A Formal Model of an Electronic Commerce System

by

Kris Hiebert

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree

Master of Science

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

A Formal Model of an Electronic Commerce System

BY

Kris Hiebert

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

Of

MASTER OF SCIENCE

Abstract

This thesis presents a formal model of an electronic commerce system, what issues are involved in the design of an electronic commerce system, and how formal methods can be applied to the electronic commerce domain. Also presented is an architecture for electronic commerce as well as a formal specification of that architecture using the Unified Modeling Language (UML) and the Z specification language. A small prototype of an electronic commerce system based on the architecture is also presented. The thesis concludes by giving recommendations for future research in this field.

List of Original Contributions

1. Sylvanus A. Ehikioya and Kristofer J. Hiebert. A Formal Model of Electronic Commerce. In *First International Conference on Software Engineering, Networking and Parallel and Distributed Computing (SNPD-00)*, Champagne-Ardenne, France, May 2000.

2. Sylvanus A. Ehikioya and Kristofer J. Hiebert. A Formal Specification of an On-line Transaction. In *First International Conference on Software Engineering, Networking and Parallel and Distributed Computing (SNPD-00)*, Champagne-Ardenne, France, May 2000.

3. Sylvanus A. Ehikioya and Kristofer J. Hiebert. Agents Negotiation in Electronic Commerce Transactions. In *First Annual International Conference on Computer and Information Science (ICIS-01)*, The Grosvenor Resort, Orlando, Florida, U.S.A., October 3-5, 2001.

Acknowledgements

The creation of this thesis would not be possible without the help of many people. First, I would like to thank my advisor, Dr. Sylvanus A. Ehikioya, for the assistance, understanding, and coaching he provided. Whether it was editing my latest draft, allowing me to bounce ideas off of him, or lending me books and materials, I could always sense his desire for my success. Without him, it would not have been possible.

I would also like to acknowledge the members of my examining committee, Dr. Peter C. J. Graham and Dr. Robert McLoed. I would like to thank them for their time and assistance in reviewing and critiquing this thesis.

I would like to thank all my friends and family members who encouraged me to move forward with this thesis. The constant questioning of when my thesis was to be complete forced me to work harder and made me believe the work I was doing was important.

Finally, I would like to thank my wife Susan. With eternal patience and an iron will, she supported me from beginning to end and moved me along when I stalled. I consider myself very lucky to have found someone like her to spend the rest of my life with.

Contents

1 Introduction 1

1.1 Benefits of Electronic Commerce 1

1.2 Problem Definition 3

1.3 Formal Methods and Electronic Commerce 6

1.4 Significance of this Thesis 8

1.5 Organization of this Thesis 9

2 Background Information and Related Work 11

2.1 Design Issues in Electronic Commerce 11

2.2 Web-based Architecture for Electronic Commerce 14

2.3 UML and the Z Specification Language 17

2.3.1 The Unified Modeling Language (UML) 17

2.3.2 The Z Specification Language 18

3 A Formal Model of an Electronic Commerce System 23

3.1 Model Description 24

3.1.1 The Buyer 24

3.1.2 The Seller 25

3.1.3 The Electronic Commerce System 27

3.1.4 Support Systems 29

3.2 Unified Modeling Language Description 30

3.2.1 Class Diagrams 31

3.2.2 Use Case Diagrams 41
3.2.3 Sequence Diagrams 43
3.2.4 State Diagrams 52
3.2.5 Activity Diagram 55
3.2.6 Deployment Diagram for the Electronic Commerce System 56
3.3 Z Specification Language Model Description 59
3.3.1 Z Language Specification 59
3.3.2 Z Specification Verification 114

4 E-Commerce System Prototype **121**
4.1 Application Design 121
4.2 Implementation 122
4.2.1 Welcome Screen 122
4.2.2 Buyer Summary Screen 124
4.2.3 Browsing for Goods 124
4.2.4 Check Out with Products 125
4.2.5 Receipt for Completed Purchase 126

5 Conclusions and Future Work **127**
5.1 Conclusions 127
5.2 Summary of Contributions 128
5.3 Future Work 128

A The Z Notation **133**

References **137**

List of Figures

1.1 A Simple Electronic Commerce Architecture [13] 2

2.1 Three-Tier Client/Server Architecture for Electronic Commerce 14

2.2 Query and Retrieval of Product Information (modified from [32]) 16

3.1 Electronic Commerce System Classes 32

3.2 Buyer Classes 33

3.3 Seller Classes 34

3.4 Inventory Classes 35

3.5 Search Classes 36

3.6 Comparaison Classes 37

3.7 Transaction Classes 39

3.8 Payment Classes 40

3.9 Buyer Use Case 42

3.10 Seller Use Case 42

3.11 Buyer Login 43

3.12 Buyer Logout 44

3.13 View Transaction History 44

3.14 Search Inventory 45

3.15 View Product Details 46

3.16 Compare Products 47

3.17 Purchase Product - Part I 48

3.18 Purchase Product - Part II 49

3.19 Purchase Product - Part III 50
3.20 View Sales History 51
3.21 Manage Products and Inventory 52
3.22 Transaction Class State Diagram 53
3.23 Payment Activity Diagram 57
3.24 Electronic Commerce System Deployment Diagram 58
3.25 Z-Eves Syntax and Type Check of Z Specification 116
3.26 Z-Eves Domain Check of Z Specification 117
3.27 Z-Eves Domain Check Proof Example 118
3.28 Z-Eves Domain Check Results 119

4.1 Development using Microsoft Visual Studio .NET 122
4.2 Welcome Screen 123
4.3 Buyer Summary Screen 123
4.4 Searching Product Inventory 124
4.5 Check Out with Products 125
4.6 Receipt for Completed Purchase 126

Chapter 1

Introduction

With the increase in Internet technologies in the past few years, businesses and consumers have begun to move their activities towards the digital domain. This new mode of transactions and services, executed over computer networks and outside of the traditional brick and mortar store, is called electronic commerce. These methodologies and technologies provide businesses and consumers the ability to buy and sell things like physical goods, financial vehicles such as stocks and bonds, and services electronically [22].

The basic idea behind electronic commerce is the exchange of value measured in monetary terms (money) for some products or services over the Internet. Generally, in electronic commerce one party purchases some goods or services from another party. The first party can be called the buyer, purchaser, or client while the latter may be called the seller, provider, or supplier. Besides these two basic entities, there are many other support structures needed to ensure the purchasing transaction is established and completed correctly.

1.1 Benefits of Electronic Commerce

As usage of the Internet increases, more and more businesses and consumers are using electronic commerce to purchase and promote goods and services in new and different ways. Some of the benefits of using electronic commerce include:

1. Automation of Business Processes

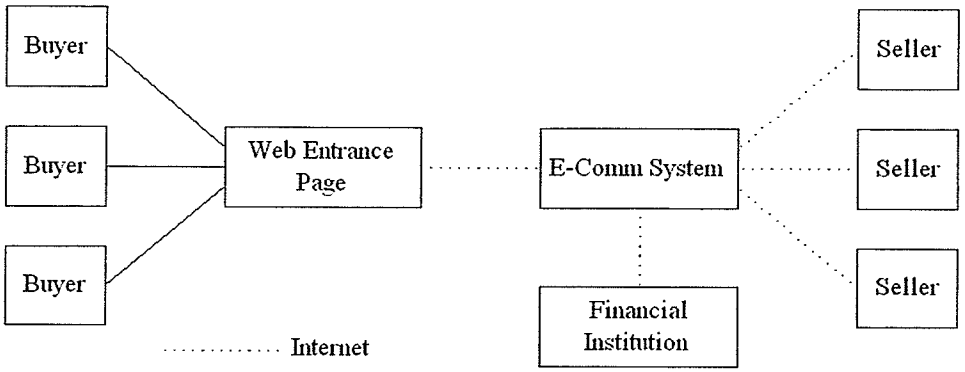

Figure 1.1: A Simple Electronic Commerce Architecture [13]

By using an electronic interface and computerizing the buying process, many of the steps required to complete a transaction can be done all at once, with little effort required by either the consumer or the provider. Billing forms, acquiring shipping information, and payment processing can be done automatically. This offers a savings for both the buyer and seller as they invest less time and effort in a transaction. This automation especially offers a savings in repetitive or bulk transactions.

2. Client Customization

   By using the Internet to provide business services to different clients, different views of those services can be presented automatically and with custom enhancements for each. Customization allows a business to show specific views to different clients. Clients can also customize their shopping experience. By saving information on searches to be run again at a later time or by customizing their interface, clients can shorten their time spent searching for and purchasing goods.

3. Comparative Abilities

   Some of the current electronic commerce systems allow consumers to compare products from different manufacturers based on specified criteria [23]. These systems allow a consumer to contrast different attributes of products against each other and can also recommend certain items based on user input. For example, if a consumer wants to buy a CD-player the user can compare a single model over many different stores or

all the models at a single store or across different stores. These comparative abilities can give the consumer the necessary information to fulfill a purchasing need.

4. Increased Efficiency

   When a business offers its services or products through the Internet, the business and the consumers can save both time and money. The business saves some costs by the automation process as less manpower is needed for tasks that are now handled by computer. Also, different parts of a given transaction between a business and a consumer can occur simultaneously and complete in less time, thus allowing for more transactions at lower cost. Electronic commerce is also more efficient for the consumer because he/she no longer has to physically go from one store to another store to shop. The process of buying the goods customers want is available on their computer and any goods can be delivered directly to them.

5. Global Coverage

   The Internet is reaching more and more people. Everyone connected to the Internet can theoretically purchase any goods that are available through electronic commerce Web sites. This global coverage can allow consumers to acquire rare or hard to find items and allow businesses to expand into untapped marketplaces.

1.2 Problem Definition

To achieve a correct electronic commerce system that is valuable to its owners and users, three problems must be resolved, as in other software systems design; gathering the requirements, specifying the operations of the system, and verifying that the results gained from the first two activities are correct. These problems are examined in details below.

1. Requirements Gathering

   Ehikioya [10, 11], and Ehikioya and Hiebert [13] describe several requirements of electronic commerce systems. For an electronic commerce system to be usable it must be secure, reliable, and correct in its transactions. Much of the information exchanged,

such as credit card numbers, inventory amounts, shipping and billing addresses and client names, are very important and sensitive. The need to maintain the privacy of such information requires that the security of electronic commerce systems be very tight. Users of electronic commerce systems, both consumers and businesses, expect consistent behavior in what they see and use. The system should be functional and on-line as much as possible (except during periods of planned downtime, e.g., for maintenance), as any shutdowns are equivalent to a store closing its doors. Each business must be able to provide services for all its customers as well. If a customer comes from any part of the world or needs things shipped or ordered according to an unconventional timetable, then the provider's electronic commerce solution must be able to handle that request in a timely fashion.

Correctness is an essential trait of any electronic commerce system. Important events in any electronic commerce transaction must be guaranteed to be correct. These important events and attributes include any transfer of funds, current amounts of inventory, and shipping and billing addresses. Correct record keeping by the system is also very important because many tasks are now automated and the system produces a variety of reports such as invoicing and inventory reports. If correctness is not enforced, a business can rapidly lose its clients as they cannot trust the system because there is no guarantee of correct results from transactions.

The design of an electronic commerce system is very complex because of its distribution and the many different technologies that must work together. All heterogeneous components must be identified beforehand and solutions must be found to make them work together seamlessly. This includes existing legacy applications and infrastructure already used by any merchant or organization that wishes to offer electronic services over the internet. Any new system created must integrate tightly with any existing systems and environments, otherwise the organization will not leverage past investments in technology. In addition, electronic commerce applications typically require that many different computer disciplines work together. This includes such areas as security, databases, electronic payment, transaction processing, distributed systems,

artificial intelligence, and multimedia. To understand how to build a good electronic commerce system, all of these aspects must be carefully examined. Defining all of the requirements accurately and completely is a pre-requisite for designing a complete electronic commerce system.

2. Operations Discovery

   Even once all of the requirements have been gathered, the operations of an electronic commerce system must be fully determined. These operations include all the actions taken by the users, owners and administrators of the system. For example, a user (customer) may request a purchase over the Internet, the system processes the order, and creates an invoice and shipping notice. Each of these actions affects, and are affected by, different entities in the system. Understanding each of the operations and its role in the system is very important in designing a holistic electronic commerce system.

3. Verification of the Results of Requirements Gathering and Operations Discovery

   Once all the requirements are determined and the electronic commerce system operations specified, the system may not necessarily be correct. There must be a way to determine if the design is complete and correct. Since these systems are inherently complex to design and proper and correct operation is necessary, formal methods should be used to create and verify the design. By using formal methods to design an electronic commerce system, design related problems can be detected and corrected early and this enhances product quality [37]. With a formal model, formal proofs can be developed to ensure the correctness of the specification created and, thus, the correctness of the electronic commerce system designed.

These three areas (requirements gathering, operations discovery, and verification of results) in building electronic commerce systems are the focus of this thesis.

1.3 Formal Methods and Electronic Commerce

Developing an electronic commerce system is a complex operation involving many different disciplines. These areas include databases, user-interfaces, transaction controls, cryptography and software engineering. When constructing an electronic commerce system, often many people are involved and they must have a clear understanding of the final goal and the steps necessary to achieve the development objectives. Thus, there is a need for clear communication among the development team members. One tool that assists in this direction is the use of a formal method. Formal methods can ease the complexity of designing and implementing electronic commerce solutions by using mathematical notations to precisely specify a design. Some of the benefits of using formal methods include [9, 12] :

1. A clear understanding of the system.

   Formal methods explicitly demonstrate all of the components of the system and their interactions. Someone examining a formal specification can determine many of the salient facts concerning that specification in a short time period. This straightforward definition helps in the later stages of development and allows everyone involved to have a common understanding of the system.

2. The formalization process can reveal ambiguities.

   Any questionable requirement will become evident when using formal methods. Unknown and unclear aspects of a set of requirements are identified as the model is developed because every aspect of the system is fully stated and verified.

3. Incompleteness and contradictions in the informal definition.

   Incompleteness in the requirements can be identified by a "metaphorical hole" that would be present where the information must be included to make the specification complete. If a component has not been thought of yet, it will become apparent when the other components are defined. Any contradictions in the requirements become clear and identifiable as the modeling evolves. The areas responsible for the conflict

can be found and re-examined to determine how to resolve the problem caused by the contradicting requirements.

4. Verification of correctness of the transactions thereby enhancing their reliability.

   Formal methods provide a method to verify any formal design. This verification checks for inconsistencies, contradictions and missing components. Once the design has been checked correctly, the design will be correct. This is very important for electronic commerce transactions because if they are not correct users will not trust the system.

5. Provides an abstract view of the system.

   Formal methods precisely specify the behaviour of a system by concentrating on its functions in order to manage complexity and promote correctness, extensibility, maintainability, reusability, and understanding. The formal design of a system can be viewed from different granularities, showing how each part of a system interacts with others.

Formal methods may be combined with other semi-formal methods to augment the potency of a formal specification. Such semi-formal methods often use graphical representations and natural language to bridge the gap between a plain language problem definition and a provable mathematical specification. Such diagrams aid in showing the interactions, dataflows, process flows, and the states of the objects used in electronic commerce. From these a more formalized tool or language can be used to describe the model in a robust and correctness preserving fashion.

It is important to note that there are many formal methods available and that not all of them are applicable to the electronic commerce domain. A very rigorous specification is necessary to properly represent the many interactions between the different components and actors/users in the electronic commerce system. A rigorous technique is chosen because it can detect possible omissions or ambiguities in any of the processes or definitions. The formal specification language (a component of a formal method) used must accurately describe each static object in an electronic commerce system as well as correctly specify

all of the pre- and post-operation conditions. This rigorous specification is critical in an electronic commerce system as tangible and valuable items are being exchanged and errors are not permissible.

There exist tools that can examine formal specifications and determine if the requirements have been correctly demonstrated [34]. These methods and the tools to prove the correctness of specifications written using the methods are essential for creating a high quality product.

1.4 Significance of this Thesis

This thesis is significant for a number of reasons.

1. By using formal methods with appropriate tools the important properties of electronic commerce transactions can be captured and the correctness and reliability of the transactions can be guaranteed. This formal specification is desirable in the electronic commerce domain due to the fiscal nature of the transactions. The checking of the specifications is achieved, in this thesis, through the use of the Z-Eves [34] type and syntax-checking tool for the Z [35, 37] specification language. The Enterprise Architect [30] tool is used to check the consistency of the Unified Modeling Language(UML) [20] diagrams in this thesis.

2. By rigorous specification of electronic commerce transactions, complexity is reduced and the problems clarified. Since all aspects of electronic commerce are examined in detail, any ambiguities or errors can be identified and corrected at an early stage.

3. A formal specification of an electronic commerce system provides a practical case study and is applicable to real world problems. The type of electronic commerce system that is applicable to this type of approach is made clear while also showing how the integration of different techniques can be used to solve a specific problem.

1.5 Organization of this Thesis

The remainder of this thesis is organized as follows: Chapter 2 examines the characteristics of electronic commerce systems and reviews formal methods. Chapter 3 presents a model of a business to consumer(B2C) electronic commerce system using UML diagrams and Z notation. Chapter 4 describes a prototype implementation of the electronic commerce system created from the model presented in Chapter 3. Finally, Chapter 5 contains my conclusions and a roadmap for future work.

Chapter 2

Background Information and Related Work

This chapter describes the design issues related to electronic commerce as well as an Internet-based architecture for retrieval and presentation of data to the end user that is usable in electronic commerce systems. An overview of formal methods, including the Z specification language and the Unified Modeling Language is then presented.

2.1 Design Issues in Electronic Commerce

In electronic commerce systems, different classifications can be made depending on the users of a particular system. Boll et al. [3] explain two different kinds of electronic commerce systems: business-to-business and private consumer-to-business. Business-to-business systems involve transactions between two companies. Generally these companies already have a business relationship and a large, secure network infrastructure between them. Electronic Data Interchange, or EDI, has been used in the past to enable companies to complete business-to-business of transactions [2, 3]. A private consumer-to-business system is meant for users who generally wish to purchase goods from a business, such as an online retail store. The systems can also be split into two further categories based on the number of users at each end [3]. There can be a single supplier servicing multiple clients or a

multiple suppliers servicing a multiple clients. Maes [31] also describes three models of electronic commerce similar to those described in [3]. These are business-to-business(B2B), business-to-customer(B2C), and customer-to-customer(C2C). The first two correspond to the similarly named formats in [3] but the third, customer-to-customer, is used to describe such things as consumer online auctions. An example of online auctions is eBay [8] where customers place bids on merchandise provided by other customers. The focus of this thesis is on business-to-customer(B2C) electronic commerce. Business-to-consumer electronic commerce is the most common form of electronic commerce. Business-to-business electronic commerce systems are somewhat similar to business-to-customer systems but there are some key differences in the participants involved and the environment in which such a system operates. A business-to-business system generally deals with large corporate customers moving goods and services to one another on a large scale. These shipments, such as inventory for stores or parts needed for manufacturing processes, can be time sensitive and large in number. Customer-to-customer electronic commerce has different requirements than business-to-customer electronic commerce. In a customer-to-customer system, the system itself simply acts as a medium for different members of the public to transact business directly. In the case of auction-style electronic commerce systems, such as eBay, goods are offered for sale by other members of the general public. While a business-to-customer system also offers goods to the public, it embodies the spirit of a more traditional bricks-and-mortar store where goods are offered for sale from a company at fixed prices. Negotiation and bargaining do take place in business-to-customer systems between one consumer and multiple vendors (i.e. via comparison shopping) while in customer-to-customer systems one vendor offers a single item for sale with multiple buyers placing bids towards a final price (i.e. an auction).

These different forms of electronic commerce require that the system under development must satisfy different prerequisites. Guttman et al. [23] state that electronic commerce generally requires such things as security, trust between parties, payment mechanisms, intermediaries, on-line catalogs of product information, some behind-the-scenes management, as well as providing a welcome shopping environment through multimedia. Other similar

requirements proposed by Aoun [1] are secure transactions, practical payment methods, and the use of intelligent agents for product searching. There have been difficulties in realizing these requirements. Bichler [2] cautions that the creation of electronic commerce applications is a risk because of a lack of application-level interoperability, reusable components, and an absence of industry standards.

Many different methods and examples of programming solutions in electronic commerce exist [2, 3, 7, 16, 21, 29, 31, 33]. Component-based design and component-oriented programming are based around a small kernel with features being added via functionality objects called components [2]. A component is a specific piece of functionality that can be accessed by other software through a specified interface. Bichler [2] proposes a component-based system development life-cycle to allow components to be programmed for the electronic commerce environment. Also, Bichler [2] discusses different groups which are trying to promote inter-operability, such as the Open Trading Protocol (OTP), Open Financial Exchange (OFX) and Open Buying on the Internet (OBI). The RMP system [3] is used in rural area to give small- to medium-sized enterprises the ability to trade produce over the Internet. The system is specified as a multi-client multi-server system with many defined processes. These components enable a logical framework to be built from knowledge of the local commercial situation. The RMP system also defines a typical electronic commerce transaction as having the following steps: Searches for available products, places an order, negotiates the price, upon acceptance and delivery issues an invoice and then finally triggers payment. Dogac et al. [7] use workflows to model the electronic commerce market-place. By giving buyers and sellers templates of the transactions to perform, the system is aware of what methods and communications can take place. The model of a marketplace also has defined system objects and specific functions defined for each object. Jennings et al. [29] describe the objects that are needed to accomplish a business process in the ADEPT system while Reich [33] provides an example of an auction market. He [24] provides an Object Z formal specification of an online Bazaar system (a variant of an auction system). Ehikioya and Hiebert [15], Maes et al. [31], and Guttman et al. [21] provide frameworks describing how agents can act to model customer's behaviours in conducting electronic commerce activities.

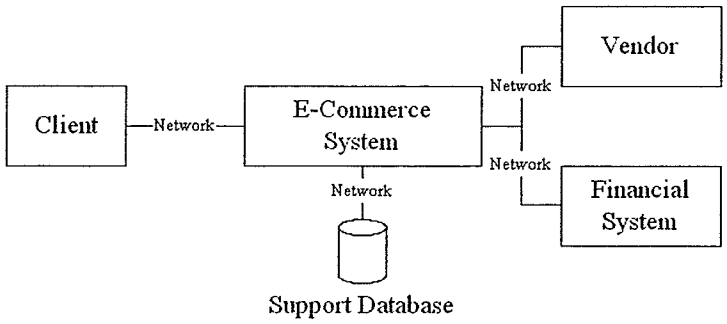

Figure 2.1: Three-Tier Client/Server Architecture for Electronic Commerce

They also provide a set of rules for the agents to follow during negotiation on behalf of the user. Finally, Ehikioya and Walowetz [16] give an exacting overview of a transaction in an electronic commerce system and they describe rules to ensure a transaction is successful and complete.

From the above related work, it is clear that many implementations and ideas have been developed to support electronic commerce. However, most of these are specialized in nature or have a very specific problem domain. There is an absence of a model that can be applied to various electronic commerce scenarios or suited to a multiple markets.

2.2 Web-based Architecture for Electronic Commerce

To provide access to an electronic commerce system, a generic three-tier client/server architecture [32] can be used with some additional features. Figure 2.1 shows a generic three-tier client/server architecture. The components of figure 2.1 are described below.

1. Client

   The client uses a Web browser to access the electronic commerce system. The client can shop at any electronic commerce site on any computer provided access to the Internet is available and the client machine has a browser. If the client is designed to connect only to one electronic commerce system by running an application, it would lose the flexibility gained from the Internet. When the client, electronic commerce server, and support database are viewed as a three-tier system, the client is the first

tier.

2. Electronic Commerce Server

   The electronic commerce server takes the burden of running the actual application from the client machine and allows communication of requests with that client through a Web browser. It also connects to the vendor and financial systems as well as the support databases needed by the system. The electronic commerce server is responsible for transforming any inputs, outputs, and requests into different formats that both the client and support systems can understand and use. In addition, the electronic commerce server is responsible for the initiation of communications for electronic payment for the goods/services ordered by the client. The electronic commerce server is the second tier in a three tier system.

3. Vendor and Financial Systems

   A vendor system contains information about the goods/services offered for sale on the electronic commerce system while the financial system keeps track of all the information needed for each transfer of funds. The vendor and financial systems are distributed over the Internet and are accessible via Web queries. These systems are not considered part of the three-tier client/electronic commerce server system as they are unique entities. Each may be its own N-tier application.

4. Support Databases

   The support databases contain all the information needed by the electronic commerce system while operational. It is local to the electronic commerce system and is the third tier in the three-tier environment.

There are many different steps involved in an electronic commerce transaction. Each of these steps involves different entities in the architecture. For example, products can be requested by a client in Figure 2.2 using the Hypertext Transfer Protocol(HTTP) [5]. A Web server accepts any user input and the specified URL and communicates with the Database

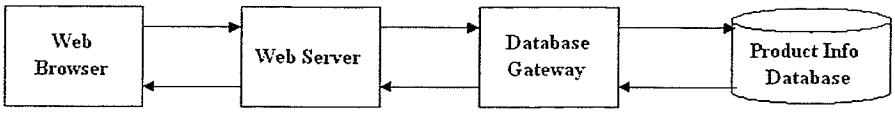

Figure 2.2: Query and Retrieval of Product Information (modified from [32])

Gateway which processes them into a query understandable by the vendor's system. This is usually done with some program such as a CGI [19] (Common Gateway Interface) script.

The data from the vendor is formatted into HTML in the database gateway by another CGI program and given to the Web server which sends the results to the client. There are many other tools that can be used for retrieving data from databases on the Web, including JDBC [3] and ColdFusion [25]. Each of these tools may be considered for an implementation, depending on the specifics of the system being considered.

In an electronic commerce system, there are many different sources of data that may be accessed and modified in a transaction. These sources are distributed over the Internet, hidden behind the Web storefront, and implemented using different database management systems. Because of the many different formats and systems used to represent data, difficulties arise when these systems attempt to share data and work cohesively. This is a data integration problem. The following factors contribute to the difficulty of consolidating Web data sources. According to Ozsu and Valduriez [32], there can be a dynamic number of data sources which may change frequently, the data sources themselves may have different computational speeds, and the data may have varying levels of structure from source to source. Also, Florescu et al. [17] state that the data may be embedded in HTML or may be behind a form interface (such as CGI), so little is usually known about the data source; the different sources are autonomous, and they can evolve into different forms. This means that data that was once accessible to the system may change to a form that is unusable.

An additional problem exists because of the nature of web-based applications such as electronic commerce. When a client uses an electronic commerce system, that client should not have to re-enter an account name and password to access a different pages. This problem arises because HTTP used by a web server is stateless and does not remember the client

information received. To overcome this statelessness problem, a session for that client must be managed by a web-based electronic commerce system to support unlimited shopping and browsing with a single authentication.

In an electronic commerce system there must also be some method of payment from buyer to seller for the goods selected. Many methods exist for this kind of operation; credit cards, on-line bank accounts, and e-cash. The transfer of money must be carefully guarded and must protect the personal and financial information as it travels over the Internet.

2.3 UML and the Z Specification Language

In this thesis, UML and the Z specification language are used to model an electronic commerce system. Each of these languages is described below, giving an overview of the language, features, advantages, and drawbacks.

2.3.1 The Unified Modeling Language (UML)

UML, created in 1997 by the Object Management Group [20, 36], is a system of graphical representations for describing different aspects of object oriented software.

UML comprises nine main diagram types [18]: Activity, Collaboration, Component, Class, Deployment, Object, Sequence, State and Use Case. Activity diagrams are often used to model business processes or activities and are similar in presentation to a logical flow chart. Collaboration and sequence diagrams are similar because they both show the step by step interactions between objects and classes. Component diagrams show the breakdown of code modules and their relationships to one another. Class and object diagrams showcase the entities in the model and their associations. The key difference between object and class diagrams is that class diagrams give a view of the data in the system at rest while object diagrams show a snapshot of an instantiated system. Deployment diagrams show how software is distributed across an enterprise (for distributed systems). State diagrams show how a particular class may change state internally during an operation or process. A use case diagram shows the principal actors in the system and describes the operations that

a user would see between those actors.

With these diagrams, the same information may be represented in different ways providing multiple views of a particular situation. However, this can permit some ambiguity. In addition, different organizations use UML differently and there is no standard usage methodology. Nevertheless, UML is still a very useful tool for describing object oriented software.

