A Formal Perspective to Modelling Electronic Commerce Transactions

Sylvanus A. Ehikioya *

Abstract

Many models of e-commerce applications use informal approaches and human intuition although their underlying data model is sound. Consequently, most of the designs contain inconsistencies and some hidden brittle spots that do not manifest until the applications are used. In addition, informal design approaches leave the determination of the correctness of system designs to the intuition of the designers. Informal design approaches do not provide the degree of correctness and reliability required for e-commerce transactions.

This paper formalizes requirements for electronic transactions using Z. The specification describes constraints relating to the uniqueness of customers, accounts, products, and stores, to the availability of products, to the validity of customers, and to maintaining sufficient funds. The formal specification has been mechanically checked using Z/EVES and is therefore well-formed in terms of syntax and types.

Keywords: *Formal methods, e-commerce, Z.*

1 Introduction

With the advancement in telecommunication technology and sophistication in software development, many activities relating to sales and purchase processes, such as production, storage, administration, advertisement, ordering, delivery, accounting and billing, payment, and even quality control, and customer care and post-sales support, have been moved to the digital domain. Thus, having a robust, reliable, and efficient e-commerce system that is accessible from any corner of the world will greatly enhance global competitiveness.

This paper puts forward an important idea — the formal specification of the initial design of the behaviour of an e-commerce system. It is likely that the volume of e-commerce will increase and transactions must be processed by highly reliable software. Formal specification is an important technique for increasing reliability. Consequently, examples of formalization for e-commerce should be of great interest. Formal specification methods are scarely used in industrial projects despite their advantages and the maturity of theories and tools [4]. Most of the e-commerce software applications developed and deployed are based on *ad hoc*, intuitive, and informal development approaches. Thus, ambiguity, inconsistencies, unproven artifacts and sometimes incorrect executions characterise the requirements and design specifications. This problem often permeates to the final products that deviate from the initial requirements specifications and manifest unexpected failures during actual use. These problems arise because informal methods do not explicitly capture the behaviour of

*Department of Computer Science, University of Manitoba, Winnipeg, MB, Canada, R3T 2N2, ehikioya@cs.umanitoba.ca

e-commerce transactions, which is critical to e-commerce systems' development, especially for the design phase. The dynamic characteristics of e-commerce transactions should be formalized in terms of the static structural descriptions in order to specify and verify the system behaviour using rigourous techniques.

Sometimes the mere process of formalizaton is useful because formalization forces designers to think through every aspect of a system carefully. Even if the formal specification is not actually used, the design may be better because of the formalization activity. Adopting a formal approach can guarantee that the specifications are unambiguous, consistent, and correct, all of which are assured via proof mandate. Formal methods provide support for rigorous analysis capabilities and hence provide a basis for rigorous software development. Formal methods are key to improving software quality.

This paper presents a well-defined formal model for both the static and dynamic behaviours of e-commerce system and their integration. The formal specification is written in a well-known specification language, Z [28, 29], and checked mechanically for syntax and types conformance, consistency, and correctness using the Z-EVES [32] tool. This paper is useful because it helps systems developers to design and implement an e-commerce system that forms an integral part of an organisation's information technology system using a formal specification methodology. In particular, this paper focuses on the key transactional processing requirements for e-commerce applications and reduces the complexities involved in e-commerce transaction processing so that programmers can easily construct actual systems from the specification. This paper is significant for a number of reasons:

- It provides a robust, reliable and correctly verified design model of e-commerce application. In particular, we explore how formal modelling can be used to expand and complement the capabilities of traditional software development. The paper discusses the technical details of the design of back-end transactional processing and core services of e-commerce applications using standard formal notation.
- The design we present reduces the complexity of transaction processing issues in the business-to-business and business-to-consumer e-commerce domains.
- This paper provides a practical environment. The basic principles of our design are applicable to solving real-world problems while also providing a stimulating learning experience to other system developers.
- By documenting the artifacts of the system, the relevant elements in e-commerce application are clearer.
- This paper highlights the characteristics of e-commerce systems suitable for this approach.

The rest of this paper is organized as follows: Section 2 sets the context for the specification presented in this paper, while Section 3 summarizes the current research directions in e-commerce. Section 4 assesses the use and advantages of our approach, while Section 5 enumerates the attributes of e-commerce applications. Section 6 describes the specification in Z. In Section 7, we present a sample use case scenario for the model. We conclude the paper in Section 8.

2 Problem Domain

The system discussed in this paper will be used for viewing various products a company has to offer and ordering online. A user will be able to create a user account that will store the information needed to make purchases. The system will have a virtual shopping cart that the user can add items to and remove items from. When done, the user can purchase the items in the cart. The system will have search facilities so that users can more readily find the items they wish or show items of a certain type.

The system will also be used for administration purposes. The company can view or print sales reports, get client information for mailing lists, view their inventory and make any necessary changes to the information. The system can also show the number of specific products currently in stock so the administrator can purchase more stock if needed.

This type of system requires a secure environment to provide the necessary e-commerce functionality because the client and the Ordering System exchange sensitive information (such as credit card numbers and other personal data). Therefore, there must be a mechanism in place to prevent a client's information from falling into the wrong hands. There is also the need to prevent a malicious individual from impersonating clients and thereby charging them for products they did not order. To prevent security breaches, we recommend using various measures such as user authentication, verification and validation, and encryption. In addition, to prevent impersonation, a client receives a notification message after making an order. Sending a message serves dual purposes, confirming that the client made the order, and providing the client with details about the order. In this paper, we omit the specification of these security aspects in order to focus on the transactional processing aspects of e-commerce. This security component will be the subject of another discussion.

3 Related Work

The activities involved in e-commerce transactions cut across many disciplines, such as law, marketing, networking, and databases to mention a few. Consequently, various aspects of e-commerce have been the focus of research activities recently [11, 16, 27]. Each e-commerce model or prototype focuses on one or more specific aspects of the e-commerce issues identified in [9]. We will not discuss the details of these in this paper but a brief outline of the research directions will be appropriate.

Research efforts that focus on development of search agents specifically for e-commerce [2, 3, 20, 21] dominate the literature. The capability of these search agents is basic and require extensive vertical modification to cater to modern e-commerce search requirements and metaphors. Besides search agents, Hallam-Barker [18, 19] proposes e-commerce payment protocols, while Group of Ten [17] discusses the legal issues in electronic payments. In addition, O'Mahony, *et al* [23] provide a comprehensive guide that identifies various methods of transferring money over the Internet (including CyberCash, Secure Electronic Transactions (SET), and Ecash), and provides important insights into how each system works. Other areas associated with e-commerce transactions include Web tunneling and security [1] and the mechanisms to secure the accounting process of metering Web sites [22].

Tygar [31] describes the application of classical atomicity properties to e-commerce transactions. Ehikioya [9] provides a model of multi-agent system for distributed transactions, one of the core operations performed by any e-commerce application, which supports concurrent execution and provides automatic communication between the agents.

Formal modelling of e-commerce transactions is beginning to gain a foothold. For example, Ehikioya and Barker [12] provide mechanisms for handling the internals of e-commerce transactions at the transaction management level using an event-based and causally aware framework to describe the communication / interaction patterns between elements of the system. In addition, formal modelling of e-commerce transactions is available in [10, 14, 15]. The work reported in this paper is different from those in [10, 14, 15] although they all address e-commerce transactions. While Ehikioya and Hiebert [14] use the Unified Modeling Language (UML) to model e-commerce transactions, in [15] they use Z notation as the modelling formalism for transactions in a specific domain, the automotive market. The model in this paper is scalable and generic, thereby permitting customization of the system.

In parallel to the above, research efforts in the traditional areas of data modelling, systems architecture, and business process improvements in the context of e-commerce applications are continuing. For example, Bustard, *et al* [6] and Standing [30] provide cutting edge approaches for developing systems models that bridge the traditional gap between information systems and software engineering and let a developer effectively manage change and outcomes. They discussed the underlying theory and practical techniques associated with data modeling and design. Similarly, Rajput [24] identifies the major building blocks of an e-commerce system and provides insights into understanding how these building blocks interact to form an effective e-commerce system. These components provide the framework for the complete understanding a developer needs to build e-commerce architecture for the various types of commercial transactions — business-to-business, business-to-consumer, and intra-business. Also, Reynolds [26] provides a pragmatic development approach to building e-commerce applications.

Although e-commerce systems developers can significantly gain from the approaches outlined in these sources [6, 24, 26, 30], the treatment of the approaches is rather informal and leaves the determination of the correctness of system designs to the intuition of the designers. This situation is not ideal for e-commerce transactions because of the high premium on correctness and reliability features. Therefore, using a formally-based technique can assist in alleviating this problem.

