvdorder2 = this.initialDemandAmount;

rcvdorder3 = this.initialDemandAmount;

```

```
// total the retailers' orders

// this should also be the order placed by supplier to upper stream

totalRcvdOrder=rcvdorder1+rcvdorder2+rcvdorder3;

// add to the received total order to memory

totalRcvOrderHistory.addLast(this.totalRcvdOrder);

totalRcvOrderLine.addLast(this.totalRcvdOrder);

//add to the received order from retailer history memory

rcvOrderHistory1.addLast(this.rcvdorder1);

rcvOrderHistory2.addLast(this.rcvdorder2);

rcvOrderHistory3.addLast(this.rcvdorder3);

//add to the supplied-to-retail history memory

suppliedHistory1.addLast(this.initialDemandAmount);

suppliedHistory2.addLast(this.initialDemandAmount);
```

```
suppliedHistory3.addLast(this.initialDemandAmount);

// add to the supplied to retailers total memory

suppliedHistory.addLast(this.supplied1+this.supplied2+this.supplied3);

unfilledOrderHistory1.addLast(0);

unfilledOrderHistory2.addLast(0);

unfilledOrderHistory3.addLast(0);

// initial state

stock = 0;

stockHistory.addLast(stock);

backOrder = 0;

backHistory.addLast(backOrder);

// get the initial supplier's order forecasts
```

```
// and add them to order history memory

this.currentOrderS=rcvdorder1 + rcvdorder2 + rcvdorder3;

// for the first L=1 period, orders are set

for(int i = 0; i < Elle; i++){
    this.orderLine.addLast(this.currentOrderS);

    this.orderHistoryS.addLast(this.currentOrderS);

    // add to the individual order memory for supplier

    orderLine1.addLast(this.initialDemandAmount);

    orderLine2.addLast(this.initialDemandAmount);

    orderLine3.addLast(this.initialDemandAmount);

    orderHistory1.addLast(this.initialDemandAmount);

    orderHistory2.addLast(this.initialDemandAmount);

    orderHistory3.addLast(this.initialDemandAmount);

}
```

```
    rcvdorder1 = 0;

    rcvdorder2 = 0;

    rcvdorder3 = 0;

}

{@ScheduledMethod(start=1, interval=1)

public void calculateSupplier(){

    if (getTickCount() % 4 == 1){

        // now the supplier will do the job

        // 1. check inventory after having received the shipment from
        upper stream

        shipment= orderLine.getFirst();

        // update the order line (the shipment queue from the upper stream)
```

```
    orderLine.removeFirst();

    // get the current stock

    this.stock = getStock();

    // add the shipment to the current stock

    this.stock += shipment;

    totalStock = stock;

}

else if (getTickCount() % 4 == 2){

    // do nothing at this tick

}

else if (getTickCount() % 4 == 3){

    // do nothing at this tick

}
```

```
else if (getTickCount() % 4 == 0){  
  
    // 2. receive the order(s) from the retailers and update stock  
  
    //current period, this is actual Y(t) from retailer  
  
    //before doing all these, need to get the last retailer ordered  
  
    //to find the (Y(t)predict - Y(t)actual)  
  
  
  
    YtPredict1= orderLine1.getFirst();  
  
    orderLine1.removeFirst();  
  
    YtPredict2= orderLine2.getFirst();  
  
    orderLine2.removeFirst();  
  
    YtPredict3= orderLine3.getFirst();  
  
    orderLine3.removeFirst();  
  
  
  
    rcvdorder1 = RetailerAgent.orderLineR1.getLast();  
  
    //RetailerAgent.orderLineR1.removeFirst();
```

```
        rcvOrderHistory1.addLast(this.rcvdorder1);

        rcvdorder2 = RetailerAgent2.orderLineR2.getLast();

//RetailerAgent2.orderLineR2.removeFirst();

        rcvOrderHistory2.addLast(this.rcvdorder2);

        rcvdorder3 = RetailerAgent3.orderLineR3.getLast();

//RetailerAgent3.orderLineR3.removeFirst();

        rcvOrderHistory3.addLast(this.rcvdorder2);

YtActual1 = rcvdorder1;

YtActual2 = rcvdorder2;

YtActual3 = rcvdorder3;

// total the orders

totalRcvdOrder=rcvdorder1+rcvdorder2+rcvdorder3;

// prepare the variable for calculating the order to upper stream
```

```
//prevDemandS = totalRcvOrderLine.getLast();

//currentDemandS = totalRcvdOrder;

// add the received total order to memory

totalRcvOrderHistory.addLast(this.totalRcvdOrder);

totalRcvOrderLine.addLast(this.totalRcvdOrder);

// the "supplied" variable is the current shipment (to-be)

// received by the retailer in next period

supplied1 = 0;

supplied2 = 0;

supplied3 = 0;

// taking into consideration of backorders of last period

// as well as the scenario of smaller stock available

// note that the backorder is the sum of individual unfilled orders
```

```
/* backorder handled differently in different cases*/  
  
/*-----*/  
  
// before start, we need find out the backorders of last period for  
each retailer  
  
unfilledOrder1 = unfilledOrderHistory1.getLast();  
  
unfilledOrder2 = unfilledOrderHistory2.getLast();  
  
unfilledOrder3 = unfilledOrderHistory3.getLast();  
  
// as well as the total backorder so far  
  
// in fact this is the total of individual accumulated backorders.  
  
backOrder = getBackOrder();
```

```
if((this.stock - backOrder) >= 0){  
  
    // now that we have sufficient stock for the backorder  
  
    // let's find if the stock is good for the received orders  
  
    // first, get the stock after backorders  
  
    this.stock -= backOrder;  
  
    // then consider the received orders  
  
    // first, we have plenty of stock  
  
    // all orders will be fulfilled  
  
    if((this.stock-totalRcvdOrder) >= 0){  
  
        stock -= totalRcvdOrder;  
  
        fulfilRate = 1.0; // this will not be necessary  
  
        ////////////////////////////////  
    }  
}
```

```
// now that we have got the fulfilRate  
  
// it is time to compute the shipment that will be  
delievered next period  
  
// this calculation is based on the received order of  
this period  
  
// so that the back orders will be fulfilled faster  
  
// thus a retailer is receiving his ordered quantity of  
(L+L) periods ago, plus  
  
// (if there are any) the backordered quantity of last  
period; note here, the last period backorder.  
  
// All backorders (including the costs) in their case  
are ignored  
  
// Also, this backorder info should be used when  
computing the next order by supplier
```

supplied1 = unfilledOrder1 + rcvdorder1;

supplied2 = unfilledOrder2 + rcvdorder2;

supplied3 = unfilledOrder3 + rcvdorder3;

```
// all backorders become zero - the initial state
```

```
backOrder = 0;
```

```
unfilledOrder1 = 0;
```

```
unfilledOrder2 = 0;
```

```
unfilledOrder3 = 0;
```

```
}
```

```
//then in another case, the stock is not sufficient
```

```
// we need to calculate the fulfil rate
```

```
else if((this.stock-totalRcvdOrder) < 0){
```

```
// the remaining stock is everything we can offer to
```

```
the retailers
```

```

// this case (totalRcvdOrder == 0) is not possible
since stock > 0,

// but in case there is a bug in the program and to
avoid the case of "devided by zero"

if(totalRcvdOrder == 0){

    //no order received, so no shipment

    fulfilRate = 0;

}

else{

    fulfilRate = this.stock / totalRcvdOrder;

}

///////////////////////////////
// now that we have got the fulfilRate

// it is time to compute the shipment that will be
delievered next period

```

// this calculation is based on the received order of
this period

// so that the back orders will be fulfilled faster

// thus a retailer is receiving his ordered quantity of
(L+L) periods ago, plus

// (if there are any) the backordered quantity of last
period; note here, the last period backorder.

// All backorders (including the costs) in their case
are ignored

// Also, this backorder info should be used when
computing the next order by supplier

supplied1 = (int)Math.floor(rcvdorder1 * fulfilRate)
+ unfilledOrder1;

supplied2 = (int)Math.floor(rcvdorder3 * fulfilRate)
+ unfilledOrder2;

supplied3 = (int)Math.floor(rcvdorder3 * fulfilRate)
+ unfilledOrder3;

```
// since stock >= backorder, previous backorders are  
all fulfilled
```

```
// thus the new backorders are only the current ones
```

```
unfilledOrder1 = (int)Math.floor(rcvdorder1 * (1 -  
fulfilRate));
```

```
unfilledOrder2 = (int)Math.floor(rcvdorder2 * (1 -  
fulfilRate));
```

```
unfilledOrder3 = (int)Math.floor(rcvdorder3 * (1 -  
fulfilRate));
```

```
// update the back order
```

```
// can use the totalRcvdOrder - stock to get the same  
answer
```

```
backOrder = unfilledOrder1 + unfilledOrder2 +  
unfilledOrder3;
```

```
// update the stock
```

```

        this.stock = 0;

    }

}

// this next case should also be considered and the calcuation of
supplied is different

else if((this.stock - backOrder) < 0){

    // total available for shipment will be the current stock only

    // this stock needs to be shared proportionally by the
retailers

    // the fulfilRate needs to be calculated here

    // this case (backOrder == 0) is not possible, but in case
there is a bug in the program

    // and to avoid the case of "devided by zero"

```

```

        if(backOrder == 0){

            //no backorder, so no share

            fulfilRate = 0;

        }

        else{

            fulfilRate = this.stock / backOrder;

        }

        // now that we have got the fulfilRate

        // it is time to compute the shipment that will be delivered

        next period

        // this calculation is based on the received order of this

        period

        // so that the back orders will be fulfilled faster

        // thus a retailer is receiving his (accumulated, if any)

        backorder quantity only,
    
```

// all new order becomes the new backorder to be fulfilled
next period

// Lee et al (2000)'s model does not consider the backorder
by retailers and the suppliers.

// All backorders (including the costs) in their case are
ignored

// Also, this backorder info should be used when computing
the next order by supplier

// the supplied (shipment to retailers) are only for partially
fufilling the backorders

// this is a very rare case

supplied1 = (int)Math.floor(unfilledOrder1 * fulfilRate);

supplied2 = (int)Math.floor(unfilledOrder1 * fulfilRate);

supplied3 = (int)Math.floor(unfilledOrder1 * fulfilRate);

```
// since stock < backorder, previous backorders are not all  
fulfilled
```

```
// thus the new backorders are only the current order plus  
the left over
```

```
unfilledOrder1 = unfilledOrder1 - supplied1 + rcvdorder1;
```

```
unfilledOrder2 = unfilledOrder2 - supplied2 + rcvdorder2;
```

```
unfilledOrder3 = unfilledOrder3 - supplied3 + rcvdorder3;
```

```
// update the stock info
```

```
this.stock = 0;
```

```
// update the back order
```

```
// can use the backOrder - stock + totalRcvdOrder to get the  
same answer
```

```
backOrder = unfilledOrder1 + unfilledOrder2 +  
unfilledOrder3;
```

```
}
```

```
// after all above calculations, now we do some paper work and  
store the data
```

```
// add the individual backorders to memory for next period's  
calculation
```

```
unfilledOrderHistory1.addLast(unfilledOrder1);
```

```
unfilledOrderHistory2.addLast(unfilledOrder2);
```

```
unfilledOrderHistory3.addLast(unfilledOrder3);
```

```
// add to the supplied-to-retailer history memory
```

```
suppliedHistory1.addLast(this.supplied1);
```

```
suppliedHistory2.addLast(this.supplied2);
```

```
suppliedHistory3.addLast(this.supplied3);
```

```
// add to the supplied to retailers total memory

suppliedHistory.addLast(this.supplied1+this.supplied2+this.supplied3);

/* backorder handled differently

stock -= totalRcvdOrder;

// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}

else if (stock >= 0){

    backOrder = 0;

}
```

```
*/  
  
// store the stock and backorder data to memory  
  
stockHistory.addLast(stock);  
  
backHistory.addLast(backOrder);  
  
// all backorders become zero - the initial state  
  
backOrder = 0;  
  
unfilledOrder1 = 0;  
  
unfilledOrder2 = 0;  
  
unfilledOrder3 = 0;  
  
// the stock is back to initial state  
  
stock = 0;
```

```
// 3. calculate the order to the upper stream  
  
// assume all supplier's orders are fulfilled from the upper stream.  
  
// uses the AR1 process described in Lee, et al. (2000)  
  
  
  
// need to change to moving average  
  
  
  
//mean of past orders received from each retailer  
  
  
  
  
  
theMean1 =0.0;  
  
theMean2 =0.0;  
  
theMean3 =0.0;  
  
  
  
  
  
  
  
  
  
testValue = getArrayMean(rcvOrderHistory1, 30);  
  
  
  
  
  
  
  
  
  
//standard deviation of past orders received from each retailer
```

```
theSdev1 = getSdev(rcvOrderHistory1,30);

theSdev2 = getSdev(rcvOrderHistory2,30);

theSdev3 = getSdev(rcvOrderHistory3,30);

//standard deviation of past orders placed with the upper stream

//when info is not shared

theSdevS1 = getSdev(orderHistory1, 30);

theSdevS2 = getSdev(orderHistory2, 30);

theSdevS3 = getSdev(orderHistory3, 30);

//when shared info

//theSdevS1 = 30;

//theSdevS2 = 40;

//theSdevS3 = 20;

RandomHelper.setSeed(getTickCount());
```

```

// get the Normal distribution value: retailer 1

RandomHelper.createNormal(theMean1, theSdev1);

theErr1 = RandomHelper.getNormal().nextInt();

// get the Normal distribution value: retailer 2

RandomHelper.createNormal(theMean2, theSdev2);

theErr2 = RandomHelper.getNormal().nextInt();

// get the Normal distribution value: retailer 3

RandomHelper.createNormal(theMean3, theSdev3);

theErr3 = RandomHelper.getNormal().nextInt();

// the AR1 formula, the order quantity considers the change in
order-up-to levels

// get each of three parts for the calculation

```

```

// retailer 1

part1_1 = dValue + (int)Math.floor(Roo*YtActual1);

part2_1 = (int)((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr1;

// when info is not shared

part3_1 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* (YtPredict1 -
YtActual1)) /(1-Roo));

// when info is shared

//part3_1 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))*
RetailerAgent.getSharedErr1() /(1-Roo)));

// sum them up

currentOrder1 = Math.max(0,part1_1 + part2_1 - part3_1);

// retailer 2

part1_2 = dValue + (int)Math.floor(Roo*YtActual2);

```

```

part2_2 = (int) (((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr2);

// when info is not shared

part3_2 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* (YtPredict2 -
YtActual2)) /(1-Roo));

// when info is shared

//part3_2 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* 
RetailerAgent2.getSharedErr2()) /(1-Roo));

// sum them up

currentOrder2 = Math.max(0,part1_2 + part2_2 - part3_2);

// retailer 3

part1_3 = dValue + (int) Math.floor(Roo*YtActual3);

part2_3 = (int) (((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr3);

// when info is not shared

part3_3 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* (YtPredict3 -
YtActual3)) /(1-Roo));

```

```
// when info is shared

//part3_3 = (int)((Roo*((1-Math.pow(Roo, (Elle+1)))*
RetailerAgent3.getSharedErr3()) /(1-Roo)));

// sum them up

currentOrder3 = Math.max(0,part1_3 + part2_3 - part3_3);

orderLine1.addLast(currentOrder1);

orderLine2.addLast(currentOrder2);

orderLine3.addLast(currentOrder3);

orderHistory1.addLast(currentOrder1);

orderHistory2.addLast(currentOrder2);

orderHistory3.addLast(currentOrder3);

// the total order is the sum of individual orders
```

```

this.currentOrderS = currentOrder1 + currentOrder2 +
currentOrder3;

// backorders also need to be ordered

//this.currentOrderS += backOrder;

//this.currentOrderS = Math.max(0, currentOrderS+backOrder-
stock);

totalPredicted = currentOrderS;

// if stock is high, set order to zero

if(getStock()-getBackOrder() >= currentOrderS){

    currentOrderS = 0;

}

// if stock is low, do not forget the backorder

else if (getStock()-getBackOrder() < currentOrderS){
```

```
if (getStock() <= currentOrderS){  
    this.currentOrderS += getBackOrder();  
}  
  
}  
  
// add the orders to memory  
  
this.orderLine.addLast(currentOrderS);  
  
this.orderHistoryS.addLast(currentOrderS);  
  
// 4. calculate the total cost per unit shipped  
  
// in case total stock is zero  
  
//if (totalStock == 0){  
//    totalStock = 1;  
//}  
}
```

```
        this.totalCostAvg = (this.invCost * this.getStock() + this.shortCost  
        * this.getBackOrder() +  
  
        this.fixedOrderCost *  
        ((this.currentOrderS > 0)?1 : 0)+  
  
        this.purchaseCost * this.shipment);  
  
    }  
  
}
```

```
public static int getShipmentToR1(){  
  
    return suppliedHistory1.getLast();  
  
}  
  
public static int getShipmentToR2(){  
  
    return suppliedHistory3.getLast();  
  
}
```

```
public static int getShipmentToR3(){  
  
    return suppliedHistory3.getLast();
```

```
}
```

```
public int isSupplier(){
```

```
    return 1;
```

```
}
```

```
public double getTotalCost(){
```

```
    return this.totalCostAvg;
```

```
}
```

```
public int getTested(){
```

```
    return this.getBackOrder() - this.getStock();
```

```
}
```

```
public int getStock(){
```

```
    return stockHistory.getLast();
```

```
}
```

```
public int getBackOrder(){  
  
    return backHistory.getLast();  
  
}
```

```
public int getShipment(){  
  
    return this.shipment;  
  
}
```

```
public int getCurrentOrderS(){  
  
    return currentOrderS;  
  
}
```

```
public double getFulfilRate(){  
  
    return fulfilRate;
```

```
}

public int getCurrentDemand(){

    return currentDemandS;

}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

public int getTheErr1(){

    return theErr1;

}

public int getTheErr2(){
```

```
    return theErr2;
```

```
}
```

```
public int getTheErr3(){
```

```
    return theErr3;
```

```
}
```

```
public int getPart1_1(){
```

```
    return part1_1;
```

```
}
```

```
public int getPart2_1(){
```

```
    return part2_1;
```

```
}
```

```
public int getPart3_1(){
```

```
    return part3_1;
```

```
}
```

```
public int getPart1_2(){
```

```
    return part1_1;
```

```
}
```

```
public int getPart2_2(){
```

```
    return part2_1;
```

```
}
```

```
public int getPart3_2(){
```

```
    return part3_1;
```

```
}
```

```
public int getPart1_3(){
```

```
    return part1_1;
```

}

public int getPart2_3(){

 return part2_1;

}

public int getPart3_3(){

 return part3_1;

}

public int getCurrentOrder1(){

 return currentOrder1;

}

public int getCurrentOrder2(){

 return currentOrder2;

```
}

public int getCurrentOrder3(){

    return currentOrder3;

}

public int getRcvdorder1(){

    return rcvdorder1;

}

public int getRcvdorder2(){

    return rcvdorder2;

}

public int getRcvdorder3(){

    return rcvdorder3;

}

public double getTheSdev1(){

    return theSdev1;

}
```

```
public double getTheSdev2(){
```

```
    return theSdev2;
```

```
}
```

```
public double getTheSdev3(){
```

```
    return theSdev3;
```

```
}
```

```
// get the variance
```

```
public double getTheVar1(){
```

```
    return theSdev1*theSdev1;
```

```
}
```

```
public double getTheVar2(){
```

```
    return theSdev2*theSdev2;
```

```
}
```

```
public double getTheVar3(){
```

```
    return theSdev3*theSdev3;
```

```
    }

public double getMean(){

    return testValue;

}

public int getYtActual1(){

    return YtActual1;

}

public int getYtPredict1(){

    return YtPredict1;

}

public int getYtActual2(){

    return YtActual2;

}

public int getYtPredict2(){

    return YtPredict2;

}
```

```
}
```

```
public int getYtActual3(){
```

```
    return YtActual3;
```

```
}
```

```
public int getYtPredict3(){
```

```
    return YtPredict3;
```

```
}
```

```
public double getTheSdevS1(){
```

```
    return theSdevS1;
```

```
}
```

```
public double getTheSdevS2(){
```

```
    return theSdevS2;
```

```
}
```

```
public double getTheSdevS3(){
```

```
    return theSdevS3;
```

```
}

// get the variance

public double getTheVarS1(){

    return theSdevS1*theSdevS1;

}

public double getTheVarS2(){

    return theSdevS2*theSdevS2;

}

public double getTheVarS3(){

    return theSdevS3*theSdevS3;

}

public double getRoo(){

    return Roo;

}

public int getTotalRcvdOrder(){

    return totalRcvdOrder;
```

```
    }  
  
    public int getTotalPredicted(){  
  
        return totalPredicted;  
  
    }  
  
}
```

File 6: Moderator.java

/* The moderator is designed to handle the relationships between retailers.

* With the addition of this moderator, the process becomes three steps:

*

* 1. retailers get demand and calculate the stocks and backorders

* supplier receives the shipment from upper stream (with time lag)

* and orders from the retailers

* and prepares (predict) order to upper stream (with time lag)

* 2. moderator receives the stock and backorders info from retailers;

* moderator uses the relationship data to process the stock sharing

* moderator updates the stocks & backordres and send them back to retailers

* 3. retailers calculate (predict) the current demands plus the backorders

* and add them to queue as the orders to supplier (with time lag)

*

- * Each complete loop of order process includes three ticks, while each tick
 - * completes one step.

*/

```
package repast.simphony.demo.sc;

import repast.simphony.annotate.AgentAnnot;
import repast.simphony.engine.schedule.ScheduledMethod;

@AgentAnnot(displayName="Moderator")

public class Moderator extends SCagent{

    // constructor

    public Moderator(){

        //initiation here, if necessary
    }
}
```

}

// the generic variables for backorder, stock

// note the capital letters

protected int B1;

protected int B2;

protected int S1;

protected int S2;

protected int doShare = 0; //by default, two retailers do not share

protected int doShare12=0;

protected int doShare13=0;

protected int doShare23=0;

protected int bOrder1;

```
protected int bOrder2;

protected int bOrder3;

protected int stock1;

protected int stock2;

protected int stock3;

// the logic below can only handle small number of nodes.

// a more elegant algorithm may be developed in the later version

// to handle much larger (or any if possible) number of nodes

@ScheduledMethod(start=1, interval=1)

public void Moderating(){

    if (getTickCount() % 4 == 1){

        // do nothing

    }

}
```

```
else if (getTickCount() % 4 == 2){

    // Note this is the tick #2 in the period

    // reserved just for the moderator

    // get the customizable parameters

    // in fact the ratio parameters

    getInitParameters();

/* skip this moderator

Ratio12 = 0.0;

Ratio13 = 0.0;

Ratio23 = 0.0;

*/



// get the current stock for each retailer

stock1 = RetailerAgent.getStock();
```

```
stock2 = RetailerAgent2.getStock();

stock3 = RetailerAgent3.getStock();

// get the current backorder for each retailer

bOrder1 = RetailerAgent.getBackOrder();

bOrder2 = RetailerAgent2.getBackOrder();

bOrder3 = RetailerAgent3.getBackOrder();

// remove the last record of stock and backorder

// so that the updated could be added back later

// after the interaction processes that follow

RetailerAgent.stockHistory.removeLast();

RetailerAgent2.stockHistory.removeLast();

RetailerAgent3.stockHistory.removeLast();
```

```

RetailerAgent.backHistory.removeLast();

RetailerAgent2.backHistory.removeLast();

RetailerAgent3.backHistory.removeLast();

//first get the values of doShare

doShare12=getDoShare(Ratio12, bOrder1, bOrder2);

doShare13=getDoShare(Ratio13, bOrder1, bOrder3);

doShare23=getDoShare(Ratio23, bOrder2, bOrder3);

// if all the doShare values are zero,
(doShare12+doShare13+doShare23==0)

// no one shares, all the backorder and stock data pass through w/o
change;

// if (doShare12+doShare13+doShare23==3), this is not possible.

// there is a special case where all three retailers become one

// this should be handled separately as one supplier, one retailer,

```

```
// which is reduced to a simple case, and is basically covered in the  
  
// validation part of the essay (the one on one relationship)  
  
// this may be a case of joint venture or alliance.  
  
//  
  
// Another type of cases are also that only two are ALWAYS  
sharing,  
  
// and one is left out. These latter special cases may be taken cared  
of  
  
// by altering the value of the ratios at run time.  
  
//  
  
// all sharing schemes are independent, meaning, for example,  
  
// if R1 shares with both R2 and R3, R2 and R3 do not share by  
default.  
  
// if only two are sharing  
  
if(doShare12+doShare13+doShare23==1){
```

```
if(doShare12 == 1){

    // the other two values are zero

    // only R1 and R2 are sharing

    // process the sharing of stocks here

    sharedBy2(bOrder1, bOrder2, stock1, stock2);

    this.bOrder1 = B1;

    this.bOrder2 = B2;

    this.stock1 = S1;

    this.stock2 = S2;

    restoreInit();

}
```

```
else if(doShare13 == 1){

    // the other two values are zero

    // only R1 and R3 are sharing
```

```
// process the sharing of stocks here

sharedBy2(bOrder1, bOrder3, stock1, stock3);

this.bOrder1 = B1;

this.bOrder3 = B2;

this.stock1 = S1;

this.stock3 = S2;

restoreInit();

}

else if(doShare23 == 1){

    // the other two values are zero

    // only R2 and R3 are sharing

    // process the sharing of stocks here

    sharedBy2(bOrder2, bOrder3, stock2, stock3);

    this.bOrder2 = B1;

    this.bOrder3 = B2;
```

```

        this.stock2 = S1;

        this.stock3 = S2;

        restoreInit();

    }

}

else if(doShare12+doShare13+doShare23==2){

    // one node is sharing with two other nodes

    if(doShare12==0){    // R1 and R2 are not sharing

        // R1/R3 and R2/R3 are sharing, R3 is the center

        // R3 picks the one with higher ratio to share first,

        // if not enough, more from the 2nd choice

        if(Ratio13 > Ratio23){

            // R1/R3 first

            sharedBy2(bOrder1, bOrder3, stock1,
stock3);

```

```
        this.bOrder1 = B1;

        this.bOrder3 = B2;

        this.stock1 = S1;

        this.stock3 = S2;

        restoreInit();

// then R2/R3

// check if R2/R3 are still available for

sharing

if (getStillShare(bOrder2, bOrder3) == 1){

    sharedBy2(bOrder2, bOrder3, stock2,
stock3);

        this.bOrder2 = B1;

        this.bOrder3 = B2;

        this.stock2 = S1;

        this.stock3 = S2;
```

```
        }

    else {

        //do nothing

    }

restoreInit();

}

else if(Ratio13 <= Ratio23){

    // R2/R3 first

    sharedBy2(bOrder2, bOrder3, stock2,
stock3);

    this.bOrder2 = B1;

    this.bOrder3 = B2;

    this.stock2 = S1;

    this.stock3 = S2;

    restoreInit();
}
```

```
// then R1/R3

// check if R1/R3 are still available for
sharing

if (getStillShare(bOrder1, bOrder3) == 1){

    sharedBy2(bOrder1, bOrder3, stock1,
stock3);

    this.bOrder1 = B1;

    this.bOrder3 = B2;

    this.stock1 = S1;

    this.stock3 = S2;

}

else {

    //do nothing

}
```

```

        restoreInit();

    }

}

// same logic is applied to the other cases respectively

// may be replaced by some method, to be worked later

else if(doShare13==0){

    // R1/R2 and R2/R3 are sharing, R2 is the center

    // R2 picks the one with higher ratio to share first,

    // (if not enough, more from the 2nd choice - 2 b

implemented)

    // method call sharedValues(...)

    if(Ratio12 > Ratio23){

        // R1/R2 first

        sharedBy2(bOrder1, bOrder2, stock1,
stock2);

```

```
        this.bOrder1 = B1;

        this.bOrder2 = B2;

        this.stock1 = S1;

        this.stock2 = S2;

        restoreInit();

// then R2/R3

// check if R2/R3 are still available for

sharing

if (getStillShare(bOrder2, bOrder3) == 1){

    sharedBy2(bOrder2, bOrder3, stock2,
stock3);

        this.bOrder2 = B1;

        this.bOrder3 = B2;

        this.stock2 = S1;

        this.stock3 = S2;
```

```
    }

    else {

        //do nothing

    }

    restoreInit();

}

else if(Ratio13 <= Ratio23){

    // R2/R3 first

    sharedBy2(bOrder2, bOrder3, stock2,
stock3);

    this.bOrder2 = B1;

    this.bOrder3 = B2;

    this.stock2 = S1;

    this.stock3 = S2;

    restoreInit();

}
```

```
// then R1/R2

// check if R1/R2 are still available for
sharing

if (getStillShare(bOrder1, bOrder2) == 1){

    sharedBy2(bOrder1, bOrder2, stock1,
stock2);

    this.bOrder1 = B1;

    this.bOrder2 = B2;

    this.stock1 = S1;

    this.stock2 = S2;

}

else {

    //do nothing

}

restoreInit();
```

```

    }

}

else if(doShare23==0){

    // R1/R2 and R1/R3 are sharing, R1 is the center

    // R1 picks the one with higher ratio to share first,

    // (if not enough, more from the 2nd choice - 2 b

implemented)

    // method call sharedValues(...)

    if(Ratio12 > Ratio13){

        // R1/R2 first

        sharedBy2(bOrder1, bOrder2, stock1,
stock2);

        this.bOrder1 = B1;

        this.bOrder2 = B2;

        this.stock1 = S1;

        this.stock2 = S2;

```

```
restoreInit();

// then R1/R3

// check if R1/R3 are still available for
sharing

if (getStillShare(bOrder1, bOrder3) == 1){

    sharedBy2(bOrder1, bOrder3, stock1,
stock3);

    this.bOrder1 = B1;

    this.bOrder3 = B2;

    this.stock1 = S1;

    this.stock3 = S2;

}

else {

    //do nothing

}
```

```
        restoreInit();

    }

else if(Ratio13 <= Ratio23){

    // R1/R3 first

    sharedBy2(bOrder1, bOrder3, stock1,
stock3);

    this.bOrder1 = B1;

    this.bOrder3 = B2;

    this.stock1 = S1;

    this.stock3 = S2;

    restoreInit();

    // then R1/R2

    // check if R1/R2 are still available for
sharing
```

```
        if (getStillShare(bOrder1, bOrder2) == 1){  
  
            sharedBy2(bOrder1, bOrder2, stock1,  
stock2);  
  
            this.bOrder1 = B1;  
  
            this.bOrder2 = B2;  
  
            this.stock1 = S1;  
  
            this.stock2 = S2;  
  
        }  
  
        else {  
  
            //do nothing  
  
        }  
  
        restoreInit();  
  
    }  
  
}
```

```
// the above code will give the new stock and backorder
info

// the retailers update their stocks and backorders
accordingly

// using the static variables from here

// the supplier gets the adjusted order info from here

// and pass-through the Moderator the order to the retailers

}

else if(doShare12+doShare13+doShare23==3){

    // local variable

    int total;

    total = stock1+ stock2+stock3 -
(bOrder1+bOrder2+bOrder3);
```

// if total available stock after removing overstock is greater
than zero

// all backorders will be fulfilled this period

// and the rest of the total stock will be divided by 2

if (total >=0){

stock1 = total / 3;

stock2 = total / 3;

stock3 = total / 3;

bOrder1 = 0;

bOrder3 = 0;

bOrder3 = 0;

}

// if total available stock is less than overstock

// then both stocks will be set to zero and overstock be
splitted

else if (total <0){

bOrder1 = total / (-3);;

bOrder3 = total / (-3);;

bOrder3 = total / (-3);;

stock1 = 0;

stock2 = 0;

stock3 = 0;

}

}

// now we have the new stock and backorder data

```
// need to feed them back to the retailers

// for all other cases not mentioned or processed

// the original stock and backorder data is added back to the
memory

addUpdatedBack();

}

else if (getTickCount() % 4 == 3){

    // do nothing

}

else if (getTickCount() % 4 == 0){

    // do nothing

}

}
```

```

// get the value of doShare (only takes value 1 or 0)

// if both are sharing, then only one has positive backorder,
// otherwise they will not share,
// no sharing when both backorder is zero, and
// no sharing when both backorder is positive

public int getDoShare(double r, int b1, int b2){

    doShare = (int)Math.floor(Math.random() + r);

    if (doShare == 1){

        if ((b1==0 && b2==0)|| (b1>0 && b2>0)){

            // nothing to share

            doShare = 0;

        }

    }

    if (r == 1.0){

        doShare = 1;

    }

}

```

```
    return doShare;

}

// still sharing? Re-check it.

public int getStillShare(int b1, int b2){

    if ((b1==0 && b2==0)||(b1>0 && b2>0)){

        // nothing to share

        doShare = 0;

    }

    else {

        doShare = 1;

    }

    return doShare;

}
```

```
// method to calculate the backorders and stocks after sharing
```

```
public void sharedBy2(int b1, int b2, int s1, int s2){
```

```
    this.B1=b1;
```

```
    this.B2=b1;
```

```
    this.S1=s1;
```

```
    this.S2=s2;
```

```
    *****/
```

```
// if both are sharing, then only one has positive backorder
```

```
// and only one has positive stock (implied)
```

```
// otherwise they will not share
```

```
if (B1>0){ // meaning S1 == 0, S2>=0, B2 == 0
```

```
    if(B1>S2){
```

```
        B1-=S2;
```

S2=0;

}

else if(B1<=S2){

S2=B1;

B1=0;

}

}

else if(B2>0){ // meaning S2 == 0, S1>=0, B1 == 0

if(B2>S1){

B2=S1;

S1=0;

}

else if(B2<=S1){

S1=B2;

B2=0;

```
    }

}

/*this simpler algorithm simply split the stock or backorder between two*/

/*
// Local variable

int total = S1+S2-B1-B2;

// if total available stock after removing overstock is greater than zero

// all backorders will be fulfilled this period

// and the rest of the total stock will be divided by 2

if (total >= 0) {

    S1 = total / 2;

    S2 = total / 2;

    B1=0;
```

```
B2=0;

}

// if total available stock is less than overstock

// then both stocks will be set to zero and overstock be splitted

else if (total < 0){

    S1=0;

    S2=0;

    B1 = total / (-2);

    B2 = total / (-2);

}

*/



}

// restore the initial class variables

public void restoreInit(){
```

```

B1=0;

B2=0;

S1=0;

S2=0;

}

// add values of updated stock and backorder back to retailers' memory

public void addUpdatedBack(){

    RetailerAgent.stockHistory.addLast(stock1);

    RetailerAgent2.stockHistory.addLast(stock2);

    RetailerAgent3.stockHistory.addLast(stock3);

    RetailerAgent.backHistory.addLast(bOrder1);

    RetailerAgent2.backHistory.addLast(bOrder2);

    RetailerAgent3.backHistory.addLast(bOrder3);

}

```

```
public int getDoShare12(){  
    return doShare12;  
}  
  
public int getStock1(){  
    return stock1;  
}  
}
```

File 7: RetailerAgent.java

```
package repast.simphony.demo.sc;

//import java.util.ArrayList;

import java.util.LinkedList;

import repast.simphony.engine.environment.RunEnvironment;

import repast.simphony.engine.schedule.ScheduledMethod;

import repast.simphony.parameter.Parameters;

import repast.simphony.random.RandomHelper;

import repast.simphony.annotate.AgentAnnot;

/*************************************************************************/
```

/* The following is the working code for validation part of the dissertation.

* note: to reactivate this code, need to also remove the marks in the middle

* of the code which marks "continues..."

```
@AgentAnnot(displayName="Retailer Agent")
```

```
public class RetailerAgent extends SCagent {
```

```
    //the current demand generated
```

```
    protected int currentDemand;
```

```
    // previous demand for retailer
```

```
    protected int prevDemand;
```

```
    //shipment received by retailer agent
```

```
    protected int shipment;
```

```
//total stock (inventory) on hand

protected static int stock=0;

//total backordered quantity

protected static int backOrder=0;

protected static int sharedErr1;

// this is for calculating the average cost only

protected int totalStock=0;

// the storage for orders placed by the retailer agent

protected LinkedList<Integer> orderHistoryR1 = new LinkedList<Integer>();

// the storage for received shipment from the supplier agent

//protected static ArrayList shipRcvHistory = new ArrayList();

//Orders from down stream in the queue

protected LinkedList<Integer> orderLine = new LinkedList<Integer>();

// Orders to upper stream in the queue
```

```
protected static LinkedList<Integer> orderLineR1 = new LinkedList<Integer>();  
  
// The storage for keeping the history of demand  
  
protected LinkedList<Integer> demandStream=new LinkedList<Integer>();  
  
// the storage for keeping the history of total cost per unit  
  
protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();  
  
  
  
  
// The seed for generating demand  
  
//protected int theDemandSeed;  
  
  
  
  
protected int testValue;  
  
  
  
  
public RetailerAgent(){  
  
    //the constructor, initialize values  
  
    final Double Sdev=30.0;  
  
    // The seed for generating demand
```

```

final int theDemandSeed = 1;

RandomHelper.setSeed(theDemandSeed);

// get the Normal distribution values (zero mean, constant variance)

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

// get the customizable parameters

getInitParameters();

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

currentDemand = initialDemandAmount;

```

```

        demandStream.addLast(this.currentDemand);

        // get the initial retailer's order forecasts

        // and add them to order history memory

        orderLineR1.addLast(this.currentDemand);

        this.orderHistoryR1.addLast(this.currentDemand);

    }

    // calculate the order quantity in the following step()

    // It uses the AR1 process described in Lee, et al. (2000)

    // this hard working agent works from tick 1 with interval of 1

    // each 4 ticks are one period

    @ScheduledMethod(start=1, interval=1)

    public void calculateRetailer(){

        if (getTickCount() % 4 == 1){

```

```

// Calculate the demand (AR1 process) from customers, add the
demand to a container

// more demand forecasting functions to be added

theErr = getNormalValue();

sharedErr1 = theErr;

prevDemand = demandStream.getLast();

// $D(t) = d + R_{\text{oo}} * D(t-1) + \text{theErr}$ 

currentDemand = dValue + (int)(Roo*prevDemand) + theErr;

// add the demand to container

demandStream.addLast(currentDemand);

// now the retailer will do the job

// 1. check/update stock after received the shipment from supplier

this.shipment = SupplierAgent.suppliedHistory1.getLast();

stock += this.shipment;

```

```

totalStock = stock;

/*
 * the backorder handled differently
 */

// 2. fulfill the backorder; and receive customer order and update
inventory

stock = stock - backOrder;

// stock < 0? This case is rare, but needs to be taken care of

if (stock < 0){

    backOrder = (int)Math.abs(stock) + currentDemand;

    stock = 0;

}

else if (stock >= 0){

    stock = stock - currentDemand;

}

// again, stock < 0? This case is a little more likely

```

```
if (stock < 0){  
  
    backOrder = (int)Math.abs(stock);  
  
    stock = 0;  
  
}  
  
else if (stock >= 0){  
  
    backOrder = 0;  
  
}  
  
}  
  
*/  
  
/* continues...  
  
// stock level after sales to the customers  
  
stock = stock - currentDemand;  
  
// stock < 0? hopefully not, backorder means lost sales
```

```
if (stock < 0){  
  
    backOrder = (int)Math.abs(stock);  
  
    stock = 0;  
  
}  
  
else if (stock >= 0){  
  
    backOrder = 0;  
  
}  
  
}  
  
else if (getTickCount() % 4 == 2){  
  
    // do nothing  
  
}  
  
else if (getTickCount() % 4 == 3){  
  
    // 3. calculate order to send to supplier
```

```

// the AR1 formula

this.currentOrder = Math.max(0,
currentDemand+(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo,
(Elle+1))/(1-Roo)));

//this.currentOrder += backOrder;

//this.currentOrder = Math.max(0, currentOrder+backOrder-stock);

// if stock is high, set order to zero

if(stock-backOrder >= currentOrder){

    currentOrder = 0;

}

//if (stock<=0){

//    this.currentOrder += backOrder;

//}

orderLineR1.addLast(currentOrder);

```

```

this.orderHistoryR1.addLast(currentOrder);

//4. calculate the total cost per unit in stock

// in case total stock is zero

if (totalStock == 0){

    totalStock = 1;

}

this.totalCostAvg = (this.invCost * stock + this.shortCost *

backOrder +

        this.fixedOrderCost *

((this.currentOrder > 0)?1 : 0) +

        this.purchaseCost * this.shipment);

//this.testValue=(Integer)orderHistory.get(getTickCount());

```

```
}
```

```
else if (getTickCount() % 4 == 0){
```

```
// do nothing
```

```
}
```

```
}
```

```
public int getShipment(){
```

```
    return shipment;
```

```
}
```

```
public int getStock(){
```

```
    return stock;
```

```
}
```

```
public int getBackOrder(){
```

```
    return backOrder;
```

```
}
```

```
public LinkedList<Integer> getOrderHistory(){
```

```
    return orderHistoryR1;
```

```
}
```

```
public static LinkedList<Integer> getOrderLine(){
```

```
    return orderLineR1;
```

```
}
```

```
public static int getSharedErr1(){
```

```
    return sharedErr1;
```

```
}
```

```
public int isRetailer(){
```

```
    return 1;
```

```
}
```

```
public double getTotalCostAvg(){
```

```
    return this.totalCostAvg;
```

```
}
```

```
public int getPrevDemand(){
```

```
    return prevDemand;
```

```
}
```

```
public int getCurrentOrder(){
```

```
    return this.currentOrder;
```

```
}
```

```
public int getTheErr(){
```

```
        return theErr;

    }

public int getCurrentDemand(){

    return currentDemand;

}

// test the generated demand

// public int getTestValue(){

//     return testValue;

//}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

}
```

```
// get the uniform demand value

//public int getUniformValue(){

//    return (Integer)this.uniformStream.get(getTickCount());

//}

*/



/*********************/



/*
 * The following is the code for testing the interactions
 *
 */

*/
```

```
@AgentAnnot(displayName="Retailer Agent")

public class RetailerAgent extends SCagent {

    //the current demand generated
    protected int currentDemand;

    // previous demand for retailer
    protected int prevDemand;

    //shipment received by retailer agent
    protected int shipment;

    //total stock (inventory) on hand
    protected static int stock=0;

    // the memory that keeps the history of stock levels
    protected static LinkedList<Integer> stockHistory = new LinkedList<Integer>();

    //total backordered quantity
```

```
protected static int backOrder=0;

// the memory that keeps the history of backorder levels

protected static LinkedList<Integer> backHistory = new LinkedList<Integer>();

// The error term from the demand

protected int theErr;

protected static int sharedErr1;

// this is for calculating the average cost only

protected int totalStock=0;

// the storage for orders placed by the retailer agent

protected LinkedList<Integer> orderHistoryR1 = new LinkedList<Integer>();

// the storage for received shipment from the supplier agent

//protected static ArrayList shipRcvHistory = new ArrayList();

//Orders from down stream in the queue
```

```
protected LinkedList<Integer> orderLine = new LinkedList<Integer>();  
  
// Orders to upper stream in the queue  
  
protected static LinkedList<Integer> orderLineR1 = new LinkedList<Integer>();  
  
// The storage for keeping the history of demand  
  
protected LinkedList<Integer> demandStream=new LinkedList<Integer>();  
  
// the storage for keeping the history of total cost per unit  
  
protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();  
  
// The seed for generating demand  
  
//protected int theDemandSeed;  
  
protected int testValue;  
  
public RetailerAgent(){  
    //the constructor, initialize values
```

```

// get the customizable parameters

getInitParameters();

final Double Sdev=Sdev1;

// The seed for generating demand

final int theDemandSeed = 1;

RandomHelper.setSeed(theDemandSeed);

// get the Normal distribution values (zero mean, constant variance)

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

}

```

```
Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

currentDemand = initialDemandAmount;

demandStream.addLast(this.currentDemand);

// get the initial retailer's order forecasts

// and add them to order history memory

orderLineR1.addLast(this.currentDemand);

this.orderHistoryR1.addLast(this.currentDemand);

// inital stock

stockHistory.addLast(initialDemandAmount);

// initial backorder

backHistory.addLast(initialDemandAmount);

}
```

```

// calculate the order quantity in the following step()

// It uses the AR1 process described in Lee, et al. (2000)

// this hard working agent works from tick 1 with interval of 1

// each 4 ticks are one period

@ScheduledMethod(start=1, interval=1)

public void calculateRetailer(){

    if (getTickCount() % 4 == 1){

        // Calculate the demand (AR1 process) from customers, add the
        demand to a container

        // more demand forecasting functions to be added

        theErr = getNormalValue();

        sharedErr1 = theErr;
    }
}

```

```

// get the current d value

dValue = (int)(dValue*(1+growthRate)+0.9);

prevDemand = demandStream.getLast();

// $D(t) = d + R_{t-1} * D(t-1) + \text{theErr}$ 

currentDemand = dValue + (int)(Roo*prevDemand) + theErr;

// add the demand to container

demandStream.addLast(currentDemand);

// now the retailer will do the job

// 1. check/update stock after received the shipment from supplier

this.shipment = SupplierAgent.suppliedHistory1.getLast();

//shipment = 100;

// get the current stock

stock = getStock();

stock += this.shipment;

```

```
totalStock = stock;

//retrieve the back order

backOrder = getBackOrder();

// the backorder handled differently

// 2. fulfill the backorder; and receive customer order and update
inventory

// since the quantity delivered to customer does not really matter in
the calculation

// thus they are not kept in the memory in the implementation,
though they can be.

stock = stock - backOrder;

// stock < 0? This case is rare, but needs to be taken care of

if (stock < 0){

    backOrder = (int)Math.abs(stock) + currentDemand;
```

```
stock =0;
```

```
}
```

```
else if (stock>=0){
```

```
    stock = stock - currentDemand;
```

```
// again, stock < 0? This case is a little more likely
```

```
if (stock < 0){
```

```
    backOrder = (int)Math.abs(stock);
```

```
    stock = 0;
```

```
}
```

```
else if (stock >= 0){
```

```
    backOrder = 0;
```

```
}
```

```
}
```

```
/* the simplest way of handling backorders

// stock level after sales to the customers

stock = stock - currentDemand;

// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}

else if (stock >= 0){

    backOrder = 0;

}

*/

// store the stock and backorder data to memory

stockHistory.addLast(stock);
```

```
    backHistory.addLast(backOrder);

    // the backorders is back to initial state

    backOrder = 0;

    // the stock is back to initial state

    stock = 0;

}

else if (getTickCount() % 4 == 2){

    // do nothing

}

else if (getTickCount() % 4 == 3){

    // 3. calculate order to send to supplier
```

```

// the AR1 formula

this.currentOrder = Math.max(0, currentDemand +
(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo, (Elle+1)))/(1-Roo)));

currentDemand = 0;

prevDemand = 0;

//this.currentOrder += backOrder;

//this.currentOrder = Math.max(0, currentOrder+backOrder-stock);

// if stock is high, set order to zero

if(getStock()-getBackOrder() >= currentOrder){

    currentOrder = 0;

}

// if stock is low, do not forget the backorder

else if (getStock()-getBackOrder() < currentOrder){

    if (getStock() <= currentOrder){


```

```
        this.currentOrder += getBackOrder();

    }

}

orderLineR1.addLast(currentOrder);

this.orderHistoryR1.addLast(currentOrder);

//4. calculate the total cost per unit in stock

// in case total stock is zero

if (totalStock == 0){

    totalStock = 1;

}

this.totalCostAvg = (this.invCost * getStock() + this.shortCost *

getBackOrder() +
```

```
        this.fixedOrderCost *  
((this.currentOrder > 0)?1 : 0)+  
  
        this.purchaseCost * this.shipment);  
  
//this.testValue=(Integer)orderHistory.get(getTickCount());  
  
}  
  
else if (getTickCount() % 4 == 0){  
  
    // do nothing  
  
}  
  
}  
  
public int getShipment(){  
  
    return shipment;  
  
}
```

```
public static int getStock(){

    return stockHistory.getLast();

}

public static int getBackOrder(){

    return backHistory.getLast();

}

public LinkedList<Integer> getOrderHistory(){

    return orderHistoryR1;

}

public static LinkedList<Integer> getOrderLine(){

    return orderLineR1;

}
```

```
public static int getSharedErr1(){
```

```
    return sharedErr1;
```

```
}
```

```
public int isRetailer(){
```

```
    return 1;
```

```
}
```

```
public double getTotalCostAvg(){
```

```
    return this.totalCostAvg;
```

```
}
```

```
public int getPrevDemand(){
```

```
    return prevDemand;
```

```
}
```

```
public int getCurrentOrder(){

    return this.currentOrder;

}
```

```
public int getTheErr(){

    return theErr;

}
```

```
public int getCurrentDemand(){

    return currentDemand;

}
```

```
// test the generated demand

// public int getTestValue(){

//     return testValue;
```

```

//}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

// get the uniform demand value

//public int getUniformValue(){

//    return (Integer)this.uniformStream.get(getTickCount());

//}

}


```

File 8: RetailerAgent2.java

```
package repast.simphony.demo.sc;
```

```
//import java.util.ArrayList;  
  
import java.util.LinkedList;  
  
  
  
import repast.simphony.engine.environment.RunEnvironment;  
  
import repast.simphony.engine.schedule.ScheduledMethod;  
  
import repast.simphony.parameter.Parameters;  
  
import repast.simphony.random.RandomHelper;  
  
import repast.simphony.annotate.AgentAnnot;  
  
*****
```

/* The following is the working code for validation part of the dissertation.

* note: to reactivate this code, need to also remove the marks in the middle

* of the code which marks "continues..."

```
@AgentAnnot(displayName="Retailer Agent")
```

```
public class RetailerAgent2 extends SCagent {
```

```
    //the current demand generated
```

```
    protected int currentDemand;
```

```
    // previous demand for retailer
```

```
    protected int prevDemand;
```

```
    //shipment received by retailer agent
```

```
    protected int shipment;
```

```
    //total stock (inventory) on hand
```

```
    protected static int stock=0;
```

```
    //total backordered quantity
```

```
    protected static int backOrder=0;
```

```
protected static int sharedErr2;

// this is for calculating the average cost only

protected int totalStock=0;

// the storage for orders placed by the retailer agent

protected LinkedList<Integer> orderHistoryR2 = new LinkedList<Integer>();

// the storage for received shipment from the supplier agent

//protected static ArrayList shipRcvHistory = new ArrayList();

//Orders from down stream in the queue

protected LinkedList<Integer> orderLine = new LinkedList<Integer>();

// Orders to upper stream in the queue

protected static LinkedList<Integer> orderLineR2 = new LinkedList<Integer>();

// The storage for keeping the history of demand

protected LinkedList<Integer> demandStream=new LinkedList<Integer>();
```

```
// the storage for keeping the history of total cost per unit

protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();

// The seed for generating demand

//protected int theDemandSeed;

protected int testValue;

public RetailerAgent2(){

    //the constructor, initialize values

    final Double Sdev=40.0;

    // The seed for generating demand

    final int theDemandSeed = 2;

    RandomHelper.setSeed(theDemandSeed);

    // get the Normal distribution values (zero mean, constant variance)
```

```

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

// get the customizable parameters

getInitParameters();

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

currentDemand = initialDemandAmount;

demandStream.addLast(this.currentDemand);

// get the initial retailer's order forecasts

// and add them to order history memory

```

```

        orderLineR2.addLast(this.currentDemand);

        this.orderHistoryR2.addLast(this.currentDemand);

    }

// calculate the order quantity in the following step()

// It uses the AR1 process described in Lee, et al. (2000)

// this hard working agent works from tick 1 with interval of 1

// each 4 ticks are one period

@ScheduledMethod(start=1, interval=1)

public void calculateRetailer(){

    if (getTickCount() % 4 == 1){

        // Calculate the demand (AR1 process) from customers, add the
        demand to a container

        // more demand forecasting functions to be added
    }
}

```

```

theErr = getNormalValue();

sharedErr2 = theErr;

prevDemand = demandStream.getLast();

//D(t) = d + Roo *D(t-1) + theErr

currentDemand = dValue + (int)(Roo*prevDemand) + theErr;

// add the demand to container

demandStream.addLast(currentDemand);

// now the retailer will do the job

// 1. check/update stock after received the shipment from supplier

this.shipment = SupplierAgent.suppliedHistory2.getLast();

stock += this.shipment;

totalStock = stock;

/* the backorder handled differently

```



```
    }

    else if (stock >= 0){

        backOrder = 0;

    }

}

*/
/* continues...

// stock level after sales to the customers

stock = stock - currentDemand;

// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;
```

```

    }

else if (stock >= 0){

    backOrder = 0;

}

else if (getTickCount() % 4 == 2){

    // do nothing

}

else if (getTickCount() % 4 == 3){

    // 3. calculate order to send to supplier

    // the AR1 formula

    this.currentOrder = Math.max(0,
        currentDemand+(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo,
            (Elle+1))/(1-Roo)));
}

```

```
//this.currentOrder += backOrder;

//this.currentOrder = Math.max(0, currentOrder+backOrder-stock);

// if stock is high, set order to zero

if(stock-backOrder >= currentOrder){

    currentOrder = 0;

}

//if (stock<=0){

//    this.currentOrder += backOrder;

//}

orderLineR2.addLast(currentOrder);

this.orderHistoryR2.addLast(currentOrder);

//4. calculate the total cost per unit in stock
```

```

// in case total stock is zero

if (totalStock == 0){

    totalStock = 1;

}

this.totalCostAvg = (this.invCost * stock + this.shortCost *

backOrder +

        this.fixedOrderCost *

((this.currentOrder > 0)?1 : 0) +

        this.purchaseCost * this.shipment);

//this.testValue=(Integer)orderHistory.get(getTickCount());

}

else if (getTickCount() % 4 == 0){

    // do nothing
}

```

```
}
```

```
}
```

```
public int getShipment(){
```

```
    return shipment;
```

```
}
```

```
public int getStock(){
```

```
    return stock;
```

```
}
```

```
public int getBackOrder(){
```

```
    return backOrder;
```

```
}
```

```
public LinkedList<Integer> getOrderHistory(){
```

```
        return orderHistoryR2;

    }

public static LinkedList<Integer> getOrderLine(){

    return orderLineR2;

}

public static int getSharedErr2(){

    return sharedErr2;

}

public int isRetailer(){

    return 1;

}

public double getTotalCostAvg(){
```

```
    return this.totalCostAvg;  
  
}
```

```
public int getPrevDemand(){  
  
    return prevDemand;  
  
}
```

```
public int getCurrentOrder(){  
  
    return this.currentOrder;  
  
}
```

```
public int getTheErr(){  
  
    return theErr;  
  
}
```

```
public int getCurrentDemand(){

    return currentDemand;

}

// test the generated demand

// public int getTestValue(){

//     return testValue;

//}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

// get the uniform demand value

//public int getUniformValue(){
```

```
//      return (Integer)this.uniformStream.get(getTickCount());  
  
//}  
  
}  
  
*/
```

```
*****
```

```
/*
```

```
* The following is the code for testing the interactions
```

```
*
```

```
*/
```

```
@AgentAnnot(displayName="Retailer Agent")

public class RetailerAgent2 extends SCagent {

    //the current demand generated

    protected int currentDemand;

    // previous demand for retailer

    protected int prevDemand;

    //shipment received by retailer agent

    protected int shipment;

    //total stock (inventory) on hand

    protected static int stock=0;

    // the memory that keeps the history of stock levels

    protected static LinkedList<Integer> stockHistory = new LinkedList<Integer>();

    //total backordered quantity

    protected static int backOrder=0;
```

```
// the memory that keeps the history of backorder levels

protected static LinkedList<Integer> backHistory = new LinkedList<Integer>();

// The error term from the demand

protected int theErr;

protected static int sharedErr2;

// this is for calculating the average cost only

protected int totalStock=0;

// the storage for orders placed by the retailer agent

protected LinkedList<Integer> orderHistoryR2 = new LinkedList<Integer>();

// the storage for received shipment from the supplier agent

//protected static ArrayList shipRcvHistory = new ArrayList();

//Orders from down stream in the queue

protected LinkedList<Integer> orderLine = new LinkedList<Integer>();
```

```
// Orders to upper stream in the queue

protected static LinkedList<Integer> orderLineR2 = new LinkedList<Integer>();

// The storage for keeping the history of demand

protected LinkedList<Integer> demandStream=new LinkedList<Integer>();

// the storage for keeping the history of total cost per unit

protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();

// The seed for generating demand

//protected int theDemandSeed;

protected int testValue;

public RetailerAgent2(){

    //the constructor, initialize values
```

```
// get the customizable parameters

getInitParameters();

final Double Sdev=Sdev2;

// The seed for generating demand

final int theDemandSeed = 2;

RandomHelper.setSeed(theDemandSeed);

// get the Normal distribution values (zero mean, constant variance)

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

Parameters p = RunEnvironment.getInstance().getParameters();
```

```
this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

currentDemand = initialDemandAmount;

demandStream.addLast(this.currentDemand);

// get the initial retailer's order forecasts

// and add them to order history memory

orderLineR2.addLast(this.currentDemand);

this.orderHistoryR2.addLast(this.currentDemand);

// inital stock

stockHistory.addLast(initialDemandAmount);

// initial backorder

backHistory.addLast(initialDemandAmount);

}
```

```
// calculate the order quantity in the following step()

// It uses the AR1 process described in Lee, et al. (2000)

// this hard working agent works from tick 1 with interval of 1

// each 4 ticks are one period

@scheduledMethod(start=1, interval=1)

public void calculateRetailer(){

    if (getTickCount() % 4 == 1){

        // Calculate the demand (AR1 process) from customers, add the
        demand to a container

        // more demand forecasting functions to be added

        theErr = getNormalValue();

        sharedErr2 = theErr;

        // get the current d value
    }
}
```

```

dValue = (int)(dValue*(1+growthRate)+0.9);

prevDemand = demandStream.getLast();

//D(t) = d + Roo *D(t-1) + theErr

currentDemand = dValue + (int)(Roo*prevDemand) + theErr;

// add the demand to container

demandStream.addLast(currentDemand);

// now the retailer will do the job

// 1. check/update stock after received the shipment from supplier

this.shipment = SupplierAgent.suppliedHistory2.getLast();

//shipment = 100;

// get the current stock

stock = getStock();

stock += this.shipment;

totalStock = stock;

```

```
//retrieve the back order  
  
backOrder = getBackOrder();  
  
  
  
// the backorder handled differently  
  
// 2. fulfill the backorder; and receive customer order and update  
inventory  
  
  
// since the quantity delivered to customer does not really matter in  
the calculation  
  
  
// thus they are not kept in the memory in the implementation,  
though they can be.  
  