2.3.2 The Z Specification Language

The Z specification language has evolved over the past years into its present form. The Z specification language is based on set theory and mathematical logic [35, 37] and includes mathematical types, first order predicate calculus, relations, functions and variables. Z uses these elements as well as constructs called schemas to formally define system requirements. The schemas are used to describe a system piece by piece, capturing both static and dynamic aspects. A schema can be created independently and isolated from others but can also be related and combined to give a greater or differing view of a system. These components of the Z language allow the abstract definition of what a system should do without actually detailing how it will be done. Questions about any of the operations of a system may be answered confidently once a Z-based specification of the system is created because all conditions of that operation have been defined. When natural language is used to describe the operations or states of a system in a Z specification there is much less ambiguity than when natural language is used alone without mathematical backing.

The Z specification language, through available tools such as Z/EVES [34], also allows for the proof of correctness of a specification through type and syntax checking. These proofs are important as they demonstrate that the system requirements have been properly represented and are fundamentally correct. In addition, refinement of any specification is possible by adding further precision to each schema and definition. Such refinement, allows a specification to become closer to the actual implementation of the system and still be provably correct in its nature. Because of the benefits described above and preliminary results in [14], a formal specification model of an electronic commerce system using the

Z specification language is a viable option, especially when used in conjunction with a semi-formal method, such as UML.

One of the main constructs in the Z specification language is the schema. A schema represents an abstract or real component of a system and all of the component's properties. By creating a specification using these schemas many of the complexities of a system can be decomposed into smaller, more manageable units, and can also be abstracted to show relationships between concepts and to encapsulate information.

For the following specification, several conventions are used. These include:

- ? : an input variable
- ! : an output variable
- $'$: a variable that is in a post-operation state (i.e. an updated variable)
- Ξ : identifies that there was no state change in the schema
- Δ : identifies that there was a state change in the schema

Appendix A contains a detailed list of other Z notations used in this thesis. To illustrate the above conventions and others used throughout the specification, some simple examples are now provided.

The two types below are created to represent a concept in a specification. The examples are a definition for a *STRING*, which is simply a collection of characters in some order, and a *STATUS* which may have one of the values in the list following the new type's name.

$[STRING]$
$STATUS ::= start \mid complete \mid error$

There are two methods of representation of schemas, horizontal and vertical. A simple schema, representing some basic information about a customer, represented as a horizontal schema, might be defined as follows:

$Customer \mathrel{\widehat=} [customerID : STRING;\ name : STRING;\ credit\_limit : \mathbb{N}_1]$

Of note in the definition above is the use of the previously defined type *STRING* and the value of the *credit_limit* must be greater than or equal to 1.

Constant values can also be represented in a Z specification by identifying a type and a maximum value. For example, the maximum credit limit of any customer is defined below.

$$LIMIT : \mathbb{N}_1$$
$$\hline$$
$$LIMIT \leq 1000$$

The customer schema can then be redefined to incorporate this maximum value for a credit limit and represented using a vertical schema as follows.

Customer
$$cutomerID : STRING$$
$$name : STRING$$
$$credit\_limit : \mathbb{N}_1$$
$$\hline$$
$$credit\_limit \leq LIMIT$$

Two schemas are defined below to show how state change operations are used in conjunction with schemas to modify information stored in a schema and how information can be passed to, and retrieved from, a schema. The *IncreaseCreditLimit* schema adds some amount to the customer's credit limit and the *GetCustomerName* schema returns the name of the customer.

IncreaseCreditLimit
$$\Delta Customer$$
$$amount? : \mathbb{N}_1$$
$$\hline$$
$$credit\_limit' = credit\_limit + amount?$$

The *IncreaseCreditLimit* schema uses the Δ operator to show that the *Customer* schema has changed in state. The new value of the *credit_limit* is shown by the $'$ token. Finally the ? operator is used to identify an input value to the schema, in this case *amount*, which must be greater than or equal to 1.

GetCustomerName
$$\Xi Customer$$
$$customer\_name! : STRING$$
$$\hline$$
$$customer\_name! = name$$

The *GetCustomerName* schema users the Ξ operator to show that the Customer schema

has not changed but the properties are being accessed to produce some result. The variable holding the output value, *customer_name* in this case, is marked with the ! identifier.

Chapter 3

A Formal Model of an Electronic Commerce System

This chapter presents a model of an electronic commerce system that allows buyers and sellers to interact in an organized and productive manner. This model resembles an ordinary marketplace with many different customers looking for various items and many different vendors with goods for sale. These multiple buyers can interact with the multiple sellers, browsing and searching through product inventories and selecting goods for purchase from different merchants. When this concept is applied to the electronic domain, there are many different customers coming to a single electronic commerce portal for a variety of different goods instead of customers visiting several different vendor sites for the goods they need. An electronic commerce portal can be described as a single entry point (i.e. web page) that allows a client access to a multiple vendor electronic commerce offerings. From a vendor's perspective, this is beneficial as many different potential buyers have the opportunity to browse its products.

The next section describes the components of the model; giving a breakdown of their attributes and operations used in the system and detailing some of the benefits of this model. A detailed analysis of each part of this electronic commerce system is also included in this section and show graphical representations of how the system components interact giving a detailed account of the processes carried out in the virtual marketplace using UML.

Finally, a formal specification of the model using the Z specification language is presented.

3.1 Model Description

This section outlines the model of the electronic commerce system, detailing the major objects. The main entities in the model are the buyer, the seller, the electronic commerce marketplace, and support systems. The details of these entities are provided in this section.

The basic tenet of this electronic commerce model is that there exists some buyers accessing a listing of goods/services they may wish to purchase that are provided by some sellers and that each event is managed by the electronic commerce system. The buyer and seller each have priorities that they wish represented in the system. Buyers may want the facility to find goods/services that interest them quickly and efficiently while sellers want to ensure that their products are viewed and purchased by as many buyers as possible. The electronic commerce system ensures that each business transaction is secure and executes completely in addition to providing a forum for the buyers and sellers to interact. The electronic commerce system needs the support systems' services to complete its tasks successfully.

3.1.1 The Buyer

The buyer is a consumer that wishes to purchase one or more products from any number of sellers through the electronic commerce system. Buyers access the system via a web page portal. A buyer may create a personalized account upon initial visit which is necessary to perform some secure transactions. The buyer then has a unique identifier and password with which he/she accesses the electronic commerce system. Billing information about a buyer including name, address, phone, fax, email, and any other details are kept so that when the buyer purchases goods/services in the system an invoice can be generated and sent. Once an order is made, it must be shipped to some location. The shipping addresses stored for the buyer can contain many addresses. This allows many different destinations to be kept on file and removes the need for re-entry when more than one order is sent to

the same address. Payment information, which is information about the different payment methods the buyer has, is also stored. The electronic commerce system also keeps credit card and bank account numbers, expiry dates and other pertinent related information. A purchase history is also kept for each buyer. This history includes information on each transaction undertaken such as goods purchased, time and date of purchase, method of payment, and billing and shipping addresses.

When a buyer logs in using a unique identifier and password, the virtual marketplace creates a session for the buyer. When the buyer logs out, the session is closed and becomes part of the buyer's history in the electronic commerce system. A buyer can add, modify, or delete his/her own existing shipping addresses and payment methods. The buyer can browse or search his/her transactions in the system to check shipping dates or compare purchases against financial records for accuracy or product warranty information. The buyer also has access to billing and login information and can make changes as necessary, such as changing a billing address or password modification. The buyer can browse for items they wish to purchase using an interface supplied by the virtual marketplace. The buyer can search by product number, type or price. The buyer can search using different criteria and can, at any time, go back or start a new search. Once the buyer finds an item to buy or once he/she has narrowed the criteria by some limiting factors, different products can be compared to determine which would be the optimal selection. The criteria for this comparison can be price, shipping options, or other characteristics of the products. Finally, the buyer can purchase selected products during a session in the electronic commerce system. The buyer pays according to one of the payment methods, and a bill is sent to the billing address, and the products are shipped to the shipping address specified by the buyer.

3.1.2 The Seller

The seller is a corporate entity selling a variety of products. In the system, sellers are represented by a human agent who has access to the seller's login and password information, set up in a similar fashion to the buyer. The system maintains information about the seller; such as name, location, and contact information including customer service phone

numbers and email addresses. For a seller to receive payments for its goods/services sold in the electronic commerce system, some payment information is needed. This information includes account numbers and financial institution information such as address and contact details. The system needs to keep track of what products are available and which seller has them as well as their prices, availability information, and other product characteristics.

When a human agent logs in (for the seller), a session is created to manage the agent's activities in the system. When an agent logs out, the agent's session is closed and all the operations in that session become part of the seller's history in the system. The operations of a seller include modifying corporate information, viewing sales and payment information, searching the master inventory of all products, viewing and comparing products, and managing the products the seller has for sale in the system. The seller can view and modify its corporate information and make any changes to address, profile, contact information, or password attributes. This functionality allows the seller to remotely manage its information. The seller can also view its sales history in the system. A seller can enter a date range, product number, order number, or other keyword to retrieve matching transactions. The seller must also be able to edit payment information. A seller can modify account information such as the account number, the type of the account, and any other information pertaining to the account or financial institution. The seller can browse for items they wish to investigate using the virtual market's interface. The seller can search the product database using different criteria, such as product number, type or price, across not only its own products but those of its competitors in the system. Once the seller narrows its search by some limiting factor, different products can be compared. The criteria for this comparison can be price, shipping options, or different characteristics of products. Sellers can compare products to ensure they remain competitive and to possibly identify any weaknesses in the range of products they offer. The seller can use this information to produce business plans, to move into new markets, or move out of old unprofitable markets/products. Finally, the seller can manage its own products in the system. If the seller needs to add or remove some of its products or modify any of the characteristics of a particular product, the seller does so through the virtual market interface. These operations allow a seller to be in complete

control of its operations and only contact the administrators of the electronic commerce system in the event of an error or if special service is required. The seller is responsible for fulfilling buyer orders received from the electronic commerce system. The seller is expected to ship the ordered items to the buyer in the time frame specified in the publicly available product information.

3.1.3 The Electronic Commerce System

The system that manages the virtual marketplace has many different tasks to complete and large amounts of data to maintain. The electronic commerce system is best described by breaking it down into logical components. Each of these components is briefly described. The marketplace manager must be able to manage buyer and seller sessions, keep a master inventory of products, manage each transaction from beginning to end, manage all the seller financial accounts, handle all queries on the inventory, and allow comparisons between products on behalf of buyers and sellers.

There are two components of the electronic commerce system for managing the account information of buyers and sellers. The buyer session handler is responsible for loading all of the pertinent information about a buyer when needed. This information includes buyer payment methods, the products currently selected by the buyer, and any information required from any of the Web pages in the system previously accessed by the buyer, including any set preferences for viewing information. The seller session handler functions in the same manner as the buyer session handler except that it handles seller information, needs, and requests.

The master inventory contains a list of all the products available to the buyers accessing the system. The master inventory consists of all the products that each seller is offering. The products are managed by each seller and the master inventory keeps track of availability of products and shipping times. The sellers update the master inventory as they add or remove products.

Each time a buyer decides to purchase any products from the virtual marketplace, the buyer initiates a transaction within the system. Each transaction has a definite beginning

and a definite end point. Each transaction must end in some state that will not harm any of the parties in the marketplace. Transactions involve one buyer, the electronic commerce system and one or more sellers. The transaction manager is responsible for each of these transactions. Many different transactions can be ongoing in the system at the same time and the completeness and correctness of each is important to the integrity of the entire system.

The process of purchasing one or more items from the system requires multiple steps. Before this process begins, the buyer must select one or more products from any number of sellers. The buyer is shown all of the relevant details about the selected purchase including the product, quantity, price, shipping charges, and any applicable taxes. This information is reviewed by the buyer and if it is satisfactory, the buyer confirms the purchase. If the buyer does not confirm the purchase, the buyer may remove any erroneous items, send a request for more information to the electronic commerce system in the form of a customer service request, or continue shopping for other products. Once the buyer has confirmed the goods he/she wishes to purchase, the buyer can select a shipping address or can enter a new address to ship to if required. Each product to be purchased can be sent to a different address or all can be sent to the same one. After all goods have been assigned shipping addresses the buyer must select a method of payment for this purchase. In the same manner as choosing shipping addresses, the buyer can use an existing payment method or create a new one. All of the details of this purchase are once again presented to the buyer for confirmation once the payment method has been decided. Once the buyer confirms the finalization of all products in the transaction, payment is processed through the electronic commerce system. The transaction manager checks all the details of the purchase to ensure that they are correct and then delivers the purchase requests to each seller that has one of their products listed in the buyer's purchase.

Once the orders have been sent, the transaction manager builds an invoice for the purchase and records all of the details in its account manager for payment to the seller of the products. An entry is created at this time in the buyer's transaction history for the completed sale and the invoice is presented to the buyer (on their computer screen

and is sent to the email address(es) specified in their buyer information). The seller is responsible for shipping the products to the buyer upon receipt of the order created during the transaction.

Buyers accessing the system through their account may not know the exact name or product number of a product that they wish to purchase. The buyer can access a browse and search function where details such as product type, name, model numbers or other characteristics can be entered and used to find matching products. By using this function, a buyer can quickly find the products they are interested in and easily change the criteria to further narrow the scope to a more specific range of products. This search uses the master inventory of products. Such searches are handled in the electronic commerce system by a search engine.

When using the search function, buyers may find a particular product or types of products they are interested in and want to compare for the lowest price, best combination of features, or some other criteria they desire. The comparison engine in the electronic commerce system is used for this purpose. The comparison engine reports which products match the supplied criteria, ordered by the strength of the matching criteria. In addition to the buyer using this feature, a seller may also use the comparison engine to compare its products with those of its competitors to determine if it is offering a good and fair price, etc., for its own products.

3.1.4 Support Systems

The electronic commerce system requires certain external entities to complete its tasks correctly. These entities assist with the supply of goods to the system and allow for the transfer of funds from a buyer to the seller.

Each seller has one or more suppliers that provide the products that the seller offers for sale. The seller buys a 'lot' of a particular product at an agreed price and at a guaranteed shipping time. This method of purchasing goods/services for sale allows the sellers to determine what products they will offer in the electronic commerce system to buyers and to set a price and shipping time for them. When a product is purchased from a seller, an

order is sent to the supplier to ship the items to the address specified by the buyer. The order gets from the seller to the supplier through different methods, such as mail, fax, email or another business-to-business electronic commerce system.

When a buyer purchases goods or services from a seller, money must be transferred from the buyer to the seller as payment. For this to happen in a correct and secure manner, several third party entities must be involved. The buyer and the seller both have financial institutions which hold their monetary assets. This can be a credit card company, bank, or other such monetary institution. In addition to the financial institutions, a method of authorization and settlement, called a processing gateway, must exist. This processing gateway allows for secure communication between the electronic commerce system and the buyer's and seller's financial institutions. Some current incarnations of these processing gateways include PayPal [26], Verisign [27], and any of the telephone solutions seen in any store that accepts credit or debit cards.

3.2 Unified Modeling Language Description

The Unified Modeling Language (UML) [20] is used to model the system described above (Section 3.1) in graphical form. Class, sequence, use case, state, activity and deployment diagrams provide a better understanding of the electronic commerce system and make clear the connections between the different components. Each type of diagram offers a different view of some components of the electronic commerce system. Class diagrams show a static view of the components of a system and allow the representation of the relationships between classes. Sequence diagrams show the operational steps of a logical task by identifying and linking all the components involved. State diagrams show how a given component changes over time; providing a further level of detail on components requiring transitions over time. Activity diagrams show the logical and programmatic flow of a task and what conditions result in certain outcomes. Finally, deployment diagrams show how a hardware or abstract software implementation of a system will be structured in terms of where the various system components will be placed. These diagrams give a

robust description of an electronic commerce system. The UML diagrams in this thesis were created using Enterprise Architect 3.50 [30].

3.2.1 Class Diagrams

In the UML, class diagrams are used to show the components of a system and the relationships between them. They are static diagrams showing data stored and operations carried out by each component. This electronic commerce system model is described by one central class diagram and then followed by several smaller diagrams detailing more specific components of the system.

Electronic Commerce System Classes

Figure 3.1 shows the major classes in the electronic commerce(EC) system. The EC System class represents the whole system containing the registered buyers and sellers, the buyer and seller sessions that are currently active, the transaction manager for completing purchases, the search engine used to track current searches and the master inventory of products available. Other components include the payment manager and the comparison engine. The operations of the EC System class start and stop the system and add and remove buyers and sellers.

Buyer Classes

Figure 3.2 shows the classes related to buyers in the electronic commerce system. The Buyer class contains the information about a specific buyer that logs into the system. The buyer has a unique ID, information about his/her location and contact details, a non-empty list of shipping addresses and a number of payment methods for products. The Buyer class contains methods for a log in and log out, modification to any of their details except their Buyer ID, viewing past transactions, searching the available inventory, to run comparisons on goods, and to view details of and purchase products. The Buyer Session Handler class is responsible for tracking all the buyers that are in the system. It contains a list of all the open buyer sessions and can open new sessions and close existing ones. The Buyer

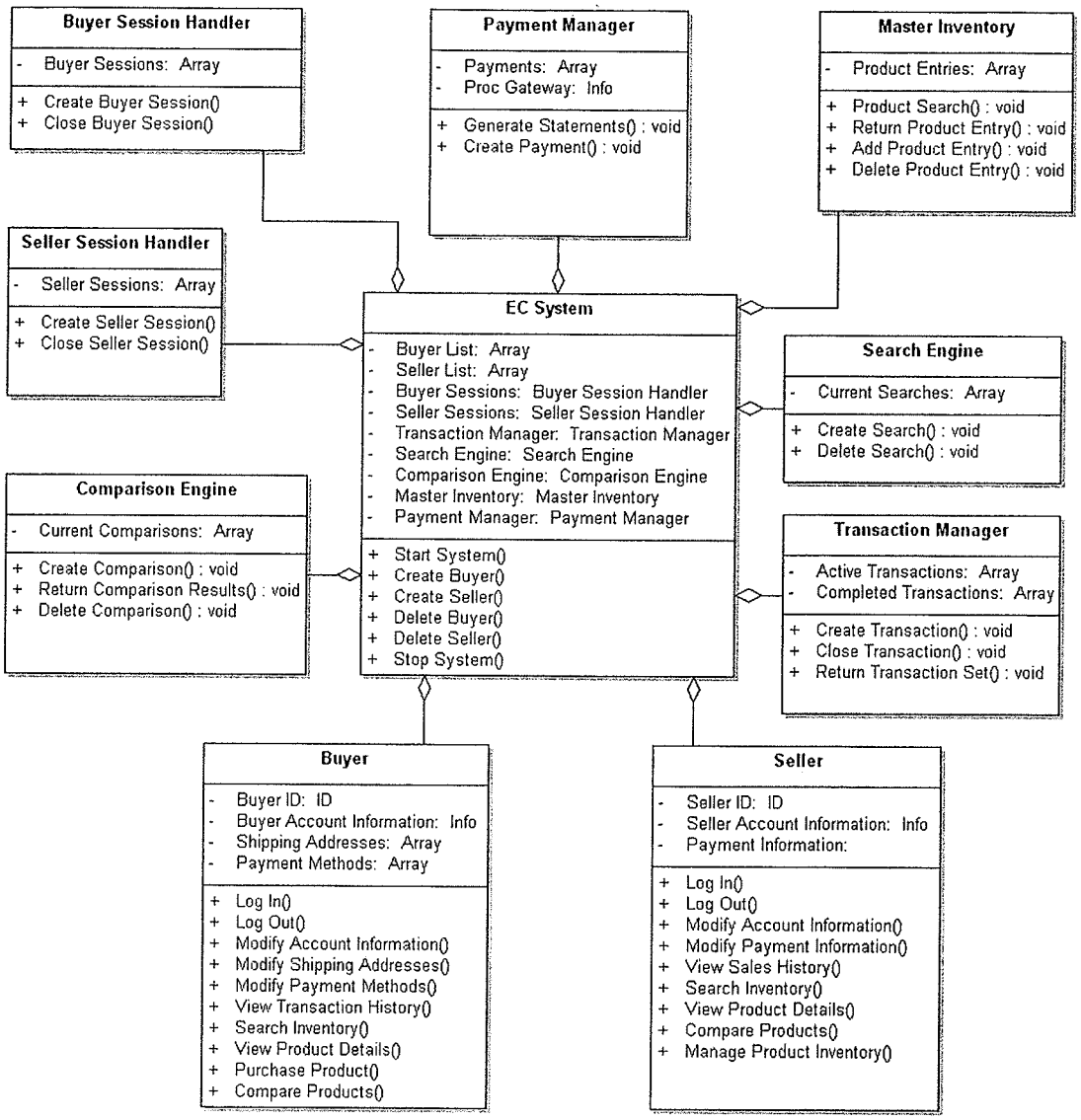

Figure 3.1: Electronic Commerce System Classes

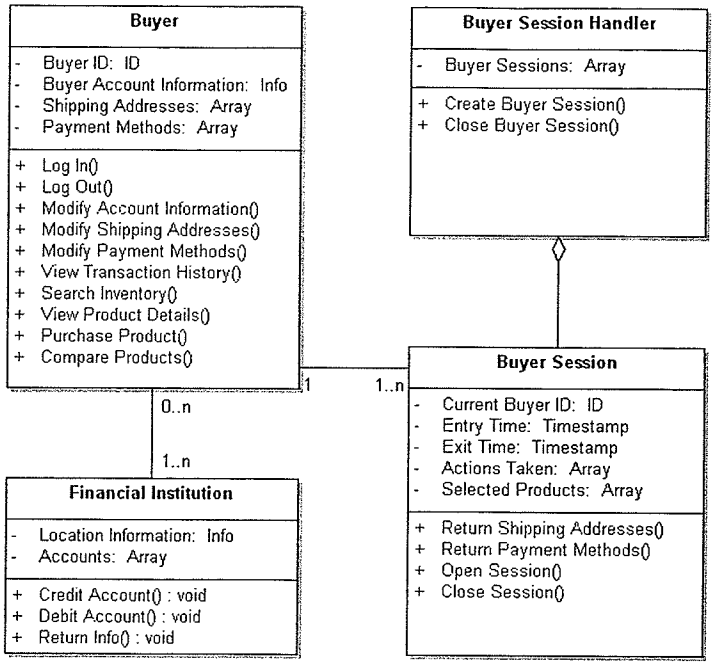

Figure 3.2: Buyer Classes

Session class contains information about the buyer's activities in the electronic commerce system. The times the buyer enters and exits the system are recorded as well as what actions the buyer undertakes while logged in and a list of any products he/she selected for purchase but are unpaid. The buyer's session is responsible for retrieving other buyer information stored in the system including shipping addresses and payment methods when required. Each buyer has one or more financial institutions that handles monetary transfers when purchasing products through the system. These financial institutions may represent a credit card company, bank, or other methods of payment and are represented by the Financial Institution class.

Seller Classes

Figure 3.3 shows the classes related to sellers in the electronic commerce system. The Seller class contains a unique ID for the seller, information about his/her location and contact details, and information about their account for payment of goods sold. The Seller class methods are similar to those of the Buyer class as the seller can log in and log out, modify

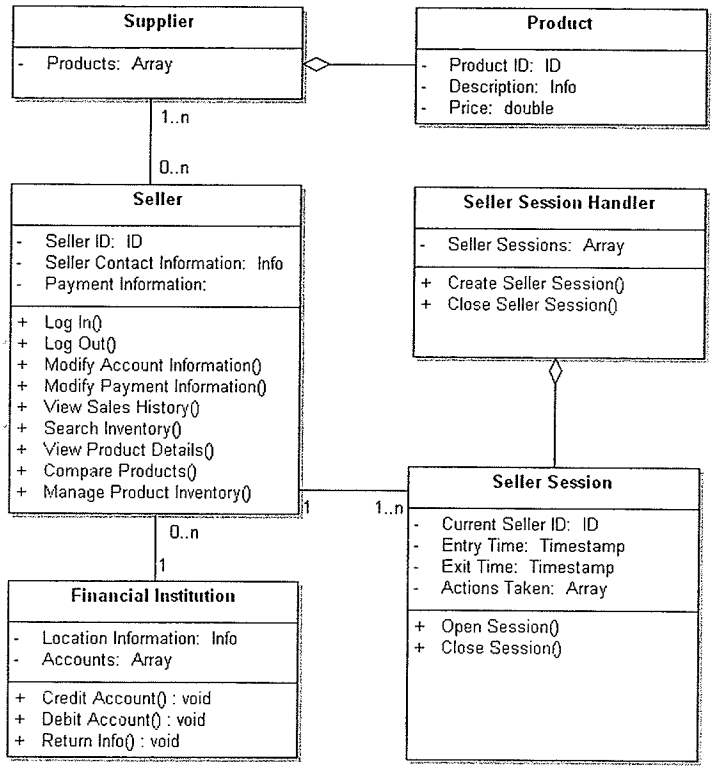

Figure 3.3: Seller Classes

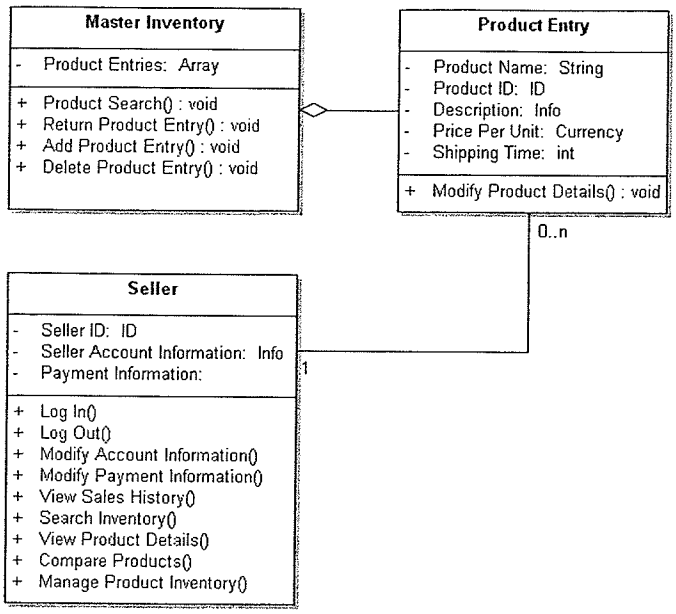

Figure 3.4: Inventory Classes

account and payment information, view sales history, search the inventory for products, compare and view the details of products, and manage product inventory. The Seller Session Handler class maintains a list of the current seller sessions in the system. It can open and close seller sessions. The Seller Session class contains information about the seller logged in, when the seller enters and exits the system and any actions the seller undertakes while logged into the system. Each seller has a financial institution, represented by the Financial Institution class, that handles its monetary transfers when buyers purchase products through the system. The financial institution may be a credit card company, bank, or other method of managing money. The products that a seller offers for sale in the electronic commerce system are provided by a supplier, shown as the Product and Supplier classes respectively. This supplier is a third party, which ensures that the seller has goods for sale to buyers and the supplier delivers the goods directly to the buyer upon receiving an order from the seller.

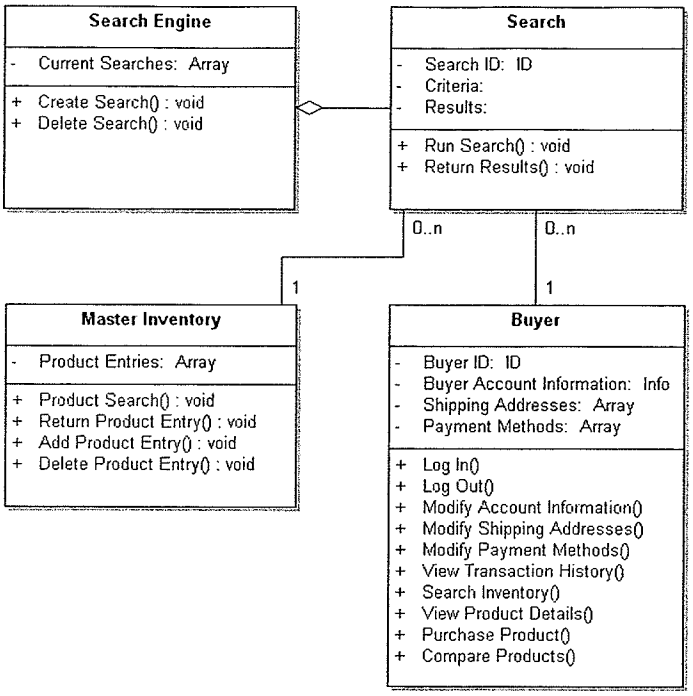

Figure 3.5: Search Classes

Inventory Classes

Figure 3.4 shows the classes responsible for products and their availability. The Master Inventory class contains a list of all the products in the system and it allows the addition and removal of products from the inventory. In addition, the master inventory returns search results and detailed information from individual product entries when searches are initiated. The Product Entry class stores information about a specific product. It contains a name and ID for each product as well as a description of the product's characteristics, price per unit, and the shipping time needed to send the product from seller to buyer. An operation of this class is to permit the modification of product properties. There is a relationship between product entries and sellers where each product entry belongs to a seller and a seller may have many different product entries.