4 Applicability of our Approach

Although several e-commerce systems models have been identified in the literature [14], the specification presented in this paper is suitable for the business-to-business and business-to-consumer domains. E-commerce applications provide various back-end transactional processing and information access services across many heterogeneous databases and different networks. These functionalities require complex interaction relationships in order to fulfill a discrete e-commerce objective. While the number of users in the business-to-business domain is much smaller than the business-to-consumer domain (e.g., the business-to-consumer domain may involve several million users whereas business-to-business may involve just a few hundred or thousand users), both require a highly reliable system that always guarantees consistent and correct results. The critical tasks of the business processes that e-commerce systems fulfill reinforce the need for the reliability requirement. In addition, e-commerce systems cohesively integrate retailers, suppliers, financial institutions, and buyers in order to attain the values of engaging in the initiatives of e-commerce. Therefore, the designer must design software reliability into the system.

The modelling of complex systems requires techniques that allow us to manage complexity and allow early detection of errors in behaviour models. Z adequately supports the principle of separation of views that is an effective means of controlling complexity. In addition, Z supports the formality and rigour required for detecting errors in requirements and design early. Developing a precise, complete, and understandable specification for e-commerce systems that enables practical, tool-supported and rigorous analysis of e-commerce models can enhance their usefulness. In addition, the insights provided by a well-defined e-commerce systems specification can help implementers choose appropriately among a variety of implementation alternatives. These observations are particularly relevant when we consider the inherent dynamic behaviour of e-commerce systems.

Our approach offers the capability to model dynamic behaviour in e-commerce systems. For example, we model dynamic behaviour by using invariants for active classes or for the whole model and pre- and post-conditions on operations of active classes. It is clearly essential that a close correspondence exist between the implementation and the design.

E-commerce provides a vital function in a business's overall information technology architecture and provides functionality that transcends departments, locations, and international boundaries. Fixing an error after a system's deployment is expensive and the error may manifest in many different parts of the system. To contain costs and the range of errors, errors should be detected and corrected early in the development process. Thus, appropriate rigorous methods for modelling both the static and dynamic aspects of e-commerce systems are necessary. The Z notation provides the desired rigour and the capability to model complex systems. This methodology, which is verifiable, produces consistent and unambiguous specification, and guarantees the correctness and reliability of transactional processing of e-commerce transactions.

The benefits of formally specifying e-commerce transactions using the approach discussed in this paper include (but are not limited to) the following:

- A formal specification provides a clear understanding of the system. It provides a solid method to model the properties of the system by using set theory and mathematical statements. In addition, it also provides the required precision.
- The formalization process can reveal ambiguities, incompleteness, and contradictions in the informal definition.
- A formal specification permits the correctness verification of the transactions thereby enhancing their reliability.
- A formal specification provides abstractions which precisely specify the behaviour of a system by concentrating on its functionality in order to manage complexity and promote correctness, extensibility, maintainability, reusability, and understanding.

Notice that some of these benefits are directly derivable from the benefits of using formal methods, which is well documented in the literature [8, 13].

5 Characteristics of E-commerce Applications

The attributes of e-commerce applications amenable to this approach are:

- Interoperability with legacy systems and infrastructure — E-commerce solutions may require access to legacy systems for tight integration with organisational resources. Such solutions must provide the desired interoperability with legacy systems. For example, many businesses already have implementations of their core business operations in non-web based environments. Therefore, to leverage past investments on information technology, the new web-enabled systems must be integrated and inter-operate with these legacy systems.
- Wide breadth of functionality — E-commerce solutions must provide a wide breadth of functionality to be successful. These functionality should cover a broad range of areas in the stages of e-commerce activity, from product location, negotiation, contract execution, product delivery to post-sales customer care, including all the back-end transactional processing requirements.

- Complex interactions of multiple applications — E-commerce solutions involve the complex interactions of many applications internal and external to the organisation, so a holistic view of the requirements is, therefore, necessary. Thus, all the requirements necessary to materialize the overall e-commerce system functionality must be included in the analysis and formally specified in the requirements capture phase. The dependencies among the applications introduce a new dimension into the e-commerce solutions environment.

- Integration of diverse technology components — In all of its forms, e-commerce uses technologies from different areas such as databases, transaction processing, distributed systems, intelligent agents, multimedia systems, security and electronic payment infrastructures, workflow systems, and varying software environments. These diverse components must be tightly integrated in order to provide the necessary environment for e-commerce transactions. For example, there must be applications to handle payments, provide content, pre- and after-sales customer care, transactional processing functionality, and address the inherent security issues of the application environment.

- Involvement of multiple operational units — A complete e-commerce solution requires services and data that may cut across multiple units, internal and external to the e-commerce system.

- Availability and correctness requirement — Availability requirement ensures e-commerce services are available to customers and other users when required while correctness guarantees the correct processing and output accuracy of requested services. These two essential elements affect quality of service guarantees. To maximize return on investment on the e-commerce solutions and satisfy customer-oriented value chains, organisational efficiencies, and revenue generation benefits, e-commerce systems must be available and produce correct and consistent results always.

Besides these core features of e-commerce applications, additional characteristics of e-commerce systems are available in [24]. The characterisation in [24] generally applies to other web-based information systems.

The new rules and peculiar requirements introduced by the e-commerce paradigm mean that e-commerce systems cannot rely on *ad hoc* or intuitive design methodologies but require a well-defined approach which is amenable to mathematical analysis so that correctness, consistency and reliability can be ascertained. In addition, the need for acceptable performance, security, scalability, and operational robustness and completeness provides compelling and logical arguments for building e-commerce solutions based on formal initiatives. Thus, any design approach for building e-commerce solutions should be methodical to ensure conformity to these requirements and their appropriate capture, to provide for manageability of the complexities, and to provide the desired return on investment in the e-commerce solutions.

6 The Specification

For lack of space, we omit a formal discussion of the Z notation in this paper. Readers unfamiliar with Z should see [28, 29, 32]. However, we annotate the specification, with clear and understandable commentary to assist the reader.

First we define the free types applicable in the design.

$[CHAR, EMAIL, EMAIL\_ADDRESS, FAX]$

For simplicity, we assume that all money values are in whole dollars. Consequently, we define the type *MONEY* as:

$MONEY == \mathbb{N}_1$

Of course, in real life, monetary values are of base type real. During implementation, the real data type should be used. Our choice of value for *MONEY* in this specification is a conscious decision in order to avoid the complexities of defining the real type in Z (since Z has no real base type) and focus on the core elements and services of an e-commerce system.

$BOOLEAN ::= True \mid False$
$SEARCH\_REPLY ::= NotFound \mid Found$
$SHIPPING\_MODE ::= Normal \mid NextDay \mid TwoBusinessDays \mid Express \mid GroundShipExp$
$PAY\_MODE ::= Cheque \mid Credit\_Card \mid Debit\_Card \mid Approved\_Credit$
$ACCOUNT\_TYPE ::= Savings \mid Current$
$DIGIT == 0 \mathrel{..} 9$
$SECONDS == \mathbb{N}$
$ORDER\_STATUS ::= Delivered \mid Shipping \mid Processing \mid Cancelled$
$FUND\_CHECK\_REPLY ::= Success \mid InsufficientFunds$

$PHONE\_NUMBER == \mathrm{seq}\, DIGIT$

$\forall p : PHONE\_NUMBER \bullet \#p \geq 7$

The operation type indicates the action the client wishes to perform.

$OPERATION ::= NewOrder \mid CancelOrder \mid ViewOrder \mid ViewOrderList$

The type *INVENTORY_INV_REPLY* represents the result of the inventory verification operation. For instance, if the quantity of an item in the inventory is insufficient to fill an order, the result "*Unavailable*" is returned.

$INVENTORY\_INV\_REPLY ::= Available \mid Unavailable$

The type *INVENTORY_REG_REPLY* represents the status of a client's transaction that is attempting to update the inventory. Note that to avoid conflicts, no two transactions may modify the inventory at the same time. Only one transaction is allowed to execute, while the others wait. Thus the task of updating the inventory is an atomic and blocking process.

$INVENTORY\_REQ\_REPLY ::= Waiting \mid Executing$

Next, we describe the structure of applicable objects in the system and state any constraint that applies to instances of the objects.

A bank account can be either a current account or a savings account. If a customer pays by cheque, the decrement account must be a current account. If a customer pays via debit card, the type of account does not matter.

$BankAccount \mathrel{\widehat=} [number : \mathbb{N}_1;\ balance : MONEY;\ type : ACCOUNT\_TYPE]$

All accounts must be unique, so the constraint below must hold at all times.