  
  
  
stock = stock - backOrder;  
  
  
// stock < 0? This case is rare, but needs to be taken care of  
  
if (stock < 0){  
  
    backOrder = (int)Math.abs(stock) + currentDemand;  
  
    stock =0;  
}
```

}

else if (stock>=0){

 stock = stock - currentDemand;

// again, stock < 0? This case is a little more likely

 if (stock < 0){

 backOrder = (int)Math.abs(stock);

 stock = 0;

}

 else if (stock >= 0){

 backOrder = 0;

}

}

```
/* the simplest way of handling backorders

// stock level after sales to the customers

stock = stock - currentDemand;

// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}

else if (stock >= 0){

    backOrder = 0;

}

*/

// store the stock and backorder data to memory
```

```
stockHistory.addLast(stock);

backHistory.addLast(backOrder);

// the backorders is back to initial state

backOrder = 0;

// the stock is back to initial state

stock = 0;

}

else if (getTickCount() % 4 == 2){

// do nothing

}

else if (getTickCount() % 4 == 3){
```

```

// 3. calculate order to send to supplier

// the AR1 formula

this.currentOrder = Math.max(0,
currentDemand+(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo,
(Elle+1))/(1-Roo)));

currentDemand = 0;

prevDemand = 0;

//this.currentOrder += backOrder;

//this.currentOrder = Math.max(0, currentOrder+backOrder-stock);

// if stock is high, set order to zero

if(getStock()-getBackOrder() >= currentOrder){

    currentOrder = 0;

}

// if stock is low, do not forget the backorder

```

```
else if (getStock()-getBackOrder() < currentOrder){
```

```
    if (getStock() <= currentOrder){
```

```
        this.currentOrder += getBackOrder();
```

```
}
```

```
}
```

```
orderLineR2.addLast(currentOrder);
```

```
this.orderHistoryR2.addLast(currentOrder);
```

```
//4. calculate the total cost per unit in stock
```

```
// in case total stock is zero
```

```
if (totalStock == 0){
```

```
    totalStock = 1;
```

```
}
```

```
        this.totalCostAvg = (this.invCost * getStock() + this.shortCost *  
getBackOrder() +  
  
        this.fixedOrderCost * ((this.currentOrder > 0)?1 :  
0)+  
  
        this.purchaseCost * this.shipment);  
  
//this.testValue=(Integer)orderHistory.get(getTickCount());  
}  
  
else if (getTickCount() % 4 == 0){  
  
    // do nothing  
}  
  
}  
  
public int getShipment(){  
  
    return shipment;  
}
```

```
public static int getStock(){

    return stockHistory.getLast();

}

public static int getBackOrder(){

    return backHistory.getLast();

}

public LinkedList<Integer> getOrderHistory(){

    return orderHistoryR2;

}

public static LinkedList<Integer> getOrderLine(){

    return orderLineR2;

}
```

```
public static int getSharedErr2(){
```

```
    return sharedErr2;
```

```
}
```

```
public int isRetailer(){
```

```
    return 1;
```

```
}
```

```
public double getTotalCostAvg(){
```

```
    return this.totalCostAvg;
```

```
}
```

```
public int getPrevDemand(){
```

```
    return prevDemand;
```

```
}
```

```
public int getCurrentOrder(){  
  
    return this.currentOrder;  
  
}
```

```
public int getTheErr(){  
  
    return theErr;  
  
}
```

```
public int getCurrentDemand(){  
  
    return currentDemand;  
  
}
```

```
// test the generated demand
```

```
// public int getTestValue(){
```


File 9:

```
package repast.simphony.demo.sc;

//import java.util.ArrayList;

import java.util.LinkedList;

import repast.simphony.engine.environment.RunEnvironment;

import repast.simphony.engine.schedule.ScheduledMethod;

import repast.simphony.parameter.Parameters;

import repast.simphony.random.RandomHelper;

import repast.simphony.annotate.AgentAnnot;
```

```
*****
```

```
/* The following is the working code for validation part of the dissertation.
```

```
* note: to reactivate this code, need to also remove the marks in the middle
```

```
* of the code which marks "continues..."
```

```
@AgentAnnot(displayName="Retailer Agent")
```

```
public class RetailerAgent3 extends SCagent {
```

```
    //the current demand generated
```

```
    protected int currentDemand;
```

```
    // previous demand for retailer
```

```
    protected int prevDemand;
```

```
    //shipment received by retailer agent
```

```
    protected int shipment;
```

```
//total stock (inventory) on hand
```

```
protected static int stock=0;
```

```
//total backordered quantity
```

```
protected static int backOrder=0;
```

```
protected static int sharedErr3;
```

```
// this is for calculating the average cost only
```

```
protected int totalStock=0;
```

```
// the storage for orders placed by the retailer agent
```

```
protected LinkedList<Integer> orderHistoryR3 = new LinkedList<Integer>();
```

```
// the storage for received shipment from the supplier agent
```

```
//protected static ArrayList shipRcvHistory = new ArrayList();
```

```
//Orders from down stream in the queue
```

```
protected LinkedList<Integer> orderLine = new LinkedList<Integer>();
```

```
// Orders to upper stream in the queue

protected static LinkedList<Integer> orderLineR3 = new LinkedList<Integer>();

// The storage for keeping the history of demand

protected LinkedList<Integer> demandStream=new LinkedList<Integer>();

// the storage for keeping the history of total cost per unit

protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();

// The seed for generating demand

//protected int theDemandSeed;

protected int testValue;

public RetailerAgent3(){

    //the constructor, initialize values

    final Double Sdev=20.0;
```

```

// The seed for generating demand

final int theDemandSeed = 3;

RandomHelper.setSeed(theDemandSeed);

// get the Normal distribution values (zero mean, constant variance)

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

// get the customizable parameters

getInitParameters();

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

```

```

        currentDemand = initialDemandAmount;

        demandStream.addLast(this.currentDemand);

        // get the initial retailer's order forecasts

        // and add them to order history memory

        orderLineR3.addLast(this.currentDemand);

        this.orderHistoryR3.addLast(this.currentDemand);

    }

    // calculate the order quantity in the following step()

    // It uses the AR1 process described in Lee, et al. (2000)

    // this hard working agent works from tick 1 with interval of 1

    // each 4 ticks are one period

    @ScheduledMethod(start=1, interval=1)

    public void calculateRetailer(){

        if (getTickCount() % 4 == 1){

```

```
// Calculate the demand (AR1 process) from customers, add the  
demand to a container  
  
// more demand forecasting functions to be added  
  
theErr = getNormalValue();  
  
sharedErr3 = theErr;  
  
prevDemand = demandStream.getLast();  
  
//D(t) = d + Roo *D(t-1) + theErr  
  
currentDemand = dValue + (int)(Roo*prevDemand) + theErr;  
  
// add the demand to container  
  
demandStream.addLast(currentDemand);  
  
// now the retailer will do the job  
  
// 1. check/update stock after received the shipment from supplier
```

```
this.shipment = SupplierAgent.suppliedHistory3.getLast();

stock += this.shipment;

totalStock = stock;

/* the backorder handled differently

// 2. fulfill the backorder; and receive customer order and update
inventory

stock = stock - backOrder;

// stock < 0? This case is rare, but needs to be taken care of

if (stock < 0){

    backOrder = (int)Math.abs(stock) + currentDemand;

    stock =0;

}

else if (stock>=0){

    stock = stock - currentDemand;

}
```

```
// again, stock < 0? This case is a little more likely

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}

else if (stock >= 0){

    backOrder = 0;

}

/* continues...

// stock level after sales to the customers

stock = stock - currentDemand;
```

```
// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}

else if (stock >= 0){

    backOrder = 0;

}

else if (getTickCount() % 4 == 2){

    // do nothing

}

else if (getTickCount() % 4 == 3){
```

```

// 3. calculate order to send to supplier

// the AR1 formula

this.currentOrder = Math.max(0,
currentDemand+(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo,
(Elle+1))/(1-Roo))));

//this.currentOrder += backOrder;

//this.currentOrder = Math.max(0, currentOrder+backOrder-stock);

// if stock is high, set order to zero

if(stock-backOrder >= currentOrder){

    currentOrder = 0;

}

//if (stock<=0){

//    this.currentOrder += backOrder;

//}


```

```

orderLineR3.addLast(currentOrder);

this.orderHistoryR3.addLast(currentOrder);

//4. calculate the total cost per unit in stock

// in case total stock is zero

if (totalStock == 0){

    totalStock = 1;

}

this.totalCostAvg = (this.invCost * stock + this.shortCost *

backOrder +

        this.fixedOrderCost *

((this.currentOrder > 0)?1 : 0) +

        this.purchaseCost * this.shipment);

```

```
//this.testValue=(Integer)orderHistory.get(getTickCount());  
  
}  
  
else if (getTickCount() % 4 == 0){  
  
    // do nothing  
  
}  
  
}  
  
public int getShipment(){  
  
    return shipment;  
  
}  
  
public int getStock(){  
  
    return stock;  
  
}
```

```
public int getBackOrder(){

    return backOrder;

}

public LinkedList<Integer> getOrderHistory(){

    return orderHistoryR3;

}

public static LinkedList<Integer> getOrderLine(){

    return orderLineR3;

}

public static int getSharedErr3(){

    return sharedErr3;

}
```

```
public int isRetailer(){

    return 1;

}

public double getTotalCostAvg(){

    return this.totalCostAvg;

}

public int getPrevDemand(){

    return prevDemand;

}

public int getCurrentOrder(){

    return this.currentOrder;

}
```

```
public int getTheErr(){

    return theErr;

}

public int getCurrentDemand(){

    return currentDemand;

}

// test the generated demand

// public int getTestValue(){

//     return testValue;

//}

// get the normal demand value

public int getNormalValue(){
```

```
        return (Integer)this.normalStream.get(getTickCount());  
  
    }  
  
    // get the uniform demand value  
  
    //public int getUniformValue(){  
  
        //      return (Integer)this.uniformStream.get(getTickCount());  
  
        //}  
  
    }  
  
*/  
  
/******************************************/
```

```
/*
```

* The following is the code for testing the interactions

*

*/

```
@AgentAnnot(displayName="Retailer Agent")
```

```
public class RetailerAgent3 extends SCagent {
```

```
    //the current demand generated
```

```
    protected int currentDemand;
```

```
    // previous demand for retailer
```

```
    protected int prevDemand;
```

```
    //shipment received by retailer agent
```

```
    protected int shipment;
```

```
//total stock (inventory) on hand

protected static int stock=0;

// the memory that keeps the history of stock levels

protected static LinkedList<Integer> stockHistory = new LinkedList<Integer>();

//total backordered quantity

protected static int backOrder=0;

// the memory that keeps the history of backorder levels

protected static LinkedList<Integer> backHistory = new LinkedList<Integer>();

// The error term from the demand

protected int theErr;

protected static int sharedErr3;

// this is for calculating the average cost only

protected int totalStock=0;
```

```
// the storage for orders placed by the retailer agent

protected LinkedList<Integer> orderHistoryR3 = new LinkedList<Integer>();

// the storage for received shipment from the supplier agent

//protected static ArrayList shipRcvHistory = new ArrayList();

//Orders from down stream in the queue

protected LinkedList<Integer> orderLine = new LinkedList<Integer>();

// Orders to upper stream in the queue

protected static LinkedList<Integer> orderLineR3 = new LinkedList<Integer>();

// The storage for keeping the history of demand

protected LinkedList<Integer> demandStream=new LinkedList<Integer>();

// the storage for keeping the history of total cost per unit

protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();

// The seed for generating demand

//protected int theDemandSeed;
```

```
protected int testValue;

public RetailerAgent3(){
    //the constructor, initialize values

    // get the customizable parameters
    getInitParameters();

    final Double Sdev=Sdev3;

    // The seed for generating demand
    final int theDemandSeed = 3;

    RandomHelper.setSeed(theDemandSeed);

    // get the Normal distribution values (zero mean, constant variance)
    RandomHelper.createNormal(0.0, Sdev);

    //RandomHelper.createUniform(0, 25);
```

```

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

currentDemand = initialDemandAmount;

demandStream.addLast(this.currentDemand);

// get the initial retailer's order forecasts

// and add them to order history memory

orderLineR3.addLast(this.currentDemand);

this.orderHistoryR3.addLast(this.currentDemand);

// init stock

```

```

    stockHistory.addLast(initialDemandAmount);

    // initial backorder

    backHistory.addLast(initialDemandAmount);

}

// calculate the order quantity in the following step()

// It uses the AR1 process described in Lee, et al. (2000)

// this hard working agent works from tick 1 with interval of 1

// each 4 ticks are one period

@ScheduledMethod(start=1, interval=1)

public void calculateRetailer(){

    if (getTickCount() % 4 == 1){

        // Calculate the demand (AR1 process) from customers, add the
        demand to a container
    }
}

```

```
// more demand forecasting functions to be added
```

```
theErr = getNormalValue();
```

```
sharedErr3 = theErr;
```

```
// get the current d value
```

```
dValue = (int)(dValue*(1+growthRate)+0.9);
```

```
prevDemand = demandStream.getLast();
```

```
// $D(t) = d + R_{t-1} * D(t-1) + \text{theErr}$ 
```

```
currentDemand = dValue + (int)(Roo*prevDemand) + theErr;
```

```
// add the demand to container
```

```
demandStream.addLast(currentDemand);
```

```
// now the retailer will do the job
```

```
// 1. check/update stock after received the shipment from supplier

this.shipment = SupplierAgent.suppliedHistory3.getLast();

//shipment = 100;

// get the current stock

stock = getStock();

stock += this.shipment;

totalStock = stock;

//retrieve the back order

backOrder = getBackOrder();

// the backorder handled differently

// 2. fulfill the backorder; and receive customer order and update

inventory

// since the quantity delivered to customer does not really matter in

the calculation
```

// thus they are not kept in the memory in the implementation,
though they can be.

```
stock = stock - backOrder;
```

```
// stock < 0? This case is rare, but needs to be taken care of
```

```
if (stock < 0){
```

```
    backOrder = (int)Math.abs(stock) + currentDemand;
```

```
    stock = 0;
```

```
}
```

```
else if (stock >= 0){
```

```
    stock = stock - currentDemand;
```

```
// again, stock < 0? This case is a little more likely
```

```
if (stock < 0){
```

```
    backOrder = (int)Math.abs(stock);
```

```
    stock = 0;
```

```
        }

    else if (stock >= 0){

        backOrder = 0;

    }

/* the simplest way of handling backorders

// stock level after sales to the customers

stock = stock - currentDemand;

// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}
```

```
else if (stock >= 0){  
  
    backOrder = 0;  
  
}  
  
*/  
  
  
  
// store the stock and backorder data to memory  
  
stockHistory.addLast(stock);  
  
backHistory.addLast(backOrder);  
  
  
  
// the backorders is back to initial state  
  
backOrder = 0;  
  
  
  
  
// the stock is back to initial state  
  
stock = 0;  
  
}
```

```

else if (getTickCount() % 4 == 2){

    // do nothing

}

else if (getTickCount() % 4 == 3){

    // 3. calculate order to send to supplier

    // the AR1 formula

    this.currentOrder = Math.max(0,
        currentDemand+(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo,
            (Elle+1))/(1-Roo))));

    currentDemand = 0;

    prevDemand = 0;

    //this.currentOrder += backOrder;

    //this.currentOrder = Math.max(0, currentOrder+backOrder-stock);
}

```

```
// if stock is high, set order to zero

if(getStock()-getBackOrder() >= currentOrder){

    currentOrder = 0;

}

// if stock is low, do not forget the backorder

else if (getStock()-getBackOrder() < currentOrder){

    if (getStock() <= currentOrder){

        this.currentOrder += getBackOrder();

    }

}

orderLineR3.addLast(currentOrder);

this.orderHistoryR3.addLast(currentOrder);

//4. calculate the total cost per unit in stock
```

```

// in case total stock is zero

if (totalStock == 0){

    totalStock = 1;

}

this.totalCostAvg = (this.invCost * getStock() + this.shortCost *

getBackOrder() + 

        this.fixedOrderCost * ((this.currentOrder > 0)?1 : 

0)+

        this.purchaseCost * this.shipment);

//this.testValue=(Integer)orderHistory.get(getTickCount());

}

else if (getTickCount() % 4 == 0){

    // do nothing
}

```

```
}
```

```
}
```

```
public int getShipment(){
```

```
    return shipment;
```

```
}
```

```
public static int getStock(){
```

```
    return stockHistory.getLast();
```

```
}
```

```
public static int getBackOrder(){
```

```
    return backHistory.getLast();
```

```
}
```

```
public LinkedList<Integer> getOrderHistory(){
```

```
        return orderHistoryR3;

    }

public static LinkedList<Integer> getOrderLine(){

    return orderLineR3;

}

public static int getSharedErr3(){

    return sharedErr3;

}

public int isRetailer(){

    return 1;

}

public double getTotalCostAvg(){
```

```
    return this.totalCostAvg;  
  
}
```

```
public int getPrevDemand(){  
  
    return prevDemand;  
  
}
```

```
public int getCurrentOrder(){  
  
    return this.currentOrder;  
  
}
```

```
public int getTheErr(){  
  
    return theErr;  
  
}
```

```
public int getCurrentDemand(){

    return currentDemand;

}

// test the generated demand

// public int getTestValue(){

//     return testValue;

//}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

// get the uniform demand value

//public int getUniformValue(){
```

```
//      return (Integer)this.uniformStream.get(getTickCount());  
  
//}  
  
}
```

File 10: repast.simphony.dataLoader.engine.ClassNameDataLoaderAction_0.xml

