Addressing Concurrency Using UML-Based Software Development

by

Jie Xiong
School of Computer Science
McGill University, Montreal

June 2004

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

Your file Votre référence
ISBN: 0-494-06474-9
Our file Notre référence
ISBN: 0-494-06474-9

Abstract

Distributed systems, systems that interact with real-time devices, responsive GUI interfaces, systems that interact with hundreds of clients simultaneously have to function correctly even in a concurrent environment. Complex concurrent activities and interactions however make the development, i.e. understanding, analyzing, designing and implementing, of such systems extremely difficult. It is important to have a systematic approach to treat the many issues when developing concurrent systems.

In this thesis, we describe an approach that addresses concurrency in all phases of object-oriented software development. We show how to identify inherent concurrency at early stages of the development, and we propose a way to systematically refine the resulting declarative specification into an object-oriented design, which controls concurrency and provides data consistency using transactions or monitors.

Key words: Concurrency, UML, Fondue, Transactions, Monitors, Object-oriented, Auction System

Résume

De nos jours, de plus en plus de logiciels doivent faire face à la concurrence inhérente dans leur environnement. Les systèmes distribués, ainsi que les systèmes temps-réels, les logiciels avec interfaces graphiques sophistiqués, et les systèmes qui gèrent des centaines de clients simultanément doivent fonctionner d'une manière correcte même en présence de parallélisme. Pourtant, les interactions coopératives et compétitives d'activités parallèles compliquent considérablement la compréhension, l'analyse, la concéption et l'implémentation de logiciels. Pour produire des applications correctes et fiables, il est important de suivre une approche systématique de traitement de la concurrence pendant le cycle de développement d'un logiciel.

Dans ce travail de maîtrise je présente une approche qui s'occupe de la concurrence pendant toutes les phases du processus de développement d'un logiciel. Initialement, je montre comment identifier la concurrence inhérente dans l'environnement, puis comment spécifier la concurrence d'une manière déclarative pendant l'analyse. Ensuite, cette spécification déclarative est transformée en une conception orienté-objet, qui gère la concurrence en utilisant les moniteurs ou les transactions.

Mots clés: concurrence, UML, Fondue, Transactions, Moniteurs, Orienté-objet, Système des enchères

Acknowledgement

I am extremely grateful to Dr. Jörg Kienzle, my thesis supervisor, for his supervision and inspiration on my work. Throughout the 2003~2004 academic year, Dr. Kienzle guided me through my research in the Software Engineering Lab at School of Computer Science of McGill University. His guidance inspired my research ideas. His constant motivations helped me overcome the difficulties I met during my research. His reviews and suggestions greatly helped me improve the quality of my thesis. I greatly benefited from the discussions with him with respect to the research. His accuracy and talent become the drive that always motivates me to learn by heart and work hard. It was with Dr. Kienzle's great help that I worked out creative solutions for this concrete research topic.

I would also like to thank my parents and my brother for supporting my pursuit for higher education. It is with their deep love and constant encouragement that I am able to finish my work here.

TABLE OF CONTENTS

List of Figures & Tables

Chapter 1. Introduction

In this chapter, we start by introducing the concepts that are in the domain of *Object-Oriented Software Development* and issues that are related to *Concurrency*. Then, a problem description reveals the core problem we are addressing in this thesis. The thesis organization briefly introduces the content of each chapter.

1.1 Background

Object-Oriented Software Development Object-oriented software development (OOSD) is a dominating method in today's software industry to let the software system model the real world. Basically, we use object-oriented model as an abstracted representation of the real entities and their relationships. These models are usually represented by using standard software engineering notations such as Unified Modeling Language (UML) [1]. Then, we use these models to capture the functionalities, services or problems that need to be solved in the real world. Finally, applications will be developed based on these models.

The basic unit of an object-oriented model is the *class*. A *class* is an abstraction of objects. Objects are instances of the *class*. An object consists of several fields that are called attributes, describing the *state* of an object. An object also contains a set of methods, describing the *behavior* of the object.

The Software Development Process In today's software engineering industry, a typical software development process usually follows the core phases of capturing requirements, analysis, design, implementation, testing and deployment on an iterative development base. A well-known example of such a process is called Rational Unified Process (RUP) [2]. Other processes could have more or less development phases than RUP and hence have different advantages and disadvantages.

In this thesis, we will follow another OOSD process called Fondue [3], a software development method that specifically addresses reactive systems. Fondue is an extension of the Fusion [4] method. An introduction of Fusion and Fondue will be presented later in chapter 3.

Concurrency Modern software applications have a growing trend of employing more concurrency control, or providing more concurrency support, as we can easily see from some examples.

In distributed client-server systems, the server usually provides multithreaded services. For instance, a library system provides online book searching and reservation services. The user interacts with the central server via the user interface at different terminals. Although each user interface is intended to be single-threaded, the server, on the contrary, usually has to correspond to multiple requests initiated from users at different terminals at the same time.

In e-Commerce applications, such as online shopping systems, the web servers usually have to handle a significant number of concurrent and multiple connections from users coming from different locations in the world. These services must be highly reliable. Data must be kept consistent in spite of concurrency and failures.

1.2 Problem Description

There has been research interest in links between OOSD and concurrency more than a decade ago. Throughout the years, object-oriented software applications (especially the Internet-based applications) are growing more and more complex. They are required to respond to an increasing number of simultaneous requirements and operations. Thus we see a growing concern for addressing concurrency in OOSD.

However, complex concurrent applications are more likely to contain software design problems, which will eventually lead to system failure. Using ad hoc solutions to address concurrency in object-oriented systems usually makes the systems unnecessarily complicated. It also makes it hard to maintain the program code of the systems and thus results in poor application performance.

The target of this thesis is to concentrate on a couple of particular means for achieving concurrency control as a concrete and systematic approach in object-oriented systems. We will discuss transaction-oriented design and monitor-based design. Ultimately, we aim at integrating transactions and monitors into software development process to address concurrency. In the meanwhile, we will emphasize using transactions as an advanced solution in more complex, highly concurrent and distributed systems.