$\forall a, b : BankAccount \mid a \neq b \bullet a.number \neq b.number$

A *CreditCard* describes the credit card object, another form of payment acceptable to the system.

$$CreditCard \mathrel{\widehat=} [number : \mathbb{N}_1;\ balance, limit : MONEY \mid balance \leq limit]$$

Similarly, all credit cards are unique.

$$\forall\, a, b : CreditCard \mid a \neq b \bullet a.number \neq b.number$$

Each product is part of a particular category. This categorization allows products to be searched or displayed based on their category, rather than in one bulky set.

$$\begin{aligned} Product \mathrel{\widehat=} [&productid : \mathbb{N}_1;\ name, category, description : \operatorname{seq} CHAR;\\ &suppliercost, ourcost : MONEY \mid suppliercost > 0 \land \#name > 0 \land\\ &\quad ourcost > suppliercost \land \#category > 0 \land \#description > 0\,] \end{aligned}$$

Every product in the inventory is unique. Thus,

$$\forall\, a, b : Product \mid a \neq b \bullet a.productid \neq b.productid$$

A supplier provides products that are bought by the company and then sold to clients. For the purpose of making orders, contact information is recorded for each supplier. It is assumed that some other module (not specified here) is responsible for placing orders with suppliers. The information stored in the Ordering System would of course be used in making new orders (with suppliers) based on current inventory levels, buying trends, etc. This stored information enables the entire [business] system to automate most of its activities and takes advantage of available technology and thus make it e-business capable.

$$\begin{aligned} Supplier \mathrel{\widehat=} [&name : \operatorname{seq} CHAR;\ products : \mathbb{P}\, Product;\ phone, fax : PHONE\_NUMBER;\\ &email : EMAIL\_ADDRESS \mid \#name > 0 \land products \neq \emptyset] \end{aligned}$$

We capture the uniqueness property of suppliers by the following constraint definition.

$$\forall\, a, b : Supplier \mid a \neq b \bullet a.name \neq b.name$$

Notice that in real business world, names of suppliers might not be unique. In such a situation, a unique identity number is assigned to each supplier. In this specification, we assume the name attribute of a supplier is unique. Similar comments apply to the objects, Bank and CreditCompany, defined below.

When a client pays for an order using a cheque or a debit card, the funds are drawn from the client's bank account. We simply abstract a bank by the schema *Bank*. Here we ignore the other components and functions of a bank since they are not vital to this specification. Interested readers should refer to [5, 6] for the details of formal specification of banking transactions.

$$Bank \mathrel{\widehat=} [name : \operatorname{seq} CHAR;\ accounts : \mathbb{P}\, BankAccount \mid accounts \neq \emptyset \land \#name > 0]$$

Every bank is unique, so

$$\forall\, a, b : Bank \mid a \neq b \bullet a.name \neq b.name$$

A credit company provides credit cards to clients. Credit cards can be used to pay for orders made using the Ordering System. The Ordering System must communicate with a credit company to determine if a particular client has sufficient credit available to pay for the order. We assume the credit company provides an electronic means for the Ordering System to request client information, without human intervention.

$$CreditCompany \mathrel{\widehat{=}} [name : \mathrm{seq}\, CHAR;\ cards : \mathbb{P}\, CreditCard \mid cards \neq \emptyset \land \# name > 0]$$

Similarly, credit granting companies, for example, Visa, Master Card, Discover, and American Express, are unique. Thus,

$$\forall\, a, b : CreditCompany \mid a \neq b \bullet a.name \neq b.name$$

A client is an individual or organization that purchases products from the company. Clients interact with the system by submitting orders using any of the supported interfaces. All clients must have a client ID number to be able to place orders, which means they must register before using the Ordering System. A client must provide either credit card or bank account information, depending on the default mode of payment the client specifies. The default mode of payment is used when a client makes an order and does not specify a different mode of payment for the order.

$$\begin{aligned} Client \mathrel{\widehat{=}} [&clientid : \mathbb{N}_1;\ address, name : \mathrm{seq}\, CHAR;\ fax, phone : PHONE\_NUMBER;\\ &email : EMAIL\_ADDRESS;\ defaultpaymode : PAY\_MODE;\\ &account : BankAccount;\ creditcard : CreditCard \mid \# name > 0 \land \# address > 0]\end{aligned}$$

Every client is distinct, so

$$\forall\, a, b : Client \mid a \neq b \bullet a.clientid \neq b.clientid$$

The company that owns the Ordering System provides approved credit, *ApprovedCredit*, to its clients. Only clients who request approved credit will be provided with it.

$$ApprovedCredit \mathrel{\widehat{=}} [clientid : \mathbb{N}_1;\ balance, limit : MONEY \mid balance \leq limit]$$

Similarly, every *ApprovedCredit* is unique. Notice that this constraint is derived from the uniqueness of clients, since approved credits are granted to only registered clients.

$$\forall\, a, b : ApprovedCredit \mid a \neq b \bullet a.clientid \neq b.clientid$$

The shopping cart is an interface entity. It keeps track of all of the items a client wishes to purchase. When the client finally places an order, the list of items in the shopping cart is added to the order. Due to the nature of the web or phone interface, clients may abandon operations without explicitly indicating to the system that they have done so. Therefore, the shopping cart has a time limit associated with it. If the shopping cart is not utilized for an extended period of time, it is assumed that the operation was abandoned and the cart is emptied.

$$ShoppingCart \mathrel{\widehat{=}} [client : Client;\ items : \mathbb{P}(Product \times \mathbb{N}_1);\ timelimit : SECONDS]$$

The email system is the source of email orders. Emails are taken from the incoming list in first-in first-out (FIFO) order and processed to produce an order. Confirmation messages may be created by the Ordering System and sent to the client by placing them in the outgoing list.

$$\begin{aligned} EmailSystem \mathrel{\widehat{=}} [&incoming, outgoing : \mathrm{iseq}\, EMAIL \mid\\ &\quad (\forall\, a : EMAIL \mid \langle a \rangle \ \mathsf{in}\ incoming \bullet \neg\, (\langle a \rangle \ \mathsf{in}\ outgoing)) \land\\ &\qquad (\forall\, b : EMAIL \mid \langle b \rangle \ \mathsf{in}\ outgoing \bullet \neg\, (\langle b \rangle \ \mathsf{in}\ incoming))\ ]\end{aligned}$$

The fax system operates in a manner identical to the email system, but utilizes faxes.

$$\begin{aligned} \mathit{FaxSystem} \mathrel{\widehat{=}} [&\mathit{incoming}, \mathit{outgoing} : \operatorname{iseq} \mathit{FAX} \mid \\ &(\forall a : \mathit{FAX} \mid \langle a \rangle \mathsf{\ in\ } \mathit{incoming} \bullet \neg (\langle a \rangle \mathsf{\ in\ } \mathit{outgoing})) \land \\ &(\forall b : \mathit{FAX} \mid \langle b \rangle \mathsf{\ in\ } \mathit{outgoing} \bullet \neg (\langle b \rangle \mathsf{\ in\ } \mathit{incoming}))] \end{aligned}$$

Each order is given a distinct order ID number for record-keeping purposes. The field *destaddress* indicates the location to which the order is to be delivered.

$$\begin{aligned} \mathit{Order} \mathrel{\widehat{=}} [&\mathit{orderid} : \mathbb{N}_1;\ \mathit{client} : \mathit{Client};\ \mathit{items} : \mathbb{P}(\mathit{Product} \times \mathbb{N}_1);\ \mathit{paymode} : \mathit{PAY\_MODE}; \\ &\mathit{shipping}, \mathit{subtotal}, \mathit{tax}, \mathit{total} : \mathit{MONEY};\ \mathit{shipmode} : \mathit{SHIPPING\_MODE}; \\ &\mathit{status} : \mathit{ORDER\_STATUS};\ \mathit{destaddress} : \operatorname{seq} \mathit{CHAR} \mid \#\mathit{destaddress} > 0 \land \\ &\mathit{items} \neq \emptyset \land \mathit{total} = \mathit{subtotal} + \mathit{tax} + \mathit{shipping}] \end{aligned}$$

Every order is different, even if the same client makes them. For example, there could be variations in the date, time, ordered items, or destination address in orders by the same client. Thus, the *orderid* provides a mechanism to uniquely identify an order. The following constraint must hold at all times.

$$\forall a, b : \mathit{Order} \mid a \neq b \bullet a.\mathit{orderid} \neq b.\mathit{orderid}$$

A client's transaction represents an operation being performed by a single client. Since multiple clients may use the Ordering System at the same time, there can be multiple clients' transactions in existence at one time.

$$\mathit{ClientTransaction} \mathrel{\widehat{=}} [\mathit{client} : \mathit{Client};\ \mathit{order} : \mathit{Order};\ \mathit{action} : \mathit{OPERATION}]$$