```
<string>repast.simphony.demo.sc.ContextCreator</string>
```

AGENT BASED MODELING FOR
SUPPLY CHAIN MANAGEMENT:
EXAMINING THE IMPACT OF INFORMATION SHARING

A dissertation submitted to:
Kent State University Graduate School of Management
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

by

Xiaozhou (David) Zhu

December, 2008

Dissertation written by

Xiaozhou (David) Zhu

B.S., Shanghai Institute of Mechanical Engineering, 1989

M.B.A., Kent State University, 1999

M.A. Computer Science, Kent State University, 2002

Ph.D., Kent State University, 2008

Approved by:

Chair, Doctoral Dissertation Committee

Member, Doctoral Dissertation Committee

Member, Doctoral Dissertation Committee

Member, Doctoral Dissertation Committee

Accepted by:

Doctoral Director, Graduate School of
Management

Dean, Graduate School of Management

Acknowledgements

Although only my name appears on the cover of this dissertation, a large number of important people in my life have contributed to its production and completion, to whom I own my great gratitude, and because of whom my graduate experience has become cherishable forever.

I would like to gratefully and sincerely thank Dr. Marvin Troutt for his guidance, understanding, patience, and most importantly, his friendship during my graduate studies at Kent State University. I have been amazingly fortunate to have an advisor who gave me the freedom to explore on my own and at the same time the guidance to recover when my steps faltered. Marvin taught me how to question thoughts and express ideas. His patience and support helped me overcome many crisis situations and finish this dissertation. I am also thankful to him for encouraging the use of correct grammar and consistent notation in my writings and for carefully reading and commenting on countless revisions of this manuscript.

Dr. Murali Shanker, has been always there to listen and give advice. I am deeply grateful to him for his assistance and guidance in getting my graduate career started on the right foot. I am also thankful for the long discussions that helped me sort out the technical details of my work.

Dr. Michael Hu's insightful comments and constructive criticisms at different stages of my research were thought-provoking and they helped me focus my ideas. I am grateful to him for holding me to a high research standard and enforcing strict validations for each research result, and thus teaching me how to do research. Michael has also been a close friend and I did enjoy being invited to eat out with him and having him pay for the meals.

I am also grateful to the former or current staff at Kent State University, for their various forms of support during my graduate study. My special thanks go to Ms. Pam Silliman, who has been so helpful in facilitating my study and teaching in the department, as well as so cheerful that I did enjoy the six years in the program.

Many friends have helped me stay sane through these difficult years. Their support and care helped me overcome setbacks and stay focused on my graduate study. I greatly value their friendship and I deeply appreciate their belief in me.

Most importantly, none of this would have been possible without the love and patience of my family. My family, to whom this dissertation is dedicated to, has been a constant source of love, concern, support and strength through all these years. I would like to express my heart-felt gratitude to my wife Dr. Sijing Zong, my dearest daughters Ruobing and Renee, my parents Xiping Zhu and Aihua Yu, my parents-in-law Shouyin Zong and Tianzhi Shi, as well as many others.

Table of Contents

Chapter 1 Introduction and Background	8
1.1 Introduction	8
1.2 Literature Review	16
1.2.1 Models in SCM Research	17
1.2.1.1 Define the Supply Chain	17
1.2.1.2 SCM Defined	19
1.2.1.3 Categories of SCM Models	22
1.2.1.3.1 Models of Contracting Relationships	26
1.2.1.3.2 Models of Information	31
1.2.1.3.2.1 Models of Information Sharing	31
1.2.1.3.2.2 Information Distortion: the Bullwhip Effect	36
1.2.1.3.3 Models of Operational Relationships	39
1.2.1.3.3.1 Pricing Models	39
1.2.1.3.3.2 Inventory Models	41
1.2.1.3.3.3 Capacity Models	43
1.2.1.3.4 Summary of SCM Model Research	44
1.3 Information Sharing Research in SCM	45
Chapter 2 Methodology	50
2.1 Agent Based Modeling	50
2.1.1 Agents	52
2.1.2 Multi-Agent Based Simulation (MABS)	56
2.1.3 Agent Structure Design	62
2.1.4 Agent Based Modeling in SCM	63
Chapter 3 The Model	72

3.1 Conceptual Model	72
3.1.1 Components and Connections	74
3.1.2 Modeling Structure	76
3.2 General consideration of model design architecture	86
3.2.1 Environment	87
3.2.2 Agent objectives and behaviors	88
3.2.2.1 Agent objectives	89
3.2.2.2 Agent behaviors	90
3.2.3 Important Parameters	92
3.2.4 The Architecture	93
3.3 Model implementation	98
 Chapter 4 Experimental Design and Analysis	104
4.1 External Validation	104
4.1.1 The Results	119
4.2 Experimental Design (Internal Validation)	128
4.2.1 The Results	134
 Chapter 5 Conclusions and Future Research	150
5.1 Summary and Conclusions	153
5.2 Future Research	157
5.3 Software Packages for Building and Running the Agent-Based Models	159
 Bibliography	164
 Appendix The Java Code for the Model Implementation	180

Chapter 1 Introduction and Background

1.1 Introduction

A supply chain is a complex dynamic system. Every member (or agent) within the system engages repeatedly in local interactions, giving rise to many flows such as information, material, orders, money, personnel, and capital as defined by Forrester (1959). These flows in turn feed back into the determination of local interactions. The result is an intricate system of interdependent feedback loops connecting micro behaviors, interaction patterns, and flows of different types.

Researchers have grappled with the modeling of supply chain management systems in the past two decades. Nevertheless, the industrial dynamic systems devised by Jay W. Forrester in the late 1950's, in which he systematically defines the levels and flows in a industrial dynamic system, still remains the fundamental paradigm that frame the way many researchers in this field construct models for their research. As detailed in his later work, Forrester (1968) asserts that industrial dynamics, described as the application of feedback to social systems, "is evolving toward a theory of structure in systems ... In high-order, nonlinear systems, with multiple loops and both positive and negative feedbacks, are found the modes of behavior which have been so puzzling in management ..."

The most salient structural characteristic of Forrester's dynamic system model is its strong dependence on the information flow over other flows in the system. All agent interactions are mediated through information feedback mechanisms. However, when researchers study the problems of information exchange (sharing) in a supply chain system as discussed in the chapter that follows, the interactions among the agents are simply and intentionally ignored. This negligence is broadly evidenced in the most common (prevailing) and traditional type of research published in such prestigious journals as Management Science and Operations Research. In these widely-read published studies, every member in a system only takes available information associated with cost (e.g., order quantity, demand forecasting, etc.) as given aspects of their decision problems. Thus, outside of their control, their decision problems reduce to simple optimization problems with no perceived dependence on the action of other agents. This observation holds for the decision problems faced by both suppliers and retailers. Most such research tries to establish models to find the equilibrium (with or without certain mathematical boundaries) and the equilibrium values for the linking cost variables are determined by inventory levels, demand forecasting, and other factors; they are not determined by the actions of, and interactions among suppliers and retailers, or any other agency supposed to actually reside within the system. These mathematically generated equilibriums do not address, and were not meant to address, how flows in a supply chain actually take place in real-world industrial dynamic systems through various forms of information sharing. What Forrester (1968) says still remains mostly true: "... the mathematical orientation of management science, the concentration on analytical solutions, and the optimization objectives could cope only with rather simple situations.

They excluded treatment of the more complex management relationships and also force neglect of most nonlinear phenomena.” In an era that research in many research fields (take behavioral finance as an excellent example) believe that the organizational and human individual behaviors should be taken into considerations in the studies involving organizational and/or human interactions. A famous study is the virtual stock market using the agent based modeling concepts in Santa Fe Institute (see LeBaron, 1998, 2000, and 2001). It seems to the author that the studies of how such behaviors may have impact on a supply chain management system are becoming inevitable.

What, specifically, is meant by “information sharing” in a supply chain? Thumbing through the papers collected on the topics that are related to information sharing, we surprisingly find that there is not a single author that has ever given the convincing and widely agreeable definition. Researchers simply take the magical phrase and apply it to their topical area with the perception that the meaning of it is naturally clear and specific. As a result, with no surprise, the scope of information sharing varies widely in the collected literature. This dissertation does not intend to supply such a characterization that is globally accepted, but a working definition is necessary to draw a line that confines the research scope of the study.

Literally, “information sharing” consists of two meaningful components: information and sharing, which propose three questions: “what information is shared?”, “how does sharing take place?”, and, putting the two components together, “how (and how much) information is shared?” The answer to the first question of the type of information can be collectively answered by authors in the field, which covers broadly the (advanced)

demand (forecasting) (Thonemann, 2002; Bourland et al., 1996; Fisher and Raman, 1996; Gurnani and Tang, 1999; Donohue, 2000), inventory (Bourland et al., 1996; and many papers in other areas that consider inventory), capacity (replenishment policy) (Gavirneni, et al., 1999; Gallego and Ozer, 2003; Graves, 1999; Cachon and Fisher, 2000,), general information costs (Lee et al., 2000; Raghunathan, 2001; Chen et al., 2000), and external information feed (competition, cooperation, etc.) (Li, 2002; Padmanabhan and Png, 1997; Anand and Mendelson, 1997). To answer the second question of how sharing takes place, we may look into the interactions between the parties involved, which is the area that has not been well studied in literature. Some authors look into the research that the third question is to examine and try to find out how much information is shared. Our study defines four different modes of information sharing in an attempt to pave a way for future research to possibly extract a general model that uses agents to simulate the behaviors of each member in a supply chain and scrutinize how interactions among these members may affect the benefits of information sharing.

As noted above, information sharing as an important field of study in supply chain management is typically studied using standard mathematic approaches that focus on the equilibrium states of a system. Apart from many significantly important advantages of the traditional approaches, one major drawback of them is the lack of flexibility to accommodate dynamic interactive activities in the systems and thus, due to such interactions, the dynamics and commonly computationally intractable nature of the systems is partially or totally ignored. Fortunately, over time, increasingly sophisticated tools are developed to permit researchers to investigate supply chain management

research in increasingly compelling ways. Some of these tools involve advances in computational power and artificial intelligence (AI) that uses software agents.

From the AI or general computational point of view, multi-agent systems (also known as the agent based modeling, i.e., in short the ABM) and autonomous agents represent a new way of analyzing, designing, and implementing complex software systems. A supply chain, as we have noted earlier, is a complex system and may be represented in a software environment (i.e. simulation), which is obvious that we may apply autonomous agents and ABM to it. The agent-base view offers a powerful repertoire of tools, techniques, and metaphors that have the potential to considerably improve the way in which people conceptualize and implement many types of research. One important reason for the application of ABM in a supply chain system is the capability of modeling the sophisticated patterns of information flow interactions. Examples of common types of interactions include, as carefully detailed by Jennings et al. (1998):

- cooperation (working together towards a common goal, such as to minimize inventory costs);
- coordination (organizing problem solving activity so that harmful interactions are avoided or beneficial interactions are exploited, such as timing in production and transportation); and
- negotiation (coming to an agreement which is acceptable to all the parties involved, such as when the capacity is constrained, a retailer may need to negotiate with the supplier or other retailers about the quantity they may be entitled to or request for).

It is the flexibility and high-level nature of these interactions that distinguish MAS from other approaches and provide the underlying power of the paradigm.

With such understanding as the important role that information sharing may play in a supply chain, the critical impact of interactions among agents in the system on the information sharing, and the clear scope of study about information sharing, this dissertation attempts to answer the following questions:

- How can ABM help us in SCM modeling when examining the impact of information sharing? And
- How much information should be shared to achieve optimal supply chain performance?

To answer these questions as well as to explore the problem structure in more depth, we first investigate the current literature on SCM modeling and identify a way of categorization; we then examine the literature of ABM development and extract the ABM modeling methods in computer science discipline; and thirdly we explore the connections between ABM and SCM modeling to examine whether the ABM approach is an improved alternative to traditional numerical analysis methods. We develop a multi-agent based simulation modeling system for a two-stage supply chain that consists of a manufacturer (supplier) and three retailers and analyze the benefit of information sharing to the chain. We model the members in this supply chain using multi agents, taking into account the interactions between / among these members, to evaluate and compare the benefits of information sharing to those from the existing literature.

The intended contribution of this research is described below. On the one hand, existing literature suggests that the dominating methodology for modeling a supply chain is to develop a mathematically sound theory and to use numerical examples to provide artificial evidence that supports the theory. This approach may be pragmatically of less usefulness for the managers due to its lack of the ability of capturing what is happening in the ever-changing market place. As Cachon and Fisher (2000) listed, the traditional approach, when examining the value of shared information, has many limitations such as 1, the demand is known to the retailers; 2, the retailers are identical; 3, there exists only a single source for inventory; 4, there are no capacity constraints; 5, there are no incentive conflicts among the supply chain's firms; and 6, firms choose rational ordering policies. Although some of the assumptions are still used in our model design and development for our particular case, ABM in fact is capable of relax all the assumptions in the future research. This approach developed in this dissertation encompasses the intelligent agents found in computer science, which are capable of (at least in theory, though many concepts are yet to be implemented due to the limitations of technology advancement nowadays) modeling the interactions among members in a supply chain. Thus, using the mathematically sound theory with the pragmatically useful techniques, a more realistic analytical environment is set up for better decision making.

On the other hand, only a very small number of papers exist using multi-agent simulation approach and they simply model the one-to-one control type of relationships between supplier and retailers. This approach not only does not fully take advantage of the rich capabilities of the agents; but also it fails to consider that there exist interactions among agents (i.e., the members of the supply chain) and as stated previously does not give a

complete, if not incorrect, picture of the subjects being studied. This dissertation explores the connections between multi-agent system approach widely found in computer science and the mathematical modeling method in SCM and shows that agent based modeling is a good alternative.

The third intended contribution is that although an ample number of studies have studied the impact of information sharing in a supply chain system, these studies all look at the supply chain as a vertical structure in which the information and material flows solely vertically. We claim that the supply chain should not be as simple as the most commonly used the two echelon systems in which one supplier interacts with each individual retailer and the retailers do not exchange information and materials at all. The horizontal interaction / information sharing may play an important role in supply chain performance improvement. The new dimension we add to the system makes the supply chain not flat any more, it becomes a 3-D type of structure. Our study opens the opportunities for examining a supply chain system that is closer to the real world, the reality.

Human and organizational behaviors are very important factors that have great impacts on the performance of a system (like SCM) to be modeled. Any models (theoretical) without considering these factors are prone to imperfection, even failure, when applied to a real situation. Multi-agent simulation provides us a powerful and flexible instrument to accomplish such tasks that mathematical models and numerical analysis fail to do. Our method in the dissertation is to apply the simple but generic information-feedback systems theory found in other disciplines (mostly in science and applied science) and the intelligent agents borrowed from the contemporary artificial intelligence research and

applications to model the inter-relationships and interactions among the subjects (firms in a supply chain) studied. Each of such subjects is modeled by an agent cluster. A modular design approach is applied to the architecture to ensure full flexibility and expandability.

The structure of the dissertation is laid out as follows. We first review the current literature. The literature review in this chapter covers models in SCM, multi-agent simulation in SCM, the research in multi-agent simulation, and information-feedback systems theory. We then describe our methodology in chapter 2. In this chapter, introduce the agent based modeling. A conceptual model is illustrated afterwards in Chapter 3, in which we develop a conceptual agent based model for our experiment architecture; and in Chapter 4, we compare our results from the agent based model with those from the literature as the external validation; and the experimental design is outlined in detail in this chapter. The implementation of the model outlined uses Java agents running in Repast (an open-source agent model implementation environment). The analysis and discussion complete this chapter. Chapter 5 concludes the dissertation. Future research directions are discussed.

1.2 Literature Review

As Ganeshan et al. (1999) point out, efforts to describe and explain supply chain management (SCM) have recently led to a plethora of research and writing in this field. At the same time, the level of attention to SCM received in business practices nowadays also heavily influences the growing interest in SCM research. Several researchers have attempted to provide some taxonomies and frameworks to present both practitioners and

academics cohesive information that explains the SCM concept and emphasizes the variety of research work being accomplished in this area (See Bowersox, 1969; Laugley, 1992; Bechtel and Jayaram, 1997; Ganeshan, et al., 1999; and Tan, 2000). These authors, among others, have reviewed relevant streams of thoughts in SCM research, provided integrative frameworks to help design and manage supply chains, or categorized the existing research on SMC to offer organized information for other researchers.

Despite the efforts of previous authors, we believe that the growing and rich literature in SCM warrants a close examination of SCM models published in major operations research and management journals. By doing so, we will be able to better understand what the areas are studied and how these areas of study are modeled, and synthesize future directions of SCM research. We classify these models found in major academic journals into three categories, namely, contracting relationship models, information models, operational relationship models.

1.2.1 Models in SCM Research

1.2.1.1 Define the Supply Chain

Supply chains have existed ever since business has been organized to bring products and services to customers (Kumar, 2001). Many variations are found in literature on the same theme when defining a supply chain. There are two major views of supply chain in the literature.

One school takes the system view which can be found in Houlihan (1985), Jones and Riley (1984), Stevens (1989), Scott and Westbrook (1991), and Lamming (1996). This

theme of thought believes that supply chain is a system of suppliers, manufacturers, distributors, retailers, and customers where materials flow downstream from suppliers to customers and information flows in both directions. As Mentzer et al. (2001) put it: a supply chain is a set of entities (organizations or individuals) directly involved in the upstream and downstream flows of products, services, finances, and/or information from a source to a customer.

Other authors view supply chain as a network of organizations and their associated activities that work together, usually in a sequential manner, to produce value for the consumer (Kumar, 2001). Swaminathan et al. (1998) completes this network view by defining a supply chain as a network of autonomous or semiautonomous business entities collectively responsible for procurement, manufacturing and distribution activities associated with one or more families of related products.

Both views have accurately described the entities, activities, and missions of a supply chain from different perspectives and each has its own emphasis. For the system view, it focuses on the processes of making from raw material to final products and how these products are handed to customers in an effective and efficient way, as well as how information is passed within this system to support those processes. While the network view aims to explain the supply chain through the inter-relations and inter-actions between each entities involved. These entities are highly interdependent when it comes to improving performance of the supply chain in terms of objectives such as on-time delivery, quality assurance, and cost minimization (Swaminathan et al., 1998). Authors with the latter view identify different functions (groups of entities working closely) in a

supply chain such as, procurement of material, transformation of material to intermediate and finished products, and distribution of finished products to customers, etc. (Lee and Billington, 1993).

1.2.1.2 SCM Defined

The term SCM appears to have originated in the early 1980s when Oliver and Webber (1982) discuss the potential benefits of strategically integrating the internal business functions of purchasing, manufacturing, sales and distribution (Harland, 1996). The idea of SCM emerges from logistics management integration, which shifts the focus from materials management to the movement of material throughout the firm in an organic and systemic way to greatly improve the effectiveness and the efficiency of the operation (La Londe ad Masters, 1994).

SCM is different from logistics management, as often confused by practitioners and academics. According to the definition given by the Council of Supply Chain Management Professionals (CSCMP), “Logistics management is that part of supply chain management that plans, implements, and controls the efficient, effective forward and reverse flow and storage of goods, services, and related information between the point of origin and the point of consumption in order to meet customers’ requirements.” (CSCMP, 2005) The transition of logistics (materials) management to SCM is a result that what were hitherto considered mere logistics problems have emerged as much more significant issues of strategic management. Houlihan (1985) studies firms in a variety of industries in the USA, Japan and Western Europe and finds that the traditional approach of seeking trade-offs among the various conflicting objectives of key functions such as purchasing,

production, distribution and sales, along the supply chain no longer worked very well and calls for a new approach, which is SCM.

SCM consists of a decision support system, which is concerned with determining supply, production and stock levels in raw materials, subassemblies at different levels of the given Bills of Material (BoM), end products and information exchange through (possibly) a set of factories, depots and dealer centers of a given production and service network to meet fluctuating demand requirements (Escudero et al., 1999). The necessity of managing the supply chain is mainly contributed by three factors, as indicated by Kumar (2001). These factors include: first, customers become more cost and value conscious and demand more, varied, often individualized value from the supply chain; second, the modern information and communication technologies enable the firms to obtain an overview of the entire supply chain so that they can redesign and manage it to meet this demand; finally, the emergence of global markets and global sourcing have stretched these supply chains over inter continental distances. As a result, the accumulated demand variety, uncertainty, costs, distances, and time lags on a global scale make it even more imperative that these long chains be managed efficiently and effectively. Consequently the focus shifted from the competitive advantage of firms to competitive advantages of entire supply chains.

The definitions of SCM differ across authors. For examples, Monczka et al. believe that SCM is a concept “whose primary object is to integrate and manage the sourcing, flow, and control of materials using a total systems perspective across multiple functions and multiple tiers of suppliers.” Jones and Riley (1995) state that SCM deals with the total

flow of material from suppliers through end users. Cooper et al. (1997) define SCM as an integrative philosophy to manage the total flow of a distribution channel from supplier to the ultimate user. Other examples of definitions can be found in La Londe and Masters (1994), Stevens (1998), and Houlihan (1985). These definitions can be classified into three categories: a management philosophy, implementation of a management philosophy, and a set of management processes. Put them together, Mentzer et al. (2001) give a definition: SCM is “the systemic, strategic coordination of the traditional business functions and the tactics across these business functions within a particular company and across business within the supply chain, for the purposes of improving the long-term performance for the individual companies and the supply chain as a whole.”

It is worthwhile to mention that owing to the rapid advances of information and communication technology, the most recent development in the concept of SCM includes the Internet as one of the key influential factors. The e-commerce, business-to-business (B2B) and business-to-customers (B2C) through the Internet, is transforming organizations and organizational processes and creating new opportunities and challenges for domestic and international companies and their supply chains as the Internet is enabling greater integration of businesses and a blurring of traditional organizational boundaries (Bakos, 1998, Hitt et al., 1999, Lancioni et al., 2000, Overby and Min 2001, Johnson and Whang, 2002). This transformation dramatically changes the relationships between the entities in a supply chain and the perception of the traditional SCM. There is at this moment not a definition of this new SCM from the literature available. As noticed by Lancioni et al. (2000), to date, there have been few academic studies examining the development and use of the Internet in SCM.

For the purpose of this dissertation, we give a working definition of SCM which is built on the Mentzer et al. (2001)'s as well as others definition.

SCM is the system that aims to coordinate the interrelations and interactions among networked business functions, to manage the information flows between these business functions, and to provide strategic and tactical decisions in an effective and efficient way with the help of available information and communication technologies within a particular company and across business within the supply chain, for the purposes of improving the long-term performance for the individual companies and the supply chain as a whole.

1.2.1.3 Categories of SCM Models

The objective of classifying SCM models is to provide a clear overview of the types of models that have been researched in the current major literature organized in the way that we can decide and formulate a model for the dissertation to work from.

There are only a small number of papers that classify the research in SCM. Meixell and Gargeya (2005) review decision support models for the design of global supply chains and assess the fit between the research literature in this area and the practical issue of global supply chain design. Their review is thorough and well organized. However, it does not fit the purpose of our review for the following reasons:

- 1) The review focuses on models for designing a supply chain rather than the management of the supply chains.

2) It investigates the models based on the years that they are developed, i.e., period prior to 1990, between 1991 and 1995, from 1996 to 2000, and years after 2000.

This would not give us a clear view of the focuses of the models.

3) It looks into the models by using four dimensions – decision variables, performance measurements, supply chain integration, and globalization considerations.

4) It does not scrutinize the models based on the problems that each model is trying to solve.

Ganeshan et al. (1999) classify the SCM research into three broad perspectives; competitive strategy, firm focused tactics, and operational efficiencies. These perspectives are further divided into many subcategories. The authors try to give a broad review of updated chart of the historical development of SCM and their focus is on much broader field in SCM research and has not specifically synthesized the models in SCM. Therefore it will not fit the purpose of this paper, either.

Many models have been developed in the SCM literature. Authors address various issues from different aspects of SCM. In this review, we group their focuses into three categories, which are information models, contracting relationship models, and operational relationship models. This categorization is broad and based on the following questions that each categorical model needs to answer:

- What is the focus that the target model is trying to place on? Is it on information or relationships (contracting or operational) issues? Each type of modeling has its own specific questions as described in the following explanations.

➤ Information models:

- ✓ What type of information is the focus of the study? One of the common information often addressed in the literature, for example, is in a typical two-echelon supply chain model the demand information to examine the bullwhip effect (Lee et al. 1997). Other information may include the sales data (real time or batch mode), inventory level information, delivery information, etc. Readers who are interested in more detailed description of the types of information may be referred to Lee and Whang (2000).
- ✓ How information is handled in SCM? Such studies include the use of traditional or electronic data exchange technology to make the information available when needed. Some of common technology include EDI (Electronic Data Interchange), Intranet, Internet, VPN, and many others.
- ✓ What are the roles of information in SCM modeling? Information of different types may play different roles and the sharing of different information may have different impacts to the supply chain performance in magnitude as well as levels.
- ✓ How does the model deal with information distortion, information asymmetry, information sharing, and other phenomenon found in SCM? What are the impacts and how does the model manage to cope with them?

➤ Contracting Relationship models:

- ✓ What are the relationships that SCM modeling is interested in – manufacturer/supplier-retailer, retailer-customer, or others?

- ✓ How does the contract affect the relationship between the entities in SCM and how does a model map this relationship and find the solution to the problem if there is any?
 - ✓ How does a model deal with the supplier selection issues?
- Operational Relationship models
- ✓ What are the capacity, pricing, or inventory policies and how can a model find the optimal one?
 - ✓ What are the variables that a model identifies to work with the capacity, pricing, or inventory problems?
 - ✓ How does the uncertainty in demand impact the decisions on capacity choice and allocation, pricing, and inventory policies? What are the models examining them and how do these models work?

This review focuses on the model-based literature and we conducted a search using library databases covering the major journals in management science and operations management, such as *Management Science*, *Operations Research*, *European Journal of Operational Research*, *Decision Sciences*, *the Journal of Supply Chain Management*, *Decision Support Systems*, etc. Research papers found are grouped into four categories, each of which has subcategories. Figure 1 provides a guideline of how this classification is structured in this paper. The following section gives detailed descriptions of each category.

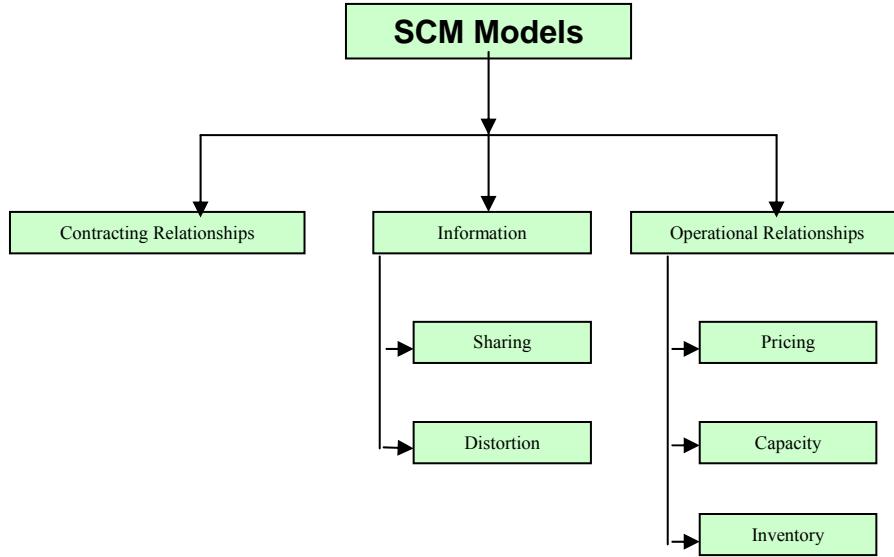


Figure 1: SCM Models Categorization Structure

1.2.1.3.1 Models on Contracting Relationships

Harland (1996) views SCM as the management of supply relationships, i.e., “an intermediate type of relationship within a spectrum ranging from integrated hierarchy (vertical integration) to pure market.” This perspective of SCM has as its foundations an industrial organization and contract view of the firm as a nexus of contracts (Aoki, 1990). Models are found to address the relationship problems related to contracting, incentives, and supplier selection.

Hui and Beath (2001) point out that the natural artifact to analyze the inter-organizational relationship in modern industrial organization is the contract. Contracts not only serve as simply a list of rules, but also define the tone and nature of the relationship. The contract has become an exemplary subject for studying SCM because it is the written artifact that results from firms in a supply chain trying to understand the nature of the relationship

(Walden, 2002). The models that study contracts in supply chain are summarized in

Table 1.

Corbett and DeCroix (2001) study how contracts affect consumption of indirect materials, which are consumed during the production process but do not become part of the final product, by influencing the amount of effort supplier and customer exert to reduce that consumption. The authors create a mathematical model to analyze the impact of shared-savings contracts on channel profits and material consumption assuming that the variable component is linear in the quantity of indirect material used. While such linear contracts can yield higher profits as well as possibly lower consumption, the resulting equilibrium effort levels are generally not channel-optimal outcome since such contracts do not necessarily lead to lower consumption.

Corbett et al. (2005) further the study in the stream by addressing the questions of determining the optimal shared-savings contract from the supplier's perspective, other than the channel perspective adopted in Corbett and DeCroix (2001), in a more general setting. The authors use the double moral hazard framework, in which both parties (consumer and supplier) decide how much effort to exert by trading off the cost of their effort against the benefits that they will obtain from reduced consumption. The model they use extends the double moral hazard literature to allow for a broader class of cost-of-effort functions, including the linear functions found in practice, and shows that the supplier's optimal contract still consists of a fixed part and a variable part which is linear in consumption.

Similarly, the research of Tsay (1999) considers a supply chain consisting of two independent agents, a supplier (e.g., a manufacturer) and its customer (e.g., a retailer). He finds that the retailer, who serves an uncertain market demand and typically provides a planning forecast of its intended purchase, has incentive to initially over-forecast before eventually purchasing a lesser quantity. The supplier must in turn anticipate such behavior in its production quantity decision. This individually rational behavior results in an inefficient supply chain. The author applies the quantity flexibility (QF) contract which couples the retailer's commitment to purchase no less than a certain percentage below the forecast with the supplier's guarantee to deliver up to a certain percentage above. An equilibrium solution is given and the efficiency of QF contract is evaluated through numerical analysis. Under certain conditions, this method can allocate the costs of market demand uncertainty so as to lead the individually motivated supplier and customer (retailer) to the system-wide optimal outcome.

In the same line of research, instead of focusing on using the quantity flexibility contracts, Eppen and Iyer (1997) study the backup agreements (contracts) between a catalog company and manufacturers – a scheme to provide upstream sourcing flexibility for fashion merchandise. As authors define, a backup agreement states that if the catalog company commits to a number of units for the season, the manufacturer holds back a constant fraction of the commitment and delivers the remaining units before the start of the fashion season. After observing early demands, the catalog company can order up this backup quantity for the original purchase cost and receive quick delivery but will pay a penalty cost for any of the backup units it does not buy. They use a dynamic programming model to derive the form of the optimal policy. The model is evaluated by

performing a retrospective parallel test on the data obtained from a catalog company and they conclude that backup agreements can benefit both the retailer and the manufacturer and the adjusting the order commitment in response to the offered percentage to hold by the retailer can have a significant impact on expected profit.

Cachon and Lariviere (2001) compare revenue sharing contracts with other types of contracts we have mentioned above, among others. They found the revenue sharing contracts are at least equivalent with, and most times superior to other types of contracts. Their base model has a supplier selling to a single retailer. The retailer makes two decisions that determine the total revenue generated over a single selling period: the number of units to purchase form a supplier and the retail price. The functions derived show that revenue sharing coordinates this supply chain; i.e., the retailer chooses supply chain optimal actions (quantity and price) and the supply chain's profit can be arbitrarily divided between the firms. Further, a single revenue sharing contract can coordinate a supply chain with multiple non-competing retailers even if the retailers have different revenue functions. However, revenue sharing generally does not coordinate competing retailers when each retailer's revenue depends on its quantity, its price, and the actions of the other retailers.

Cachon (2004) looks into the above issues from inventory risk management perspective. He considers, as others, a supply chain with one risk-neutral supplier buying a product one risk-neutral retailer and demand for the product being stochastic over a single selling season, and studies how the allocation of inventory risk through various types of contracts influences a supply chain's performance and its division of profit. Three types

of wholesale price contracts are modeled. With the push contract, the retailer must pre-book inventory and the supply only produces the retailer's pre-book quantity, therefore, all inventory risk is pushed onto the retailer. In contrast, with the pull contract, the retailer pulls inventory from the supplier with at-once orders, thereby leaving the supplier with all inventory risk. The third type is the advance-purchase discounts, which blends both push and pull by having two wholesale prices. The retailer may pre-book some inventory at a lower price than at-once wholesale price to bear the risk on that inventory; and the supplier may produce additional inventory in anticipation of at-once orders to bear the risk on that additional production. Cachon (2004)'s research shows that the allocation of inventory risk matters for supply chain efficiency even if firms are risk neutral. It is also shown that without changing the wholesale price, merely shifting the inventory risk from one firm to another can improve supply chain efficiency and increase profit at both firms. In addition, if the firms are willing to share inventory risk via advance-purchase discounts, then supply chain coordination is achievable with any division of the supply chain's profit.

A more recent study on shared demand forecasts in a supply chain is conducted by Cachon and Lariviere (2001). The authors identify three key components of a software development contract – product definition, intellectual property protection and payment. They develop a game-theoretic model to incorporate incentive and information issues associated with the payment structure in software contracting. They derive the structure of a viable contract that aligns the incentives of the contracting parties and produces the same efficient equilibrium outcome as in in-house development. However this model has several drawbacks because of some unrealistic assumptions, such as players in the game

are risk-neutral; the design will not change in the middle of the development; and it assumes that the value of a design is exactly determined based on the prototype.

Baiman et al. (2001) model the contracting relationship between a supplier and a buyer, which not necessary is a retailer. However, their focus is on the buyer outsourcing the production of some part to the supplier. They study the interactive effect of the performance metric chosen and the architecture of the product being manufactured on the incentive efficiency of the supply chain. The performance matrices studied are the occurrence of an external (final product fails after sale) and /or internal failure (defective product supplied by the supplier).

Contracting relationship is one of the most important artifacts in SCM. Number of literature can be found and we pick several representative ones to demonstrate the main stream models that have been studied to date.

1.2.1.3.2 Models on Information

SCM is concerned with coordination of independent enterprises in order to improve the performance through the whole supply chain by considering their individual needs. One of the important issues of the coordination is to manage the product and production information among them (Lau et al., 2004). The models that study the management of information in supply chain deal mainly with the issues as related to information sharing and information distortion.

1.2.1.3.2.1 Models on Information Sharing

Research on SCM information modeling has different focus as stated above. However, literature shows that the emphasis of this theme of research heavily leans to information sharing. The most commonly asked questions are, “What is the value of information and information sharing?” “What and how information are shared in supply chain?” and “How do we model the information sharing in supply chain?” The answers to these questions are the models as summarized in the following text.

Gavirneni et al. (1999), Lee et al. (2000), and Raghunathan (2001) model the value of information and information sharing by incorporating information flow between a supplier and a retailer in a typical two-echelon supply chain. Gavirneni et al. (1999)'s model captures the capacitated setting of a supply chain. They consider three situations with three models: 1) a traditional model where there is no information to the supplier prior to a demand to him except for past data; 2) the supplier knows the (s, S) policy (i.e., at the supplier-retailer interface there is an implicit fixed ordering cost) used by the retailer as well as the end-item demand distribution; and 3) the supplier has full information about the state of the retailer. The authors' computational results of these models show that information is always beneficial but the degrees may vary. That is, in the case that end-demand variance is high, or the value of $\Delta = S - s$ is very high or very low, the benefits of additional information is not as great as in the other case when end-item variance is moderate and the value of Δ is not extreme.

While the research by Gavirneni et al.. (1999) is based on demand processes that are independent and identically distributed over time; thus the benefit of information sharing lies in the supply's capability to react to the retailer's needs via the knowledge of the

retailer's inventory levels to help reduce uncertainties in the demand process faced by the supplier, Lee et al. (2000) examine a different situation in which the underlying demand process is a simple auto correlated AR(1) process. A supplier can benefit from obtaining information about the demand from the retailer because it would enable the supplier to derive a more accurate forecast of future orders placed by the retailer. The analytical and mathematical analyses show that the supplier can benefit from inventory reduction and cost reduction with information sharing (of demand). In addition, the authors find that under the conditions that underlying demand is highly correlated over time, highly variable, or when the lead time is long, the supplier obtains larger benefits.

However, the subsequent research by Raghunathan (2001) reaches a different result. He finds that the supplier's benefit is insignificant when the parameters of the AR(1) process are known to both parties, as in Lee et al. (2000). The key reason for the discrepancy is that Lee et al. (2000) assume that the supplier also uses an AR(1) process to forecast the retailer order quantity. Nonetheless, the supplier can reduce the variance of its forecast further by using entire order history to which it has access. Thus, when intelligent use of the already available internal information (order history) suffices, there is no need to invest in inter-organizational systems for information sharing. However, the AR(1) demand process of Lee et al. (2000) may be too simplified. Some retailer actions such as promotion, price reduction, and advertising taken during next period may change the AR(1) demand process over time. In this case, if the retailer actions are independent of each other, the information sharing will still be valuable because the retailer actions can not be inferred by the supplier using order history. In a more general case, the normality assumption of the AR(1) demand model of Lee et al. (2000) may also be over-simplified.

A slightly different research from Gavirneni et al. (1999) and Lee et al. (2000) is by Cachon and Fisher (2000), who study the value of sharing demand (received by retailers) and inventory data (inventory replenishment policy, as well as the current inventory positions) in a model with one supplier, N identical retailers, and stationary stochastic consumer demand. Cachon and Fisher (2000) compare a traditional information policy that does not use shared information with a full information policy that does exploit shared information to allocate the supply chain inventory based on the retailers' actual inventory positions rather than the number of batches they order. Both numerical and simulation based analysis show that the latter case offers cost savings. In addition to the contributions offered by the other authors, Cachon and Fisher (2000) developed a lower bound over all feasible policies. They find that the cost difference between the traditional information policy and the lower bound is an upper bound on the value of information sharing. The more interesting result is that by contrasting the value of information sharing with the benefits of information technology (IT), i.e., faster and cheaper order processing, the authors conclude that implementing IT to accelerate and smooth the physical flow of goods through a supply chain is significantly more valuable than using IT to expand the flow of information. This view is supported by the research of Srinivasan et al. (1994), who use baseline logit model to explore the relationship between shipping discrepancies and two factors (JIT schedules and EDI integration) that capture the level of vertical information integration between a firm and its suppliers. This study suggests that gains from JIT systems can be substantially increased by modern information technology support.

Although information sharing has the potential to dramatically improve supply chain performance, some research find this may not always be true. A recent paper by Terwiesch et al. (2005) studies information sharing from the demand forecast perspective. The authors find that the retailer's forecasting behavior can be characterized by the frequency and magnitude of forecast revisions it requests (forecast volatility) as well as by the fraction of orders that are forecasted but never actually purchased (forecast inflation). They model the evolution of a soft order to a firm order and ultimately to a delivered piece of equipment in the form of a two-stage process. The first state captures the fact that soft orders can either end up as firm orders, that is, the buyer places an order, or be cancelled. In the second state, a firm order will see a delivery time. They find that forecast sharing does not provide benefits to the performance of the supply chain. This may be explained from two perspectives. As for the supplier, the forces that prevent effective forecast sharing are forecast volatility and forecast inflation. The supplier is not willing to allocate production capacity to the soft order that has changed multiple times and penalizes the retailer for inflated forecasts through longer delivery times. On the other hand, retailer will provide more aggressive forecasts to those suppliers that have failed to deliver previous orders on time. This follows the logic of the repeated prisoner's dilemma game and establishes that both retailer and supplier apply a tit-for-tat strategy. The authors call for research to analyze supply chain coordination in repeated game settings and to overcome the forecast volatility problem.

Most of the research as reviewed above study the models that apply to information sharing for inter-organizational supply chain management and very few has looked into the intra-organizational supply chain. Unlike others, Chen considers a supply chain

whose members are divisions of the same firm. Under the assumption that he division managers share a common goal to optimize the overall performance of the supply chain, which is difficult to achieve in an inter-organizational supply chain, the author develops a team solution that reveals that information lead time in determining the optimal replenishment strategies offer cost savings.

1.2.1.3.2.2 Information Distortion: the Bullwhip Effect

As noted in the previous text, one important mechanism for coordination in a supply chain is the information flows among members of the supply chain. These information flows have a direct impact on the production scheduling, inventory control and delivery plans of individual members in the supply chain. Several researchers (Forrester, 1959, Sterman, 1989, Lee et al. 1997, etc.) in management science notice that there exists systematic distortion in demand information as it is passed along the supply chain in the form of orders. This information distortion is called bullwhip effect.

The “bullwhip effect” was first coined by Lee et al. (1997) in their earlier working paper on the same topic. This is a phenomenon where orders to the supplier tend to have larger variance than sales to the buyer (i.e. demand distortion), and the distortion propagates upstream in a amplified form (i.e., variance amplification). Evidence of the bullwhip effect is reported by Sterman (1989) in an inventory management experimental context called “Beer Distribution Game.” This experiment, under the linear cost structure, shows that the variances of orders amplify as one moves up in the supply chain. The author interprets the phenomenon as a consequence of players’ systematic irrational behavior, or “misperceptions of feedback.” Another evidence is reported by Kahn (1987), who

models inventory behavior that incorporates stockouts, backlogs, and serially correlated demand in a supply chain. He reports that the variance of production will exceed the variance of sales even in the absence of movements in productivity or costs. Lee et al. (1997) in their study develop mathematical models of supply chain that capture essential aspects of the institutional structure and optimizing behaviors of members who are assumed to be rational and optimizing. These assumptions are important because the authors employ mathematical models to explain the outcome of rational decision making (for example, the supplier may allocate the supply in proportion to the retailer's share of total sales of the product), as opposed to deriving an optimal decision rule for managers, which may cause the nonproductive gaming (for example, the manager of a retailer who assesses the possibility of being placed on allocation by the manufacturer due to insufficient supply. He may exaggerate the actual demand to get the greater allocation).

There are four causes of bullwhip effect, namely, demand signal processing, the rationing game, order batching, and price variations. Lee et al. (1997) use four models to investigate the effects of the four causes that lead to systematic distortions of information in the order-replenishment transactions of a standard supply chain. Table 3 lays out the mathematical models from their research in addition to the research by others in this field.

At the same time, Metters (1997) uses a classical approach to determine the optimal policies by dynamic programming as the model used by Zipkin (1989). He concludes that a lack of inter-organizational communication combined with large time lags between receipt and transmittal of information are at the root of the problem.

Cachon (1999) models the supply chain demand variability using the same settings as Lee et al. (1997) with one supplier, N retailers, and stochastic demand. In addition to the findings of lee et al. (1997), the author shows that the supplier's demand variance will generally decline as the retailers' order interval is lengthened or as their batch size is increased. By reducing supplier demand variance with scheduled ordering policies, the total supply chain costs can be lowered.

Another important finding about bullwhip effect is reported in Chen et al. (2000). Beginning with a simple supply chain model with one retailer and one supplier, the authors quantify the bullwhip effect by considering two of the factors: demand forecasting and order lead times. They show that if a retailer periodically updates the mean and variance of demand based on observed customer demand data, then the variance of the orders placed by the retailer will be greater than the variance of demand. The authors then extend the results to multiple-stage supply chains and find that providing each stage of the supply chain with complete access to customer (centralized) demand information can significantly reduce this increase in variability. However, the bullwhip effect can not be eliminated fully as noticed by other authors (Metters, 1996; Cachon, 1999)

In summary, the bullwhip effect phenomenon has been described in the literature over many years; however, it is only in the past decade that the full extent of the problem has been recognized, which has stimulated the interest of a number of researchers. It is noticeable that better information sharing between the members in a supply chain may help mitigate the damages of bullwhip effect. The advances in information technology

may offer better and more efficient channels and methods for information sharing. Some future research may investigate the impact of bullwhip effect in the information age and define new models to alleviate deficiency caused by this phenomenon.

1.2.1.3.3 Models on Operational Relationships

Operational relationships in SCM are widely researched and they mainly cover the models in pricing, inventory, and capacity management. The following sections review the literature in these three sub-categories.

1.2.1.3.3.1 Pricing Models

Not all supply chain models are directly related to information sharing. However, to make the categorization complete, we include the pricing models here briefly.

Research in pricing models are mostly related to “timing” for limited capacity product, that is, prices are determined based on the time priorities (the user needs the product now – higher priority, or he can wait until the product is available after a delay – lower priority), and the prices are set based on the priorities. Several papers study the internal pricing for service facilities (Mendelson and Whang, 1990; and Dewan and Mendelson, 1990), others study the pricing policies in the competing firms in a supply chain (Lederer and Li, 1997). Also others study the pricing for optimal bundling strategies for information goods.

Dewan and Mendelson (1990) study optimal pricing and capacity decisions for a service facility in an environment where users’ delay cost is important. Their model assumes a

general nonlinear delay cost structure and incorporates the tradeoff between the delay cost and capacity cost. A queuing model is defined by considering a flow of service requests generated by a continuum of atomistic individual users to the system. Each job is set a price by the service department and each user makes individual decision on whether or not to submit his jobs for service. Under certain restrict assumptions, the job arrival and service times are modeled in a queue. Based on the short-run and long-run situations, the authors develop the optimal pricing scheme that would lead to an optimal utilization of the available capacity.

Also using the queuing theory, Mendelson and Whang (1990) conduct a similar research and derive a pricing mechanism which is optimal and incentive-compatible in the sense that the arrival rates and execution priorities jointly maximize the expected net value of the system while being determined, on a decentralized basis, by individual users. The resulting price structure from their model reveals how the factors of job length and priority each contribute to the overall costs inflicted by a job on the rest of the system.

The queuing theory is also used by the research by Lederer and Li (1997), when they study how delay costs affects prices, operating policies, sales, and firm profits in a competitive environment. This paper assumes that firm capacity, processing variability and cost function are all fixed. The prices are determined by the competitive equilibrium developed in the paper.

One type of the products in a supply chain is information goods – software or other copyrighted materials that are distributed online (almost costless via data networks such as Internet) or in stores. The existing traditional theory and practice in SCM are not

suitable for providing clear guidance on how digital information goods should be packaged, priced, and sold. Bakos and Brynjolfsson (1999) analyze the strategy of bundling a large number of information goods and selling the bundle for a fixed price. The authors use statistical techniques to provide strong asymptotic results and bounds on profits for bundles of any arbitrary size. It is shown that the model can be used to analyze the bundling of complements and substitutes, bundling in the presence of budget constraints, and bundling of goods with various types of correlations and how each of these conditions can lead to limits on optimal bundle size.

1.2.1.3.3.2 Inventory Models

Inventory management may be one of the most important fields and takes a major portion of the SCM models researched in the category of operational relationships.

An early paper by Topkis (1968) considers the problems associated with an inventory system in which demands for stock are prioritized based on its classes of importance. The author investigates the conditions that will satisfy the optimal rationing policy. A later research by Nahmias and Demmy (1981) continues the study and model an inventory system which maintains stock to meet both high and low priority demands. They analyze the following type of control policy: there is a support level, say $K > 0$, such that when the level of on hand stock reaches K , all low priority demands are backordered while high priority demands continue to be filled. Both continuous review and periodic review systems are considered. They compare fill rates when there is rationing and when there is no rationing for specified values of the reorder point, order quantity and support level.

Along the same line, Axsater (1993) by comparing the one-for-one replenishments at the retailers, studies the exact and approximate evaluation of general installation stock policies where both the retailers and the warehouse order in batches. Similar to most others, the author assumes an inventory system with N identical retailers. Other assumptions are stationary and independent poison demand, significant order costs, and constant lead time, etc. Such strict assumptions make the mode rigorous but the relevance of it to the practices is questionable.

A different inventory system is considered by Ha (1997). The author investigates the inventory rationing in a make-to-stock production system with several demand classes and lost sales. The simple queuing model and numerical analysis show that the optimal policy can be characterized by a sequence of monotone stock rationing levels. For each demand class, there exists a stock rationing level at or below which it is optimal to start rejecting the demand or this class in anticipating of future arrival of higher priority demands.

Rather than studying the inventory rationing, Cachon and Zipkin (1999) consider two games. In both, the supply chain stages independently choose base stock policies to minimize their costs. The games differ in how the firms track their inventory levels (in one, the firms are committed to tracking echelon inventory; in the other they track local inventory). The authors compare the policies chosen under this competitive regime to those selected to minimize total supply chain costs, i.e., the optimal solution. The analysis shows that the games nearly always have a unique Nash equilibrium, and it differs from the optimal solution, which results in that competition reduces efficiency.

Furthermore, the two games' equilibriums are different, so the tracking method influences strategic behavior. The authors show that the system optimal solution can be achieved as a Nash equilibrium using simple linear transfer payments.

Also use the game theory, Cachon (2001) continues the research and examines the supply chain inventory game, in which the firms manage inventory with reorder point policies; competition leads to a pure strategy Nash equilibrium in reorder points, which is a set of reorder points such that no player can lower its cost by deviating from the equilibrium, assuming the other players play their equilibrium strategies; there are no profitable unilateral deviations. The model investigates the competitive behavior in the supply chain inventory game and shows that Nash equilibria exist in reorder point policies. The author also suggests cooperation strategies available to the firms to help improve supply chain performance.

1.2.1.3.3 Capacity Models

The research in capacity for the supplier in a supply chain are very closely linked to and mostly embedded in the studies of other areas such as inventory and pricing, as well as contracting relationships. Only a few models specialize in discussing the particular topics for this type of relationship as how to find the optimal solutions for choosing and allocating capacity instead of through price mechanisms. The general assumptions for this type of models are that supplier has limited capacity and retailer orders exceed available capacity.

Cachon and Lariviere (1999) discuss the allocation game and find the Bayesian equilibrium to investigate the allocation mechanisms that are manipulable and induce retailers to misrepresent their needs (bullwhip effect) and those that are truth-inducing and lead to truthful reporting of retailer information. The authors also discuss the benefits for the retailers from restricting the supply chain to truth-inducing mechanisms. Finally they show how the chosen allocation mechanism influences how much capacity the supplier elects to build. In general, the authors focus is on the impact of the quality of information (truthfulness) and the mechanisms to induce them.

Linear programming models, incorporating the concept of planned lead times with multi-period capacity consumption are used by Spitter et al. (2005) to solve the general capacitated assembly problem. They propose two linear program formulations to find the optimal solution. One is that the capacity restrictions are incorporated using cumulative inequalities. The decision variables involved relate to quantities released at the start of a period and quantities processed in a period. The other one is that the cumulative inequalities are replaced by more detailed equalities. The decision variables involved in this formulation relate to quantities released in a particular period and processed in another (later) period.

1.2.1.3.4 Summary of SCM Model Research

The models presented above are established on very sound mathematical and logical analysis; however, most of them have strict assumptions. For example, in a two-echelon supply, as used in most models, the supplier and retailer are all risk-neutral.

Theoretically, this is not a problem, but in practice, it is very hard to find a risk-neutral

firm. Another example of restrictions is that demand is stochastic, which may not hold in practice. These questionable assumptions lead to a call for rigorous inter-disciplinary research to establish a framework that apply the theoretically strong models to a practical situation with the consideration of firms' behaviors that may violate some or all of the assumptions in those models one way or the other.

As part of the literature review for the dissertation, this section looks into the models for SCM in the major literature in the management and decision science journals and classifies them into three categories. The sole purpose of the categorization is to provide a clear and precise view of the SCM models literature, based on which we can find the gaps in the current research.

1.3 Information Sharing Research in SCM

Theoretical research on information sharing was seen in economics literature (Novshek and Sonnenschein 1982, Clarke 1983, and Vives 1988). Raith (1996) presents a general model which encompasses the existing economic models on information sharing as special cases. He shows the incentives for information sharing is highly related to the cost, i.e., firms exchange information when an improvement of the information about market conditions is valid only as far as information about own demand or cost is concerned, i.e., reduced. Some mixed analysis is presented and the benefit of information sharing is considered conditional.

Only in the middle of 1990's, researchers in operations management started to recognize that the value of information is a central issue when they studied the inventory

management in a supply chain and they substituted for one another (Chen 1998). For example, Hariharan and Zipkin (1995) show that advanced warnings from customers of their orders reduce inventory; and the additional information obtained through various marketing means will do the same (Milgrom and Roberts 1998). These examples mainly focus on the value of better communication between the customer and the supplier as described above. Similarly, it also creates value to improve the communication between supply-chain members. For example, Bourland et al. (1996) investigate the operational side of the tradeoff between communication and inventory and demonstrate that new technology such as EDI (Electronic Data Interchange) enables the supplier to obtain more accurate and timely information from the point of sale in the supply chain, and in turn the retailer could reduce its inventories also. Therefore, as Cachon and Fisher (2000) point out, it is not the question of whether information sharing improves supply chain performance, but how.

A comprehensive literature search shows the research of the value information sharing mainly focuses on two areas: inventory management and demand information. These two areas are not exclusive and in fact most literature emphasizing one have to take into account for the other. Some studies bring in the external factors such as retail competition (Padmanabhan and Png 1997, and Li 2002) and horizontal multimarket coordination (Arnand and Mendelson 1997). There are also researchers who focus on a particular supply chain, such as Wal-Mart, in which centralized demand information is shared (Chen, 1998), but this dissertation does not consider this type of serial inventory system.

Inventory models, as one of the most studied areas in supply chain management, are an important tool for evaluating the benefit of information sharing. Bourland et al. (1996) compare warehouse inventory costs with and without timely demand information and analyze the cost differential due to timely information sharing. They document that more timely shared demand information produces better results in terms of inventory reduction with more inventory benefits produced in a setting of higher demand variability (seemingly a positive way of mitigating the bull-whip effect). Cachon and Fisher (2000) confirm that “capturing and sharing real-time demand information is the key to improved supply chain performance”. These authors focus on sharing demand and inventory data, and they assert that the difference between supply chain costs under traditional (partial) and full information is one measure of the value of shared information.

In contrast to Lee et al. (2000) and Gavirneni et al. (1999), who assume there exists a perfectly reliable exogenous source of inventory and information sharing therefore has no impact on the retailer because its orders are always received in full after a fixed number of periods, Cachon and Fisher (2000) assume that the supplier is the only source of inventory and so information sharing may impact the retailers by changing the supplier’s order quantities or allocations.

However, some author doubt that information sharing has positive impact on the supply chain. For example, Graves (1999) studies a similar model to Cachon and Fisher (2000)’s, with the assumption that there is no outside inventory source, and concludes that information sharing provides no benefit to the supply chain.

Many research put emphasis on modeling the effects of demand. Lee et al. (1997) find that sharing retailer demand and inventory policy information reduces the supplier's demand variance (the bullwhip effect), which should benefit the supply chain, but they do not quantitatively measure this benefit. Chen et al. (2000) continue this study and quantify this effect for simple, two-echelon supply chains consisting of a single retailer and a single supplier. They include two factors that are commonly assumed to cause the bullwhip effect: demand forecasting and order lead times. Demand forecasting information is important both for the supplier as well as the retailer. Gurnani and Tang (1999) present a nested newsvendor model for determining the optimal order quantity with uncertain cost and demand forecast updating. Donohue (2000) suggests that including demand forecast updating in the supply contracts can coordinate the supplier and retailer to act in the best interest of the channel, which is to reduce the overall cost and increase the efficiency of the supply chain. Thonemann (2002) confirms that sharing forecasting information (advance demand information) can improve supply-chain performance, and he shows the benefits of such shared information to both the supplier and retailers, however it increases the bullwhip effect. Gallego and Ozer (2003) find the optimal replenishment policies for multi-echelon inventory problems under advance demand information.

Lee et al. (2000) suggest that when customer demands are highly correlated over time, the benefit of sharing demand information with the supplier is high. Raghunathan (2000) argues that Lee et al.'s results are based on an impractical assumption that supplier uses only the most recent order information to forecast the future orders. In reality, however, as the author indicates, the supplier may make the prediction of future demand basing on

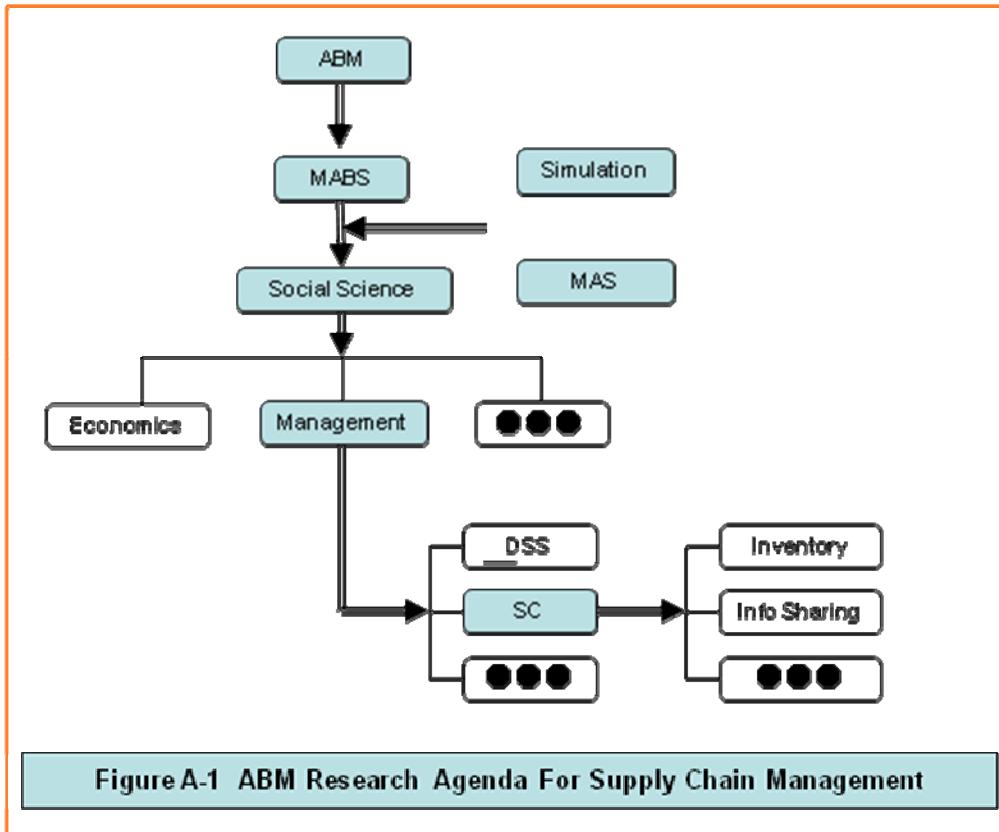
the archived historical order information. Nevertheless, the author's model and proofs are also problematic. For example, the author observes that the order quantities over time, ... $t-2, t-1, t$, are used to forecast the order quantities of next period ($t+1$), the benefit of information sharing approaches 0 as $t \rightarrow \infty$. The proof follows the assumption that the variance is constant – and this assumption may prove to be very unrealistic and perhaps dangerous when applied in the ever-changing competitive market place.

Chapter 2 Methodology

Instead of using the above mentioned traditional modeling methods, an emerging alternative to the supply chain research field is the new modeling approach using agents, the independent small computer programs may be used to represent the individual entities in the simulated world and interact with each other to reveal and help explain the phenomenon that are difficult or impossible to model using the traditional modeling methods. The underling scientific method is simulation. This thesis uses agent-based modeling as the methodology.

2.1 Agent Based Modeling

Figure A-1 shows the logic that links the different parts of ABM research in supply chain management. We start with ABM by combining the concept of multi-agent based simulation (MABS), which have been widely studied in the computer science and engineering fields; the research method of simulation, which is regarded one of the important ways of doing science; and then the agent-based modeling that is based on the theoretical application of MABS and simulation to social science. Then our center of attention turns to supply chain (SC), in particular to information sharing. We define four different modes of information sharing as the conceptual models and the experimental design will be based on these modes.



ABM was coined by Axelrod (2006) in his earlier papers. The purpose of ABM is suitable for the social science objective that is to “understand not only how individuals behave but also how the interaction of many individuals leads to large-scale outcomes. Understanding a political or economic system requires more than an understanding of the individuals that comprise the system. It also requires understanding how the individuals interact with each other, and how the results can be more than the sum of the parts.” By applying this concept to a supply chain, ABM is also very well suitable for modeling and understanding how individual members (organizations) behave, and how the interaction of these members can lead to improved supply chain performance. The system view of ABM method can easily be integrated into the conceptual model of this study.

ABM uses multi-agent based simulation as its essential research method. It is important to understand what it intends to study, and how simulation works. This section will review the literature on these research fields and discuss how ABM can be applied to supply chain research when evaluating the value of information sharing.

Simulations with intelligence were made possible by Artificial Intelligence (AI). AI, first coined by McCarthy (1959), is defined as a subfield of computer science that aims to construct agents that exhibit aspects of intelligent behavior and the notion of “agent” is thus central to AI (Wooldridge and Jennings, 1995). The theories of AI have been advancing along with the development of information technologies including computer hardware and software, computing theories, database, networking technologies, etc. One of the most important applications of AI, among others, is multi-agent based simulation. There has been an intense flowering of interest in the subject in many different fields such as the studies of social and ecological systems (Tsvetovat and Carley, 2004; Drougoul and Ferber, 1992; and Barr, 2004), transportation and geographies (Balmer, et al. 2004), marketing (Janssen and Jager, 2003; Lopez-Sanchez, et al., 2004), and many others. Several studies are also found in SCM model research.

2.1.1 Agents

Although agent research was started in the early 1980’s (for example, Rosenschein and Genesereth (1985) presented rational agents in an Artificial Intelligence conference), it became popular only around 1994 when several key papers appeared. The special issue of Communications of the ACM in the year published such papers as Maes’ now-classic paper on “Agents that reduce work and information overload” (Maes, 1994), Norman’s

conjectures on “How might people interact with software agents” (Norman, 1994), as well as the “Software agents” by Genesereth and Ketchpel (1994). In this period, agents are application programs in the software development (so-called agent-based software engineering) that aims to facilitate the creation of applications able to interoperate in the heterogeneous and dynamic software environment (Genesereth and Ketchpel, 1994).

The milestone of agent theory development was the “Intelligent Agents: Theory and Practice” authored by Wooldridge and Jennings (1995). The authors summarized and synthesized the research in the field. Their work provides an insight into what an agent is, how the notion of an agent can be formalized, how appropriate agent architectures can be designed and implemented, how agents can be programmed, and the types of applications for which agent-based solutions have been proposed. After thorough investigation, they note that although the term “agent” is widely used, there does not exist a universally accepted definition. There are in general two notions of agency, namely, weak and strong notions.

For the weak notion of agency, the authors define the term agent as a hardware or (more usually) software-based computer system that enjoys such properties as being autonomous, social, reactive, and pro-active. The strong notion of agency states that an agent is a computer system that, in addition to having the properties identified above, is either conceptualized or implemented using concepts that are more usually applied to human. Some of the added properties include mentalistic notions (e.g., knowledge, belief, intention, and obligation), emotions, mobility, veracity, and rationality.

There exist some other definitions. An example is Franklin and Graesser (1997) who, in their taxonomy, list many different definitions from various sources in academia and industry – including the one mentioned above – and propose the definition: “An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.”

The varieties of agent design can be found in the computer science literature like Earl et al. (2005), Kargl et al. (1999), Potok, et al. (2003), etc. The applications of agents, in addition to simulations, covers many areas in AI, namely, game playing, speech recognition, understanding natural language, computer vision, expert systems, and heuristic classification.

Certain points may need to be clarified as it comes to the understanding of an agent:

- There is no such thing called a general/all-purpose agent. Unlike many other concepts such as knowledge management for example, agent has been well defined though several versions exist. An agent, fundamentally, is the stand alone piece of software (program) that can accomplish a certain set of tasks. Many properties have been presented in Wooldridge and Jennings (1995).
- An agent is used to present an individual entity in the system. Such anti-team may be a human and animal and organization, and so forth. A system may be a financial market or a supply chain. An agent in the system is governed by behavioral rules. These rules are usually very simple but well defined. Some examples include what action to take when price of a stock goes below a

threshold by a stockholder agent, or whether a new order is triggered when a demand arrives from the customer by a retailer agent.

- The decisions of the agents are coded in an agent based model on the basis of the action of other agents. They may also be based on the results of its own previous decision or decisions, or based on the results of the previous decisions of other agents. We may use a decision tree or decision table to model the decision process of the entity that an agent tries to represent. Sometimes these decisions are also called the agent rules and behaviors. Agents rules are the logic that governs the behaviors of an agent which include a set of logical operators and “if...then” statements. Agent behaviors are model specific, which are a set of rules that tell an agent what to do under certain circumstances. In the implementation, rules and behaviors are the logic in the form of functions, the model specific functions that tell how the agents or interacting with or related to each other; what results may be generated by certain agent actions; how the results affect the behaviors of the agents.
- The design of an agent has its limitations. It's almost impossible to create an agent that can perform everything a human can, however we still far from knowing what and how the human brain works, because we don't know ourselves. Only when a human behavior is understood and can be described in detail clearly, it is possible modeled by an agent. Just as J. von Neumann indicated in a talk on computers given in Princeton in 1948, the only real limitations on making “machines which think” (in fact the programs that do the job) are our limitations in not knowing exactly what “thinking” consists of. “But of course, a mere

machine can't really think, can it?" von Newmann said, "You insist that there is something a machine cannot do. If you will tell me precisely what it is that a machine cannot do, then I can always make a machine which will do just that!"

Agents are never designed to "replace" human, and agents can never do that.

Nevertheless, the improvements in technology and artificial intelligence make us slowly but steadily closer in making agents "think" the way that human do. As knowledge advances, we are able to define better agents and their behaviors to help use reproduce more and more features of the real world, more and more accurately. No one knows whether there is some natural end to this process, or whether it will go on indefinitely.

2.1.2 Multi-Agent Based Simulation (MABS)

MABS was naturally proposed as the software environment in which agents are organized to interact with each other, which was also once named "federated system" (Genesereth and Ketchpel, 1994). The goal of MABS is to create a system that interconnects separately developed agents, thus enable the ensemble to function beyond the capabilities of any singular agent in the set-up (Nwana and Ndumu, 1999). In their classic work, the latter authors pose major challenges in agent research, and hence MABS research, such as information discovery problem, communication problem, ontology, legacy software integration problem, the reasoning and coordination problem, and the monitoring problem. To date, none of the above problems are close to being solved. Due to the lack of true understanding of what MABS is and what MABS can offer, when

applying this method, many researchers attempt to provide multi-agent solutions to the wrong problems, or even worse, many of them do not really “own” real MABS problems.

Simulation is regarded as the third way of doing science. Axelrod (2006) argues that it “starts with a rigorously specified set of assumptions regarding an actual or proposed system of interest” like deduction, but “it does not prove theorems with generality.” Instead, a simulation generates data that can be analyzed inductively. Unlike typical induction, however, the simulated data comes from a controlled computational experiments rather than direct measurement of the real world. While induction can be used to find patterns in data, and deduction can be used to find consequences of assumptions, simulation modeling can be used as an aid intuition. Simulation as a tool has been widely used in management (social science) field. It is particularly useful for evaluating supply chain reengineering efforts which may have the potential to impact performance in a big way. As explained by Swaminathan et al. (1998), simulation is “the only viable platform for detailed analysis for alternative solutions” in most organizations when making organizational reengineering decisions.

Multi-Agent-Based Simulation is the integration of multi-agent technology and simulation that creates an improved tool for doing science. Exploiting the growing capabilities of computers and advancement in artificial intelligence research that focuses primarily on multi-agent systems, MABS enrich the tool set for attacking the problems in social science that simply applying mathematical analytical modeling approaches may not be adequate because simulations using numerical analysis lack of capabilities of responding to changes in the testing environment.

It is important to note that when applying the concepts and tools from pure computer science and engineering fields that look at the world in an objective and physical way to social science (including management) that views the world with more subtleness and subjectivity, some variations and modifications may be necessary. For example, the agents, which are referred to as bundled data and behavioral methods representing an entity constituting in part a computational constructed world as described in Tesfatsion (2006), can range from active data-gathering decision-makers with sophisticated learning (adaptive) capabilities to passive world features with no cognitive functioning. Moreover, agents can be composed of other agents, thus permitting hierarchical constructions (agent clusters).

The objectives of MABS modeling take four forms: empirical, normative, heuristic, and methodological, as detailed in Axelrod (2006). The primary objective is empirical understanding which asks the question of “why have particular large-scale regularities evolved and persisted, even when there is little top-down control?” We seek causal explanations grounded in the repeated interactions of agents operating in realistically rendered worlds. Ideally, the agents should have the same flexibility of action in their worlds as their corresponding entities have in the real world. In particular, the cognitive agents should be free to behave in accordance with their own beliefs, preferences, institutions, and physical circumstances without the external imposition of equilibrium conditions. The key issue is whether particular types of observed global regularities can be reliably generated from particular types of agent-based worlds.

A second objective is normative understanding: How can agent-based models be used as laboratories for the discovery of good designs? We are interested in evaluating whether designs proposed from processes such as information sharing and member interactions will result in desirable supply chain performance over time. The general approach is akin to filling a bucket with water to determine if it leaks. An agent-based world is constructed that captures the salient aspects of a social systems such as a supply chain operating under the design. The world is then populated with privately motivated agents with learning capabilities and allowed to develop over time. The key issue is the extent to which the resulting world outcomes are efficient, fair, and orderly, despite attempts by agents to gain individual advantage through strategic behavior.

A third objective is heuristic: How can greater insight be attained about the fundamental causal mechanisms in social systems? That is, how can we understand fully a social system (e.g., a supply chain) through systematic examination of their potential dynamical behaviors under alternatively specified initial conditions? Even if the assumptions used to model a social system are simple, the consequences can be far from obvious if the system is composed of many interacting agents. The key issue is the extent to which coordination of supply chain activities emerges and persists as the members collectively learn how to make their production and pricing decisions.

A fourth objective is methodological advancement: How best to provide MABS modeling researchers with the methods and tools they need to undertake the rigorous study of social systems through controlled computational experiments, and to examine the compatibility of experimentally-generated theories with real-world data? To produce

compelling analyses, the researchers need to model the salient structural, institutional, and behavioral characteristics of social systems. They need to formulate interesting theoretical propositions about their models, evaluate the logical validity of these propositions by means of carefully crafted experimental designs, and condense the reported information from their experiments in a clear and compelling manner. Finally, they need to test their experimentally-generated theories against real-world data.

MABS has been applied to social science research. One of the examples is found in Janssen and Jager (2003). The authors use agents to simulate the consumer behaviors to systematically investigate the effects of different network structures. They model the utility of products, cognitive processing by consumers, and product characteristics; and put them in the social networks. Their experiment indicates that besides psychological needs and decision processes, the size and shape of the network involved in consumer decision making have an important influence on how the market organizes itself. A small proportion of consumers may have an exceptional influence on the consumptive behavior of others. Market dynamics is a self-organized property depending on the interaction between the agents' decision-making process (heuristics), the product characteristics, and the structure of interactions between agents. Lopez-Sanchez et al. (2004) show that MABS can be used for strategic decision-making. Their focus is on modeling the structure and behavior of the on-line music B2C (Business to Consumer) market and their constituents or stakeholders. Agents are used to model the product providers and consumers and several behaviors are specified in the simulation, such as those of buy best offers, buy cheapest offers, loyalty, follower, satisfaction, etc. Agents act autonomously according to their interests and interact with other agents inside the market environment.

An important application of MABS is in social systems as an effective tool for reasoning about human and group behavior. Tsvetovat and Carley (2004) believe that the effectiveness is enhanced when the algorithms lead the simulated agents to behave as humans behave, rather than doing what is optimal for the task. More effectiveness can be achieved when the model's inputs are real data and the generated outputs are comparable to actual data files in the real world. Their simulations show that MABS has the potential power for addressing real world issues and can enable a more realistic and extendable architecture for addressing policy issues in a manner comparable to human behavior.

Barr (2004) uses MABS to investigate the role of common knowledge in establishing conventional communication systems. His agents are “egocentric” in how they produce and interpreted signals; and use the signaling game theory to model the behaviors of agents. He shows that large populations of agents can establish and sustain conventional signaling systems without common knowledge. His study indicates that MABS can be used for not only pragmatic decision making as in marketing research, but also building theories that contribute to the development of a general understanding of social systems

Ecological systems are often used to study the process of emergence, which in turn can be applied to sociological systems. The behaviors in ant colonies have drawn interests from many researchers. Drogoul and Ferber (1992) used MABS as a tool for modeling such societies. Agents (ants) are defined with simple tasks, e.g., queen for laying eggs, spy ants for getting outside information, brood agents that represent the three steps of growth (the eggs, the larvae, and the cocoons), etc. Different agent behaviors are defined and a reactive simulation environment is created to simulate an ant nest. Their work provides a tool for modeling societies of simple agents.

MABS also appears to be useful in other areas such as transportation. Balmer et al. (2004) apply MABS to intelligent transportation systems (ITS) to predict travelers' behavior. Agents are used to simulate individual travelers directly and they will react to the system based on certain behavioral rules (e.g., a traveler will take the route that costs her the least time). The agents have the ability of learning but only have local knowledge about the system. Thus optimal solutions for individuals cannot always be computed and the simulation uses heuristics to find good but non-optimal solutions. Also, all agents learn simultaneously, which is called co-evolution. This implies that the traditional simulation theory (one agent is learning while all other agents have fixed strategies) is not a suitable strategy. The scope of the simulation is large which generates approximately five million trips. The authors show that the MABS is a perfect technology to evaluate ITS since it is possible to implement individual behavioral rules for each individual traveler and thus allowing for a differentiated and segmented response of the traveler population.

Other areas of application include, as Jennings et al. (1998) indicate, process control, telecommunications, air traffic control, information management, E-commerce, Games, Healthcare, etc.

2.1.3 Agent Structure Design

The structural design of the agents deals with the specifications of how the agent can be decomposed into the constructions of a set of component modules and how these modules should be made to interact based on certain behavioral rules. Wooldridge and Jennings (1995) thoroughly investigate the agent architectures available to the time and conclude that the classic deliberative, symbolic paradigm is the dominant approach in AI

and this trend goes on. However, this may give way to reactive architectures, which have two key properties: “situatedness” and “embodiment” (real intelligence is situated in the world, not in disembodied systems); and intelligence and emergence (intelligent behavior arises as a result of an agent’s interaction with its environment; also, intelligence is not an innate, isolated property) (Brooks, 1991a, 1991b).

An example of the reactive agent architecture is by Drogoul and Ferber (1992), who program each agent as an object and put these agents in a basic environment called space. Basic communication mechanism is implemented through stimulus-reaction scheme. An agent is seen as consisting of a set of behaviors (tasks). Agents in the simulation exhibit flexible mechanisms of behavior selection and they can take former experiences of interactions with their environment into account when choosing their future behavior. The activation of a behavior also integrates non-environmental conditions such as motivations.

Nwana (1996) studies the software agents – agents in general are the same as those we have discussed in the preceding text – and classified the agent architectures into six types: collaborative (deliberative, symbolic), interface, mobile, internet, reactive, and hybrid. Each type has its own strengths and deficiencies. The architecture that is proposed in our study tries to maximize the strengths and minimizes the deficiencies of these architectures. To this end the hybrid approach appears to be more appropriate.

2.1.4 Agent Based Modeling in SCM

Other than mathematically analyze the model using numerical test, empirical study and simulation are also important methodologies in SCM research. Simulation has in the recent years been adopted in management and social science research and has been regarded as the third way of doing science (Axelrod, 2005). Agent-based simulation which encompasses the simulation technique with the advances in artificial intelligence is characterized by the existence of many agents who interact with each other with little or no central direction, nor human interference. This type of simulation only appears in the academic literature in last decade.

Only a small number of research papers could be found that adopt agent-based simulation as a methodology in supply chain management literature. One of the important works is by Swaminathan et al. (1998), who find that supply chain reengineering (improvement) is critical to the companies exposed to global economy and striving to meet customer expectations regarding cost and service. As the reengineering process is a strategic move, it requires detailed risk analysis. Since quantitative analysis provide insights into current trends but not prescriptive, simulation becomes the only viable platform for detailed analysis for alternative solutions. The authors design a multi-agent framework in which different agents are specified and different control mechanisms are defined. The purpose of this framework is to provide members in the supply chain a customizable decision support tool that can help managers to understand the costs, benefits, and risks associated with various alternatives.

Garcia-Flores and Wang (2002) propose a multi-agent system to model the three flows (money, information, and material) in a chemical supply chain. Their design is very

specific for use in the chemical supply chain. The main emphasis of this design is on the processes of paints and coatings production in a plant in addition to certain simplified relationships up- and down-stream. The major addition in the agent design to those by Swaminathan et al. (1998) is that they use one of the common agent communication languages (ACL) to specify in detail the mechanism of communications between agents.

A shift of interests is found to move from traditional to net-enabled supply chain (using the Internet or other types of electronic telecommunication media) research. Some researchers name this as e-supply chain (Poirier and Bauer, 2000) or e-chain (Singh et al., 2005). The later authors present an agent-enabled architecture that exhibits information transparency (the availability of information through out the supply chain in an unambiguously interpretable format) and enable enhanced interaction among participants in an e-chain. The focus of the system is largely on the supplier-buyer relationships and processes. The authors describe the process of how a discovery agent matches the buyer and supplier based on the buyer's demand, supplier capacity and reputation, etc.; how transaction is promoted by a transaction agent; and the control mechanism facilitated by the monitoring agent. This platform can be applied to multi-buyer multi-supplier supply chain environment

Some researchers use agent-based simulation to study the traditional SCM topics in a new way. Such example is Lin et al. (2005), who examine the effects of trust mechanisms on supply-chain performance in an e-commerce environment. Their research framework has particular focus of exploring the trust mechanism, based on the integrated view of Mayer et al. (1995), in facilitating information flows and transactions

within a supply chain. The authors have not described how agents and agent functions are defined and they implement their study on the Swarm platform, one of the agent-based simulation software packages available.

So the question is: “What makes agent-based modeling a good choice in SCM research?”

Mathematical models in the traditional SCM research use mathematical analysis. Due to the difficulties of obtaining empirical data for validation, simulation techniques are used. One of the popular options, among many stochastic procedures, is to use Monte Carlo simulation, which is a great technique that relies on repeated computation and random or pseudo-random number and is used when it is infeasible or impossible to compute an exact result with a deterministic algorithm. Although Monte Carlo method has wide uses in applied sciences and mathematics, as well as in some business applications such as the calculation of risk in business, evaluation of investment projects, etc., and shares many benefits of agent-based modeling (ABM), it does not work as well in many other areas.

ABM is a totally different and new approach that models a system as a collection of autonomous decision-making entities called agents. Therefore, ABM has become increasingly popular as a modeling approach in the social sciences because it enables one to build models where individual entities and their interactions are directly represented. The modeler has absolute and total control, if desired, of each single agent, and as in a physics or chemical experiment, he/she also has complete control of simulation environment. In comparison with variable-based approaches using structural equations, or system-based approaches using differential equations, agent-based simulation offers the possibility of modeling individual heterogeneity, representing explicitly agents’

decision rules, and situating agents in a geographical or another type of space. It allows modelers to represent in a natural way multiple scales of analysis, the emergence of structures at the macro or societal level from individual action, and various kinds of adaptation and learning, none of which is easy to do with other modeling approaches.

There are many characteristics that make ABM a powerful modeling technique, which may be first seen through the definition of it. Though many different descriptions of ABM may be found, we believe the following is a good working definition:

“...Agent-based modeling is a computational method that enables a researcher to create, analyze, and experiment with models composed of agents that interact within an environment.” (Gilbert, 2007)

This definition has several terms that worth further exploration. Using an example may get us better idea. A conceptual implementation of Sterman (1989) Beer Game supply chain will serve the purpose.

First, ABM is a form of *computational* social science. That is, it involves building models using computer programs. Similar to any other programs, the computational model takes one or more inputs (just as the independent variables if use regression model), and some outputs (similar to the dependent variables). The program itself presents the processes that transfer the inputs to outputs. In Beer Game, the program includes all the participants, i.e., the agents; the inputs are the orders and shipments passing through the chain; and the outputs are the costs to evaluate the significance of

individual decisions, while these individual decisions (or decision rules) plus the embedded system variables are the processes.

Second, in physics, chemistry and some parts of biology, or other so-called hard science, the main and standard method of research in proving the theory or exploring the unknown is to conduct *experiments*, the same process can be repetitively tested many times.

Conducting experiments in most of the social science is impossible or undesirable because the subjects are human beings. Even an experiment is desired and proceeded, the tester does not have control on the heterogeneity of different subjects as one does in, for example, a physics experiment, where one can apply some treatment to an isolated system by controlling the quality, quantity of the subjects, the temperature and humidity of the environment, as well as the duration that can be as exact as to millisecond. Such isolation in the social systems is generally impossible, if not unethical.

ABM simulation provides such a tool for researchers of social science to have the luxury of doing the experiments as those of hard science. The experiments may be repeated many times, using a range of parameters or allowing some factors to vary randomly. The human subjects are replaced by the computer programs, i.e., the agents. The environment is virtually under full control of the researchers as it is programmed in the software. The behaviors of the agents are also under control, though to date many human behaviors are still not replicable due to the constraints of limited understanding of ourselves. We do not understand why, for example, among countless many others, one reacts to one thing that is different from another one's reaction. Thus it's impossible to program these unknown behaviors. It's the hope that the progress in the research of human psychology,

artificial intelligence, etc. will make us more understand our behaviors and the ABM will be more precise in modeling the phenomena that involve human interactions.

However, although each individual is different from each other, in some cases there may be finite categories of possible reactions. For example, in the stock market when investors hear the news of higher unemployment rate in the previous quarter, some may be pessimistic and believe that the economy will experience a recession and the stock market may crash soon. Thus the reaction is to “sell”. On the other hand, some other investors may believe this is temporary and optimistically believe that economy is in good shape and the unemployment rate will soon drop to the normal level and thus they will buy stocks at the perceived low price. There might be other reactions such as wait-and-see, etc. This makes it possible for the researchers to categorize their reactions at the high (aggregate) level, and program the agents to see how the change of one factor may affect the results of the experiment, as one does in physics (hard science).

Getting back to the Beer Game example, the game has been designed simple enough and has been experimented on many groups of individuals of different kinds. However, by reading the results of Sterman (1989), we find that because of the lack of control of individual participants in addition to human errors, the useful samples after four years of work is only 11 out of 48 (i.e., 44 out of 192 subjects). This causes great waste of time and money. With the help of ABM, we may set up an experiment and repeat it as much as desired and change any parameters.

The third term is the *Agents*. Agent based models consist of agents that interact within an environment such as the supply chain. Agents are either separate computer programs or,

more commonly, distinct parts of a program that are used to present social actors – individual people (customers, managers, employees, etc.), organizations such as firms (suppliers, retailers, etc.), or bodies such as nations or states. They are programmed to react to the computational environment in which they are located, where this environment is a model of a real environment in which the social actors operate.

As will be seen in our model implementation, a crucial feature of agent-based models is that the agents can interact, that is, they can pass information or messages (such as the order quantity, demand pattern, i.e., distribution and parameters) to each other and act on basis of what they learned from these messages. The messages may represent spoken dialogue between people in some cases, or more indirect means of information flow, such as the observation of another agent all the detection of the effects of another agent's actions. The possibility of modeling such agent-to-agent interactions is the main way in which agent-based modeling differs from other types of computational models.

The last term is the *environment*. The environment is the virtual world in which the agents act, which is as important as the agents themselves. The environment may be in different forms or of different types. It may be an entirely neutral medium with little or no effect on the agents, or in other models, the environments may be as carefully crafted as the agents themselves. Commonly, environments represent geographical spaces, for example, eating models concerning residential segmentation, where the environment simulates some of the physical features of a city, and in models of international relations, where the environment maps states and nations. This type of environment is often seen in the research of GIS (Geographical Information Systems) field. Models in which

the environment presents a geographical space are sometimes called spatial explicit. In other models, the environment could be a space, but one represents not geography but some other features. For example, scientists can be modeled in knowledge space. This type of agent-based modeling may be used to create expert systems. In these spatial models, the agents have coordinates to indicate their location. Another option is to have no representation at all but to link agents together into a network in which the only indication of an agent's relationship to other agents is the list of agents to which it is connected by network links.

Some prevailing agent-based modeling environments include Repast (Recursive Porous Agent Simulation Toolkit), Swarm, Ascape, all of which support pure java programs and Repast supports Python and Microsoft .Net, and Netlogo, which is a cross-platform multi-agent programmable modeling environment and support click and drop model creation. In our model implementation, we select Repast and the agents are coded using Java programming language.

Chapter 3 The Model

The original intension of the author was to establish a general agent based model structure (or framework) that can be customized to solve a wide range of complex system problems by following the infamous “Make everything as simple as possible, but not simpler.” by Albert Einstein. However, the other part of the slogan that states “but not simpler” proves true as it was later found that agent based modeling has its limitations, one of which is that, common to all modeling techniques, as Bonabeau (2002) puts it, “a model has to serve a purpose”. A general purpose model is not sufficient to define and deliver such a purpose. The model has to be domain-specific and with just the right amount of details to serve its purpose. How much is “right” remains an art more than a science.

3.1 Conceptual Model

Most papers in the literature that we have examined show that the models investigating the value of information sharing look into a simple 2-echelon supply chain structure. Although we may model the entire supply chain from raw material to production to retailers and then to the final customers, we for simplicity, will look into the commonly used 2-echelon approach. When the simplified model is established, it will be relatively easy to expand it to any levels of the supply chain of different types of structure.

Also, it is well known that there are many types of flows within a supply chain, such as, materials, orders, money, personal, capital equipment, and information, as indicated by Forrester (1959), and information sharing may have different levels of impact on these

flows to certain degrees. Again for simplicity and without losing generality, we assume that this type of impact will hold constant throughout the simulation and the focus of the study will lie on those related to the supplier in terms of operational cost such as inventory and returns; and those related to the retailers such as backlog and inventory.

It is necessary to point out that most literature consider the cost savings at the supplier side, and a few others included the retailers when they model the value of information sharing. Only some consider both sides. I find it inadequate to consider either side alone because the interactions between the supplier and retailers may have effects on how the value is maximized. However, it is understandable that these types of interactions may be very difficult to capture using the mathematical modeling approach because there is not a generic pattern for the interactions for various types of supply chain, and the mathematical models cannot adapt to the dynamics in the process of information exchange. Multi-agent approach may be an appropriate method to overcome these shortcomings.

There exist a small number of research results that start to consider the agent based modeling approach. However, these papers, although they claim that they use multi-agents in the simulation, use the agents in a very simple way, which simply perform the roles of gate-keepers type of controlling units in the simulation. For example, in Swaminathan et al. (1998), the agents may take the inputs from the downstream and pass them to the upstream. They may look up the current inventory level and decide whether or not to order more. The agents communicate in a virtual hierarchical structure and

perform very simple tasks based on the simple yes/no logic. They do not preserve their own local memory, thus possess no learning capacities.

The agent based model we design attempts to provide a simple view of a complex system, i.e., we consider the supply chain studied as a complex system, in which members are not isolated and they interact with each other. Members are represented by agents, the computerized representation of the entities in a supply chain who perform specific tasks. The design and modeling of the agents, as well as interactions between / among the agents are described.

For the purpose of this dissertation, that is to develop a multi-agent system that simulates the information sharing process in a supply chain so that the benefits of information sharing can be evaluated more precisely and realistically, I find there are several important components in the system and there exist connections between them.

3.1.1 Components and Connections

As was mentioned earlier, from a system view point, everything in a system is connected to each other in a certain way. The components in the system are briefly discussed below, and the connections between these components follow. More details will be given in the later part of this chapter when explanations of the model are described.

Information: This study lays its focus on information sharing within a supply chain. Information is the key as it is the center of the diagram. Information flow can be viewed as a two-way process. Supplier and retailers share the information both ways, the type of information and degree (in our experimental design as

shown in the next chapter, the degree is defined as three levels: high, medium, and low, respectively) of such sharing may vary. We categorized them into four modes as further discussed later (Section 3.1.2).

- o Impact of Information Sharing

Many papers study the impact of information sharing based on the values or costs. This research follows the same route. Multi-agent simulation is used to identify the total cost saving that information sharing brings to the supply chain.

- o Supplier / Retailers

These are two commonly used terms in supply chain literature. No additional meaning is added to them in this study.

- o Relationships / Interactions

Relationships are defined as the results of the interactions between supplier and retailers (vertical interactions), as well as between retailers (horizontal interactions). Current literature mostly examines the vertical information sharing for a good reason: the traditional definition of a supply chain is the collection of upstream and downstream type of flows (Forrester, 1959), thus it is natural to investigate the effect of information sharing in the same fashion. Since nothing is isolated and everything is interrelated to each other in certain way, it becomes natural to ask the question, “How will the interactions between the retailers

impact the information sharing in the supply chain?" A few researchers have partially addressed this problem, but no overall picture has been drawn.

- o Agents

Software and hardware components that can accomplish certain tasks with certain characteristics that traditional software and hardware do not have are called agents. A discussion of the definition of an intelligent agent can be found in Wooldridge and Jennings (1995) and a discussion in more detail is given in the previous chapter.

3.1.2 Modeling Structure

The information flow may be modeled differently in different modes (or types). As most literature suggests, in the first three modes presented below, the information flows both directions in vertical directions with focus on evaluating the benefit of information sharing from the supplier's perspective. The information sharing in these three modes has direct effect on the performance of a supply chain (Li 2002). Although it is well documented that information indeed flows both ways between the supplier and retailers, most authors (some exceptions such as Cachon and Fisher 2000 who believe that information sharing may have impact on the retailers; and Gurnani and Tang 1999 who study the impact of demand forecasting on retailer's optimal ordering policy) when they evaluate the values of information sharing pay most attention to the flows from downstream (retailers) and ignore the other half from the upstream (supplier). Another aspect that the current literature is missing is that members in a system are not isolated

from each other and there exist interactions. These interactions may have direct or indirect impact on the information sharing studied.

This dissertation extends the views of information sharing from purely half vertical (one way) to full vertical (two ways) with horizontal interactions. External factors are also considered. I may use the term indirect effect (Li, 2002) to address the impact of information sharing in this new mode. I hereby summarize four such modes as described below.

Mode 1: This mode is called “No Information Sharing” by following the model description of existing literature. “No Information” in fact is not a very accurate term. As generally understood, in a system, there is always certain information exchange to maintain the links between components. This is the lowest level at which the least information is shared. Only the necessary information flows between the supplier and retailers so that orders are placed and delivered. Lee et al. (2000), Raghunathan (2000), and Gavirneni et al. (1999) model this as “no information sharing” as the base case for comparison. This is similarly considered as “traditional information sharing” (Cachon and Fisher, 2000) that supplier only observes the retailers’ orders. The information from the supplier to retailers is minimal and is not considered in the research found. See Figure 1 for the graphic representation of this mode. It is noted that this mode is close to the “closed system” notion in the system theory (Bertalanffy, 1968). However, since everything is connected one way or the other, it is not possible to find a real closed system, thus no “no information sharing” mode exists. The presentation of this mode is solely for comparison

purpose. The mathematical analysis such as that in Lee et al. (2000) shall be sufficient to serve for this purpose.

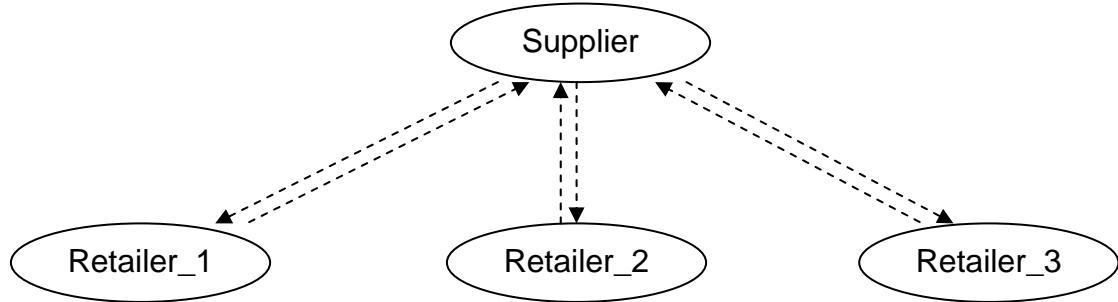


Figure 1 – Mode 1: No Information Sharing

Mode 2: Thonemann (2002) models this as aggregate information sharing, at which retailers share with supplier information about whether they will place an order of some product in the next time period, but do not share information about which product they will order and which of several potential suppliers will receive the order. Gavirneni et al. (1999) name it as partial information sharing, where supplier knows the (s, S) , i.e., order up-to, policies used by the retailers as well. It is assumed the future demands for the supplier are independent of past demand. This assumption may be relaxed using processes such as autoregressive, AR(1) as described in Lee et al. (2000), or ARMA (autoregressive with moving average) described in Graves (1999) and Thompson (1975). Figure 3 shows this mode. Note that the diagram differs from that of mode 1.

- The type of arrows. This shows that more information is shared in this mode. Such information may be as the details of the s-S policy from the retailers (solid lines), or the inventory levels and capacity information from the supplier (dotted lines).
- Demand information flows into the retailers. However, this information does not go further upstream because in this mode, the supplier does not get this information from the retailers, while some authors such as Gavirneni et al. (1999) include the information of end-item demand distribution into this mode. For this reason, the lines related to demand information are all dotted.
- Demand can be stationary or non-stationary, depending on how the model is specified. Many literature studies the stationary demand (Thonemann 2000, Cachon and Fisher, 2000, and Gallego and Ozer 2003), others investigate the non-stationary demand (Lee et al. 2000, Hu 2003, and Graves 1999).

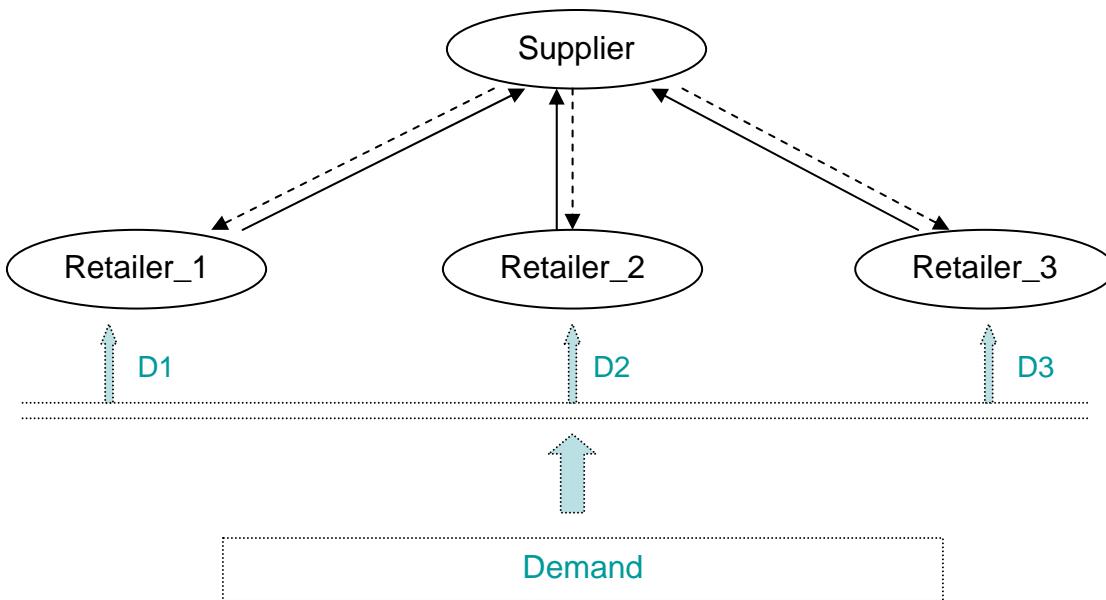


Figure 2 – Mode 2: Partial Information Sharing

Mode 3: full information sharing between supplier and retailers, as modeled by Lee et al. (2000) and Raghunathan (2000) where the supplier has full information about the state of the retailer (Gavirneni et al. 1999), or the “detailed” information sharing as described by Thonemann (2002) where supplier gets the detailed information of the orders to be placed. Again, the current literature focuses on the benefits of information sharing from the supplier side and few consider the retailer side. For example, as Lee et al. (2000) suggest, information sharing by the retailers provides significant inventory reduction and cost savings to the supplier, and the retailers can, on the basis of the potential benefits of such savings to the supplier, negotiate arrangements with the supplier to reduce their own overhead and processing costs.

As the phrase “full information” suggests, the supplier and retailers have established communication channel and trust so that information can flow both way flawlessly. It is assumed that the information flow shall be real time and decisions made without delay. (However due to many reasons such as limitations in information technology, there exists information delay and thus decision delay. This delay will add costs to the supply chain due to, for example, delayed delivery, and the existing literature fails to notice and inform this. It is not the goal of this study to address this problem.) The differences between this mode and the other two are as follows.

- Solid arrows, which represent the assumed solid relationships between the parties. Solid here means the trust, goal congruent, etc., which are the dimensions that can

be found in a large number of research papers and are out of the scope of this study.

- Demand plays an important role in this mode. Demand information is shared between supplier and retailers and it does have impact on the supply chain. Individual demands are assumed to be identical as in majority of the literature, i.e., $D_1 = D_2 = D_3$.

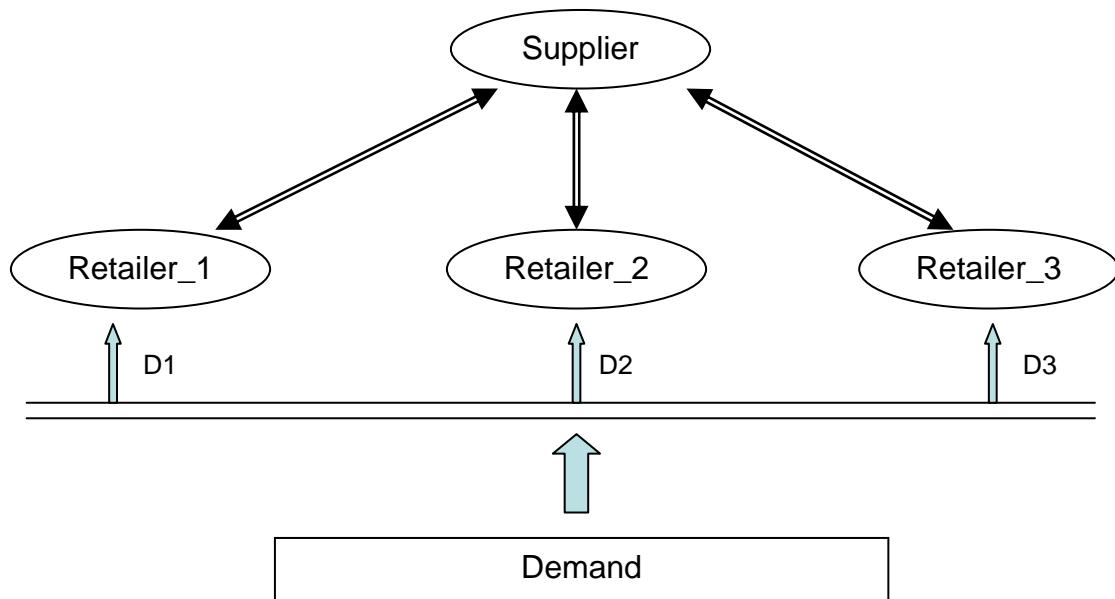


Figure 3 – Mode 3: Full Information Sharing

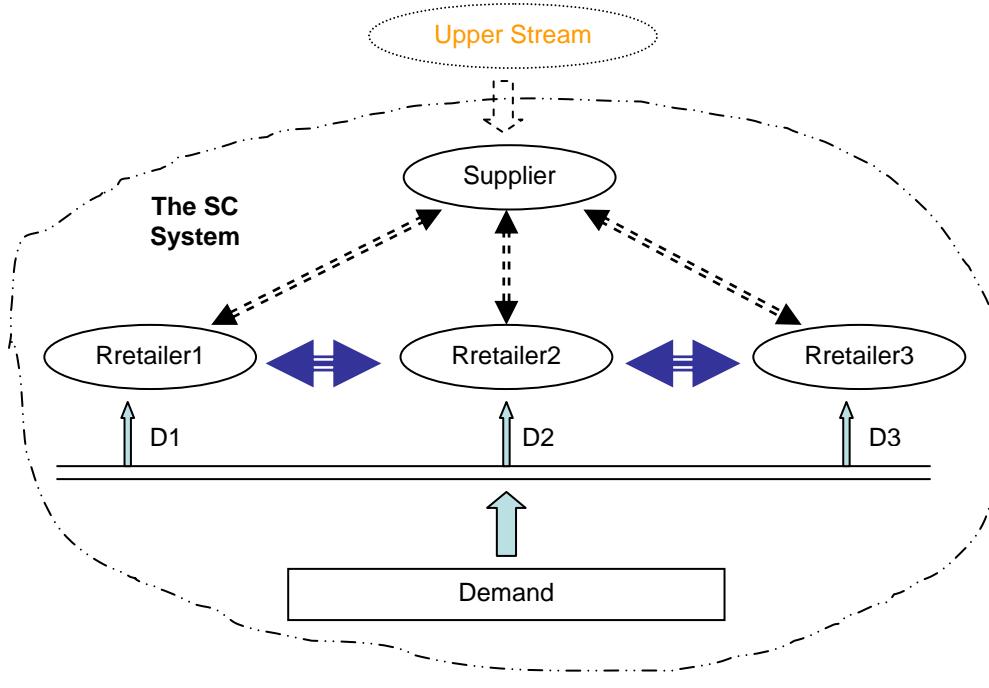
Mode 4: comprehensive information sharing. This level of information sharing has only been partially modeled in the existing literature. For instance, Li (2002) considers competitions among retailers when they model the information sharing in a supply chain.

Different from the other three modes, which consider only the vertical information flows, the comprehensive information sharing mode (Figure 4) factors in the horizontal

interactions as well as the external forces. This mode applies the systems thinking¹ approach: it considers the supply chain as a complete open system. Based on this approach, each part (the supplier, retailers, external demand, etc.) are all considered as the components of the supply chain system and they interact with each other. The difference is obvious: it is not the traditional vertical hierarchy any more, rather, it may be viewed as a networked system. The horizontal interactions appear to be a new area of research. The information, along with other elements such as material/products, flows among different parts that compose the system. Figure 4 indicates these relationships.

Multi-agent simulation is applied to evaluate the impact of comprehensive information sharing. The approach has been shown to be effective for investigating information sharing in supply chains in the literature (Lau et al. 2004, Swaminathan et al. 1998).

¹ An extensive reference list may be found at the web page maintained by W. Huitt at:
<http://chiron.valdosta.edu/whuitt/materials/sysphil.html>



**Figure 4 – Mode 4: Comprehensive Information Sharing
The System View**

As shown in the figure above, the retailers are no longer information isolated from each other. The interactions among them give them the opportunity to exchange information at different levels, depending on the relationships established between them.

The information sharing between retailers is considered as a form of interaction. Interactions are the fundamental research area in the organizational research and may have many different forms (such as, cooperation, negotiation, competition, trust, etc.) They may be between individuals or groups of human beings or organizations. However, our study focuses on the consequences of organizational interactions, i.e., the relationships established and enhanced due to the interactions, but not the forms of different interactions.

The buyer-seller relationships between the seller (supplier) and buyer (retailers), also known as the channel-relationships, as well as strategic alliances literature mainly focus on the issues in a vertical structure. This vertical structure view looks at the system in an incomplete way. It overlooks the fact that the nature of the activities at each level of the vertical structure may have impact on the dynamics of the whole system: organizations at the same level compete for resources or customers of the lower level, and seek possible interorganizational cooperation to, for example, minimize the transaction costs. As Cravens and Shipp (1993) put it, independent organizations collaborate to increase the competitive advantage of each organization, and these cooperative relationships are of escalating importance. Rather than competing independently against other organizations in the market-place, organizations are forming cooperative interorganizational relationships to take advantage of dissimilar strengths of different organizations (Borys and Jemison, 1989). For example, as the authors indicated, over time, “many firms develop more sophisticated supplier arrangements in which reciprocal interdependence becomes strategically important, purpose broadens, and stability mechanisms become institutionally based as well as contractual.”.

The relationships between these organizations as the result of such interactions of cooperation are the focus of the thesis. In particular, we examine the impact of information sharing (the cooperation side) on the costs of the buyers (retailers), and in turn on those of the buyer (supplier). The theoretical support for such examination may be found in Barringer and Harrison (2000) which tries to investigate the theoretical paradigms that explain inter-organizational relationship formation. The authors summarize the six widely used paradigms, including transaction costs, resource

dependence, strategic choices, stakeholder theory, learning theory, and institutional theory, as shown in Table X1. A brief discussion of the relevance of these paradigms to this thesis follows.

Perhaps Transaction Costs theory, the first paradigm of Barringer and Harrison (2000), is the best fit for the inter-organizational relationships due to information sharing. Using the simple problem settings (or “domain” as we use in this thesis) for examining the impact of information sharing on inter-organizational relationships of a supply chain, following the theoretical conclusions of many previous research works published, we find one of the ultimate motivations for the organizations (or firms) to share information is to lower the transaction costs. The previous findings mostly have pointed out that information sharing has positive impact on an organization’s transaction costs, thus an improved (closer) relationship between the buyer and retailer. It is natural to apply this theoretical paradigm to state that other than the impact on the vertical structure of the supply chain, information sharing between retailers at the horizontal level also plays a role in reducing the transaction costs of organizations.

Clearly, Information sharing can as well be rightfully categorized into the Resource Dependence paradigm. This theory is rooted in an open system framework, which argues that organizations must engage in exchanges with their environment to obtain resources (Scott, 1987). This theory is different from the resource-based view (RBV) (Mahoney and Pandian, 1992, Barney, 1991) which states that rare and difficult to imitate internal firm resources are key to the firm’s acquisition and maintenance of sustainable, competitive advantage, thus RBV’s focus is more internal, although some of the

important internal sources may be obtained from external sources. The resource dependence theory states, otherwise, that the resources must be obtained from external sources for an organization to survive or prosper.

Information is well regarded as a resource (Wade and Hulland, 2004) and its value (Howard, 1966) or “usefulness” may be evaluated based on certain determinants (or factors, namely, flexibility, payoff function, degree of uncertainty, and the nature of information system) (Hilton, 1981). Some information may be obtained internally, which complies with the definition of RBV, while other information may only be acquired externally from other sources or organizations through exchange or sharing. The latter type of information fits the characteristics of resource dependence paradigm.

Information sharing may not be applied in other paradigms in forming the inter-organizational relationships. For example, the Strategic Choice theory² studies the factors that provide opportunities for firms to increase in competitiveness or market power. The profit and growth are typically the major firm objectives that drive strategic behavior. The strategic alliances may be a better application of such paradigm.

3.2 General consideration of model design architecture

As we mentioned in the beginning of the chapter, we need to specify the domain or purpose of our agent based model. Simply put, the purpose of the model is to examine the interactions due to information sharing between the retailers in a simple 2 level

² The strategic Choice Theory was developed when industrial relations in the U.S. were changing rapidly and the then popular conventional theories, such as the Dunlop's systems model, were overly static and could not accommodate interactions. The theory was first found published by Kochan, T. A., et al. (1984). "Strategic Choice and Industrial Relations Theory," *Industrial Relations*, 23(1), 16-39.

supply chain. However, a full scale application with multiple suppliers and retailers may be implemented based on the results of this essay. That said, our implementation takes a small representative portion of a supply chain, as most current SCM studies do, and supposes a supply chain with one supplier and three retailers with one product. More retailers and more products may be added in the future research based on the similar principles as described below to examine impact of the multiple interactions among retailer agents to the system. Instead of assuming that all the retailers are identical as seen in most SCM literature, we allow each retailer to have its own decision process using a probability procedure (as later shown in the implementation, this procedure is a decision logic using the input from the simulation and deciding certain actions to be taken).

3.2.1 Environment

All agents are put into an environment (or space) where they can interact with others based on certain behavioral rules and design objectives. This environment provides the channel of communication between agents and may also include nonreactive objects (for example the input such as the demand. To keep the implementation simple and without losing the generality, the demand, as in the approaches found in previous research (including all that are cited in this essay), is regarded as independent and not reacting to the actions of the retailers and/or the suppliers.). It is convenient to route all communication between agents through the environment, not only because this is the natural way to do it, corresponding to the role of the environment in human affairs, but also because it makes monitoring the agents easier in implementation. It also means that

messages from one agent to another can be temporarily stored in the environment at runtime, reducing the likelihood that the results of the simulation will depend on the accidents of the order in which agent code is executed. This ensures that the messages are delivered to the recipients in the order as expected. And in fact this brings up the necessary and important component in the simulation: the time.

The only time unit (steady and uniform) is one period and each agent does one set of tasks at each period. The simulation proceeds as though orchestrated by a clock, tick by tick, while each period may include multiple ticks. At each period, all the agents are given a turn. Thus, time is modeled in discrete time steps.

To make it simpler without affecting the results of the simulation, we assume that agents all work full time and there are no breaks such as weekends and holidays. The simulation software environment is fully synchronized. This is very important because without synchronicity, the simulation may not run reliably.

3.2.2 Agent objectives and behaviors

The objectives of an agent may vary depending on the types of agents to be modeled. For example, the objective of a supplier is to minimize its inventory cost and deliver the products to retailers on time. However, the objective of a retailer may be different in a sense that it seeks not to lose sales by ordering excessive inventory. A real life example may be the currently hard to find Wii game console designed and manufactured by Japanese company Nintendo. The supplier has limited production capacity while the demand on the market is high. The supplier needs to carefully evaluate the true market

demand before it dispatch the products. The reason is simply for retailers, they have the incentive to exaggerate the demand and get a bigger shipment. In the most recent trip to the Target Store, I found the store had a big inventory of the console, while I found online the buyers complain about the delivery delays for the console.

The agent behaviors that we will model are in the form of rules. Based on these rules, an agent knows when to initiate an action and how to react when a request (an inquiry or a response) is received from another agent.

3.2.2.1 Agent objectives

The supplier and retailers may have very different design objectives due to their unique positions in the business relationship. The supplier holds the product, but it needs information from the retailers to know what the future orders are. Retailers order product from the supplier. However the shipment from the supplier may need a prolonged period to arrive. So the retailers may coordinate with each other their stock levels to reduce the chance of losing business due to insufficient stocks or to lower the inventory cost due to excessive product in stock. So the ultimate design objectives for these agents are:

- Supplier:
 - o Reduce the inventory cost (including overstock and backlog) by ordering carefully calculated quantity of inventory. This requires good communications with the retailers to obtain their (s, S) parameters.
 - o Deliver the product to retailers without delay
- Retailer:

- Reduce the inventory cost (including overstock and backlog). This can be achieved by ordering the inventory according to the forecast.
- When the stock reduces to a level too low, it should seek to get immediate help from other retailers at a premium to increase the stock.
- When the stock is too high, it may ship some of its stock to other retailers to reduce the stock.

(In the future research, we may add more dynamics to the system: when there is not sufficient inventory for all retailers, the supplier has the negotiation power to decide the price and quantity for each retailer. However, this situation changes when the demand is lower than expected and there are excessive inventory of product. The retailers may have more power in negotiating the price and quantities of the product to buy.)

The differences in the design objectives between the supplier and retailers determine the discretions in the interactions they engage in with each other.

3.2.2.2 Agent behaviors

- Supplier: as defined, a supplier provides products for the retailers to sell. We assume that these products can be made by the supplier without purchase of any components and, as a result, the supply chain under consideration ends there.
Also we assume that the supplier has unlimited capacity
 - The supplier receives orders from the retailers when their inventory reaches reorder level

- The supplier needs a fixed amount of time T_{mk} to make the product (any units) (future research may add the production capacity constraints to the supplier). The value of T_{mk} may be customized by a user at runtime
 - The supplier needs a fixed amount of time T_{tr} to transport the orders to retailers. The value of T_{tr} may be customized by a user at runtime
 - The supplier may ask the retailer to provide their (s, S) information and they may get this information in a full information sharing case. The supplier uses this information together with the demand information received from the retailers, by using the approach described in Lee et al. (2000), prepare the demand forecast and order the inventory for next period.
 - The supplier has its own demand forecasting model if the retailers (s, S) information is not available and the supplier will calculate its own (q, Q) values in each 30 day period based either on the (s, S) values from the retailers, or when that information is not available, on the forecast values it obtains. New production will start when the inventory is below q and stop when it reaches Q .
 - When inventory available is not sufficient to satisfy the orders placed by multiple retailers, the supplier needs to decide the quantity each retailer may receive. A simple supplier decision process is used.
- Retailer:

- Each retailer every day receives demand at the beginning of the day and it checks if it has sufficient stock. If it does, it will reduce the stock level by subtracting the demand amount.
- At the end of the day, it will check if the stock level is below a preset reorder-level value “ s ”, if not, it will do nothing, otherwise it will order more stock to order-up-to level “ S ” from the supplier.
- In addition to this reordering process, the retailer needs to have a long term forecasting of the average demand in the next 30 days. On the first day each 30 day period, it will adjust the values of “ s ” and “ S ” for that period based on that forecast.
- In addition to these activities, each retailer agent will communicate through messages with its peers when it is at lower-than- s stock level so that an immediate replenishment (from its peers) may achieve to avoid falling into the situation that it has insufficient inventory before the supplier’s shipment arrives, which may take several days.
- During the above communication, a retailer may respond with yes or no to the request from other retailers. The decision is made based on the retailer’s decision process. More details of this process may be found in the explanation of the implementation details.

3.2.3 Important Parameters

- Demand distribution: Demands arrive every day, which lower retailers’ inventory levels and increase profit account. Demand arrival can be of any distributions as

long as it is coded in the simulation implementation. Some commonly used distributions may include normal, uniform, and exponential distributions (Gavirneni, et al., 1999). Demand distribution parameters may be set as changeable so that a user can change the parameter values to compare the results under different demand distributions.

- Decision Process for Supplier / Retailer: Decision processes are used by every agent for making such decisions as whether to accept or reject a proposal from another agent. The supplier and retailers may use different decision processes because of their different positions in the simulation. More details may be found in the explanation of the implementation details.
- Demand Forecasting Process: Our implementation uses the Java engine to generate normally distributed random variables as the demand for each retailer. Then retailers than use the Lee et al. (1997) AR1 process to generate the demand forecasting for next period. However, for future research, there are some other forecasting models available, such as moving averages, exponential smooth function, etc.

3.2.4 The Architecture

A simplest and most basic type of interaction involves two parties, who follow the basic behaviors in communications: one party sends a request and the other sends a response back. Based on the information system IOP model (Zachman, 1987), the request from P1 (short for Party 1) is regarded as the input to the system (or sub-system) of such two players, and the input is processed by the request-receiver P2 (short for Party 2). P2

generates the response, which is regarded as the output. The output is sent back to P1 (this output is named as feedback in this essay following the information feedback model of Forrester 1959), who takes it as the input and decide whether further communication is necessary. When this process continues, the interactions between P1 and P2 enters a loop until one of them decides to interrupt the communication because a satisfactory result is generated. There is another possibility, which is that the P2's response is used as the output of the system (or sub-system) and becomes the input of another system (or sub-system). We show this interaction process in Figure 5. It is noted that we replace the P1 and P2 with X and Y because in the later Figures, we use X, Y, and Z to represent three retailers in a supply chain system. Because the above described process is common across the supply chain (system), and many other systems, we simplify it as an interaction process with one arrow (input) in and two arrows (feedback and output) out. As shown in Figure 5. With this simplified symbol the used-to-be complex flowchart of system processes become much more readable and possible to fit on one page as in Figure 6. Note that the interaction processors are created for flowcharting and programming purposes (in a program, the interaction processor can be implemented as a method in Java or function call in C and other languages, which makes the implemented program modularized and easier). Each interaction processor may be different in itself because, for example, the process (such as individual decision process and consequent actions taken) within may be very different and agent specific.

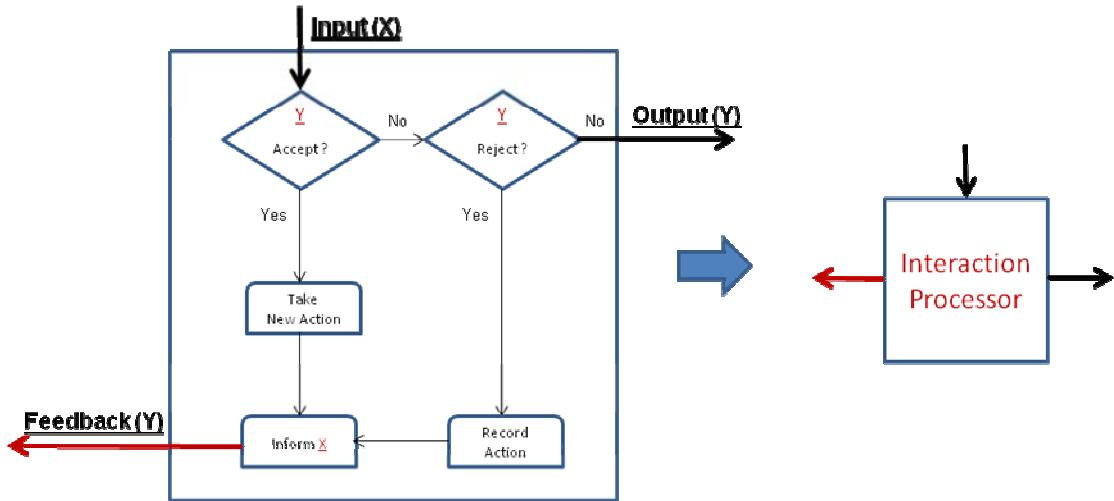


Figure 5 Interaction Processor

We may use the pseudo code to represent the logic of the interaction processor with explanation the follows:

```

//Given two agents x and y, and x initiates a proposal
function Interaction(x,y)
{
    proposal(x) accepted?
    If no, proposal rejected, than
        y prepares counter offer (if any) and send message to x: output(y)
    If yes, proposal accepted, than
        y takes new action, and send message to x: feedback(y)
}

```

The logic of the processor is very simple: to take the input from, say, agent x, and y processes it with certain output. Note that in the pseudo code, we have not included the necessary supporting functions to implement the decisions of y, nor have we provided the description of the functions `feedback(y)` and `output(y)`. This processor may be coded as a Java public method module in the implementation package that can be called by any agent. A common usage of the processor method is to put it in a loop: x sends a proposal to y and y (processor is called once here); and y sends the counter offer (the processor is

called again, but the input will be from y and x makes the decision); once x receives the counter offer, it decides, using the processor, its action, and if a new offer is formed, the loop continues. The loop may continue as long as necessary. However, it is not practical to do so for the following reasons. First, for synchronization consideration, since each period has a fixed set of ticks (time units, say miliseconds) and the number of ticks should be within reasonable limit so that the simulation will not run too slow, the interactions between any agents are bounded to the number ticks in a period. Second, in the real world, there is also a time limit as of one project should be completed with a limited time – the pressure of the timing may be much bigger than the simulation itself. However, we can not undermine the use of the interactiton processor. This method may be called many times and it will consume a significant amount of computing resource if it is not coded carefully. This may be briefly noticed in the flowchart in Figure 6.

One thing worth mention about the interaction processor is that this is a logical implementation and the simple interaction between two agents. Simply put, one agent takes input from another agent, processes the input, and outputs the result. This is a very typical IPO model in the information system literature. The output result is again used as the input and it is process and output to next... The implementation puts this in a loop. Depending on the implementation logic, the number of loops may vary. There exist numerous such interactions in a say, 3-agent environment as shown in Figure 6.

The flowchart in Figure 6 uses the interaction processors to simplify otherwise very busy interactions among three agents. In the flowchart, we assume there are three agents interacting with each other. Retailer X sends a request (proposal) of, for example

exchanging the overstock at certain price to retailers Y and Z. Y receives the request and will accept/reject it using the (implementation specific) decision rules in the interaction processor. Y may contact Z for additional stock. Same story happens on the Z side. We may find that loops with interaction processors are used in the flowchart. This chart may be expanded when more agents are included in the interaction.

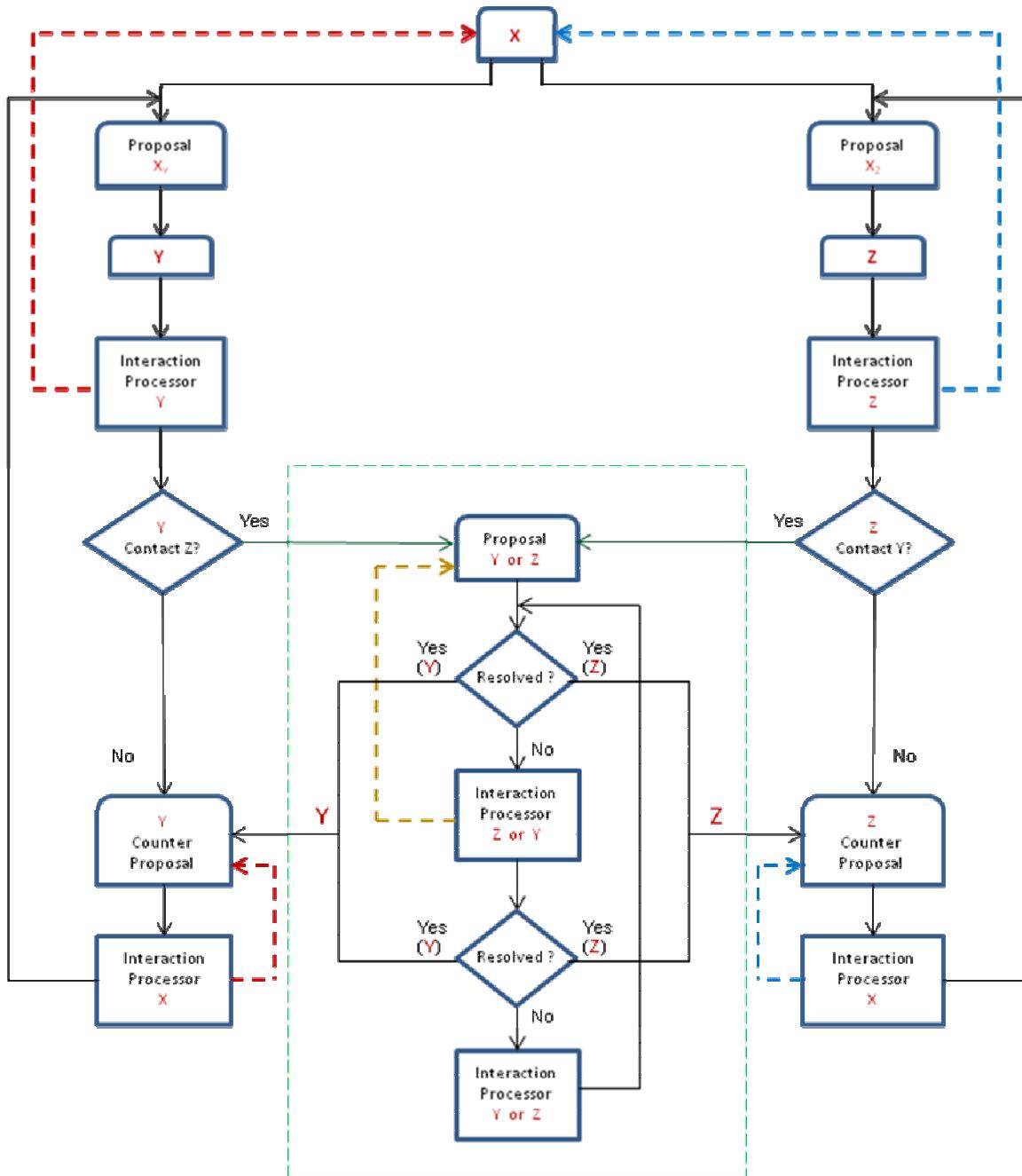


Figure 6 Flowchart of Interactions between retailers

3.3 Model implementation

Our model is implemented using language in Repast. Java is an objected oriented computer programming language that is suitable for modeling the individual entities. Details of the programming language features may be found in many textbooks and are not listed here. The essential characteristics of the object-oriented paradigm that are especially suitable for agent based modeling are embedded in each agent. For example, an agent is considered a class object and it possesses many attributes about itself such as inventory level, cost structure/variables, product library, etc. This agent also has many predefined behavioral rules for guiding the actions that the agent may take. Such behavioral rules may be, what to do when a customer demand arrives (the triggering events and the triggered behaviors), how to decide the next order level (the decision rules), when and from whom to replenish the inventory (the interaction and communication logic), etc. The implemented behavioral rules are also called functions or methods in the computer programming terms.

After the agents are coded in computer programs, they are placed in an environment to interact with each other. The publicly available simulation package Repast (version name, Simphony) is one of such implementation environments, which is also written in Java programming language and enables the processes of the agent based model. Details of Repast framework may be found from its own website (www.repast.com). What is worth of mentioning is the major benefits of using the Repast in this essay, which are that our programs written in Java are of fully compatible with the Repast framework (because Repast is in fact written in Java) and can be plausibly easily debugged and tested. We

also use the Java program development interface Eclipse, which has also been integrated with Repast. The combination of these tools gives us the possibility to develop the agent based model from scratch within a sensibly short period of time with the requirement of author's Java programming capability as well as the knowledge of agent based modeling.

The coded model runs within a specific domain or scope, that is,

- There is one product in the system
- There are three retailers and one supplier in the system. It is possible to add more retailers and suppliers to the system in the future implementation.
- Retailers are all equally distanced from the supplier, thus the transaction costs and time of transfer are the same for each
- Retailers are in different independent market segments so that they are not competing, though they sell the same product
- Demands in each location is independent and demand over time fall in normal distribution
- Retailers are not so far apart, thus the transaction time between them is negligible.

This assumption may be relaxed in future implementation.

- Information sharing between retailers and supplier is the demand information, which has been extensively examined in many previous research. This essay does not focus on how the information is shared, but the results. We use the approach in one of the previous research, in particular, Lee et al. (2000), for the external validation purpose. Information sharing between retailers is the stock (inventory) data including the overstock and backorders.

- When stock movement between retailers is necessary and there is a chance of its happening, the possibility of such stock sharing is calculated based on the relationship (high, medium, low) between the retailers.
- We assume that information sharing and relationship positively relate to each other, and to simplify, the links between them is linear. (It is possible to assume the exponential function will work too, as used in the friendship model in social study.)
- The relationship may also be predetermined by the contracts and negotiations between retailers which detail the terms and conditions of such relationship.
- The overhead of stock movement, including any transaction and transportation costs, at this moment, is set to a fixed and small (compared to the costs of overstock or under-stock) value for each time. Both sides (overstock and backorder) have the incentives to share the costs due to market fluctuations.
- In future implementation, such overhead may either be fixed by contracts; or vary based on the degree of the relationships.

There are four types of agents, namely, the demand agent, retailer agent, moderator agent, and supply agent. Each type of agent inherits SC agent³, which is the super (parent) class (a class which gives a method or methods to a Java subclass). The inheritance hierarchical structure is shown in Figure X1. The four types of agents are arranged in a framework as shown in Figure X2. The arrows indicate the inheritance relationships

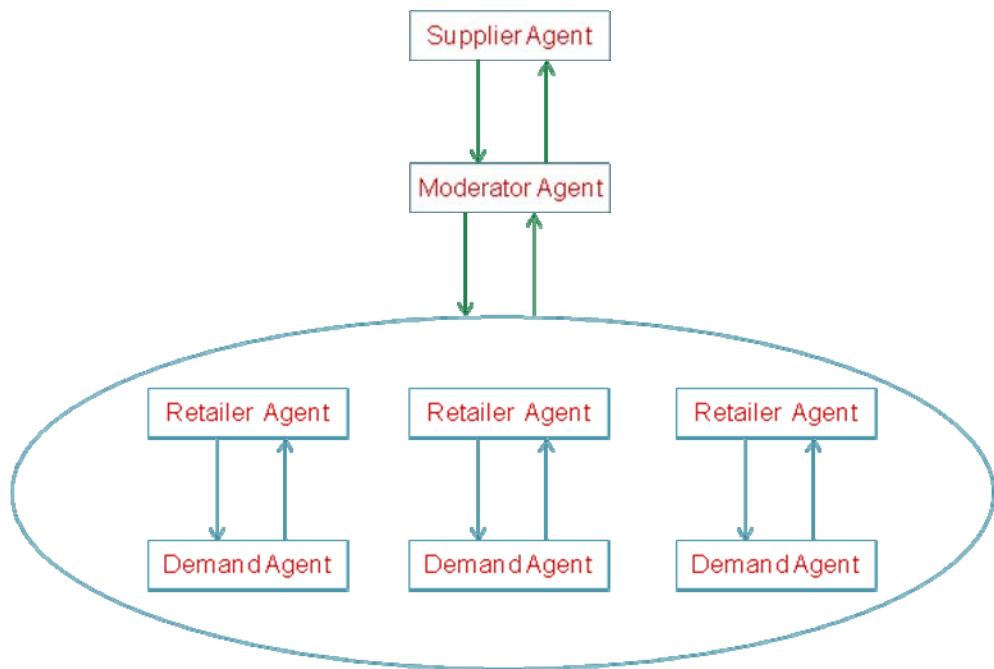
³ It is named as SC agent because this is the super class agent and it is the parent of all supply chain sub-class agents. The modeling approach follows the inheritance structure of object oriented programming language, which is considered one of the best fit for agent based modeling. More details about object oriented programming as well as Java programming language may be found at www.sun.com.

between the superclass SC agent and the subclass agents (A Java superclass is a class which gives a method or methods to a Java subclass.).



* The arrows show the inheritance relationships of the class objects

Table x1 Agent Class Objects of the Model



* The arrows show the direction of information and stock flow

Table x2 the Model

The demand agent is the simplest. The only task of this agent is to generate demand flow to feed into the retailer agent. There are three demand agents, one for each retailer. We consider external (customer) demand for a single product occurs at each retailer, where

the underlying demand process faced by each retailer is a simple auto-correlated AR(1) process. This approach may also be found in Lee et al. (2000), Kahn (1987, and Miller (1986). Let D_t be the AR(1) demand process at a retailer, where

$$D_t = d + \rho D_{t-1} + \varepsilon_t$$

Where D_t is the demand forecast at time t , D_{t-1} is the demand at $t-1$, d is a constant in AR(1), $d > 0$, ρ is the autocorrelation coefficient, $-1 < \rho < 1$, and ε_t is *i.i.d.* normally distributed with mean zero and constant variance.

The retailer agent is the seller to the customer. It takes the orders from the customer and reloads its inventory from the supplier. Since the retailer faces the customer and we assume there is no delay in delivering the product to the customer. The optimal order decision of the retailer follows the formula in Lee et al. (2000):

$$Y_t = D_t + \frac{\rho(1 - \rho^{L+1})}{1 - \rho} (D_t - D_{t-1})$$

Where Y_t is the retailer order quantity at time t , D_t is the demand forecast at time t , D_{t-1} is the demand at $t-1$, d is a constant in AR(1), $d > 0$, ρ is the autocorrelation coefficient, $-1 < \rho < 1$, L is the delivery delays (lead time) from supplier.

The full implementation programming code may be found in the appendix.

Chapter 4 Experimental Design and Analysis

This chapter implements the model as described in the previous chapter and analyze the results. There are two parts in the implementation. First we conduct an external validation by comparing the results of the model with those from the leading management journal (Management Science, namely). Then we conduct the experimental design and check the internal validity of the model.

4.1 External Validation

The encoded model needs validation. The task of validation, i.e., to show that the model is fit-for-purpose, is a complex one because the model is based on a gross simplification of the real situation and the experimental conditions are necessarily far removed from severe market or economic conditions. For example, the model does not allow for the complex behaviors of a real manager of the supplier who may stop or reduce supplying products to one of the retailers because of some social (for example, boycott) or political (for example, on different sides of the political agenda) issues between them. Also the experiments are necessarily at a much smaller scale than the real supply chain, as we mentioned earlier, where in a real market one supplier may have more retailers and one retailer may find multiple suppliers for an identical product under different brand names, involving one product rather than hundreds and thousands of products.

The model requires numerical schemes and contains uncertain market conditions. Thus verification, to check that the model has been correctly coded, and validation, to ensure

that the model is an adequate representation of reality, is required before great reliance can be placed on the results of the experiments. Verification is a relatively straightforward task, and our model has been carefully checked and compared to the other published (and thus assumed validated) model that we use for validation as we will explain in more details below.

Our verification and validation process uses the model descriptions and implementation of Lee et al. (2000), which shows within the context of a two-level supply chain consisting of a manufacturer (supplier) and a retailer and a non-stationary AR(1) end demand that the supplier would experience great savings when the retailer shared its demand information. Our model is coded based on the following setups and descriptions, which are very well summarized in Raghunathan (2001) and we borrow from it here.

1. Setup: A simple two-level supply chain model is considered, which consists of one retailer and one manufacturer. The external demand for a single item occurs at the retailer, where the underlying process faced by the retailer is a simple AR(1) process. Some of the notation and assumptions follow.
2. Model Notation
 - a. t : Time Period, $t = 0, 1, 2, 3, \dots$
 - b. D_t : Demand faced by retailer at time period t
 - c. Y_t : Retailer's order quantity at the end of time period t
 - d. L : Replenishment lead-time for the supplier and supplier
 - e. H : Unit holding cost per time period
 - f. P : Unit storage cost per time period

- g. ρ : The coefficient in the AR(1) demand function
 - h. d : The constant in the AR(1) demand function
3. Assumptions
- a. The AR(1) demand model: $D_t = d + \rho D_{t-1} + \epsilon_t$
 - b. ϵ_t is normally distributed with mean 0 and variance σ^2
 - c. $\sigma \ll d$
 - d. $0 < \rho < 1$
 - e. All shortages are backordered (backorder do not get back to the ordering process but are used for calculation the total costs)
 - f. The demand process is common knowledge to all the supplier and retailers
4. Additional assumptions when the model is used to validate the bullwhip effect
 (these conditions are proposed in Lee et al. (1997))
- a. Past demands are not used for forecasting⁴
 - b. Resupply is infinite with a fixed lead time⁵
 - c. There is no fixed order cost⁶
 - d. Purchase cost of the product is stationary over time⁷
5. Information structure

⁴ Raghunathan (2001) argues that supplier may reduce the variance of its forecast further by using the entire order history to which it has access, thus information sharing may not have significant impact on the supplier's stock and costs. A further investigation may be made in future research.

⁵ the capacitated supply chain case may be introduced to the model in the future implementation following the Gavirneni et al. (1999) paper

⁶ this assumption may be relaxed because when it is very small, it does not make significant impact to the total cost

⁷ also, this assumption may be tested in the future implementation

- a. When there is no information sharing, the supplier receives only Y_t at the end of time period t from the retailer
- b. When there is information sharing, the supplier receives Y_t and D_t at the end of time period t from the retailer

The ordering decision of the retailer is

$$Y_t = D_t + \frac{\rho(1-\rho^{(l+1)})}{1-\rho} (D_t - D_{t-1})$$

Supplier's anticipated retailer order quantity for period $(t+1)$ is

$$Y_{t+1} = d + \rho Y_t + \frac{1-\rho^{(l+2)}}{1-\rho} \varepsilon_{t+1} + \frac{\rho(1-\rho^{(l+1)})}{1-\rho} \varepsilon_t$$

It is assumed that the supplier is aware of the fact that the demand process D_t follows an AR(1) process, with known parameter d , ρ , and σ . This assumption is considered reasonable by the authors (Lee et al. 2000) because such information about the underlying demand process can be communicated to the supplier through periodic discussion with the retailer, or the supplier can be provided with historic demand data from which such information can be readily deduced with sufficient accuracy. The authors implied that this is already the information sharing. Indeed, without the knowledge of such parameters, as shown in Gavirneni et al. (1999), the costs would be higher under information sharing when the supplier has no or partial knowledge about the underlying process. The latter paper has considered, as one of the three situations, where absolutely no information flows from the retailer to the supplier prior to a demand to him

except for past data. The author concludes that in this situation, the supplier suffers the highest costs, comparing to other two, namely, partial information sharing and full information sharing.

In the case of information sharing, the supplier knows both the retailer's order quantity Y_t , and the error term ϵ_t (through the sharing of information of D_t) when he determines the order for next period.

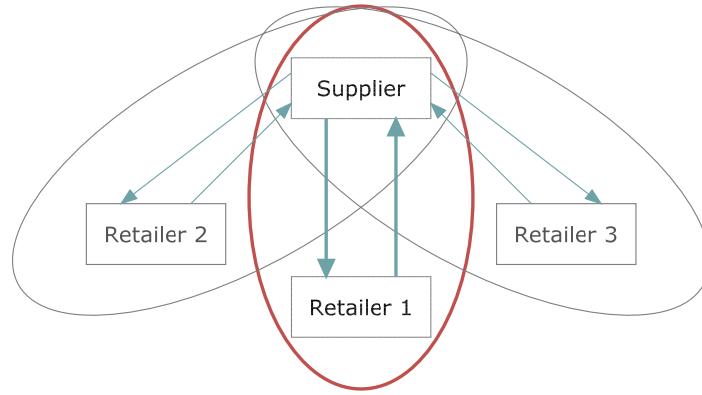


Figure 1 One Supplier Three Retailers: Three Pairs

Our validation model is illustrated in Figure 1. The underlying input is the independent demand for each retailer (not shown in the figure). The Lee et al. (2000) only consider one supplier and one retailer. In our model, we add another two retailers. This does not change the model dynamics since the supplier interacts with each retailer individually and the decisions of the supplier is made based on the individual interaction. In another words, our model is simply three Lee et al. (2000) running once at a time. In the figure, we use the oval to make the separation clear. It is assumed that each individual keeps his information locally and no global information is available. When information sharing

occurs, one individual shares his partial or all local information with another. This assumption is implied in Lee et al. (2000) and is important to be clarified for the implementation (programming) purpose.

Since the three groups work identically (except the parameters may be initialized with different values) and there is no need to examine all separately, we only look into one of them as shown in Figure 2.

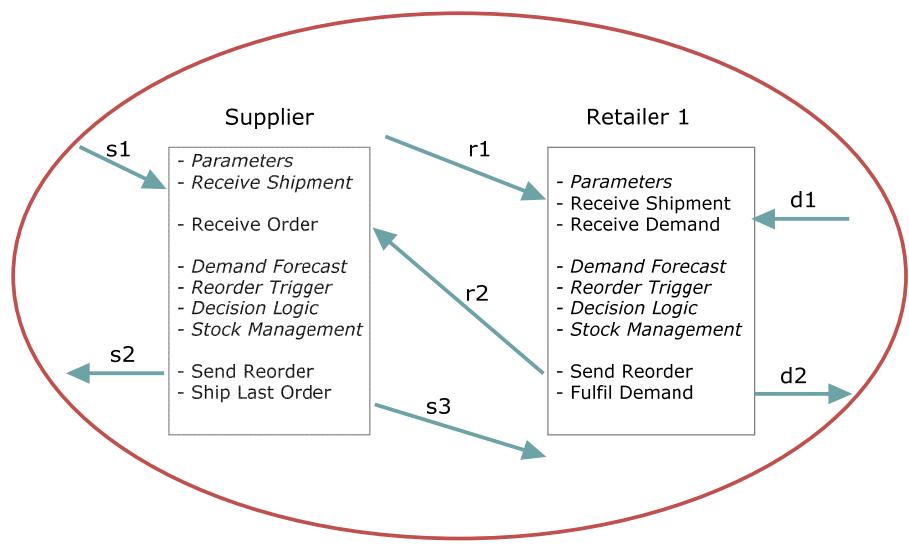


Figure 2 One Supplier and One Retailer: One Pair

Figure 2 is the model as described in Lee et al. (2000). Detailed explanations of the symbols and processes follow. The essential interactions between the two entities and the processes within each entity are similar for those with or without information sharing (readers are referred to the explanation for Figure 5 - Interaction Processor in Chapter 3).

d1: Demand arrives at the retailer. It is assumed the demand follows the AR(1) process.

All the historical and current order information is stored in the memory of the demand agent. This information is local to the demand agent only.

r1: Initialization. Retailer initializes the model run time parameters for retailer from the design, receives the demand for the period from the demand agent, and receives the shipment of his order of last period (It is assumed in our model implementation that lead time $L = 1$). All the information (parameters, demand, shipment) are stored in the local memory of the retailer agent.

s1: Initialization. Supplier initializes the model run time parameters for supplier from the design, and receives the shipment of his last order from the upper stream. It is assumed in our model implementation that lead time $L = 1$. It is also assumed that the upper stream has unlimited capacity. (In fact, the value of information in capacitated supply chain may also be modeled in the future, such as the one that is described in Gavirneni et al. (1999)). All the information (parameters, demand, shipment) are stored in the local memory of the supplier agent.

r2: After the retailer sets up the initial parameters, receives the demand and shipment, it activates the module to check if the current stock is sufficient for the demand as well as the predicted next demand (the forecast algorithm also follows the AR(1) process). The result may trigger another module to decide the quantity of the order for next period. All these modules are in the stock management sub-system of the retailer's decision logics. If no order will be placed, a zero order will be recorded. It is assumed that no negative order is allowed. Again, the retailer keeps all the information received and computed in

its own local memory. The cost calculation module reports the total costs for the current period, including the overstock (stock holding cost), under-stock (i.e., cost of backorders), and stock purchase costs. When everything is completed, the retailer sends the reordering information to the supplier and delivers the product to fulfill the demand. In the experiment of information sharing, the retailer also sends the demand information to the supplier.

d2: The demand agent receives the delivered product and saves the information to its memory.

s2: This is the continuation of S1. The supplier receives the order just placed by the retailer and uses its own stock management sub-system to check the current stock level for this and next period (forecasted) demand from the retailer. The forecast algorithm theoretically can be of anything, such as moving average, exponential forecasting, economic ordering quantity method, etc., but for the validation purpose, following the description of Lee et al. (2000), we use the given formula described in Chapter 3. The results from the stock management process may trigger the reorder process and the decision logic will decide the quantity of the reorder for next period. A no order will be recorded as zero for book-keeping purpose. Again, the supplier keeps all the information received and computed in its own local memory. The cost calculation module reports the total costs for the current period, including the overstock (stock holding cost), under-stock (i.e., cost of backorders), and stock purchase cost. A fixed ordering cost is added if an order is placed. When everything is completed, the supplier sends the reordering information to the upper stream.

s3. Supplier ships the order to the retailer. The retailer will receive this shipment in the next period and the process described above repeats. As we assume that lead time for both the supplier and retailer is $L=1$, the retailer usually receives his order placed two periods ago. When the lead time is larger, which may be true in the real situations, the retailer's delivery schedule may be stretched much longer.

The demand agent is the simplest. The only purpose of the agent is to generate demand for each period. It takes the initial parameters including the initial demand and the input and use the AR(1) process to calculate the output, i.e., the demand value for the current period. It is noticeable that the demand generation may be customized to fit the specific implementation if necessary. Figure 3 shows the simple flowchart for the demand agent. Once the demand is generated, the retailer takes it as the input. And the flow of information and product moves on to that as shown in Figure 4, the detailed implementation of the retailer in the model.

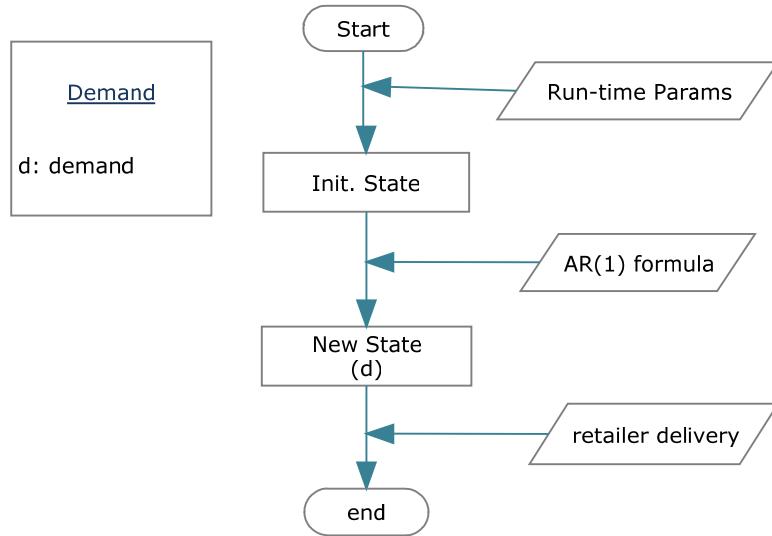


Figure 3 Demand Agent Process Flowchart

There are three retailer agents and they all run at the same time with the same pace (synchronized). We will only use one of the three to make the explanation because they are identical runs except they take different parameters and other data inputs. The retailer agent first, as the demand agent, receives the run-time parameters to initialize to the start state with the current stock and backorder information retrieved from his local memory. When the initial state is established, the agent updates its state to state A after the demand of the period from the demand agent and shipment (shipped last period) from the supplier is received. The shipment is added to the stock and all other information (run-time parameters, backorder, demand) is kept as the state variables to be used in the stock management sub-system. The stock updating logic is very simple in this validation process, following the description of Lee et al. (2000), which is shown in the code below. Decision logic determines the stock level and the backorder level, based on the values of the demand received.

```

stock -= totalRcvdOrder;

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

    supplied = totalRcvdOrder - stock;

}

else if (stock >= 0){

```

```
backOrder = 0;

supplied = totalRcvdOrder;