As a result of addressing concurrency, we will be able to identify the need for using transactions or monitors in the system and finally elaborate a structured way of establishing their boundaries.

1.3 Thesis Preparation

Prior to this thesis, I have done a reading course under the guidance of Professor Jörg Kienzle. From the reading course I gained profound understanding of Fusion,

Fondue, and RUP, which helped me build a solid background with respect to software development process in object-oriented systems.

In parallel to this master's thesis, I have contributed part of my work to the paper *Addressing Concurrency during Software Development,* which has been submitted to the UML2004 conference for review. The authors are Professor Jörg Kienzle at McGill University, Professor Shane Sendall at University of Geneva and me. My thesis extended the problems addressed in this paper and solutions proposed in this paper.

1.4 Thesis Organization

This thesis contains eight chapters. The content of each chapter is briefly described below.

Chapter one is the introduction of the entire thesis. In this chapter, essential knowledge background for this thesis is briefly introduced. After describing our targeted problem and our aim to solve the problem, the contents of each chapter are introduced.

In chapter two we gave an overview of the Online Auction System[1]. The overview introduced the rules of the Auction System that are from the most well known auction type, English auction. In addition, the overview introduced the services provided in the system, the possible user activities and the physical architecture of the system.

Chapter three is an introduction to the Fondue development method and an overview of *Concurrency* related issues. The introduction to Fondue included a brief introduction to Fusion, a brief introduction to UML, an explanation of their relationships with Fondue and a summary of Fondue. The overview of

[1] For simplicity reason, from now on in the thesis we will just call the Online Auction System as Auction System.

concurrency explained the concept of concurrency, the possible problems that are related to concurrency in software applications and our proposed solutions.

Chapter four is about Fondue requirement engineering, which is the first phase in the Fondue development process. In this chapter we discussed the Fondue style use cases. To present a systematic view of the Auction System, we elaborated on the *Buy and Sell Item by Auction* use case.

In chapter five we illustrated the Fondue analysis process by working on the Auction System case study. The purpose of this chapter is to show how we conduct the Fondue analysis step by step and how we extend Fondue by specifying concurrency that the system has to deal with in the analysis phase. New notations especially designed to demonstrate concurrent states of the models have been added to the original analysis models.

Chapter six is the chapter for design and implementation. Our purpose in this chapter is to find a design that can provide the required functionalities and the requested concurrency and data consistency. In order to illustrate how to handle concurrency related problems, we made sequential design and concurrent designs focusing on the `placeBid` operation in the Auction System.

In chapter seven, we proposed some new thoughts about addressing concurrency under a more complex environment. To illustrate our new concern, we altered the `placeBid` example and suggested some future work with respect to the modification.

Chapter eight is the conclusion of this thesis. We reviewed the Fondue development process with respect to the auction case study. Finally we concluded our systematic approach to handle concurrency in a higher level of abstraction.

1.5 Abbreviations

The meanings of abbreviations used in this thesis are explained below:

UML	Unified Modeling Language
OCL	Object Constraint Language
RUP	Rational Unified Process
OOSD	Object-Oriented Software Development
OOSE	Object-Oriented Software Engineering
OMT	Object Modeling Technique
GUI	Graphic User Interface
ACID	Atomic, Consistency, Isolation, Durability

Chapter 2.

Overview of Online Auction System

The original information of the auction service example comes from [5] and an informal description of the Auction System is found in [6]. The overview that will be given here is based on the two references mentioned above. In addition, there are also some specialized auction websites where live examples can be found. Their rules can be used as references for the system we are introducing. These websites include eBay, uBid and iBazar. Among them, iBazar has been bought by eBay; uBid was founded in 1997 and now it is a major force in the e-Commerce world. The URLs of these websites are as follows:

eBay http://www.ebay.com

uBid http://www.ubid.com

iBazar http://www.ibazar.com

2.1 System Architecture

Figure 1 illustrates the physical architecture of the Auction System.

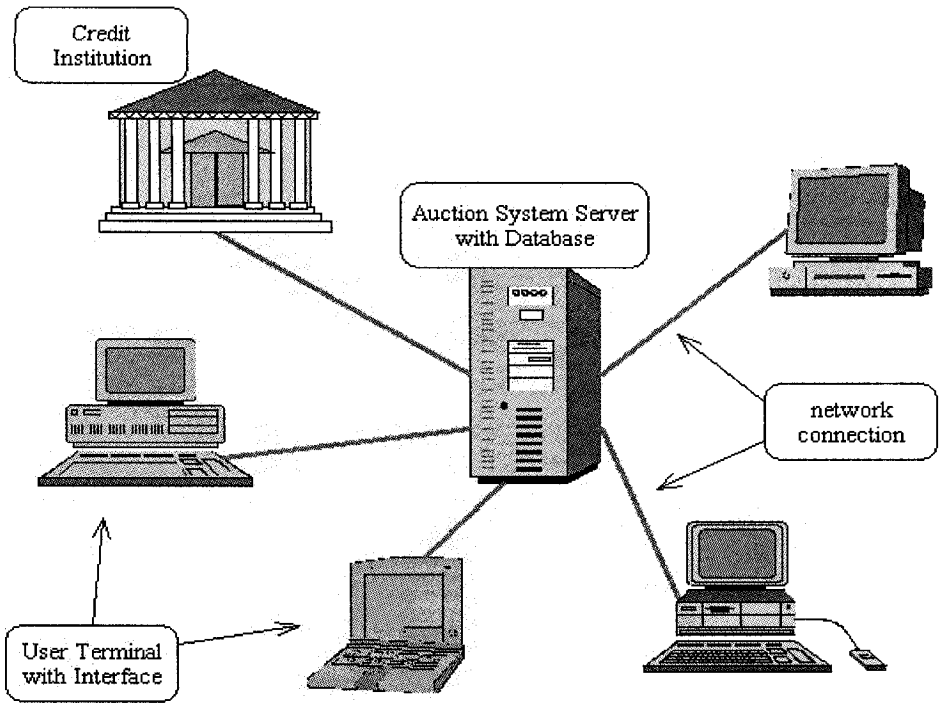

Figure 1. Auction System Architecture

As we can find out from the above figure, the whole system consists of a dedicated central server, a network that connects a group of computers and credit institutions. The server can communicate with the credit institutions via the network. The computers that are connected to the network are the terminals that users can use to access the system. A Graphical User Interface (GUI) is provided on each terminal. The terminals can run on different operating systems, such as Windows, Unix, Linux and Macintosh OS.