It is vital to associate a *ClientTransaction* and the corresponding *Order* with the correct client. Thus, the following constraint must hold.

$$\forall o : \mathit{Order};\ t : \mathit{ClientTransaction} \mid t.\mathit{order} = o \bullet t.\mathit{client} = o.\mathit{client}$$

The inventory contains a list of all of the products the company sells and the quantities that are available. The hold list contains items that have been ordered, but for which payment has not yet been received. Only one transaction may modify an inventory item at one time. The transaction that is currently allowed to modify the inventory is placed in the active field. All other transactions that wish to access the inventory are placed in the waiting list until they are able to execute. The *updateInProgress* flag indicates whether or not inventory levels are being updated at the warehouse. When this flag is set to *True*, no transaction may modify the inventory item.

$$\begin{aligned} \mathit{Inventory} \mathrel{\widehat{=}} [&\mathit{items} : \mathbb{P}(\mathit{Product} \times \mathbb{N});\ \mathit{holdlist} : \mathbb{P}(\mathit{Product} \times \mathbb{N}_1); \\ &\mathit{waitinglist} : \operatorname{iseq} \mathit{ClientTransaction};\ \mathit{active} : \mathbb{P}\, \mathit{ClientTransaction}; \\ &\mathit{updateinprogress} : \mathit{BOOLEAN} \mid \#\mathit{active} \leq 1 \land (\forall c : \mathit{ClientTransaction} \mid \\ &c \in \mathit{active} \bullet \neg (\langle c \rangle \mathsf{\ in\ } \mathit{waitinglist})) \land (\forall p : \mathit{Product} \bullet p \in \operatorname{dom} \mathit{holdlist} \Rightarrow \\ &p \in \operatorname{dom} \mathit{items})] \end{aligned}$$

The *AccountPayable* is a single record representing an account due to a supplier from the company.

$$\mathit{AccountPayable} \mathrel{\widehat{=}} [\mathit{balance}, \mathit{payment} : \mathit{MONEY};\ \mathit{supplier} : \mathit{Supplier} \mid \mathit{payment} > 0]$$

Similarly, *AccountReceivable* is a single record of an account due to the company from a client.

$$AccountReceivable \mathrel{\widehat=} [balance, payment : MONEY;\ client : Client \mid payment > 0]$$

The next two operations, *MakePayment* and *ReceivePayment*, are used to make payment to suppliers and receive payment from clients, respectively. The *balance* and the amount paid or received, *payment*, are recorded in the appropriate account record type.

$$\begin{aligned} MakePayment \mathrel{\widehat=} [&\Delta AccountPayable;\ amount? : MONEY;\ supID? : \mathrm{seq}\, CHAR \mid \\ &amount? \geq 0 \land balance \geq 0 \land supID? = supplier.name \land \\ &payment' = amount? \land balance' = balance - amount?] \end{aligned}$$

$$\begin{aligned} ReceivePayment \mathrel{\widehat=} [&\Delta AccountReceivable;\ amount? : MONEY;\ clientID? : \mathbb{N}_1 \mid \\ &amount? \geq 0 \land balance \geq 0 \land clientID? = client.clientid \land \\ &payment' = amount? \land balance' = balance - amount?] \end{aligned}$$

The order database contains all of the information that is needed by the Ordering System. The variable *acceptedbanks* is a set of all banks that the company is able to deal with to receive payment from a client. Likewise, the *acceptedcards* field lists all credit cards acceptable to the company.

$$\begin{aligned} OrderDatabase \mathrel{\widehat=} [\ &inventory : Inventory;\ clients : \mathbb{P}\, Client;\ acceptedbanks : \mathbb{P}\, Bank; \\ &approvedcredit : \mathbb{P}\, ApprovedCredit;\ accountspayable : \mathbb{P}\, AccountPayable; \\ &accountsreceivable : \mathbb{P}\, AccountReceivable;\ suppliers : \mathbb{P}\, Supplier; \\ &shippingrates : \mathbb{P}(SHIPPING\_MODE \times MONEY);\ orders : \mathbb{P}\, Order; \\ &acceptedcards : \mathbb{P}\, CreditCompany\ \mid \\ &clients \neq \emptyset \land suppliers \neq \emptyset \land inventory.items \neq \emptyset \land \\ &(\forall\, o : Order \bullet o \in orders \Rightarrow (\exists\, c : Client \bullet c \in clients \land c = o.client)) \land \\ &(\forall\, s : Supplier \mid s \in suppliers \bullet s.products \subseteq \mathrm{dom}\, inventory.items) \land \\ &(\forall\, a : ApprovedCredit \bullet a \in approvedcredit \Rightarrow \\ &\qquad (\exists\, c : Client \bullet c \in clients \land c.clientid = a.clientid)) \land \\ &(\forall\, a : AccountPayable;\ s : Supplier \bullet \\ &\qquad a \in AccountPayable \land s = a.supplier \Rightarrow s \in suppliers) \land \\ &(\forall\, b : AccountReceivable;\ c : Client \bullet \\ &\qquad b \in AccountReceivable \land c = b.client \Rightarrow c \in clients) \land \\ &(\forall\, s, t : Supplier \mid (s \neq t \land s \in suppliers \land t \in suppliers) \bullet \\ &\qquad s.products \cap t.products = \emptyset)\ ] \end{aligned}$$

The Ordering System represents the application which processes orders from clients. Transactions is a list of all of the client transactions currently executing in the system.

$$\begin{aligned} OrderingSystem \mathrel{\widehat=} [\ &transactions : \mathbb{P}\, ClientTransaction;\ carts : \mathbb{P}\, ShoppingCart; \\ &db : OrderDatabase;\ emailsystem : EmailSystem;\ faxsystem : FaxSystem] \end{aligned}$$

The following three operations are performed by an anonymous user. Any user of the Ordering System can perform them without specifying a client ID number, or even being a registered user. The operation *FindProduct* searches the inventory to see if a product with a particular name exists.

$$\begin{aligned} FindProduct \mathrel{\widehat=} [\ &\Xi OrderingSystem;\ name? : \mathrm{seq}\, CHAR;\ result! : SEARCH\_REPLY \mid \\ &\quad \mathbf{if}\ (\exists\, p : Product \bullet p \in \mathrm{dom}\, db.inventory.items \land p.name = name?) \\ &\qquad \mathbf{then}\ (result! = Found)\ \mathbf{else}\ (result! = NotFound)] \end{aligned}$$

The next schema definition displays the detailed description of a product to the user.

$$\begin{aligned}&\mathit{ViewProductDescription} \mathrel{\widehat=} [\, \Xi \mathit{OrderingSystem};\ \mathit{name}? : \mathrm{seq}\, \mathit{CHAR};\\ &\quad \mathit{description}! : \mathrm{seq}\, \mathit{CHAR} \mid \#\mathit{name}? > 0 \land\\ &\quad (\exists\, p : \mathit{Product} \mid p \in \mathrm{dom}\, \mathit{db.inventory.items} \land\\ &\qquad p.\mathit{name} = \mathit{name}? \bullet \mathit{description}! = p.\mathit{description})]\end{aligned}$$

The next operation creates a list of products in a category. The list can be viewed by the user.

$$\begin{aligned}&\mathit{ViewProductList} \mathrel{\widehat=} [\, \Xi \mathit{OrderingSystem};\ \mathit{category}? : \mathrm{seq}\, \mathit{CHAR};\ \mathit{list}! : \mathbb{P}\, \mathit{Product} \mid\\ &\quad \#\mathit{category}? > 0 \land (\forall\, p : \mathit{Product} \mid p \in \mathrm{dom}\, \mathit{db.inventory.items} \land\\ &\qquad p.\mathit{category} = \mathit{category}? \bullet \mathit{list}! = \mathit{list}! \cup \{p\})]\end{aligned}$$

The following four operations (*GetCart*, *AddToCart*, *RemoveFromCart*, and *ViewCart*) can only be performed by a registered client. *GetCart* provides clients with new empty shopping carts which they can use to keep track of the items they want to buy. *AddToCart* adds a particular item to a client's shopping cart.