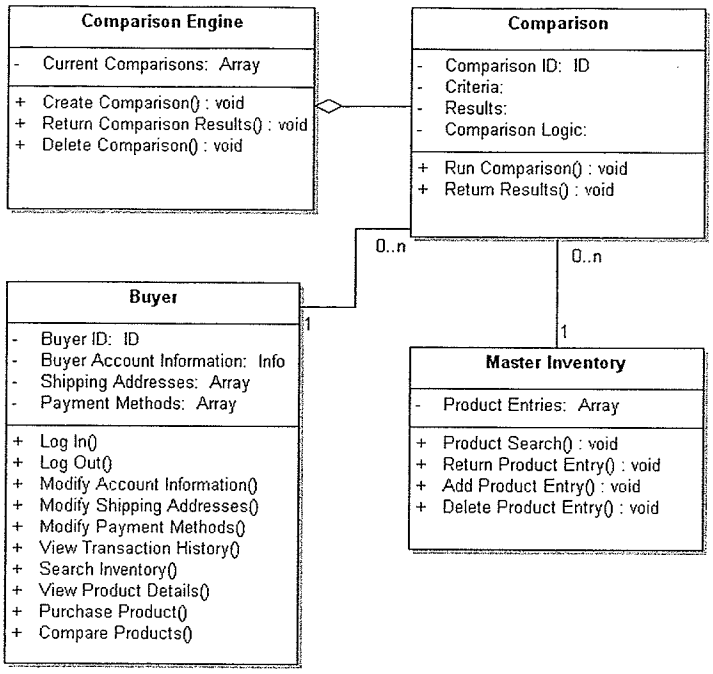

Figure 3.6: Comparaison Classes

Search Classes

Figure 3.5 shows the classes that are needed to manage and search for products in the electronic commerce system. The Search Engine class contains a list of all the current searches(represented by the Search class) in the system and allows the creation of new searches and the removal of completed searches. The Search class contains the specific details of an individual search. Each search has a unique identifier, some search criteria and results that match the criteria. The operations of the Search class include executing a search and returning search results to the buyer or seller that initiates the search. The Search class contacts the master inventory for product details to acquire the search results. The Buyer class is included here to show that each search belongs to a particular buyer and that any buyer can have concurrent searches in the system. It is important to note that a seller can also request a search operation.

Comparison Classes

Comparisons between products are handled by the classes shown in Figure 3.6. The Comparison Engine class manages the current comparisons in the system and creates new comparisons, returns results of completed comparisons and removes old comparisons. Each Comparison class represents a contrast of different product characteristics. Each comparison has a unique ID for use by the system, some criteria for determining the results and a logic component. The logical component of each Comparison class contains what operators are to be used to determine the results including minimums or maximums. The Comparison class applies those criteria and logic to the Master Inventory class to obtain results and then returns them to the comparison engine for release to a buyer or seller. Buyers may have multiple comparisons in the system, similar to searches, while any comparison belongs to only one buyer. As noted in the Search classes description, a seller may also compare products.

Transaction Classes

The Transaction class diagram shows the necessary classes for a buyer to purchase goods through the electronic commerce system. The Transaction Manager class contains attributes for all the active and completed transactions in the system and manages the creation and deletion of Transaction objects. The transaction manager is also responsible for returning completed transaction histories to buyers and sellers as requested. The Transaction class represents a purchase activity in the system. The Transaction class contains the ID of the buyer making the purchase, product and purchase information on each of the sellers involved, and an invoice. The Transaction class builds the list of sellers and products involved, sends orders for shipping to sellers, and creates an invoice for the transaction. It also invokes the payment manager to handle the electronic transfer of money. The Invoice class represents the information sent to the buyer upon successful completion of the transaction. It includes buyer details and a list of all purchases made. Each entry in this list includes seller details, the shipping address of each purchase, the billing address, product information, quantity, price, payment method, and the date of the purchase. Each

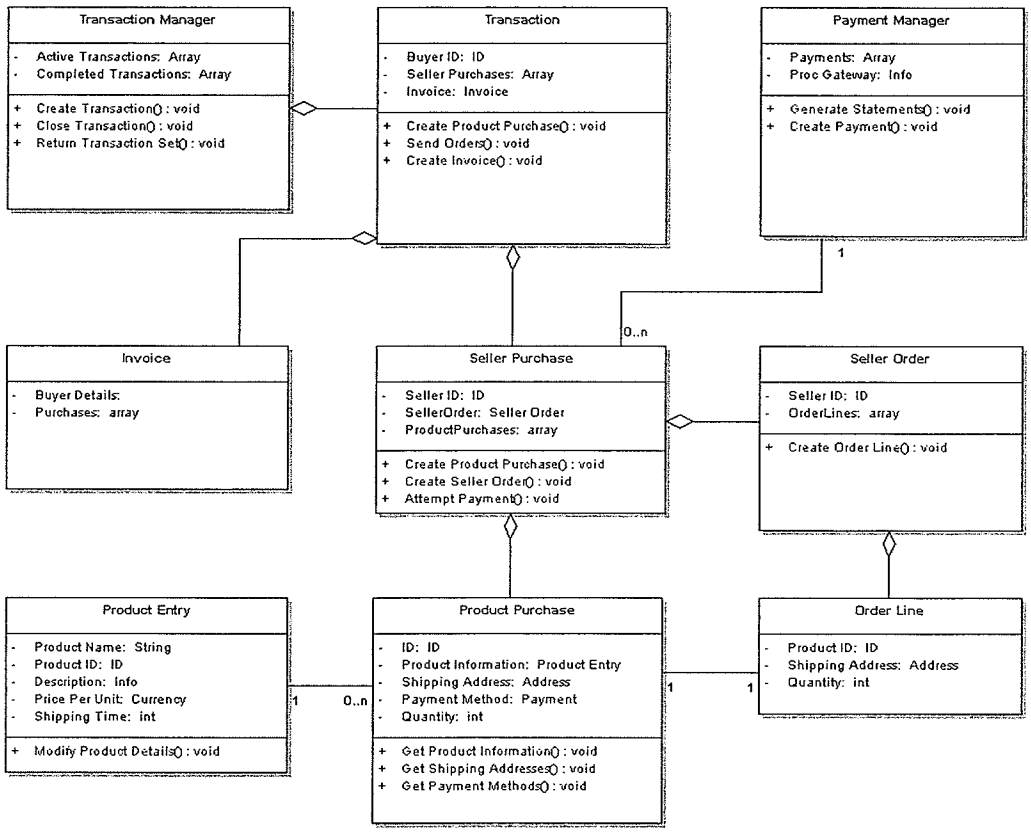

Figure 3.7: Transaction Classes

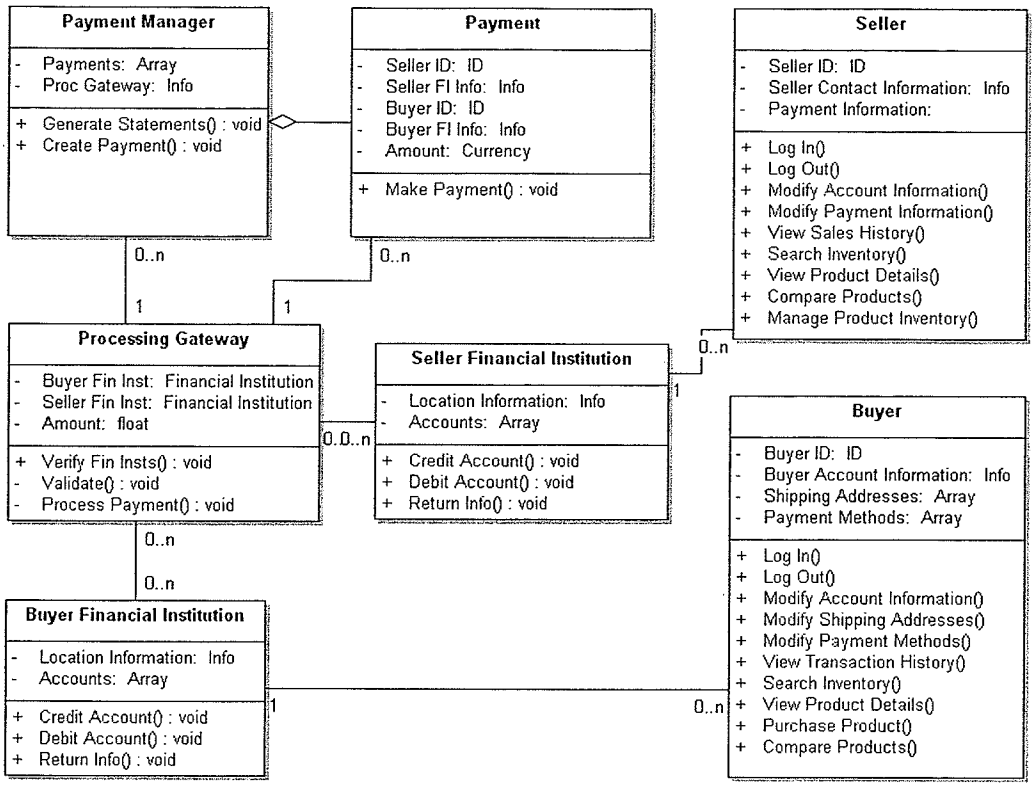

Figure 3.8: Payment Classes

Transaction class contains an instance of the Seller Purchase class for each seller involved in the transaction. This class contains the ID of the seller involved in the sale, a list of product purchases, and an order line for the seller. Each product purchase acquires and confirms product information, shipping addresses and payment methods from the buyer. The Product Entry class is shown in this diagram to demonstrate that each product purchase must have one product entry but that any particular product entry can appear in zero or many product purchases. Each Seller Order object consists of one or more Order Line objects and a seller id. Each Order Line object represents which product to ship, the shipping address for delivery, the quantity of items on order, and a date when the shipment must take place according to promises made by the seller in the product description.

Payment Classes

Figure 3.8 shows the classes involved in making payment as part of a purchase transaction. The main controlling object is the payment manager, which is responsible for generating a payment for each supplier involved in a transaction. It contains a list of all payments and location and usage information for a processing gateway to use for electronic fund transfers. The Payment class is responsible for communicating with the various parties involved in the purchase and getting the appropriate information to each. It contains information about both the buyer and seller(s), and their respective financial institutions and the amount(s) to be transferred. The processing gateway is a third party entity, usually outside the electronic commerce system, responsible for the actual verification, validation and processing between the buyer's financial institution and seller(s) financial institution(s).

3.2.2 Use Case Diagrams

Use case diagrams show the basic operations between actors in a system generally from the point of view of a user. A use case diagram presents an abstract view, at a high level, of interactions between major components of a system. In this thesis, the use cases of the buyer and seller in the electronic commerce system are presented. Any of the other previously defined classes could be represented in use cases but are omitted in this thesis.

Buyer Use Case

Figure 3.9 shows the actions that a buyer can take in the electronic commerce system. The actions include logging in and out of the system, viewing the buyer's transaction history, searching the entire inventory of products, viewing details about products, making comparisons between products, purchasing products from the system, and managing shipping addresses and payment methods.

Seller Use Case

Figure 3.10 shows all the operations that a seller can perform in the electronic commerce system. The operations include logging in and out of the system, viewing the seller's

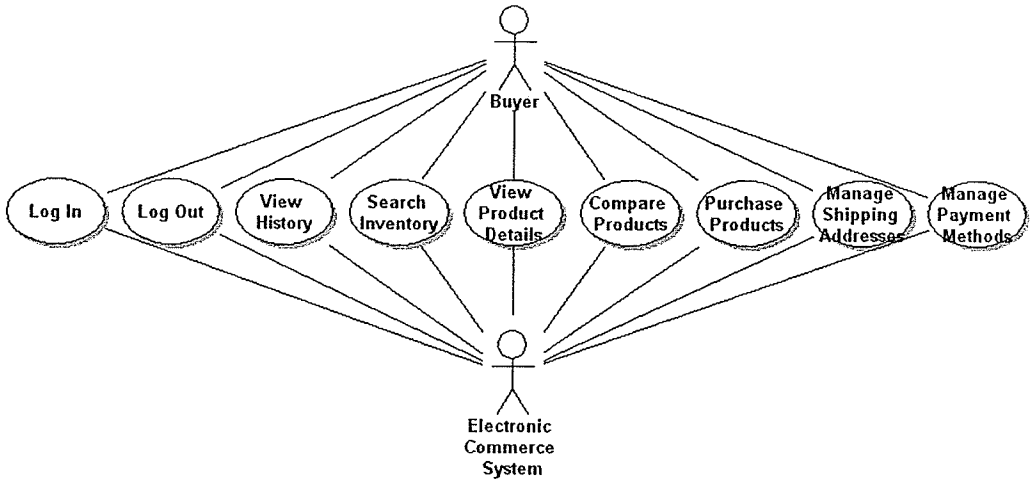

Figure 3.9: Buyer Use Case

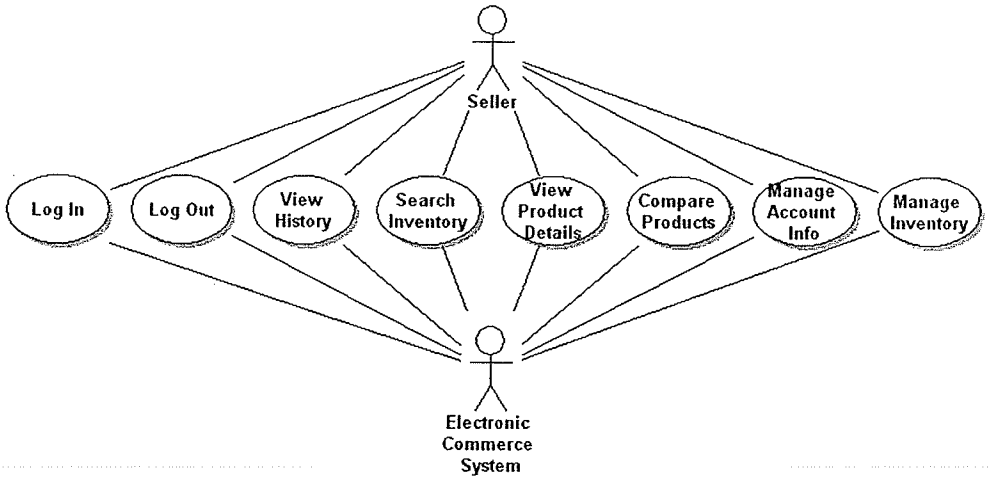

Figure 3.10: Seller Use Case

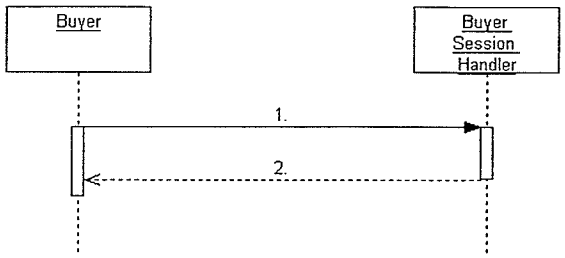

Figure 3.11: Buyer Login

sales history, searching the product inventory, viewing product details, making comparisons between products, managing financial account information and making changes to the inventory of products.

Although not all use cases have been presented for the electronic commerce system, the following sequence diagrams use the buyer and seller use cases to give further detail of the actors and steps involved. Each of the activities performed in the use cases are now presented as sequence diagrams.

3.2.3 Sequence Diagrams

Sequence diagrams show the interactions between different objects in a system. A sequence diagram shows the order of operations and process flow as well as gives more detailed operations for a corresponding use case.

Buyer Login

When a buyer wishes to enter the electronic commerce system, they must log in. The steps necessary for this activity are detailed below.

1. Request Login - The buyer enters the proper identification information via the buyer session handler and submits this information over the Internet.

2. Process Login - If the information entered is correct the buyer session handler opens a session for the buyer and returns welcome information to the buyer.

Figure 3.11 illustrates the buyer login activity.

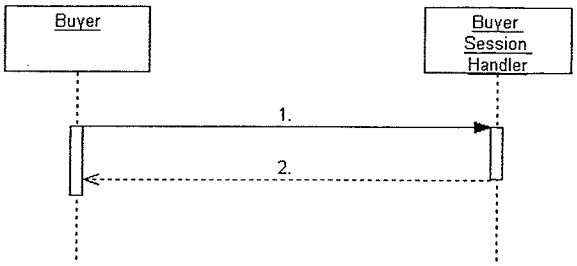

Figure 3.12: Buyer Logout

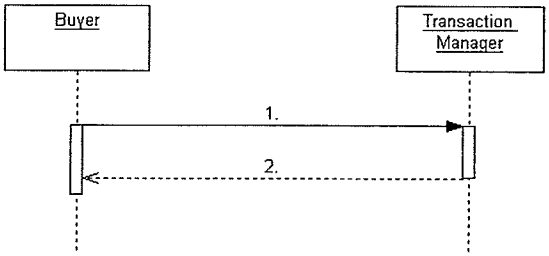

Figure 3.13: View Transaction History

Buyer Logout

Similar to the buyer login, when a buyer exits the electronic commerce system the buyer logout activity occurs.

1. Request Logout - The buyer wishes to leave the system and activates the logout option through the web interface.

2. Process Logout - The buyer session handler saves the buyer's current state in the system, updates any information that may have changed and displays a screen to the buyer indicating a successful logout.

Figure 3.12 shows the sequence diagram for the Buyer Logout activity.

View Transaction History

The activity in the buyer use case for investigating a buyer's transaction history is demonstrated next.

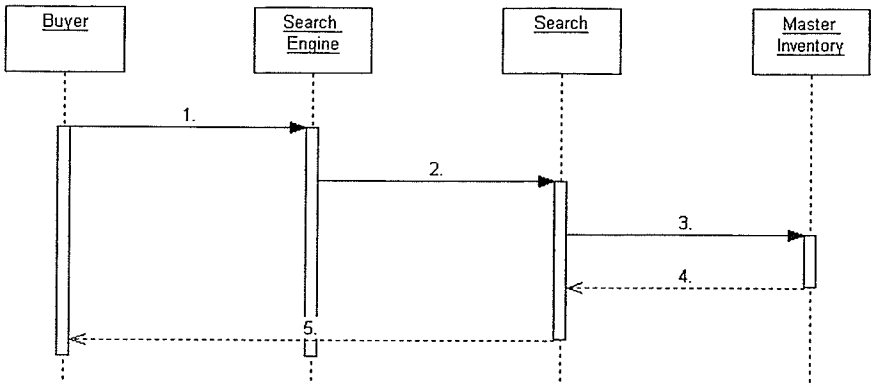

Figure 3.14: Search Inventory

1. Request Transaction History - The buyer requests, through a web page interface, to view some or all of the buyer's previous transactions. This interface allows the specification of several key criteria to return a subset of the buyer's transactions if desired.

2. Return Transactions - The transaction manager processes the request and returns the data requested.

Figure 3.13 shows the View Transaction History activity in as a sequence diagram.

Search Inventory

The buyer activity of searching the master inventory of products is next shown as a sequence diagram.

1. Request Search Inventory - The buyer requests information about certain products by specifying search criteria through a web interface presented by the search engine.

2. Create New Search - The search engine creates a new Search object to handle the details of the search request.

3. Query Master Inventory - The Search object takes the criteria specified by the buyer and presents it as a query to the master inventory for processing.

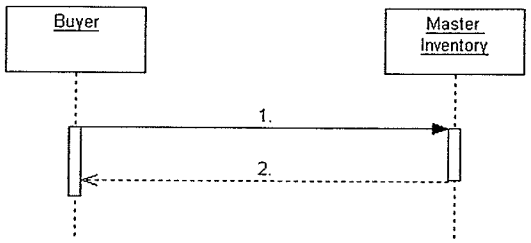

Figure 3.15: View Product Details

4. Return Products - The master inventory returns the product information for those products matching the buyer's search criteria.

5. Display Products - The Search object formats the product information returned from the buyer's query for the buyer to browse.

Figure 3.14 shows the sequence diagram of the search inventory activity.

View Product Details

A buyer may view the details of a particular product in the master inventory. That activity is described next.

1. Request Product Details - The buyer desires more information about a specific product and requests those product details from the master inventory.

2. Return Product Details - The master inventory returns all the details of the selected product (formatted appropriately) to the buyer.

Figure 3.15 shows the sequence diagram illustration for the view product details activity.

Compare Products

The activity of comparing multiple products in the master inventory is now shown.

1. Request Comparison - The buyer selects some products in the list from a previous product search and requests that a comparison be run on them by the comparison engine using a specified criteria.

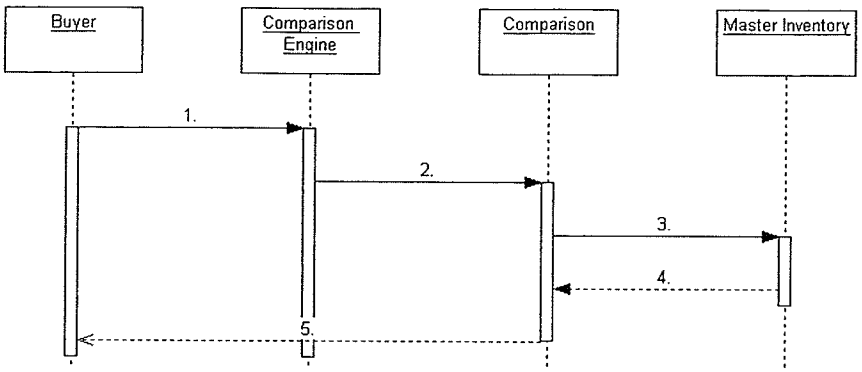

Figure 3.16: Compare Products

2. Create New Comparison - The comparison engine creates a new Comparison object to handle the buyer's request.

3. Request Product Details - For each product that is to be compared the Comparison object gets the product details from the master inventory.

4. Return Product Details - The master inventory returns all of the product details needed by the Comparison object so it can use the data to give the buyer an idea of the strengths and weaknesses of the goods involved in the comparison.

5. Return Comparison Results - Once the Comparison object receives the product details from the master inventory and completes its comparison of the products, it returns the comparison results to the buyer.

Figure 3.16 shows the sequence diagram for the Compare Products activity.

Purchase Product

The sequence of events that may occur in a Purchase Product use case activity may change, depending on errors or actions by other components of the system. Figures 3.17-3.19 show a successful purchase operation of one Product from one Seller. If a Buyer wishes to purchase multiple products from different Sellers, many of the steps below will be repeated as many times as needed. To show the core functionality of this operation only a best case scenario is

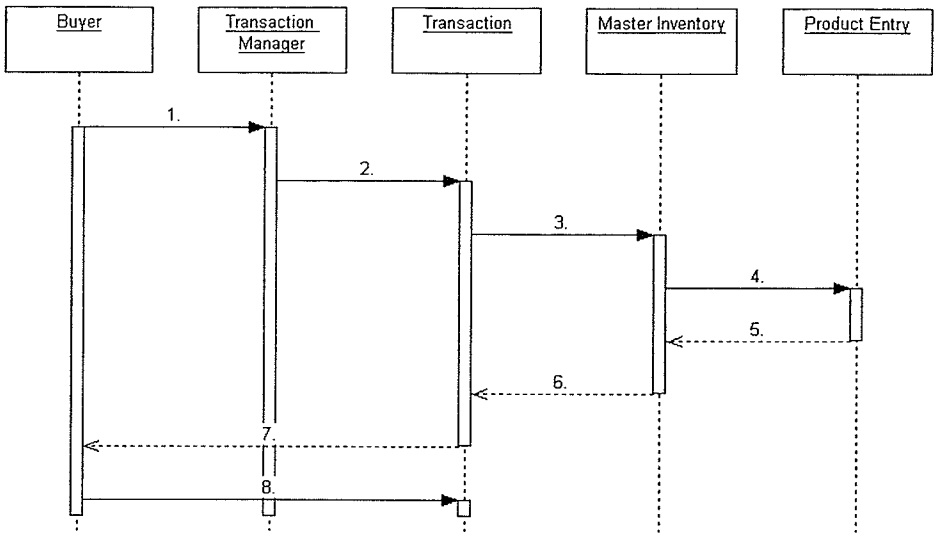

Figure 3.17: Purchase Product - Part I

considered for the sequence diagrams. The diagrams presented next detail the steps taken by the electronic commerce system when a product is purchased by a buyer.

1. Initiate Purchase - The buyer notifies the system via the transaction manager that he/she desires to purchase some product.

2. Create New Transaction - The transaction manager creates a new Transaction object and initiates the transaction to complete the work required for the purchase.

3. Request Product Details - The Transaction object requests the product entry information needed from the master inventory.

4. Get Product Entry - The master inventory requests some further information from the product entry involved in the transaction.

5. Return Product Entry - The product entry returns the information needed by the Transaction object to the master Inventory.

6. Return Product Information - The master inventory passes the relevant product details to the Transaction object for the purchase.

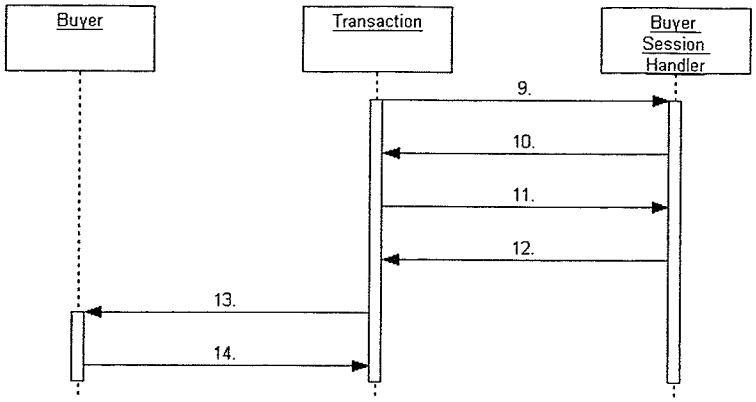

Figure 3.18: Purchase Product - Part II

7. Request Product Confirmation - The Transaction object sends the product information to the buyer in an understandable format for confirmation that this product is to be purchased.

8. Submit Product - The buyer confirms the product to purchase.

9. Get Shipping Addresses - The Transaction object requests the available shipping addresses of the buyer from the buyer session handler which stores this information.

10. Return Shipping Addresses - The buyer session handler returns to the Transaction object the default shipping addresses that the buyer has selected previously. The buyer has two other options available but not shown here. A buyer may create a new shipping address on the spot or select one of their existing addresses. For brevity, the diagram shows the selection of a default shipping address.

11. Get Payment Methods - The Transaction object requests the available payment methods for the buyer from the buyer session handler which stores this information.

12. Return Payment Methods - The buyer session handler returns to the Transaction object all the payment methods that the buyer has previously entered in the system.

13. Request Payment Method - The Transaction object takes the different payment methods from the buyer session handler, formats them and presents them to the buyer for selection.

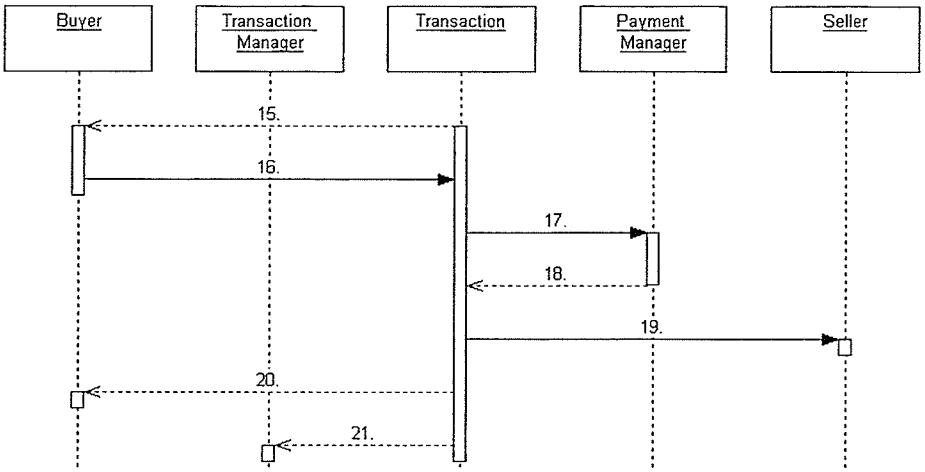

Figure 3.19: Purchase Product - Part III

14. Submit Payment Method - The buyer has the option of selecting a previously entered payment method or creates a new payment method to use. The former option is shown in this diagram but if a new payment method is created, the Transaction object will request the buyer session handler to add the new payment method.