2.2 General Services and Rules

Briefly, registered users of the system can browse items, buy and sell items in the system. To buy or sell items, the users must follow the pre-specified auction rules. In our case study, we will only use the rules of the English Auction as example.

Price Setting To conduct an English Auction, the item for sale will have a minimum starting price, i.e. a price set by the seller that is supposed to be low enough to attract buyers (other registered users) to start bidding on the item.

There should also be a minimum increment price for each valid new bid. That is, a user can make as many bids as he/she wishes till the end of the auction, but each new bid should obey the minimum increment rule.

Winning an Auction In an English Auction, the user who makes the highest valid bid wins the auction. If there is a winner of the auction, the system will withdraw the winner's bid amount from his/her account. After charging a commission fee of the winning bid, the rest amount of the bid money will be deposited into the seller's account.

Registration All interested users who want to participate in the Auction System must sign up to become registered users [2]. Required registration information includes the user's real name, address, email address, desired username and password. Once a user successfully registered with the system, he/she will become a customer. If a customer wants to buy or sell item in the system, he/she will be required to input his/her credit information. That is, the customer will provide his/her credit card number to the system. In addition, he/she will specify a certain amount of money that can be withdrawn from his/her credit card and then be transferred to his/her associated bidding account in the Auction System. The bidding account[3] is specifically used for the buying (via placing bids) and selling activities.

Since a customer has credit card and an account, he/she can transfer fund between the credit card and the account. For instance, if a user's account balance is not sufficient to place a valid bid, he/she can ask the system to debit a certain amount of money from his/her credit card and deposit it to his/her account. The money in the user's account can also be transferred back by the system to his/her credit card in case of need. Since a customer can place multiple bids, the sum of

[2] By default and to keep it simple, we will just call registered users customers in the following chapters of the thesis

[3] For simplification reason, we will call bidding account as account in the following chapters of the thesis

all bids he/she placed should never exceed the total amount of money available in his/her account.

2.3 Customer activities

A registered user (customer) can have a series of activities in the system, depending on the procedure of the auction and his/her purpose.

Login A customer must login the Auction System using his/her username and password before he/she can use the system. Once logged in, the customer can browse the current auctions, consult the history of an ongoing auction, join an ongoing auction, sell item by starting an auction, and manage his/her account.

Browse current Auctions After successful login, a customer can browse a list of auctions that are currently active. The system keeps a list of active auctions and it shows the title of each active auction with a description of the item for sale, the closing time of the auction and the current highest bid amount.

Bidding in an Auction A customer who wishes to place bid in an auction must first request to join the auction. The system only allows the customer to join an active auction. An active auction means the auction is still open.

To place a bid in an auction, the customer must follow certain rules:

1. The seller him/her self cannot bid, i.e. the bidder of the auction must *not* be the starter of the auction.
2. The initial bid should be at least as high as the minimum starting price of the auction item. Each new valid bid should satisfy the minimum increment rule.
3. A valid bid must ensure guaranteed balance on the bidder's account. That is, while playing a new bid, the customer's account balance should be no lower than the sum of or his/her pending bids plus the amount of the new bid.

As long as the customer satisfies the above requirements, he/she is allowed to place bids across as many auctions as he/she wishes.

The Auction System supports concurrent operations. Therefore, multiple customers can place bids in the auction at the same time. More generally, multiple users can interact with the system simultaneously.

Starting an Auction A customer who wants to sell in the Auction System acts as a seller. By default, one auction has one item for sell. The auction will be started by the seller.

To start an auction, the seller must provide enough information to the auction by means of filling out an item form. In the form, the seller will give a title of the item for sale, provide a detailed description for the item, set a minimum starting price, reserve price and minimum increment, the starting date of the auction and the duration of the auction. The duration can be either a fixed period or a pre-defined time out. For instance, a fixed period could be 7 days, 14 days or even 30 days. A pre-defined time out could be a one-day idle period since the last bid.

The seller has the right to cancel the auction anytime before it is started. Once the auction has been started, the auction will be active and the seller cannot cancel it.

Closing an Auction Usually there are two ways to determine the end of an auction. As we mentioned before, some auctions have a fixed period of duration, e.g 7 days or 14 days or 30 days. Upon the end of the duration, the auction is closed. Alternatively, some other auctions set a time-out value to every new bid. That is, if there is no new valid bid after a certain time-out since the current valid highest bid, the auction will be closed and the winner will be announced.

Upon an auction's closing, if there is not a single valid high bid, i.e. if none of the customers has placed a valid bid or the highest bid does not meet the reserve price set by the seller, the auction is regarded as unsuccessful. In this case, neither the seller nor the buyer who participated in the bidding will be charged.

Consult Auction History Each auction keeps track of all the bids placed in the auction. Once a customer joins an active auction, he/she can browse the bid history of the auction.

2.4 Goods Delivery after Auction

Usually it is the seller's responsibility to send out the item to the winner of the auction soon after the auction is closed. Once the winner receives the item, he/she can vote on the quality of the delivery, which will later on be reflected as the seller's credit ranking in the system by means of recording seller's history. The auction site eBay is an example of this. Other auction sites could have different ways to guarantee goods delivery. For example, some sites will hold the winner's money and will not deposit it to the seller's account until the winner receives the goods and is satisfied on its condition.

2.5 Fault-Tolerance Requirements

Software fault tolerance is highly desired in the Auction System. In fact, the system must be able to tolerate any failure during the operations. As we mentioned in the problem statement section of chapter one, the system is required to have the ability to handle concurrently executed operations, i.e. be able to interact with multiple users simultaneously. In addition, if there is any system crash, the state of the system should not be corrupted. Specifically, money transfer between different accounts should be atomically executed. Any partial execution of such operations is not allowed in any case.