$$\begin{aligned}&\mathit{GetCart} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{clientid}? : \mathbb{N}_1 \mid \mathit{carts}' = \mathit{carts} \cup\\ &\quad \{s : \mathit{ShoppingCart} \mid (\exists_1\, c : \mathit{Client} \mid c \in \mathit{db.clients} \land\\ &\qquad c.\mathit{clientid} = \mathit{clientid}? \bullet (s.\mathit{client} = c \land s.\mathit{items} = \emptyset))\}]\end{aligned}$$

$$\begin{aligned}&\mathit{AddToCart} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{clientid}?, \mathit{productid}?, \mathit{quantity}? : \mathbb{N}_1 \mid\\ &\quad (\exists\, c : \mathit{ShoppingCart};\ p : \mathit{Product} \mid c \in \mathit{carts} \land c.\mathit{client}.\mathit{clientid} = \mathit{clientid}? \land\\ &\quad p \in \mathrm{dom}\, \mathit{db.inventory.items} \land p.\mathit{productid} = \mathit{productid}? \bullet\\ &\qquad (c.\mathit{items}' = c.\mathit{items} \cup \{(p, \mathit{quantity}?)\} \lor (\exists\, x : \mathbb{N}_1 \mid (p, x) \in c.\mathit{items} \bullet\\ &\qquad\quad c.\mathit{items}' = c.\mathit{items} \setminus \{(p, x)\} \cup \{(p, x + \mathit{quantity}?)\})))]\end{aligned}$$

The *AddToCart* operation should be implemented as a real-time operation so that it checks the inventory database to ensure that the quantity required, *quantity*?, is available to satisfy the order. This will greatly enhance the utility and functionality of the system. The *RemoveFromCart* operation removes an unwanted item from the client's shopping cart.

$$\begin{aligned}&\mathit{RemoveFromCart} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{clientid}?, \mathit{productid}?, \mathit{quantity}? : \mathbb{N}_1 \mid\\ &\quad (\exists\, c : \mathit{ShoppingCart};\ p : \mathit{Product};\ x : \mathbb{N}_1 \mid c \in \mathit{carts} \land (p, x) \in c.\mathit{items} \land\\ &\quad c.\mathit{client}.\mathit{clientid} = \mathit{clientid}? \land p.\mathit{productid} = \mathit{productid}? \bullet\\ &\qquad c.\mathit{items}' = c.\mathit{items} \setminus \{(p, x)\} \cup \{(p, x - \mathit{quantity}?)\})\,]\end{aligned}$$

Clients can use the *ViewCart0* operation to view the items they have placed in their shopping carts.

$$\begin{aligned}&\mathit{ViewCart0} \mathrel{\widehat=} [\, \Xi \mathit{OrderingSystem};\ \mathit{clientid}? : \mathbb{N}_1;\ \mathit{list}! : \mathbb{P}(\mathit{Product} \times \mathbb{N}_1) \mid\\ &\quad (\exists\, c : \mathit{ShoppingCart};\ p : \mathit{Product};\ x : \mathbb{N}_1 \mid c \in \mathit{carts} \land\\ &\qquad c.\mathit{client}.\mathit{clientid} = \mathit{clientid}? \land (p, x) \in c.\mathit{items} \bullet \mathit{list}! = \mathit{list}! \cup \{(p, x)\})]\end{aligned}$$

During view mode, a client may modify the cart. This ability to modify cart's content is achieved using the following definition.

$$\mathit{ViewCart} \mathrel{\widehat=} \mathit{ViewCart0} \land \mathit{RemoveFromCart} \land \mathit{AddToCart}$$

The operations *AddToCart*, *RemoveFromCart*, and *ViewCart* apply to orders that have not yet been submitted to the system for processing.

The operation *InitSystem* initializes the Ordering System so that it is ready to process orders from clients.

$$\begin{aligned}&\mathit{InitSystem} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem} \mid \mathit{transactions}' = \emptyset \wedge \mathit{carts}' = \emptyset \wedge \\ &\quad \mathit{emailsystem}'.\mathit{outgoing} = \langle\rangle \wedge \mathit{faxsystem}'.\mathit{outgoing} = \langle\rangle \,]\end{aligned}$$

The operation *CreateOrder* takes the contents of a client's shopping cart and composes a new order from it.

$$\begin{aligned}&\mathit{CreateOrder} \mathrel{\widehat=} [\, \Xi \mathit{OrderingSystem};\ \mathit{clientid}? : \mathbb{N}_1;\ \mathit{paymode}? : \mathit{PAY\_MODE};\\ &\quad \mathit{shipmode}? : \mathit{SHIPPING\_MODE};\ \mathit{destaddress}? : \operatorname{seq} \mathit{CHAR};\ \mathit{order}! : \mathit{Order} \mid\\ &\quad\quad (\exists\, c : \mathit{ShoppingCart} \mid c \in \mathit{carts} \wedge c.\mathit{client}.\mathit{clientid} = \mathit{clientid}? \bullet\\ &\quad\quad\quad \mathit{order}!.\mathit{items} = c.\mathit{items} \wedge \mathit{order}!.\mathit{client} = c.\mathit{client} \wedge\\ &\quad\quad\quad\quad \mathit{order}!.\mathit{paymode} = \mathit{paymode}? \wedge \mathit{order}!.\mathit{shipmode} = \mathit{shipmode}? \wedge\\ &\quad\quad\quad\quad\quad \mathit{order}!.\mathit{destaddress} = \mathit{destaddress}? \wedge \mathit{order}!.\mathit{status} = \mathit{Processing}) \,]\end{aligned}$$

A new transaction starts when a client wishes to create a new order, or cancel a previously submitted order. The new transaction is added to the list of transactions currently active in the system.

$$\begin{aligned}&\mathit{StartTransaction} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{order}? : \mathit{Order} \mid (\exists\, t : \mathit{ClientTransaction} \mid\\ &\quad t.\mathit{order} = \mathit{order}? \wedge t.\mathit{action} = \mathit{NewOrder} \wedge t.\mathit{client} = \mathit{order}?.\mathit{client} \bullet\\ &\quad\quad \mathit{transactions}' = \mathit{transactions} \cup \{t\}) \,]\end{aligned}$$

We define the signature of two functions, *parseFax* and *parseEmail*, which are used to parse a fax message or an email message to create an order, respectively.

$$\mathit{parseFax} : \mathit{FAX} \rightarrow \mathit{Order}$$

$$\mathit{parseEmail} : \mathit{EMAIL} \rightarrow \mathit{Order}$$

The operation *GetFaxOrder* retrieves a fax from the fax system if any is available, and parses it to make a new order. Similarly, *GetEmailOrder* retrieves an e-mail from the email system if any is available, and parses it to make a new order.

$$\begin{aligned}&\mathit{GetFaxOrder} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{order}! : \mathit{Order} \mid \mathit{faxsystem}.\mathit{incoming} \neq \langle\rangle \wedge\\ &\quad \mathit{order}! = \mathit{parseFax}(\mathit{head}\, \mathit{faxsystem}.\mathit{incoming}) \wedge\\ &\quad \mathit{faxsystem}'.\mathit{incoming} = \mathit{tail}\, \mathit{faxsystem}.\mathit{incoming} \,]\end{aligned}$$

$$\begin{aligned}&\mathit{GetEmailOrder} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{order}! : \mathit{Order} \mid \mathit{emailsystem}.\mathit{incoming} \neq \langle\rangle \wedge\\ &\quad \mathit{order}! = \mathit{parseEmail}(\mathit{head}\ \mathit{emailsystem}.\mathit{incoming}) \wedge\\ &\quad \mathit{emailsystem}'.\mathit{incoming} = \mathit{tail}\ \mathit{emailsystem}.\mathit{incoming} \,]\end{aligned}$$

The next two operations control access to the inventory. The *RequestInventoryAccess* operation is performed when a new transaction wishes to access the inventory. If the inventory is available, the transaction is allowed to execute and the reply "Executing" is returned. Otherwise the transaction is placed in a waiting list and the reply "Waiting" is returned.

$$\begin{aligned}&\mathit{RequestInventoryAccess} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{transaction}? : \mathit{ClientTransaction};\\ &\quad \mathit{reply}! : \mathit{INVENTORY\_REQ\_REPLY} \mid \mathit{transaction}? \in \mathit{transactions} \wedge\\ &\quad (\mathbf{if}\ (\mathit{db}.\mathit{inventory}.\mathit{updateinprogress} = \mathit{True} \wedge \mathit{db}.\mathit{inventory}.\mathit{active} \neq \emptyset)\\ &\quad \mathbf{then}\ (\mathit{db}'.\mathit{inventory}.\mathit{waitinglist} = \mathit{db}.\mathit{inventory}.\mathit{waitinglist} \frown\\ &\quad\quad \langle \mathit{transaction}? \rangle \wedge \mathit{reply}! = \mathit{Waiting})\\ &\quad \mathbf{else}\ (\mathit{db}'.\mathit{inventory}.\mathit{active} = \mathit{db}.\mathit{inventory}.\mathit{active} \cup \{\mathit{transaction}?\} \wedge\\ &\quad\quad \mathit{reply}! = \mathit{Executing})) \,]\end{aligned}$$