15. Display Purchase Details - The Transaction object takes all of the information including the product to be purchased, shipping address selected, and payment method chosen, and displays it to the buyer for final confirmation.

16. Confirm Purchase - The buyer reviews the details of the purchase and submits authorization to finalize the transaction.

17. Request Payment - The Transaction object contacts the payment manager and initiates the electronic payment of the purchase. Details necessary for the payment to proceed are sent to the payment manager along with all of the other information needed.

18. Payment Results Returned - The payment manager, or specific Payment object, returns the results of the payment operations to the Transaction object. For the purpose of this example, we assume that the payment is successful.

19. Send Order - The Transaction object sends the completed order for the goods to the

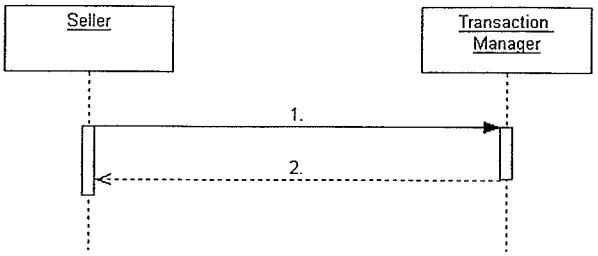

Figure 3.20: View Sales History

seller to fulfill. This order is passed to the supplier who provides the goods to the seller and then the product will be shipped.

20. Send Invoice and Notification - The Transaction object sends an invoice showing the details of the purchase to the buyer along with a notification that the purchase was successful. The buyer can then save or print the information shown for record-keeping.

21. Return Transaction Message - The Transaction object, having completed all of its required tasks, returns a message to the transaction manager that it has completed. The transaction manager can then archive the transaction and remove it from the list of active Transaction objects.

View Sales History

The activity to view the sales history from the seller use case diagram is shown next.

1. Request Sales History - The seller wishes to see a list of the sales made in the past set time period and submits this request to the transaction manager.

2. Return Transactions - The transaction manager returns all of the corresponding transactions to the seller in an understandable format.

Figure 3.20 shows the sequence diagram for the View Sales History use case.

Manage Products and Inventory

A seller has the ability to manage the products that it offers in the electronic commerce system. This steps in this activity are shown next.

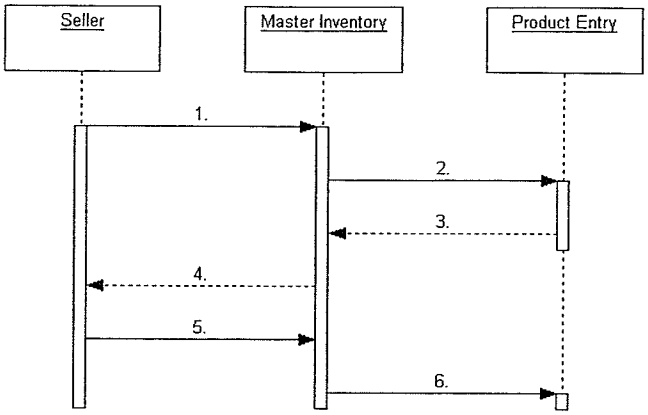

Figure 3.21: Manage Products and Inventory

1. Request Modify Inventory Item - The seller wishes to modify an inventory item in the system and sends a request to the master inventory.

2. Retrieve Inventory Entry - The Master Inventory requests the current details of the specific inventory item to be modified from the Product Entry.

3. Return Product Information - The product entry returns the product information to the master inventory.

4. Display Product Information - The master inventory displays the product information for editing to the seller.

5. Submit Product Modifications - The seller sends the modified product information to the master inventory for entry into the product entry.

6. Save Product Information - The master inventory, after checking the validity of the information received, sends the product information to the product entry to be saved.

Figure 3.21 shows a sequence diagram for Managing Products and Inventory.

3.2.4 State Diagrams

In UML, State diagrams (or Statecharts) show the dynamic or changing state of one class. Each state in a statechart as well as the reason for advancing to the next state are described.

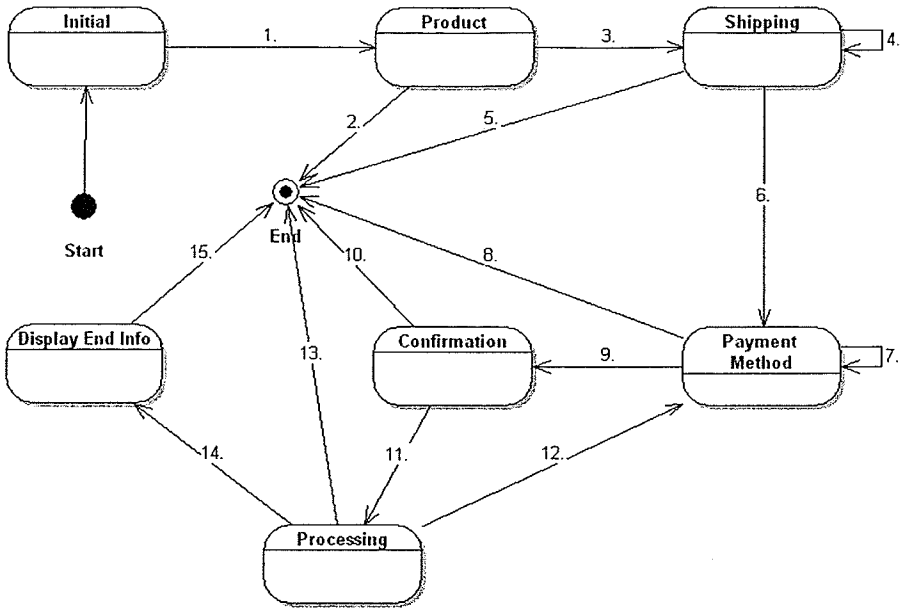

Figure 3.22: Transaction Class State Diagram

Figure 3.22 shows a statechart for the Transaction class.

Transaction State Diagram

States There are seven named states for a Transaction and these states represent the entire life cycle that a transaction passes through. Each state is described below.

- Initial - When a transaction is initialized it enters this state. It currently contains only information about the Buyer and Seller involved in the transaction.
- Product - Transition to this state occurs when the Transaction object receives the product information from the master inventory which is presented to the buyer for verification.
- Shipping - Transition to the 'Shipping' state occurs when the shipping addresses for the buyer, from the buyer session handler, are presented to the buyer for selection.
- Payment Method - The 'Payment Method' state occurs when the payment methods for the buyer, from the buyer session handler, are presented to the buyer for selection.

- Confirmation - This state represents the final decision point for the buyer to back out of the transaction.
- Processing - The "behind the scenes" actions take place in the 'Processing' state as the payment for the goods is resolved and reports are generated along with any possible error messages.
- Display End Information - Upon successful completion of the 'Processing' state the transaction enters the 'Display End Information' state. Here the purchase information is presented to the buyer and the transaction is complete.

Activities Fifteen different activities occur in a Transaction that cause a transition from one state to another. A brief explanation of each of these activities follows:

1. Display Product - The transaction shows the product information to the buyer from the master inventory and corresponding product entry. The state changes from 'Initial' to 'Product'.
2. Deny Product & Quit - The buyer decides not to purchase a particular product and quits the transaction. The state changes from 'Product' to 'End'.
3. Confirm Product - The buyer confirms the product as the one he/she wishes to purchase. The state changes from 'Product' to 'Shipping'.
4. Select Shipping Address - The buyer selects a shipping address to be used for the purchase. The state remains as 'Shipping'.
5. Deny Shipping Address & Quit - The buyer does not wish to select a shipping address and quits the transaction. The state changes from 'Shipping' to 'End'.
6. Confirm Shipping Address - The buyer confirms the shipping address for this purchase. The state changes from 'Shipping' to 'Payment Method'.
7. Select Payment Method - The buyer selects a payment method to be used for the purchase. The state remains as 'Payment Method'.

8. Deny Payment Method & Quit - The buyer does not wish to select a payment method and quits the transaction. The state changes from 'Payment Method' to 'End'.

9. Confirm Payment Method - The buyer confirms the payment method for this purchase. The state changes from 'Payment Method' to 'Confirmation'.

10. Deny Purchase Confirmation & Quit - The buyer does not wish to confirm all of the details of the transaction and quits. The state changes from 'Confirmation' to 'End'.

11. Confirm Purchase - The buyer confirms all of the purchase details including product information, shipping address, and payment method. The state changes from 'Confirmation' to 'Processing'.

12. Payment Method Failure - The payment method selected by the buyer causes an error. In recovery from the error, the transaction asks for a different payment method or requests the buyer to correct the error. The state changes from 'Processing' to 'Payment Method'.

13. System Error in Processing - There was an unrecoverable error during the processing of the payment and the transaction must terminate. The state changes from 'Processing' to 'End'.

14. Payment Succeeds - The payment operation completes successfully and the transaction details sent to create invoices for the buyer and orders for the sellers. The state changes from 'Processing' to 'Display End Info'.

15. Buyer Closes Transaction Screen - The buyer closes the transaction screen or goes to a different part of the electronic commerce system. The state changes from 'Display End Info' to 'End'.

3.2.5 Activity Diagram

Activity diagrams are similar to Statecharts. However, they demonstrate the conditional sequence of activities in a particular use case, detailing all possible situations. Activity diagrams are useful because they can show conditional logic and parallel processing.

Payment Activity Diagram

Figure 3.23 shows the steps in a payment operation. The payment operation begins with a Transaction object requesting a monetary transfer relating to a product purchase by a buyer. A new Payment object, created by the payment manager, submits the appropriate information to the processing gateway which undertakes some initial verification of identities. If this verification is unsuccessful, an error message is generated that indicates submission of incorrect information and returns control to the Transaction object where this information may be changed and resubmitted. If the verification is successful then some parallel processing occurs. Both the buyer's and the seller's financial institutions are contacted with authorization information. The financial institutions process the authorization information and return results as to whether the authorization is a success or failure. If either completes unsuccessfully, the Payment object creates another error message explaining that one of the financial institutions did not recognize the validity of the submission. Control is returned to the Transaction object so that another payment method may be chosen. If the authorization is successful then the buyer's and seller's accounts are settled by crediting and debiting monetary amounts. If this settlement is unsuccessful another exception is raised and the buyer and seller settlements that may have occurred if one succeeded while the other failed are undone the transaction is rolled back to its beginning. Another payment method can be chosen or the transaction can be reattempted. If the settlement is successful for both the buyer's and seller's financial institutions then the payment is a success and the payment can now end. In addition, the transaction can also terminate itself if some error occurs in its own processing environment occurred, rolling back all results.

3.2.6 Deployment Diagram for the Electronic Commerce System

The deployment diagram of the electronic commerce system (Figure 3.24) shows not only the breakdown of the hardware and processing but also the software distribution. In the diagram, boxes represent machines containing an operating element of the electronic commerce system. The circles in the deployment diagram represent an area for specialized communication between machines. Buyers and sellers access the system via an Internet-connected

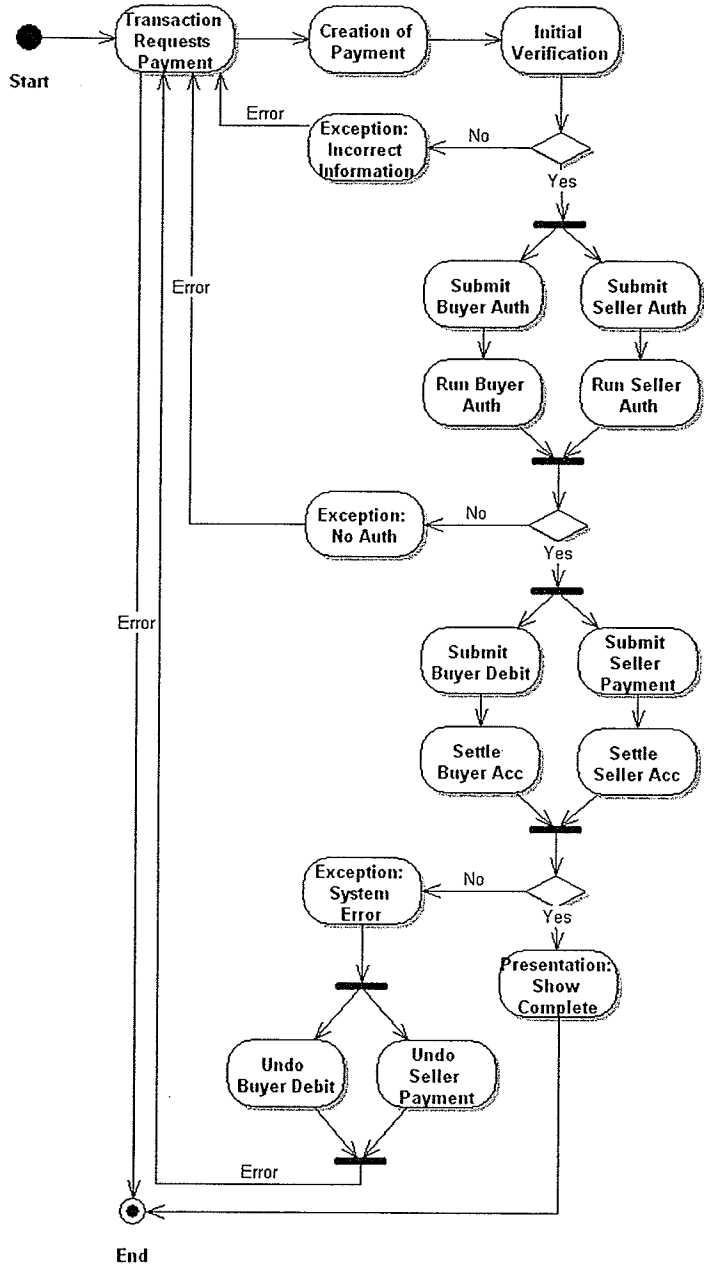

Figure 3.23: Payment Activity Diagram

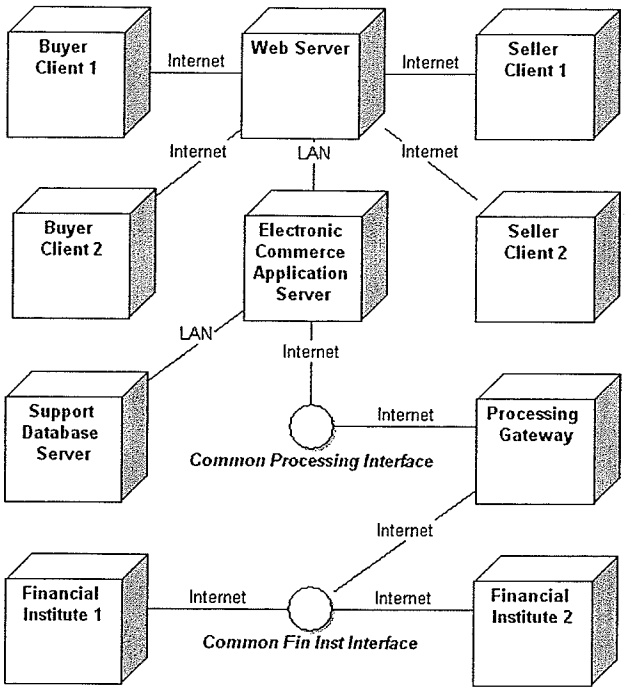

Figure 3.24: Electronic Commerce System Deployment Diagram

web server. The web server is responsible for serving all of the interface web pages used for all of the buyer's and seller's operations. The web server is connected to a dedicated electronic commerce system processing server that manages all of the actual processes carried out by the buyers and sellers. A database server also supports the electronic commerce application. These three components (the web server, electronic commerce application server and the support database server) are connected via a local area network. For the electronic processing of payments to take place during transactions, a processing gateway is necessary to handle the requests. This processing gateway manages the operations when transferring money from one financial institution to another. The processing gateway and the electronic commerce application server require a common interface to enable communication between them. This common processing interface provides a common set of functions to any system accessing the processing gateway. A similar interface, the common financial institute interface, supports communications between the processing gateway and financial institutions. The processing gateway, interfaces, and financial institutions are connected via the Internet.

3.3 Z Specification Language Model Description

A Z language specification of the electronic commerce system discussed in this thesis is now presented. This specification formally defines the requirements of the electronic commerce system model. Components of the Z specification are grouped by logical elements of the system. The section concludes with a demonstration of syntax, type and domain checking of the specification using the Z/EVES tool.

3.3.1 Z Language Specification

The Z specification language is based on mathematical concepts and it uses many symbols for its syntactic and semantic definitions. Because of these mathematical concepts and a move towards international standardization, Z specifications have a 'universal' syntax and semantics. To specify a system, this syntax can be combined with natural language, which is used to explain the mathematical symbols. By combining these two methods, a specification is understandable but still retains its mathematic foundation and remains rigorous and provable.

Common Type Definitions

The specification of the electronic commerce system requires the definition of some basic constants and types. These values are used throughout the system to describe and abstract important concepts and to allow for logical naming and use of different data types.

$[STRING, ADDRESS, ACTION, CODE]$

The above defines four basic types that are available throughout the specification. The *STRING* type represents a sequence of characters of some length and is used in a variety of ways including for names, descriptions and other character-based elements. The *ADDRESS* type logically encapsulates all of the information associated with the address of a location. This location may belong to a buyer or seller as a shipping or billing address. *ACTION* describes a particular step taken by a buyer or seller in the electronic commerce system. *ACTION* types are used by the buyer and seller session handlers to track what is done

during each session. Finally, *CODE* represents an authorization code used in payment transactions with financial institutions.

$STATUS ::= start \mid complete \mid error$
$PAYMENT\_STATUS ::= begin \mid auth \mid pay \mid end$

The *STATUS* type is used by components of the electronic commerce system that have recognizable states during processing. *STATUS* can have one of three possible values: *start*, *complete*, or *error*. *Start* signifies that the process undertaken has begun but has not completed and has not encountered an error. *Complete* simply means that the process has finished its task successfully while *error* signifies that the process has failed. Similar to *STATUS*, the *PAYMENT_STATUS* type represents the current state of a payment attempt. Each payment attempt may have the following possible values: *begin*, *auth*, *pay* or *end*. *Begin* signifies the start of the payment process and that no errors have occurred. *Auth* signifies that authorization for the payment has succeeded and *pay* represents a successful payment attempt. Finally, the *end* value signifies that an error occurred during the processing of a payment attempt.

$sys\_time : \mathbb{N}_1$

The *sys_time* variable is used throughout the system as a reference to the current time. It is used where processes are time sensitive or where time is important to a schema. This variable is stored as a natural number($\mathbb{N}_1$) greater than zero and is used in calculations with other time values in the system. This choice allows for time values to be added and subtracted from each other without violating the rules in the mathematical Z domain while still representing the concept of time.

$PAYMENT\_TYPE ::= credit\_card \mid debit \mid internet$

The *PAYMENT_TYPE* type is used in the electronic commerce system to classify payment types for a buyer. Each of these payment types represents a real world payment method. *Credit_card* identifies a credit card such as MasterCard or American Express, *debit* represents transfers directly from a financial institution and *internet* identifies such payment methods as PayPal.

Common Schemas and Definitions

The basic definitions necessary for defining more complex entities later in the specification are given below. Some of these items are used throughout the system while others may only be used to describe a unique and specific component of the system.

$$
\begin{array}{|l}
\textit{ProdChar} \\
\hline
id, pid : \mathbb{N}_1 \\
name, value : STRING \\
\hline
\end{array}
$$

The *ProdChar* schema describes a characteristic of a product supplied by the electronic commerce system. These product characteristics(*ProdChars*) describe each product in detail and are used in making comparisons between products using the comparison engine. Each product characteristic has an id, the id of the product entry it relates to(*pid*), a characteristic name, and a corresponding value. Below are two functions used to create and return product characteristics.

$$
\begin{array}{|l}
create\_prod\_char : \mathbb{N}_1 \times \mathbb{N}_1 \times STRING \times STRING \rightarrow ProdChar \\
\hline
\forall\, id, pid : \mathbb{N}_1;\ n, v : STRING;\ pc : ProdChar \bullet \\
\quad create\_prod\_char(id, pid, n, v) = pc \Rightarrow \\
\quad\quad pc.id = id \land pc.pid = pid \land pc.name = n \land pc.value = v
\end{array}
$$

The *create_prod_char* function creates a new product characteristic for a product entry. The function assigns input values to the attributes of a new product characteristic. Another function is presented below to return a specified product characteristic from a set of product characteristics.

$$
\begin{array}{|l}
return\_prod\_char : \mathbb{N}_1 \times \mathbb{P}\, ProdChar \rightarrow \mathbb{P}\, ProdChar \\
\hline
\forall\, i : \mathbb{N}_1;\ p, pc : \mathbb{P}\, ProdChar \bullet return\_prod\_char(i, p) = pc \Rightarrow \\
\quad pc \subseteq p \land (\forall\, x : ProdChar \mid x \in pc \bullet x.id = i)
\end{array}
$$

The *return_prod_char* function returns the requested characteristic of a given product from a set of product characteristic schemas. The function returns the correct product characteristic using the identifier specified.

The schemas and functions used to describe the representation of a product are presented next.

ProductEntry

$$id, seller\_id, price, ship\_time : \mathbb{N}_1$$
$$name, keywords : STRING$$
$$chars : \mathbb{P}\, ProdChar$$

$$\forall x, y : ProdChar \bullet x \in chars \land y \in chars \Rightarrow x.id \neq y.id$$

The *ProductEntry* schema represents a product class in the system. Each seller has a number of these product entries that they offer for sale to buyers. Each product entry has an id, the id of the seller, a name, a description of the product keywords used in searching, a selling price, the amount of time the product is expected to take to be shipped when ordered, and a set of characteristics describing the product. The price definition used in this schema and throughout the specification is represented as a natural number with no decimal places. This is done for the sake of simplicity and in an actual implementation of an electronic commerce system the prices would include two decimal places to match the conventional standards for monetary values. A pre-condition on this schema is that each product characteristic of a product entry be unique. A schema describing the initialization of a product entry is presented below.

InitProductEntry

$$\Delta ProductEntry$$

$$chars' = \emptyset$$

When a product entry is created, its product characteristics are empty. The definition below is useful to create a product entry.

$$create\_product\_entry : \mathbb{N}_1 \times \mathbb{N}_1 \times STRING \times STRING \times \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow ProductEntry$$

$$\forall id, sid, p, st : \mathbb{N}_1;\ n, k : STRING;\ pe : ProductEntry \bullet$$
$$create\_product\_entry(id, sid, n, k, p, st) = pe \Rightarrow$$
$$pe.id = id \land pe.seller\_id = sid \land pe.name = n \land pe.keywords = k \land$$
$$pe.price = p \land pe.ship\_time = st \land pe.chars = \emptyset$$

The *create_product_entry* function takes a set of entry parameters and returns a product entry schema. The *create_product_entry* function is used in the remainder of the specification, especially when lists of products are being generated. Once a product entry exists in the master inventory, a method to retrieve the information stored for that product is necessary.

The function below fulfills that purpose.

$$return\_product\_entry : \mathbb{N}_1 \times \mathbb{P}\, ProductEntry \rightarrow \mathbb{P}\, ProductEntry$$

$$\forall i : \mathbb{N}_1;\ p, pe : \mathbb{P}\, ProductEntry \bullet return\_product\_entry(i, p) = pe \Rightarrow pe \subseteq p \wedge (\forall x : ProductEntry \mid x \in pe \bullet x.id = i)$$

The *return_product_entry* function is used in the remainder of the specification to return a given product entry from a set of product entry schemas. The function returns the correct product entry based on the identifier specified. Additional operations on a product entry include the addition and removal of product characteristics from a product entry. The operational schemas below describe the addition and removal operations.

AddProdChar

$$\Delta ProductEntry$$
$$i?, pid? : \mathbb{N}_1$$
$$n?, v? : STRING$$

$$create\_prod\_char(i?, pid?, n?, v?) \notin chars$$
$$chars' = chars \cup \{create\_prod\_char(i?, pid?, n?, v?)\}$$

The *AddProdChar* schema takes a new id, product entry id, name, and value and creates a product characteristic in a product entry. The characteristic created must not have the same id as another characteristic for that product. To remove characteristics from a product entry, a removal operation must also exist. The *RemoveProdChar* operation provides this function.

RemoveProdChar

$$\Delta ProductEntry$$
$$id? : \mathbb{N}_1$$

$$chars' = chars \setminus return\_prod\_char(id?, chars)$$

The *RemoveProdChar* schema uses the *return_prod_char* function to retrieve a specified characteristic and then uses set difference to remove it from the set of characteristics in a product entry.

The next abstract concept is *Action*. Schemas and definitions follow that describe an *Action* and the behaviours that are associated with it.

Action

$id, session\_id : \mathbb{N}_1$
$action\_type : ACTION$
$details : STRING$

The *Action* schema is used to document events (i.e. actions) that occur in the system. These events are usually driven by the buyer or seller and they are tracked to provide knowledge of the activities of the users of the system. Each action has a unique id, a session id that it is associated with, the type of the action, and some details of what occurred. To create an *Action* all of the component parts required are assembled by the definition below.

$$create\_action : \mathbb{N}_1 \times \mathbb{N}_1 \times ACTION \times STRING \rightarrow Action$$

$$\forall id, sid : \mathbb{N}_1;\ at : ACTION;\ d : STRING;\ a : Action \bullet$$
$$create\_action(id, sid, at, d) = a \Rightarrow a.id = id \land a.session\_id = sid \land$$
$$a.action\_type = at \land a.details = d$$

The *create_action* function takes the elements of an action and creates a new action to be used for both buyers and sellers.

The next schemas and definitions describe financial institutions and their associated subcomponents. These components provide functionality for the necessary financial operations in the electronic commerce system.

Account

$acc\_num : \mathbb{N}_1$
$balance : \mathbb{N}$

A financial institution uses *Account* to track money for an individual buyer or seller. Each account has an account number and a balance. In this electronic commerce system the balance must be greater than zero. To create an account, the following definition is applicable.

$$create\_account : \mathbb{N}_1 \times \mathbb{N} \rightarrow Account$$

$$\forall an : \mathbb{N}_1;\ b : \mathbb{N};\ a : Account \bullet create\_account(an, b) = a \Rightarrow$$
$$a.acc\_num = an \land a.balance = b$$

The *create_account* function takes a new account number and starting balance and creates a new account.

A *FinancialInstitution* represents a business that holds monetary accounts for customers. In the electronic commerce system, these customers are buyers and sellers. A financial institution can debit and credit accounts for a customer. Each financial institution has a unique name, an authorization code, and a number of accounts. Each account number is unique. The authorization code is used during the payment process to ensure the proper access is granted for a monetary transfer.

FinancialInstitution

$$name : STRING$$
$$auth\_code : CODE$$
$$accounts : \mathbb{P}\, Account$$

$$\forall x, y : Account \bullet x \in accounts \land y \in accounts \Rightarrow x.acc\_num \neq y.acc\_num$$

A general constraint over the entire electronic commerce system holds that no two financial institutions may be the same. This constraint is described by the following definition.

$$fin\_insts : \mathbb{P}\, FinancialInstitution$$

$$\forall x, y : FinancialInstitution \bullet x \in fin\_insts \land y \in fin\_insts \Rightarrow x.name \neq y.name$$

The schema below represents the initial state of a *FinancialInstitution.*

InitFinancialInstitution

$$\Delta FinancialInstitution$$

$$accounts' = \emptyset$$

When a financial institution is initialized, it has no accounts. As the electronic commerce system evolves, a financial institution can add or remove accounts as customers come and go.

AddAccount

$$\Delta FinancialInstitution$$
$$a?, b? : \mathbb{N}_1$$

$$create\_account(a?, b?) \notin accounts$$
$$accounts' = accounts \cup \{create\_account(a?, b?)\}$$

The *AddAccount* schema takes a new account number and balance and creates an account in a financial institution. The new account must not have the same account number

as a previous account. It should be possible to remove customer accounts from a financial institution. To remove an account, a method must exist to find the account.

$$return\_account : \mathbb{N}_1 \times \mathbb{P}\, Account \rightarrow \mathbb{P}\, Account$$

$$\forall\, an : \mathbb{N}_1;\ as, a : \mathbb{P}\, Account \bullet$$
$$return\_account(an, as) = a \Rightarrow a \subseteq as \land$$
$$(\forall\, x : Account \mid x \in a \bullet x.acc\_num = an)$$

The *return_account* function takes an account number and a set of accounts and returns the account that matches the id in the set of accounts. This function is used to identify information about a specific account. The *RemoveAccount* operation removes an account from a financial institution.