Chapter 3.
Fondue and Concurrency

In this chapter, we will systematically summarize the Fondue method, one of the software engineering methods that are based on object-oriented systems. A detailed introduction to Fondue can be found in [7], an electronic lecture notes about Fondue from Swiss Federal Institute of Technology. Briefly, we will cover the relationship between Fondue and UML, the relationship between Fondue and Fusion, the Fondue process and the Fondue models. Afterwards, we will discuss the concepts of concurrency and its solutions. Here, the contribution of the thesis is extending Fondue to handle concurrency in OOSD.

3.1 The Fusion Development Method

Fondue is based on Fusion, but it uses UML as notation. Fusion is a systematic and logical OOSD method originally devised by Derek Coleman and other researchers in 1994.

Compared with Fondue, Fusion comes at a relatively early stage in OOSD. Fusion integrates the essential object-orientation concepts and techniques from Object Modeling Technique (OMT) by Rumbaugh [8] and Object-Oriented Software Engineering (OOSE) by Jacobson [9]. Fusion extends these existing methods and specifically addresses reactive systems.

3.1.1 The Fusion Process

The Fusion method covers the phases of analysis, design and implementation. A special characteristic of Fusion is that Fusion has no requirements capturing phase, since business customers usually conduct the work by themselves.

The Fusion analysis describes what the system does. It includes:

- Capturing the concepts and relationships of the object model in the domain of the problem
- Creating the object model
- Developing the interface of the system
- Completing the life-cycle model and operation model
- Reviewing the analysis models by checking modeling consistency against requirements

The Fusion design describes how the system works. It includes

- Developing object interaction graphs
- Developing visibility graph which shows the structure of the object-oriented system
- Building class descriptions which specify the internal state and external interface required by each class
- Developing an inheritance graph of the classes
- Updating class descriptions with the new inheritance information

Implementation is the final stage of the Fusion method, which means mapping the design to a programming language. Generally, the implementation phase includes coding, performance inspection and testing.

3.1.2 Advantages and Disadvantages of Fusion

Compared with other OOSD methods, Fusion is relatively simple, but it is comprehensive at the same time since it covers the stages from analysis to implementation. This nature makes Fusion well suited for developing small and medium-sized systems.

The disadvantage of Fusion method is that it is relatively limited in the scope of application. For example, Fusion does not deal with user interface design and database design; Fusion does not deal with synchronization of concurrent operations in distributed systems; Fusion cannot be applied to real-time systems, etc.

3.2 UML

UML is the unification of notations used in OMT and OOSE. It also absorbed contributions from other OOSD notations, such as Harel's [10] Statecharts. UML has been adapted as a standard by the Object Management Group (OMG). The current version of UML is 1.5 and UML 2.0 is very close to completion.

As an industry-standard modeling language, UML is designed for a broad range of applications. The goal of having UML is to provide graphical tools to visualize, specify, construct and document the software systems.

UML has a set of diagrams that can be used to describe a software system from different viewpoints. In terms of modeling, different UML diagrams can be used to show different concerns within the scope of the modeled system.

3.2.1 Static UML Diagrams

By "static" we mean the diagrams that display structure, state, relationships, and functionality of the system model.

The *Use Case Diagram* describes what the system does from an external point of view.

The *Class Diagram* describes the structure of the system by identifying class entities and their relations. The relations include association, aggregation and generalization, etc.

The *Component Diagram* groups different elements of the system into components. It shows the organization of the components and the relationships among the components.

The *Deployment Diagram* reflects the run-time configurations of the elements in the system, including hardware nodes and software components that are installed on the nodes.

3.2.2 Dynamic UML Diagrams

By "dynamic" we mean the diagrams that display behaviors of the objects and elements of the system.

A *Sequence Diagram* focuses on the time issue and shows how a group of objects collaborate with each other. A sequence diagram can reflect the behavior of a use case.

A *Collaboration Diagram* displays similar information as a sequence diagram but it focuses on the message passing issue. All messages are numbered with arrows showing their ordering.

An *Activity Diagram* represents the control flow of an entire process or multiple processes of the system. The control flow consists of a set of operations where the completion of one operation invokes execution of another operation.

A *Statechart Diagram* provides a detailed view of state changes of an individual object and transitions among these states. Usually a state refers to a value of the attributes of the object being described.

3.3 The Fondue Development Method

As we mentioned, Fondue is based on Fusion, but it extends Fusion in many ways. Fondue inherited methods and models built in Fusion, but Fondue also extended these methods and models so that it can deal with a wide variety of applications.

A contribution of the thesis is that we find a way to deal with concurrency problems in OOSD by extending the Fondue method, which is impossible to be done in Fusion.

3.3.1 Novelty in Fondue

Because Fondue extended Fusion, there are important improvements or new properties of Fondue.

First of all, Fondue uses UML as notation instead of using Fusion's own notations. This makes Fondue widely understandable.

Secondly, Fondue introduces pre condition and post conditions in operation schemas using Object Constraint Language (OCL)[11]. This is special and new in Fondue. It makes the Fondue operation schema more precise with more formal specifications.

Thirdly, Fondue uses a restricted form of state diagrams to describe sequencing of system operations. Using the state diagrams makes the information visible and makes it easier to understand than describing the same information using regular expressions as done in Fusion.

Fourthly, in Fondue the *Concept Model* is finally refined into a Design Class Model which makes implementation more straightforward. Fusion does not have such a refinement.

3.3.2 The Fondue Process

Requirements Like other OOSD methods, the requirements capturing phase addresses the needs of the stakeholders. Use cases are used at this level to capture the goals of the stakeholders, and a domain model is built to establish a common vocabulary for the system being modeled.

Analysis During the analysis phase, Fondue defines the intended behavior of the system, producing a precise specification.

The *Concept Model*, *Environment Model*, *Protocol Model* and *Operation Model* are built at this stage. These models describe the conceptual classes of the system and their relationships, the operations of the system and the allowed sequence of the execution of these operations. The models will be described in more detail in chapter 5.