The *AwakenTransaction* operation takes the next available transaction from the waiting list and allows it to modify the inventory, if it is available. Transactions are removed from the waiting list in FIFO order. If the inventory is available, the first transaction in *waitninglist* is allowed to access the inventory, and the transaction and a reply of "Executing" is returned. If the inventory is not available, the first transaction in the waiting list is returned with a reply of "Waiting".

$$
\begin{aligned}
&\mathit{AwakenTransaction} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{transaction}! : \mathit{ClientTransaction};\\
&\quad \mathit{reply}! : \mathit{INVENTORY\_REQ\_REPLY} \mid\\
&\quad \mathbf{if}\ (\mathit{db}.\mathit{inventory}.\mathit{updateinprogress} = \mathit{False} \land \mathit{db}.\mathit{inventory}.\mathit{active} = \emptyset)\\
&\quad \mathbf{then}\ (\mathit{db}'.\mathit{inventory}.\mathit{active} = \{\mathit{head}\ \mathit{db}.\mathit{inventory}.\mathit{waitinglist}\} \land\\
&\quad \mathit{db}.\mathit{inventory}.\mathit{waitinglist} = \mathit{tail}\ \mathit{db}.\mathit{inventory}.\mathit{waitinglist} \land\\
&\quad \mathit{reply}! = \mathit{Executing} \land \mathit{transaction}! = \mathit{head}\ \mathit{db}.\mathit{inventory}.\mathit{waitinglist})\\
&\quad \mathbf{else}\ (\mathit{transaction}! = \mathit{head}\ \mathit{db}.\mathit{inventory}.\mathit{waitinglist} \land \mathit{reply}! = \mathit{Waiting}) \,]
\end{aligned}
$$

The operation *VerifyInventory* determines if there is sufficient inventory to fill an order.

$$
\begin{aligned}
&\mathit{VerifyInventory} \mathrel{\widehat=} [\, \Xi \mathit{OrderingSystem};\ \mathit{transaction}? : \mathit{ClientTransaction};\\
&\quad \mathit{reply}! : \mathit{INVENTORY\_INV\_REPLY} \mid \mathit{transaction}? \in \mathit{transactions} \land\\
&\quad \mathit{transaction}? \in \mathit{db}.\mathit{inventory}.\mathit{active} \land (\forall p : \mathit{Product};\ x : \mathbb{N}_1;\ q : \mathbb{N}_1 \mid\\
&\quad (p, q) \in \mathit{transaction}?.\mathit{order}.\mathit{items} \land (p, x) \in \mathit{db}.\mathit{inventory}.\mathit{items} \bullet\\
&\qquad (x \geq q \land \mathit{reply}! = \mathit{Available}) \land (x < q \land \mathit{reply}! = \mathit{Unavailable})) \,]
\end{aligned}
$$

If the inventory is sufficient to fill an order, the ordered items are placed in a hold list.

$$
\begin{aligned}
&\mathit{PlaceHold} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{transaction}? : \mathit{ClientTransaction} \mid\\
&\quad \mathit{transaction}? \in \mathit{transactions} \land \mathit{transaction}? \in \mathit{db}.\mathit{inventory}.\mathit{active} \land\\
&\quad \mathit{db}'.\mathit{inventory}.\mathit{active} = \emptyset \land (\forall p : \mathit{Product};\ x, y : \mathbb{N} \mid\\
&\quad (p, x) \in \mathit{transaction}?.\mathit{order}.\mathit{items} \land (p, y) \in \mathit{db}.\mathit{inventory}.\mathit{items} \land y \geq x \bullet\\
&\quad\ \ \mathit{db}'.\mathit{inventory}.\mathit{items} = \mathit{db}.\mathit{inventory}.\mathit{items} \setminus \{(p, y)\} \cup \{(p, y - x)\} \land\\
&\qquad \mathit{db}'.\mathit{inventory}.\mathit{holdlist} = \mathit{db}.\mathit{inventory}.\mathit{holdlist} \cup \{(p, x)\}) \,]
\end{aligned}
$$

If an order is cancelled later, due to an error or insufficient funds, the hold on the items in the inventory is removed. The items are removed from the hold list and placed back in the inventory.

$$
\begin{aligned}
&\mathit{RemoveHold} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{transaction}? : \mathit{ClientTransaction} \mid\\
&\quad \mathit{transaction}? \in \mathit{transactions} \land \mathit{transaction}? \in \mathit{db}.\mathit{inventory}.\mathit{active} \land\\
&\quad \mathit{db}'.\mathit{inventory}.\mathit{active} = \emptyset \land (\forall p : \mathit{Product};\ x, y : \mathbb{N} \mid\\
&\quad (p, x) \in \mathit{transaction}?.\mathit{order}.\mathit{items} \land (p, y) \in \mathit{db}.\mathit{inventory}.\mathit{items} \bullet\\
&\quad \mathit{db}'.\mathit{inventory}.\mathit{items} = \mathit{db}.\mathit{inventory}.\mathit{items} \setminus \{(p, y)\} \cup \{(p, y + x)\} \land\\
&\qquad \mathit{db}'.\mathit{inventory}.\mathit{holdlist} = \mathit{db}.\mathit{inventory}.\mathit{holdlist} \setminus \{(p, x)\}) \,]
\end{aligned}
$$

When an order has been paid for, the inventory levels in the database are reduced to reflect the sale. The ordered items are removed from the hold list.

$$
\begin{aligned}
&\mathit{UpdateInventory} \mathrel{\widehat=} [\, \Delta \mathit{OrderingSystem};\ \mathit{transaction}? : \mathit{ClientTransaction} \mid\\
&\quad \mathit{transaction}? \in \mathit{transactions} \land \mathit{transaction}? \in \mathit{db}.\mathit{inventory}.\mathit{active} \land\\
&\quad \mathit{db}'.\mathit{inventory}.\mathit{active} = \emptyset \land (\forall p : \mathit{Product};\ x : \mathbb{N}_1 \mid\\
&\quad (p, x) \in \mathit{transaction}?.\mathit{order}.\mathit{items} \land (p, x) \in \mathit{db}.\mathit{inventory}.\mathit{holdlist} \bullet\\
&\qquad \mathit{db}'.\mathit{inventory}.\mathit{holdlist} = \mathit{db}.\mathit{inventory}.\mathit{holdlist} \cup \{(p, x)\}) \,]
\end{aligned}
$$

The function *sumitems* returns the total cost of a list of items (taking into account quantities).

$$\begin{array}{|l}
sumitems : \mathbb{P}(Product \times \mathbb{N}_1) \rightarrow MONEY \\
\hline
\forall\, m : MONEY;\ p : Product;\ items : \mathbb{P}(Product \times \mathbb{N}_1) \mid (p, m) \in items \bullet \\
sumitems(\emptyset) = 0 \land sumitems(items) = p.ourcost * m + sumitems(items \setminus \{(p, m)\})
\end{array}$$

The function *calctax* is used to calculate the amount of tax due on an order. Since the amount of tax on an order may vary depending on location, this function is left undefined.

$$\begin{array}{|l}
calctax : MONEY \rightarrow MONEY
\end{array}$$

The operation *TallyOrder* tallies the value of an order.

$$\begin{array}{l}
TallyOrder \mathrel{\widehat{=}} [\ \Xi OrderingSystem;\ trans? : ClientTransaction \mid trans? \in transactions \land \\
\qquad trans?.order.subtotal = sumitems(trans?.order.items) \land \\
\qquad trans?.order.tax = calctax(trans?.order.subtotal) \land \\
\qquad (\exists\, x : MONEY \mid (trans?.order.shipmode, x) \in db.shippingrates \bullet \\
\qquad\quad trans?.order.shipping = x) \land trans?.order.total\ = \\
\qquad\qquad trans?.order.subtotal + trans?.order.tax + trans?.order.shipping\ ]
\end{array}$$

Once a transaction completes, (i.e., either a new order has been made or an old order has been cancelled), the transaction is finished and is removed from the list of active transaction.

$$\begin{array}{l}
EndTransaction \mathrel{\widehat{=}} [\ \Delta OrderingSystem;\ trans? : ClientTransaction \mid \\
\qquad trans? \in transactions \land transactions' = transactions \setminus \{trans?\}\ ]
\end{array}$$

We define below some important utility functions that are called by other modules in the system.

$$\begin{array}{|l}
decrementaccount : BankAccount \times MONEY \rightarrow MONEY \\
\hline
\forall\, b : BankAccount;\ m : MONEY \bullet decrementaccount(b, m) = b.balance - m
\end{array}$$

$$\begin{array}{|l}
decrementcard : CreditCard \times MONEY \rightarrow MONEY \\
\hline
\forall\, c : CreditCard;\ m : MONEY \bullet decrementcard(c, m) = c.balance - m
\end{array}$$

$$\begin{array}{|l}
availablecredit : CreditCard \rightarrow MONEY \\
\hline
\forall\, c : CreditCard \bullet availablecredit(c) = c.limit - c.balance
\end{array}$$

$$\begin{array}{|l}
availableacredit : ApprovedCredit \rightarrow MONEY \\
\hline
\forall\, a : ApprovedCredit \bullet availableacredit(a) = a.limit - a.balance
\end{array}$$

$$\begin{array}{|l}
decrementacredit : ApprovedCredit \times MONEY \rightarrow MONEY \\
\hline
\forall\, a : ApprovedCredit;\ m : MONEY \bullet decrementacredit(a, m) = a.balance - m
\end{array}$$

The next three operations discussed below are used to receive payment from the client. The operation that is executed depends on the mode of payment selected by the client.