RemoveAccount

$$\Delta FinancialInstitution$$
$$a\_n? : \mathbb{N}_1$$

$$accounts' = accounts \setminus return\_account(a\_n?, accounts)$$

The *RemoveAccount* schema uses the *return_account* function to retrieve a specified account and then uses the set difference method to remove it from the set of accounts in the financial institution.

To use the accounts to handle funds, they must have the ability to increase and decrease their balances.

$$credit\_account : Account \times \mathbb{N}_1 \rightarrow \mathbb{N}_1$$

$$\forall\, a : Account;\ amount, new\_bal : \mathbb{N}_1 \bullet credit\_account(a, amount) = new\_bal \Rightarrow$$
$$new\_bal = a.balance + amount$$

The *credit_account* function demonstrates how to credit an account in a financial institution. An account and amount to be deposited are given as inputs and the amount is added to the account balance. When this function is used by a financial institution the new value of the account balance is assigned to the results.

$$debit\_account : Account \times \mathbb{N}_1 \rightarrow \mathbb{N}$$

$$\forall\, a : Account;\ amount : \mathbb{N}_1;\ new\_bal : \mathbb{N} \bullet debit\_account(a, amount) = new\_bal \Rightarrow$$
$$new\_bal = a.balance - amount$$

The *debit_account* function demonstrates how to debit an account. An account and amount to be withdrawn are given as inputs and the amount is subtracted from the account balance. When this function is used by a financial institution the new value of the account balance is assigned to the results.

Buyers and sellers need to identify a suitable payment method. Each payment method encapsulates the pertinent information needed to transfer funds. A payment method schema follows.

PaymentMethod

$$
\begin{array}{l}
id, acc\_num : \mathbb{N}_1 \\
name, finame : STRING \\
auth\_code : CODE \\
payment\_type : PAYMENT\_TYPE
\end{array}
$$

The *PaymentMethod* schema represents the information needed for the payment manager of the electronic commerce system to initiate a payment. Each payment method has an id, name, financial institution name, authorization code, and payment type.

$$
\begin{array}{l}
create\_payment\_method : \mathbb{N}_1 \times \mathbb{N}_1 \times STRING \times STRING \times \\
\qquad\qquad CODE \times PAYMENT\_TYPE \rightarrow PaymentMethod \\
\hline
\forall i, a\_n : \mathbb{N}_1;\ n, fin : STRING;\ a\_c : CODE; \\
\quad p\_t : PAYMENT\_TYPE;\ p : PaymentMethod \bullet \\
\qquad create\_payment\_method(i, a\_n, n, fin, a\_c, p\_t) = p \Rightarrow \\
\qquad p.id = i \land p.acc\_num = a\_n \land p.name = n \land p.finame = fin \land \\
\qquad p.auth\_code = a\_c \land p.payment\_type = p\_t
\end{array}
$$

The *create_payment_method* function creates a payment method given the proper components. This definition assigns the input values given to the attributes of the new payment method. In addition to creating a payment method, a function is needed that returns an identified payment method from a collection. This function is necessary to return information about a payment method for use in transactions and for the removal of payment methods.

$$
\begin{array}{l}
return\_payment\_method : \mathbb{N}_1 \times \mathbb{P}\, PaymentMethod \rightarrow \mathbb{P}\, PaymentMethod \\
\hline
\forall i : \mathbb{N}_1;\ pm, p : \mathbb{P}\, PaymentMethod \bullet \\
\quad return\_payment\_method(i, pm) = p \Rightarrow p \subseteq pm \land \\
\qquad (\forall x : PaymentMethod \mid x \in p \bullet x.id = i)
\end{array}
$$

The *return_payment_method* function returns a set of payment methods specified by a unique id from the set of all possible payment methods.

In the electronic commerce system, the actions that a buyer or seller have taken in the system are important. To capture this information, session information is vital.

Session

$$id, owner, entry, exit : \mathbb{N}_1$$
$$actions : \text{seq}\, Action$$
$$entry \leq exit$$

The *Session* schema represents a user's activity from login to logout in the electronic commerce system. Each time a buyer or seller enters the system, a session is created. Each session has an id, the id of the buyer or seller called the *owner*, an entry and exit timestamp, and a sequence of actions of the user. For every session, the entry time must be before the exit time. A *Session* must be initialized when a buyer or seller enters the system and the schema below describes that operation.

InitSession

$$\Delta Session$$
$$actions = \langle\rangle$$

The *InitSession* schema sets the initial sequence of actions in a session to be empty. Once a *Session* has been initialized, it must be updated with current user information including owner id and entry and exit times. The operational schema below describes that action.

ModifySession

$$\Delta Session$$
$$id?, ow?, en?, ex? : \mathbb{N}_1$$
$$id? = id$$
$$owner' = ow?$$
$$entry' = en?$$
$$exit' = ex?$$

The *ModifySession* schema takes new values for a session as inputs and applies them to the individual values of a *Session*. Each *Session* records the actions taken by a buyer or

seller. The operation *AddSessionAction* adds a new action to a session.

AddSessionAction

$$\Delta Session$$
$$id? : \mathbb{N}_1$$
$$at? : ACTION$$
$$d? : STRING$$

$$actions' = actions \frown \langle create\_action(id?, id, at?, d?) \rangle$$

The *AddSessionAction* schema takes an id, an action type, and a description and creates a new action using the *create_action* definition. The new action is appended to the end of the sequence of actions for the session through the use of the concatenation operator.

$$create\_session : \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow Session$$

$$\forall id, ow, en, ex : \mathbb{N}_1;\ s : Session \bullet create\_session(id, ow, en, ex) = s \Rightarrow$$
$$s.id = id \land s.owner = ow \land s.entry = en \land s.exit = ex$$

The *create_session* function takes a unique id, the owner, and entry and exit timestamps and returns a session schema with those input values applied as initial values. The *return_session* function provides the ability to find an existing session in the list of current sessions.

$$return\_session : \mathbb{N}_1 \times \mathbb{P}\, Session \rightarrow \mathbb{P}\, Session$$

$$\forall i : \mathbb{N}_1;\ ss, s : \mathbb{P}\, Session \bullet return\_session(i, ss) = s \Rightarrow$$
$$s \subseteq ss \land (\forall x : Session \mid x \in s \bullet x.id = i)$$

The *return_session* function takes the id of a session and a set of active sessions and returns a set of sessions containing a single item with an id matching the input value.

Schemas and Definitions related to Buyers

The buyer is now described using schemas and functions. These definitions abstract a buyer and its possible operations.

Buyer

$$
\begin{array}{l}
id : \mathbb{N}_1 \\
name, phone, email : STRING \\
address : ADDRESS \\
shipping\_addresses : \mathbb{P}\, ADDRESS \\
payment\_methods : \mathbb{P}\, PaymentMethod \\
\hline
\forall x, y : ADDRESS \bullet x \in shipping\_addresses \land \\
\quad y \in shipping\_addresses \Rightarrow x \neq y \\
\forall i, j : PaymentMethod \bullet i \in payment\_methods \land \\
\quad j \in payment\_methods \Rightarrow i.id \neq j.id
\end{array}
$$

The *Buyer* schema contains the attributes of a buyer in the electronic commerce system. Each buyer has a id, name, address, phone number and email address. The address stored by the buyer outside of the shipping addresses represents the buyer's home or invoicing address. This address may have a matching shipping address but is used for all other mailings. In addition to these attributes, each buyer also has a set of shipping addresses and a set of payment methods. The shipping addresses are used to keep track of where goods purchased by the buyer may be sent. The payment methods are set up by the buyer so that the buyer may purchase goods using more than one type of payment. Every item in each set is unique. This constraint eliminates duplicate shipping addresses and payment methods. The *InitBuyer* schema defines the initialization of a buyer.

InitBuyer

$$
\begin{array}{l}
\Delta Buyer \\
\hline
shipping\_addresses' = \emptyset \\
payment\_methods' = \emptyset
\end{array}
$$

The *InitBuyer* schema initializes the buyer so that the list of initial shipping addresses and payment methods are set to be the empty set.

Once a buyer has been initialized, a mechanism is required to modify the information about a buyer. The operational schema *ModifyBuyer*, defined below, captures this requirement.

ModifyBuyer

$\Delta Buyer$
$id? : \mathbb{N}_1$
$n?, p?, e? : STRING$
$a? : ADDRESS$

$id' = id?$
$name' = n?$
$address' = a?$
$email' = e?$
$phone' = p?$

The *ModifyBuyer* schema provides the mechanism to change the values of a buyer. These changes may be requested by the buyer due to a change of address, name, or different phone number or email address. The schema takes an id, name, address, phone number and email address and sets the corresponding buyer values. The buyer may need to add and remove shipping addresses and payment methods as required. These operations are defined below.

AddShippingAddress

$\Delta Buyer$
$a? : ADDRESS$

$a? \notin shipping\_addresses$
$shipping\_addresses' = shipping\_addresses \cup \{a?\}$

The *AddShippingAddress* schema takes a new address and appends it to the existing list of shipping addresses for the buyer. A precondition for the execution of the operation is that the new address must not already be in the list of existing shipping addresses. The operation to remove a shipping address follows.

DeleteShippingAddress

$\Delta Buyer$
$a? : ADDRESS$

$a? \in shipping\_addresses$
$shipping\_addresses' = shipping\_addresses \setminus \{a?\}$

The *DeleteShippingAddress* schema takes an address and removes it from the list of existing shipping addresses. For this operation to execute, the shipping address to be deleted must be in the list of the buyer's shipping addresses. Like the shipping addresses, payment

methods can be added and deleted from a buyer. The operational schemas representing these functions are given next.

AddBuyerPaymentMethod

$\Delta Buyer$
$n?, fin? : STRING$
$a\_c? : CODE$
$i?, a\_n? : \mathbb{N}_1$
$p\_t? : PAYMENT\_TYPE$

$create\_payment\_method(i?, a\_n?, n?, fin?, a\_c?, p\_t?) \notin payment\_methods$
$payment\_methods' = payment\_methods \cup$
$\quad \{create\_payment\_method(i?, a\_n?, n?, fin?, a\_c?, p\_t?)\}$

The *AddBuyerPaymentMethod* schema adds a new payment method to the existing payment methods for the buyer by using the *create_payment_method* function. This schema also states that the payment method to be created must not be in the existing payment methods. Just as a payment method can be added to a buyer, the functionality must also exist to remove a payment method.

RemoveBuyerPaymentMethod

$\Delta Buyer$
$id? : \mathbb{N}_1$

$return\_payment\_method(id?, payment\_methods) \subseteq payment\_methods$
$payment\_methods' = payment\_methods \setminus$
$\quad return\_payment\_method(id?, payment\_methods)$

The *RemoveBuyerPaymentMethod* schema takes the id of the payment method to be removed from the set of existing payment methods. Using the *return_payment_method* function, the schema returns a set containing the payment method identified by the id supplied. That payment method is then removed from the set of payment methods using the set difference operator. The initial constraint on the schema states that the payment method to be removed must be in the set of *payment_methods* for the buyer. Since the return_payement_method function returns a single element set, the subset equality operator($\subseteq$) is used instead of the element operator($\in$).

The session information stored for a buyer differs slightly from the initial definition presented for a *Session*. The system keeps track of the items that a buyer wishes to purchase

but has not paid for yet. This electronic shopping cart is used across buyer logins and is maintained by a session when a buyer is active in the system.

BuyerSession

$$Session$$
$$products : \mathbb{P}\, ProductEntry$$

The *BuyerSession* schema represents the activities of a buyer when he/she enters the electronic commerce system. This schema inherits the *Session* schema described earlier and adds a new component, products. The new attribute is a list of all the product entries that the buyer has added to his/her electronic shopping cart. A *BuyerSession* also requires an initialization.

InitBuyerSession

$$\Delta BuyerSession$$

$$actions = \langle\rangle$$
$$products = \emptyset$$

The *InitBuyerSession* schema initializes the buyer's session. The schema sets the sequence of actions taken by the buyer to be empty, and sets the set of products chosen by the buyer in the current session to be empty. The ability to modify a *BuyerSession* is also important and this functionality is provided by the next operational schema.

ModifyBuyerSession

$$\Delta BuyerSession$$
$$id?, ow?, en?, ex? : \mathbb{N}_1$$

$$id? = id$$
$$owner' = ow?$$
$$entry' = en?$$
$$exit' = ex?$$

The *ModifyBuyerSession* schema modifies the attributes of a *BuyerSession* to represent changes. The schema accepts an id, owner, entry and exit time, and changes the values for a *BuyerSession* accordingly. In addition to modifying a *BuyerSession*, mechanisms to support the addition and removal of items from a buyer's shopping cart are vital.

AddBuyerSessionProductEntry

$$\Delta BuyerSession$$
$$i?, sid?, p?, st? : \mathbb{N}_1$$
$$n?, d? : STRING$$

$$create\_product\_entry(i?, sid?, n?, d?, p?, st?) \notin products$$
$$products' = products \cup \{create\_product\_entry(i?, sid?, n?, d?, p?, st?)\}$$

The *AddBuyerSessionProductEntry* adds a new product entry to the set of products in the buyer's electronic shopping cart. The schema does this using the *create_product_entry* function provided with the inputs for a product entry. The pre-condition of the schema operation is that the new product entry must not already exist in the set of product entries. Similarly, a schema to remove a product entry is defined as follows:

RemoveBuyerSessionProductEntry

$$\Delta BuyerSession$$
$$id? : \mathbb{N}_1$$

$$return\_product\_entry(id?, products) \subseteq products$$
$$products' = products \setminus return\_product\_entry(id?, products)$$

The *RemoveBuyerSessionProductEntry* schema is used when a buyer decides to remove a product entry from the list of products to purchase during a session. This schema uses the *return_product_entry* function to get the selected product from the electronic shopping cart. The initial constraint on the schema is that the product entry to be removed must exist in the set of products for the buyer.

The *BuyerSession* schema and other functions and operational schemas are used in the electronic commerce system by a buyer session handler. The definition of a buyer session handler follows.

BuyerSessionHandler

$$buyer\_sessions : \mathbb{P}\, BuyerSession$$

$$\forall x, y : BuyerSession \bullet x \in buyer\_sessions \land$$
$$y \in buyer\_sessions \Rightarrow x.id \neq y.id$$

The *BuyerSessionHandler* schema contains a set of all of the currently active buyer sessions in the electronic commerce system. Each buyer session must be unique. An additional

schema is required to initialize the buyer session handler.

InitBuyerSessionHandler

$$\Delta BuyerSessionHandler$$

$$buyer\_sessions' = \emptyset$$

The *InitBuyerSessionHandler* schema initializes the buyer session handler and sets the set of active buyer sessions to be empty. For use in the electronic commerce system, buyer sessions need to be added and removed from the session handler.

$$create\_buyer\_session : \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{N}_1 \times \text{iseq}\, Action \times \mathbb{P}\, ProductEntry \rightarrow BuyerSession$$

$$\forall id, ow, en, ex : \mathbb{N}_1;\ a : \text{iseq}\, Action;\ p : \mathbb{P}\, ProductEntry;\ bs : BuyerSession \bullet create\_buyer\_session(id, ow, en, ex, a, p) = bs \Rightarrow bs.id = id \land bs.owner = ow \land bs.entry = en \land bs.exit = ex \land bs.actions = a \land bs.products = p$$

The *create_buyer_session* operator creates a new buyer session. This function takes an id, owner, entry and exit time, sequence of actions and set of product entries and creates a buyer session. To use this function, an operational schema modifies the buyer session handler.

AddBuyerSession

$$\Delta BuyerSessionHandler$$
$$id?, ow?, en?, ex? : \mathbb{N}_1$$
$$a? : \text{iseq}\, Action$$
$$p? : \mathbb{P}\, ProductEntry$$

$$create\_buyer\_session(id?, ow?, en?, ex?, a?, p?) \notin buyer\_sessions$$
$$buyer\_sessions' = buyer\_sessions \cup \{create\_buyer\_session(id?, ow?, en?, ex?, a?, p?)\}$$

The *AddBuyerSession* schema adds a new buyer session to the list of currently active sessions. The schema states that the new session to be added must be unique to the list of current sessions and that it is created using the *create_buyer_session* definition.

The ability to remove buyer sessions from the buyer session handler is also required and is defined below.

$$return\_buyer\_session : \mathbb{N}_1 \times \mathbb{P}\, BuyerSession \rightarrow \mathbb{P}\, BuyerSession$$

$$\forall i : \mathbb{N}_1;\ bss, bs : \mathbb{P}\, BuyerSession \bullet$$
$$return\_buyer\_session(i, bss) = bs \Rightarrow bs \subseteq bss \land$$
$$(\forall x : BuyerSession \mid x \in bs \bullet x.id = i)$$

The *return_buyer_session* function finds a specified buyer session in a set of buyer sessions. The operator returns a set of buyer sessions where the id matches the id provided as an input to the definition. To remove the buyer session from the handler, the operational schema below uses *return_buyer_session*.

RemoveBuyerSession

$$\Delta BuyerSessionHandler$$
$$id? : \mathbb{N}_1$$

$$return\_buyer\_session(id?, buyer\_sessions) \subseteq buyer\_sessions$$
$$buyer\_sessions' = buyer\_sessions \setminus return\_buyer\_session(id?, buyer\_sessions)$$

The *RemoveBuyerSession* schema takes the id of a buyer session as an input and uses the *return_buyer_session* function to retrieve that buyer session from the list of buyer sessions in the buyer session handler. This retrieved value is then removed from the set of buyer sessions using the set difference operator. The pre-condition on the schema is that the buyer session to be removed must already exist in the set of buyer sessions stored in the buyer session handler.

Seller Schemas and Definitions

The seller is now described using schemas and functions. These definitions outline what components make up a seller and describe what possible operations a seller can perform. A schema describing the seller is given below.

Seller

$$id : \mathbb{N}_1$$
$$name, phone, email : STRING$$
$$address, shipping\_address : ADDRESS$$
$$payment\_method : PaymentMethod$$

The *Seller* schema represents each seller in the electronic commerce system. This schema contains information about a seller including a unique id, name, address, phone number,

email address, payment method and shipping address. The payment method is used to settle payments between buyers and sellers during a transaction and the shipping address can be provided to buyers to allow them to return goods for warranty or repair work. To make changes to a *Seller*, the *ModifySeller* operational schema is used as defined below.

ModifySeller

$$\Delta Seller$$
$$i? : \mathbb{N}_1$$
$$n?, p?, e? : STRING$$
$$a?, s\_a? : ADDRESS$$

$$id' = i?$$
$$name' = n?$$
$$address' = a?$$
$$phone' = p?$$
$$email' = e?$$
$$shipping\_address' = s\_a?$$

The *ModifySeller* schema takes a number of inputs for a seller and assigns them to the appropriate attributes. The only attribute not assigned at this time is the payment method attribute which is set by the *AddSellerPaymentMethod* schema defined below.

AddSellerPaymentMethod

$$\Delta Seller$$
$$pmid?, pma\_n? : \mathbb{N}_1$$
$$pmn?, pmfin? : STRING$$
$$pma\_c? : CODE$$
$$pmp\_t? : PAYMENT\_TYPE$$

$$payment\_method' = create\_payment\_method(pmid?, pma\_n?, pmn?, pmfin?, pma\_c?, pmp\_t?)$$

The *AddSellerPaymentMethod* schema modifies the seller schema by using the function *create_payment_method* to assign a new payment method to the seller. To accomplish the creation of the payment method, inputs for the payment method attributes are needed, including id, name, financial institution name, authorization code, and the payment type of the financial service. In addition to adding a new payment method to a *Seller*, the ability to modify that payment method is also required.

ModifySellerPaymentMethod

$\Delta Seller$
$i?, a\_n? : \mathbb{N}_1$
$n?, fin? : STRING$
$a\_c? : CODE$
$p\_t? : PAYMENT\_TYPE$

$i? = payment\_method.id$
$payment\_method'.name = n?$
$payment\_method'.finame = fin?$
$payment\_method'.acc\_num = a\_n?$
$payment\_method'.auth\_code = a\_c?$
$payment\_method'.payment\_type = p\_t?$

The *ModifySellerPaymentMethod* schema allows a seller to update a payment method. The schema uses a number of inputs matching the attributes of a payment method and applies them to the seller payment method accordingly.

The electronic commerce system tracks the activities of a seller by maintaining a session each time the seller visits the system. A session handler manages these seller sessions. The seller session handler is defined below.

SellerSessionHandler

$seller\_sessions : \mathbb{P}\, Session$

$\forall x, y : Session \bullet x \in seller\_sessions \land y \in seller\_sessions \Rightarrow x.id \neq y.id$

The *SellerSessionHandler* schema represents the session handler that manages all the sessions currently open by sellers in the electronic commerce system. This schema contains a set of sessions, each representing one seller that is currently active in the electronic commerce system. The schema also states that each session is unique. The initialization of a *SellerSessionHandler* is defined by the operational schema given below.

InitSellerSessionHandler

$\Delta SellerSessionHandler$

$seller\_sessions' = \emptyset$

The *InitSellerSessionHandler* schema initializes the seller session handler by setting the set of seller sessions to empty.

The seller session handler also requires the ability to add and remove sessions. These operational schemas are defined below.

AddSellerSession

$$\Delta SellerSessionHandler$$
$$id?, ow?, en?, ex? : \mathbb{N}_1$$

$$create\_session(id?, ow?, en?, ex?) \notin seller\_sessions$$
$$seller\_sessions' = seller\_sessions \cup \{create\_session(id?, ow?, en?, ex?)\}$$

The *AddSellerSession* schema adds a new seller session to the seller session handler. The schema takes as inputs the attributes needed to create a new session and creates a new session using the *create_session* function. The schema also states that the new seller session to be added must not exist in the seller sessions already stored by the seller session handler. The 'non-membership' operator($\notin$) is used to check that the results of the *create_session* function does not currently exist in the *seller_sessions*.The ability to remove sessions from the seller session handler is defined next.

RemoveSellerSession

$$\Delta SellerSessionHandler$$
$$id? : \mathbb{N}_1$$

$$return\_session(id?, seller\_sessions) \subseteq seller\_sessions$$
$$seller\_sessions' = seller\_sessions \setminus return\_session(id?, seller\_sessions)$$

The *RemoveSellerSession* schema removes a current seller session from the list of seller sessions by specifying its id and using the *return_session* function to find the correct seller session. The pre-condition on the schema is that the session to be removed must exist in the set of seller sessions.

A seller provides goods for sale in the electronic commerce system that are purchased through a supplier. Each of these suppliers sells products in the form of product lots. The sellers then make the products purchased in one of these lots available at lesser quantities to the buyers. The following schemas and functions describe this supply chain of goods for the sellers.

Product

$id, supid, price : \mathbb{N}_1$
$desc : STRING$

The *Product* schema represents an item that a supplier makes available to a seller for sale in the electronic commerce system. Each product has an id, a supplier providing the product, a description of its characteristics and a individual unit price. To modify the product information, an operational schema is presented next.

ModifyProduct

$\Delta Product$
$id?, p? : \mathbb{N}_1$
$d? : STRING$

$id' = id?$
$desc' = d?$
$price' = p?$

The *ModifyProduct* schema demonstrates the modification of a product schema. This operational schema takes the attributes of a product as inputs and sets them to be the new attributes of a product. Each product is unique, as stated by the constraint below.

$products : \mathbb{P}\, Product$

$\forall x, y : Product \bullet x \in products \land y \in products \Rightarrow x.id \neq y.id$

The *products* constraint specifies that every product across the entire electronic commerce system must be unique. Each product is presented to a seller by a supplier in a product lot, described below.

ProductLot

$id, quantity : \mathbb{N}_1$
$product : Product$

The *ProductLot* schema represents a quantity of a certain product made available to a seller by a supplier. Each product lot has a unique id, a product associated with it and the quantity of that product. The ability to modify a product lot is important to a supplier and the *ModifyProductLot* operational schema, defined below, provides this support.

ModifyProductLot

$$\Delta ProductLot$$
$$id?, q? : \mathbb{N}_1$$

$$id' = id?$$
$$quantity' = q?$$

The *ModifyProductLot* schema is used to change the quantity of a product and the lot id. In addition to modifying a product lot, a supplier also needs to be able to add a new product lot to the product lots that have currently been made available to sellers.

$$create\_product\_lot : \mathbb{N}_1 \times Product \times \mathbb{N}_1 \rightarrow ProductLot$$

$$\forall id, q : \mathbb{N}_1;\ p : Product;\ pl : ProductLot \bullet create\_product\_lot(id, p, q) = pl \Rightarrow$$
$$pl.id = id \land pl.product = p \land pl.quantity = q$$

The *create_product_lot* function is used to create a new product lot from the provided inputs. The schema takes the attributes for the creation of a new product lot and returns the completed item by assigning each of the input parameters to the appropriate attribute for the new product lot. To retrieve information or remove product lots, a function is required to return a product lot.

$$return\_product\_lot : \mathbb{N}_1 \times \mathbb{P}\, ProductLot \rightarrow \mathbb{P}\, ProductLot$$

$$\forall id : \mathbb{N}_1;\ pl, p : \mathbb{P}\, ProductLot \bullet return\_product\_lot(id, pl) = p \Rightarrow$$
$$p \subseteq pl \land (\forall x : ProductLot \mid x \in p \bullet x.id = id)$$

The *return_product_lot* function is used to return a selected product lot. It takes a set of product lots and a unique product lot id as inputs and returns the matching product lot in the set.

With the definition of products and product lots, a supplier and its operations can now be defined.

Supplier

$$name, phone : STRING$$
$$address : ADDRESS$$
$$product\_lots : \mathbb{P}\, ProductLot$$

The *Supplier* schema represents a supplier that makes product lots available to sellers for sale in the electronic commerce system. Each supplier has a name, address, and phone

number as well as a number of products lots available for sale. Next is an operational schema to initialize a supplier.

InitSupplier

$$\Delta Supplier$$

$$product\_lots' = \emptyset$$

The *InitSupplier* schema initializes the supplier schema by setting the product lots set to empty. After initialization other operations can be carried out on a supplier.

AddSupplierProductLot

$$\Delta Supplier$$
$$id?, q? : \mathbb{N}_1$$
$$p? : Product$$

$$create\_product\_lot(id?, p?, q?) \notin product\_lots$$
$$product\_lots' = product\_lots \cup \{create\_product\_lot(id?, p?, q?)\}$$

The *AddSupplierProductLot* schema adds a new product lot to those available for sale by the supplier. The schema takes inputs for the id, quantity, and product for a product lot and adds a new product lot to the existing lots using the *create_product_lot* function. The schema also states that the product lot must be unique relative to the other product lots offered by the supplier. To remove a product lot from a supplier the *RemoveSupplierProductLot* operational schema is defined.

RemoveSupplierProductLot

$$\Delta Supplier$$
$$id? : \mathbb{N}_1$$

$$return\_product\_lot(id?, product\_lots) \subseteq product\_lots$$
$$product\_lots' = product\_lots \setminus return\_product\_lot(id?, product\_lots)$$

The *RemoveSupplierProductLot* schema represents removal of a product lot from a supplier. The set of available product lots is reduced by the lot identified by an id given as an input to the schema and uses the *return_product_lot* function to retrieve the correct product lot from the current listing. There is a constraint on the schema that the product lot to be removed must be in the supplier's product lots.

Additional functions used by the electronic commerce system to retrieve information

from a seller are defined below. These functions are used during a purchase transaction.

$$return\_seller\_name : \mathbb{P}\, Seller \times \mathbb{N}_1 \rightarrow STRING$$

$$\forall\, sl : \mathbb{P}\, Seller;\ sid : \mathbb{N}_1;\ sn : STRING \bullet return\_seller\_name(sl, sid) = sn \Rightarrow (\exists_1\, x : Seller \mid x \in sl \bullet sn = x.name \Rightarrow x.id = sid)$$

The *return_seller_name* function returns the name of a seller from a list of all the sellers in the system given the id of the seller.

$$return\_seller\_address : \mathbb{P}\, Seller \times \mathbb{N}_1 \rightarrow ADDRESS$$

$$\forall\, sl : \mathbb{P}\, Seller;\ sid : \mathbb{N}_1;\ sa : ADDRESS \bullet return\_seller\_address(sl, sid) = sa \Rightarrow (\exists_1\, x : Seller \mid x \in sl \bullet sa = x.address \Rightarrow x.id = sid)$$

The *return_seller_address* function is similar to the *return_seller_name* function. The *return_seller_address* operation returns the address of a seller given a unique id and a set of all the sellers in the electronic commerce system.

$$return\_seller\_payment\_method : \mathbb{P}\, Seller \times \mathbb{N}_1 \rightarrow PaymentMethod$$

$$\forall\, sl : \mathbb{P}\, Seller;\ sid : \mathbb{N}_1;\ spm : PaymentMethod \bullet return\_seller\_payment\_method(sl, sid) = spm \Rightarrow (\exists_1\, x : Seller \mid x \in sl \bullet spm = x.payment\_method \Rightarrow x.id = sid)$$

The *return_seller_payment_method* function is used to return the payment method associated with a seller in the system identified by an id given as an input parameter.