}
```

Then the new state B is established with the newly updated information. The agent now delivers the product to the demand agent (the customer) at the quantity determined from the previous state, and enters state C by computing the quantity of the current reorder to be sent to the supplier. If the calculated value is zero, no order will be sent and a book-keeping will be made to keep all the historical records.

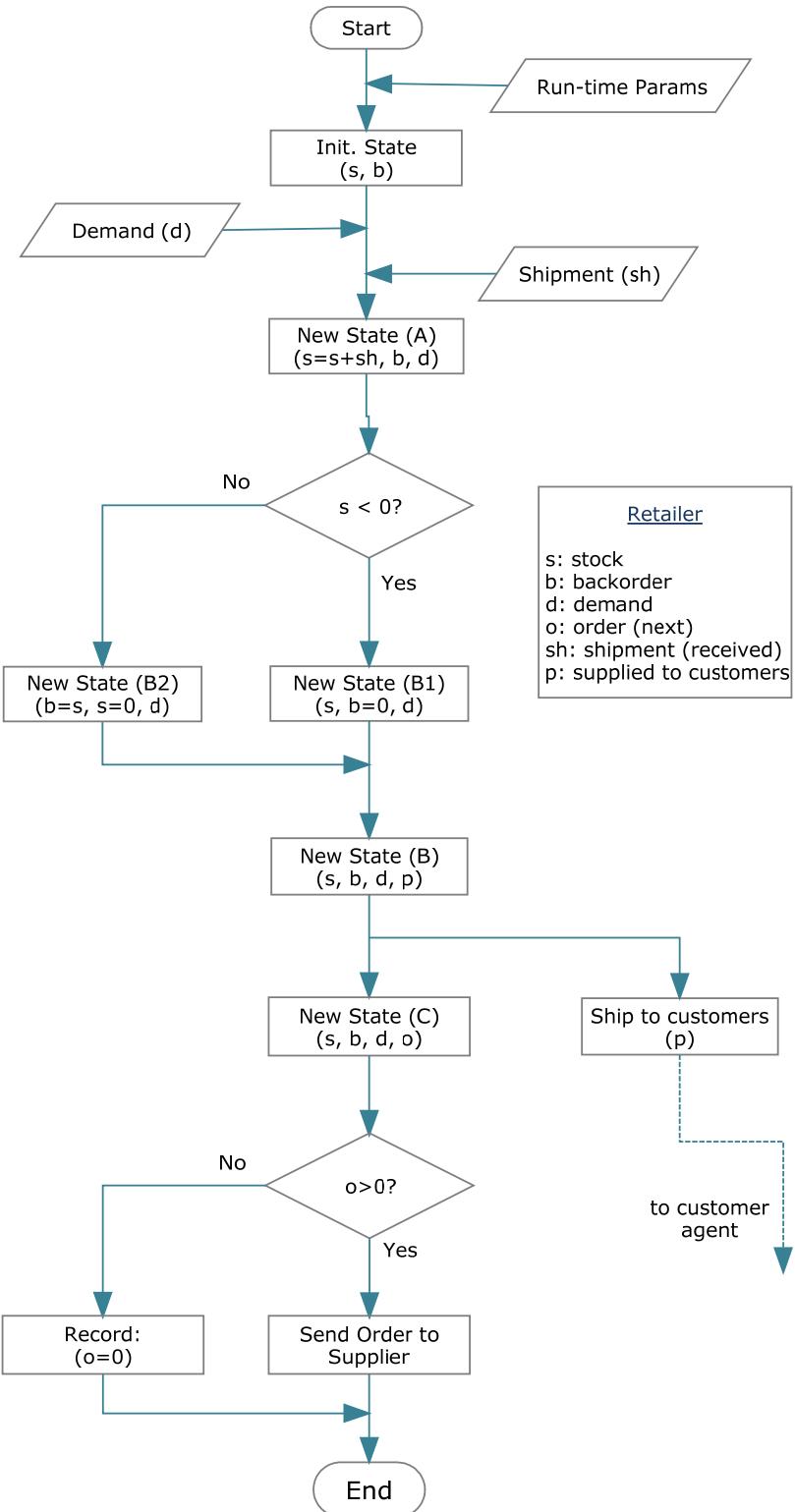


Figure 4 Retailer Agent Process Flowchart

When the period starts, the supplier starts at the same time as the retailers and it will end the same time as the retailers, too. The Repast's built-in JAVA multi-thread multi-task synchronization mechanism ensures that the periods start and end at the exact same time and commands are executed when expected. This synchronization is critically important for the precise and full control of every detail of the processes running at the run-time. If synchronization is not warranted, the retailers may, for example, not know the order received from the demand agent is from which period, or the supplier may, for another example, delay the shipment to the retailers certain periods. The fallouts of such abnormal behaviors of the model executing environment will totally void the purpose of the agent base modeling because the user would not have any control of the program run-time and could not justify the validity of the results from the model, no matter how many times he runs and how consistent (or reliable) the results may be. Figure 5 draws the flowchart for the supplier. From the flowchart perspective, we may find that Figures 4 and 5 are similar; however, the implementation code is different for each because the underlying logic for interaction process as well as the decision logic may be very different.

Similar to other agents, the supplier agent first picks up the run-time parameters and retrieves the stored stock and backorders (including the backorder for individual retailers) information from its local memory before it enters to the initial state. When the initial state is confirmed, the supplier receives the current (individual) orders from the retailers and the shipment for the order placed in last period from the upper stream. As we have assumed that there is no production or inventory capacity limit at the upper stream, the supplier receives the quantity exactly as ordered.

When the new state A is entered, the supplier has all the information for managing the stock. The individual orders are aggregated and the same procedure as the retailer is used to determine the current stock level as well as the possible backorder quantities. The newly computed information brings the supplier to the state B in which the supplier is armed with all the information for computing the order for next period, as well as the available stock for delivering to the retailers. When the stock is sufficient, all retailers will get what they have ordered. However, when the stock is low due to the variation of the demand and fluctuation in the calculated demand, the retailers are shipped the quantity proportionally of the available stock using the calculated fulfill rate. The model supplies the Lee et al. (2000) functions (as shown earlier) for calculating the reordering amount. The new state C is the final book-keeping and reorder finalizing state. The supplier now saves all the state information to the memory and sends the order to the upper stream. This completes the period and the new period repeats from the next tick of the clock.

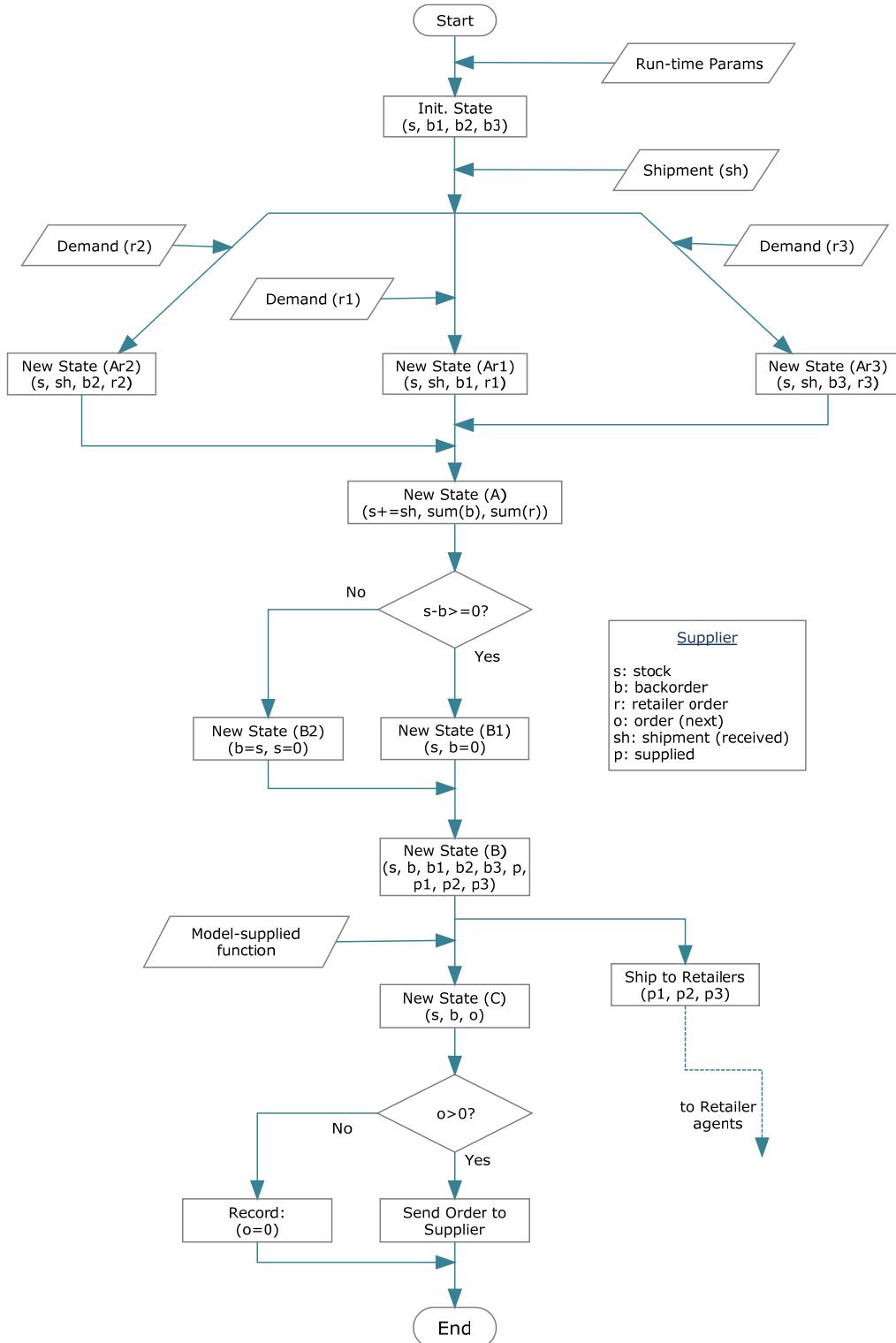


Figure 5 Supplier Agent Process Flowchart

4.1.1 The Results

Following the process in Lee et al. (2000), we vary the AR(1) demand process coefficient ρ from 0 to 0.9 to examine the impact of ρ . With the same parameters given in the paper, we generate the random demand for 2000 consecutive time periods and we compute the simulated average actual (on-hand) stock levels for the supplier. The intention is to analyze the impact of information sharing on inventory levels, inventory reduction, and inventory costs.

Figure 1 Impact of ρ on Average Manufacturer's On-hand Inventory

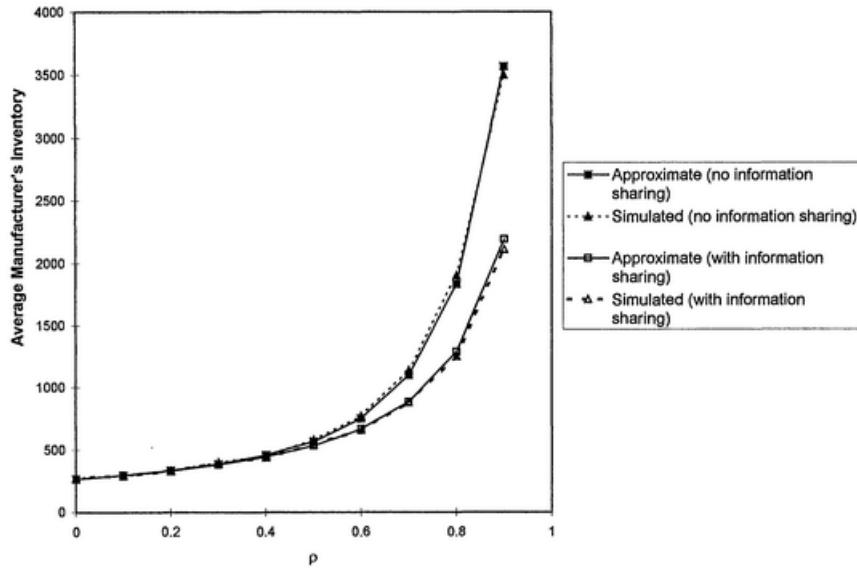


Figure 6 Impact of ρ on Average Supplier's Stock, Adapted from Lee et al. (2000)

Figure 6 is adapted from Lee et al. (2000). It reports the approximated and simulated average supplier's stock when ρ varies from 0 to 0.9, with and without information sharing. It is observed that the average supplier's inventory increases as ρ increases. The results that we have obtained from the agent simulation shown in Figure 7 is almost identical to those of Lee et al. (2000)'s. We easily can observe the upward trend of the

inventory levels as the value of ρ increase, and the average level of stock without information sharing is higher than that with information sharing. This figure shows the impact of ρ on the aggregate average stock level at the supplier side. The Lee et al. (2000) result is based on the interactions between one supplier and one retailer, while we have three retailers. We also check the impact to ρ on the stock specifically reserved by the supplier for each individual retailer. The figures 8 through 10 that follow show such investigation. Very similar, if not identical, to the results from Lee et al. (2000), we find the pattern we just described: upward trend of the average stock levels when ρ varies from 0 to 0.9; and information sharing does have impact on the stock for each individual retailer that reduces the average stock level. The consistency may be further found when we examine the percentage stock reduction and stock costs.

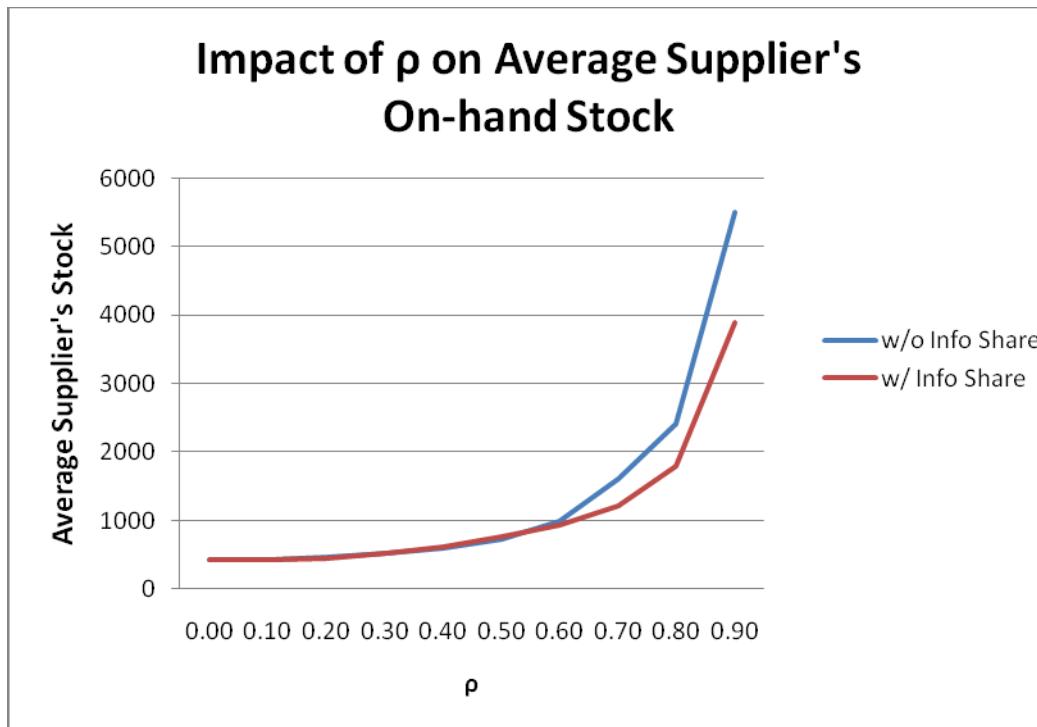


Figure 7 Impact of ρ on Average Supplier's On-hand Stock

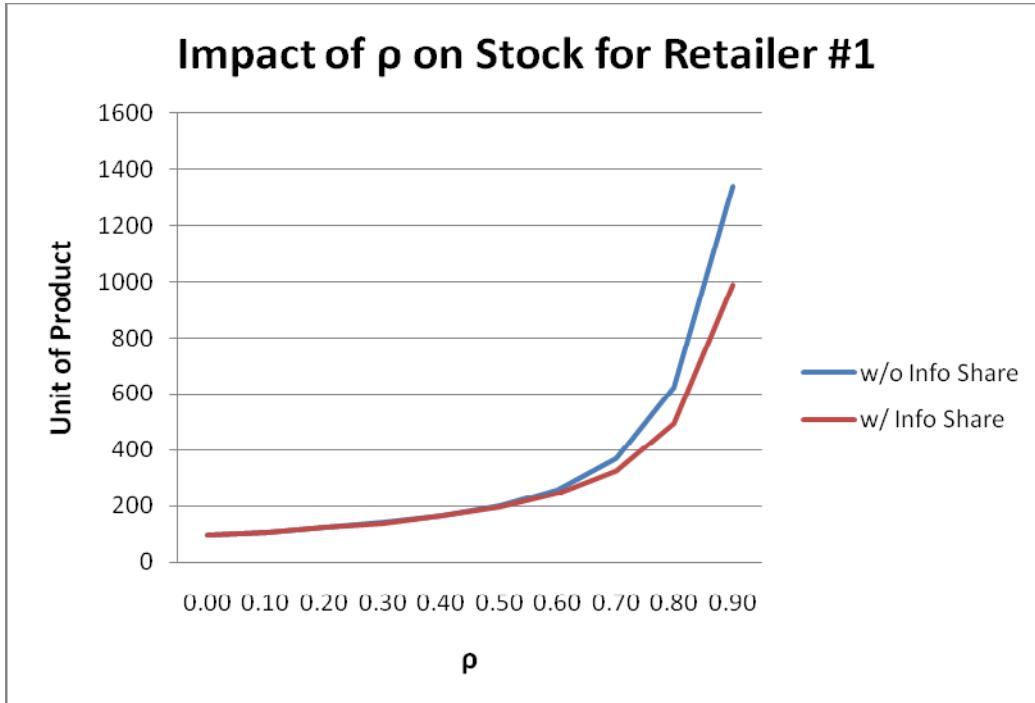


Figure 8 Impact of ρ on Stock for Retailer #1

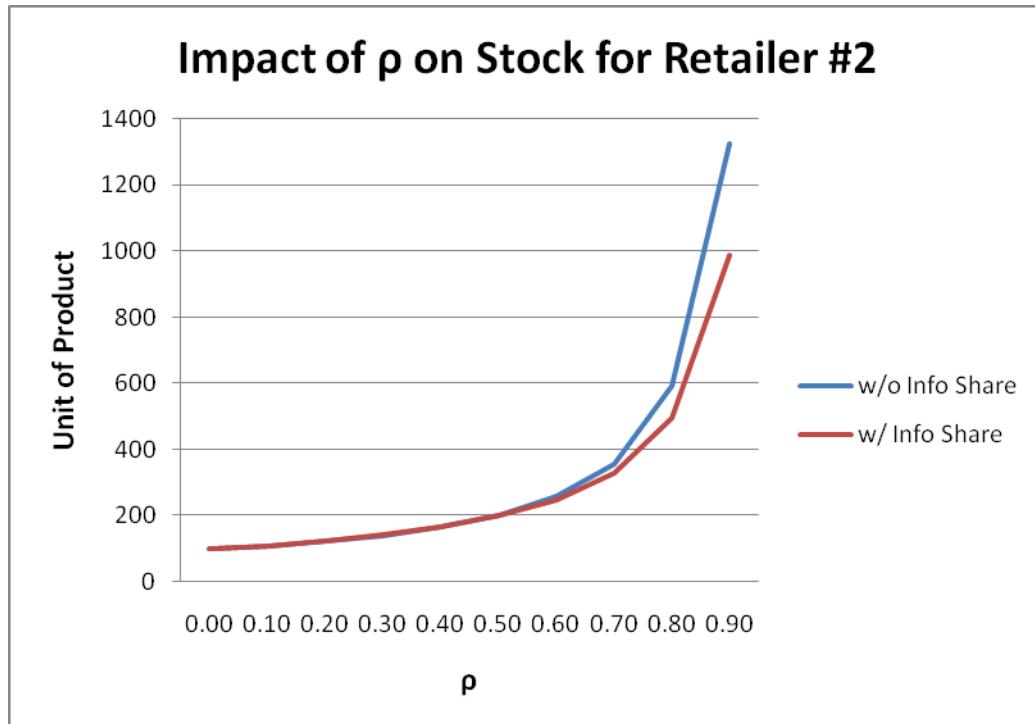


Figure 9 Impact of ρ on Stock for Retailer #2

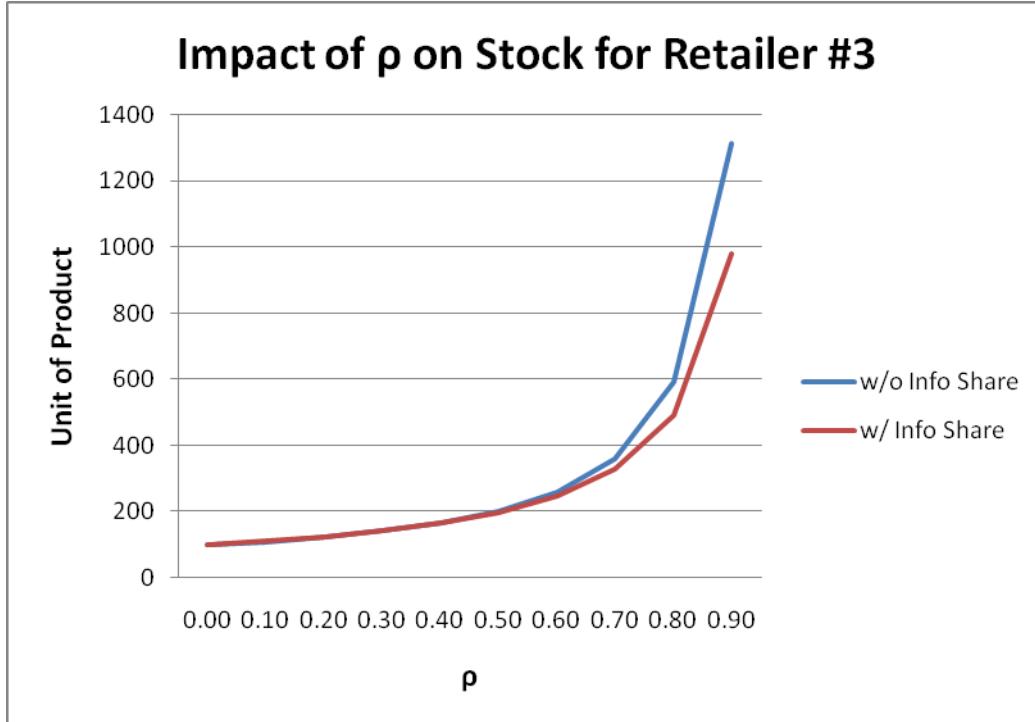


Figure 10 Impact of ρ on Stock for Retailer #3

Figure 2 Impact of ρ on Average Cost

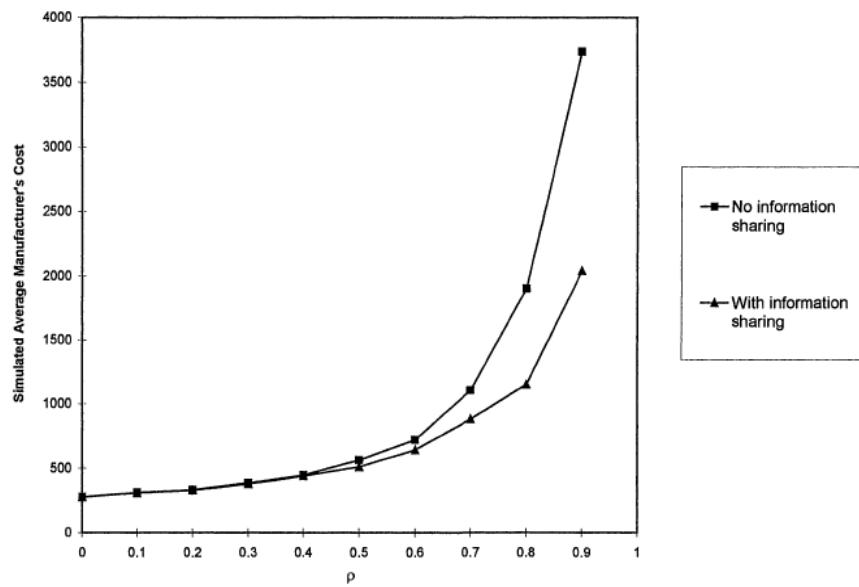


Figure 11 Impact of ρ on Average Cost

Figure 11 is adapted again from Lee et al. (2000), which shows the similar to Figure 6 observation for the impact of ρ on the average cost with and without information sharing. Figure 12 is the result from the agent based model. We once more see the consistent pattern of the impact of ρ as well as that of the information sharing.

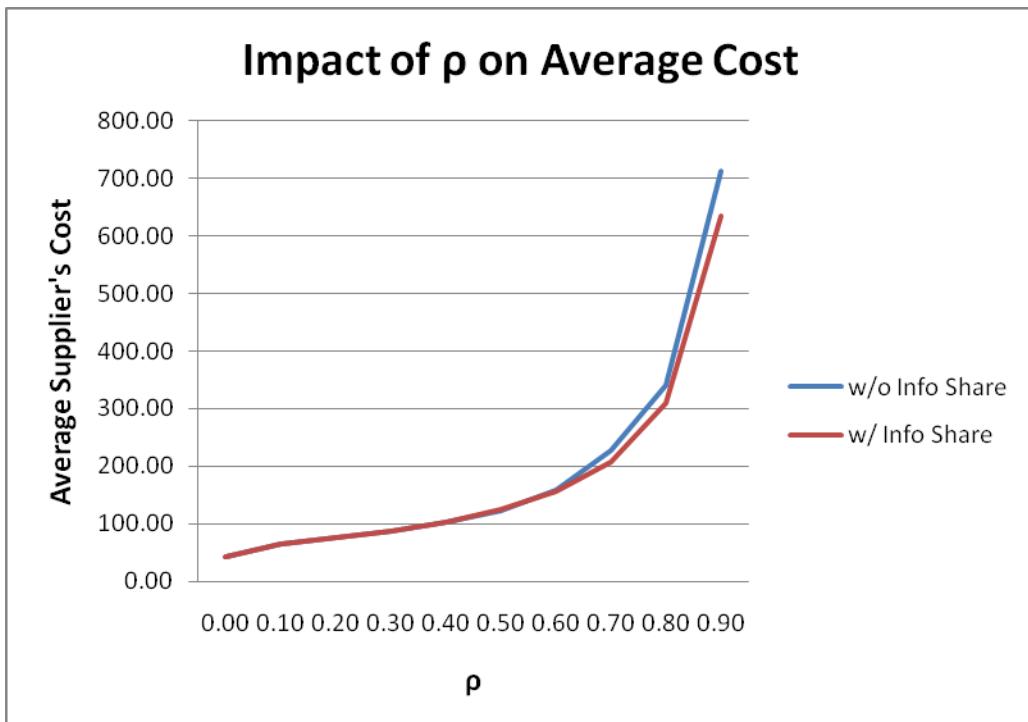


Figure 12 Impact of ρ on Cost Reduction

Figure 13 is adapted again from the same paper, which reports the percentage of stock reduction from information sharing. Figures 6, 12 and 13 all suggest that information sharing enables the supplier to reduce average stock, and this stock reduction, both in absolute terms or as a percentage of stock, is greater when ρ is greater. In fact these phenomena are consistent with the analytical findings as presented by the authors. Our results in figures 7, 12 and 13, as well as figures 8 through 10 are consistent with the

authors' findings. Thus we are confident that the model coded produces meaningful results.

Figure 3 Impact of ρ on Percentage of Inventory Reduction from Information Sharing

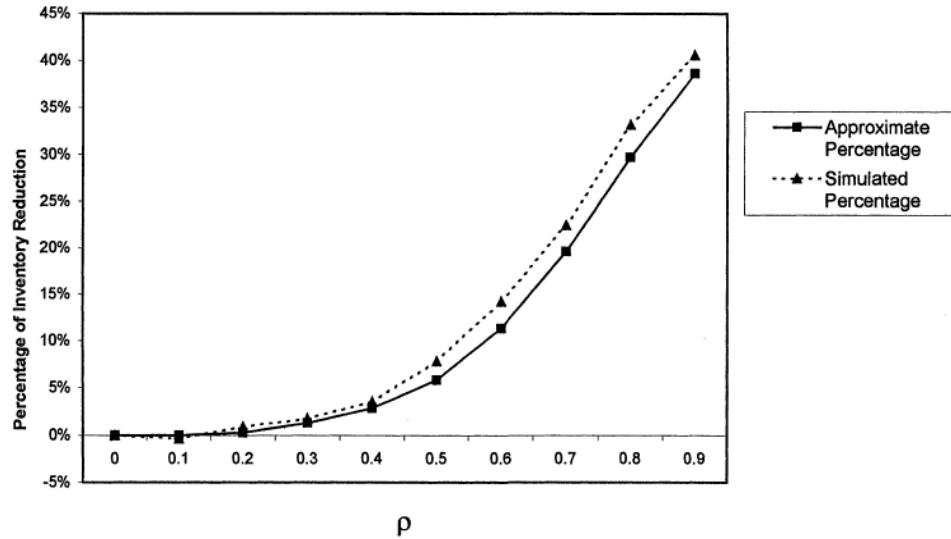


Figure 13 Impact of ρ on Percentage of Stock Reduction from Information Sharing

Lee et al. (1997) study the bullwhip effect phenomenon, i.e., the variance of order may be larger than that of sales, and the distortion tends to increase as one moves upstream. Four sources of bullwhip effect are analyzed, namely, demand signal processing, rationing game, order batching, and price variations. To make the case simple, we only use the agents to examine the impact of demand signal processing to the bullwhip effect. That is, the supplier receives true demand information from the retailer when information is shared. We will compare the results against the findings of Lee et al. (1997).

Figure 1 Orders vs. Sales

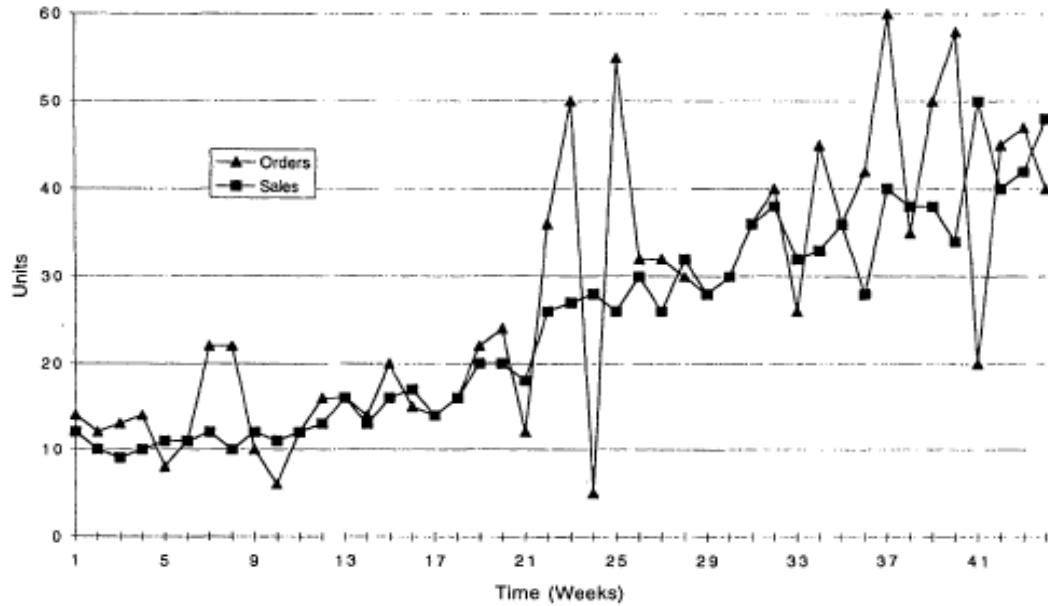


Figure 14 Bullwhip Effect: Orders vs. Sales, Adapted from Lee et al., 1997

Figure 14 depicts the authors' exemplary demonstration of the bullwhip effect in the market place based on real data but manipulated to maintain confidentiality, which shows a retail store's sales of a product, alongside the retailer's orders issued to the manufacturer. The figure clearly highlights the distortion in demand information. The retailer's orders do not coincide with the actual retail sales. In the real case as depicted in the figure, we cannot "assume" that the retailer knows the demand distribution and process (such as the AR(1) process in our case) and all the parameters that we use are not known to the retailer, the information distortion is obvious from the very end of the supply chain (the customers demand vs. the retailer's sales forecast). However, this is not the case in our agent base model settings, which follows the settings in Lee et al. (2000). In our case, the retailer knows the demand process that is AR(1) and it has all the

information about the parameters. So, to make the cases comparable, we move the supply chain up and try to find the information distortion at the next level between the supplier and retailers.

The results are depicted in Figures 15 and 16.

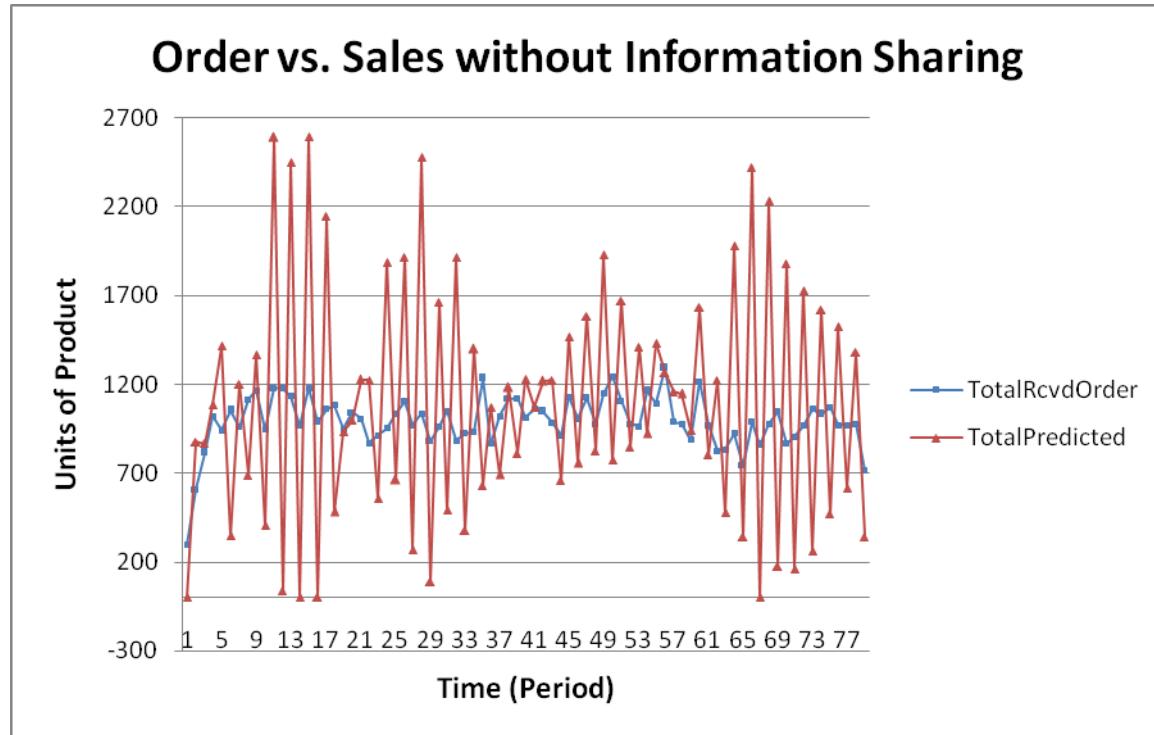


Figure 15 Order vs. Sales without Information Sharing

Figure 15 simply recreates the bullwhip effect very nicely. It is obvious that the blue demand from the retailers is fluctuate much less than the forecasted quantities by the supplier. The only difference between this figure and the Lee et al. (1997)'s is that the retailer order's trend does not go up as the demand from the customers does not, per the settings of the model parameters. The reproduction of the bullwhip effect gives us the

confidence that the agent base model that we have created is producing the appropriate results.

As Lee and Whang (2000) suggest that information sharing is one of the solutions that has positive impact on the bullwhip effect, i.e., to reduce the variations in the forecast. Figure 16 shows the support to their suggestion.

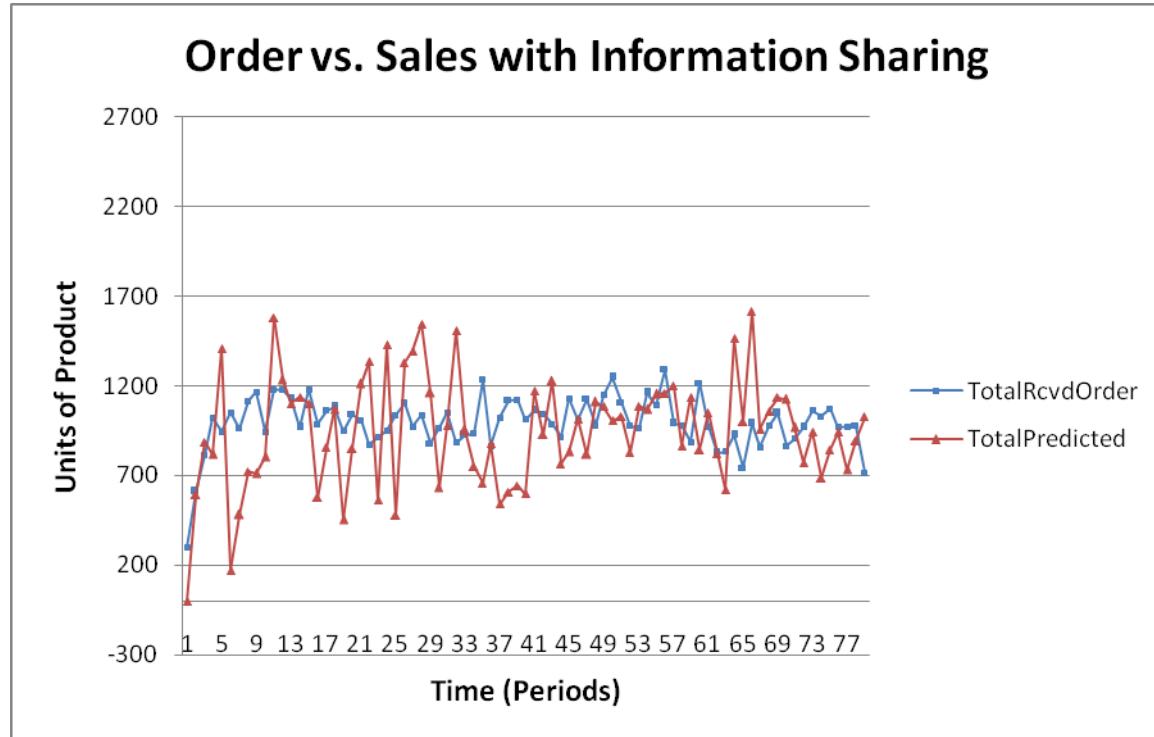


Figure 16 Order vs. Sales with Information Sharing

In this figure, the total orders from the retailers are the same since we repeated the experiment based on the exact same parameters⁸. An obvious improvement is found as the variability of the forecasts is much smaller comparing to the line in Figure 15.

⁸ This is exactly one of the benefits of using agent based modeling method as mentioned earlier as we can recreate the same scenario precisely as many times as needed.

The above experiments concludes the validation process and from the careful experiments and comparison, we conclude that the agent based model that we have created is sound and works as expected to create valid and reliable results.

4.2 Experimental Design (Internal Validation)

Now that we have inspected the validity of using agent based modeling to examine the impact of information sharing to the performance of a supply chain, we make some modifications to our model to investigate the effects of the interactions between the retailers to the supply chain. The external validation process explained above is simply a special case in our interaction experiments, i.e., the interaction coefficients between the retailers are zero, and all the information and material flows between the retailers and supplier pass through the moderator, thus we only study the one to one relationship between supplier and retailers. With interactions, using the moderator, the information flows are managed by the moderator and the supplier no longer interacts directly with the individual retailers, but the moderator. This is shown in Figure IM-1, which uses the dotted lines and faded color to differ from the model in the originally proposed (Table x2). We use the moderator in our internal experimental design, i.e., the validation process.

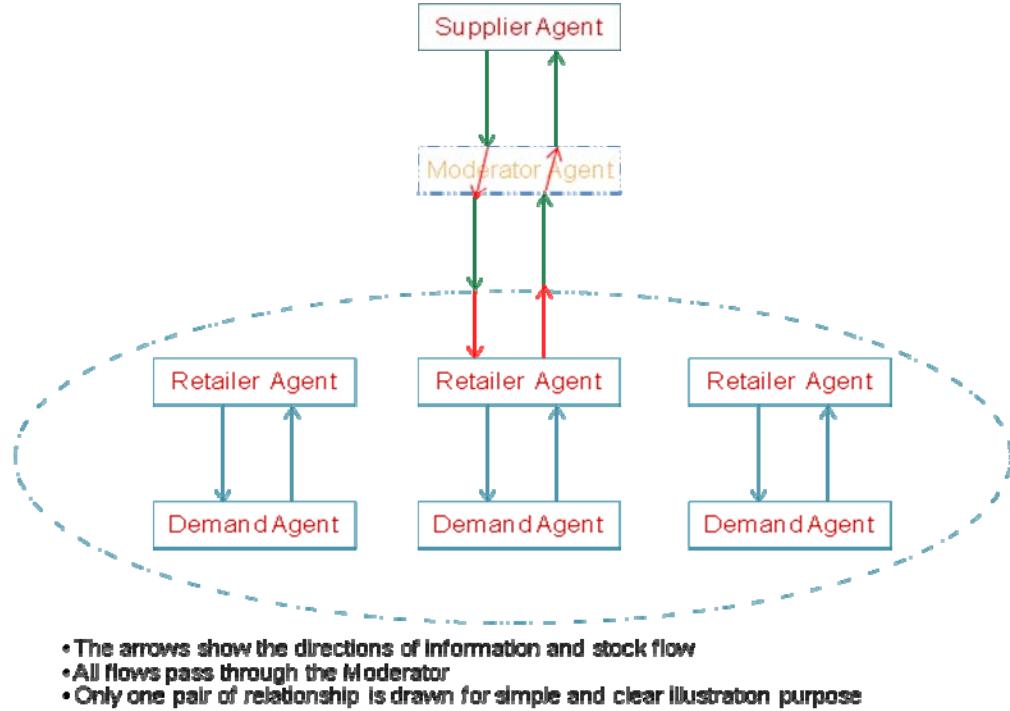


Figure IM-1 The Pass-through Model

The supply chain is a demand driven model and our agent based model follows the same logic. The volatility level of the demand (the variance, in fact we use the standard deviation σ as used in Lee et al. 2000) from the demand agent will have different impacts on the other end of the supply chain, i.e., the supplier agent. As the demand process is an AR(1) process in our model (demand processes of this form have been assumed by several authors (e.g., Chen et al., 2000, Lee et al., 1997), another factor may have effects on the stock and total cost of the supplier agent, which is the coefficient ρ in the demand generation function. Finally, as the agent based model has encoded the relationships between the retailer agents, the experiment design will also test what effects of the levels of relationships may have on the supplier performance.

Like the literature we have examined in the design of the experiment, we also assume that all levels in the supply chain (namely, the supplier and retailers) use the same demand

data, the same inventory policy and the same forecasting technique, which allows us to determine the impact of just the demand forecasting, as well as the interaction/relationships, without considering the impact of different forecasting techniques or different inventory policies across the levels. However, in the future research, we may examine the impact of different forecasting techniques, as well as the different inventory policies. Thanks to the modularizing features of agent based modeling, the new experiments may be designed in a much easier manner by simply adding more functional modules (methods) to the agents, or simply add some more agent types, comparing to the traditional procedural programming and mathematical programming which may require the re-writing of the whole implementation.

The experimental design uses factorial design of a 3x3 matrix that has three levels for the standard deviation σ and three levels for coefficient ρ as shown below. The dependent variable is the total inventory costs of the supplier for each period.

		σ		
		H	M	L
P	H	Test result	Test result	Test result
	M	Test result	Test result	Test result
	L	Test result	Test result	Test result

Where

H: High

M: Medium

L: Low

In the experiment, we setup three levels for each of σ and ρ . The values of these variables are:

		Retailer 1	Retailer 2	Retailer 3	Supplier
σ	H	180	140	100	140
	M	80	60	40	60
	L	30	20	10	20
ρ	H	0.80	0.80	0.80	0.80
	M	0.50	0.50	0.50	0.50
	L	0.20	0.20	0.20	0.20

Also the level of the relationships cor has three levels as show below (in fact we may treat this as the third dimension to the 3x3 factorial design matrix. However, it is difficult to visualize the 3D effects, so we break them down to sets of 2D tables with one parameter at fixed level):

		Retailer 12	Retailer 13	Retailer 23	Medium
cor	H	0.85	0.90	0.95	0.90
	M	0.55	0.60	0.65	0.60
	L	0.25	0.30	0.35	0.30

Varying the relationship levels (H, M, L) reflect different business scenarios as described below. The assumptions of the scenarios are the same as the description in the model implementation design, which are that in the market there only exist one supplier, one product, and three independent retailers. The demand for each retailer is independent of others, thus these retailers are not competing with each other. As argued in the well-regarded paper Axelrod and Hamilton (1981) published in *Science* and the later book “The Evolution of Cooperation” (Axelrod, 1984 and 2006), in a situation where there exist interactions, cooperative relationship will evolve. However Axelrod and Hamilton have not discussed the levels of such cooperative relationship, which define them in our implementation as high, medium, and low. In another words, when the relationship level is high, the chance (or probability) of helping each other is high; and when it is medium or low, the chances are medium or low.

- Scenario 1: This is the situation that the value of relationship may be low when the retailers do not often interact with each other for help or offering help. In such situation, without active interactions and more specifically, communications, information sharing is at minimum and the chance of exchanging stock information is low. Thus we set the value to 0.3 or lower, meaning, when needs rise, the agents only have 30 percent of chance to cooperate. It is possible when one retailer finds that getting help from another retailer may be more economically plausible, his request may be rejected due to the lack of communications and thus the adequate cooperative relationship between each other. This situation may be improved when interactions and exchange of information between retails increase, and cooperation will evolve.

- Scenario 2: The value of relationship is set to the medium range. In this situation, a concrete cooperative relationship has not been established. For example, the retailers have not setup the contractual and formal cooperative relationship, and a retailer is willing to help others because he gets help for the others too. This follows the Tit for Tat strategy of Axelrod and Hamilton (1981). The mutually informally agreed relationship is not guaranteed and the information sharing is volunteered. Thus we set the average value of the relationship to 0.6, meaning there is an average 60% of chance to cooperate.
- Scenario 3: The value of relationship is set to high. A concrete cooperative arrangement is found in this situation. For example, the retailers may have certain type of contractual relationship, such as strategic alliances, or other type of cooperative contracts. With the formal relationship, the interactions between the retailers are on the regular basis and information sharing is a usual matter. Thus when needs rises, the retailers will very likely help each other. So we set the value to 0.9, meaning, there is 90% of chance to cooperate. Note that even at the high relationship level, the chance of cooperation is not 100%. There are many explanations for this setting, one of which is that each retailer is assumed to be an egoist and to act rationally. For the details of

The data for the experimental design implementation will be filled in the table following as shown below:

$cor (0.30, 0.60, 0.90)$				
		σ		
		H	M	L
ρ	H	Test result	Test result	Test result
	M	Test result	Test result	Test result
	L	Test result	Test result	Test result

The test results in the table will be replaced with the data as the supplier agent average total costs. These values are the average values from the runs of the model for 150 periods. One table is created for each relationship cor value.

The runtime parameters include, in addition above values, stock cost per unit = 0.2, back order cost per unit = 0.2, purchase cost = 0.4, fixed order cost = 0.0.

There are changes to the program code for the implementation. Since the interaction will cause the changes to the back order level and in turn affect the overall cost at the supplier side, in our program we add the logic of handling the backorders. All the backorders at the retailers and suppliers are recorded and shall be fulfilled at the coming periods.

4.2.1 The results

Since we have implemented our factorial design at three dimensions: demand standard deviation, demand AR(1) process coefficient, and the relationships *cor*, we get nine matrices plus one without retailer agents' interaction matrix for comparison purpose.

We use the three relationship levels for the three scenarios as explained above. At each level of the relationships, we run nine sets of experiments. For example, at the low relationships, we have three levels of standard deviations σ and three levels of AR(1) coefficient ρ , which makes 3×3 sets. Each set of the experiments needs to run 200 times, from which we find the average total costs for the supplier. After all three levels of relationships are completed, we will have 3×3 tables for data analysis, one table for each level of relationships. Also for the comparison purpose, we run the base set of experiments that does not involve the relationship, i.e., the information flows pass through the moderator between the retailers and the supplier, as stated earlier. This is the base case of average costs when there is no interaction, which makes the four cases for shown in the charts that follow.

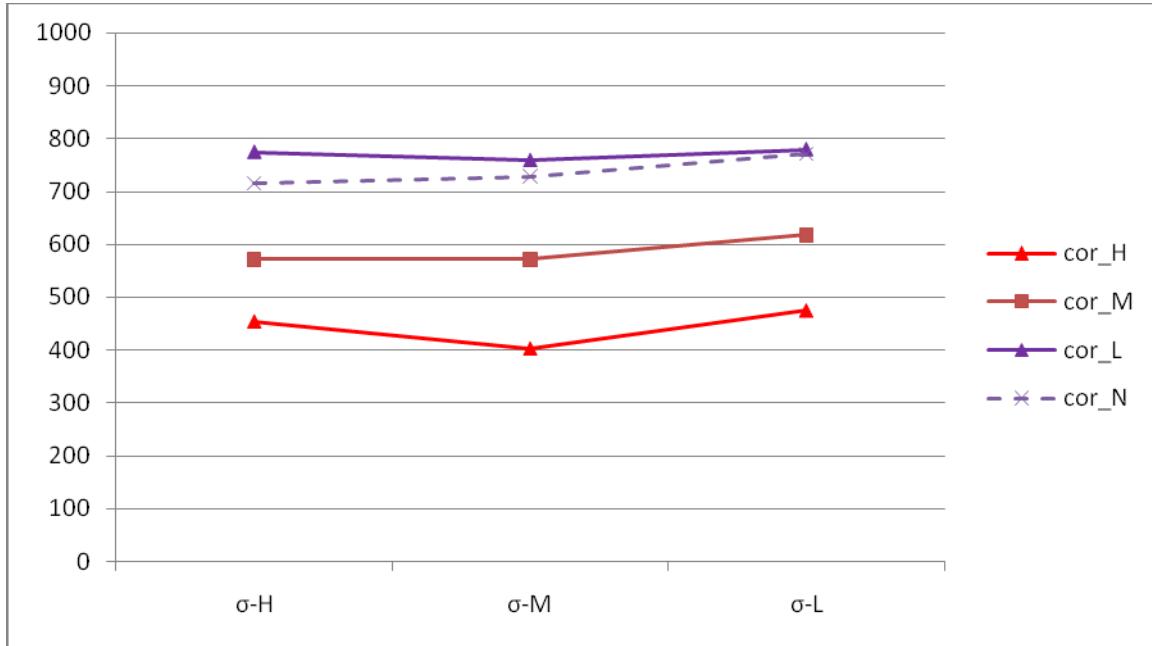


Figure 17 Effect of Information Sharing with Respect to Relationships and σ

The results are aggregated into one chart as shown in Figure 17. In this chart, at an average level of ρ (there are three levels of ρ and we average the results from these three levels and plot them there), the general conclusion is that the higher the relationship level between the retailers, the better cost savings we get. The lowest blue curve with diamond marks shows the significant savings at a high relationship level. The next lower level is the results from medium relationship, which ranges from about 550 to 650 and is much lower than the curve for the results from without relationship (the curve with x marks). The interesting curve is that from the low relationship level. It indicates the slightly higher total costs when retailers have only some interactions. This is mainly due to the results from the experiments that use low ρ and medium ρ , as shown in the charts that follow. We find that when relationship level is high between the retailers, the total costs are significantly lower than those without relationships.

The breakdowns of the impact of different level of ρ is showing in figures 19 to 21 When we revisit the Figure 11 adapted from Lee et al. (1997) which is shown in the external validation part above, we notice that when ρ is at lower ($0 \leq \rho \leq 0.4$) level, the average cost at the supplier does not vary very much and the curve is almost flat. We confirm this trend in our experiment shown in Figure 12. This indicates when the value of ρ is not very big the impact of ρ on the demand is relatively small (or none if $\rho=0$), thus the variation of the demand will be minimum to not significant. Therefore we may expect that the small variance at the demand end due to the nature of the AR(1) process will cause relatively small fluctuations at the supplier side. Since the two retailers will share the stock only at low or not so high levels as the possibility of such interaction is decided by the “IsShared()” test, when interaction is added, this may in fact cause the higher fluctuation of the demand seen from the supplier side.

“IsShared()” is a function embedded in the implementation that will decide whether the pair of retailers do interact given the relationship levels (H, M, and L) as well as the levels of current stock levels at both side. The yield possibility of interaction may be low when demand variation is low. More details may be found in the code in the appendix.

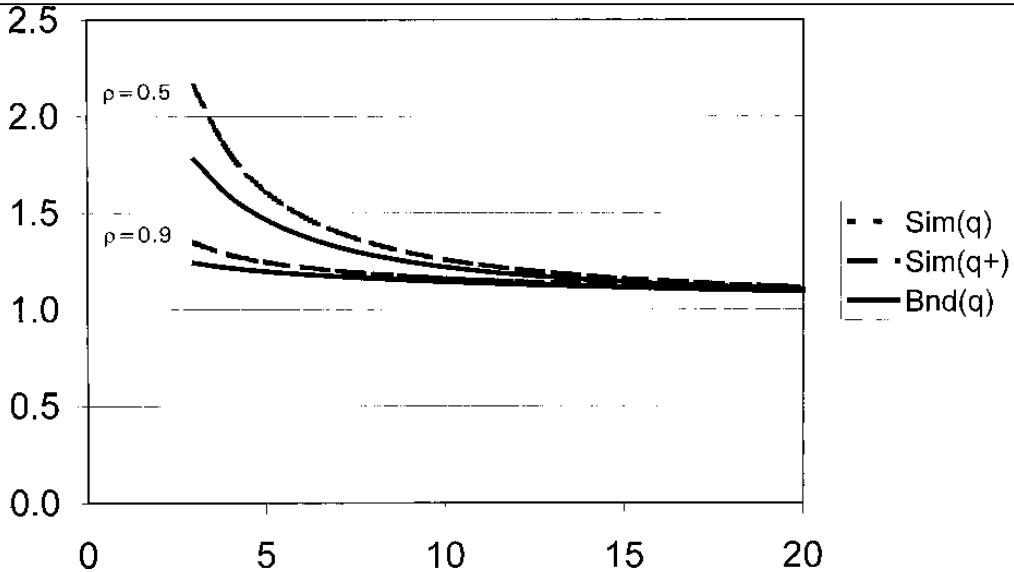


Figure 18 (Adapted from Chen et al. 1997) $\text{Var}(q)/\text{Var}(D)$ for $p = 0.5$ and 0.9

The above statement (without interactions at the retailer side) is mathematically proved in Chen et al.(1997) and Figure 18 depicts the difference between the variance at the supplier as well as the retailer sides. In this chart, the Y-axis is $\text{Var}(q)/\text{Var}(D)$, where $\text{Var}(q)$ is the variance of order placed by the supplier and $\text{Var}(D)$ is the variance of the demand received by the retailer. Chen et al. uses the same AR(1) demand process. The X-axis is the number of periods the supplier uses to find the moving average of the order received from retailers. This is to simulate the situations from minimum information to sufficient information at the supplier side. It is clear that the lower the value of ρ , the higher fluctuations at the supplier side (the value of $\text{Var}(q)/\text{Var}(D)$ get greater). With higher volatility, the cost of accommodating such volatility gets higher.

When interactions between the retailers are involved, which always happens when necessary. That is, when the value of relationship is high, the retailers share the overstock and backorders, which will reduce the variance of their orders to the supplier.

In turn, therefore, the supplier side will show less volatility and thus smaller average total costs. This is shown in Figure 19. The curve of cor_H is line at the bottom. The dotted line is from the base case without interactions. We notice that the curves for lower and medium relationship levels show higher than base case costs.

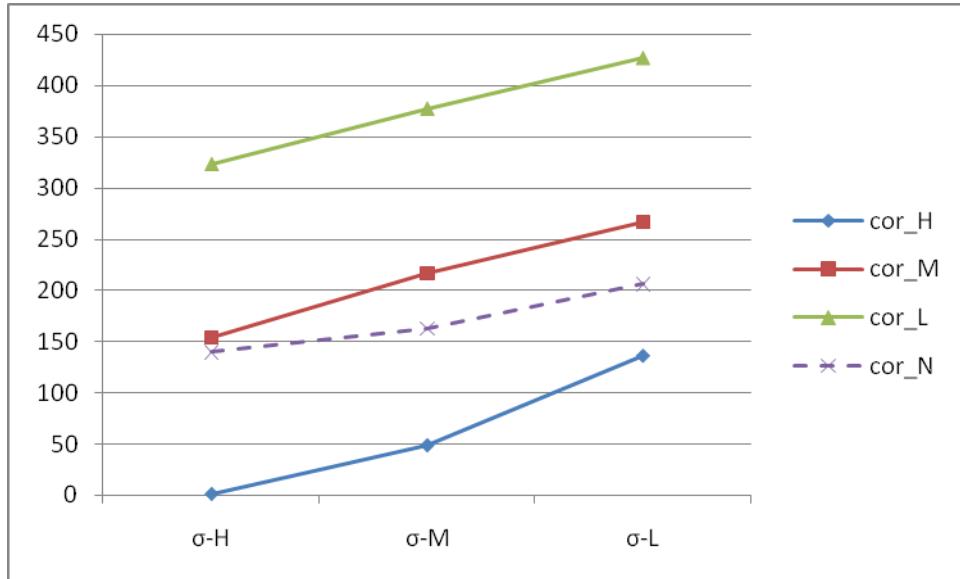


Figure 19 The Levels of Costs When ρ is Low

We may use an example to explain this abnormal result. To make the case simpler, without losing generality, we only examine two retailers with independent demand sources the same as in our implementation, and one supplier in the supply chain. Adding more retailers will not change the dynamics but add complexity, which thus is not necessary. Also we suppose one retailer (A) has relatively small variations in demand and the other's (B) relatively bigger. When the transactions start, A produces somewhat steady ordering flow up to the supplier, while the reordering quantities from B change with bigger deviations. If no interactions are involved, the commonly documented bullwhip effect will be observed as many researchers have proved. Suppose the

relationships between the retailers are at the low or medium levels, which means that the possibility of sharing information and stock is at a level that they tend not to happen. This may result in a distorted demand pattern in the eyes of the supplier because the variations of the demand may seem higher. Therefore, the supplier, without the knowledge of whether the two retailers have shared stock or not, may be “fooled” by the resulting reordering quantities received from the retailers and believe the quantity of order received is the true quantity and computes the underlying demand patterns accordingly. The supplier thus adjusts his own ordering quantity in favor of the adjusted demand. As proved in the bullwhip effect theory, the misperception results in an exaggerated variance of demand along the supply chain. The consequence is shown in our Figure 19 where the two curves of low and medium relationships at low ρ are above the base case line.

When the value of ρ reaches the medium level, the results start to show the advantages of interactions between the retailers. Figure 20 is the cost levels for medium ρ at the supplier side.

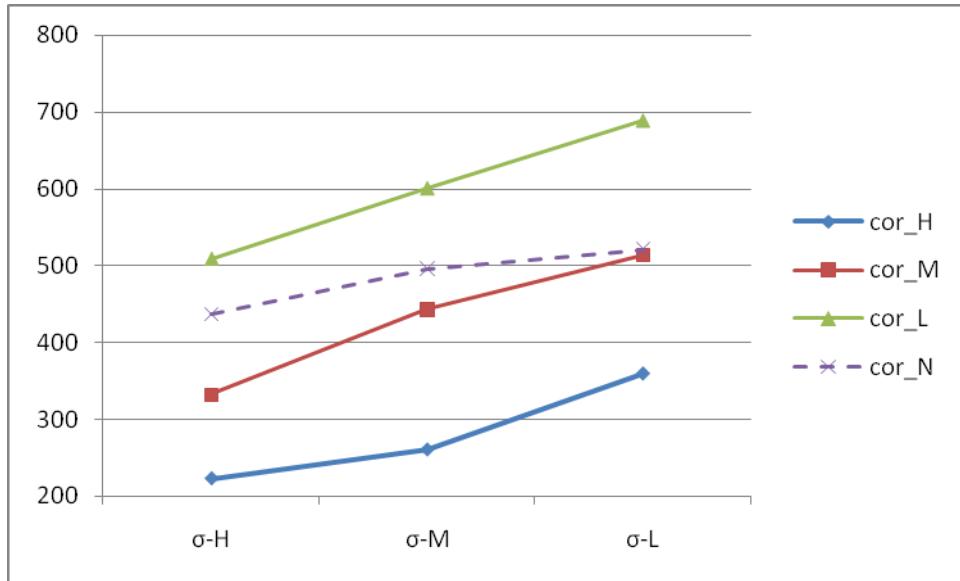


Figure 20 The Levels of Costs When ρ is Medium

When ρ is increased from low to high, by comparing this chart with the previous one, we notice that the *cor_N* (no relationship) curve range increases from about 150-200 to around 450-520, an increase of about two times the original; while the *cor_H* (high relationship) increases from about 10-180 to 230-360, about two times of the original. The significant change to the picture is *cor_M* (medium relationship) which also increases by about doubled from 150-270 to 330 -520. Although *cor_L* (low relationship) still shows higher costs at range of 500-700, an increase of only 1.5 times from about 350-450. It shows the trend of slower increase in costs for the cases with interactions when ρ goes from low to medium than that without interaction. Another observation from both figure 19 and 20 is that the curve goes upwards as the average standard deviation (or variance if the values are squared) of the demands for each retailer. It is quite easy to explain because the higher standard deviation values at the demand side, the more possible that one or more of the retailers will run under/over stock and raise the

need for sharing stock. Using the *IsShared()* test (the programming logic that tests whether the two agents may share the information or not), it is more likely the interaction is granted. Thus, the costs will be lower at the higher standard deviation level.

The last case to examine is when the value of ρ is at the highest level (Figure 21). This case needs more explanation since the pattern of the curve has changed, or the dynamics of the system show significant difference. Firstly, we notice the range costs of *cor_N* (no relationship) keeps its growth trend with values more than tripled from those at the medium ρ level at about 1580-1590. This brings the *cor_N* curve to the very top for all the cases investigated.

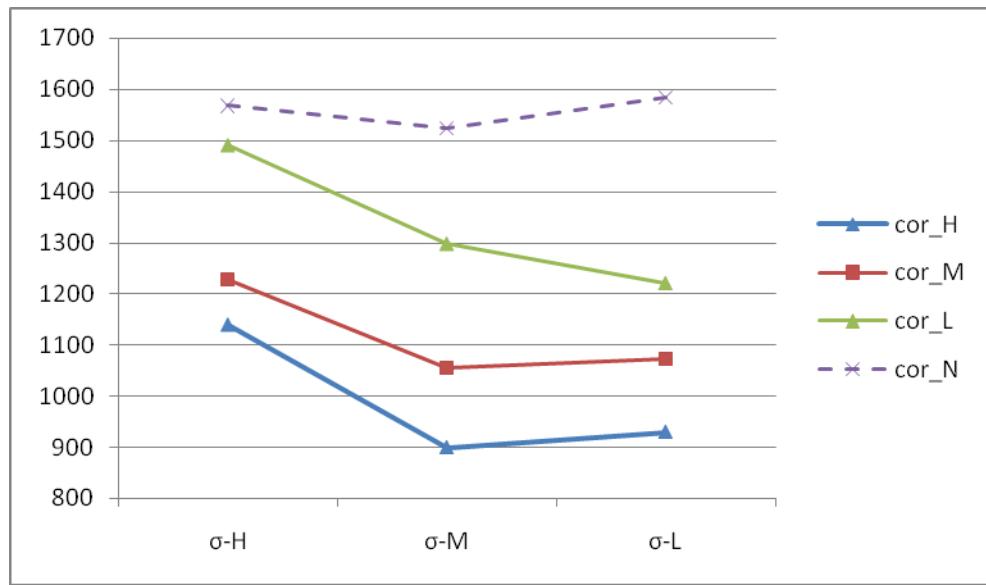


Figure 21 The Levels of Costs When ρ is High

Secondly, the curve patterns have changed. In the other cases, we notice that the curve increase continuously when ρ is of low and medium values. This is because when ρ gets greater and close to one, for the AR(1) demand process, the variance of the demand:

$$D_t = d + \rho D_{t-1} + \varepsilon_t,$$

can easily be derived as

$$Var(D_t) = \sigma^2 / (1 - \rho^2)$$

When ρ gets greater, it is obvious $Var(D_t)$ will increase to a very large value because $(1 - \rho^2)$ gets smaller. In fact

$$\lim_{\rho \rightarrow 1} Var(D_t) = \infty$$

While

$$\lim_{\rho \rightarrow 0} Var(D_t) = \sigma^2$$

At very high variances, the benefits from the interactions may not offset the volatility of the demand processes at the retailer level. Thus due to the high variance, though all are reduced to the levels lower or much lower than the one without interaction, the average costs for the high average deviation values are still very high as shown in the chart.

However, when the average demand deviation is lowered to medium and low levels, the curves resume the original patterns.

As we mentioned in the model description, we have three (relationships) 3x3 factorial design, which give us a three-dimension type of experimental design. We will get nine two-dimension tables as the results. So far we have examined three of them, which are

those that look at the impact of ρ . Now we turn our attention to the other two parameters, or independent variables that may affect the total average costs at the supplier side.

First we examine the relations between the ρ and σ at the fixed retailer relationship levels. The sets of experiments follow precisely the same procedures as described earlier, except we change the parameter relationship values rather than the values of ρ . Again we first look at the aggregate average costs at each level in one chart.

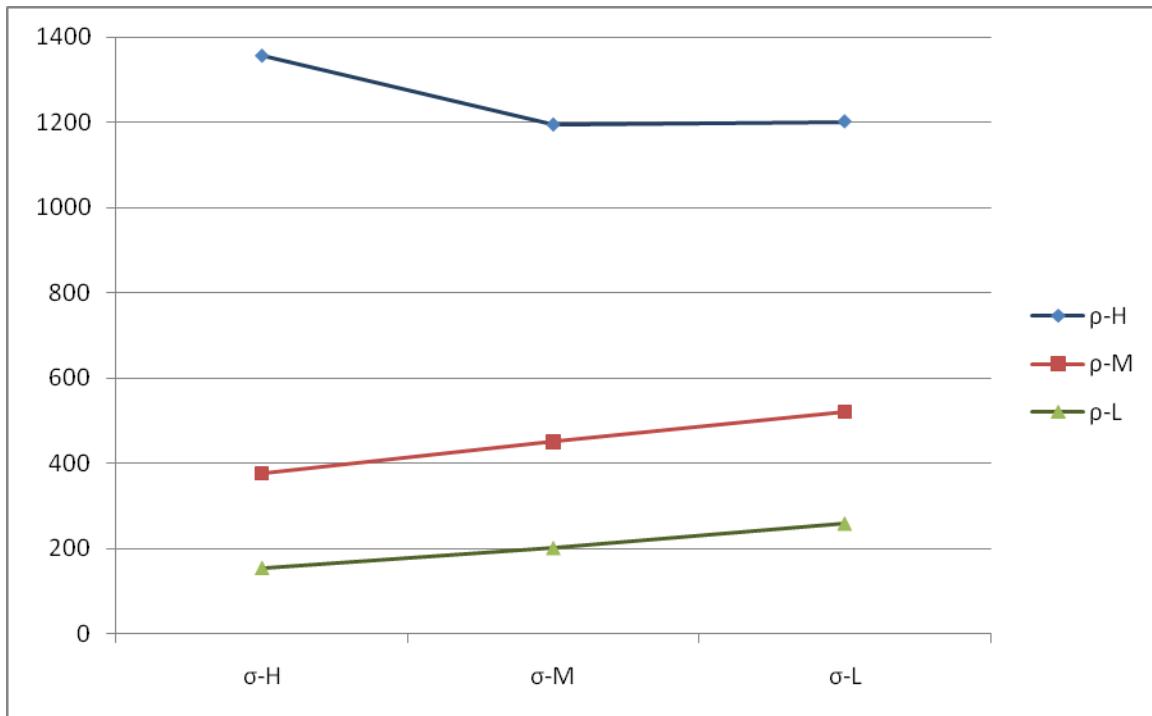


Figure 22 Effect of Information Sharing with Respect to σ and ρ

Figure 22 is a very consistent result with those shown in Figure 17. First, we see that the cost savings are significantly lower at the high and medium ρ levels, in addition to their obvious advantages to the non-relationship case as marked with purple (viewable if printed in color) x of dotted line above. The savings do not present when ρ level is low.

We have used an example earlier to explain reason for the latter case. Again we notice that the trend of the savings along the X-axis (standard deviation) is not continuously increasing. The best scenario according to our experiment is that the cost savings are maximized when ρ level is high and standard deviation of the demand process is at the medium level when relationship in general is in place.

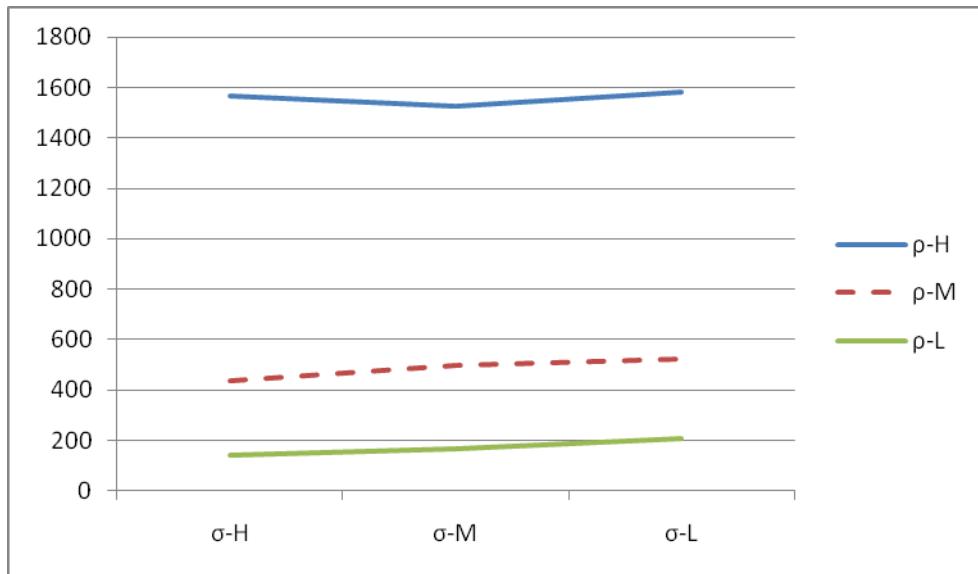


Figure 23 The Levels of Costs without no Interactions

Let us look now at the individual levels of relationship. The base case is the one without relationship as shown in Figure 23 above. This is the same pattern as found in Lee et al. (2000): at the fixed standard deviation level, the total average cost is larger when ρ level is higher. We show again this in Figure 5-8. The authors data are generated by fixing the standard deviation level to 0.7 and varying the value of ρ from zero to 0.9 (as noted early, if the value of ρ is too close to 1, the stock cost will be extremely high.). Clearly, increasing the value of ρ at a fixed σ level will increase the average stock cost. Our experiments exactly follow such a pattern. Again as our external validation has proved,

the agent based model that we have designed and implemented is capable of producing meaningful results as those previously observed and presented in the literature. However, the previous research has a large gap. They only examine the single point scenario and have not investigated multiple points scenarios in order to find the trend when standard deviation varies value from high to low. Our factorial design fills this gap.

Figure 1 Impact of ρ on Average Manufacturer's On-hand Inventory

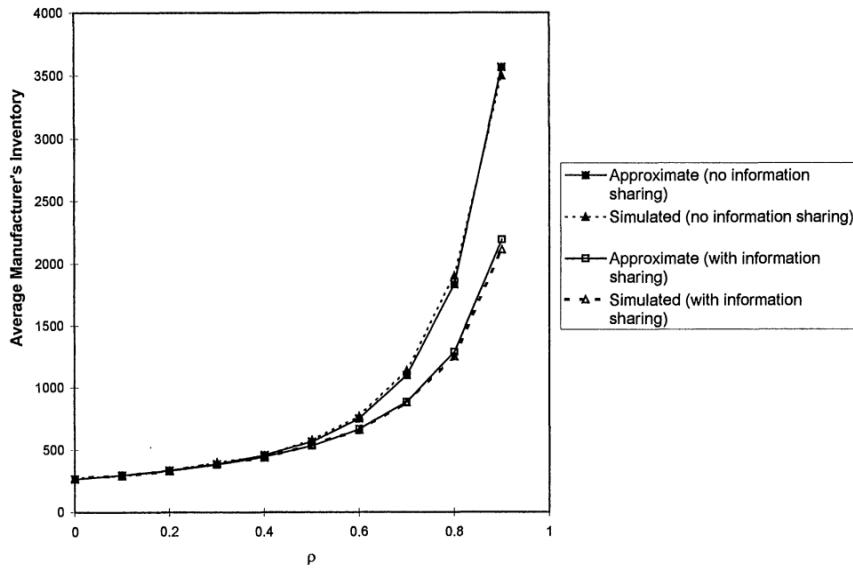


Figure 24 Impact of ρ on Average Stock Costs, Adapted from Lee et al. (2000)

Next we look at the cases with different levels of relationships. We list them in Figure 25 to 27. It is not a surprise that at all relationship values, the general pattern does not change: when level of standard deviation changes from high to low, at the low and medium levels of ρ , the average stock total costs go upward. This is because the benefits of the interaction are more significant at the experiments with higher volatility than otherwise. The difference is found for the curve of higher ρ . Though the general pattern does not change, i.e. the higher ρ produces higher average total costs, the trend however

goes straight downwards at low relationship level, goes downwards from σ_H to σ_M and is flat to σ_L at medium relationship level, and goes back to normal at high relationship level from σ_M to σ_L . As the reasons given in the previous discussion, when ρ is at its high level and standard deviation is high, the benefits of the relationship may not offset the higher costs due to higher volatility from the demand process, and thus consistently we see higher average cost output from the experiments. Nevertheless, when standard deviation goes lower, the interactions start to show the advantage and correct the curve back to normal.

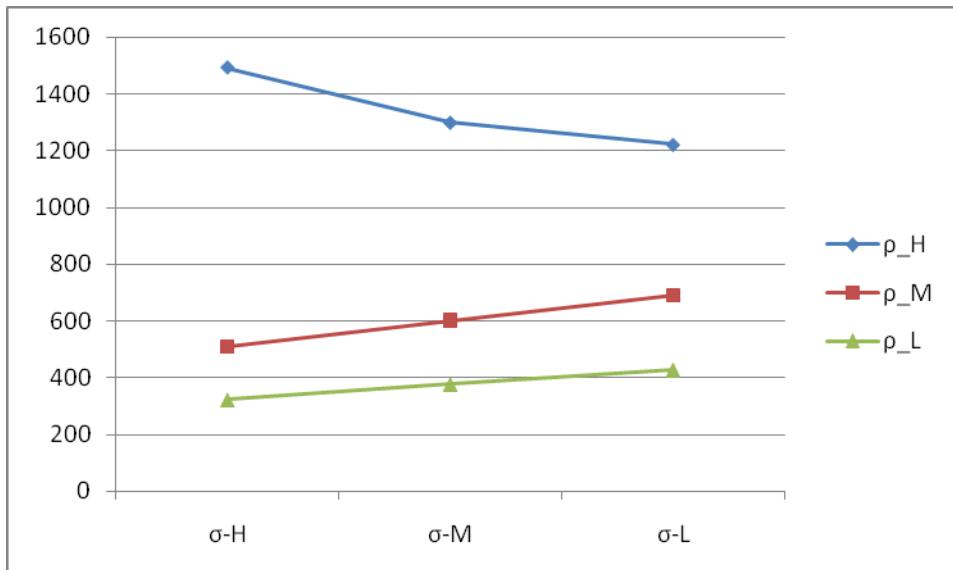


Figure 25 The Average Costs Curve when Relationship is Low

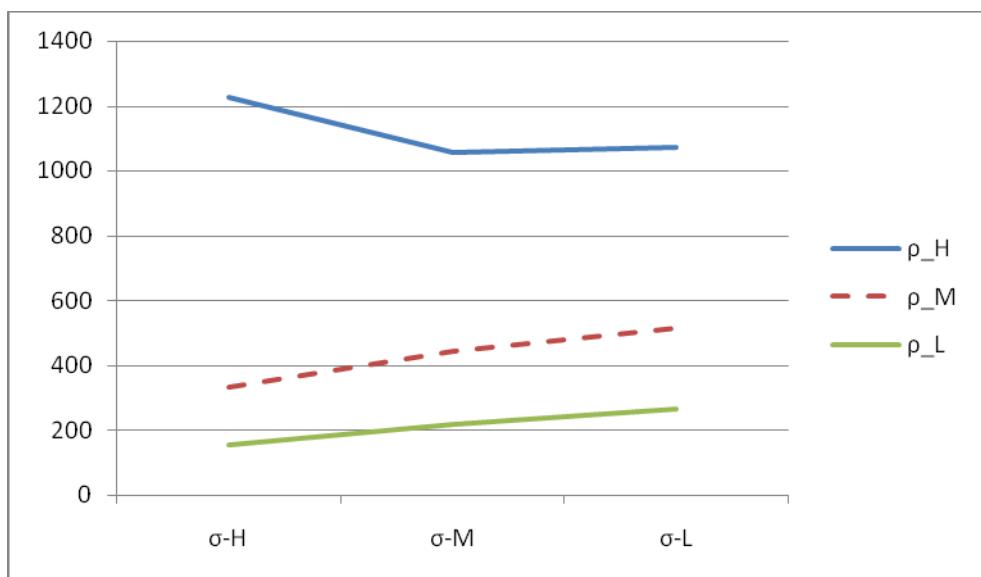


Figure 26 The Average Costs Curve when Relationship is Medium

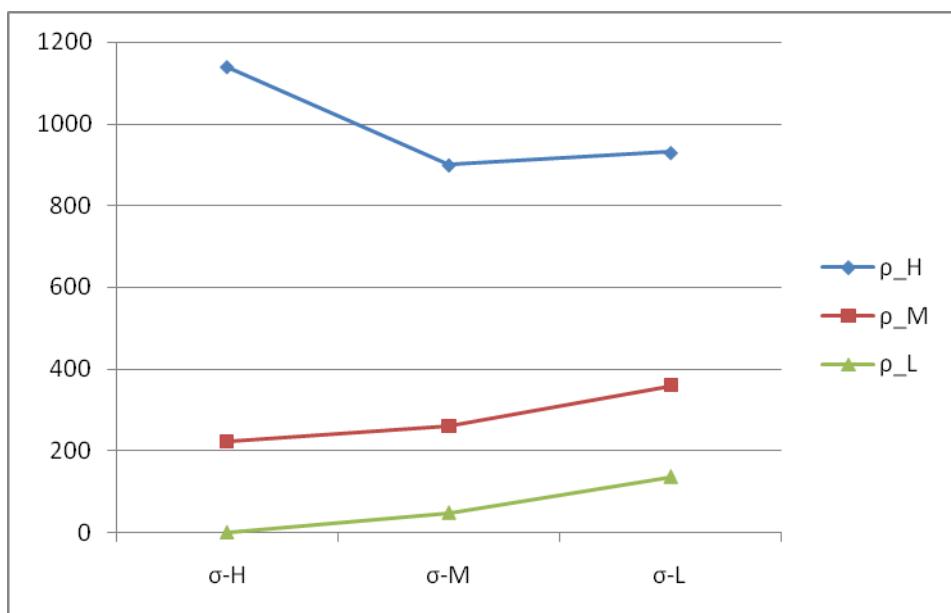


Figure 27 The Average Costs Curve when Relationship is High

The last set of runs is to fix the values of standard deviation and examine the relations between the interactions and the value of ρ . In fact this set of experiments just verified our observations and analysis above.

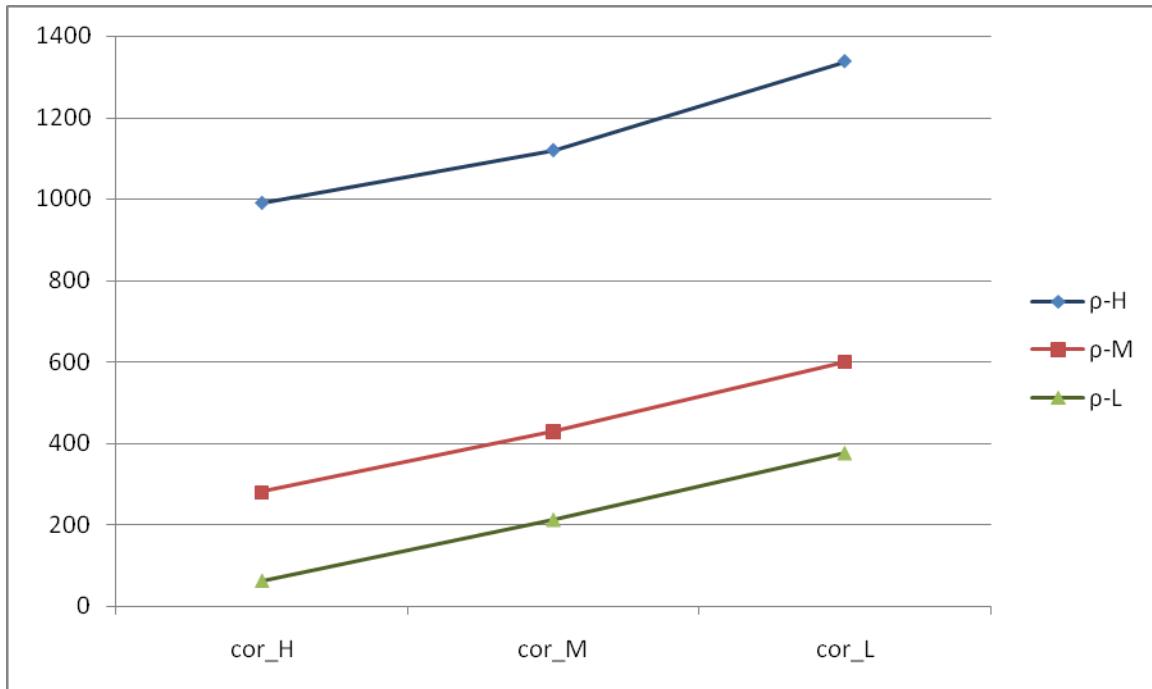


Figure 28 Effect of Information Sharing with Respect to Relationships and ρ

Figure 28 is the aggregate average costs chart that shows the relations between ρ and interactions. Three levels of standard deviation are used in the experiments and they are averaged to create this chart. Recall Figure 24 adapted from Lee et al. (2000) that at fixed demand standard variance or standard deviation, the total inventory costs goes up when the values of ρ increases, and the larger the value of ρ , after it passes about medium, the much faster the costs grow. We can see this pattern here. When we look at the distance between each curve, from smaller value of ρ to the higher ρ , at any points of relationship, including the one without relationship, the curves line one above the other

corresponding to the values of ρ . A noticeable much bigger gap is found between the medium ρ curve and the high ρ curve, indicating the fact of the faster growth of costs when the medium point is passed, exactly as shown in Lee et al. (2000). The curves also very nicely lay out the fact that at a fixed level of standard deviation, the lower the value of relationships, the higher the total average costs at the supplier side.

Figures 29 to 31 are the breakdowns of this aggregate chart. These charts are very similar in telling the stated analysis above validate the agent based model which follows the general theory in supply chain management as well as display the improvement in cost saving when interactions are put into action.

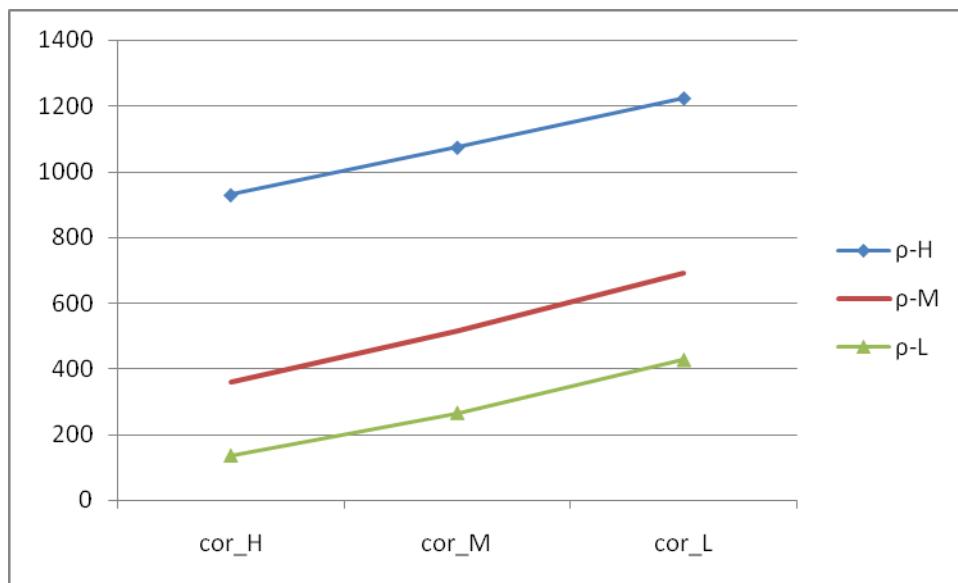


Figure 29 The Average Cost Curve with Low Standard Deviation

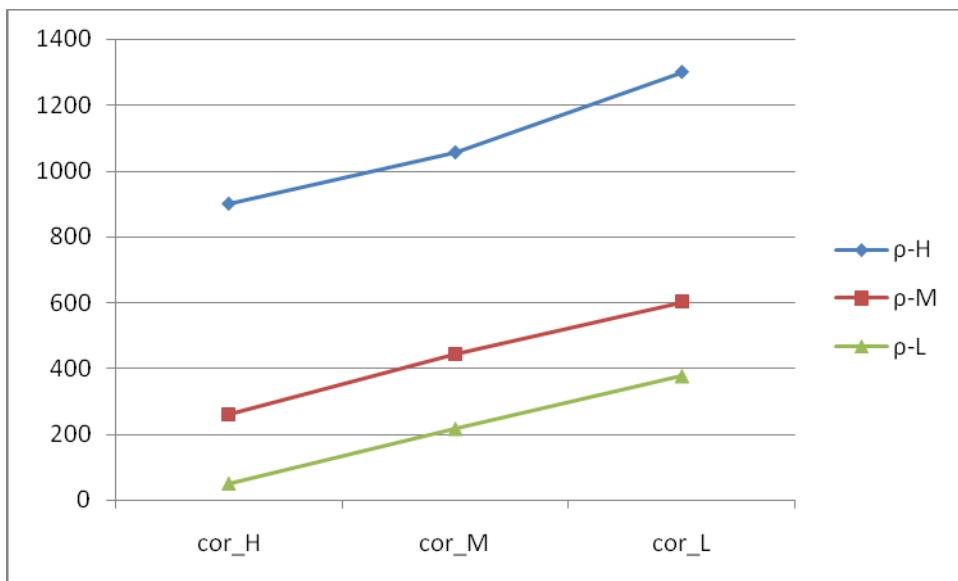


Figure 30 The Average Cost Curve with Medium Standard Deviation

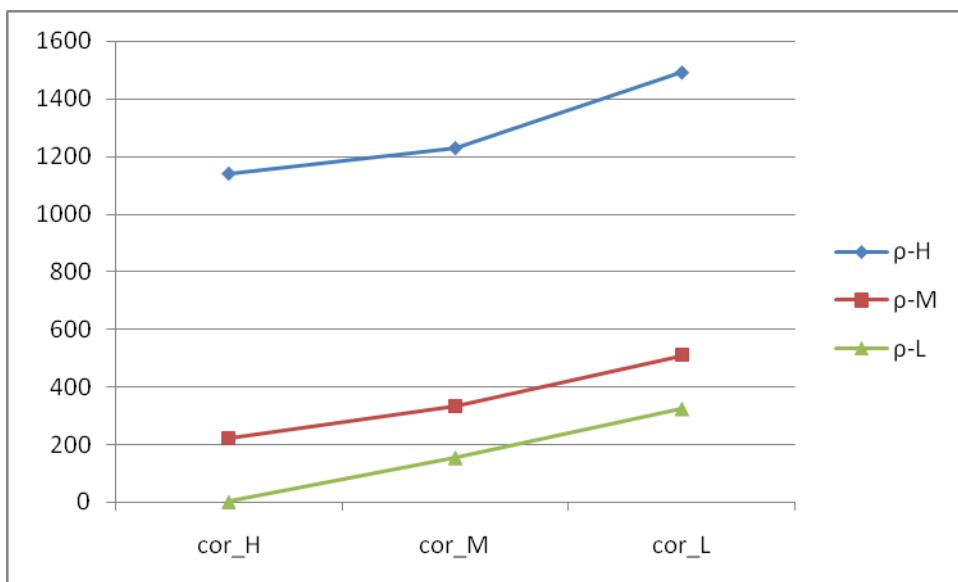


Figure 31 The Average Cost Curve with high Standard Deviation

This concludes the experimental design. Our factorial design has meaningfully produced the results that follow the theories in previous research and serves its purpose as the tool for internal validation.

Chapter 5 Conclusions and Future Research

Topics in supply chain management have been widely researched and the issues of information sharing are getting more attention in the age when information technology reaches such a height at the speed that has never been fully and readily taken advantage of. The information technology nowadays is able to and makes it possible to have the information or data transferred or communicated fast as well as accurate. However, the advancement of the technology does not guarantee the applications. Research is necessary to examine the impact of the information technology as well as the effects of information sharing in more detail, if not in more renovating ways to fully utilize the information technology that researchers twenty years ago were dying for. Artificial intelligence (AI in short and more commonly) is one of the areas that benefit from information technology. One of the major ideas of AI, from software engineering perspective only, is to create the applications that are embedded with advanced logic and algorithms to "think" like a human does. This has, still, been a far-reaching goal for hundreds and thousands of scientists and researchers since the term first coined by John McCarthy in 1956. The concept of artificial intelligence may be traced back to the time in the Greek myths of Hephaestus and Pygmalion which incorporated the idea of intelligent robots (such as Talos) and artificial beings (such as Galatea and Pandora)⁹. Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an intelligent agent is a system that perceives its environment and takes actions which maximize its chances of success. However, agents are not very often, yet,

⁹ See McCorduck, Pamela (2004), *Machines Who Think* (2nd ed.), Natick, MA: A. K. Peters, Ltd., pp. 4-5. Of course, the modern research only appeared in last century. The first working AI programs were written in 1951 to run on the Ferranti Mark I machine of the University of Manchester: a checkers-playing program written by Christopher Strachey and a chess-playing program written by Dietrich Prinz.

used in social science studies. Nevertheless, the literature in the recent years find the dramatic growth in papers published using agent based modeling and agent based simulation.

This thesis attempts to be part of the recent trend and push the research of applying intelligent agents in social science, in particular the supply chain management.

5.1 Summary and Conclusions

In this essay we have demonstrated that interactions between retailers do have plausible consequences on the supply chain performance. In particular, we look at the total average costs at the supplier side and find significant improvement in cost reduction as the interactions among the retailers increase. We applied the agent based model as the research methodology and through the external validation, We confirm that the new method produces the exact results as those from the previous research. We use agents to represent the firms in the supply chain and program the behaviors for each agent by applying the existing mathematically proved functions. We also use the agents to demonstrate the bullwhip effect. We have shown that without information sharing, the supplier needs to periodically update the mean and variance of demand based on observed demand data through orders received from retailers, then the variance of the orders placed by the supplier will be greater than the variance of the demand. More importantly, providing sufficient information of the underlying demand to the supplier can significantly reduce this increase in variability. However, we have also shown that the bullwhip effect will exist even when demand information is shared by all and all use the same forecasting technique and inventory policy.

To further explain this point, consider the retailers. Notice that if the retailers knew the mean and the variance of customer demand, then the orders placed by the retailers would be exactly equal to the customer demand and there would be no increase in variability.

However, in the model we implemented, even though the retailers have complete knowledge of the observed customer demands, they must still estimate the mean and variance of demand. As a result, the supplier sees an increase in variability.

When interactions are considered, the dynamics of the supply chain system change. In our model implementation, the information sharing between the retailers is considered as a type of interaction through communications. Such interactions will affect the relationships between the retailers. Therefore we use the values of the relationships as the proxy to examine how information sharing, thus interactions, may have impact on the supply chain performance. The model reveals the positive relationships between the two variables: interactions and performance.

Starting from the review of literature that develops different theories and models in supply chain management, we explore the different categories in model development. The supply chain management is mainly researched in four different categories, namely, information models, operational relationships models, and the contracting relationships models. By looking into each category, we summarize the theoretical development in the research in each area and find that, though agent based modeling has been well emphasized in many other fields of social sciences, very few are found in dealing with the problems in supply chain management.

Our model is coded using Java programming language, which models everything in the world as abstract objects and classes (groups of objects of the same nature, indeed; and they are objects themselves). An object in Java is like the real object in life. For example, we may define a car with wheels, body, engine, color, etc. in life, and in Java we define the exact same things. Another very important feature of Java, a Object Oriented Programming language as often called, is that each object may have behaviors coded as methods (similar to the functions and procedures in other programming languages). The model developer may design all types of logic and algorithms, as long as he may find and develop, and then the objects are able to “think” as close to our “thinking” as the program permits. This gives a great advantage to the model research world as the new tool to mimic the real world and by using the information technology and computing power to possibly create the models so real that other methods are not capable of doing. The similarity of the programming language to the real life concepts makes it a perfect candidate for implementing the models that the individual actors are important rather than just to look at the general outputs of hypothetical groups.

In addition to the external validation, we have also the experimental design. It is a 3 3x3 factorial design. The variables include the average total costs of the supplier, as well as the AR(1) demand process coefficient, the variance of the demand process, and the value of the relationships between retailers. The last variable is the proxy for the interaction, i.e., the information sharing. The results of the experimental design mainly follow the theoretical conclusions on the effects of the demand process coefficient as well as the variance of the demand process. However the improvement in the cost saving is

noticeable and we conclude that the information sharing between the retailers (horizontally) does have positive impact on the supply chain performance.

One of the major contributions of this essay is that unlike the previous researchers who examine the supply chain in a vertical way, we look at it in both ways, vertical and horizontal. This gives us a more complete picture of the system, though still not fully complete, as mentioned later in this chapter. Our approach to the system opens the possibilities of not just investigating the simple point to point interactions, but point to group of interacting points.

This essay would be incomplete if we did not mention several important limitations of our model and results. The main drawback of our model is the assumption that information sharing / interaction has linear relations with the relationships between the retailers. This is because of the lack of theory and empirical evidence. However, due to the scalability of the agent based model that we have developed, it is relatively easy if some new connections between the interaction and relationships is defined and coding into different software module is relatively easy and adding the new module to the model will be of an easy task comparing to the task of defining such links. Nevertheless, the change of the functions that define the links between the interaction and relationships may not change the model dynamic much as we can expect that the smoothed reordering curve due to the interactions may subsequently reduce the variance at the supplier side and thus provide cost reduction benefits. This is yet to be tested. Another limitation of our model is the fact that we are using the optimal order-up-to policy, as defined in Lee et al. (2000), but rather the policy defined in Chen et al. (2000). The order-up-to policy is

widely used in literature and made the assumption that the retailer has all the information about the underlying demand, including the values of the ρ and σ . These assumptions are noticeably not very practical as to the difficulties of obtaining such information. Chen et al. (2000) use the moving average to compute the mean lead time demand and standard deviation of the lead time forecast error. It is important to point out that policies of the latter form are frequently used in practice and the moving average is one of the most commonly used forecasting techniques in practice. In the future research, we may use this function to test our model performance.

Finally the assumption of one supplier and three retailers could not clearly capture many of the complexities involved in real-world supply chain. Although our model has been much more complex than most of the models that are developed in literature, which most commonly assume one supplier and one (or two) retailers because it is not necessary to add more retailers for the interactions between them are ignored. Although still having not examined the impact of more than one supplier (more than one supplier will introduce a much complex pattern to the dynamics of the supply chain as all the buy-seller relationships will start to take effect), our model has successfully experimented and demonstrated the significant improvement in supply chain performance due to the interactions between retailers.

5.2 Future Research

Chen et al. (2000) claim that the overstock will not affect the variability of the retailer or supplier. They verified their claim by a simulation. However, the authors have not shown how the stocks are handled if not sent directly back to the upstream without

charge. I have some doubts of their findings and propose that the variability may vary if the stocks are handled differently.

The literature has not examined the impact of different demand (or sales) forecasting functions to the order variability, inventory level, as well as the total costs. Some empirical tests may be useful to find the best forecast functions so that the agent based model created may produce more realistic results. This is the advantage of the use of intelligent agents in the model that give the model the adaptability to changes and learn from the different sources as external inputs, in addition to the past errors, just as a real firm does (at least as close to this goal as possible).

The literature, in order to make the analysis simple and mathematically traceable, totally ignored the possible impact of the backorders to the total costs at each level of the supply chain; and of course they have not examined the approaches of how the backorders are handled and what they may affect the inventory level as well as the total costs. Our thesis has noticed this gap and, other than unrealistically ignoring them, has investigated the simplest approach of handling the overstock by fulfilling them in the next period and additional quantity is added to the forecasted reorder amount.

Last but not the least, the literature on information sharing in the supply chain may require great efforts to fill the gap of examining what types of information should be shared. There is a great momentum behind the growth in supply chain model research. Stepping back by taking a broad view of the literature on the said research, however, causes a degree of discomfort. The vast majority of articles focus on finding the optimal ordering quantities, upper and lower bounds of certain algorithms, or discuss the

equilibrium points, while little attention is devoted to the types of information to be shared and how different types of information may impact the supply chain performance and in what degree.

5.3 Software Packages for Building and Running the Agent-Based Models

REPAST (Recursive Porous Agent Simulation Toolkit) (<http://repast.sourceforge.net/>), originally developed at University of Chicago, now maintained in the Argonne National Laboratory, is a widely used, free, and open-source, agent-based modeling and simulation toolkit. Three Repast platforms are currently available, each of which has the same core features but a different environment for these features. Repast Simphony (Repast S) extends the Repast portfolio by offering a new approach to simulation development and execution. Repast is one of the most used agent based simulation frameworks and is under very active updates. It has the root of the SWARM toolkit as introduced below. The current version is Repast Simphony 1.0. The Repast software is available to the general public under GNU licensing terms.

SWARM (http://www.swarm.org/wiki/Main_Page) is a software package for multi-agent simulation of complex systems, originally developed at the Santa Fe Institute (<http://www.santafe.edu>). Swarm is intended to be a useful tool for researchers in the study of agent based models. Swarm software comprises a set of code libraries which enable simulations of agent based models to be written in the Objective-C or Java computer languages. These libraries will work on a very wide range of computer platforms. The basic architecture of Swarm is the simulation of collections of concurrently interacting agents: with this architecture, we can implement a large variety

of agent based models. The Swarm software is available to the general public under GNU licensing terms. Every year SWARM hosts (together with Repast) the Swarm Fest conference in Chicago.

Netlogo (<http://ccl.northwestern.edu/netlogo/>) is less popular than the above two. It still has its place in the applications. NetLogo is a multi-agent programming language and integrated modeling environment. NetLogo was designed in the spirit of the Logo programming language to be "low threshold and no ceiling," that is to enable easy entry by novices and yet meet the needs of high powered users. The NetLogo environment enables exploration of emergent phenomena. It is particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of independent "agents" all operating concurrently. This makes it possible to explore the connection between the micro-level behavior of individuals and the macro-level patterns that emerge from the interaction of many individuals. NetLogo was developed at Northwestern University and has been funded by the National Science Foundation and other foundations.

Bibliography

Anand, Krishnan and Mendelson, Haim (1997), "Information and Organization for Horizontal Multimarket Coordination", *Management Science*, Vol. 43, No. 12, pp. 1609-1627.

Aoki, Masahiko; Gustafsson, Bo; and Williamson, Oliver (1990) (Editors), "The Firm as a Nexus of Treaties", Swedish Collegium for Advanced Study in the Social Sciences series, Sage, London, UK.

Axelrod, Robert; and Hamilton, William D. (1981), "The Evolution of Cooperation", *Science*, No. 211, pp. 1390-1296.

Axelrod, Robert (1984 and 2006), "The Evolution of Cooperation", Basic Book, New York.

Axelrod, Robert (2005), "Advancing the Art of Simulation in the Social Sciences", in Handbook of Research on Nature Inspired Computing for Economy and Management, Jean-Philippe Rennard (Ed.), Hersey, PA: Idea Group.

Baiman, Stanley; Fischer, Paul E.; Rajan, madhav V. (2001), "Performance Measurement and Design in Supply Chains", *Management Science*, Vol. 47, No. 1, pp. 173-188.

Bakos, Yannis (1998), "The Emerging role of electronic marketplaces on the Internet", *Communications of the ACM*, Vol. 41, August, pp. 35-42.

Bakos, Yannis, and Brynjolfsson, Erik (1999), "Bundling Information Goods: Pricing, Profits, and Efficiency", *Management Science*, Vol. 45, No. 12, pp. 1613-1630.

Balmer, Michael; Nagel, Kai; and Raney, Bryan (2004), "Large-Scale Multi-Agent Simulations for Transportation Applications", *Intelligent Transportation Systems*, Vol. 8, pp. 205-221.

Barney, Jay (1991), "Firm Resources and Sustained Competitive Advantage", *Journal of Management*, Vol. 17, No. 1, pp. 99-120.

Banker, Rajiv D.; Kalvenes, Joakim; Patterson, Raymond A. (2000), ""information technology, Contract Completeness, and Buyer-Supplier Relationships", the 21st International Conference on Information Systems (ICIS-00), Brisbane, Australia, 2000

Barr, Dale J. (2004), "Establishing Conventional communication Systems: Is Common Knowledge Necessary?", *Cognitive Science*, Vol. 28, pp. 937-962.

Barringer, Bruce; and Harrison, Jeffrey (2000), "Walking a Tightrope: Creating Value through Inter-organizational Relationships", *Journal of Management*, Vol. 26, No. 3, pp. 367-403.

Batty M.; Dodge M.; Jiang B.; and Smith A. (1999), "Geographical information systems and urban design", In: Stillwell J., Geertman S. and Openshaw S. (eds), *Geographical Information and Planning*, Springer, Heidelberg, pp. 43-65.

Bechtel, Christian; and Jayaram, Jayanth (1997), "Supply Chain Management: A Strategic Perspective", International Journal of Logistics Management, Vol 8 No 1, 1997, p. 15-34.

Bernstein, Fernando, and Federgruen, Awi (2005), "Decentralized Supply Chains with Competing Retailers Under Demand Uncertainty", Management Science, Vol. 51, No. 1, pp. 18-29.

Bertalanffy, Ludwig Von (1968), General System Theory: Foundations, Development, Applications.

Borys, Bryan; and Jemison, David (1989), "Hybrid Arrangements as Strategic Alliances: Theoretical issues in organizational combinations", Academy of Management Review, Vol. 14, No. 2, pp. 234.

Bowersox, D. J. (1969), "Readings in Physical Distribution Management: the Logistics of Marketing". Eds. Bowersox, D.J.; la Londe, B.J.; and Smykay, E.W.; New York: MacMillan

Bourland, Karla E.; Powell, Stephen G.; and Pyke, David F. (1996), "Theory and Methodology: Exploiting Timely Demand Information to Reduce Inventories", European Journal of Operational Research, Vol. 92, pp. 239-253.

Bowersox, D. J. (1969), "Readings in Physical Distribution Management: the Logistics of Marketing". Eds. Bowersox, D.J.; la Londe, B.J.; and Smykay, E.W.; New York: MacMillan.

Brereton, Pearl (2004), "The Software Customer/Supplier Relationship",
Communications of the ACM, Vol. 47, No. 2, pp. 77-81.

Brooks, R. A. (1991a), "Intelligence without reason", in: Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, Australia, pp. 569-595.

Cachon, Gerard P. (1999), "Managing Supply Chain Demand Variability with Scheduled Ordering Policies", Management Science, Vol. 45, No. 6, pp. 843-856.

Cachon, Gerard P. and Fisher, Marshall (2000), "Supply Chain Inventory Management and the Value of Shared Information", Management Science, Vol. 46, No. 8, pp. 1032-1048.

Cachon, Gerard P., and Zipkin, Paul H. (1999), "Competitive and Cooperative Inventory Policies in a Two-State Supply Chain", Management Science, Vol. 45, No. 7, pp. 936-953.

Cachon, Gerard P. (2001), "Stock Wars: Inventory Competition in a Two-Echelon Supply Chain with Multiple Retailers", Management Science, Vol. 49, No. 5, pp. 658-674.

Cachon, Gerard P. (2004), "The Allocation of Inventory Risk in a Supply Chain: Push, Pull, and Advance-Purchase Discount Contracts", Management Science, Vol. 50, No. 2, pp. 222-238

Cachon, Gerard P., and Lariviere, Martin A. (2001), "Capacity Choice and Allocation: Strategic Behavior and Supply Chain Performance", Management Science, Vol. 45, No. 8, pp. 1091-1108.

Cachon, Gerard P., and Lariviere, Martin A. (2001), "Contracting to Assure Supply: How to share Demand Forecasts in a Supply Chain", Management Science, Vol. 47, No. 5, pp. 627-646.

Cachon, Gerard P., and Lariviere, Martin A. (2005), "Supply Chain Coordination with Revenue-Sharing Contracts: Strengths and Limitations", Management Science, Vol. 51, No. 1, pp. 30-44.

Chen, Fangruo (1998), "Echelon Reorder Points, Installation Reorder Points, and the Value of Centralized Demand Information", Management Science, Vol. 44, No. 12, Part 2 of 2, pp. S221-S234.

Chen, Frank; Drezner, Zvi; Ryan, Jennifer K.; and Simchi-Levi, David (2000), "Quantifying the Bullwhip Effect in a Simple Supply Chain; The Impact of Forecasting, Lead Times, and Information," Management Science, Vol. 46, No.3, pp. 436-443.

Chetty, Sylvie; and Holm, Desiree Blankenburg (2000), "Internationalisation of small to medium-sized manufacturing firms: a network approach", International Business Review, Vol. 9, pp. 77-93

Clarke, Richard (1983), "Collusion and the Incentives for Information Sharing", The Bell Journal of Economics, Vol. 14, pp. 383-394.

Cooper, Martha C.; Lambert, Douglas M.; and Pagh, Janus D. (1997), "Supply Chain Management: More Than a New Name for Logistics", International Journal of Logistics Management, Vol 8 No 1, 1997, p. 1-14.

Corbett, Charles J.; DeCroix, Gregory A. (2001), "Shared-Savings Contracts for Indirect materials in Supply Chains: Channel Profits and Environmental Impacts", Management Science, Vol. 47, No. 7, pp. 881-893.

Corbett, Charles J.; DeCroix, Gregory A.; Ha, Altert Y. (2005), "Optimal Shared-Savings contracts in Supply Chains: Linear Contracts and Double Moral hazard", European Journal of Operational Research, Vol. 163, pp. 653-667.

Cravens, David; and Shipp, Shannon (1993), "Analysis of Co-operative Interorganizational Relationships, Strategic Alliance Formation, and Strategic Alliance Effectiveness", Journal of Strategic Marketing, Vol. 1, No. 1, pp. 55-70.

Craves, Stephen (1999), "A Single-Item Inventory Model for a Nonstationary Demand Process", Manufacturing & Service Operations Management, Vol. 1, No. 1, pp. 50-61.

CSCMP (2005), <http://www.cscmp.org/Website/Resources/Terms.asp>

Darl, Charles; Remolina, Emilio; Ong, Jim (2005), "Distributed Troubleshooting Agents", in: Proceedings of the 2nd International Conference on Autonomic computing (ICAC05).

Dewan, Sanjeev, and Mendelson, Haim (1990), "User Delay Costs and Internal Pricing for a Service Facility", Management Science, Vol. 36, No. 12, pp. 1502-1517.

Donohue, Karen L. (2000), "Efficient Supply Contracts for Fashion Goods with Forecast Updating and Two Production Modes", *Management Science*, Vol. 46, No. 11, pp. 1397-1411.

Drogoul, Alexis, and Ferber, Jacques (1992), "Multi-Agent Simulation as a Tool for Modeling Societies: Application to Social Differentiation in Ant colonies", In: Proceedings of MAAMAW92: 4th European Workshop on Modeling Autonomous Agents in a Multi-Agent World, Italy.

El-Ansary, Adel; Stern, Louis (1972), "Power Measurement in the Distribution Channel", *Journal of Marketing Research*, Vol. 9, February, pp. 47-52.

Eppen, Gary d., and Iyer, Ananth V. (1997), "Backup Agreements in Fashion Buying – The Value of Upstream Flexibility", *Management Science*, Vol. 43, No. 11, pp. 1469-1484

Escudero, L. F.; Galindo, E.; Garcia, G.; Gomez, E.; and Sabau, V. (1999), "Schumann, a Modeling Framework for Supply Chain Management", *European Journal of Operational Research*, Vol. 119, pp. 14-34.

Ford, David (1980), "The Development of Buyer-Seller Relationships in Industrial Markets", *European Journal of Marketing*, Vol. 14, No. 5/6, pp. 339-353.

Forrester, Jay W. (1969), *Industrial Dynamics*, The MIT Press, Cambridge, Massachusetts.

Franklin, Stan, and Graesser, Art (1997), “Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents”, in: Muller, J.P., Wooldridge, M.J., and Jennings, N.R. (Eds.) Intelligent Agents III: Agent Theories, Architectures, and languages, Springer-Verlag, Berlin, pp. 21-35.

Gallego, Guillermo and Ozer, Ozalp (2003), “Optimal Replenishment Policies for Multiechelon Inventory Problems under Advance Demand Information”, Manufacturing & Service Operations Management, Vol. 5, No. 2, pp. 157-175.

Ganeshan, Ram; Jack, Eric; Magazine, M. J.; and Stephens, Paul (1999), “A Taxonomic Review of Supply Chain Management Research”, Quantitative Models for Supply Chain Management, Eds. Tayur, Sridhar; Ganeshan, Ram; and Magazine Michael, Boston: Kluwer Academic Publishers, pp. 839.

Garcia-Flores, Rodolfo, and Wang, Xue Zhong (2002), “A Multi-Agent System for Chemical Supply Chain Simulation and Management Support”, OR Spectrum, No. 24, pp. 243-370.

Gavirneni, Srinagesh; Kapuscinski, Roman; and Tayur, Sridhar (1999), “Value of Information in Capacitated Supply Chains”, Management Science, Vol. 45, No. 1, pp. 16-24.

Gilbert, Nigel (2007), “Agent-Based Modeling, Series: Quantitative Applications in the Social Sciences”, SAG Publications.

Gurnani, Haresh and Tang, Christopher S. (1999), "Optimal Ordering Decisions with Uncertain Cost and Demand Forecast Updating", Management Science, Vol. 45, No. 10, pp. 1456-1462.

Harland, C. M. (1996), "Supply Chain Management: Relationships, Chains and Networks", British Journal of Management, vol. 7, Special Issue, pp. S63-S80

Heide, Jan (1994), "Interorganizational Governance in Marketing Channels", Journal of Marketing, Vol. 58, No. 1, pp. 71-85.

Heide, Jan; and John, George (1990), "Alliances in Industrial Purchasing: The Determinants of Joint Action in Buyer-Supplier Relationships", Journal of Marketing Research, Vol. 27, February, pp. 24-36.

Hilton, Ronald (1981)"The Determinants of Information Value: Synthesizing Some General Results", Management Science, Vol. 27, No. 1, pp. 57-64.

Hitt, M. A.; Ireland, D. R.; and Hoskisson, R. E. (1999), Strategic Management, Cincinnati, OH: South-Western College Publishing, Chapter 1.

Houlihan, John B. (1985), "International Supply Chain Management", International Journal of Physical Distribution & Materials Management, Vol. 15, pp. 22-38.

Howard R.A. (1966), "Information Value Theory", Systems Science and Cybernetics, IEEE Transactions, Vol. 2, No. 1, pp. 22-26.

Hui, Pamsy P., and Beath, Cynthia M. (2001), "The IT Sourcing Process: A Research Framework", The Annual Meeting of the Academy of Management, Washington, DC.

Janssen, Marco A., Jager, Wander (2003), "Simulating market dynamics: Interactions between Consumer Psychology and Social Networks", Artificial life, Vol. 9, 343-356.

Jennings, Nicholoas R., Sycara, Katia, and Wooldridge Michael (1998), "A Roadmap of Agent Research and Development", Autonomous Agents and Multi-Agent Systems, Vol. 1, No. 1, pp. 7-38.

Johns, T. C., and Riley, D. W. (1984), "Using Inventory for Competitive Advantage through Supply Chain Management", International Journal of Physical Distribution & Materials Management, Vol. 15, pp. 16-26.

Johnson, M. Eric, and Whang, Seungjin (2002), "E-Business and Supply Chain Management: An Overview and Framework", Production and Operations management, Vol. 11, No. 4, pp. 413-423.

Kahn, James A. (1987), "Inventories and the Volatility of Production", The American Economic Review, Vol. 77, No. 4, pp. 667-679.

Kargl, Frank; Illmann, Torsten; and Weber Michael (1999), CIA – a Collaboration and Coordination Infrastructure for Personal Agents", DAIS 99, Helsinki, Finland.

Kenney, J. F. and Keeping, E. S. (1951) Mathematics of Statistics, Pt. 2, 2nd ed. Princeton, NJ: Van Nostrand.

- Kumar, Kuldeep (2001), "Technology for Supporting Supply Chain Management: Introduction", Communications of the ACM, Vol. 44, No. 6, pp. 58-61.
- La Londe, Bernard J., and Masters, James (1994), "Emerging Logistics Strategies: Blueprints for the next Century", International Journal of Logistics Management, Vol. 24, No. 7, pp. 35-47
- LeBaron, Blake (1998), "Technical Trading Rules and Regime Shifts in Foreign Exchange", In: Acar, E., Satchell, S. (Eds.), Advanced Trading Rules, Butterworth-Heinemann, London, pp. 5-40.
- LeBaron, Blake (2000), "Agent-Based Computational Finance: Suggested Readings and Early Research", Journal of Economic Dynamics & Control, Vol. 24, No. 5-7, pp. 679-702.
- LeBaron, Blake (2001), "Evolution and Time Horizons in an Agent-Based Stock Market", Macroeconomic Dynamics, Vol. 5, No. 2, pp. 225-254.
- Lancioni, R. A.; Smith, M. F.; and Oliva, T. A. (2000), "The Role of the Internet in Supply Chain Management", Industrial Marketing Management, Vol. 29, No. 1, pp. 45-56.
- Lau Jason S. K.; Huang, George Q.; and Mak, K. L. (2004), "Impact of Information Sharing on Inventory replenishment in Divergent Supply Chain", International Journal of Production Research, Vol. 42, No. 5, 919-941.

Laugley, C. J. (1992), "The Evolution of the Logistics Concept, in Logistics: The Strategic Issues, in Christopher, Martin (1992), Logistics: The Strategic Issues, Chapman and Hall, London, UK.

Lederer, Phillip J., and Li, Lode (1997), "Pricing, Production, Scheduling, and Delivery-Time Competition", Operations Research, Vol. 45, No. 3, pp. 407-420.

Lee, Hau L., and Billington, Corey. (1993), "Material Management in Decentralized Supply Chains", Operations Research, Vol. 41, No. 5, pp. 835-847.

Lee, Hau, Padmanabhan, V, and Whang, Seungjing (1997), "Information Distortion in a Supply Chain: the Bullwhip Effect", Management Science, Vol. 43, No. 4, pp. 546-558.

Lee, Hau, and Whang, Seungjing (2000), "Information Sharing in a Supply Chain", International Journal of Manufacturing Technology and Management, Vol. 1, No. 1, pp. 79-93.

Lee, Hau L.; So, Kut C.; and Tang Christopher S. (2000), "The Value of Information Sharing in a Two-Level Supply Chain", Management Science, Vol. 46, No. 5, pp. 626-643.

Lin, Fu-ren, Sung, Yu-wei, and Lo, Yi-pong (2005), "Effects of Trust Mechanisms on Supply-Chain Performance: A Multi-Agent Simulation Study", International Journal of Electronic Commerce, Vol. 9, No. 4, pp. 91-112.

Lopez-Sanchez, Maite; Noria, Xavier; Rodriguez, Juan A.; Gilbert, N.; and Schuster, S. (2004), “Multi-Agent Simulation Applied to On-Line Music Distribution market”, in Proceedings of the 4th International Conference on Web Delivering of Music.

Mahoney, Joseph; and Pandian, Rajendran (1992), “The Resource-Based View Within the Conversation of Strategic Management”, Strategic Management Journal, Vol. 13 No. 5, pp. 363.

Mayer, R. C., Davis, J. H., and Schoorman, F. D. (1995), “An Integrative Model of Organizational Trust”, Academy of Management Review, Vol. 20, No. 3, pp. 709-734.

Meixell, Mary J., and Gargeya, Vidyaranya (2005), Global Supply Chain Design: A Literature Review and Critique, Transportation Research Part E, Vol 41, pp. 531-550.

Mendelson, Haim, and Whang Seungjin (1990), “Optimal Incentive-Compatible Priority Pricing for the M/M/1 Queue”, Operations Research, Vol. 38, No. 5, pp. 870-883.

Mentzer, John T.; DeWitt, William; Keebler, James S.; Min, Soonhon; Nix Nancy W.; Smith, Carlo D.; and Zacharia, Zach G. (2001), “Defining Supply Chain Management”, Journal of Business Logistics, Vol. 22, No. 2, pp. 1-25

Metters, Richard (1987), “Quantifying the Bullwhip Effect in Supply Chain”, Journal of Operations Management, Vol. 15, pp. 89-100

Milgrom, Paul, and Roberts, John (1988), “Communication and Inventory as Substitutes in Organizing Production”, Scandinavian Journal of Economics, Vol. 90, No. 3, pp. 275-289.

Monczka, Robert; Trent, Robert; and Handfield, Robert (1998), "Purchasing and Supply Chain Management", Cincinnati, OH: South-Western College Publishing, Chapter 8.

Nahmias, Steven, Demmy, Steven W. (1981), "Operating Characteristics of an Inventory System with Rationing", *Management Science*, Vol. 27, No. 11, 1236-1245.

Nilsson, Nils (1998), *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann Publishers.

Novshek, William and Sonnenschein, Hugo (1982), "Fulfilled Expectations Cournot Duopoly with Information Acquisition and Release", *The Bell Journal of Economics*, Vol. 13, pp. 214-218.

Nwana, Hyacinth S (1996), "Software Agents: An Overview", *Knowledge Engineering Review*, Vol. 11, No. 3, pp. 205-244.

Oliver, R. Keith; and Webber, Michael D. (1982), "Supply Chain management: Logistics Catches up with Strategy", in Christopher, Martin (1992), *Logistics: the Strategic Issues*, Chapman and Hall, London, UK, pp. 63-75.

Overby, Jeffrey W., and Min Soonhong (2001), "International Supply Chain Management in an Internet Environment: A Network-Oriented approach to Internationalization", *International Marketing Review*, Vol. 14, No. 4pp. 392-420.

Padmanabhan, V. and Png, I.P.L. (1997), "Manufacturer's Returns Policies and Retail Competition", *Marketing Science*, Vol. 16, No. 1, pp. 81-94.

Parkhe, Arvind (1993), "Strategic Alliance Structuring: A Game Theoretic and Transaction Cost Examination of Interfirm Cooperation", *Academy of Management Journal*, Vol. 36, No. 4, pp. 794-829.

Poirier, C. C., and Bauer, M. (2000), "E-Supply Chain: Using the Internet to Revolutionize Your Business", Berrett-Koehler Publishers.

Poole, David; Mackworth, Alan & Goebel, Randy (1998), *Computational Intelligence: A Logical Approach*, Oxford University Press.

Potok, Thomas E.; Elmore, mark; Reed, Joel; and Sheldon, Frederick T. (2003), "VIPAR: Advanced Information Agents Discovering Knowledge in an Open and Changing Environment", in: *Proceedings of 7th World Multiconference on Systemics, Cybernetics and Informatics Special Session on Agent-Based Computing*, Orlando, FL, pp. 28-33.

Raghunathan, Srinivasan (2001), "Information Sharing in a Supply Chain: A Note on its Value when Demand Is Nonstationary", *Management Science*, Vol. 47, No. 4, pp. 605-610.

Raith, Michael (1996), "A General Model of Information Sharing in Oligopoly", *Journal of Economic Theory*, Vol. 71, pp. 260-288.

Raman, Ananth and Fisher, Marshall (1996), "Reducing the Cost of Demand Uncertainty through Accurate Response to Early Sales", *Operations Research*, Vol. 44, No. 1, Special Issue on New Directions in Operations Management (Jan. – Feb., 1996), pp. 87-99.

Scott, C., and Westbrook, R. (1991), "New Strategic Tools for Supply Chain Management", International Journal of Physical Distribution & Logistics Management, Vol. 21, pp. 22-33.

Russell, Stuart J. & Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, NJ: Prentice Hall.

Siguaw, Judy; Simpson, Penny; and Baker, Thomas (1998), "Effects of Supplier Market Orientation on Distributor Market Orientation and the Channel Relationship: The Channel Relationship: The Distributor Perspective", Journal of Marketing, Vol. 62, July, pp. 99-111.

Singh, Bahul, Salam, A. F., and Iyer, Lakshmi (2005), "Agents in E-Supply Chains: Realizing the Potential of Intelligent Informed intermediary-Based E-Marketplaces", Communications of the ACM, Vol. 48, No. 6, pp. 109-115"

Spitter, J. M.; Hurkens, C. A. J.; de Kok, A. G.; and Lenstra, J. K. (2005), "Linear Programming Models with Planned lead Times for Supply Chain Operations Planning", European Journal of Operational Research, Vol. 163, pp. 706-720.

Srinivasan, Kanna; Kekre, Sunder; Mukhopadhyay, Tridas (1994), " Impact of Electronic Data Interchange Technology on JIT Shipments", Management Science, Vol. 40, No. 10, pp. 1291-1304.

Sterman, John D. (1989), "Modeling Managerial Behavior: Misperceptions of Feedback in a Dynamic Decision Making Experiment", *Management Science*, Vol. 35, No. 3, pp. 321-339.

Stevens, Graham C. (1989), "Integrating the Supply Chains", *International Journal of Physical Distribution & Materials Management*, Vol. 8, No. 8, pp. 3-8.

Swaminathan, Jayashankar; Smith, Stephen F.; and Sadeh, Norman M. (1998), "Modeling Supply Chain Dynamics: A Multiagent Approach", *Decision Sciences Journal*, Vol. 29, No. 3, pp. 607-632.

Tan, Keah Choon (2000), "A Framework of Supply Chain Management Literature", *European Journal of Purchasing and Supply Management*, Vol. 7, pp. 39-48.

Terwiesch, Christian; Ren Z. Justin; Ho, Teck H.; and Cohen, Morris A. (2005), "An Empirical Analysis of Forecast Sharing in the Semiconductor Equipment Supply Chain", *Management Science*, Vol. 51, No. 2, pp. 208-220.

Thonemann, U. W. (2002), "Improving Supply-Chain Performance by Sharing Advance Demand Information", *European Journal of Operational Research*, Vol. 142, pp. 81-107.

Topkis, Donald M. (1968), "Optimal Ordering and Rationing Policies in a Nonstationary Dynamic Inventory Model with n Demand Classes", *Management Science*, Vol. 15, No. 3, pp. 160-176.

Tsay, Andy A. (1999), "The Quantity Flexibility Contract and Supplier-Customer Incentives", *Management Science*, Vol. 45, No. 10, pp. 1339-1358.

Tsvetovat, Maksim, and Carley, Kathleen M. (2004): "Modeling Complex Socio-technical Systems using Multi-Agent Simulation Methods". KI (www.kuenstliche-intelligenz.de), 18(2): 23-28.

Vives, Xavier (1984), "Duopoly Information Equilibrium: Cournot and Bertrand", Journal of Economic Theory, Vol. 34, pp. 71-94.

Wade, Michael; and Hulland, John (2004), "Review: The Resource-Based View and Information Systems Research: Review, Extension, and Suggestions FOR Future Research", MIS Quarterly, Vol. 28, No. 1, pp. 107. Walden, Eric A. (2000), "On the Structure and Function of Outsourcing Contracts: an Integrative Analysis of the Economics Behind Vendor-Client Relationships", Working Paper, MIS Research Center, University of Minnesota.

Whang, Seungjing (1992), "Contracting for Software Development", Management Science, Vol. 38, No. 3, pp. 307-324.

Whittaker, E. T. and Robinson, G. (1967), "Determination of the Constants in a Normal Frequency Distribution with Two Variables" and "The Frequencies of the Variables Taken Singly." §161-162 in The Calculus of Observations: A Treatise on Numerical Mathematics, 4th ed. New York: Dover, pp. 324-328.

Wooldridge, Michael, and Jennings, Nicholas R. (1995), "Intelligent Agents: Theory and Practice", Knowledge Engineering Review, Vol. 10, No. 2, pp. 115-152.

Zachman, J. A. (1987), "A Framework for Information Systems Architecture", IBM Systems Journal, Vol 26, No. 3, pp. 276 -292.

Zhang, Yiyi; Guo, Lei; and Georganas, Nicolas D. (2000), "AGILE: An Architecture for Agent-Based Collaborative and Interactive Virtual Environments", in: Proceedings of the Workshop on Application of Virtual Reality Technologies for Future Telecommunication Systems, IEEE Globecom 2000 Conference, San Francisco.

Zipkin, Paul (1989), "Critical Number Policies for Inventory Models with Periodic Data", Management Science, Vol. 35, No. 1, pp. 71-80.

Appendix The Java Code for the Model Implementation

File 1: model.score