When a client pays for an order using a cheque or a debit card the funds are drawn from the client's bank account. The Ordering System must inquire from the bank to ensure that the client has sufficient funds to cover the amount of the cheque or debit transaction. As with a credit company, it is assumed that the bank provides an electronic means for the Ordering System to request client information. Similarly, the Ordering System must verify that the client has sufficient funds to cover the cost of items when paying via credit card or approved credit.

$$\begin{aligned}
\mathit{PayByAccount} \mathrel{\widehat=} [\, & \Xi \mathit{OrderingSystem};\ \mathit{reply}! : \mathit{FUND\_CHECK\_REPLY}; \\
& \mathit{trans}? : \mathit{ClientTransaction} \mid \mathit{trans}? \in \mathit{transactions} \land \\
& \mathit{trans}?.\mathit{order}.\mathit{paymode} = \mathit{Cheque} \lor \mathit{trans}?.\mathit{order}.\mathit{paymode} = \mathit{Debit\_Card} \land \\
& \mathbf{if}\ (\mathit{trans}?.\mathit{client}.\mathit{account}.\mathit{balance} \geq \mathit{trans}?.\mathit{order}.\mathit{total}) \\
& \mathbf{then}\ (\mathit{trans}?.\mathit{client}.\mathit{account}.\mathit{balance} = \\
& \qquad \mathit{decrementaccount}(\mathit{trans}?.\mathit{client}.\mathit{account}, \mathit{trans}?.\mathit{order}.\mathit{total}) \land \\
& \qquad\quad \mathit{reply}! = \mathit{Success})\ \mathbf{else}\ (\mathit{reply}! = \mathit{InsufficientFunds})\ ]
\end{aligned}$$

$$\begin{aligned}
\mathit{PayByCredit} \mathrel{\widehat=} [\, & \Xi \mathit{OrderingSystem};\ \mathit{reply}! : \mathit{FUND\_CHECK\_REPLY}; \\
& \mathit{trans}? : \mathit{ClientTransaction} \mid \mathit{trans}? \in \mathit{transactions} \land \\
& \mathit{trans}?.\mathit{order}.\mathit{paymode} = \mathit{Credit\_Card} \land \\
& \mathbf{if}\ (\mathit{availablecredit}(\mathit{trans}?.\mathit{client}.\mathit{creditcard}) \geq \mathit{trans}?.\mathit{order}.\mathit{total}) \\
& \mathbf{then}\ (\mathit{trans}?.\mathit{client}.\mathit{account}.\mathit{balance} = \\
& \qquad \mathit{decrementcard}(\mathit{trans}?.\mathit{client}.\mathit{creditcard}, \mathit{trans}?.\mathit{order}.\mathit{total}) \land \\
& \qquad\quad \mathit{reply}! = \mathit{Success})\ \mathbf{else}\ (\mathit{reply}! = \mathit{InsufficientFunds})\ ]
\end{aligned}$$

$$\begin{aligned}
\mathit{PayByApprovedCredit} \mathrel{\widehat=} [\, & \Xi \mathit{OrderingSystem};\ \mathit{reply}! : \mathit{FUND\_CHECK\_REPLY}; \\
& \mathit{trans}? : \mathit{ClientTransaction} \mid \mathit{trans}? \in \mathit{transactions} \land \\
& \quad \mathit{trans}?.\mathit{order}.\mathit{paymode} = \mathit{Approved\_Credit} \land (\exists\, a : \mathit{ApprovedCredit} \bullet \\
& \qquad \mathbf{if}\ (a \in \mathit{db}.\mathit{approvedcredit} \land a.\mathit{clientid} = \mathit{trans}?.\mathit{client}.\mathit{clientid} \land \\
& \qquad\quad \mathit{availableacredit}(a) \geq \mathit{trans}?.\mathit{order}.\mathit{total}) \\
& \qquad \mathbf{then}\ (a.\mathit{balance} = \mathit{decrementacredit}(a, \mathit{trans}?.\mathit{order}.\mathit{total}) \land \\
& \qquad\qquad \mathit{reply}! = \mathit{Success})\ \mathbf{else}\ (\mathit{reply}! = \mathit{InsufficientFunds}))\ ]
\end{aligned}$$

Using the above three operations, we now define *PaymentMethod* as follows:

$$\mathit{PaymentMethod} \mathrel{\widehat=} \mathit{PayByApprovedCredit} \lor \mathit{PayByCredit} \lor \mathit{PayByAccount}$$

The next operation, *RecordKeeping*, records all of the details of a transaction. For new orders, the order details are stored in the order database, and accounts receivable is updated.

$$\begin{aligned}
\mathit{RecordKeeping} \mathrel{\widehat=} [\, & \Delta \mathit{OrderingSystem};\ \mathit{trans}? : \mathit{ClientTransaction} \mid \\
& \mathit{trans}? \in \mathit{transactions} \land \mathit{db}'.\mathit{orders} = \mathit{db}.\mathit{orders} \cup \{\mathit{trans}?.\mathit{order}\} \land \\
& (\exists\, a : \mathit{AccountReceivable} \mid (a.\mathit{client}.\mathit{clientid} = \mathit{trans}?.\mathit{client}.\mathit{clientid} \land \\
& \quad a.\mathit{payment} = \mathit{trans}?.\mathit{order}.\mathit{total}) \bullet \\
& \qquad \mathit{db}'.\mathit{accountsreceivable} = \mathit{db}.\mathit{accountsreceivable} \cup \{a\})\ ]
\end{aligned}$$

$$\begin{aligned}
\mathit{PrintInvoice} \mathrel{\widehat=} [\, & \Xi \mathit{OrderingSystem};\ \mathit{trans}? : \mathit{ClientTransaction};\ \mathit{order}! : \mathit{Order} \mid \\
& \mathit{trans}? \in \mathit{transactions} \land \mathit{order}! = \mathit{trans}?.\mathit{order}\ ]
\end{aligned}$$

Using the above components as basic building blocks, the actual purchase process is composed as follows:

$$\begin{aligned}
\mathit{PurchaseItem} \mathrel{\widehat=} & ((\mathit{GetCart} \mathbin{⨾} \mathit{FindProduct} \mathbin{⨾} \mathit{AddToCart} \mathbin{⨾} \mathit{CreateOrder} \mathbin{⨾} \mathit{TallyOrder} \mathbin{⨾} \\
& \mathit{PaymentMethod}) \land \mathit{RecordKeeping} \land \mathit{PrintInvoice} \land \mathit{UpdateInventory}) \mathbin{⨾} \\
& \quad \mathit{EndTransaction}
\end{aligned}$$

Notice that some of the components are composed sequentially. In fact, because Z has no built-in concurrency features, concurrent (or parallel) composition of *RecordKeeping*, *PrintInvoice*, and *UpdateInventory* could not be explicitly shown. However, during implementation, where the development tools and environment permit, these operations should be executed in parallel. The only pre-condition for their concurrent execution is that the *PaymentMethod* should commit execution.

7 Use Cases

In this section, we give a sample of the use cases that were produced during the behaviour requirements capture stage. A use case description provides "a step-by-step breakdown of the interaction between the user and the system for the particular case" [5]. Each use case description represents the usual way in which the actor will go through the particular transaction or function from end to end. Constantine [7] similarly describes use cases. A scenario represents specific paths through a use case. The use case scenarios guide the developers in relation to the sequence of steps required to perform a specific task.

A Sample Use Case Scenario:
A client places an order using the system. The following are necessary steps in making and processing the order:

- An order is received through either fax, phone, email, or web page.
- The order is converted into a generic order format.
- Check the order for correctness (i.e., verify that required information has been appropriately filled).
  - If order is not correct, then notify client of the errors.
- Verify order.
  - If payment is by cheque or debit card, verify that the client's bank account contains available funds.
    - If insufficient funds in bank account, then notify client of the problem and ask for other methods of payment.
  - If payment is by credit card, check client's credit rating.
    - If client's credit rating is not good, notify client of the problem and ask for other methods of payment.
  - If payment is by approved credit from our company, check client's credit limit.
    - If credit limit is not enough to cover payment, then notify client of the problem. Ask for other methods of payment.
    - If client does not have approved credit, notify client of problem. Notify user of where to get approved credit or ask for other methods of payment.
- Process order and update database.
- Update inventory.
- Make order record.