Master Inventory Schemas and Definitions

The inventory of products available for sale are specified next. These products are supplied by different sellers and are available to all buyers. The schemas and functions presented below define the components of the master inventory.

MasterInventory

$$available\_products : \mathbb{P}\, ProductEntry$$

$$\forall\, x, y : ProductEntry \bullet x \in available\_products \wedge y \in available\_products \Rightarrow x.id \neq y.id$$

The *MasterInventory* schema represents the inventory of all products available in the system. The list of all the products is stored as a set of product entries. The master inventory is used for querying and selecting goods for purchase by the users of the system.

A pre-condition on this schema is that each product entry is unique. The initialization of the master inventory is defined as follows.

InitMasterInventory

$$\Delta MasterInventory$$

$$available\_products = \emptyset$$

The *InitMasterInventory* schema initializes the master inventory by setting the initial set of products available to empty. The ability to add and remove product entries to/from the master inventory is important and the operations to support this functionality are defined below.

AddProductEntry

$$\Delta MasterInventory$$
$$i?, sid?, p?, st? : \mathbb{N}_1$$
$$n?, k? : STRING$$

$$create\_product\_entry(i?, sid?, n?, k?, p?, st?) \notin available\_products$$
$$available\_products' = available\_products \cup \{create\_product\_entry(i?, sid?, n?, k?, p?, st?)\}$$

The *AddProductEntry* schema creates a new product entry and adds it to the set of available products. The schema accomplishes this by using the *create_product_entry* function. A pre-condition of the operation is that the product entry to be created must not exist in the available products.

The operational schema to remove a product entry from the master inventory is defined as follows:

RemoveProductEntry

$$\Delta MasterInventory$$
$$id? : \mathbb{N}_1$$

$$return\_product\_entry(id?, available\_products) \subseteq available\_products$$
$$available\_products' = available\_products \setminus return\_product\_entry(id?, available\_products)$$

The *RemoveProductEntry* schema removes a product entry from the list of available products by identifying the product entry to be removed using the *return_product_entry*

function with an id supplied as an input. A pre-condition on this schema is that the product entry to be removed must exist in the set of available products.

$$master\_inventory\_product\_search : \mathbb{P}\, ProductEntry \times STRING \rightarrow \mathbb{P}\, ProductEntry$$

$$\forall\, pl, r : \mathbb{P}\, ProductEntry;\ k : STRING \bullet$$
$$master\_inventory\_product\_search(pl, k) = r \Rightarrow$$
$$(\forall\, x : ProductEntry \mid x \in r \bullet x.keywords = k)$$

The *master_inventory_product_search* function takes a set of product entries and search criteria as inputs and returns a set of product entries where a keyword matches the search criteria provided. In this specification, this is shown as an equality even though in a real electronic commerce system the agent searching for goods would be much more specific in its search criteria and allow for many other options including less than, greater than, etc. For simplicity and conciseness, only equality is shown. This function is used during the search process in the electronic commerce system.

Search Engine Schemas and Definitions

This search capabilities in the electronic commerce system are described below using schemas and functions. These definitions specify the components of a search and their possible operations. A schema describing a search is given below.

Search

$$id : \mathbb{N}_1$$
$$criteria : STRING$$
$$results : \mathbb{P}\, ProductEntry$$

The *Search* schema represents a single search activity executed by the search engine. Each search has a unique id, some search criteria and a set of results containing product entries. The initialization of the search schema follows.

InitSearch

$$\Delta Search$$

$$results' = \emptyset$$

The *InitSearch* schema initializes the search schema by setting the results obtained to

the empty set. The ability to modify a search is also important.

ModifySearch

$$\Delta Search$$
$$id? : \mathbb{N}_1$$
$$c? : STRING$$

$$id' = id?$$
$$criteria' = c?$$

The *ModifySearch* schema permits the modification of the id or criteria of a search. These new values are inputs to the schema and are assigned accordingly. After a search is initialized and values have possibly been modified, the electronic commerce system needs the ability to execute the search for products against the master inventory.

ExecuteSearch

$$\Delta Search$$
$$mi? : MasterInventory$$

$$results' = master\_inventory\_product\_search(mi?.available\_products, criteria)$$

The *ExecuteSearch* schema modifies the *Search* schema by setting the results to be a set of product entries using the *master_inventory_product_search* function to return results from the master inventory. Searches are created and deleted by the search engine in the electronic commerce system. The following schemas and functions describe the search engine and its operations.

$$create\_search : \mathbb{N}_1 \times STRING \rightarrow Search$$

$$\forall id : \mathbb{N}_1;\ c : STRING;\ s : Search \bullet create\_search(id, c) = s \Rightarrow$$
$$s.id = id \land s.criteria = c \land s.results = \emptyset$$

The *create_search* function takes a unique id and a string criteria and creates a new search. The id and criteria are set to the appropriate inputs while the set of results is set to the empty set. The function describing how a search is identified by the search engine is described next.

$$return\_search : \mathbb{N}_1 \times \mathbb{P}\, Search \rightarrow \mathbb{P}\, Search$$

$$\forall i : \mathbb{N}_1;\ s, sr : \mathbb{P}\, Search \bullet return\_search(i, s) = sr \Rightarrow$$
$$sr \subseteq s \land (\forall x : Search \mid x \in sr \bullet x.id = i)$$

The *return_search* function finds and returns a search description from a listing of all search descriptions. An input of an id is given by the search engine and the result of the function is the search description that contains an id matching that value. The search engine, which holds the active search descriptions in the system, is described as follows:

SearchEngine

$$current\_searches : \mathbb{P}\, Search$$

The *SearchEngine* schema manages all the active search descriptions in the system. This schema contains a set of all the search descriptions currently in the system. The *InitSearchEngine* schema describes the initialization of the *SearchEngine*.

InitSearchEngine

$$\Delta SearchEngine$$

$$current\_searches = \emptyset$$

The *InitSearchEngine* schema initializes the search engine so that the set of current search descriptions is empty. As the buyers and sellers request search descriptions, the search descriptions are added to the search engine. The operation to add a search description is specified below.

AddSearch

$$\Delta SearchEngine$$
$$i? : \mathbb{N}_1$$
$$c? : STRING$$

$$create\_search(i?, c?) \notin current\_searches$$
$$current\_searches' = current\_searches \cup \{create\_search(i?, c?)\}$$

The *AddSearch* schema adds a new search description to the set of current search descriptions in the search engine. This is accomplished by using the *create_search* function. A pre-condition on the search description to be added is that it must not be the same as any of the other search descriptions currently in the system.

The schema below describes the removal of a search description.

RemoveSearch

$\Delta SearchEngine$
$id? : \mathbb{N}_1$

$return\_search(id?, current\_searches) \subseteq current\_searches$
$current\_searches' = current\_searches \setminus return\_search(id?, current\_searches)$

The *RemoveSearch* schema takes a unique id and removes the search description containing that id from its list of current search descriptions. The removal is accomplished by first using the *return_search* function to identify the search description and then use the set exclusion operator to remove the identified search description from the set of search descriptions. The constraint on this schema is that the search description to be removed must exist in the set of current search descriptions.

Comparison Engine Schemas and Definitions

This product comparison capabilities in the electronic commerce system are described next. In the context of this electronic commerce system, a comparison is a task applied to the master inventory of to retrieve information on what products best match criteria provided by the user. The following definitions abstract the components of comparisons and comparison operations. To define the results of a comparison, it is necessary to give a ranking of each product that applies to the logic provided in a comparison. A schema representing this ranking is presented next.

ProdRank

$pid, cid, rank : \mathbb{N}_1$

$rank \leq 100$

The *ProdRank* schema describes the result of a comparison product characteristic($ProdChar$) applied to a particular product. Each product is ranked on each criteria provided and returned to the comparison for calculation of a final product entry listing. A schema describing comparisons is given below.

Comparison

$id : \mathbb{N}_1$
$chars : \text{iseq}\, ProdChar$
$rankings : \mathbb{P}\, ProdRank$
$results : \text{iseq}\, ProductEntry$

The *Comparison* schema represents a comparison of products in the system. Each comparison has an id, comparison operators in the form of product characteristics desired, a set of product rankings, and a set of product entries which match the criteria. Initialization of a comparison is specified below.

InitComparison

$\Delta Comparison$

$chars' = \langle\rangle$
$rankings' = \emptyset$
$results' = \langle\rangle$

The *InitComparison* schema initializes a comparison by setting the set of characteristics, rankings, and results to be empty. A function to delete a comparison from a sequence of comparisons is defined as follows:

$delete\_prod\_char : \mathbb{N}_1 \times \text{iseq}\, ProdChar \rightarrow \text{iseq}\, ProdChar$

$\forall\, pid : \mathbb{N}_1;\ old : \text{iseq}\, ProdChar \bullet$
$(old = \langle\rangle \Rightarrow delete\_prod\_char(pid, old) = \langle\rangle) \land$
$((head\ old).id = pid \Rightarrow delete\_prod\_char(pid, old) = tail\ old) \land$
$((head\ old).id \neq pid \Rightarrow delete\_prod\_char(pid, old) =$
$\langle head\ old\rangle \frown delete\_prod\_char(pid, tail\ old))$

The *delete_prod_char* function deletes a product characteristic from a sequence of product characteristics given an id. This function recursively searches the sequence until it finds the product characteristic identified by the id given as a parameter. Similarly, the *AddCompChar* schema provides the ability to add product characteristics to a comparison.

AddCompChar

$\Delta Comparison$
$i?, pid? : \mathbb{N}_1$
$n?, v? : STRING$

$delete\_prod\_char(i?, chars) = chars$
$chars' = chars \frown \langle create\_prod\_char(i?, pid?, n?, v?) \rangle$

The *AddCompChar* schema takes a new id, product entry id, name, and value and creates a product characteristic in a comparison. The characteristic created must not have the same id as another characteristic for that comparison. The operational schema to remove a characteristic from a comparison is presented next.

RemoveCompChar

$\Delta Comparison$
$id? : \mathbb{N}_1$

$chars' = delete\_prod\_char(id?, chars)$

The *RemoveCompChar* schema uses the *delete_prod_char* function to delete a specified characteristic from the sequence of product characteristics used in the comparison.

$compute\_rank : STRING \times STRING \rightarrow \mathbb{N}_1$

The *compute_rank* function determines the strength of a match, a value between 0 and 100, between two values of a characteristic. If the characteristic values match exactly, the value would be 100 and if they have no similarity or likeness a value of 0 would be returned. The exact value will be determined by an artificial intelligence module represented by this function.

$get\_rank : STRING \times STRING \times \mathbb{P}\, ProdChar \rightarrow \mathbb{N}_1$

$\forall n, v : STRING;\ pc : \mathbb{P}\, ProdChar;\ r : \mathbb{N}_1 \bullet get\_rank(n, v, pc) = r \Rightarrow$
$\quad (\exists_1 x : ProdChar \mid x \in pc \bullet r = compute\_rank(v, x.value) \Rightarrow n = x.name) \lor$
$\quad (\forall y : ProdChar \mid y \in pc \bullet r = 0 \Rightarrow n \neq y.name)$

The *get_rank* function takes the name and value of a comparison characteristic and a set of product characteristics and sets the rank of the product ranking to either the results of the *compute_rank* function or 0. The value assigned depends on the degree of match of the name passed as a parameter to the name of a product characteristic from the set of

products' characteristics. This function is used by the operation described below.

$$rank\_product : ProdChar \times ProductEntry \rightarrow ProdRank$$

$$\begin{aligned}
&\forall pc : ProdChar;\ pe : ProductEntry;\ pr : ProdRank \bullet rank_product(pc, pe) = pr \Rightarrow \\
&\quad pr.pid = pe.id \land pr.cid = pc.id \land \\
&\quad pr.rank = get_rank(pc.name, pc.value, pe.chars)
\end{aligned}$$

The *rank_product* function creates a new product ranking based on the desired product characteristic and the product entry provided as inputs. This function uses the *get_rank* function to compute the value for the product ranking. This function is used by the definition below.

$$return\_rankings : ProdChar \times \mathbb{P}\, ProductEntry \rightarrow \mathbb{P}\, ProdRank$$

$$\begin{aligned}
&\forall pc : ProdChar;\ mi : \mathbb{P}\, ProductEntry;\ pr : \mathbb{P}\, ProdRank \bullet \\
&\quad return_rankings(pc, mi) = pr \Rightarrow \\
&\quad (\forall pe : ProductEntry \mid pe \in mi \bullet \\
&\qquad pr \neq \emptyset \Rightarrow pr = pr \cup \{rank_product(pc, pe)\} \land \\
&\qquad pr = \emptyset \Rightarrow pr = \{rank_product(pc, pe)\})
\end{aligned}$$

The *return_rankings* function applies the *rank_product* function to each member of a set of product entries taken as an input. This set of product entries is the master inventory in the system and the results are returned as a set of product rankings. The function *return_comparison_rankings*, described below, uses *return_rankings* as part of its processing.

$$return\_comparison\_rankings : \mathbb{P}\, ProductEntry \times \mathrm{iseq}\, ProdChar \rightarrow \mathbb{P}\, ProdRank$$

$$\begin{aligned}
&\forall mi : \mathbb{P}\, ProductEntry;\ pc : \mathrm{iseq}\, ProdChar \bullet \\
&\quad (pc = \langle\rangle \Rightarrow return_comparison_rankings(mi, pc) = \emptyset) \land \\
&\quad (pc \neq \langle\rangle \Rightarrow return_comparison_rankings(mi, pc) = \\
&\qquad return_rankings(head\ pc, mi) \cup return_comparison_rankings(mi, tail\ pc))
\end{aligned}$$

The *return_comparison_rankings* function takes a set of product entries, in this case the entire master inventory, and a sequence of comparison characteristics and returns a set of product rankings that at least partially match the conditions set out in the logic. This definition is based on some artificial intelligence used by the electronic commerce system to arrive at the correct set of products and provide the requestor with the product entries that will suit their needs. Now that the comparison method has been described, an operational schema can be shown that updates a *Comparison* with rankings.

DetermineRankings

$\Delta Comparison$
$mi? : MasterInventory$

$chars \neq \langle\rangle$
$rankings' = return\_comparison\_rankings(mi?.available\_products, chars)$

The *DetermineRankings* schema uses the *return_comparison_rankings* function to determine the rankings of a comparison using the comparison operators(desired product characteristics) stored for that comparison. A pre-condition on the schema is that the list of characteristics to compare against must not be empty. To build the comparison results from the rankings, additional definitions are necessary.

$create\_comparison\_results : \mathbb{P}\, ProdRank \times \mathbb{P}\, ProductEntry \rightarrow \text{iseq}\, ProductEntry$

The *create_comparison_results* function takes a listing of product rankings along with the master inventory and builds a set containing the ordered results of the comparison. The operational schema defined below demonstrates this function.

DetermineResults

$\Delta Comparison$
$mi? : MasterInventory$

$results' = create\_comparison\_results(rankings, mi?.available\_products)$

The *DetermineResults* schema uses the *return_comparison_results* function to provide a listing of the product entries that best match the criteria provided by a comparison.

Comparisons are stored in the electronic commerce system and are described by the following functions and schemas.

$create\_comparison : \mathbb{N}_1 \rightarrow Comparison$

$\forall id : \mathbb{N}_1;\ c : Comparison \bullet create\_comparison(id) = c \Rightarrow$
$\quad c.id = id \land c.chars = \langle\rangle \land c.rankings = \emptyset \land c.results = \emptyset$

The *create_comparison* function creates a new comparison from given input. It takes a unique id and initializes the remaining attributes of a comparison. To return an existing comparison for information purposes or for deletion, the function below is useful.

$$return\_comparison : \mathbb{N}_1 \times \mathbb{P}\, Comparison \rightarrow \mathbb{P}\, Comparison$$

$$\forall i : \mathbb{N}_1;\ c, cr : \mathbb{P}\, Comparison \bullet return\_comparison(i, c) = cr \Rightarrow cr \subseteq c \wedge (\forall x : Comparison \mid x \in cr \bullet x.id = i)$$

The *return_comparison* function retrieves a desired comparison from the set of comparisons given an id as an input. This function can be used to retrieve information about a comparison or be used in the process of deletion of a comparison. The management of comparisons in the electronic commerce system is described next.

ComparisonEngine

$$current\_comparisons : \mathbb{P}\, Comparison$$

The *ComparisonEngine* schema manages all the comparisons in the electronic commerce system. This schema keeps a list of all the current comparisons in the system and this list grows and shrinks as comparisons are added or deleted. To be operational, the comparison engine must first be initialized. The operational schema representing this initialization is defined as follows.

InitComparisonEngine

$$\Delta ComparisonEngine$$

$$current\_comparisons = \emptyset$$

The *InitComparisonEngine* schema initializes the comparison engine by setting the set of current comparisons to be empty. The comparison engine must be able to add and remove comparisons from its listing.

AddComparison

$$\Delta ComparisonEngine$$
$$i? : \mathbb{N}_1$$

$$create\_comparison(i?) \notin current\_comparisons$$
$$current\_comparisons' = current\_comparisons \cup \{create\_comparison(i?)\}$$

The *AddComparison* schema adds a new comparison to the comparison engine's list of active comparisons. This schema uses the *create_comparison* function, and an id and comparison operators to create the new comparison. The operational schema for removal of a comparison is similarly defined as:

RemoveComparison

$\Delta ComparisonEngine$
$id? : \mathbb{N}_1$

$return\_comparison(id?, current\_comparisons) \subseteq current\_comparisons$
$current\_comparisons' = current\_comparisons \setminus$
$\quad return\_comparison(id?, current\_comparisons)$

The *RemoveComparison* schema removes a comparison identified by the input id using the *return_comparison* definition. This schema is used when a comparison is complete or has been cancelled by the user or the system. A pre-condition on the schema is that the comparison to be removed must be a member of the set of current comparisons.

Transaction Schemas and Definitions

Purchase transactions are now described using schemas and functions. These definitions outline the components of transactions and their possible operations. A schema describing an orderline, a component of a transaction, is given below.

OrderLine

$product\_id, shipping\_date, quantity : \mathbb{N}_1$
$name : STRING$
$shipping\_address : ADDRESS$

The *OrderLine* schema represents a request from the electronic commerce system to a seller to ship a quantity of a single product to a buyer. An order line consists of the id of the product purchased, the name and shipping address to appear on the shipping label, the date the order must be shipped by, and the quantity that is to be shipped. A function that is used by a *SellerOrder* to create a new order line is presented next.

$create\_order\_line : \mathbb{N}_1 \times STRING \times ADDRESS \times \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow OrderLine$

$\forall pid, sd, qu : \mathbb{N}_1;\ n : STRING;\ a : ADDRESS;\ o : OrderLine \bullet$
$\quad create\_order\_line(pid, n, a, sd, qu) = o \Rightarrow o.product\_id = pid \wedge$
$\quad\quad o.name = n \wedge o.shipping\_address = a \wedge$
$\quad\quad\quad o.shipping\_date = sd \wedge o.quantity = qu$

The *create_order_line* function is used to create a new order line for a *SellerOrder*. The function takes all of the attributes of an *OrderLine* as inputs and creates a new order line by

assigning those inputs to the corresponding values in the order line schema. The *SellerOrder* schema is defined below:

SellerOrder

$$seller\_id : \mathbb{N}_1$$
$$order\_lines : \mathbb{P}\ OrderLine$$

$$\forall a, b : OrderLine \bullet a \in order\_lines \land b \in order\_lines \Rightarrow a \neq b$$

The *SellerOrder* schema represents a collection of *OrderLine* objects meant for a single seller as the result of a transaction. The schema contains the id of the seller to send the order to and a list of order lines. The constraint on the schema states that each order line in the *SellerOrder* must be unique. The next schema, *InitSellerOrder*, initializes the *SellerOrder*.

InitSellerOrder

$$\Delta SellerOrder$$
$$sid? : \mathbb{N}_1$$

$$seller\_id' = sid?$$
$$order\_lines' = \emptyset$$

The *InitSellerOrder* schema initializes the *SellerOrder* schema by setting the seller id to the value of the corresponding input and the set of order lines to empty. Another component used during a transaction, *ProductPurchase*, is defined as:

ProductPurchase

$$id, quantity : \mathbb{N}_1$$
$$product : ProductEntry$$
$$shipping\_address : ADDRESS$$
$$payment\_method : PaymentMethod$$
$$status : STATUS$$

The *ProductPurchase* schema represents the purchase of a product in the electronic commerce system by a buyer. Each product purchase is specific to one product, shipping address and payment method. The attributes of a product purchase include an id, the product entry for the product, the shipping address for the goods, the payment method to pay for the goods, the quantity of product purchased, and the current status. A function

to create a new product purchase, is presented below.

$$
\begin{array}{|l}
\hline
create\_product\_purchase : \mathbb{N}_1 \times ProductEntry \times ADDRESS \times \\
\qquad\qquad PaymentMethod \times \mathbb{N}_1 \rightarrow ProductPurchase \\
\hline
\forall\, i, qu : \mathbb{N}_1;\ pe : ProductEntry;\ a : ADDRESS; \\
\quad pm : PaymentMethod;\ p : ProductPurchase \bullet \\
\qquad create\_product\_purchase(i, pe, a, pm, qu) = p \Rightarrow \\
\qquad\quad p.id = i \land p.product = pe \land p.shipping\_address = a \land \\
\qquad\quad p.payment\_method = pm \land p.quantity = qu \land p.status = start \\
\hline
\end{array}
$$

The *create_product_purchase* function creates a new product purchase. It takes inputs mirroring the attributes of a product purchase and binds them appropriately.

With the specification of *ProductPurchase*, a schema to add a new order line is defined as follows.

AddOrderLine

$$
\begin{array}{|l}
\Delta SellerOrder \\
n? : STRING \\
pp? : ProductPurchase \\
\hline
order\_lines' = order\_lines \cup \{create\_order\_line(pp?.product.id, n?, \\
\quad pp?.shipping\_address, pp?.product.ship\_time + sys\_time, pp?.quantity)\} \\
\hline
\end{array}
$$

The *AddOrderLine* schema takes the name to appear on the shipping label as an input parameter and uses the *create_order_line* function to create an order for the seller to fulfill. The parameters passed to the *create_order_line* definition come from the existing information stored in the product purchase. The effective shipping date is determined by taking the shipping time from the product entry and adding the current system time, defined at the beginning of the specification.

SellerPurchase

$$
\begin{array}{|l}
id, seller\_id : \mathbb{N}_1 \\
seller\_order : SellerOrder \\
product\_purchases : \mathbb{P}_1\ ProductPurchase \\
\hline
\forall\, x, y : ProductPurchase \bullet x \in product\_purchases \land y \in product\_purchases \Rightarrow \\
\qquad x.id \neq y.id \\
\hline
\end{array}
$$

The *SellerPurchase* schema represents all the information about the products being purchased from a single seller by a buyer. This schema contains an id, the seller's id, the

order sent to that seller for shipping, and a list of the specific product purchases involved. The constraint is that each product purchase must be unique. The *InitSellerPurchase* schema initializes the *SellerPurchase* schema.

InitSellerPurchase

$$\Delta SellerPurchase$$
$$id?, sid? : \mathbb{N}_1$$
$$pp? : \mathbb{P}_1\ ProductPurchase$$

$$id' = id?$$
$$seller\_id' = sid?$$
$$product\_purchases' = pp?$$

The *InitSellerPurchase* operation sets the initial values of the *SellerPurchase* schema. The input values are the id, seller id, and a listing of product purchases. The function presented below, *create_seller_purchase*, is needed by the transaction to add a seller purchase.

$$create\_seller\_purchase : \mathbb{N}_1 \times \mathbb{N}_1 \times \mathbb{P}_1\ ProductPurchase \rightarrow SellerPurchase$$

$$\forall id, sid : \mathbb{N}_1;\ pp : \mathbb{P}_1\ ProductPurchase;\ sp : SellerPurchase \bullet$$
$$create\_seller\_purchase(id, sid, pp) = sp \Rightarrow$$
$$sp.id = id \land sp.seller\_id = sid \land sp.product\_purchases = pp$$

The *create_seller_purchase* function creates a new *SellerPurchase* given the inputs as parameters. This function is used in the processing of user transactions.

Upon successful completion of a transaction, an invoice is sent to the buyer showing the details of the purchase. The schemas and functions necessary to specify an invoice are presented next.

BuyerDetails

$$name, phone, email : STRING$$
$$address : ADDRESS$$

The *BuyerDetails* schema represents the portion of an invoice that contains the buyer information including name, address, phone number and email. The schema describing information about each purchase is described below.

PurchaseDetails

$$seller\_name, product\_name, payment\_method\_name : STRING$$
$$seller\_address, shipping\_address : ADDRESS$$
$$product\_price, shipping\_time, quantity : \mathbb{N}_1$$

The *PurchaseDetails* schema represents the information stored about one product involved in a transaction. Each purchase detail record contains the seller's name and address, product name and price, the quantity of product, the shipping time and address, and the payment method name. For an entire transaction all *PurchaseDetails* are combined and displayed together on a single invoice.

Invoice

$$buyer\_details : BuyerDetails$$
$$purchases : \mathbb{P}_1\ PurchaseDetails$$

The *Invoice* schema represents the document given to the buyer at the completion of a transaction in the electronic commerce system. The invoice contains information about the buyer involved in the transaction and a purchase detail entry for each product purchased in the transaction. Other operations on an invoice are described next, beginning with a function used to create buyer details.

$$create\_buyer\_details : \mathbb{N}_1 \times \mathbb{P}_1\ Buyer \rightarrow BuyerDetails$$

$$\forall bid : \mathbb{N}_1;\ bl : \mathbb{P}_1\ Buyer;\ bd : BuyerDetails \bullet create\_buyer\_details(bid, bl) = bd \Rightarrow (\exists_1 x : Buyer \mid x \in bl \bullet x.id = bid \wedge bd.name = x.name \wedge bd.address = x.address \wedge bd.phone = x.phone \wedge bd.email = x.email)$$

The *create_buyer_details* function takes an id and a buyer and creates a buyer details schema by assigning the appropriate values to the buyer details schema from the buyer schema. This function is used when an invoice is created during the processing of a transaction. The product details entry for each product purchased is created using a similar function defined below.

$$create\_product\_details : ProductPurchase \times \mathbb{N}_1 \times \mathbb{P}_1\, Seller \rightarrow PurchaseDetails$$

$$\begin{aligned}
&\forall\, pp : ProductPurchase;\ sid : \mathbb{N}_1;\ sl : \mathbb{P}_1\, Seller;\ pd : PurchaseDetails \bullet \\
&\quad create_product_details(pp, sid, sl) = pd \Rightarrow \\
&\quad pd.seller_name = return_seller_name(sl, sid) \land \\
&\quad pd.seller_address = return_seller_address(sl, sid) \land \\
&\quad pd.product_name = pp.product.name \land \\
&\quad pd.product_price = pp.product.price \land \\
&\quad pd.quantity = pp.quantity \land \\
&\quad pd.shipping_time = pp.product.ship_time \land \\
&\quad pd.shipping_address = pp.shipping_address \land \\
&\quad pd.payment_method_name = pp.payment_method.name
\end{aligned}$$

The *create_product_details* function builds purchase details using a seller and the product purchase information. The purchase details are built by assigning the correct attributes from the seller and product purchase to define the purchase details needed for the invoice. The attributes needed from the seller are obtained by using the *return_seller_name* and *return_seller_address* functions. A function is needed to modify the structure of the set of product purchases to clarify its use in the specification. The signature of the function, *transform_product_purchases*, is given as:

$$transform\_product\_purchases : \mathbb{P}\, ProductPurchase \rightarrow \text{seq}\, ProductPurchase$$

Since the *create_product_details* function creates the product details for a single product purchase in a transaction, a function is needed to create all the product details for the *SellerPurchase*.

$$create\_purchase\_details : \text{seq}\, ProductPurchase \times \mathbb{N}_1 \times \mathbb{P}_1\, Seller \rightarrow \mathbb{P}\, PurchaseDetails$$

$$\begin{aligned}
&\forall\, pp : \text{seq}\, ProductPurchase;\ sid : \mathbb{N}_1;\ sl : \mathbb{P}_1\, Seller;\ pd : \mathbb{P}\, PurchaseDetails \bullet \\
&\quad (pp = \langle\rangle \Rightarrow create_purchase_details(pp, sid, sl) = \emptyset) \land \\
&\quad (\#pp > 0 \Rightarrow create_purchase_details(pp, sid, sl) = \\
&\qquad \{create_product_details(head\ pp, sid, sl)\} \cup \\
&\qquad\quad create_purchase_details(tail\ pp, sid, sl))
\end{aligned}$$

The *create_purchase_details* function uses a sequence of *ProductPurchases* from a *SellerPurchase* to generate all the purchase details for the invoice. This function recursively processes the sequence of *ProductPurchases* and applies the function *create_product_details* to each. The *create_invoice_purchases* function, defined below, uses the *create_purchase_details* function.

$$create\_invoice\_purchases : \text{seq}\, SellerPurchase \times \mathbb{P}_1\, Seller \rightarrow \mathbb{P}\, PurchaseDetails$$

$$\begin{aligned}
&\forall sp : \text{seq}\, SellerPurchase;\ sl : \mathbb{P}_1\, Seller;\ pd : \mathbb{P}\, PurchaseDetails \bullet \\
&\quad (sp = \langle\rangle \Rightarrow create_invoice_purchases(sp, sl) = \emptyset) \land \\
&\quad (\#sp > 0 \Rightarrow create_invoice_purchases(sp, sl) = create_purchase_details(\\
&\qquad transform_product_purchases((head\ sp).product_purchases), \\
&\qquad\quad (head\ sp).seller_id, sl) \cup create_invoice_purchases(tail\ sp, sl))
\end{aligned}$$

The *create_invoice_purchases* function takes a sequence of *SellerPurchases* and a set of sellers and returns a set of purchase details. If the sequence of seller purchases is empty, which represents cases where there are no more seller purchases to create purchase details for, the empty set is returned. Otherwise, if there are elements in the seller purchase sequence the returned value is set to the first entry in the seller purchase sequence along with the list of sellers passed to the *create_purchase_details* function combined with the *create_invoice_purchases* function applied to the remaining product purchases. A function, *create_invoice*, to create a new invoice for use in the transaction is now described.

$$create\_invoice : \text{seq}\, SellerPurchase \times \mathbb{P}_1\, Seller \times \mathbb{P}_1\, Buyer \times \mathbb{N}_1 \rightarrow Invoice$$

$$\begin{aligned}
&\forall sp : \text{seq}\, SellerPurchase;\ sl : \mathbb{P}_1\, Seller;\ bl : \mathbb{P}_1\, Buyer;\ bid : \mathbb{N}_1;\ in : Invoice \bullet \\
&\quad create_invoice(sp, sl, bl, bid) = in \Rightarrow \\
&\qquad in.buyer_details = create_buyer_details(bid, bl) \land \\
&\qquad\quad in.purchases = create_invoice_purchases(sp, sl)
\end{aligned}$$

The *create_invoice* function builds an invoice by assembling the buyer and purchase details. The function takes a sequence of seller purchases, the list of sellers, the list of buyers, and a buyer id and creates an invoice. The function uses the *create_buyer_details* and *create_invoice_purchases* functions.