Design During the design phase, the *Interaction Model*, *Dependency Model*, *Inheritance Model* and *Design Class Model* are produced. The models here display the attributes and methods of each class, the inheritance relationships between classes if any, and the interaction among the classes and how these interactions implement the system operations.

Implementation During the implementation phase, the design is mapped to a particular programming language. Fondue has pre-defined mapping for Ada95 and Java.

Verification Verification here means consistency check. In other words, Fondue defines rules that allow a developer to check the models at each phase for consistency (correctness and completeness).

3.3.3 The Fondue Notations

As we mentioned before, Fondue uses UML notations. Table 1 shows the corresponding relationship between the Fondue models and the UML diagrams.

The Fondue Models	UML Diagrams *
Use Case Model	Use Case Diagrams and Text
Domain Model	Class Diagram
Concept Model	Class Diagram
Environment Model	Collaboration Diagram
Protocol Model	State Diagram
Interaction Model	Collaboration Diagram
Dependency Model	Class Diagram
Inheritance Model	Class Diagram
Design Class Model	Class Diagram
Implementation Class Model	Class Diagram

Table 1. Fondue models VS. UML notations

* : The star symbol here indicates that Fondue makes use of the UML notation for its own model. This is not direct mapping but shows the corresponding relationships between Fondue models and UML diagrams. For example, The Fondue *Concept Model* uses UML *Class diagram*

3.3.4 The Fondue Models

In the following sections, we will introduce the main models we will use for our case study and briefly introduce other models.

The Environment Model As we can see from table 1, the Fondue *Environment Model* makes use of the UML *Collaboration Diagram*. The environment model consists of a system and a set of actors. The actors send input message(s) to the system and receive output message(s) from the system.

An input message will trigger an event in the system. An event can also be triggered by time (called a time-triggered event). These two kinds of events are called input events. An input event has an effect on the system, such as a change of system state or outputting of a message. The effect together with its associated input event is called a *system operation*.

The entire environment model consists of a set of input messages (that invokes a corresponding set of system operations) sent from external actors to the system, and a set of corresponding output messages returned form the system to the actors. Note that at any one point of time within the system, there can only be a single input event and thus a single system operation active. If there are multiple operations that need to be executed simultaneously, we will need to provide some technique to handle concurrency. Details regarding this issue will be discussed later on.

The Domain Model The *Domain Model* extracts and identifies the concepts in the problem domain. It also establishes relationships between the concepts. For example, classes are extracted from the specification of a problem. Relationships between classes are established afterwards. These relationships include *Association, Aggregation, Generalization, Specialization* and so on. The classes and their relationships form the domain model.

The Concept Model The *Concept Model* is a subset of the *Domain Model*. It is created by adding the system boundary to the domain model. All objects, classes and relationships that belong to the environment are excluded from the concept model. The actors and their communication paths to the system that originally belong to the domain model are also excluded from the concept model. Classes and relationships of the concept model only specify concepts that belong to the system itself.

The Operation Model The *Operation Model* specifies effects of the system operations on the conceptual state specified in the concept model. In addition, the generated output messages are specified. Every system operation specified in the environment model must be described in the operation model by an operation schema. The template for the Fondue operation schema is shown in figure 2.

Operation:	The system class name followed by the operation name and its parameter list, if any.
Description:	A description of the purpose and effects of the operation
Notes:	Any additional comments of the operation (optional)
Use Cases:	List of related use cases. (optional)
Scope:	List of all classes and associations involved in the operation
Message:	List of message types that are output by the operation together with their receiving actor classes
New:	List of the names of the new objects that are to be created by the operation
Alias:	List of names that act like aliases (optional)
Pre:	Pre stands for precondition. It is a Boolean expression written in OCL, representing a condition that must be met in order for the operation to be defined.
Post:	Post stands for post condition. It is a Boolean expression written in OCL, representing the effects of the operation on the system

Figure 2. Fondue template for operation schema

In order to guarantee the post condition to be true, the precondition must be met before executing the operation. Otherwise, the effect of the operation is undefined. Statements in the post condition are instantaneous semantics, meaning that each state change is executed atomically.

The Protocol Model The *Protocol Model* is depicted by a *Statechart diagram*, which allowed sequencing of input messages that can be sent to the system throughout its lifetime. That is, from the initial state of the system to the final state of the system.

The protocol model does not use the full power of UML *Statecharts*. The advantage is that such a diagram keeps a high level of abstraction, which is good for the analysis phase. Moreover, it avoids duplicated information that is already presented in the concept model and environment model.

The Interaction Model The *Interaction Model* shows how the run-time interaction among objects takes place to support the functionality specified in the operation model.

The interaction model consists of collaboration diagrams, and pseudo code if needed. Since a collaboration diagram can only show interactions among objects, pseudo code might be needed to describe complicated algorithms within a method. Every operation schema from the analysis phase must be described by a collaboration diagram.

To create an interaction model, firstly all relevant objects that are involved in the operations must be identified. (The operation schema from the analysis phase can provide related information.) Then the roles of the objects can be established, such as which object is the controller and which objects are the collaborators. Afterwards, the messages and the message sending paths between

objects must be decided. Finally, we need to check consistency between the collaboration diagrams.

The Design Class Model The *Design Class Model* is built based on the completed *Interaction Model.* The *Dependency Model* and the *Inheritance Model* are also helpful when building the *Design Class Model.*

The dependency model describes the dependencies among classes. The communication paths between the interacting objects are also shown in the dependency model. The inheritance model describes the inheritance structure between classes. The design class model consists of all design classes with attributes and methods used in all collaboration diagrams of the interaction model, and all associations among these classes.

3.3.5 The Visualized View for Fondue

Figure 3 shows a visualized workflow indicating how the analysis is realized in the design during the Fondue process. The workflow is based on the fact that the use case model has been built in the requirement elicitation phase. In the figure, the *Environment Model, Concept Model, Protocol Model* and *Operation Model* belong to the analysis phase. The *Interaction Model, Dependency Model, Design Class Model* and *Inheritance Model* belong to the design phase.