```
<?xml version="1.0" encoding="UTF-8"?>

<score:SContext xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:score="http://scoreabm.org/score" label="sc" ID="sc" pluralLabel="scs">

  <attributes label="initialDemandAmount" ID="initialDemandAmount"
    pluralLabel="initialDemandAmounts" sType="INTEGER" defaultValue="100"
    units="unit"/>

  <attributes label="orderLag" ID="orderLag" pluralLabel="orderLags"
    sType="INTEGER" defaultValue="1" units="tick"/>

  <attributes label="Roo" ID="Roo" pluralLabel="Roos" sType="FLOAT"
    defaultValue="0.8"/>

  <attributes label="dValue" ID="dValue" pluralLabel="dValues" sType="INTEGER"
    defaultValue="100" units="unit"/>

  <attributes label="numAgent" ID="numAgent" pluralLabel="numAgents"
    sType="INTEGER" defaultValue="1"/>
```

```

<attributes label="GrowthRate" ID="GrowthRate" pluralLabel="GrowthRates"
sType="FLOAT" defaultValue="0.001" units="" />

<attributes label="Ratio12" ID="Ratio12" pluralLabel="Ratio12s" sType="FLOAT"
defaultValue="0" />

<attributes label="Ratio13" ID="Ratio13" pluralLabel="Ratio13s" sType="FLOAT"
defaultValue="0" />

<attributes label="Ratio23" ID="Ratio23" pluralLabel="Ratio23s" sType="FLOAT"
defaultValue="0" />

<attributes label="SharingCase" ID="SharingCase" pluralLabel="SharingCases"
sType="INTEGER" defaultValue="0" />

<attributes label="Sdev1" ID="Sdev1" pluralLabel="Sdev1s" sType="FLOAT"
defaultValue="30" />

<attributes label="Sdev2" ID="Sdev2" pluralLabel="Sdev2s" sType="FLOAT"
defaultValue="20" />

<attributes label="Sdev3" ID="Sdev3" pluralLabel="Sdev3s" sType="FLOAT"
defaultValue="10" />

<implementation package="repast.simphony.demo.sc" className="ContextCreator"
basePath="" />

<agents label="RetailerAgent" ID="retailerAgent" pluralLabel="RetailerAgents">

```

```
<implementation package="repast.simphony.demo.sc" className="RetailerAgent"/>

</agents>

<agents label="Moderator" ID="moderator" pluralLabel="Moderators">

<implementation package="repast.simphony.demo.sc" className="Moderator"
basePath="" />

</agents>

<agents label="testAgent2" ID="testAgent2" pluralLabel="testAgent2s">

<implementation className="testAgent2"/>

</agents>

<agents label="testAgent1" ID="testAgent1" pluralLabel="testAgent1s">

<implementation className="testAgent1"/>

</agents>

<agents label="RetailerAgent3" ID="retailerAgent3" pluralLabel="RetailerAgent3s">

<implementation package="repast.simphony.demo.sc" className="RetailerAgent3"
basePath="" />

</agents>
```

```
<agents label="SupplierAgent" ID="supplierAgent" pluralLabel="SupplierAgents">

    <implementation package="repast.simphony.demo.sc" className="SupplierAgent"/>

</agents>

<agents label="RetailerAgent2" ID="retailerAgent2" pluralLabel="RetailerAgent2s">

    <implementation package="repast.simphony.demo.sc" className="RetailerAgent2"/>

</agents>

<agents label="SCagent" ID="SCagent" pluralLabel="SCagents">

    <implementation className="SCagent"/>

</agents>

<projections xsi:type="score:SGrid" label="Grid" ID="grid" pluralLabel="Grids"/>

</score:SContext>
```

File 2: ContextCreator.java

```
package repast.simphony.demo.sc;

import repast.simphony.context.space.grid.GridFactoryFinder;
import repast.simphony.context.Context;
import repast.simphony.dataLoader.ContextBuilder;
import repast.simphony.space.grid.GridBuilderParameters;
import repast.simphony.space.grid.RandomGridAdder;
import repast.simphony.space.grid.WrapAroundBorders;

public class ContextCreator implements ContextBuilder<SCagent> {

    public Context<SCagent> build(Context<SCagent> context) {
```

```

int xdim = 30;

int ydim = 30;

//Change the code to the following to make the values customizable

//Need to add the variables to the score file.