With the definition of the subcomponents of a transaction complete, schemas and functions describing a transaction and a management mechanism for transactions are presented.

Transaction

$$\begin{aligned}
&id, buyer_id : \mathbb{N}_1 \\
&seller_purchases : \mathbb{P}\, SellerPurchase \\
&invoice : Invoice \\
&status : STATUS
\end{aligned}$$

$$\begin{aligned}
&\forall x, y : SellerPurchase \bullet x \in seller_purchases \land y \in seller_purchases \Rightarrow \\
&\quad x.id \neq y.id
\end{aligned}$$

The *Transaction* schema represents a buyer's attempt to purchase some goods in the electronic commerce system from one or more sellers. Each transaction has an id, the buyer's id, a set containing the seller purchases, an invoice and a status for the transaction. A constraint on a transaction is that each seller purchase must be unique. An additional constraint on transactions is given below.

$$transactions : \mathbb{P}\ Transaction$$

$$\forall x, y : Transaction \bullet x \in transactions \land y \in transactions \Rightarrow x.id \neq y.id$$

This constraint states that each transaction in the system must be unique. The initial state of a transaction is defined by:

InitTransaction

$$\Delta Transaction$$

$$seller\_purchases' = \emptyset$$
$$status' = start$$

The *InitTransaction* schema initializes the transaction so that the set of seller purchases is empty and the status is set to *start*. This signifies that the transaction has begun and has not ended either correctly or in error.

With the definition of a transaction and its initial state, operational schemas and functions can be presented to show the creation and use of the information stored in a transaction.

$$transform\_seller\_purchases : \mathbb{P}\ SellerPurchase \rightarrow \text{seq}\ SellerPurchase$$

The *transform_seller_purchases* definition changes the set of seller purchases into a sequence of seller purchases. This function is useful in the definition of the operational schema, *AddTransactionInvoice* defined below, because it aids in the use of the *create_invoice* function.

AddTransactionInvoice

$\Delta Transaction$
$bl? : \mathbb{P}\, Buyer$
$sl? : \mathbb{P}\, Seller$

$status = complete$
$invoice' = create\_invoice(transform\_seller\_purchases(seller\_purchases),$
$\quad sl?, bl?, buyer\_id)$

The *AddTransactionInvoice* schema creates the invoice for the transaction by using the *transform_seller_purchases* and *create_invoice* functions along with the list of buyers and sellers in the electronic commerce system. A pre-condition on the creation of an invoice is that status is equal to *complete*, signifying the correct end to a transaction and providing impetus to produce the invoice.

To manage transactions, the *TransactionManager* schema is defined, which describes the management of all transactions in the electronic commerce system.

TransactionManager

$active, completed : \mathbb{P}\, Transaction$

$active \cap completed = \emptyset$
$\forall x, y : Transaction \bullet x \in active \land y \in completed \Rightarrow x.id \neq y.id$

The *TransactionManager* schema is responsible for the management of all the transactions in the system. This schema contains a list of all the completed and active transactions in the system. Constraints on this schema are that each transaction be unique and that each transaction must exist in either *active* or *completed*. The next schema defines the initialization of the transaction manager.

InitTransactionManager

$\Delta TransactionManager$

$active' = \emptyset$
$completed' = \emptyset$

The *InitTransactionManager* schema initializes the transaction manager so that the sets of active and completed transactions are empty.

Payment Schemas and Definitions

To complete a transaction, payment must be made from the buyer to the seller. The schemas and functions presented next detail the requirements of a payment.

Payment

$$id, amount : \mathbb{N}_1$$
$$buyer\_pm, seller\_pm : PaymentMethod$$
$$status : STATUS$$

The *Payment* schema represents the information necessary to settle transactions financially (i.e. transfer money from a buyer to a seller). A Payment has an id, an amount, a buyer and seller payment method, and a status. The payment methods contain the financial information needed for the monetary transfer. The status of the payment is used to indicate the outcome of the processing and may either be success or failure. A function is needed for the creation of a payment in the electronic commerce system.

$$create\_payment : \mathbb{N}_1 \times \mathbb{N}_1 \times PaymentMethod \times PaymentMethod \rightarrow Payment$$

$$\forall id, am : \mathbb{N};\ bpm, spm : PaymentMethod;\ p : Payment \bullet$$
$$create\_payment(id, am, bpm, spm) = p \Rightarrow$$
$$p.id = id \land p.amount = am \land p.buyer\_pm = bpm \land$$
$$p.seller\_pm = spm \land p.status = start$$

The *create_payment* function takes the input for a payment schema and creates a *Payment* by assigning the values given to the correct attributes. The ability to find a payment in the list of payments maintained by the system is important so that a current status of the payment may be reported to the system and users. The *return_payment* function describes that operation.

$$return\_payment : \mathbb{N}_1 \times \mathbb{P}\, Payment \rightarrow \mathbb{P}\, Payment$$

$$\forall i : \mathbb{N}_1;\ p, results : \mathbb{P}\, Payment \bullet return\_payment(i, p) = results \Rightarrow$$
$$results \subseteq p \land (\forall x : Payment \mid x \in results \bullet x.id = i)$$

The *return_payment* function finds a payment in a set of payments given an id. For payments to succeed, additional third parties are required for communication and the transfer of monetary amounts. Several schemas and functions are presented below to aid in this process.

PaymentAttempt

$$id : \mathbb{N}_1$$
$$buyer\_status, seller\_status : PAYMENT\_STATUS$$
$$payment : Payment$$

The *PaymentAttempt* schema contains the data needed by the processing gateway to make a payment from a buyer to seller. Each *PaymentAttempt* contains an id, the status of the payment for the buyer and seller financial institutions and payment information contained in the *Payment* schema. A function to create a new *PaymentAttempt* is presented below.

$$create\_payment\_attempt : \mathbb{N}_1 \times Payment \rightarrow PaymentAttempt$$

$$\forall id : \mathbb{N}_1;\ p : Payment;\ pa : PaymentAttempt \bullet$$
$$create\_payment\_attempt(id, p) = pa \Rightarrow pa.id = id \land pa.payment = p \land$$
$$pa.buyer\_status = begin \land pa.seller\_status = begin$$

The *create_payment_attempt* function creates a new *PaymentAttempt* by taking an id and payment information as inputs. The buyer and seller status are set to *start* to signify the beginning of the payment process for the buyer and the seller. The *return_payment_attempt* function, defined below, is used to return an existing *PaymentAttempt* from the processing gateway.

$$return\_payment\_attempt : \mathbb{N}_1 \times \mathbb{P}\, PaymentAttempt \rightarrow \mathbb{P}\, PaymentAttempt$$

$$\forall i : \mathbb{N}_1;\ pa, result : \mathbb{P}\, PaymentAttempt \bullet return\_payment\_attempt(i, pa) = result \Rightarrow$$
$$result \subseteq pa \land (\forall x : PaymentAttempt \mid x \in result \bullet x.id = i)$$

The *return_payment_attempt* function uses an id to locate a *PaymentAttempt*. The *ProcessingGateway* and associated functions are defined next.

ProcessingGateway

$$current\_payments : \mathbb{P}\, PaymentAttempt$$
$$fin\_insts : \mathbb{P}\, FinancialInstitution$$

$$\forall a, b : PaymentAttempt \bullet a \in current\_payments \land b \in current\_payments \Rightarrow$$
$$a.id \neq b.id$$
$$\forall x, y : FinancialInstitution \bullet x \in fin\_insts \land y \in fin\_insts \Rightarrow$$
$$x.name \neq y.name$$

The *ProcessingGateway* schema is a third-party resource for the processing of payments.

This schema contains sets of *PaymentAttempt* and *FinancialInstitution* schemas. The *current_payments* property represents the current attempted payments undergoing processing by the gateway and the set of *fin_insts* represents the banking resources available. Two constraints on *ProcessingGateway* are that each *PaymentAttempt* and each *FinancialInstitution* are unique. A function to initialize the *ProcessingGateway* is defined as:

InitProcGate

$$\Delta ProcessingGateway$$

$$current\_payments' = \emptyset$$
$$fin\_insts' = \emptyset$$

The *InitProcGate* operation initializes a *ProcessingGateway* by setting the initial set of current payments and the set of financial institutions to be the empty set. The next operations specify the details of the addition and removal of payment attempts and financial institutions from the *ProcessingGateway*. The *AddPayAttempt* schema is presented first.

AddPayAttempt

$$\Delta ProcessingGateway$$
$$id? : \mathbb{N}_1$$
$$p? : Payment$$

$$create\_payment\_attempt(id?, p?) \notin current\_payments$$
$$current\_payments' = current\_payments \cup \{create\_payment\_attempt(id?, p?)\}$$

The *AddPayAttempt* operational schema uses the *create_payment_function* to add a new payment attempt to the set of current payment attempts. The constraint is that the payment attempt to be created must not already exist in the set of current payment attempts.

RemovePayAttempt

$$\Delta ProcessingGateway$$
$$id? : \mathbb{N}_1$$

$$return\_payment\_attempt(id?, current\_payments) \subseteq current\_payments$$
$$current\_payments' = current\_payments \setminus$$
$$\quad return\_payment\_attempt(id?, current\_payments)$$

A payment attempt is removed from the set of current payment attempts using the *RemovePayAttempt* operational schema. Utilizing the *remove_payment_attempt* function

and the set difference operator, a payment attempt identified by a given id is removed from the set of current payment attempts. Similar operations required for financial institutions are presented next.

AddFinInst

$$\Delta ProcessingGateway$$
$$fi? : FinancialInstitution$$

$$fi? \notin fin\_insts$$
$$fin\_insts' = fin\_insts \cup \{fi?\}$$

The *AddFinInst* operation modifies the set of financial institutions by adding a financial institution given as an input. The constraint on this operation is that the financial institution to be added must not already exist in the current set. The *return_fin_inst* function is defined below.

$$return\_fin\_inst : STRING \times \mathbb{P}\, FinancialInstitution \rightarrow \mathbb{P}\, FinancialInstitution$$

$$\forall n : STRING;\ fi, result : \mathbb{P}\, FinancialInstitution \bullet return\_fin\_inst(n, fi) = result \Rightarrow$$
$$result \subseteq fi \wedge (\forall x : FinancialInstitution \mid x \in result \bullet x.name = n)$$

The *return_fin_inst* function returns a selected financial institution from a list of financial institutions given an id as an input. This function is used below by *RemoveFinInst*:

RemoveFinInst

$$\Delta ProcessingGateway$$
$$name? : STRING$$

$$return\_fin\_inst(name?, fin\_insts) \subseteq fin\_insts$$
$$fin\_insts' = fin\_insts \setminus return\_fin\_inst(name?, fin\_insts)$$

The *RemoveFinInst* operation removes a specified financial institution from a processing gateway, using the *remove_fin_inst* function. The constraint on the schema is that the financial institution to be removed must exist in the set of financial institutions.

With the specification of a payment, payment attempt, processing gateway and associated functions, a mechanism for managing payments is useful.

PaymentManager

$$current\_payments : \mathbb{P}\, Payment$$
$$proc\_gateway : ProcessingGateway$$

$$\forall x, y : Payment \bullet x \in current\_payments \land y \in current\_payments \Rightarrow x.id \neq y.id$$

The *PaymentManager* schema represents the mechanism for managing all the payments attempted by the system. The payment manager contains a sequence of current payments as well as information about the processing gateway to be used in the transfer of money between the buyer's and seller's financial institutions. The constraint on the schema is that each *Payment* in the payment manager is unique. An operational schema used to initialize the payment manager is described next.

InitPaymentManager

$$\Delta PaymentManager$$

$$current\_payments' = \emptyset$$

The *InitPaymentManager* operation initializes the payment manager by setting the set of current payments to empty. Other values of the payment manager require modification during the operation of the system.

ModifyProcessingGateway

$$\Delta PaymentManager$$
$$pg? : ProcessingGateway$$

$$proc\_gateway' = pg?$$

The *ModifyProcessingGateway* schema allows the payment manager to modify the information stored about the processing gateway by setting a new value given as an input. Another requirement of the system is to be able to add new payments to the payment manager.

To initiate payment in the system the following function is used by a *SellerPurchase* in a transaction:

$$consolidate\_payments : \mathbb{P}_1\, ProductPurchase \rightarrow \mathbb{P}_1\, Payment$$

The *consolidate_payments* function groups each of the product purchases in a seller

purchase by payment method and sums the amounts of each product purchase. The set of payments created is presented for use by the payment manager by the *RequestPayment* schema:

RequestPayment

$$\Xi SellerPurchase$$
$$p! : \mathbb{P}_1\ Payment$$

$$p! = consolidate\_payments(product\_purchases)$$

The output of the *RequestPayment* schema is a list of the current payments required for a particular seller purchase in a transaction. This operation uses the *consolidate_payments* function to create the set of payments used by the *AddPayments* operational schema defined below.

AddPayments

$$RequestPayment$$
$$\Delta PaymentManager$$
$$p? : \mathbb{P}_1\ Payment$$

$$p? = p!$$
$$\forall pp, cp : Payment \mid pp \in p? \land cp \in current\_payments \bullet pp.id \neq cp.id$$
$$current\_payments' = current\_payments \cup p?$$

The *AddPayments* schema adds a set of payments, created in the *RequestPayment* schema, to the current payments stored by the payment manager. The constraint on the schema is that each payment in the set to be added must be unique when compared to each existing payment in the *PaymentManager*.

RemovePayment

$$\Delta PaymentManager$$
$$pid? : \mathbb{N}_1$$

$$return\_payment(pid?, current\_payments) \subseteq current\_payments$$
$$current\_payments' = current\_payments \setminus return\_payment(pid?, current\_payments)$$

The *RemovePayment* schema is used to remove a payment from the set of current payments using the *return_payment* function and the set difference operator. The pre-condition on the schema is that the payment to be removed must exist in the current set

of payments.

With the preceding definitions of the required building blocks, a series of schemas and functions are now presented to demonstrate the transfer of funds between financial institutions through the processing gateway.

$$auth\_fin\_inst : STRING \times CODE \times FinancialInstitution \rightarrow PAYMENT\_STATUS$$

$$\begin{aligned}
&\forall n : STRING;\ a_c : CODE;\ fi : FinancialInstitution \bullet \\
&\quad (auth_fin_inst(n, a_c, fi) = auth \Rightarrow fi.name = n \wedge fi.auth_code = a_c) \vee \\
&\quad (auth_fin_inst(n, a_c, fi) = end \Rightarrow fi.name \neq n \vee fi.auth_code \neq a_c)
\end{aligned}$$

The *auth_fin_inst* function uses a name and authorization code and compares them against the name and authorization code of an existing financial institution. If the values match, the financial institution authorizes the credit or debit to take place and returns a payment status of *auth*. If either of the values do not match, a payment status of *error* is returned. The schema below uses the *auth_fin_inst* function for initial processing of a payment attempt.

AuthorizePayment

$$\begin{aligned}
&\Delta PaymentAttempt \\
&bfi?, sfi? : FinancialInstitution \\
&\hline
&buyer_status = begin \\
&seller_status = begin \\
&buyer_status' = auth_fin_inst(payment.buyer_pm.finame, \\
&\quad payment.buyer_pm.auth_code, bfi?) \\
&seller_status' = auth_fin_inst(payment.seller_pm.finame, \\
&\quad payment.seller_pm.auth_code, sfi?)
\end{aligned}$$

The *AuthorizePayment* operational schema attempts to gain authorization from the buyer's and seller's financial institutions to modify account balances contained within each. The buyer and seller *PAYMENT_STATUS* are modified by the *auth_fin_inst* function. The pre-conditions on *AuthorizePayment* are that buyer and seller status must both be *begin*. The function defined next is used to debit a financial institution account.

$$debit\_fin\_inst\_acc : \mathbb{N}_1 \times \mathbb{N}_1 \times FinancialInstitution \rightarrow PAYMENT\_STATUS$$

$$\begin{aligned}
&\forall\, a_n, am : \mathbb{N}_1;\ fi : FinancialInstitution \bullet \\
&\quad (debit_fin_inst_acc(a_n, am, fi) = pay \Rightarrow \\
&\qquad (\exists_1\, a : Account \mid a \in fi.accounts \bullet a.acc_num = a_n \land \\
&\qquad a.balance = debit_account(a, am))) \lor \\
&\quad (debit_fin_inst_acc(a_n, am, fi) = end \Rightarrow \\
&\qquad (\forall\, a : Account \mid a \in fi.accounts \bullet a.acc_num \neq a_n))
\end{aligned}$$

The *debit_fin_inst_acc* function uses an account number and an amount to debit a financial institution account using the *debit_account* function. The result of this function is a modification to the payment status of a payment. If the account number exists, the amount is debited from the account and a payment status of *pay* is returned. If the account is not found then a payment status of *end* is returned. Similarly, *credit_fin_inst_acc* is used to credit a financial institution account:

$$credit\_fin\_inst\_acc : \mathbb{N}_1 \times \mathbb{N}_1 \times FinancialInstitution \rightarrow PAYMENT\_STATUS$$

$$\begin{aligned}
&\forall\, a_n, am : \mathbb{N}_1;\ fi : FinancialInstitution \bullet \\
&\quad (credit_fin_inst_acc(a_n, am, fi) = pay \Rightarrow \\
&\qquad (\exists_1\, a : Account \mid a \in fi.accounts \bullet a.acc_num = a_n \land \\
&\qquad a.balance = credit_account(a, am))) \lor \\
&\quad (credit_fin_inst_acc(a_n, am, fi) = end \Rightarrow \\
&\qquad (\forall\, a : Account \mid a \in fi.accounts \bullet a.acc_num \neq a_n))
\end{aligned}$$

The *credit_fin_inst_acc* function uses the *credit_account* function to credit the account of a financial institution supplied a parameter. If the account number provided exists in the financial institution then a payment status of *pay* is returned. If the account number does not exist, a payment status of *end* is returned. For debiting a buyer account, the *BuyerPayment* schema is presented next.

BuyerPayment

$$\begin{aligned}
&\Delta PaymentAttempt \\
&bfi? : FinancialInstitution \\
&\hline
&buyer_status = auth \\
&seller_status = auth \\
&buyer_status' = debit_fin_inst_acc(payment.buyer_pm.acc_num, \\
&\quad payment.amount, bfi?)
\end{aligned}$$

The *BuyerPayment* operational schema debits a buyer's account at their financial insti-

tution by using the *debit_fin_inst_acc* function. The account number from the buyer payment method and the amount from the payment in the payment attempt are used as input parameters along with the buyer financial institution. The pre-conditions on this schema are that the buyer and seller status must both be *auth*, meaning that the debit of the buyer may not occur until the financial institutions of the buyer and seller involved have both provided authorization for the transfer of funds. The second part of the monetary transfer, *SellerPayment*, is defined below.

SellerPayment

$$\Delta PaymentAttempt$$
$$sfi? : FinancialInstitution$$

$$buyer\_status = pay$$
$$seller\_status = auth$$
$$seller\_status' = credit\_fin\_inst\_acc(payment.seller\_pm.acc\_num,$$
$$\quad payment.amount, sfi?)$$

The *SellerPayment* operation uses the *credit_fin_inst_acc* function to credit the seller's account at their financial institution account with the amount specified in the payment amount. The pre-conditions on the schema are that the buyer status must be *pay* and the seller status must be *auth*. This ensures that the buyer's account has been successfully debited and the seller's financial institution has authorized the transfer. After the completion of the payment attempt, the outcome is presented back to the transaction.

$$return\_pay\_att\_status : PAYMENT\_STATUS \times PAYMENT\_STATUS \rightarrow STATUS$$

$$\forall bs, ss : PAYMENT\_STATUS \bullet$$
$$\quad (return\_pay\_att\_status(bs, ss) = complete \Rightarrow bs = pay \land ss = pay) \lor$$
$$\quad (return\_pay\_att\_status(bs, ss) = error \Rightarrow bs \neq pay \lor ss \neq pay)$$

The *return_pay_att_status* function returns the status of a payment using the results of the buyer and seller payment status from the payment attempt. The function sets the status to *complete* if the buyer and seller status were both *pay*. The value *error* the result of the function if either the buyer or seller status was not *pay*. This function is used in the schema below.

SetPaymentStatus

$\Delta PaymentAttempt$
$status! : STATUS$

$payment'.status = return\_pay\_att\_status(buyer\_status, seller\_status)$
$status! = payment'.status$

The *SetPaymentStatus* modifies the payment attempt and sets an output value equal to the status of the payment using the *return_pay_att_status*. This value is then set in the transaction payment by the following schema:

ReceivePaymentStatus

$\Delta Payment$
$status? : STATUS$

$status' = status?$

The *ReceivePaymentStatus* schema sets the status of a payment to a value passed as an input. This operation is used to record the status of a payment that was attempted by the processing gateway.

With the definition of all aspects of payments in the electronic commerce system, an integrating schema for payments is presented below.

IntegratedPaymentManager

$PaymentManager$
$AddPayments$
$RemovePayment$
$AuthorizePayment$
$BuyerPayment$
$SellerPayment$
$ModifyProcessingGateway$

The *IntegratedPaymentManager* schema groups all aspects of payment in a transaction into an integrated schema. This integration allows for the use of the separate schemas by the electronic commerce system together.

Electronic Commerce System

Based on the components specificed, a representation of the entire electronic commerce system is now presented.

ElectronicCommerceSystem

$$
\begin{array}{l}
buyers : \mathbb{P}\, Buyer \\
sellers : \mathbb{P}\, Seller \\
buyer\_session\_handler : BuyerSessionHandler \\
seller\_session\_handler : SellerSessionHandler \\
transaction\_manager : TransactionManager \\
search\_engine : SearchEngine \\
comparison\_engine : ComparisonEngine \\
master\_inventory : MasterInventory \\
payment\_manager : IntegratedPaymentManager \\
\hline
\forall x, y : Buyer \bullet x \in buyers \land y \in buyers \Rightarrow x.id \neq y.id \\
\forall u, v : Seller \bullet u \in sellers \land v \in sellers \Rightarrow u.id \neq v.id
\end{array}
$$

The *ElectronicCommerceSystem* schema contains all the components in the entire system. The schema contains a set buyers and a set sellers that have registered with the system. The buyer and seller session handlers are used to track when a buyer or seller is active in the electronic commerce system. The transaction manager controls all the transactions, where a buyer wishes to purchase some goods, in the electronic commerce system. The transaction manager works in concert with the payment manager to complete these transactions. The payment manager is responsible for transferring the money involved in transactions. The search engine and comparison engine are tools used by the buyers and sellers to find, compare, and contrast products from the master inventory in the electronic commerce system. The master inventory contains all of the products available for purchase from the sellers. There are two pre-conditions on this schema describing the uniqueness of each buyer and seller in the electronic commerce system. The next operational schema initializes the system.

InitElectronicCommerceSystem

$$
\begin{array}{l}
\Delta ElectronicCommerceSystem \\
\hline
buyers' = \emptyset \\
sellers' = \emptyset
\end{array}
$$

The *InitElectronicCommerceSystem* schema initializes the entire electronic commerce system by setting the initial set of buyers and sellers to be empty. The information used for the operation of the electronic commerce system is needed after initialization. The operational schema that modifies the attributes of the system is presented below.

LoadElectronicCommerceSystem

$\Delta ElectronicCommerceSystem$
$b? : \mathbb{P}\, Buyer$
$s? : \mathbb{P}\, Seller$
$bsh? : BuyerSessionHandler$
$ssh? : SellerSessionHandler$
$tm? : TransactionManager$
$se? : SearchEngine$
$ce? : ComparisonEngine$
$mi? : MasterInventory$
$pm? : IntegratedPaymentManager$

$buyers' = b?$
$sellers' = s?$
$buyer\_session\_handler' = bsh?$
$seller\_session\_handler' = ssh?$
$transaction\_manager' = tm?$
$search\_engine' = se?$
$comparison\_engine' = ce?$
$master\_inventory' = mi?$
$payment\_manager' = pm?$

The *LoadElectronicCommerceSystem* schema is needed to supply the operational information needed by the electronic commerce system. This schema represents the setup of the electronic commerce system for use. After the successful completion of this schema, the system is ready to receive buyers and sellers and to carry out searches, comparisons, transactions and payments.

3.3.2 Z Specification Verification

The specifications were checked for correctness using the Z/EVES tool [34]. The checks performed include syntax, type, and domain checking. Z/EVES reads a specification from a file and parses the Z symbols to build schemas, axiomatic definitions, and rules.

Syntax and Type Checking

In Z/EVES the only mandatory checking of a Z specification is syntax and type checking [34]. Syntax checking ensures that the specification is written correctly and contains no errors which prevent proper parsing. Without correct syntax, as in any other programming, errors will either appear in the result or terminate the parsing altogether. Examples of improper syntax in a Z specification would be leaving certain elements out of a pre-formed statement, incorrect spelling of names, and erroneous characters. Type checking ensures that the specification uses the proper types at all times. This applies to all rules, schemas, and functions defined in a specification. An example of an incorrect type is to assign a string value to a numeric field.

In the Z/EVES program, syntax and type checking are accomplished during the "read" command. Figure 3.25 shows a syntax and type checking session of the specifications in this thesis using the Z/EVES tool. An input file can read and a user can scroll through the results to find errors syntax and type errors.

Domain Checking

Domain checking of a Z specification can be accomplished in the Z/EVES program by reading in a specification, identifying proofs that can be tested, and writing a proof script to demonstrate those proofs. Domain checking is done to ensure that all expressions written are meaningful. Figure 3.26 demonstrates all of the possible domain checks that can be accomplished based on the Z specifications presented in this chapter.