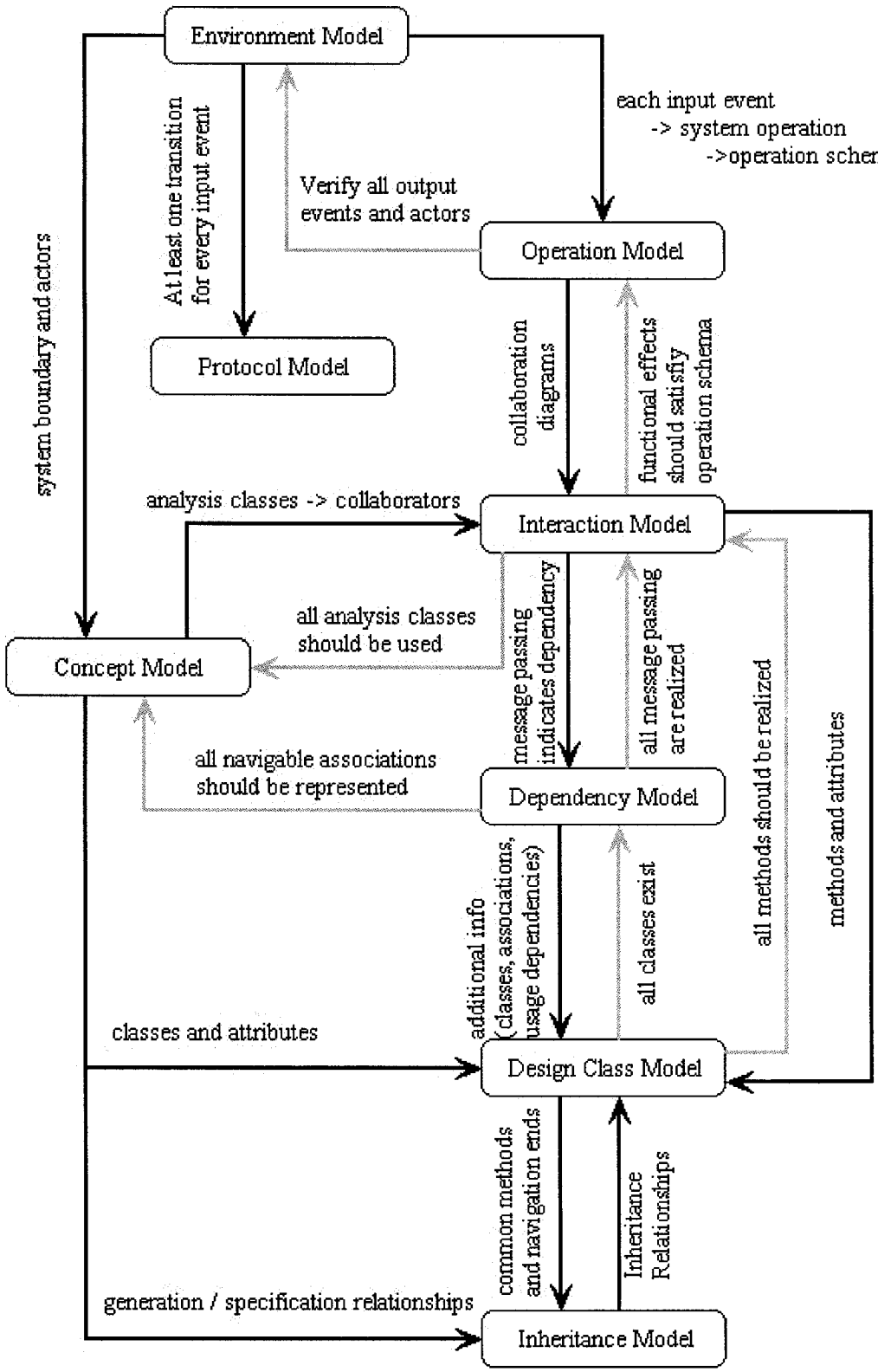

Figure 3. Fondue, from Analysis to Design

3.4 Addressing concurrency

Concurrency is our major concern in this thesis. However, the Fondue method does not automatically handle concurrency. A major purpose of this thesis is to extend the Fondue method to deal with concurrency. We will discuss concepts and solutions about concurrency here. How the concurrency related problems are solved during the Fondue development process will be illustrated in the following chapters.

3.4.1 Concurrency

Where does it come from? In general, concurrency refers to simultaneous execution of multiple processes or operations in computer systems. Concurrency exists in both centralized and distributed systems. In a centralized system, a concurrent situation could be that the system is sending out messages while listening to mouse clicks at the same time. In a distributed system, the concurrent situation arises when two or more operations from different client sides are trying to access the same piece of data on the server at the same time. We are more interested in dealing with concurrency in distributed systems.

What is the problem? Suppose we have an online banking system. At a certain time, the bank teller is depositing a customer's weekly salary into his/her account. In the meanwhile, the customer is trying to transfer funds from his/her account. Both of the operations will modify the account balance. Before transferring funds, the customer needs to check the account balance to see if he/she has enough money. In this case, the customer should not be able to get the balance until the deposit operation is finished. Otherwise, the customer will get an incorrect balance and make a wrong decision on whether to transfer funds or not.

3.4.2 Solution

Briefly, the point in addressing concurrency is to make updates atomic. In our example above, the customer should not see an intermediate balance when the deposit operation is being executed. He can see the balance either before or after the deposit. In addition, the deposit operation and the fund transfer operation cannot modify the balance at the same time. Even if the two modifications are invoked simultaneously, one modification should be blocked outside the data until the other modification is finished.

3.4.2.1 Monitor – A Simple Solution

The concept of monitor originally comes from a paper of Hoare [12] in 1974. The use of Monitors is a simple technique that guarantees atomic updates.

When we apply monitor to some data (for instance, an object, a variable), the monitor adds a lock to the data so that the data inside the monitor is not accessible from the outside until the lock is released. For example, if there are multiple procedure calls that are trying to access some data inside the monitor simultaneously, only one procedure call enters the monitor at any one time. All other procedure calls have to wait outside of the monitor until the current procedure in the monitor is finished and leaves the monitor. This property ensures atomic updates on the data.