//Parameters p = RunEnvironment.getInstance().getParameters();

//numberAgents = (Integer)p.getValue("numAgent");

//xdim = (Integer)p.getValue("gridWidth");

//ydim = (Integer)p.getValue("gridHeight");

GridFactoryFinder.createGridFactory(null).createGrid("Grid", context,
    new GridBuilderParameters<SCagent>(new
    WrapAroundBorders(),
    new RandomGridAdder<SCagent>(), false, xdim, ydim));

// create the agents and add to the context

```

```
/*
RetailerAgent agentR = null;

for (int r = 0; r < 2; r++)

{

    agentR = new RetailerAgent();

    //agentR.setName("AgentR-"+r);

    context.add(agentR);

}

*/
// for now, we start with one retailer agent and one supplier agent

RetailerAgent agentR1 = null;

agentR1 = new RetailerAgent();

agentR1.setName("AgentR1");

context.add(agentR1);

RetailerAgent2 agentR2 = null;
```

```
agentR2 = new RetailerAgent2();
```

```
agentR2.setName("AgentR2");
```

```
context.add(agentR2);
```

```
RetailerAgent3 agentR3 = null;
```

```
agentR3 = new RetailerAgent3();
```

```
agentR3.setName("AgentR3");
```

```
context.add(agentR3);
```

```
SupplierAgent agentS = null;
```

```
agentS = new SupplierAgent();
```

```
agentS.setName("AgentS");
```

```
context.add(agentS);
```

```
Moderator Mod = null;
```

```
Mod = new Moderator();
```

```
Mod.setName("Mod");

context.add(Mod);

testAgent1 TA1 = null;

TA1 = new testAgent1();

TA1.setName("TA1");

context.add(TA1);

testAgent2 TA2 = null;

TA2 = new testAgent2();

TA2.setName("TA2");

context.add(TA2);

return context;

}
```

File 3: scenario.xml

```
<?xml version="1.0" encoding="UTF-8" ?>

<Scenario>

<repast.simphony.dataLoader.engine.ClassNameDataLoaderAction context="sc"
file="repast.simphony.dataLoader.engine.ClassNameDataLoaderAction_0.xml" />

</Scenario>
```

File 4: SCagent.java

```
//This is the base agent class that implements some  
  
//fundamental methods common to all agents  
  
//Note that many parts of the code are borrowed from the Beer Game  
  
//implementation by Wolfgang Hoscheck  
  
  
  
package repast.simphony.demo.sc;  
  
  
  
  
  
  
import java.util.*;  
  
//import repast.engine.environment.RunEnvironment;  
  
import repast.simphony.engine.environment.RunEnvironment;  
  
//import repast.simphony.engine.schedule.ScheduledMethod;  
  
import repast.simphony.parameter.Parameters;  
  
//import repast.simphony.random.RandomHelper;
```

```
import repast.simphony.annotate.AgentAnnot;
```

```
@AgentAnnot(displayName = "SC Agent")
```

```
public class SCagent {
```

```
    public SCagent(){
```

```
        //the constructor
```

```
}
```

```
//total cost
```

```
protected double totalCostAvg;
```

```
//the unit cost of stock on hand
```

```
protected double invCost=0.1;
```

```
//the unit cost of stock in shortage
```

```
protected double shortCost=0.2;
```

```
//the cost of placing an order (no matter how big the order is)

protected double fixedOrderCost=0.0;

//the unit cost of goods for each order placed

protected double purchaseCost=0.4;

//the quantity of current order

protected int currentOrder;

//initial demand from the agent of lower level

protected int initialDemandAmount;

//number of retailer agents in the simulation

protected int numberAgent;

// the set of distribution variables

//-----  
protected int normalValue;  
  
protected ArrayList<Integer> normalStream = new ArrayList<Integer>();
```

```
protected int uniformValue;

protected ArrayList<Integer> uniformStream = new ArrayList<Integer>();

//-----
// The agent's name. This is set in the context creator

protected String theName;

protected String theName_Ls = "Supplier";

protected String theName_Lr1 = "Retailer1";

protected String theName_Lr2 = "Retailer2";

protected String theName_Lr3 = "Retailer3";

// List some label names for display purpose only

protected String stock_Ls = "Stock_s";
```

```
protected String stock_Lr1 = "Stock_r1";
```

```
protected String stock_Lr2 = "Stock_r2";
```

```
protected String stock_Lr3 = "Stock_r3";
```

```
protected String backorder_Ls = "backorder_s";
```

```
protected String backorder_Lr1 = "backorder_r1";
```

```
protected String backorder_Lr2 = "backorder_r2";
```

```
protected String backorder_Lr3 = "backorder_r3";
```

```
protected String shipment_Ls = "shipment_s";
```

```
protected String shipment_Lr1 = "shipment_r1";
```

```
protected String shipment_Lr2 = "shipment_r2";
```

```
protected String shipment_Lr3 = "shipment_r3";
```

```
protected String L1 = "lable_1";
```

```
protected String L2 = "lable_2";
```

```
protected String L3 = "lable_3";  
  
//protected double mean;  
  
//**** the unit of cost is in Dollars ****/  
  
// The current tick count, declared here, used by agents  
  
//protected int tickCount;  
  
// The constant for calculating next order value  
  
// This number is read in from the parameter list  
  
protected double Roo;  
  
protected int Elle;  
  
// standard deviation for calculating demand  
  
protected double Sdev1;
```

```
protected double Sdev2;

protected double Sdev3;

protected int dValue;

protected double growthRate;

protected int sharingCase;

protected double Ratio12;

protected double Ratio13;

protected double Ratio23;

// the demand history received from the demand agent

//protected ArrayList demandHistory = new ArrayList();

/*

```

* alpha is the parameter from Sterman: it is the stock adjustment to inventory,

* between desired inventory and actual inventory.

* Stock adjustment to inventory = alpha_s * (desired inventory - inventory).

* Note that our definition of alpha is the same as Sterman's alpha_s.

* We do not use an explicit alpha_sl since this can be found from beta.

*/

protected double alpha = 0.30;

/*

* theta is the parameter from Sterman:

* expected demand(t+1) = theta * demand(t) + (1 - theta)*expected demand(t)

*

* $0 \leq \theta \leq 1$

*/

protected double theta = 0.0;

/*

* beta is a parameter from Sterman: it is the relative weight attached to the

* pipeline vs. stock discrepancies from desired levels. beta = alpha_sl / alpha_s

*/

protected double beta = 0.15;

/*

* totalDesiredInventory is Sterman's "Q". It is a measure of the desired inventory

* relative to the desired pipeline.

* totalDesiredInventory = desired inventory + beta * desired pipeline

*/

protected double q = 17.00;

/*

// the inventory units.

protected double stock;

// the orders not fulfilled

protected double backorders;

```
// the inventory cost. In units of dollars per unit item per unit time period.
```

```
protected double inventoryCost = 0.50;
```

```
// the shortage cost. In units of dollars per unit item per unit time period.
```

```
protected double shortageCost = 2.00;
```

```
// the order cost. The cost per order that is fixed, does not include the cost
```

```
// of goods. In units of dollars per order.
```

```
protected double fixedOrderCost = 1.00;
```

```
// the purchase cost. The cost of goods ordered in units of dollars per unit.
```

```
protected double purchaseCost = 2.00;
```

```
// the order rate.
```

```
protected double currentOrder;
```

```
// the actual order that was demanded by the customer
```

```
protected double actualDemand;
```

```
// the forecasted order (demand) that was predicted by the forecaster
```

```
protected double forecastedDemand;
```

```
// the expected order tracker.
```

```
protected double lastExpectedOrder;

// the actual order tracker.

protected double ordersInPipeline;

// the total cost.

protected double totalCost;

// the historical (actual) demand.

protected LinkedList actualDemandHistory;

// the historical (forecasted) demand.

protected LinkedList forecastedDemandHistory;

*/



// method to find the mean of an arrayList of size greater than 0

public double getArrayMean ( LinkedList<Integer> A, int lengthOfMovingAvg){
```

```
int arraySize;  
  
int sum=0;  
  
arraySize = A.size();  
  
double mean;  
  
int theRange=0;  
  
  
  
  
if(arraySize < lengthOfMovingAvg){  
  
    theRange = 0;  
  
}  
  
else if (arraySize >=lengthOfMovingAvg){  
  
    theRange = arraySize - lengthOfMovingAvg;  
  
}  
  
  
  
  
for (int i = (arraySize-1); i>=theRange; i--){  
  
    sum += A.get(i);  
  
}
```

```
//find the mean

mean = sum/(arraySize-theRange);

return mean;

}

// method to find the mean of an arrayList of size greater than 0

// the input variable is the arrayList<Integer>

public double getSdev ( LinkedList<Integer> A, int lengthOfMovingAvg ){

    double Sdev=0.0;

    double variance=0.0;

    double theSquaredSum=0.0;

    int arraySize;

    double mean;

    int theRange=0;

    int sum=0;
```

```
arraySize = A.size();

if(arraySize < lengthOfMovingAvg){

    theRange = 0;

}

else if (arraySize >=lengthOfMovingAvg){

    theRange = arraySize - lengthOfMovingAvg;

}

for (int i = (arraySize-1); i>=theRange; i--){

    sum += A.get(i);

}

//find the mean

mean = sum/(arraySize-theRange);

//sum(Xi-mean)^2
```

```

for (int i = (arraySize-1); i>=theRange; i--){

    theSquaredSum += (A.get(i)-mean)* (A.get(i)-mean);

}

if (arraySize == 0){

    arraySize = 1;

}

//varince = (sum(Xi-mean)^2)/N

variance = (theSquaredSum)/arraySize;

Sdev= (int)Math.abs(Math.sqrt(variance));

return Sdev;

}

```

```

// get the current tick count

public int getTickCount(){

    return (int)

RunEnvironment.getInstance().getCurrentSchedule().getTickCount();

}

// get the initialized parameters

public void getInitParameters(){

    Parameters p = RunEnvironment.getInstance().getParameters();

    dValue = (Integer)p.getValue("dValue");

    Elle= (Integer)p.getValue("orderLag");

    Roo = (Double)p.getValue("Roo");

    growthRate = (Double)p.getValue("GrowthRate");

    sharingCase = (Integer)p.getValue("SharingCase");

    Ratio12 = (Double)p.getValue("Ratio12");

    Ratio13 = (Double)p.getValue("Ratio13");
}

```

```
Ratio23 = (Double)p.getValue("Ratio23");

Sdev1 = (Double)p.getValue("Sdev1");

Sdev2 = (Double)p.getValue("Sdev2");

Sdev3 = (Double)p.getValue("Sdev3");

}
```

```
public String setName(String name){

    return theName=name;

}
```

```
public String getName(){

    return theName;

}
```

```
public String getTheName_Ls(){
```

```
    return theName_Ls;
```

```
}
```

```
public String getTheName_Lr1(){
```

```
    return theName_Lr1;
```

```
}
```

```
public String getTheName_Lr2(){
```

```
    return theName_Lr2;
```

```
}
```

```
public String getTheName_Lr3(){
```

```
    return theName_Lr3;
```

```
}
```

```
public String getStock_Ls(){
```

```
    return stock_Ls;
```

```
}
```

```
public String getStock_Lr1(){
```

```
        return stock_Lr1;

    }

    public String getStock_Lr2(){

        return stock_Lr2;

    }

    public String getStock_Lr3(){

        return stock_Lr3;

    }

    public String getBackorder_Ls(){

        return backorder_Ls;

    }

    public String getBackorder_Lr1(){

        return backorder_Lr1;

    }

    }

    public String getBackorder_Lr2(){

        return backorder_Lr2;

    }
```

```
}

public String getBackorder_Lr3(){

    return backorder_Lr3;

}

public String getShipment_Ls(){

    return shipment_Ls;

}

public String getShipment_Lr1(){

    return shipment_Lr1;

}

public String getShipment_Lr2(){

    return shipment_Lr2;

}

public String getShipment_Lr3(){

    return shipment_Lr3;

}
```

```
public String getL1(){

    return L1;

}

public String getL2(){

    return L2;

}

public String getL3(){

    return L3;

}

public double getRatio12(){

    return Ratio12;

}
```

}

File 5: SupplierAgent.java

```
package repast.simphony.demo.sc;

import java.util.LinkedList;
//import java.util.ArrayList;
//import org.apache.commons.math.stat.descriptive.moment.StandardDeviation;
// see commons-math archive

import repast.simphony.engine.environment.RunEnvironment;
import repast.simphony.engine.schedule.ScheduledMethod;
import repast.simphony.parameter.Parameters;
import repast.simphony.random.RandomHelper;
import repast.simphony.annotate.AgentAnnot;
```

```
*****
```

/ The following is the working code for validation part of the dissertation.*

** note: to reactivate this code, need to also remove the marks in the middle*

** of the code which marks "continues..."*

```
@AgentAnnot(displayName="Supplier Agent")
```

```
public class SupplierAgent extends SCagent {
```

```
    // total stock (inventory) on hand
```

```
    protected int stock=0;
```

```
    // this is for calculating the average cost only
```

```
    protected int totalStock=0;
```

```
// the memory that keeps the history of stock levels

protected LinkedList<Integer> stockHistory = new LinkedList<Integer>();

//total backordered quantity

protected int backOrder=0;

// the memory that keeps the history of backorder levels

protected LinkedList<Integer> backHistory = new LinkedList<Integer>();

//the current and previous demand received from the retailers

protected int prevDemandS = 0;

protected int currentDemandS = 0;

// received orders from each agent (assume we have 3 agents)

protected int rcvdorder1 = 0;
```

```
protected int recvorder2 = 0;

protected int recvorder3 = 0;

protected int totalRevdOrder = 0;

// The storage for keeping the history of demand

protected LinkedList<Integer> demandStream=new LinkedList<Integer>();

// the current and previous orders placed by supplier to the upper stream

protected int currentOrderS = 0;

protected int totalPredicted = 0;

protected int prevOrderS = 0;

protected int currentOrder1 = 0;

protected int currentOrder2 = 0;

protected int currentOrder3 = 0;
```

```
// the memory for received orders from the retailer agents

protected LinkedList<Integer> rcvOrderHistory1 = new LinkedList<Integer>();

protected LinkedList<Integer> rcvOrderHistory2 = new LinkedList<Integer>();

protected LinkedList<Integer> rcvOrderHistory3 = new LinkedList<Integer>();

protected int YtPredict1;

protected int YtPredict2;

protected int YtPredict3;

protected int YtActual1;

protected int YtActual2;

protected int YtActual3;

protected static LinkedList<Integer> suppliedHistory1 = new  
LinkedList<Integer>();
```

```
protected static LinkedList<Integer> suppliedHistory2 = new  
LinkedList<Integer>();  
  
protected static LinkedList<Integer> suppliedHistory3 = new  
LinkedList<Integer>();  
  
// the memory for the total received orders  
  
protected LinkedList<Integer> totalRcvOrderHistory = new  
LinkedList<Integer>();  
  
protected LinkedList<Integer> totalRcvOrderLine = new LinkedList<Integer>();  
  
// the memory for orders placed by the supplier agent  
  
protected LinkedList<Integer> orderHistoryS = new LinkedList<Integer>();  
  
protected LinkedList<Integer> orderHistory1 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> orderHistory2 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> orderHistory3 = new LinkedList<Integer>();
```

```
// temp memory of the orders placed by the supplier agent

// this storage also serves as the received shipment from upper stream

// (the shipment queue from the upper stream) with a delay of "L"(Elle value).

protected LinkedList<Integer> orderLine = new LinkedList<Integer>(); // total
ordered

// shipment queue is also used to track individual predicted retailer demand

// orderLine = orderLine1+orderLine2+orderLine3

protected LinkedList<Integer> orderLine1 = new LinkedList<Integer>();

protected LinkedList<Integer> orderLine2 = new LinkedList<Integer>();

protected LinkedList<Integer> orderLine3 = new LinkedList<Integer>();

//shipment received by supplier agent from the upper stream

protected int shipment = 0;
```

```
// orders delivered to the retailers

//they are static variables to be accessed directly by the retailers

protected int supplied1 = 0;

protected int supplied2 = 0;

protected int supplied3 = 0;

//private int unfilledOrder1 = 0;

//private int unfilledOrder2 = 0;

//private int unfilledOrder3 = 0;

private double testValue;

protected LinkedList<Integer> unfilledOrderHistory1 = new
LinkedList<Integer>();
```

```
protected LinkedList<Integer> unfilledOrderHistory2 = new  
LinkedList<Integer>();  
  
protected LinkedList<Integer> unfilledOrderHistory3 = new  
LinkedList<Integer>();  
  
// the ratio of supplied (to Retailers) / stock  
  
// for calculating shipment to retailers  
  
// when stock is lower than needed  
  
private double fulfilRate = 0.0;  
  
//private int totalSupplied = 0;  
  
// the mean calculated for generating the error term of t+1  
  
protected double theMean1=0.0;  
  
protected double theMean2=0.0;  
  
protected double theMean3=0.0;
```

```
// the standard deviation of orders received calculated to generate the error term of  
t+1  
  
protected double theSdev1=0.0;  
  
protected double theSdev2=0.0;  
  
protected double theSdev3=0.0;  
  
  
  
  
  
  
// the standard deviation of orders to upper stream calculated for comparison  
  
protected double theSdevS1 = 0.0;  
  
protected double theSdevS2 = 0.0;  
  
protected double theSdevS3 = 0.0;  
  
  
  
  
  
  
  
  
// the error term generated from the normal stream  
  
protected int theErr1;  
  
protected int theErr2;  
  
protected int theErr3;
```

```
// the following are intermediate variables for calculating the next order
```

```
// since the formula is very long, we cut it into three parts
```

```
// part1 + part2 - part3
```

```
// order is placed for each retailer, so there are three sets
```

```
// part1 = d+Roo*YtActual(i)
```

```
// part2 = ((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * error(t+1)
```

```
// part3 = Roo*((1-Math.pow(Roo, (Elle+1)))/(1-Roo)) * error(t)
```

```
// error(t) = YtPredict(i) - YtActual(i)
```

```
protected int part1_1;
```

```
protected int part2_1;
```

```
protected int part3_1;
```

```
protected int part1_2;
```

```
protected int part2_2;
```

```
protected int part3_2;  
  
protected int part1_3;  
  
protected int part2_3;  
  
protected int part3_3;  
  
// if a manufacturer is added, these variables are useful, but not now  
  
//private LinkedList<Integer> shippingLine = new LinkedList<Integer>();  
  
//private LinkedList<Integer> productionLine = new LinkedList<Integer>();  
  
public SupplierAgent(){  
    //the constructor, initialize values  
    /*  
     final Double Sdev=10.0;
```

```
// The seed for generating demand

final int theDemandSeed = 0;

RandomHelper.setSeed(theDemandSeed);

// get the Normal distribution values (zero mean, constant variance)

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //    uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

/*
/* continues...

// get the customizable parameters
```

```

getInitParameters();

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

numberAgent = (Integer)p.getValue("numAgent");

//initial demand from each Retailer agents

rcvdorder1 = this.initialDemandAmount;

rcvdorder2 = this.initialDemandAmount;

rcvdorder3 = this.initialDemandAmount;

// total the retailers' orders

// this should also be the order placed by supplier to upper stream

totalRcvdOrder=rcvdorder1+rcvdorder2+rcvdorder3;

// add to the received total order to memory

totalRcvOrderHistory.addLast(this.totalRcvdOrder);

totalRcvOrderLine.addLast(this.totalRcvdOrder);

```

```
//add to the received order from retailer history memory
```

```
rcvOrderHistory1.addLast(this.rcvdorder1);
```

```
rcvOrderHistory2.addLast(this.rcvdorder2);
```

```
rcvOrderHistory3.addLast(this.rcvdorder3);
```

```
//add to the supplied-to-retail history memory
```

```
suppliedHistory1.addLast(this.initialDemandAmount);
```

```
suppliedHistory2.addLast(this.initialDemandAmount);
```

```
suppliedHistory3.addLast(this.initialDemandAmount);
```

```
unfilledOrderHistory1.addLast(0);
```

```
unfilledOrderHistory2.addLast(0);
```

```
unfilledOrderHistory3.addLast(0);
```

```
// initial state
```

```
stock = 0;

stockHistory.addLast(stock);

backOrder = 0;

backHistory.addLast(backOrder);

// get the initial supplier's order forecasts

// and add them to order history memory

this.currentOrderS=rcvdorder1 + rcvdorder2 + rcvdorder3;

// for the first L=1 period, orders are set

for(int i = 0; i < Elle; i++){

    this.orderLine.addLast(this.currentOrderS);

    this.orderHistoryS.addLast(this.currentOrderS);

    // add to the individual order memory for supplier

    orderLine1.addLast(this.initialDemandAmount);

    orderLine2.addLast(this.initialDemandAmount);
```

```
        orderLine3.addLast(this.initialDemandAmount);

        orderHistory1.addLast(this.initialDemandAmount);

        orderHistory2.addLast(this.initialDemandAmount);

        orderHistory3.addLast(this.initialDemandAmount);

    }

rcvdorder1 = 0;

rcvdorder2 = 0;

rcvdorder3 = 0;

}

@ScheduledMethod(start=1, interval=1)

public void calculateSupplier(){

    if (getTickCount() % 4 == 1){


```

```
// now the supplier will do the job

// 1. check inventory after having received the shipment from
upper stream

shipment= orderLine.getFirst();

// update the order line (the shipment queue from the upper stream)

orderLine.removeFirst();

// add the shipment to the current stock

this.stock += shipment;

totalStock = stock;

}

else if (getTickCount() % 4 == 2){

// do nothing at this tick

}
```

```
else if (getTickCount() % 4 == 3){
```

```
// do nothing at this tick
```

```
}
```

```
else if (getTickCount() % 4 == 0){
```

```
// 2. receive the order(s) from the retailers and update stock
```

```
//current period, this is actual Y(t) from retailer
```

```
//before doing all these, need to get the last retailer ordered
```

```
//to find the (Y(t)predict - Y(t)actual)
```

```
YtPredict1= orderLine1.getFirst();
```

```
orderLine1.removeFirst();
```

```
YtPredict2= orderLine2.getFirst();
```

```
orderLine2.removeFirst();
```

```
YtPredict3= orderLine3.getFirst();
```

```
orderLine3.removeFirst();

rcvdorder1 = RetailerAgent.orderLineR1.getLast();

//RetailerAgent.orderLineR1.removeFirst();

rcvOrderHistory1.addLast(this.rcvdorder1);

rcvdorder2 = RetailerAgent2.orderLineR2.getLast();

//RetailerAgent2.orderLineR2.removeFirst();

rcvOrderHistory2.addLast(this.rcvdorder2);

rcvdorder3 = RetailerAgent3.orderLineR3.getLast();

//RetailerAgent3.orderLineR3.removeFirst();

rcvOrderHistory3.addLast(this.rcvdorder2);

YtActual1 = rcvdorder1;

YtActual2 = rcvdorder2;

YtActual3 = rcvdorder3;
```

```
// total the orders

totalRcvdOrder=rcvdorder1+rcvdorder2+rcvdorder3;

// prepare the variable for calculating the order to upper stream

//prevDemandS = totalRcvOrderLine.getLast();

//currentDemandS = totalRcvdOrder;

// add the received total order to memory

totalRcvOrderHistory.addLast(this.totalRcvdOrder);

totalRcvOrderLine.addLast(this.totalRcvdOrder);

// the "supplied" variable is the current shipment (to-be)

// received by the retailer in next period

supplied1 = 0;

supplied2 = 0;

supplied3 = 0;
```

```

// taking into consideration of backorders of last period

// as well as the scenario of smaller stock available

// note that the backorder is the sum of individual unfilled orders

// (totalRcvdOrder + backOrder) is the total of current received

// order plus back orders

//unfilledOrder1 = unfilledOrderHistory1.getLast();

//unfilledOrder2 = unfilledOrderHistory2.getLast();

//unfilledOrder3 = unfilledOrderHistory3.getLast();

/* backorder handled differently

if( stock >= (totalRcvdOrder + backOrder) ){

    supplied1 = unfilledOrder1 + rcvdorder1;

    supplied2 = unfilledOrder2 + rcvdorder2;

    supplied3 = unfilledOrder3 + rcvdorder3;
}

```

```
//add to the supplied-to-retail history memory  
  
suppliedHistory1.addLast(this.supplied1);  
  
suppliedHistory2.addLast(this.supplied2);  
  
suppliedHistory3.addLast(this.supplied3);  
  
  
  
  
this.stock -= (totalRcvdOrder + backOrder);  
  
// fulfill (update) the backorders  
  
backOrder = 0;  
  
unfilledOrder1 = 0;  
  
unfilledOrder2 = 0;  
  
unfilledOrder3 = 0;  
  
  
  
  
unfilledOrderHistory1.addLast(unfilledOrder1);  
  
unfilledOrderHistory2.addLast(unfilledOrder2);  
  
unfilledOrderHistory3.addLast(unfilledOrder3);
```

```
// restore the initial values

supplied1 = 0;

supplied2 = 0;

supplied3 = 0;

}

else if( stock < (totalRcvdOrder + backOrder) ){

    // if stock could not even fulfill the previous order

    // supplied will be only a proportion of previous demand

    from retailers

        if((this.stock - backOrder) < 0){

            if(backOrder == 0){

                fulfilRate = 1;
```

```
    }

    else{

        fulfilRate = Math.abs(stock / backOrder);

    }

    supplied1 = (int)Math.floor(unfilledOrder1 *

fulfilRate);

    supplied2 = (int)Math.floor(unfilledOrder2 *

fulfilRate);

    supplied3 = (int)Math.floor(unfilledOrder3 *

fulfilRate);

//add to the supplied-to-retail history memory

suppliedHistory1.addLast(this.supplied1);

suppliedHistory2.addLast(this.supplied2);

suppliedHistory3.addLast(this.supplied3);

// current received orders go to backorder directly
```

```
unfilledOrder1 += (rcvdorder1 - supplied1);

unfilledOrder2 += (rcvdorder2 - supplied2);

unfilledOrder3 += (rcvdorder3 - supplied3);

unfilledOrderHistory1.addLast(unfilledOrder1);

unfilledOrderHistory2.addLast(unfilledOrder2);

unfilledOrderHistory3.addLast(unfilledOrder3);

backOrder = unfilledOrder1 + unfilledOrder2 +
unfilledOrder3;

stock = 0;

// restore the initial values

supplied1 = 0;

supplied2 = 0;

supplied3 = 0;

}
```

```

// if stock satisfies the previous backorders, but not all
orders from Retailers

else if((this.stock - backOrder) >= 0){

    // this is not possible, but in case there is a bug in
the program

    if(totalRcvdOrder == 0){

        //simply split the stock

        fulfilRate = 0;

    }

    else{

        fulfilRate = Math.abs(stock-backOrder) /
totalRcvdOrder;

    }

}

unfilledOrder1 = unfilledOrderHistory1.getLast();

unfilledOrder2 = unfilledOrderHistory2.getLast();

```

```
unfilledOrder3 = unfilledOrderHistory3.getLast();
```

```
supplied1 = (int)Math.floor(rcvdorder1 * fulfilRate)
```

```
+ unfilledOrder1;
```

```
supplied2 = (int)Math.floor(rcvdorder3 * fulfilRate)
```

```
+ unfilledOrder2;
```

```
supplied3 = (int)Math.floor(rcvdorder3 * fulfilRate)
```

```
+ unfilledOrder3;
```

```
//add to the supplied-to-retail history memory
```

```
suppliedHistory1.addLast(this.supplied1);
```

```
suppliedHistory2.addLast(this.supplied2);
```

```
suppliedHistory3.addLast(this.supplied3);
```

```
backOrder = totalRcvdOrder + backOrder - stock;
```

```
stock = 0;
```

```
        unfilledOrder1 += rcvdorder1 - supplied1;

        unfilledOrder2 += rcvdorder2 - supplied2;

        unfilledOrder3 += rcvdorder3 - supplied3;

        unfilledOrderHistory1.addLast(unfilledOrder1);

        unfilledOrderHistory2.addLast(unfilledOrder2);

        unfilledOrderHistory3.addLast(unfilledOrder3);

        // restore the initial values

        supplied1 = 0;

        supplied2 = 0;

        supplied3 = 0;

    }

    rcvdorder1 = 0;

    rcvdorder2 = 0;
```

```
    rcvdorder3 = 0;  
  
}  
  
*/  
  
/* continues...  
  
stock -= totalRcvdOrder;  
  
// stock < 0? hopefully not, backorder means lost sales  
  
if (stock < 0){  
  
    backOrder = (int)Math.abs(stock);  
  
    stock = 0;  
  
}  
  
else if (stock >= 0){  
  
    backOrder = 0;  
  
}
```

```
stockHistory.addLast(stock);

backHistory.addLast(backOrder);

// 3. calculate the order to the upper stream

// assume all supplier's orders are fulfilled from the upper stream.

// uses the AR1 process described in Lee, et al. (2000)

// need to change to moving average

//mean of past orders received from each retailer

theMean1 =0.0;

theMean2 =0.0;

theMean3 =0.0;
```

```
testValue = getArrayMean(rcvOrderHistory1, 30);

//standard deviation of past orders received from each retailer

theSdev1 = getSdev(rcvOrderHistory1,30);

theSdev2 = getSdev(rcvOrderHistory2,30);

theSdev3 = getSdev(rcvOrderHistory3,30);

//standard deviation of past orders placed with the upper stream

//when info is not shared

theSdevS1 = getSdev(orderHistory1, 30);

theSdevS2 = getSdev(orderHistory2, 30);

theSdevS3 = getSdev(orderHistory3, 30);

//when shared info
```

```
//theSdevS1 = 30;  
  
//theSdevS2 = 40;  
  
//theSdevS3 = 20;  
  
  
  
RandomHelper.setSeed(getTickCount());  
  
// get the Normal distribution value: retailer 1  
  
RandomHelper.createNormal(theMean1, theSdev1);  
  
theErr1 = RandomHelper.getNormal().nextInt();  
  
  
  
// get the Normal distribution value: retailer 2  
  
RandomHelper.createNormal(theMean2, theSdev2);  
  
theErr2 = RandomHelper.getNormal().nextInt();  
  
  
  
// get the Normal distribution value: retailer 3  
  
RandomHelper.createNormal(theMean3, theSdev3);  
  
theErr3 = RandomHelper.getNormal().nextInt();
```

```

// the AR1 formula, the order quantity considers the change in
order-up-to levels

// get each of three parts for the calculation

// retailer 1

part1_1 = dValue + (int)Math.floor(Roo*YtActual1);

part2_1 = (int)((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr1;

// when info is not shared

part3_1 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* (YtPredict1 -
YtActual1)) /(1-Roo));

// when info is shared

//part3_1 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))*
RetailerAgent.getSharedErr1() /(1-Roo)));

// sum them up

currentOrder1 = Math.max(0,part1_1 + part2_1 - part3_1);

```

```

// retailer 2

part1_2 = dValue + (int)Math.floor(Roo*YtActual2);

part2_2 = (int)((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr2;

// when info is not shared

part3_2 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* (YtPredict2 -
YtActual2)) /(1-Roo));

// when info is shared

//part3_2 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))*
RetailerAgent2.getSharedErr2()) /(1-Roo));

// sum them up

currentOrder2 = Math.max(0,part1_2 + part2_2 - part3_2);

// retailer 3

part1_3 = dValue + (int)Math.floor(Roo*YtActual3);

```

```

part2_3 = (int) (((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr3);

// when info is not shared

part3_3 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))*(YtPredict3 -
YtActual3)) /(1-Roo));

// when info is shared

//part3_3 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))*
RetailerAgent3.getSharedErr3()) /(1-Roo));

// sum them up

currentOrder3 = Math.max(0,part1_3 + part2_3 - part3_3);

orderLine1.addLast(currentOrder1);

orderLine2.addLast(currentOrder2);

orderLine3.addLast(currentOrder3);

orderHistory1.addLast(currentOrder1);

```

```
orderHistory2.addLast(currentOrder2);

orderHistory3.addLast(currentOrder3);

// the total order is the sum of individual orders

this.currentOrderS = currentOrder1 + currentOrder2 +
currentOrder3;

// backorders also need to be ordered

//this.currentOrderS += backOrder;

//this.currentOrderS = Math.max(0, currentOrderS+backOrder-
stock);

totalPredicted = currentOrderS;

this.orderHistoryS.addLast(currentOrderS);

// if stock is high, set order to zero
```

```

        if(stock-backOrder >= currentOrderS){

            currentOrderS = 0;

        }

//if (stock<=0){

    //      this.currentOrderS += backOrder;

//}

// add the orders to memory

this.orderLine.addLast(currentOrderS);

// 4. calculate the total cost per unit shipped

// in case total stock is zero

//if (totalStock == 0){

    //      totalStock = 1;

//}

```

```
        this.totalCostAvg = (this.invCost * this.stock + this.shortCost *  
        this.backOrder +  
  
        this.fixedOrderCost *  
        ((this.currentOrderS > 0)?1 : 0)+  
  
        this.purchaseCost * this.shipment);  
  
    }  
  
}
```

```
public static int getShipmentToR1(){  
  
    return suppliedHistory1.getLast();  
  
}  
  
public static int getShipmentToR2(){  
  
    return suppliedHistory3.getLast();  
  
}  
  
public static int getShipmentToR3(){
```

```
    return suppliedHistory3.getLast();

}

public int isSupplier(){

    return 1;

}

public double getTotalCost(){

    return this.totalCostAvg;

}

public int getTested(){

    return this.backOrder - this.stock;

}

public int getStock(){
```

```
    return stock;

}

public int getBackOrder(){

    return backOrder;

}

public int getShipment(){

    return this.shipment;

}

public int getCurrentOrderS(){

    return currentOrderS;

}

public double getFulfilRate(){
```

```
    return fulfilRate;

}

public int getCurrentDemand(){

    return currentDemandS;

}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

public int getTheErr1(){

    return theErr1;

}
```

```
public int getTheErr2(){

    return theErr2;

}
```

```
public int getTheErr3(){

    return theErr3;

}
```

```
public int getPart1_1(){

    return part1_1;

}
```

```
public int getPart2_1(){

    return part2_1;

}
```

```
public int getPart3_1(){
```

```
    return part3_1;
```

```
}
```

```
public int getPart1_2(){
```

```
    return part1_1;
```

```
}
```

```
public int getPart2_2(){
```

```
    return part2_1;
```

```
}
```

```
public int getPart3_2(){
```

```
    return part3_1;
```

```
}
```

```
public int getPart1_3(){
```

```
    return part1_1;
```

```
}
```

```
public int getPart2_3(){
```

```
    return part2_1;
```

```
}
```

```
public int getPart3_3(){
```

```
    return part3_1;
```

```
}
```

```
public int getCurrentOrder1(){
```

```
    return currentOrder1;
```

```
}
```

```
public int getCurrentOrder2(){
```

```
        return currentOrder2;

    }

    public int getCurrentOrder3(){

        return currentOrder3;

    }

    public int getRcvdorder1(){

        return rcvdorder1;

    }

    public int getRcvdorder2(){

        return rcvdorder2;

    }

    public int getRcvdorder3(){

        return rcvdorder3;

    }

    public double getTheSdev1(){

        return theSdev1;

    }
```

```
}

public double getTheSdev2(){

    return theSdev2;

}

public double getTheSdev3(){

    return theSdev3;

}

// get the variance

public double getTheVar1(){

    return theSdev1*theSdev1;

}

public double getTheVar2(){

    return theSdev2*theSdev2;

}

public double getTheVar3(){
```

```
    return theSdev3*theSdev3;

}
```

```
public double getMean(){

    return testValue;

}
```

```
public int getYtActual1(){

    return YtActual1;

}
```

```
public int getYtPredict1(){

    return YtPredict1;

}
```

```
public int getYtActual2(){

    return YtActual2;

}
```

```
public int getYtPredict2(){
```

```
        return YtPredict2;

    }

    public int getYtActual3(){

        return YtActual3;

    }

    public int getYtPredict3(){

        return YtPredict3;

    }

}

public double getTheSdevS1(){

    return theSdevS1;

}

public double getTheSdevS2(){

    return theSdevS2;

}

public double getTheSdevS3(){
```

```
    return theSdevS3;

}

// get the variance

public double getTheVarS1(){

    return theSdevS1*theSdevS1;

}

public double getTheVarS2(){

    return theSdevS2*theSdevS2;

}

public double getTheVarS3(){

    return theSdevS3*theSdevS3;

}

public double getRoo(){

    return Roo;

}

public int getTotalRcvdOrder(){
```

```
        return totalRcvdOrder;

    }

    public int getTotalPredicted(){

        return totalPredicted;

    }

}

*/
*****/* The following is the code for testing the interactions
/*

```

*/

@AgentAnnot(displayName="Supplier Agent")

public class SupplierAgent extends SCagent {

// total stock (inventory) on hand

protected int stock=0;

// this is for calculating the average cost only

protected int totalStock=0;

// the memory that keeps the history of stock levels

protected LinkedList<Integer> stockHistory = new LinkedList<Integer>();

//total backordered quantity

```
protected int backOrder=0;

// the memory that keeps the history of backorder levels

protected LinkedList<Integer> backHistory = new LinkedList<Integer>();

//the current and previous demand received from the retailers

protected int prevDemandS = 0;

protected int currentDemandS = 0;

// received orders from each agent (assume we have 3 agents)

protected int rcvdorder1 = 0;

protected int rcvdorder2 = 0;

protected int rcvdorder3 = 0;

protected int totalRcvdOrder = 0;

// The storage for keeping the history of demand
```

```
protected LinkedList<Integer> demandStream=new LinkedList<Integer>();  
  
// the current and previous orders placed by supplier to the upper stream  
  
protected int currentOrderS = 0;  
  
protected int totalPredicted = 0;  
  
protected int prevOrderS = 0;  
  
protected int currentOrder1 = 0;  
  
protected int currentOrder2 = 0;  
  
protected int currentOrder3 = 0;  
  
// the memory for received orders from the retailer agents  
  
protected LinkedList<Integer> rcvOrderHistory1 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> rcvOrderHistory2 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> rcvOrderHistory3 = new LinkedList<Integer>();
```

```
protected int YtPredict1;

protected int YtPredict2;

protected int YtPredict3;

protected int YtActual1;

protected int YtActual2;

protected int YtActual3;

protected static LinkedList<Integer> suppliedHistory1 = new  
LinkedList<Integer>();  
  
protected static LinkedList<Integer> suppliedHistory2 = new  
LinkedList<Integer>();  
  
protected static LinkedList<Integer> suppliedHistory3 = new  
LinkedList<Integer>();  
  
// total shipped to retailers each period  
  
protected LinkedList<Integer> suppliedHistory = new LinkedList<Integer>();
```

```
// the memory for the total received orders

protected LinkedList<Integer> totalRcvOrderHistory = new
LinkedList<Integer>();

protected LinkedList<Integer> totalRcvOrderLine = new LinkedList<Integer>();

// the memory for orders placed by the supplier agent

protected LinkedList<Integer> orderHistoryS = new LinkedList<Integer>();

protected LinkedList<Integer> orderHistory1 = new LinkedList<Integer>();

protected LinkedList<Integer> orderHistory2 = new LinkedList<Integer>();

protected LinkedList<Integer> orderHistory3 = new LinkedList<Integer>();

// temp memory of the orders placed by the supplier agent

// this storage also serves as the received shipment from upper stream

// (the shipment queue from the upper stream) with a delay of "L"(Elle value).
```

```
protected LinkedList<Integer> orderLine = new LinkedList<Integer>(); // total  
ordered  
  
// shipment queue is also used to track individual predicted retailer demand  
  
// orderLine = orderLine1+orderLine2+orderLine3  
  
protected LinkedList<Integer> orderLine1 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> orderLine2 = new LinkedList<Integer>();  
  
protected LinkedList<Integer> orderLine3 = new LinkedList<Integer>();  
  
//shipment received by supplier agent from the upper stream  
  
protected int shipment = 0;  
  
// orders delivered to the retailers  
  
//they are static variables to be accessed directly by the retailers  
  
protected int supplied1 = 0;
```

```
protected int supplied2 = 0;

protected int supplied3 = 0;

private int unfilledOrder1 = 0;

private int unfilledOrder2 = 0;

private int unfilledOrder3 = 0;

private double testValue;

protected LinkedList<Integer> unfilledOrderHistory1 = new
LinkedList<Integer>();

protected LinkedList<Integer> unfilledOrderHistory2 = new
LinkedList<Integer>();

protected LinkedList<Integer> unfilledOrderHistory3 = new
LinkedList<Integer>();

// the ratio of supplied (to Retailers) / stock
```

```
// for calculating shipment to retailers

// when stock is lower than needed

private double fulfilRate = 0.0;

//private int totalSupplied = 0;

// the mean calculated for generating the error term of t+1

protected double theMean1=0.0;

protected double theMean2=0.0;

protected double theMean3=0.0;

// the standard deviation of orders received calculated to generate the error term of

t+1

protected double theSdev1=0.0;

protected double theSdev2=0.0;

protected double theSdev3=0.0;
```

```
// the standard deviation of orders to upper stream calculated for comparison
```

```
protected double theSdevS1 = 0.0;
```

```
protected double theSdevS2 = 0.0;
```

```
protected double theSdevS3 = 0.0;
```

```
// the error term generated from the normal stream
```

```
protected int theErr1;
```

```
protected int theErr2;
```

```
protected int theErr3;
```

```
// the following are intermediate variables for calculating the next order
```

```
// since the formula is very long, we cut it into three parts
```

```
// part1 + part2 - part3
```

```
// order is placed for each retailer, so there are three sets
```

```
// part1 = d+Roo*YtActual(i)
```

```
// part2 = ((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * error(t+1)

// part3 = Roo*((1-Math.pow(Roo, (Elle+1)))/(1-Roo)) * error(t)

// error(t) = YtPredict(i) - YtActual(i)

protected int part1_1;

protected int part2_1;

protected int part3_1;

protected int part1_2;

protected int part2_2;

protected int part3_2;

protected int part1_3;

protected int part2_3;

protected int part3_3;
```

```
// if a manufacturer is added, these variables are useful, but not now

//private LinkedList<Integer> shippingLine = new LinkedList<Integer>();

//private LinkedList<Integer> productionLine = new LinkedList<Integer>();

public SupplierAgent(){

    //the constructor, initialize values

    /*

    final Double Sdev=10.0;

    // The seed for generating demand

    final int theDemandSeed = 0;

    RandomHelper.setSeed(theDemandSeed);

    // get the Normal distribution values (zero mean, constant variance)

    RandomHelper.createNormal(0.0, Sdev);

    //RandomHelper.createUniform(0, 25);
```

```

for(int i=0; i<10000; i++){

    //      uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());


}

/*
// get the customizable parameters

getInitParameters();

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

numberAgent = (Integer)p.getValue("numAgent");

//initial demand from each Retailer agents

rcvdorder1 = this.initialDemandAmount;

rcvdorder2 = this.initialDemandAmount;

rcvdorder3 = this.initialDemandAmount;

```

```
// total the retailers' orders

// this should also be the order placed by supplier to upper stream

totalRcvdOrder=rcvdorder1+rcvdorder2+rcvdorder3;

// add to the received total order to memory

totalRcvOrderHistory.addLast(this.totalRcvdOrder);

totalRcvOrderLine.addLast(this.totalRcvdOrder);

//add to the received order from retailer history memory

rcvOrderHistory1.addLast(this.rcvdorder1);

rcvOrderHistory2.addLast(this.rcvdorder2);

rcvOrderHistory3.addLast(this.rcvdorder3);

//add to the supplied-to-retail history memory

suppliedHistory1.addLast(this.initialDemandAmount);

suppliedHistory2.addLast(this.initialDemandAmount);
```

```
suppliedHistory3.addLast(this.initialDemandAmount);

// add to the supplied to retailers total memory

suppliedHistory.addLast(this.supplied1+this.supplied2+this.supplied3);

unfilledOrderHistory1.addLast(0);

unfilledOrderHistory2.addLast(0);

unfilledOrderHistory3.addLast(0);

// initial state

stock = 0;

stockHistory.addLast(stock);

backOrder = 0;

backHistory.addLast(backOrder);

// get the initial supplier's order forecasts
```

```
// and add them to order history memory

this.currentOrderS=rcvdorder1 + rcvdorder2 + rcvdorder3;

// for the first L=1 period, orders are set

for(int i = 0; i < Elle; i++){
    this.orderLine.addLast(this.currentOrderS);

    this.orderHistoryS.addLast(this.currentOrderS);

    // add to the individual order memory for supplier

    orderLine1.addLast(this.initialDemandAmount);

    orderLine2.addLast(this.initialDemandAmount);

    orderLine3.addLast(this.initialDemandAmount);

    orderHistory1.addLast(this.initialDemandAmount);

    orderHistory2.addLast(this.initialDemandAmount);

    orderHistory3.addLast(this.initialDemandAmount);

}
```

```
    rcvdorder1 = 0;

    rcvdorder2 = 0;

    rcvdorder3 = 0;

}

{@ScheduledMethod(start=1, interval=1)

public void calculateSupplier(){

    if (getTickCount() % 4 == 1){

        // now the supplier will do the job

        // 1. check inventory after having received the shipment from
        upper stream

        shipment= orderLine.getFirst();

        // update the order line (the shipment queue from the upper stream)
```

```
    orderLine.removeFirst();

    // get the current stock

    this.stock = getStock();

    // add the shipment to the current stock

    this.stock += shipment;

    totalStock = stock;

}

else if (getTickCount() % 4 == 2){

    // do nothing at this tick

}

else if (getTickCount() % 4 == 3){

    // do nothing at this tick

}
```

```
else if (getTickCount() % 4 == 0){  
  
    // 2. receive the order(s) from the retailers and update stock  
  
    //current period, this is actual Y(t) from retailer  
  
    //before doing all these, need to get the last retailer ordered  
  
    //to find the (Y(t)predict - Y(t)actual)  
  
  
  
  
    YtPredict1= orderLine1.getFirst();  
  
    orderLine1.removeFirst();  
  
    YtPredict2= orderLine2.getFirst();  
  
    orderLine2.removeFirst();  
  
    YtPredict3= orderLine3.getFirst();  
  
    orderLine3.removeFirst();  
  
  
  
  
    rcvdorder1 = RetailerAgent.orderLineR1.getLast();  
  
    //RetailerAgent.orderLineR1.removeFirst();
```

```
    rcvOrderHistory1.addLast(this.rcvdorder1);

    rcvdorder2 = RetailerAgent2.orderLineR2.getLast();

    //RetailerAgent2.orderLineR2.removeFirst();

    rcvOrderHistory2.addLast(this.rcvdorder2);

    rcvdorder3 = RetailerAgent3.orderLineR3.getLast();

    //RetailerAgent3.orderLineR3.removeFirst();

    rcvOrderHistory3.addLast(this.rcvdorder2);

    YtActual1 = rcvdorder1;

    YtActual2 = rcvdorder2;

    YtActual3 = rcvdorder3;

    // total the orders

    totalRcvdOrder=rcvdorder1+rcvdorder2+rcvdorder3;

    // prepare the variable for calculating the order to upper stream
```

```
//prevDemandS = totalRcvOrderLine.getLast();

//currentDemandS = totalRcvdOrder;

// add the received total order to memory

totalRcvOrderHistory.addLast(this.totalRcvdOrder);

totalRcvOrderLine.addLast(this.totalRcvdOrder);

// the "supplied" variable is the current shipment (to-be)

// received by the retailer in next period

supplied1 = 0;

supplied2 = 0;

supplied3 = 0;

// taking into consideration of backorders of last period

// as well as the scenario of smaller stock available

// note that the backorder is the sum of individual unfilled orders
```

```
/* backorder handled differently in different cases*/  
  
/*-----*/  
  
// before start, we need find out the backorders of last period for  
each retailer  
  
unfilledOrder1 = unfilledOrderHistory1.getLast();  
  
unfilledOrder2 = unfilledOrderHistory2.getLast();  
  
unfilledOrder3 = unfilledOrderHistory3.getLast();  
  
// as well as the total backorder so far  
  
// in fact this is the total of individual accumulated backorders.  
  
backOrder = getBackOrder();
```

```
if((this.stock - backOrder) >= 0){  
  
    // now that we have sufficient stock for the backorder  
  
    // let's find if the stock is good for the received orders  
  
    // first, get the stock after backorders  
  
    this.stock -= backOrder;  
  
    // then consider the received orders  
  
    // first, we have plenty of stock  
  
    // all orders will be fulfilled  
  
    if((this.stock-totalRcvdOrder) >= 0){  
  
        stock -= totalRcvdOrder;  
  
        fulfilRate = 1.0; // this will not be necessary  
  
        ////////////////////////////////  
    }  
}
```

```
// now that we have got the fulfilRate  
  
// it is time to compute the shipment that will be  
delievered next period  
  
// this calculation is based on the received order of  
this period  
  
// so that the back orders will be fulfilled faster  
  
// thus a retailer is receiving his ordered quantity of  
(L+L) periods ago, plus  
  
// (if there are any) the backordered quantity of last  
period; note here, the last period backorder.  
  
// All backorders (including the costs) in their case  
are ignored  
  
// Also, this backorder info should be used when  
computing the next order by supplier
```

supplied1 = unfilledOrder1 + rcvdorder1;

supplied2 = unfilledOrder2 + rcvdorder2;

supplied3 = unfilledOrder3 + rcvdorder3;

```
// all backorders become zero - the initial state
```

```
backOrder = 0;
```

```
unfilledOrder1 = 0;
```

```
unfilledOrder2 = 0;
```

```
unfilledOrder3 = 0;
```

```
}
```

```
//then in another case, the stock is not sufficient
```

```
// we need to calculate the fulfil rate
```

```
else if((this.stock-totalRcvdOrder) < 0){
```

```
// the remaining stock is everything we can offer to
```

```
the retailers
```

```

// this case (totalRcvdOrder == 0) is not possible
since stock > 0,

// but in case there is a bug in the program and to
avoid the case of "devided by zero"

if(totalRcvdOrder == 0){

    //no order received, so no shipment

    fulfilRate = 0;

}

else{

    fulfilRate = this.stock / totalRcvdOrder;

}

///////////////////////////////
// now that we have got the fulfilRate

// it is time to compute the shipment that will be
delievered next period

```

// this calculation is based on the received order of
this period

// so that the back orders will be fulfilled faster

// thus a retailer is receiving his ordered quantity of
(L+L) periods ago, plus

// (if there are any) the backordered quantity of last
period; note here, the last period backorder.

// All backorders (including the costs) in their case
are ignored

// Also, this backorder info should be used when
computing the next order by supplier

supplied1 = (int)Math.floor(rcvdorder1 * fulfilRate)
+ unfilledOrder1;

supplied2 = (int)Math.floor(rcvdorder3 * fulfilRate)
+ unfilledOrder2;

supplied3 = (int)Math.floor(rcvdorder3 * fulfilRate)
+ unfilledOrder3;

```
// since stock >= backorder, previous backorders are  
all fulfilled
```

```
// thus the new backorders are only the current ones
```

```
unfilledOrder1 = (int)Math.floor(rcvdorder1 * (1 -  
fulfilRate));
```

```
unfilledOrder2 = (int)Math.floor(rcvdorder2 * (1 -  
fulfilRate));
```

```
unfilledOrder3 = (int)Math.floor(rcvdorder3 * (1 -  
fulfilRate));
```

```
// update the back order
```

```
// can use the totalRcvdOrder - stock to get the same  
answer
```

```
backOrder = unfilledOrder1 + unfilledOrder2 +  
unfilledOrder3;
```

```
// update the stock
```

```
        this.stock = 0;

    }

}

// this next case should also be considered and the calcuation of
supplied is different

else if((this.stock - backOrder) < 0){

    // total available for shipment will be the current stock only

    // this stock needs to be shared proportionally by the
retailers

    // the fulfilRate needs to be calculated here

    // this case (backOrder == 0) is not possible, but in case
there is a bug in the program

    // and to avoid the case of "devided by zero"
```

```

        if(backOrder == 0){

            //no backorder, so no share

            fulfilRate = 0;

        }

        else{

            fulfilRate = this.stock / backOrder;

        }

        // now that we have got the fulfilRate

        // it is time to compute the shipment that will be delivered

        next period

        // this calculation is based on the received order of this

        period

        // so that the back orders will be fulfilled faster

        // thus a retailer is receiving his (accumulated, if any)

        backorder quantity only,
    
```

// all new order becomes the new backorder to be fulfilled
next period

// Lee et al (2000)'s model does not consider the backorder
by retailers and the suppliers.

// All backorders (including the costs) in their case are
ignored

// Also, this backorder info should be used when computing
the next order by supplier

// the supplied (shipment to retailers) are only for partially
fufilling the backorders

// this is a very rare case

supplied1 = (int)Math.floor(unfilledOrder1 * fulfilRate);

supplied2 = (int)Math.floor(unfilledOrder1 * fulfilRate);

supplied3 = (int)Math.floor(unfilledOrder1 * fulfilRate);

```
// since stock < backorder, previous backorders are not all  
fulfilled
```

```
// thus the new backorders are only the current order plus  
the left over
```

```
unfilledOrder1 = unfilledOrder1 - supplied1 + rcvdorder1;
```

```
unfilledOrder2 = unfilledOrder2 - supplied2 + rcvdorder2;
```

```
unfilledOrder3 = unfilledOrder3 - supplied3 + rcvdorder3;
```

```
// update the stock info
```

```
this.stock = 0;
```

```
// update the back order
```

```
// can use the backOrder - stock + totalRcvdOrder to get the  
same answer
```

```
backOrder = unfilledOrder1 + unfilledOrder2 +  
unfilledOrder3;
```

```
}
```

```
// after all above calculations, now we do some paper work and  
store the data
```

```
// add the individual backorders to memory for next period's  
calculation
```

```
unfilledOrderHistory1.addLast(unfilledOrder1);
```

```
unfilledOrderHistory2.addLast(unfilledOrder2);
```

```
unfilledOrderHistory3.addLast(unfilledOrder3);
```

```
// add to the supplied-to-retailer history memory
```

```
suppliedHistory1.addLast(this.supplied1);
```

```
suppliedHistory2.addLast(this.supplied2);
```

```
suppliedHistory3.addLast(this.supplied3);
```

```
// add to the supplied to retailers total memory

suppliedHistory.addLast(this.supplied1+this.supplied2+this.supplied3);

/* backorder handled differently

stock -= totalRcvdOrder;

// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}

else if (stock >= 0){

    backOrder = 0;

}
```

```
*/  
  
// store the stock and backorder data to memory  
  
stockHistory.addLast(stock);  
  
backHistory.addLast(backOrder);  
  
// all backorders become zero - the initial state  
  
backOrder = 0;  
  
unfilledOrder1 = 0;  
  
unfilledOrder2 = 0;  
  
unfilledOrder3 = 0;  
  
// the stock is back to initial state  
  
stock = 0;
```

```
// 3. calculate the order to the upper stream  
  
// assume all supplier's orders are fulfilled from the upper stream.  
  
// uses the AR1 process described in Lee, et al. (2000)  
  
  
  
// need to change to moving average  
  
  
  
//mean of past orders received from each retailer  
  
  
  
  
  
  
theMean1 =0.0;  
  
theMean2 =0.0;  
  
theMean3 =0.0;  
  
  
  
  
  
  
testValue = getArrayMean(rcvOrderHistory1, 30);  
  
  
  
  
  
  
//standard deviation of past orders received from each retailer
```

```
theSdev1 = getSdev(rcvOrderHistory1,30);

theSdev2 = getSdev(rcvOrderHistory2,30);

theSdev3 = getSdev(rcvOrderHistory3,30);

//standard deviation of past orders placed with the upper stream

//when info is not shared

theSdevS1 = getSdev(orderHistory1, 30);

theSdevS2 = getSdev(orderHistory2, 30);

theSdevS3 = getSdev(orderHistory3, 30);

//when shared info

//theSdevS1 = 30;

//theSdevS2 = 40;

//theSdevS3 = 20;

RandomHelper.setSeed(getTickCount());
```

```
// get the Normal distribution value: retailer 1

RandomHelper.createNormal(theMean1, theSdev1);

theErr1 = RandomHelper.getNormal().nextInt();

// get the Normal distribution value: retailer 2

RandomHelper.createNormal(theMean2, theSdev2);

theErr2 = RandomHelper.getNormal().nextInt();

// get the Normal distribution value: retailer 3

RandomHelper.createNormal(theMean3, theSdev3);

theErr3 = RandomHelper.getNormal().nextInt();

// the AR1 formula, the order quantity considers the change in
order-up-to levels

// get each of three parts for the calculation
```

```

// retailer 1

part1_1 = dValue + (int)Math.floor(Roo*YtActual1);

part2_1 = (int)((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr1;

// when info is not shared

part3_1 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* (YtPredict1 -
YtActual1)) /(1-Roo));

// when info is shared

//part3_1 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))*
RetailerAgent.getSharedErr1() /(1-Roo)));

// sum them up

currentOrder1 = Math.max(0,part1_1 + part2_1 - part3_1);

// retailer 2

part1_2 = dValue + (int)Math.floor(Roo*YtActual2);

```

```

part2_2 = (int) (((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr2);

// when info is not shared

part3_2 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* (YtPredict2 -
YtActual2)) /(1-Roo));

// when info is shared

//part3_2 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* 
RetailerAgent2.getSharedErr2()) /(1-Roo));

// sum them up

currentOrder2 = Math.max(0,part1_2 + part2_2 - part3_2);

// retailer 3

part1_3 = dValue + (int) Math.floor(Roo*YtActual3);

part2_3 = (int) (((1-Math.pow(Roo, (Elle+2)))/(1-Roo)) * theErr3);

// when info is not shared

part3_3 = (int)((Roo*((1-Math.pow(Roo, (Elle+1))))* (YtPredict3 -
YtActual3)) /(1-Roo));

```

```
// when info is shared

//part3_3 = (int)((Roo*((1-Math.pow(Roo, (Elle+1)))*
RetailerAgent3.getSharedErr3()) /(1-Roo)));

// sum them up

currentOrder3 = Math.max(0,part1_3 + part2_3 - part3_3);

orderLine1.addLast(currentOrder1);

orderLine2.addLast(currentOrder2);

orderLine3.addLast(currentOrder3);

orderHistory1.addLast(currentOrder1);

orderHistory2.addLast(currentOrder2);

orderHistory3.addLast(currentOrder3);

// the total order is the sum of individual orders
```

```

this.currentOrderS = currentOrder1 + currentOrder2 +
currentOrder3;

// backorders also need to be ordered

//this.currentOrderS += backOrder;

//this.currentOrderS = Math.max(0, currentOrderS+backOrder-
stock);

totalPredicted = currentOrderS;

// if stock is high, set order to zero

if(getStock()-getBackOrder() >= currentOrderS){

    currentOrderS = 0;

}

// if stock is low, do not forget the backorder

else if (getStock()-getBackOrder() < currentOrderS){

}

```

```
if (getStock() <= currentOrderS){  
    this.currentOrderS += getBackOrder();  
}  
  
}  
  
// add the orders to memory  
  
this.orderLine.addLast(currentOrderS);  
  
this.orderHistoryS.addLast(currentOrderS);  
  
// 4. calculate the total cost per unit shipped  
  
// in case total stock is zero  
  
//if (totalStock == 0){  
//    totalStock = 1;  
//}  
}
```

```
        this.totalCostAvg = (this.invCost * this.getStock() + this.shortCost  
        * this.getBackOrder() +  
  
        this.fixedOrderCost *  
        ((this.currentOrderS > 0)?1 : 0)+  
  
        this.purchaseCost * this.shipment);  
  
    }  
  
}
```

```
public static int getShipmentToR1(){  
  
    return suppliedHistory1.getLast();  
  
}  
  
public static int getShipmentToR2(){  
  
    return suppliedHistory3.getLast();  
  
}
```

```
public static int getShipmentToR3(){  
  
    return suppliedHistory3.getLast();  
  
}
```

```
}
```

```
public int isSupplier(){
```

```
    return 1;
```

```
}
```

```
public double getTotalCost(){
```

```
    return this.totalCostAvg;
```

```
}
```

```
public int getTested(){
```

```
    return this.getBackOrder() - this.getStock();
```

```
}
```

```
public int getStock(){
```

```
    return stockHistory.getLast();
```

```
}
```

```
public int getBackOrder(){  
  
    return backHistory.getLast();  
  
}
```

```
public int getShipment(){  
  
    return this.shipment;  
  
}
```

```
public int getCurrentOrderS(){  
  
    return currentOrderS;  
  
}
```

```
public double getFulfilRate(){  
  
    return fulfilRate;
```

```
}

public int getCurrentDemand(){

    return currentDemandS;

}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

public int getTheErr1(){

    return theErr1;

}

public int getTheErr2(){
```

```
    return theErr2;  
  
}
```

```
public int getTheErr3(){  
  
    return theErr3;  
  
}
```

```
public int getPart1_1(){  
  
    return part1_1;  
  
}
```

```
public int getPart2_1(){  
  
    return part2_1;  
  
}
```

```
public int getPart3_1(){
```

```
    return part3_1;
```

```
}
```

```
public int getPart1_2(){
```

```
    return part1_1;
```

```
}
```

```
public int getPart2_2(){
```

```
    return part2_1;
```

```
}
```

```
public int getPart3_2(){
```

```
    return part3_1;
```

```
}
```

```
public int getPart1_3(){
```

```
    return part1_1;
```

}

public int getPart2_3(){

 return part2_1;

}

public int getPart3_3(){

 return part3_1;

}

public int getCurrentOrder1(){

 return currentOrder1;

}

public int getCurrentOrder2(){

 return currentOrder2;

```
}

public int getCurrentOrder3(){

    return currentOrder3;

}

public int getRcvdorder1(){

    return rcvdorder1;

}

public int getRcvdorder2(){

    return rcvdorder2;

}

public int getRcvdorder3(){

    return rcvdorder3;

}

public double getTheSdev1(){

    return theSdev1;

}
```

```
public double getTheSdev2(){
```

```
    return theSdev2;
```

```
}
```

```
public double getTheSdev3(){
```

```
    return theSdev3;
```

```
}
```

```
// get the variance
```

```
public double getTheVar1(){
```

```
    return theSdev1*theSdev1;
```

```
}
```

```
public double getTheVar2(){
```

```
    return theSdev2*theSdev2;
```

```
}
```

```
public double getTheVar3(){
```

```
    return theSdev3*theSdev3;
```

```
    }

public double getMean(){

    return testValue;

}

public int getYtActual1(){

    return YtActual1;

}

public int getYtPredict1(){

    return YtPredict1;

}

public int getYtActual2(){

    return YtActual2;

}

public int getYtPredict2(){

    return YtPredict2;

}
```

```
}
```

```
public int getYtActual3(){
```

```
    return YtActual3;
```

```
}
```

```
public int getYtPredict3(){
```

```
    return YtPredict3;
```

```
}
```

```
public double getTheSdevS1(){
```

```
    return theSdevS1;
```

```
}
```

```
public double getTheSdevS2(){
```

```
    return theSdevS2;
```

```
}
```

```
public double getTheSdevS3(){
```

```
    return theSdevS3;
```

```
}

// get the variance

public double getTheVarS1(){

    return theSdevS1*theSdevS1;

}

public double getTheVarS2(){

    return theSdevS2*theSdevS2;

}

public double getTheVarS3(){

    return theSdevS3*theSdevS3;

}

public double getRoo(){

    return Roo;

}

public int getTotalRcvdOrder(){

    return totalRcvdOrder;
```

```
    }  
  
    public int getTotalPredicted(){  
  
        return totalPredicted;  
  
    }  
  
}
```

File 6: Moderator.java

/* The moderator is designed to handle the relationships between retailers.

* With the addition of this moderator, the process becomes three steps:

*

* 1. retailers get demand and calculate the stocks and backorders

* supplier receives the shipment from upper stream (with time lag)

* and orders from the retailers

* and prepares (predict) order to upper stream (with time lag)

* 2. moderator receives the stock and backorders info from retailers;

* moderator uses the relationship data to process the stock sharing

* moderator updates the stocks & backordres and send them back to retailers

* 3. retailers calculate (predict) the current demands plus the backorders

* and add them to queue as the orders to supplier (with time lag)

*

- * Each complete loop of order process includes three ticks, while each tick
 - * completes one step.

*/

```
package repast.simphony.demo.sc;

import repast.simphony.annotate.AgentAnnot;
import repast.simphony.engine.schedule.ScheduledMethod;

@AgentAnnot(displayName="Moderator")

public class Moderator extends SCagent{

    // constructor

    public Moderator(){

        //initiation here, if necessary
    }
}
```

}

// the generic variables for backorder, stock

// note the capital letters

protected int B1;

protected int B2;

protected int S1;

protected int S2;

protected int doShare = 0; //by default, two retailers do not share

protected int doShare12=0;

protected int doShare13=0;

protected int doShare23=0;

protected int bOrder1;

```
protected int bOrder2;

protected int bOrder3;

protected int stock1;

protected int stock2;

protected int stock3;

// the logic below can only handle small number of nodes.