- Update accounts receivable.
- Send appropriate information to logistics and other departments.

The use case scenario given above is only an illustration. For lack of space, we omit other use case scenarios. We used UML to model the use cases. Our use of UML in addition to the Z notation is based on our conviction that a well-designed and documented system is easier to build and maintain than one that is poorly designed and documented. Thus, we model different aspects of the system using different paradigms to ensure quality.

8 Conclusion

Effective and efficient application of information technology, particularly the Internet, can be the key to achieving organizational change and improvements that directly influence a company's success in today's highly competitive business environment. There is, therefore, a need to formally design e-commerce core services in order to guarantee their correctness.

This paper is significant because it helps systems developers to design and implement an e-commerce system that is an integral part of an organisation's entire information technology system using a formal specification methodology. This methodology, which is verifiable, produces consistent and unambiguous specification, can easily integrate with overall business strategy, and guarantees the correctness and reliability of transactional processing of e-commerce transactions. In addition, it provides the mechanisms needed to enforce business rules and control transactional processing in order to guarantee robustness, correctness, consistency, and a fail-safe environment for e-commerce transactions. These features are essential because an e-commerce system provides a vital function in a business's overall information technology architecture and provides functionality that transcends departments, locations, and international boundaries. However, using a formal method in specifying e-commerce transactions is not a panacea. Formal methods must be appropriately integrated into the development process to capture those aspects of the system that require formality in order to reap the potential benefits of using formal design methodology.

This paper is one of our approaches to providing a robust and reliable e-commerce system. There is room for several extensions to this paper. First, we need to formally specify the security sub-system so that we can have a fully integrated system. Second, the shipping functionality requires specification. Further, we need to add other e-commerce metaphors, such as *comparative shopping* and *negotiation*, to the specification. Above all, we are working on the integration of several e-commerce solutions into a generic model that is tested, versatile, scalable, and amenable to customization.

References

[1] Abadi M., Birrel A., Stata R., and Wobber E., "Secure Web Tunneling", In *Proc. of WWW7*, 1997. (available at http://www7.scu.edu.au/programme/fullpapers/1859/com1859.htm)

[2] B. Aoun, "Agent Technology in Electronic Commerce and Information Retrieval on the Internet", In *Proceedings of AUSWEB96*, 1996. (Also available at: http://www.scu.edu.au/sponsored/ausweb/ ausweb96/tech/aoun/paper.html)

[3] M. Balabanovic, "Learning to Surf: Multiagent Systems for Adaptive Web Page Recommendation", *Ph.D. Thesis*, Department of Computer Science, Stanford University, Stanford, CA 94305-90250, March 1998.

[4] L. Baresi, A. Orso, and M. Pezze, "Introducing Formal Specification Methods in Industrial Practice", *Proceedings of the 19th International Conference on Software Engineering*, Boston, Massachusetts, USA, May 17-23, 1997, IEEE Computer Society, 1997. pp 56-66.

[5] S. Bennette, S. McRobb, and R. Farmer, *Object-Oriented Systems Analysis and Design using UML*, McGraw-Hill Publishing Company, 1999.

[6] D. Bustard, P. Kawalek, and M. Norris (Editors), *Systems Modeling for Business Process Improvement*, Artech House, May 2000.

[7] L. Constantine, "The Case for Essential Use Cases", *Object Magazine*, May 1997.

[8] S. A. Ehikioya, *Specification of Transaction Systems Protocols*, Ph.D Thesis, Department of Computer Science, The University of Manitoba, Winnipeg, MB, Canada, September 1997.

[9] S. A. Ehikioya, "An Agent-based System for Distributed Transactions: A Model for Internet - based Transactions". *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, Edmonton, Alberta, Canada. 1999.

[10] S. A. Ehikioya, "A Formal Foundation for Electronic Commerce Transactions", *International Symposium on Database Technology & Software Engineering, WEB and Cooperative Systems*, Baden-Baden, Germany, August 1-4, 2000.

[11] S. A. Ehikioya, K. E. Barker, and E. A. Onibere, "Specifying Correctness in the Automation of Banking Operations". In Adagunodo E. R., Kehinde L. O., Akinde A. D., and Adigun M. O. (Editors), *Computer-Based Automation in Developing Countries (Auto-DC '95).* Lagos, Nigeria. COAN Conference Series, Volume #6, May 1995. Pages 103 - 115.

[12] S. A. Ehikioya and K. E. Barker, "A Formal Specification Strategy for Electronic Commerce", *Proc. International Database Engineering and Application Symposium*, Montreal, Canada, August 25-27, 1997, IEEE Computer Society, 1997.

[13] S. A. Ehikioya and K. E. Barker, "Towards a Formal Specification Methodology for Transaction Systems Protocols". *3rd Annual IASTED International Conference on Software Engineering and Applications (SEA'99)*, Scottsdale, Arizona, USA, October 6 - 8, 1999.

[14] S. A. Ehikioya and K. Hiebert, "A Formal Model of Electronic Commerce", *First International Conference on Software Engineering, Networking, and Parallel and Distributed Computing*, Champagne-Ardenne, France, May 19-21, 2000.

[15] S. A. Ehikioya and K. Hiebert, "A Formal Specification of an On-line Transaction", *First International Conference on Software Engineering, Networking, and Parallel and Distributed Computing*, Champagne-Ardenne, France, May 19-21, 2000.

[16] S. A. Ehikioya and S. Jayarama, "Electronic Commerce for Services and Intangible Goods". *First International Conference on Internet Computing*, Monte Carlo Resort, Las Vegas, Nevada, USA, June 26-29, 2000.

[17] Group of Ten, "Electronic Money: Consumer Protection, Law Enforcement, Supervisory and Cross Border Issues", *Bank for International Settlements*, 1997.

[18] P. M. Hallam-Barker, "Electronic Payment Schemes", *W3C*, 1995. (Also available at: http://www.w3.org/pub/WWW/Payments /roadmap.html).

[19] P. M. Hallam-Barker, "Micro Payment Transfer Protocol (MPTP) Version 0.1", *W3C*, November 1995. (Also available at: http://www.w3.org/pub/WWW/TR/WD-mptp.html).

[20] R. Inder, M. Hurst, and T. Kato, "A Prototype Agent to Assist Shoppers", In *Proc. of WWW7*, 1997. (available at http://www7.scu.edu.au/programme/posters/1856/com1856.htm)

[21] A. Moukas and G. Zacharia, "Evolving Multiagent Filtering Solution in Amalthaea", *Proc. of the 1st International Conference on Autonomous Agents*, pages 394-403, February 1997.

[22] M. Naor and B. Pinkas, "Secure Accounting and Auditing on the Web", In *Proc. of WWW7*, 1997. (available at http://www7.scu.edu.au/programme/fullpapers/1927/com1927.htm)

[23] D. O'Mahony, M. Peirce, and H. Tewari, *Electronic Payment Systems*, Artech House, 1997.

[24] W. Rajput, *E-Commerce Systems Architecture and Applications*, Artech House, June 2000.

[25] B. Reich and I. Ben-Shaul "A Componentized Architecture for Dynamic Electronic Markets", *SIGMOD Record* — Special Issue on Electronic Commerce, Vol. 27 #4, December 1998.

[26] M. Reynolds, *Beginning E-Commerce with Visual Basic, ASP, SQL Server 7.0 and MTS*, Wrox, 2000.

[27] *Special Issue on Electronic Commerce*, SIGMOD Record, Vol. 27 #4, December 1998.

[28] J. M. Spivey, *Introducing Z: A Specification Language and its Semantics*. Cambridge University Press, 1988.

[29] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd Edition, Prentice Hall International Series in Computer Science, 1992.

[30] C. Standing, *Internet Commerce Development*, Artech House, 2000.

[31] J. D. Tygar, "Atomicity versus Anonymity: Distributed Transactions for Electronic Commerce", *Proceedings of the 24th VLDB Conference*, New York, USA, 1998.

[32] *Z/EVES Version 2.0*, ORA Canada, Ottawa, Ontario, K1Z 6X3, CANADA (Also available at http://www.ora.on.ca/z-eves /welcome.html). (Also associated with this is M. Saaltink and I. Meisels, *The Z/EVES Reference Manual*, ORA Canada, December 1995; Revised October 1999).