In addition to providing mutual exclusion, monitors also have other properties. For instance, internal variables are private and they are not visible from outside the monitor, hence they do not reference data outside the monitor except through parameters.

3.4.2.2 Transaction – Advanced Solution

A transaction [13] is a logical single unit of work that groups together a set of operations performed on transactional objects (also known as data objects).

The operations *begin, commit* and *abort* are the three standard operations of a transaction. They form the boundary of a transaction. The operation *begin* marks the start of a new transaction, operations to modify transactional objects can now be executed. A transaction can *abort* during its execution. If this happens, the system will roll back to the state at the beginning of the transaction. When a transaction *commits,* it means the transaction has executed successfully and it wants to finalize the results. The effects of a committed transaction become permanent and will be visible to the outside.

A transaction has the ACID properties: *Atomicity, Consistency, Isolation and Durability.*

Atomicity A transaction is either performed completely, i.e. all of its data modifications are performed, or not at all. There is no observable intermediate state between the initial state and the result state.

Consistency Transactions preserve the consistency of the application state. The results of a transaction are considered to be consistent if the produced data satisfies all constraints and specifications of the application. Since the consistency criterion is application dependent, it is up to the programmer to write transactions in such a way that they produce consistent results.

Isolation Isolation requires that concurrently executed transactions do not affect each other. Data modifications made in a transaction are isolated from any other concurrent transactions. Transactions can share objects but data modifications must be serialized.

Durability Durability requires that the effects, or in other words, the data modification of a successfully completed transaction stays permanently in the system. Even after system failure, the system must be able to resolve the results.

These properties of a transaction ensure atomic updates on the data. Since a transaction can contain a group of operations that involve multiple transactional objects, it is a more advanced technique to handle concurrency compared with monitor.

Chapter 4.
Requirement Elicitation

This chapter is about requirement engineering, which is the first stage in Fondue method. To discover and document the functional requirements of the system, Fondue uses use case [14] as a communication means between the technical developer and the non-technical stakeholder of the software.

4.1 The Fondue Requirement Elicitation

A use case is a textual description describing the interactions of a particular actor with the system in pursuit of a precise goal. It must contain information on:

(1) How the use case starts and ends
(2) The context of the use case
(3) The actors and system behavior described as intensions and responsibilities
(4) All the circumstances in which the primary actor's goal is reached or not reached
(5) What information is exchanged

The Fondue use case addresses the behavioral requirements of the desired system in a way that is clearly related to the motivation for the system. The

motivation, in most cases, refers to the business vision of stakeholders. The general text-based use case style that Fondue uses is proposed by Cockburn [15].

Use cases in Fondue are classified into three levels: summary-level, user goal-level and subfunction-level. The summary level is the highest level, which gives a global view of all possible interactions with the system. The user goal level is the median level use case, which describes a goal that a primary actor is trying to achieve in the system. The sub-function level is the lowest level. Sub-function use cases are usually interactions that are required to be carried out in several user goal level use cases.

Fondue provides a template for use cases. (See figure 4)

Use Case: Define the use case name

Scope: Define what system is being involved

Level: Define the level of the use case

Intension: Statement of the goal and the conditions that make the goal happen

Frequency: Indicate any possible concurrency that exists in the system. (This is a new section of the template that is added by us to address concurrency)

Primary Actor: Role name or description of the primary actor

Precondition (optional): The condition in the system that has to be satisfied before the use case can be conducted

Main Success Scenario:

Use numbered steps to describe the interactions between the primary actor and the system

Extensions:

Each extension refers to a step in the main success scenario, providing either altered condition or exceptional behavior

Figure 4. Template of Fondue Use Case

4.2 The Buy and Sell Goods Use Case

Now we can apply the use case template to our case of Auction System and create the use cases. The *Buy and Sell Goods by Auction* use case (See figure 5) is the summary level use case. It presents the general view about the customer's activities within the Auction System.

Use Case: Buy and Sell Goods by Auction

Scope: Auction System

Level: Summary

Intension: The intension of the User is to buy and sell goods by auctions over time.

Frequency: A User can be involved in multiple auctions at any one time. Multiple Users can interact with the system concurrently.

Primary Actor: User (becomes Customer once he/she has registered him/herself with the system)

Main Success Scenario:

All Users must first register with the system before they have the right to use the system.

1. User registers with System, providing System with registration information.
2. System validates the registration information and enrolls the User.
3. Customer[4] identifies him/herself to System.

Steps 4-6 can be performed in parallel and individually repeated. A Customer may bid and sell in many auctions at any one time.

4. Customer increases credit with system.
5. Customer buys an item on auction.
6. Customer sells an item by auction.
7. Customer exits System.
8. Customer requests to cancel his/her enrollment.

[4] Now the User becomes a Customer since he/she successfully registered with the system

Extensions:

1a. User is already enrolled with the system. Use case continues at step 3.

2a. System ascertains that User did not provide sufficient information to register him/her.

2a.1 System informs User, use case continues at step 1.

3a. System fails to identify Customer; use case ends in failure.

Figure 5. The Buy and Sell Goods by Auction Use Case

Note the underlined phrases in the above use case refer to user goal level use cases that need to be elaborated further.