// a more elegant algorithm may be developed in the later version

// to handle much larger (or any if possible) number of nodes

@ScheduledMethod(start=1, interval=1)

public void Moderating(){

    if (getTickCount() % 4 == 1){

        // do nothing

    }

}
```

```
else if (getTickCount() % 4 == 2){

    // Note this is the tick #2 in the period

    // reserved just for the moderator

    // get the customizable parameters

    // in fact the ratio parameters

    getInitParameters();

/* skip this moderator

Ratio12 = 0.0;

Ratio13 = 0.0;

Ratio23 = 0.0;

*/



// get the current stock for each retailer

stock1 = RetailerAgent.getStock();
```

```
stock2 = RetailerAgent2.getStock();

stock3 = RetailerAgent3.getStock();

// get the current backorder for each retailer

bOrder1 = RetailerAgent.getBackOrder();

bOrder2 = RetailerAgent2.getBackOrder();

bOrder3 = RetailerAgent3.getBackOrder();

// remove the last record of stock and backorder

// so that the updated could be added back later

// after the interaction processes that follow

RetailerAgent.stockHistory.removeLast();

RetailerAgent2.stockHistory.removeLast();

RetailerAgent3.stockHistory.removeLast();
```

```

RetailerAgent.backHistory.removeLast();

RetailerAgent2.backHistory.removeLast();

RetailerAgent3.backHistory.removeLast();

//first get the values of doShare

doShare12=getDoShare(Ratio12, bOrder1, bOrder2);

doShare13=getDoShare(Ratio13, bOrder1, bOrder3);

doShare23=getDoShare(Ratio23, bOrder2, bOrder3);

// if all the doShare values are zero,
(doShare12+doShare13+doShare23==0)

// no one shares, all the backorder and stock data pass through w/o
change;

// if (doShare12+doShare13+doShare23==3), this is not possible.

// there is a special case where all three retailers become one

// this should be handled separately as one supplier, one retailer,

```

```
// which is reduced to a simple case, and is basically covered in the  
  
// validation part of the essay (the one on one relationship)  
  
// this may be a case of joint venture or alliance.  
  
//  
  
// Another type of cases are also that only two are ALWAYS  
sharing,  
  
// and one is left out. These latter special cases may be taken cared  
of  
  
// by altering the value of the ratios at run time.  
  
//  
  
// all sharing schemes are independent, meaning, for example,  
  
// if R1 shares with both R2 and R3, R2 and R3 do not share by  
default.  
  
// if only two are sharing  
  
if(doShare12+doShare13+doShare23==1){
```

```
if(doShare12 == 1){

    // the other two values are zero

    // only R1 and R2 are sharing

    // process the sharing of stocks here

    sharedBy2(bOrder1, bOrder2, stock1, stock2);

    this.bOrder1 = B1;

    this.bOrder2 = B2;

    this.stock1 = S1;

    this.stock2 = S2;

    restoreInit();

}
```

```
else if(doShare13 == 1){

    // the other two values are zero

    // only R1 and R3 are sharing
```

```
// process the sharing of stocks here

sharedBy2(bOrder1, bOrder3, stock1, stock3);

this.bOrder1 = B1;

this.bOrder3 = B2;

this.stock1 = S1;

this.stock3 = S2;

restoreInit();

}

else if(doShare23 == 1){

    // the other two values are zero

    // only R2 and R3 are sharing

    // process the sharing of stocks here

    sharedBy2(bOrder2, bOrder3, stock2, stock3);

    this.bOrder2 = B1;

    this.bOrder3 = B2;
```

```

        this.stock2 = S1;

        this.stock3 = S2;

        restoreInit();

    }

}

else if(doShare12+doShare13+doShare23==2){

    // one node is sharing with two other nodes

    if(doShare12==0){    // R1 and R2 are not sharing

        // R1/R3 and R2/R3 are sharing, R3 is the center

        // R3 picks the one with higher ratio to share first,

        // if not enough, more from the 2nd choice

        if(Ratio13 > Ratio23){

            // R1/R3 first

            sharedBy2(bOrder1, bOrder3, stock1,
stock3);

```

```
        this.bOrder1 = B1;

        this.bOrder3 = B2;

        this.stock1 = S1;

        this.stock3 = S2;

        restoreInit();

// then R2/R3

// check if R2/R3 are still available for

sharing

if (getStillShare(bOrder2, bOrder3) == 1){

    sharedBy2(bOrder2, bOrder3, stock2,
stock3);

        this.bOrder2 = B1;

        this.bOrder3 = B2;

        this.stock2 = S1;

        this.stock3 = S2;
```

```
        }

    else {

        //do nothing

    }

restoreInit();

}

else if(Ratio13 <= Ratio23){

    // R2/R3 first

    sharedBy2(bOrder2, bOrder3, stock2,
stock3);

    this.bOrder2 = B1;

    this.bOrder3 = B2;

    this.stock2 = S1;

    this.stock3 = S2;

    restoreInit();
}
```

```
// then R1/R3

// check if R1/R3 are still available for
sharing

if (getStillShare(bOrder1, bOrder3) == 1){

    sharedBy2(bOrder1, bOrder3, stock1,
stock3);

    this.bOrder1 = B1;

    this.bOrder3 = B2;

    this.stock1 = S1;

    this.stock3 = S2;

}

else {

    //do nothing

}
```

```

        restoreInit();

    }

}

// same logic is applied to the other cases respectively

// may be replaced by some method, to be worked later

else if(doShare13==0){

    // R1/R2 and R2/R3 are sharing, R2 is the center

    // R2 picks the one with higher ratio to share first,

    // (if not enough, more from the 2nd choice - 2 b

implemented)

    // method call sharedValues(...)

    if(Ratio12 > Ratio23){

        // R1/R2 first

        sharedBy2(bOrder1, bOrder2, stock1,
stock2);

```

```
        this.bOrder1 = B1;

        this.bOrder2 = B2;

        this.stock1 = S1;

        this.stock2 = S2;

        restoreInit();

// then R2/R3

// check if R2/R3 are still available for

sharing

if (getStillShare(bOrder2, bOrder3) == 1){

    sharedBy2(bOrder2, bOrder3, stock2,
stock3);

        this.bOrder2 = B1;

        this.bOrder3 = B2;

        this.stock2 = S1;

        this.stock3 = S2;
```

```
    }

    else {

        //do nothing

    }

    restoreInit();

}

else if(Ratio13 <= Ratio23){

    // R2/R3 first

    sharedBy2(bOrder2, bOrder3, stock2,
stock3);

    this.bOrder2 = B1;

    this.bOrder3 = B2;

    this.stock2 = S1;

    this.stock3 = S2;

    restoreInit();

}
```

```
// then R1/R2

// check if R1/R2 are still available for
sharing

if (getStillShare(bOrder1, bOrder2) == 1){

    sharedBy2(bOrder1, bOrder2, stock1,
stock2);

    this.bOrder1 = B1;

    this.bOrder2 = B2;

    this.stock1 = S1;

    this.stock2 = S2;

}

else {

    //do nothing

}

restoreInit();
```

```

    }

}

else if(doShare23==0){

    // R1/R2 and R1/R3 are sharing, R1 is the center

    // R1 picks the one with higher ratio to share first,

    // (if not enough, more from the 2nd choice - 2 b

implemented)

    // method call sharedValues(...)

    if(Ratio12 > Ratio13){

        // R1/R2 first

        sharedBy2(bOrder1, bOrder2, stock1,
stock2);

        this.bOrder1 = B1;

        this.bOrder2 = B2;

        this.stock1 = S1;

        this.stock2 = S2;

```

```
restoreInit();

// then R1/R3

// check if R1/R3 are still available for
sharing

if (getStillShare(bOrder1, bOrder3) == 1){

    sharedBy2(bOrder1, bOrder3, stock1,
stock3);

    this.bOrder1 = B1;

    this.bOrder3 = B2;

    this.stock1 = S1;

    this.stock3 = S2;

}

else {

    //do nothing

}
```

```
        restoreInit();

    }

else if(Ratio13 <= Ratio23){

    // R1/R3 first

    sharedBy2(bOrder1, bOrder3, stock1,
stock3);

    this.bOrder1 = B1;

    this.bOrder3 = B2;

    this.stock1 = S1;

    this.stock3 = S2;

    restoreInit();

    // then R1/R2

    // check if R1/R2 are still available for
sharing
```

```
        if (getStillShare(bOrder1, bOrder2) == 1){  
  
            sharedBy2(bOrder1, bOrder2, stock1,  
stock2);  
  
            this.bOrder1 = B1;  
  
            this.bOrder2 = B2;  
  
            this.stock1 = S1;  
  
            this.stock2 = S2;  
  
        }  
  
        else {  
  
            //do nothing  
  
        }  
  
        restoreInit();  
  
    }  
  
}
```

// the above code will give the new stock and backorder
info

// the retailers update their stocks and backorders
accordingly

// using the static variables from here

// the supplier gets the adjusted order info from here

// and pass-through the Moderator the order to the retailers

}

else if(doShare12+doShare13+doShare23==3){

// local variable

int total;

total = stock1+ stock2+stock3 -

(bOrder1+bOrder2+bOrder3);

// if total available stock after removing overstock is greater
than zero

// all backorders will be fulfilled this period

// and the rest of the total stock will be divided by 2

if (total >=0){

stock1 = total / 3;

stock2 = total / 3;

stock3 = total / 3;

bOrder1 = 0;

bOrder3 = 0;

bOrder3 = 0;

}

// if total available stock is less than overstock

// then both stocks will be set to zero and overstock be
splitted

else if (total <0){

bOrder1 = total / (-3);;

bOrder3 = total / (-3);;

bOrder3 = total / (-3);;

stock1 = 0;

stock2 = 0;

stock3 = 0;

}

}

// now we have the new stock and backorder data

```
// need to feed them back to the retailers

// for all other cases not mentioned or processed

// the original stock and backorder data is added back to the
memory

addUpdatedBack();

}

else if (getTickCount() % 4 == 3){

    // do nothing

}

else if (getTickCount() % 4 == 0){

    // do nothing

}

}
```

```

// get the value of doShare (only takes value 1 or 0)

// if both are sharing, then only one has positive backorder,
// otherwise they will not share,
// no sharing when both backorder is zero, and
// no sharing when both backorder is positive

public int getDoShare(double r, int b1, int b2){

    doShare = (int)Math.floor(Math.random() + r);

    if (doShare == 1){

        if ((b1==0 && b2==0)|| (b1>0 && b2>0)){

            // nothing to share

            doShare = 0;

        }

    }

    if (r == 1.0){

        doShare = 1;

    }

}

```

```
    return doShare;

}

// still sharing? Re-check it.

public int getStillShare(int b1, int b2){

    if ((b1==0 && b2==0)||(b1>0 && b2>0)){

        // nothing to share

        doShare = 0;

    }

    else {

        doShare = 1;

    }

    return doShare;

}
```

```
// method to calculate the backorders and stocks after sharing
```

```
public void sharedBy2(int b1, int b2, int s1, int s2){
```

```
    this.B1=b1;
```

```
    this.B2=b1;
```

```
    this.S1=s1;
```

```
    this.S2=s2;
```

```
    *****/
```

```
// if both are sharing, then only one has positive backorder
```

```
// and only one has positive stock (implied)
```

```
// otherwise they will not share
```

```
if (B1>0){ // meaning S1 == 0, S2>=0, B2 == 0
```

```
    if(B1>S2){
```

```
        B1-=S2;
```

S2=0;

}

else if(B1<=S2){

S2=B1;

B1=0;

}

}

else if(B2>0){ // meaning S2 == 0, S1>=0, B1 == 0

if(B2>S1){

B2=S1;

S1=0;

}

else if(B2<=S1){

S1=B2;

B2=0;

```

    }

}

/*this simpler algorithm simply split the stock or backorder between two*/

/*
// Local variable

int total = S1+S2-B1-B2;

// if total available stock after removing overstock is greater than zero

// all backorders will be fulfilled this period

// and the rest of the total stock will be divided by 2

if (total >= 0) {

    S1 = total / 2;

    S2 = total / 2;

    B1=0;
}

```

```
B2=0;

}

// if total available stock is less than overstock

// then both stocks will be set to zero and overstock be splitted

else if (total < 0){

    S1=0;

    S2=0;

    B1 = total / (-2);

    B2 = total / (-2);

}

*/



}

// restore the initial class variables

public void restoreInit(){
```

```
B1=0;
```

```
B2=0;
```

```
S1=0;
```

```
S2=0;
```

```
}
```

```
// add values of updated stock and backorder back to retailers' memory
```

```
public void addUpdatedBack(){
```

```
RetailerAgent.stockHistory.addLast(stock1);
```

```
RetailerAgent2.stockHistory.addLast(stock2);
```

```
RetailerAgent3.stockHistory.addLast(stock3);
```

```
RetailerAgent.backHistory.addLast(bOrder1);
```

```
RetailerAgent2.backHistory.addLast(bOrder2);
```

```
RetailerAgent3.backHistory.addLast(bOrder3);
```

```
}
```

```
public int getDoShare12(){  
    return doShare12;  
}  
  
public int getStock1(){  
    return stock1;  
}  
}
```

File 7: RetailerAgent.java

```
package repast.simphony.demo.sc;

//import java.util.ArrayList;

import java.util.LinkedList;

import repast.simphony.engine.environment.RunEnvironment;

import repast.simphony.engine.schedule.ScheduledMethod;

import repast.simphony.parameter.Parameters;

import repast.simphony.random.RandomHelper;

import repast.simphony.annotate.AgentAnnot;

/*************************************************************************/
```

/* The following is the working code for validation part of the dissertation.

* note: to reactivate this code, need to also remove the marks in the middle

* of the code which marks "continues..."

```
@AgentAnnot(displayName="Retailer Agent")
```

```
public class RetailerAgent extends SCagent {
```

```
    //the current demand generated
```

```
    protected int currentDemand;
```

```
    // previous demand for retailer
```

```
    protected int prevDemand;
```

```
    //shipment received by retailer agent
```

```
    protected int shipment;
```

```
//total stock (inventory) on hand

protected static int stock=0;

//total backordered quantity

protected static int backOrder=0;

protected static int sharedErr1;

// this is for calculating the average cost only

protected int totalStock=0;

// the storage for orders placed by the retailer agent

protected LinkedList<Integer> orderHistoryR1 = new LinkedList<Integer>();

// the storage for received shipment from the supplier agent

//protected static ArrayList shipRcvHistory = new ArrayList();

//Orders from down stream in the queue

protected LinkedList<Integer> orderLine = new LinkedList<Integer>();

// Orders to upper stream in the queue
```

```
protected static LinkedList<Integer> orderLineR1 = new LinkedList<Integer>();  
  
// The storage for keeping the history of demand  
  
protected LinkedList<Integer> demandStream=new LinkedList<Integer>();  
  
// the storage for keeping the history of total cost per unit  
  
protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();  
  
  
  
  
// The seed for generating demand  
  
//protected int theDemandSeed;  
  
  
  
  
protected int testValue;  
  
  
  
  
public RetailerAgent(){  
  
    //the constructor, initialize values  
  
    final Double Sdev=30.0;  
  
    // The seed for generating demand
```

```

final int theDemandSeed = 1;

RandomHelper.setSeed(theDemandSeed);

// get the Normal distribution values (zero mean, constant variance)

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

// get the customizable parameters

getInitParameters();

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

currentDemand = initialDemandAmount;

```

```

        demandStream.addLast(this.currentDemand);

        // get the initial retailer's order forecasts

        // and add them to order history memory

        orderLineR1.addLast(this.currentDemand);

        this.orderHistoryR1.addLast(this.currentDemand);

    }

    // calculate the order quantity in the following step()

    // It uses the AR1 process described in Lee, et al. (2000)

    // this hard working agent works from tick 1 with interval of 1

    // each 4 ticks are one period

    @ScheduledMethod(start=1, interval=1)

    public void calculateRetailer(){

        if (getTickCount() % 4 == 1){

```

```

// Calculate the demand (AR1 process) from customers, add the
demand to a container

// more demand forecasting functions to be added

theErr = getNormalValue();

sharedErr1 = theErr;

prevDemand = demandStream.getLast();

// $D(t) = d + R_{\text{oo}} * D(t-1) + \text{theErr}$ 

currentDemand = dValue + (int)(Roo*prevDemand) + theErr;

// add the demand to container

demandStream.addLast(currentDemand);

// now the retailer will do the job

// 1. check/update stock after received the shipment from supplier

this.shipment = SupplierAgent.suppliedHistory1.getLast();

stock += this.shipment;

```

```

totalStock = stock;

/*
 * the backorder handled differently
 */

// 2. fulfill the backorder; and receive customer order and update
inventory

stock = stock - backOrder;

// stock < 0? This case is rare, but needs to be taken care of

if (stock < 0){

    backOrder = (int)Math.abs(stock) + currentDemand;

    stock = 0;

}

else if (stock >= 0){

    stock = stock - currentDemand;

}

// again, stock < 0? This case is a little more likely

```

```
if (stock < 0){  
  
    backOrder = (int)Math.abs(stock);  
  
    stock = 0;  
  
}  
  
else if (stock >= 0){  
  
    backOrder = 0;  
  
}  
  
}  
  
*/  
  
/* continues...  
  
// stock level after sales to the customers  
  
stock = stock - currentDemand;  
  
// stock < 0? hopefully not, backorder means lost sales
```

```
if (stock < 0){  
  
    backOrder = (int)Math.abs(stock);  
  
    stock = 0;  
  
}  
  
else if (stock >= 0){  
  
    backOrder = 0;  
  
}  
  
}  
  
else if (getTickCount() % 4 == 2){  
  
    // do nothing  
  
}  
  
else if (getTickCount() % 4 == 3){  
  
    // 3. calculate order to send to supplier
```

```

// the AR1 formula

this.currentOrder = Math.max(0,
currentDemand+(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo,
(Elle+1))/(1-Roo)));

//this.currentOrder += backOrder;

//this.currentOrder = Math.max(0, currentOrder+backOrder-stock);

// if stock is high, set order to zero

if(stock-backOrder >= currentOrder){

    currentOrder = 0;

}

//if (stock<=0){

//    this.currentOrder += backOrder;

//}

orderLineR1.addLast(currentOrder);

```

```

this.orderHistoryR1.addLast(currentOrder);

//4. calculate the total cost per unit in stock

// in case total stock is zero

if (totalStock == 0){

    totalStock = 1;

}

this.totalCostAvg = (this.invCost * stock + this.shortCost *

backOrder +

        this.fixedOrderCost *

((this.currentOrder > 0)?1 : 0) +

        this.purchaseCost * this.shipment);

//this.testValue=(Integer)orderHistory.get(getTickCount());

```

```
}
```

```
else if (getTickCount() % 4 == 0){
```

```
// do nothing
```

```
}
```

```
}
```

```
public int getShipment(){
```

```
    return shipment;
```

```
}
```

```
public int getStock(){
```

```
    return stock;
```

```
}
```

```
public int getBackOrder(){
```

```
    return backOrder;
```

```
}
```

```
public LinkedList<Integer> getOrderHistory(){
```

```
    return orderHistoryR1;
```

```
}
```

```
public static LinkedList<Integer> getOrderLine(){
```

```
    return orderLineR1;
```

```
}
```

```
public static int getSharedErr1(){
```

```
    return sharedErr1;
```

```
}
```

```
public int isRetailer(){
```

```
    return 1;
```

```
}
```

```
public double getTotalCostAvg(){
```

```
    return this.totalCostAvg;
```

```
}
```

```
public int getPrevDemand(){
```

```
    return prevDemand;
```

```
}
```

```
public int getCurrentOrder(){
```

```
    return this.currentOrder;
```

```
}
```

```
public int getTheErr(){
```

```
        return theErr;  
    }  
  
    public int getCurrentDemand()  
  
        return currentDemand;  
  
    }  
  
    // test the generated demand  
  
    // public int getTestValue(){  
  
        //      return testValue;  
  
    //}  
  
    // get the normal demand value  
  
    public int getNormalValue(){  
  
        return (Integer)this.norm  
    }  
}
```

```
// get the uniform demand value

//public int getUniformValue(){

//    return (Integer)this.uniformStream.get(getTickCount());

//}

*/



/*********************/



/*
 * The following is the code for testing the interactions
 *
 */

*/
```

```
@AgentAnnot(displayName="Retailer Agent")

public class RetailerAgent extends SCagent {

    //the current demand generated
    protected int currentDemand;

    // previous demand for retailer
    protected int prevDemand;

    //shipment received by retailer agent
    protected int shipment;

    //total stock (inventory) on hand
    protected static int stock=0;

    // the memory that keeps the history of stock levels
    protected static LinkedList<Integer> stockHistory = new LinkedList<Integer>();

    //total backordered quantity
```

```
protected static int backOrder=0;

// the memory that keeps the history of backorder levels

protected static LinkedList<Integer> backHistory = new LinkedList<Integer>();

// The error term from the demand

protected int theErr;

protected static int sharedErr1;

// this is for calculating the average cost only

protected int totalStock=0;

// the storage for orders placed by the retailer agent

protected LinkedList<Integer> orderHistoryR1 = new LinkedList<Integer>();

// the storage for received shipment from the supplier agent

//protected static ArrayList shipRcvHistory = new ArrayList();

//Orders from down stream in the queue
```

```
protected LinkedList<Integer> orderLine = new LinkedList<Integer>();  
  
// Orders to upper stream in the queue  
  
protected static LinkedList<Integer> orderLineR1 = new LinkedList<Integer>();  
  
// The storage for keeping the history of demand  
  
protected LinkedList<Integer> demandStream=new LinkedList<Integer>();  
  
// the storage for keeping the history of total cost per unit  
  
protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();  
  
// The seed for generating demand  
  
//protected int theDemandSeed;  
  
protected int testValue;  
  
public RetailerAgent(){  
    //the constructor, initialize values
```

```

// get the customizable parameters

getInitParameters();

final Double Sdev=Sdev1;

// The seed for generating demand

final int theDemandSeed = 1;

RandomHelper.setSeed(theDemandSeed);

// get the Normal distribution values (zero mean, constant variance)

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

}

```

```
Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

currentDemand = initialDemandAmount;

demandStream.addLast(this.currentDemand);

// get the initial retailer's order forecasts

// and add them to order history memory

orderLineR1.addLast(this.currentDemand);

this.orderHistoryR1.addLast(this.currentDemand);

// inital stock

stockHistory.addLast(initialDemandAmount);

// initial backorder

backHistory.addLast(initialDemandAmount);

}
```

```

// calculate the order quantity in the following step()

// It uses the AR1 process described in Lee, et al. (2000)

// this hard working agent works from tick 1 with interval of 1

// each 4 ticks are one period

@ScheduledMethod(start=1, interval=1)

public void calculateRetailer(){

    if (getTickCount() % 4 == 1){

        // Calculate the demand (AR1 process) from customers, add the
        demand to a container

        // more demand forecasting functions to be added

        theErr = getNormalValue();

        sharedErr1 = theErr;
    }
}

```

```

// get the current d value

dValue = (int)(dValue*(1+growthRate)+0.9);

prevDemand = demandStream.getLast();

// $D(t) = d + R_{t-1} * D(t-1) + \text{theErr}$ 

currentDemand = dValue + (int)(Roo*prevDemand) + theErr;

// add the demand to container

demandStream.addLast(currentDemand);

// now the retailer will do the job

// 1. check/update stock after received the shipment from supplier

this.shipment = SupplierAgent.suppliedHistory1.getLast();

//shipment = 100;

// get the current stock

stock = getStock();

stock += this.shipment;

```

```
totalStock = stock;

//retrieve the back order

backOrder = getBackOrder();

// the backorder handled differently

// 2. fulfill the backorder; and receive customer order and update
inventory

// since the quantity delivered to customer does not really matter in
the calculation

// thus they are not kept in the memory in the implementation,
though they can be.

stock = stock - backOrder;

// stock < 0? This case is rare, but needs to be taken care of

if (stock < 0){

    backOrder = (int)Math.abs(stock) + currentDemand;
```

```
stock =0;
```

```
}
```

```
else if (stock>=0){
```

```
    stock = stock - currentDemand;
```

```
// again, stock < 0? This case is a little more likely
```

```
if (stock < 0){
```

```
    backOrder = (int)Math.abs(stock);
```

```
    stock = 0;
```

```
}
```

```
else if (stock >= 0){
```

```
    backOrder = 0;
```

```
}
```

```
}
```

```
/* the simplest way of handling backorders

// stock level after sales to the customers

stock = stock - currentDemand;

// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}

else if (stock >= 0){

    backOrder = 0;

}

*/

// store the stock and backorder data to memory

stockHistory.addLast(stock);
```

```
    backHistory.addLast(backOrder);

    // the backorders is back to initial state

    backOrder = 0;

    // the stock is back to initial state

    stock = 0;

}

else if (getTickCount() % 4 == 2){

    // do nothing

}

else if (getTickCount() % 4 == 3){

    // 3. calculate order to send to supplier
```

```

// the AR1 formula

this.currentOrder = Math.max(0, currentDemand +
(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo, (Elle+1)))/(1-Roo)));

currentDemand = 0;

prevDemand = 0;

//this.currentOrder += backOrder;

//this.currentOrder = Math.max(0, currentOrder+backOrder-stock);

// if stock is high, set order to zero

if(getStock()-getBackOrder() >= currentOrder){

    currentOrder = 0;

}

// if stock is low, do not forget the backorder

else if (getStock()-getBackOrder() < currentOrder){

    if (getStock() <= currentOrder){


```

```
        this.currentOrder += getBackOrder();

    }

}

orderLineR1.addLast(currentOrder);

this.orderHistoryR1.addLast(currentOrder);

//4. calculate the total cost per unit in stock

// in case total stock is zero

if (totalStock == 0){

    totalStock = 1;

}

this.totalCostAvg = (this.invCost * getStock() + this.shortCost *

getBackOrder() +
```

```
        this.fixedOrderCost *  
((this.currentOrder > 0)?1 : 0)+  
  
        this.purchaseCost * this.shipment);  
  
//this.testValue=(Integer)orderHistory.get(getTickCount());  
  
}  
  
else if (getTickCount() % 4 == 0){  
  
    // do nothing  
  
}  
  
}  
  
public int getShipment(){  
  
    return shipment;  
  
}
```

```
public static int getStock(){

    return stockHistory.getLast();

}

public static int getBackOrder(){

    return backHistory.getLast();

}

public LinkedList<Integer> getOrderHistory(){

    return orderHistoryR1;

}

public static LinkedList<Integer> getOrderLine(){

    return orderLineR1;

}
```

```
public static int getSharedErr1(){
```

```
    return sharedErr1;
```

```
}
```

```
public int isRetailer(){
```

```
    return 1;
```

```
}
```

```
public double getTotalCostAvg(){
```

```
    return this.totalCostAvg;
```

```
}
```

```
public int getPrevDemand(){
```

```
    return prevDemand;
```

```
}
```

```
public int getCurrentOrder(){

    return this.currentOrder;

}
```

```
public int getTheErr(){

    return theErr;

}
```

```
public int getCurrentDemand(){

    return currentDemand;

}
```

```
// test the generated demand

// public int getTestValue(){

//     return testValue;
```

```

//}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

// get the uniform demand value

//public int getUniformValue(){

//    return (Integer)this.uniformStream.get(getTickCount());

//}

}

```

File 8: RetailerAgent2.java

```
package repast.simphony.demo.sc;
```

```
//import java.util.ArrayList;  
  
import java.util.LinkedList;  
  
  
  
import repast.simphony.engine.environment.RunEnvironment;  
  
import repast.simphony.engine.schedule.ScheduledMethod;  
  
import repast.simphony.parameter.Parameters;  
  
import repast.simphony.random.RandomHelper;  
  
import repast.simphony.annotate.AgentAnnot;  
  
*****
```

/* The following is the working code for validation part of the dissertation.

* note: to reactivate this code, need to also remove the marks in the middle

* of the code which marks "continues..."

```
@AgentAnnot(displayName="Retailer Agent")
```

```
public class RetailerAgent2 extends SCagent {
```

```
    //the current demand generated
```

```
    protected int currentDemand;
```

```
    // previous demand for retailer
```

```
    protected int prevDemand;
```

```
    //shipment received by retailer agent
```

```
    protected int shipment;
```

```
    //total stock (inventory) on hand
```

```
    protected static int stock=0;
```

```
    //total backordered quantity
```

```
    protected static int backOrder=0;
```

```
protected static int sharedErr2;

// this is for calculating the average cost only

protected int totalStock=0;

// the storage for orders placed by the retailer agent

protected LinkedList<Integer> orderHistoryR2 = new LinkedList<Integer>();

// the storage for received shipment from the supplier agent

//protected static ArrayList shipRcvHistory = new ArrayList();

//Orders from down stream in the queue

protected LinkedList<Integer> orderLine = new LinkedList<Integer>();

// Orders to upper stream in the queue

protected static LinkedList<Integer> orderLineR2 = new LinkedList<Integer>();

// The storage for keeping the history of demand

protected LinkedList<Integer> demandStream=new LinkedList<Integer>();
```

```
// the storage for keeping the history of total cost per unit

protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();

// The seed for generating demand

//protected int theDemandSeed;

protected int testValue;

public RetailerAgent2(){

    //the constructor, initialize values

    final Double Sdev=40.0;

    // The seed for generating demand

    final int theDemandSeed = 2;

    RandomHelper.setSeed(theDemandSeed);

    // get the Normal distribution values (zero mean, constant variance)
```

```

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

// get the customizable parameters

getInitParameters();

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

currentDemand = initialDemandAmount;

demandStream.addLast(this.currentDemand);

// get the initial retailer's order forecasts

// and add them to order history memory

```

```

        orderLineR2.addLast(this.currentDemand);

        this.orderHistoryR2.addLast(this.currentDemand);

    }

// calculate the order quantity in the following step()

// It uses the AR1 process described in Lee, et al. (2000)

// this hard working agent works from tick 1 with interval of 1

// each 4 ticks are one period

@ScheduledMethod(start=1, interval=1)

public void calculateRetailer(){

    if (getTickCount() % 4 == 1){

        // Calculate the demand (AR1 process) from customers, add the
        demand to a container

        // more demand forecasting functions to be added
    }
}

```

```

theErr = getNormalValue();

sharedErr2 = theErr;

prevDemand = demandStream.getLast();

//D(t) = d + Roo *D(t-1) + theErr

currentDemand = dValue + (int)(Roo*prevDemand) + theErr;

// add the demand to container

demandStream.addLast(currentDemand);

// now the retailer will do the job

// 1. check/update stock after received the shipment from supplier

this.shipment = SupplierAgent.suppliedHistory2.getLast();

stock += this.shipment;

totalStock = stock;

/* the backorder handled differently

```

```
// 2. fulfill the backorder; and receive customer order and update  
inventory  
  
stock = stock - backOrder;  
  
// stock < 0? This case is rare, but needs to be taken care of  
  
if (stock < 0){  
  
    backOrder = (int)Math.abs(stock) + currentDemand;  
  
    stock = 0;  
  
}  
  
else if (stock>=0){  
  
    stock = stock - currentDemand;  
  
// again, stock < 0? This case is a little more likely  
  
if (stock < 0){  
  
    backOrder = (int)Math.abs(stock);  
  
    stock = 0;
```

```
    }

    else if (stock >= 0){

        backOrder = 0;

    }

}

*/
/* continues...

// stock level after sales to the customers

stock = stock - currentDemand;

// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;
```

```

    }

else if (stock >= 0){

    backOrder = 0;

}

else if (getTickCount() % 4 == 2){

    // do nothing

}

else if (getTickCount() % 4 == 3){

    // 3. calculate order to send to supplier

    // the AR1 formula

    this.currentOrder = Math.max(0,
        currentDemand+(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo,
            (Elle+1))/(1-Roo)));
}

```

```
//this.currentOrder += backOrder;

//this.currentOrder = Math.max(0, currentOrder+backOrder-stock);

// if stock is high, set order to zero

if(stock-backOrder >= currentOrder){

    currentOrder = 0;

}

//if (stock<=0){

//    this.currentOrder += backOrder;

//}

orderLineR2.addLast(currentOrder);

this.orderHistoryR2.addLast(currentOrder);

//4. calculate the total cost per unit in stock
```

```

// in case total stock is zero

if (totalStock == 0){

    totalStock = 1;

}

this.totalCostAvg = (this.invCost * stock + this.shortCost *

backOrder +

        this.fixedOrderCost *

((this.currentOrder > 0)?1 : 0)+

        this.purchaseCost * this.shipment);

//this.testValue=(Integer)orderHistory.get(getTickCount());

}

else if (getTickCount() % 4 == 0){

    // do nothing
}

```

```
}
```

```
}
```

```
public int getShipment(){
```

```
    return shipment;
```

```
}
```

```
public int getStock(){
```

```
    return stock;
```

```
}
```

```
public int getBackOrder(){
```

```
    return backOrder;
```

```
}
```

```
public LinkedList<Integer> getOrderHistory(){
```

```
        return orderHistoryR2;

    }

public static LinkedList<Integer> getOrderLine(){

    return orderLineR2;

}

public static int getSharedErr2(){

    return sharedErr2;

}

public int isRetailer(){

    return 1;

}

public double getTotalCostAvg(){
```

```
    return this.totalCostAvg;  
  
}
```

```
public int getPrevDemand(){  
  
    return prevDemand;  
  
}
```

```
public int getCurrentOrder(){  
  
    return this.currentOrder;  
  
}
```

```
public int getTheErr(){  
  
    return theErr;  
  
}
```

```
public int getCurrentDemand(){

    return currentDemand;

}

// test the generated demand

// public int getTestValue(){

//     return testValue;

//}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

// get the uniform demand value

//public int getUniformValue(){
```

```
//      return (Integer)this.uniformStream.get(getTickCount());  
  
//}  
  
}  
  
*/
```

```
*****
```

```
/*
```

```
* The following is the code for testing the interactions
```

```
*
```

```
*/
```

```
@AgentAnnot(displayName="Retailer Agent")

public class RetailerAgent2 extends SCagent {

    //the current demand generated

    protected int currentDemand;

    // previous demand for retailer

    protected int prevDemand;

    //shipment received by retailer agent

    protected int shipment;

    //total stock (inventory) on hand

    protected static int stock=0;

    // the memory that keeps the history of stock levels

    protected static LinkedList<Integer> stockHistory = new LinkedList<Integer>();

    //total backordered quantity

    protected static int backOrder=0;
```

```
// the memory that keeps the history of backorder levels

protected static LinkedList<Integer> backHistory = new LinkedList<Integer>();

// The error term from the demand

protected int theErr;

protected static int sharedErr2;

// this is for calculating the average cost only

protected int totalStock=0;

// the storage for orders placed by the retailer agent

protected LinkedList<Integer> orderHistoryR2 = new LinkedList<Integer>();

// the storage for received shipment from the supplier agent

//protected static ArrayList shipRcvHistory = new ArrayList();

//Orders from down stream in the queue

protected LinkedList<Integer> orderLine = new LinkedList<Integer>();
```

```
// Orders to upper stream in the queue

protected static LinkedList<Integer> orderLineR2 = new LinkedList<Integer>();

// The storage for keeping the history of demand

protected LinkedList<Integer> demandStream=new LinkedList<Integer>();

// the storage for keeping the history of total cost per unit

protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();

// The seed for generating demand

//protected int theDemandSeed;

protected int testValue;

public RetailerAgent2(){

    //the constructor, initialize values
```

```
// get the customizable parameters

getInitParameters();

final Double Sdev=Sdev2;

// The seed for generating demand

final int theDemandSeed = 2;

RandomHelper.setSeed(theDemandSeed);

// get the Normal distribution values (zero mean, constant variance)

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

Parameters p = RunEnvironment.getInstance().getParameters();
```

```
this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

currentDemand = initialDemandAmount;

demandStream.addLast(this.currentDemand);

// get the initial retailer's order forecasts

// and add them to order history memory

orderLineR2.addLast(this.currentDemand);

this.orderHistoryR2.addLast(this.currentDemand);

// inital stock

stockHistory.addLast(initialDemandAmount);

// initial backorder

backHistory.addLast(initialDemandAmount);

}
```

```
// calculate the order quantity in the following step()

// It uses the AR1 process described in Lee, et al. (2000)

// this hard working agent works from tick 1 with interval of 1

// each 4 ticks are one period

@scheduledMethod(start=1, interval=1)

public void calculateRetailer(){

    if (getTickCount() % 4 == 1){

        // Calculate the demand (AR1 process) from customers, add the
        demand to a container

        // more demand forecasting functions to be added

        theErr = getNormalValue();

        sharedErr2 = theErr;

        // get the current d value
    }
}
```

```

dValue = (int)(dValue*(1+growthRate)+0.9);

prevDemand = demandStream.getLast();

//D(t) = d + Roo *D(t-1) + theErr

currentDemand = dValue + (int)(Roo*prevDemand) + theErr;

// add the demand to container

demandStream.addLast(currentDemand);

// now the retailer will do the job

// 1. check/update stock after received the shipment from supplier

this.shipment = SupplierAgent.suppliedHistory2.getLast();

//shipment = 100;

// get the current stock

stock = getStock();

stock += this.shipment;

totalStock = stock;

```

```
//retrieve the back order  
  
backOrder = getBackOrder();  
  
  
  
// the backorder handled differently  
  
// 2. fulfill the backorder; and receive customer order and update  
inventory  
  
  
// since the quantity delivered to customer does not really matter in  
the calculation  
  
  
// thus they are not kept in the memory in the implementation,  
though they can be.  
  
  
  
  
stock = stock - backOrder;  
  
  
// stock < 0? This case is rare, but needs to be taken care of  
  
if (stock < 0){  
  
    backOrder = (int)Math.abs(stock) + currentDemand;  
  
    stock =0;  
}
```

}

else if (stock>=0){

 stock = stock - currentDemand;

// again, stock < 0? This case is a little more likely

 if (stock < 0){

 backOrder = (int)Math.abs(stock);

 stock = 0;

}

 else if (stock >= 0){

 backOrder = 0;

}

}

```
/* the simplest way of handling backorders

// stock level after sales to the customers

stock = stock - currentDemand;

// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}

else if (stock >= 0){

    backOrder = 0;

}

*/

// store the stock and backorder data to memory
```

```
stockHistory.addLast(stock);

backHistory.addLast(backOrder);

// the backorders is back to initial state

backOrder = 0;

// the stock is back to initial state

stock = 0;

}

else if (getTickCount() % 4 == 2){

// do nothing

}

else if (getTickCount() % 4 == 3){
```

```

// 3. calculate order to send to supplier

// the AR1 formula

this.currentOrder = Math.max(0,
currentDemand+(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo,
(Elle+1))/(1-Roo)));

currentDemand = 0;

prevDemand = 0;

//this.currentOrder += backOrder;

//this.currentOrder = Math.max(0, currentOrder+backOrder-stock);

// if stock is high, set order to zero

if(getStock()-getBackOrder() >= currentOrder){

    currentOrder = 0;

}

// if stock is low, do not forget the backorder

```

```
else if (getStock()-getBackOrder() < currentOrder){
```

```
    if (getStock() <= currentOrder){
```

```
        this.currentOrder += getBackOrder();
```

```
}
```

```
}
```

```
orderLineR2.addLast(currentOrder);
```

```
this.orderHistoryR2.addLast(currentOrder);
```

```
//4. calculate the total cost per unit in stock
```

```
// in case total stock is zero
```

```
if (totalStock == 0){
```

```
    totalStock = 1;
```

```
}
```

```
        this.totalCostAvg = (this.invCost * getStock() + this.shortCost *  
getBackOrder() +  
  
        this.fixedOrderCost * ((this.currentOrder > 0)?1 :  
0)+  
  
        this.purchaseCost * this.shipment);  
  
//this.testValue=(Integer)orderHistory.get(getTickCount());  
}  
  
else if (getTickCount() % 4 == 0){  
  
    // do nothing  
}  
  
}  
  
public int getShipment(){  
  
    return shipment;  
}
```

```
public static int getStock(){

    return stockHistory.getLast();

}

public static int getBackOrder(){

    return backHistory.getLast();

}

public LinkedList<Integer> getOrderHistory(){

    return orderHistoryR2;

}

public static LinkedList<Integer> getOrderLine(){

    return orderLineR2;

}
```

```
public static int getSharedErr2(){
```

```
    return sharedErr2;
```

```
}
```

```
public int isRetailer(){
```

```
    return 1;
```

```
}
```

```
public double getTotalCostAvg(){
```

```
    return this.totalCostAvg;
```

```
}
```

```
public int getPrevDemand(){
```

```
    return prevDemand;
```

```
}
```

```
public int getCurrentOrder(){  
    return this.currentOrder;  
}
```

```
public int getTheErr(){  
    return theErr;  
}
```

```
public int getCurrentDemand(){  
    return currentDemand;  
}
```

```
// test the generated demand
```

```
// public int getTestValue(){
```

```
//      return testValue;  
  
//}  
  
  
// get the normal demand value  
  
public int getNormalValue(){  
  
    return (Integer)this.normalStream.get(getTickCount());  
  
}  
  
  
  
  
  
// get the uniform demand value  
  
//public int getUniformValue(){  
  
    //      return (Integer)this.uniformStream.get(getTickCount());  
  
//}  
  
}
```

File 9:

```
package repast.simphony.demo.sc;

//import java.util.ArrayList;

import java.util.LinkedList;

import repast.simphony.engine.environment.RunEnvironment;

import repast.simphony.engine.schedule.ScheduledMethod;

import repast.simphony.parameter.Parameters;

import repast.simphony.random.RandomHelper;

import repast.simphony.annotate.AgentAnnot;
```

```
*****
```

```
/* The following is the working code for validation part of the dissertation.
```

```
* note: to reactivate this code, need to also remove the marks in the middle
```

```
* of the code which marks "continues..."
```

```
@AgentAnnot(displayName="Retailer Agent")
```

```
public class RetailerAgent3 extends SCagent {
```

```
    //the current demand generated
```

```
    protected int currentDemand;
```

```
    // previous demand for retailer
```

```
    protected int prevDemand;
```

```
    //shipment received by retailer agent
```

```
    protected int shipment;
```

```
//total stock (inventory) on hand
```

```
protected static int stock=0;
```

```
//total backordered quantity
```

```
protected static int backOrder=0;
```

```
protected static int sharedErr3;
```

```
// this is for calculating the average cost only
```

```
protected int totalStock=0;
```

```
// the storage for orders placed by the retailer agent
```

```
protected LinkedList<Integer> orderHistoryR3 = new LinkedList<Integer>();
```

```
// the storage for received shipment from the supplier agent
```

```
//protected static ArrayList shipRcvHistory = new ArrayList();
```

```
//Orders from down stream in the queue
```

```
protected LinkedList<Integer> orderLine = new LinkedList<Integer>();
```

```
// Orders to upper stream in the queue

protected static LinkedList<Integer> orderLineR3 = new LinkedList<Integer>();

// The storage for keeping the history of demand

protected LinkedList<Integer> demandStream=new LinkedList<Integer>();

// the storage for keeping the history of total cost per unit

protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();

// The seed for generating demand

//protected int theDemandSeed;

protected int testValue;

public RetailerAgent3(){

    //the constructor, initialize values

    final Double Sdev=20.0;
```

```

// The seed for generating demand

final int theDemandSeed = 3;

RandomHelper.setSeed(theDemandSeed);

// get the Normal distribution values (zero mean, constant variance)

RandomHelper.createNormal(0.0, Sdev);

//RandomHelper.createUniform(0, 25);

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

// get the customizable parameters

getInitParameters();

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

```

```

        currentDemand = initialDemandAmount;

        demandStream.addLast(this.currentDemand);

        // get the initial retailer's order forecasts

        // and add them to order history memory

        orderLineR3.addLast(this.currentDemand);

        this.orderHistoryR3.addLast(this.currentDemand);

    }

    // calculate the order quantity in the following step()

    // It uses the AR1 process described in Lee, et al. (2000)

    // this hard working agent works from tick 1 with interval of 1

    // each 4 ticks are one period

    @ScheduledMethod(start=1, interval=1)

    public void calculateRetailer(){

        if (getTickCount() % 4 == 1){

```

```
// Calculate the demand (AR1 process) from customers, add the  
demand to a container  
  
// more demand forecasting functions to be added  
  
theErr = getNormalValue();  
  
sharedErr3 = theErr;  
  
prevDemand = demandStream.getLast();  
  
//D(t) = d + Roo *D(t-1) + theErr  
  
currentDemand = dValue + (int)(Roo*prevDemand) + theErr;  
  
// add the demand to container  
  
demandStream.addLast(currentDemand);  
  
// now the retailer will do the job  
  
// 1. check/update stock after received the shipment from supplier
```

```
this.shipment = SupplierAgent.suppliedHistory3.getLast();

stock += this.shipment;

totalStock = stock;

/* the backorder handled differently

// 2. fulfill the backorder; and receive customer order and update
inventory

stock = stock - backOrder;

// stock < 0? This case is rare, but needs to be taken care of

if (stock < 0){

    backOrder = (int)Math.abs(stock) + currentDemand;

    stock =0;

}

else if (stock>=0){

    stock = stock - currentDemand;

}
```

```
// again, stock < 0? This case is a little more likely

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}

else if (stock >= 0){

    backOrder = 0;

}

/* continues...

// stock level after sales to the customers

stock = stock - currentDemand;
```

```
// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}

else if (stock >= 0){

    backOrder = 0;

}

else if (getTickCount() % 4 == 2){

    // do nothing

}

else if (getTickCount() % 4 == 3){
```

```

// 3. calculate order to send to supplier

// the AR1 formula

this.currentOrder = Math.max(0,
currentDemand+(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo,
(Elle+1))/(1-Roo))));

//this.currentOrder += backOrder;

//this.currentOrder = Math.max(0, currentOrder+backOrder-stock);

// if stock is high, set order to zero

if(stock-backOrder >= currentOrder){

    currentOrder = 0;

}

//if (stock<=0){

//    this.currentOrder += backOrder;

//}


```

```

orderLineR3.addLast(currentOrder);

this.orderHistoryR3.addLast(currentOrder);

//4. calculate the total cost per unit in stock

// in case total stock is zero

if (totalStock == 0){

    totalStock = 1;

}

this.totalCostAvg = (this.invCost * stock + this.shortCost *

backOrder +

        this.fixedOrderCost *

((this.currentOrder > 0)?1 : 0) +

        this.purchaseCost * this.shipment);

```

```
//this.testValue=(Integer)orderHistory.get(getTickCount());  
  
}  
  
else if (getTickCount() % 4 == 0){  
  
    // do nothing  
  
}  
  
}  
  
public int getShipment(){  
  
    return shipment;  
  
}  
  
public int getStock(){  
  
    return stock;  
  
}
```

```
public int getBackOrder(){

    return backOrder;

}

public LinkedList<Integer> getOrderHistory(){

    return orderHistoryR3;

}

public static LinkedList<Integer> getOrderLine(){

    return orderLineR3;

}

public static int getSharedErr3(){

    return sharedErr3;

}
```

```
public int isRetailer(){

    return 1;

}

public double getTotalCostAvg(){

    return this.totalCostAvg;

}

public int getPrevDemand(){

    return prevDemand;

}

public int getCurrentOrder(){

    return this.currentOrder;

}
```

```
public int getTheErr(){

    return theErr;

}

public int getCurrentDemand(){

    return currentDemand;

}

// test the generated demand

// public int getTestValue(){

//     return testValue;

//}

// get the normal demand value

public int getNormalValue(){
```

```
        return (Integer)this.normalStream.get(getTickCount());  
    }  
  
    // get the uniform demand value  
  
    //public int getUniformValue(){  
    //    return (Integer)this.uniformStream.get(getTickCount());  
    //}  
  
}  
  
*/  
*****  
/*
```

* The following is the code for testing the interactions

*

*/

```
@AgentAnnot(displayName="Retailer Agent")
```

```
public class RetailerAgent3 extends SCagent {
```

```
    //the current demand generated
```

```
    protected int currentDemand;
```

```
    // previous demand for retailer
```

```
    protected int prevDemand;
```

```
    //shipment received by retailer agent
```

```
    protected int shipment;
```

```
//total stock (inventory) on hand

protected static int stock=0;

// the memory that keeps the history of stock levels

protected static LinkedList<Integer> stockHistory = new LinkedList<Integer>();

//total backordered quantity

protected static int backOrder=0;

// the memory that keeps the history of backorder levels

protected static LinkedList<Integer> backHistory = new LinkedList<Integer>();

// The error term from the demand

protected int theErr;

protected static int sharedErr3;

// this is for calculating the average cost only

protected int totalStock=0;
```

```
// the storage for orders placed by the retailer agent

protected LinkedList<Integer> orderHistoryR3 = new LinkedList<Integer>();

// the storage for received shipment from the supplier agent

//protected static ArrayList shipRcvHistory = new ArrayList();

//Orders from down stream in the queue

protected LinkedList<Integer> orderLine = new LinkedList<Integer>();

// Orders to upper stream in the queue

protected static LinkedList<Integer> orderLineR3 = new LinkedList<Integer>();

// The storage for keeping the history of demand

protected LinkedList<Integer> demandStream=new LinkedList<Integer>();

// the storage for keeping the history of total cost per unit

protected LinkedList<Double> totalCostHistory = new LinkedList<Double>();

// The seed for generating demand

//protected int theDemandSeed;
```

```
protected int testValue;

public RetailerAgent3(){
    //the constructor, initialize values

    // get the customizable parameters
    getInitParameters();

    final Double Sdev=Sdev3;

    // The seed for generating demand
    final int theDemandSeed = 3;

    RandomHelper.setSeed(theDemandSeed);

    // get the Normal distribution values (zero mean, constant variance)
    RandomHelper.createNormal(0.0, Sdev);

    //RandomHelper.createUniform(0, 25);
```

```

for(int i=0; i<10000; i++){

    //uniformStream.add(i,RandomHelper.getNormal().nextInt());

    normalStream.add(i, RandomHelper.getNormal().nextInt());

}

Parameters p = RunEnvironment.getInstance().getParameters();

this.initialDemandAmount=(Integer)p.getValue("initialDemandAmount");

//initial demand from customers

currentDemand = initialDemandAmount;

demandStream.addLast(this.currentDemand);

// get the initial retailer's order forecasts

// and add them to order history memory

orderLineR3.addLast(this.currentDemand);

this.orderHistoryR3.addLast(this.currentDemand);

// init stock

```

```

    stockHistory.addLast(initialDemandAmount);

    // initial backorder

    backHistory.addLast(initialDemandAmount);

}

// calculate the order quantity in the following step()

// It uses the AR1 process described in Lee, et al. (2000)

// this hard working agent works from tick 1 with interval of 1

// each 4 ticks are one period

@ScheduledMethod(start=1, interval=1)

public void calculateRetailer(){

    if (getTickCount() % 4 == 1){

        // Calculate the demand (AR1 process) from customers, add the
        demand to a container
    }
}

```

```
// more demand forecasting functions to be added
```

```
theErr = getNormalValue();
```

```
sharedErr3 = theErr;
```

```
// get the current d value
```

```
dValue = (int)(dValue*(1+growthRate)+0.9);
```

```
prevDemand = demandStream.getLast();
```

```
// $D(t) = d + R_{t-1} * D(t-1) + \text{theErr}$ 
```

```
currentDemand = dValue + (int)(Roo*prevDemand) + theErr;
```

```
// add the demand to container
```

```
demandStream.addLast(currentDemand);
```

```
// now the retailer will do the job
```

```
// 1. check/update stock after received the shipment from supplier

this.shipment = SupplierAgent.suppliedHistory3.getLast();

//shipment = 100;

// get the current stock

stock = getStock();

stock += this.shipment;

totalStock = stock;

//retrieve the back order

backOrder = getBackOrder();

// the backorder handled differently

// 2. fulfill the backorder; and receive customer order and update

inventory

// since the quantity delivered to customer does not really matter in

the calculation
```

// thus they are not kept in the memory in the implementation,
though they can be.

```
stock = stock - backOrder;
```

```
// stock < 0? This case is rare, but needs to be taken care of
```

```
if (stock < 0){
```

```
    backOrder = (int)Math.abs(stock) + currentDemand;
```

```
    stock = 0;
```

```
}
```

```
else if (stock >= 0){
```

```
    stock = stock - currentDemand;
```

```
// again, stock < 0? This case is a little more likely
```

```
if (stock < 0){
```

```
    backOrder = (int)Math.abs(stock);
```

```
    stock = 0;
```

```
        }

    else if (stock >= 0){

        backOrder = 0;

    }

/* the simplest way of handling backorders

// stock level after sales to the customers

stock = stock - currentDemand;

// stock < 0? hopefully not, backorder means lost sales

if (stock < 0){

    backOrder = (int)Math.abs(stock);

    stock = 0;

}
```

```
else if (stock >= 0){  
  
    backOrder = 0;  
  
}  
  
*/  
  
  
  
// store the stock and backorder data to memory  
  
stockHistory.addLast(stock);  
  
backHistory.addLast(backOrder);  
  
  
  
// the backorders is back to initial state  
  
backOrder = 0;  
  
  
  
  
// the stock is back to initial state  
  
stock = 0;  
  
}
```

```

else if (getTickCount() % 4 == 2){

    // do nothing

}

else if (getTickCount() % 4 == 3){

    // 3. calculate order to send to supplier

    // the AR1 formula

    this.currentOrder = Math.max(0,
        currentDemand+(int)((currentDemand-prevDemand)*Roo*(1-Math.pow(Roo,
            (Elle+1))/(1-Roo))));

    currentDemand = 0;

    prevDemand = 0;

    //this.currentOrder += backOrder;

    //this.currentOrder = Math.max(0, currentOrder+backOrder-stock);
}

```

```
// if stock is high, set order to zero

if(getStock()-getBackOrder() >= currentOrder){

    currentOrder = 0;

}

// if stock is low, do not forget the backorder

else if (getStock()-getBackOrder() < currentOrder){

    if (getStock() <= currentOrder){

        this.currentOrder += getBackOrder();

    }

}

orderLineR3.addLast(currentOrder);

this.orderHistoryR3.addLast(currentOrder);

//4. calculate the total cost per unit in stock
```

```

// in case total stock is zero

if (totalStock == 0){

    totalStock = 1;

}

this.totalCostAvg = (this.invCost * getStock() + this.shortCost *

getBackOrder() + 

        this.fixedOrderCost * ((this.currentOrder > 0)?1 : 

0)+

        this.purchaseCost * this.shipment);

//this.testValue=(Integer)orderHistory.get(getTickCount());

}

else if (getTickCount() % 4 == 0){

    // do nothing
}

```

```
}
```

```
}
```

```
public int getShipment(){
```

```
    return shipment;
```

```
}
```

```
public static int getStock(){
```

```
    return stockHistory.getLast();
```

```
}
```

```
public static int getBackOrder(){
```

```
    return backHistory.getLast();
```

```
}
```

```
public LinkedList<Integer> getOrderHistory(){
```

```
        return orderHistoryR3;

    }

public static LinkedList<Integer> getOrderLine(){

    return orderLineR3;

}

public static int getSharedErr3(){

    return sharedErr3;

}

public int isRetailer(){

    return 1;

}

public double getTotalCostAvg(){
```

```
    return this.totalCostAvg;  
  
}
```

```
public int getPrevDemand(){  
  
    return prevDemand;  
  
}
```

```
public int getCurrentOrder(){  
  
    return this.currentOrder;  
  
}
```

```
public int getTheErr(){  
  
    return theErr;  
  
}
```

```
public int getCurrentDemand(){

    return currentDemand;

}

// test the generated demand

// public int getTestValue(){

//     return testValue;

//}

// get the normal demand value

public int getNormalValue(){

    return (Integer)this.normalStream.get(getTickCount());

}

// get the uniform demand value

//public int getUniformValue(){
```

```
//      return (Integer)this.uniformStream.get(getTickCount());  
  
//}  
  
}
```

File 10: repast.simphony.dataLoader.engine.ClassNameDataLoaderAction_0.xml

```
<string>repast.simphony.demo.sc.ContextCreator</string>
```