Figures 3.27 and 3.28 demonstrate the use of domain checking in the Z specification. Figure 3.27 shows the domain proof for the *credit_account* definition. The Z/EVES commands *try lemma* and *prove by reduce* are used together to produce a proof result of *true* for all domains of *credit_account*.

By using the domain checks provided by the Z/EVES tool, all the possible proofs available can be reduced as shown in Figure 3.28. The results demonstrated are similar to Figure 3.26. However, the Z/EVES tool now lists which possible domain checks have been proven and which remain. The entire list of proofs can be submitted to the domain checking process

Figure 3.25: Z-Eves Syntax and Type Check of Z Specification

Figure 3.26: Z-Eves Domain Check of Z Specification

Figure 3.27: Z-Eves Domain Check Proof Example

Figure 3.28: Z-Eves Domain Check Results

to ensure their correctness. This process requires the creation of several instantiations of variables and environments. This is a substantial amount of work for a specification of this size and scope. The domain checking of the entire specification in this thesis is a subject for future work.

Chapter 4

E-Commerce System Prototype

This chapter presents an partial implementation of an electronic commerce system based on the model presented in Chapter 3. The design methodology, tools used, and the design environment are discussed. The chapter concludes with a description of the application and shows some sample screen shots of the application.

4.1 Application Design

The prototype was designed using the UML diagrams and the Z specifications presented in Chapter 3. The UML class diagrams were used to create the database tables and relationships and the sequence diagrams aided in establishing program flow between components. The Z specification was used to define the rules for each object in the system.

The tools and platform used to create the prototype include Microsoft SQL Server 2000, Microsoft Visual Studio .NET, and Microsoft Internet Information Server installed on a personal computer running Windows 2000 Professional. Microsoft SQL Server 2000 was used for the storage of the database constructs needed by the application and was selected for its ease of installation and use with Microsoft Visual Studio .NET and Microsoft Internet Information Server(IIS). The application was created in Visual C# in Microsoft Visual Studio .NET using aspx pages to display data access components on web pages using Microsoft's *code behind* [6] methodology. Microsoft Internet Information Server was used in

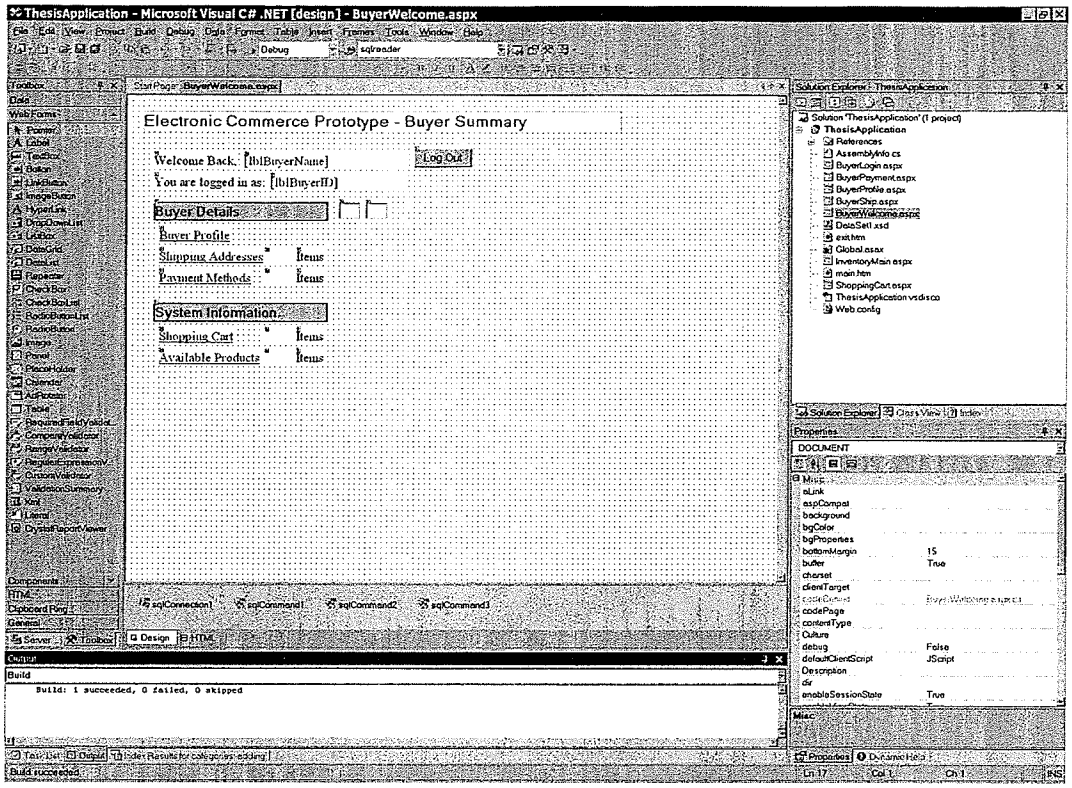

Figure 4.1: Development using Microsoft Visual Studio .NET

conjunction with the Microsoft .NET Framework to serve the web pages onto the Internet and to provide a gateway to the information stored in the database for the application. This combination of tools allowed for the quick creation and deployment of web-based applications on a single machine connected to the Internet. Figure 4.1 contains a screen shot of development using Microsoft Visual Studio .NET.

4.2 Implementation

Several key areas of operation of the electronic commerce system have been selected for demonstration using screen shots from the application together with textual explanations.

4.2.1 Welcome Screen

Figure 4.2 shows the welcome screen of the electronic commerce system. The navigation options on this screen include Buyer Log In, Seller Log In, and Search Inventory. The buyer

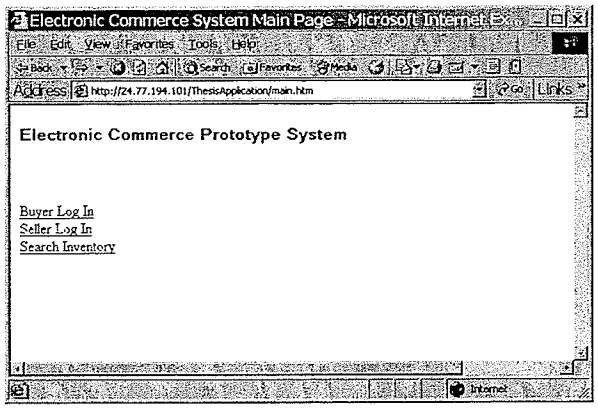

Figure 4.2: Welcome Screen

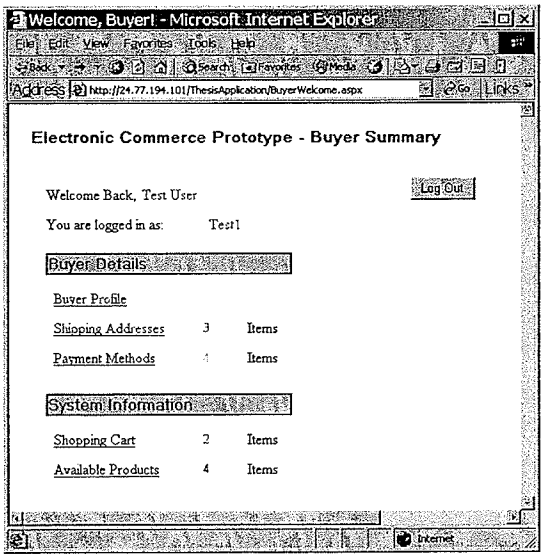

Figure 4.3: Buyer Summary Screen

and seller log in screens, which are invoked by clicking on the links, allow an existing buyer or seller entry to the system. Additionally, new buyer and seller accounts can be created by following these links. The third link brings up the search interface. This link allows for anonymous access to the product information stored in the system and allows potential buyers to browse the goods offered without the inconvenience of creating a buyer's account.

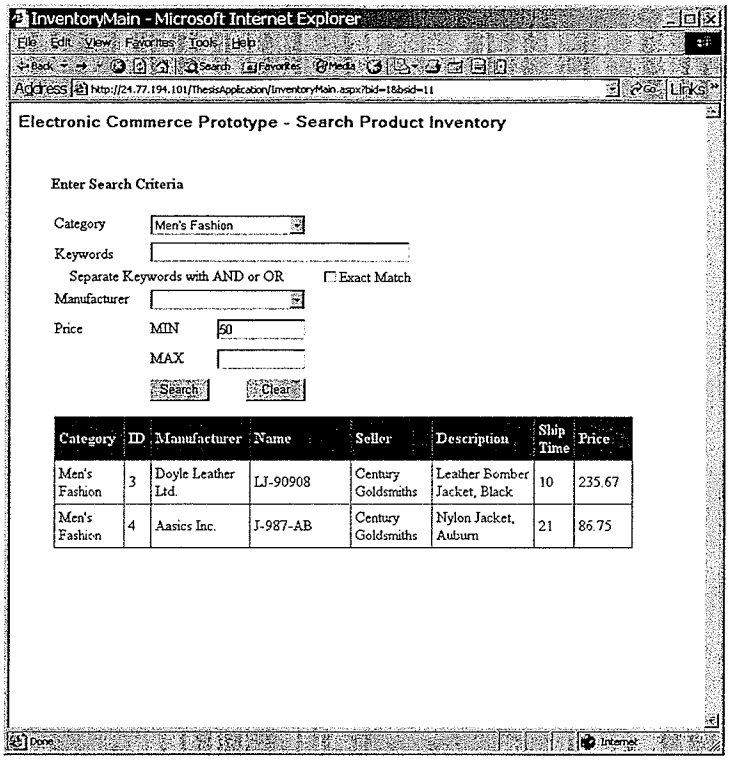

Figure 4.4: Searching Product Inventory

4.2.2 Buyer Summary Screen

Figure 4.3 shows the summary screen for a buyer. After a successful login, the system presents to the buyer a summary of his/her current status in the system. The page contains links presenting detailed information about, and functions for, the buyer. The page displays the buyer's name and login id along with links to the shipping addresses, payment methods, products for purchase in his/her shopping cart, and the products in the inventory. These links load other pages with additional information and allow the buyer to modify personal data, manage items in the shopping cart, and search the product inventory.

4.2.3 Browsing for Goods

Figure 4.4 shows a snapshot of the product inventory search interface. The user can enter values for several different criteria including product category, manufacturer, a minimum and maximum price, and any number of keywords. The drop down list on the form is

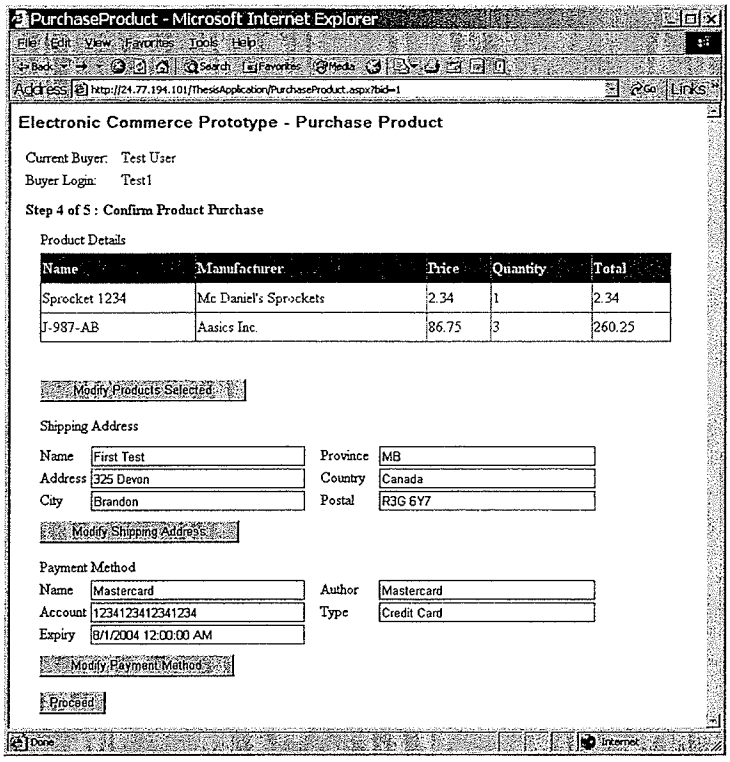

Figure 4.5: Check Out with Products

dynamic, limiting selections for manufacturer based on the category selected. The keywords entered must be separated by AND or OR. This search criteria is then parsed by the system and a full text scan of the products is executed using these values. The two buttons below the criteria fields execute the query or clear the fields back to empty. Upon execution of the search, the results are displayed, in tabular form, directly underneath the criteria entry area. Each product displayed may be selected and viewed in detail by selecting the product id and comparisons between products may be made by the comparison button.

4.2.4 Check Out with Products

Figure 4.5 shows the final screen for a product purchase in the electronic commerce system. Each item to be purchased is presented along with a quantity, shipping address and payment method. The quantity is the quantity of the item the buyer wishes to purchase, the shipping address is the destination to which the goods should be shipped, and the payment method is the payment type for that product as selected by the buyer. Once the Proceed button

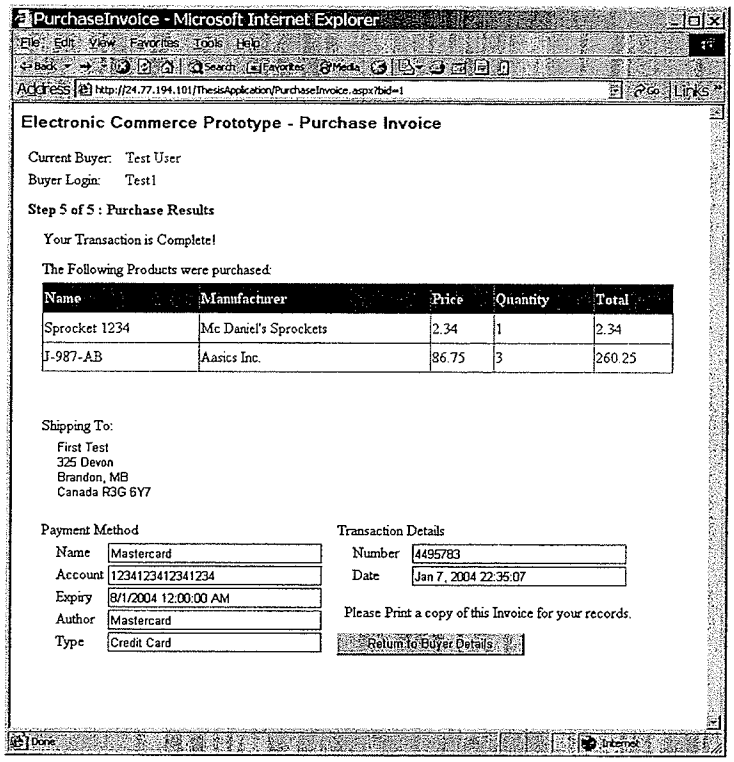

Figure 4.6: Receipt for Completed Purchase

is clicked, the system processes all the data presented and contacts the processing gateway to begin payment for the items. After processing is complete, a report of the status will be presented to the buyer as shown in Figure 4.6.

4.2.5 Receipt for Completed Purchase

After processing payments for goods in a buyer's shopping cart, the system displays a final screen to the buyer. Figure 4.6 shows the results of purchases and provides details on shipping times, addresses, and amounts charged to the different buyer payment methods. This information is presented with a unique number to allow for order tracking in the system at a later date. All successfully completed payments result in orders sent to the sellers involved to ship the goods to the addresses designated by the buyer. A buyer can now return to normal use of the system, including searching for more products, viewing buyer information, or logging out.

Chapter 5

Conclusions and Future Work

This chapter concludes the thesis by recapping what has been presented in previous chapters, presenting a summary of the contributions made by this thesis, and giving an outline of possible future work in this area.

5.1 Conclusions

This thesis presented a thorough discussion of the use of formal methods in the electronic commerce domain. In Chapter 1, the benefits of electronic commerce, problems that had to be considered, and an introduction to formal methods was given. Chapter 2 began with an outline of the design issues in electronic commerce, using many different methodologies and examples of previously created systems. Additionally, a Web-based architecture for an electronic commerce system was presented. The chapter concluded with a discussion of the Unified Modeling Language(UML) and the Z specification language. Chapter 3 presented an original model of an electronic commerce system. The model was then described using UML Class, Use Case, Sequence, State, Activity and Deployment diagrams. Using the UML and textual descriptions of the electronic commerce system, a Z specification based on mathematical relations and textual explanations was presented. Chapter 3 closed with the use of the Z/EVES tool to show the syntax and type checking of the Z specification and to show domains available for testing and how they can be tested. Finally, Chapter 4

contained a discussion of a prototype system created from the model presented in Chapter 3 and explained the tools and architecture used to create the prototype. Chapter 4 also provided some screen shots of the implemented system.

5.2 Summary of Contributions

This thesis has made the following contributions:

- The thesis formalized the main requirements of an electronic commerce system.
- The thesis identified and integrated the elements required for successful operation of an electronic commerce system, including databases, security, networks, distributed systems, and artificial intelligence.
- The thesis provided a case study for modeling electronic commerce using the Unified Modeling Language and the Z specification language.
- The thesis provided a partial prototype of an electronic commerce system.

5.3 Future Work

This thesis provides opportunity for further research in the following areas:

- Alternate forms of electronic commerce.

  In addition to business-to-consumer electronic commerce, as presented in this thesis, business-to-business and consumer-to-consumer methods of electronic commerce models could be examined. This is important as the requirements and operations of different types of electronic commerce could be compared and contrasted. Common components and type-specific problems could be identified. This could be accomplished by the application of formal methods to other forms of electronic commerce systems design, enabling a full description of the problem domains and the ability to compare them with the requirements presented in this thesis.

- Further examination of electronic commerce transactions and payments made over the Internet.

  A key component of any electronic commerce system is the ability to correctly carry out transactions for goods or services and transfer monetary amounts between buyers and sellers. A further analysis of the communication methods and information exchange between stakeholders in this process might provide benefits in several areas. By clearly defining the rules of a transaction and the required components and communications, a checklist of steps that must be performed, and points of failure could be identified. This also applies to payments between clients in the system as these transfers must be correct, precise, and secure. Further work in this area might include the use of formal methods to specify each component of transactions, the further identification of an order of operations in transactions and payments, and the creation of a common application and network interface for implementing electronic commerce transactions.

- Security implications of electronic commerce.

  Security is essential in an electronic commerce system where sensitive information about clients, products, and financial matters are communicated over the Internet. The security of financial information is of utmost importance as this information, if left unsecured, can be used without the owner's permission or knowledge. Formal methods can be used to identify the areas of an electronic commerce system that require secure access and can assist in the application of a security methodology to an electronic commerce implementation. The locations and strengths of security needed can be identified and incorporated into a specification and design.

- Enhanced intelligent searching and comparison methods.

  The ability to find a product or supplier on the Internet assists the shopping experience of a consumer. Additionally, the ability to compare characteristics of products for purchase, aids the consumer in finding the goods that best suit the customers needs at the best price possible. With enhanced searching and comparison capabilities, a

broader marketplace is opened to the consumer and more information can be gathered to support product purchases. This leads to a higher satisfaction for the consumer with purchases made over the Internet. Future work in this area includes examination of agent-based search and comparison programs that use artificial intelligence and distributed computing and further research in the field of data mining. Agents can be used to automate searches and comparisons across electronic commerce sites as bots and also aid in automating negotiation for prices and quantities.

- Further application of object-oriented software engineering concepts to electronic commerce.

  As suggested in this thesis with the Unified Modeling Language, object-oriented concepts can be applied to the electronic commerce domain. Existing electronic commerce systems vary widely in approach and implementation with custom modules and differing communication methods. Concepts such as encapsulation, inheritance, and polymorphism used in electronic commerce can result in common code libraries, methods of communication, and a common architecture. Results would see an ease of communication between electronic commerce systems and shorter implementation time for new entries into the marketplace.

- Analysis of data storage methods used in electronic commerce.

  Data storage and manipulation is an important part of an electronic commerce system. Between electronic commerce systems, data structures, methods of presentation, querying, and database transactions can vary. Information storage and use within an electronic commerce system may differ and speed and accuracy are important. Future work in this area might include the use of formal methods to identify common components and storage methods, optimization strategies for data storage including tools such as XML and specially tuned database servers for querying, and a data architecture for electronic commerce transactions. Many different application architectures are already using UML to present structured data on the Internet that a framework for electronic commerce could be developed. Additionally, future work might include

research and development of "wrappers" to allow heterogeneous data sources to be connected.

Appendix A

The Z Notation

This section lists the meaning of the Z notations used in this thesis. This list of the Z notation was taken from [4] and [28].

Z Paragraphs, Declarations

[X]	given set
$S \mathrel{\widehat=} T$	horizontal schema definition
$X == e$	abbreviation definition
$T ::= A \mid B\langle\langle E \rangle\rangle$	free type definition

Expressions, Schema Expressions

(a, b)	tuple
{a, b}	set display
$X \times Y$	cross product
let $V == E \bullet P$	local definition
ΔS	schema name prefix
ΞS	schema name prefix
$S \fatsemi T$	sequential composition

Numbers and Finiteness

$\mathbb{N}$	natural numbers
$\mathbb{N}_1$	positive integers
$\mathbb{Z}$	integers
$\mathbb{F}$	finite set
$\mathbb{F}_1$	non-empty finite set
$\#$	number of members of a finite set
div	division

Predicates

$=$	equality
$\in$	membership
$\wedge$	conjuction
$\vee$	disjunction
$\Rightarrow$	implication
$\Leftrightarrow$	equivalence
$\forall$	universal quantification
$\exists$	existential quantification
$\exists_1$	unique quantification

Relations, Functions

$\leftrightarrow$	relation
$\mapsto$	maplet (ordered pair)
dom	domain
ran	range
$\lhd$	domain restriction
$\rhd$	range restriction

$\lhd$	domain anti-restriction
$\rhd$	range anti-restriction
$\sim$	relational inversion
$(\!\vert\!)$	relational image
$\oplus$	overriding
*	reflexive transitive closure
$\pfun$	partial function
$\fun$	total function

Sets

$\neq$	inequality
$\notin$	non-membership
$\emptyset$	empty set
$\subseteq$	subset
$\subset$	proper subset
$\cup$	set union
$\cap$	set intersection
$\setminus$	set difference
$\bigcup$	generalized union

Sequences

seq	finite sequence
seq_1	non-empty finite sequence
head	first element
last	last element
front	all but the last element
disjoint	disjointness

References

[1] Bassam Aoun. Agent Technology in Electronic Commerce and Information Retrieval on the Internet. In *AUSWEB96 - The Second Australian World Wide Web Conference*, pages 240–246, Gold Coast, Australia, July 1996.

[2] Martin Bichler, Arie Segev, and J. Leon Zhao. Component-based E-Commerce: Assesment of Current Practices and Future Directions. *SIGMOD Record*, 27(4):7–14, 1998.

[3] Susanne Boll, Wolfgang Klas, and Bernard Battaglin. Design and Implementation of RMP - A Virtual Electronic Market Place. *SIGMOD Record*, 27(4):48–53, 1998.

[4] ORA Canada. *Z/Eves Quick Reference Card.* Available at: ftp://ftp.ora.on.ca/pub /doc/975493-07.ps.Z.

[5] World Wide Web Consortium. *World Wide Web Consortium.* Available at: http://www.w3.org/.

[6] Microsoft Corporation. *ASP.NET Code-Behind Model Overview.* Available at: http://support.microsoft.com/default.aspx?scid=kb;EN-US;303247.

[7] Asuman Dogac, Ilker Durusoy, Sena Nural Arpinar, Nesime Tatbul, Pinar Koskal, Ibrahim Cingil, and Nazife Dimililer. A Workflow-based Electronic Marketplace on the Web. *SIGMOD Record*, 27(4):25–31, 1998.

[8] eBay Inc. *eBay Auctions.* Available at: http://www.ebay.com.

[9] Sylvanus A. Ehikioya. *Specification of Transaction Systems Protocols.* PhD thesis, Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada, September 1997.

[10] Sylvanus A. Ehikioya. A Formal Perspective to Modelling Electronic Commerce Transactions. *Columbian Journal of Computation*, 2(2):21–40, 2000.

[11] Sylvanus A. Ehikioya. A Formal Characterization of Electronic Commerce Transactions. *International Journal of Computer and Information Sciences*, 2(3):97–117, 2001.

[12] Sylvanus A. Ehikioya and Ken E. Barker. Towards a Formal Specification Methodology for Transaction Systems Protocols. In *3rd Annual IASTED International Conference on Software Engineering and Applications (SEA '99)*, pages 374–380, Scottsdale, Arizone, USA, October 1999.

[13] Sylvanus A. Ehikioya and Kristofer J. Hiebert. A Formal Model of Electronic Commerce. In *First International Conference on Software Engineering, Networking and Parallel and Distributed Computing (SNPD-00)*, pages 400–409, Champagne-Ardenne, France, May 2000.

[14] Sylvanus A. Ehikioya and Kristofer J. Hiebert. A Formal Specification of an On-line Transaction. In *First International Conference on Software Engineering, Networking and Parallel and Distributed Computing (SNPD-00)*, pages 3–10, Champagne-Ardenne, France, May 2000.

[15] Sylvanus A. Ehikioya and Kristofer J. Hiebert. Agents Negotiation in Electronic Commerce Transactions. In *First Annual International Conference on Computer and Information Science (ICIS-01)*, pages 278–285, The Grosvenor Resort, Orlando, Florida, U.S.A., October 3-5 2001.

[16] Sylvanus A. Ehikioya and Trevor Walowetz. A Formal Specification of Transaction Systems in Distributed Multi-Agents Systems. In *ISCA 14th International Conference on Computers and their Applications*, pages 378–383, Cancun, Mexico, April 1999.

[17] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, 1998.

[18] Centre for Software Engineering Ltd. Unified Modelling Language. *Technical Briefing Note*, (8), 2000. Available at: ftp://ftp.cse.dcu.ie/pub/briefing/8_1uml.pdf.

[19] National Center for Supercomputing Applications. *National Center for Supercomputing Applications.* Available at: http://hoohoo.ncsa.uiuc.edu/cgi/intro.html.

[20] Object Management Group. *The Unified Modelling Language, Ver 1.5, 2000.* Available at: http://www.omg.org/uml.

[21] Robert H. Guttman and Pattie Maes. Agent-Mediated Integrative Negotiation for Retail Electronic Commerce. *Lecture Notes in Computer Science*, 1571:70–90, 1999.

[22] Robert H. Guttman, Pattie Maes, Anthony Chavez, and Daniel Dreilinger. Results from a Multi-Agent Electronic Marketplace Experiment. In *Poster Proceedings of Modeling Autonomous Agents in a Multi-Agent World (MAAMAW'97)*, Ronneby, Sweden, May 1997.

[23] Robert H. Guttman, Alexandros G. Moukas, and Pattie Maes. Agent-mediated Electronic Commerce: A Survey. *Knowledge Engineering Review*, 13(2):143–152, June 1998.

[24] June He. A Fromal Specification and Design of an Online Bazaar System. Master's thesis, University of Manitoba, Winnipeg, Manitoba, Canada, September 2002.

[25] Macromedia Inc. *Macromedia ColdFusion MX.* Available at: http:// www.macromedia.com/software /coldfusion/.

[26] PayPal Inc. *PayPal.* http://www.paypal.com.

[27] VeriSign Inc. *VeriSign.* Available at: http://www.verisign.com.

[28] Indratmo. A Formal Specification of Web-Based Data Warehouses. Master's thesis, University of Manitoba, Winnipeg, Manitoba, Canada, 2001.

[29] Nicholas R. Jennings, Timothy J. Norman, and Peyman Faratin. ADEPT: An Agent-Based Approach to Business Process Management. *SIGMOD Record*, 27(4):32–39, 1998.

[30] Sparx Systems Pty Ltd. *Enterprise Architect 3.50.* Available at: http://www.sparxsystems.com.au/.

[31] Pattie Maes, Robert H. Guttman, and Alexandros G. Moukas. Agents That Buy and Sell. *Communications of the ACM*, 42(3):81, 1999.

[32] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition.* Prentice-Hall, Upper Saddle River, NJ, 1999.

[33] Benny Reich and Israel Ben-Shaul. A Componentized Architecture for Dynamic Electronic Markets. *SIGMOD Record*, 27(4):40–47, 1998.

[34] Mark Saaltink. *The Z/EVES User's Guide*, 1997. Available at: http://www.ora.on.ca/z-eves/documentation.html.

[35] J. M. Spivey. *The Z Notation: A Reference Manual (2nd Edition).* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[36] Roel Wieringa. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Computing Surveys*, 30(4):459–527, 1998.

[37] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement and Proof.* Prentice-Hall, 1996.