Figure 6 shows the *Buy and Sell Goods by Auction Use Case Diagram.*

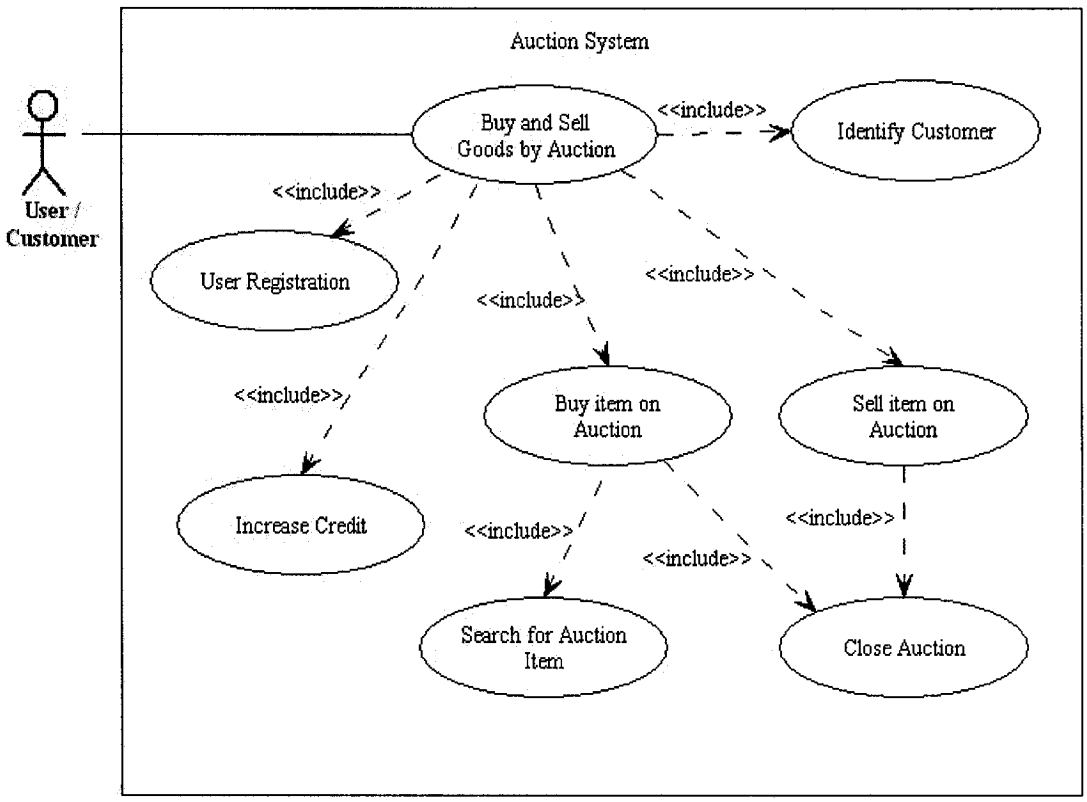

Figure 6. The Buy and Sell Goods by Auction Use Case Diagram

In the above diagram, each short text description circled in an ellipse represents a use case. Each dashed line with an arrowhead indicates a directed

connection between two use cases and the <<include>> stereotype specifies the hierarchical relationship between the two use cases. As indicated in figure 6, all use cases are included within the scope of the auction system. The solid line between the external actor and the summary level use case indicates the interaction relationship between the customer and the system.

In addition, the user goal level use case, *Buy item on Auction*, will be presented in detail (see figure 7) since it is closely related to the `placeBid` example we will use in the following chapters to address concurrency.

Use Case: Buy item on Auction
Scope: Auction System
Level: User Goal
Intension: The intension of the Customer is to follow the auction, which may then evolve into an intention to buy an item by auction, i.e. he/she may choose to bid for an item.
Frequency: The Customer may bid in many different auctions at any one time.
Primary Actor: Customer
Precondition: The Customer has already identified him/herself to the System
Main Success Scenario:
Customer may leave the auction and come back again later to look at the progress of the auction, without effect on the auction; in this case, the Customer is required to join the auction again.

1. Customer searches for an item under the auction.
2. Customer requests System to join the auction of the item.
3. System presents a view of the auction to Customer.

The steps 4-5 can be repeated according to the intensions and bidding policy of the Customer

4. Customer makes a bid on the item to System
5. System validates the bid, records it, secures the bid amount from Customer's credit, releases the security on the previous high bidder's

credit (only when there was a previous standing bid), informs participants of new high bid, and updates the view of the auction for the item with new high bid to all Customers that are joined to the auction.

The user has the high bid for the auction

6. System closes the auction with the winning bid by Customer

Extensions:

2a. Customer requests System not to pursue item further; use case ends in failure

3a. System informs Customer that auction has not started: use case ends in failure.

3b. System informs Customer that auction is closed: use case ends in failure.

5a. System determines that bid does not meet the minimum increment.

5a.1 System informs Customer; use case continues at step 4.

5b. System determines that Customer does not have sufficient credit to guarantee the bid:

5b.1 System informs Customer; use case ends in failure.

6a. Customer is not the highest bidder:

6a.1. System closes the auction; use case ends in failure.

Figure 7. The Buy Item on Auction Use Case

From the use case, it can be seen that the customer's intention to buy an item in the auction triggers the interactions between the customer and the system. In the body of the use case, The Main Success Scenarios and Extensions of the use case describe the interactions between the system and the external actor.

In both the summary level use case (figure 5) and the user goal level use case (figure 7), we used the *Frequency* section to indicate the possible occurrences of concurrent interactions between the actors and the system. This is interesting in the Fondue use case model because in this way it can be described in the context of concurrent applications.

Chapter 5. Analysis

The Fondue analysis is conducted by describing the system and its environment using a collection of models, each model describing a different aspect or view. This chapter walks through the Fondue analysis phase by working on the Auction System case study.

In our specification of the Auction System, we created the *Environment Model*, *Concept Model*, *Protocol Model* and *Operation Model*. As an extension to the traditional Fondue models, we added new notations in most of these models to address concurrency. To clearly indicate potential occurrences of concurrent operations in the system, we invented a set of reference tables based on the protocol model. These tables work with the sequential operation schema to help the developer identify shared concepts and eventually transform a sequential operation model into a concurrent version.

5.1 The Fondue Analysis Process

In the Fondue requirement elicitation phase, we have developed the *Use Case Model* and specified the problem domain. The Fondue analysis phase comes right after the requirement elicitation. Typically, the process of analysis follows the steps as described below:

(1). Develop the *Environment Model*

(2). Develop the *Concept Model*

(3). Develop the *Behavior Model*, which consists of the *Protocol Model* and the *Operation Model*

(4) Check the specification of the models for consistency and completeness

5.2 Environment Model

The *Environment Model* consists of a set of input messages sent from the actors (i.e. entities external to the system), and the corresponding set of output messages sent from the system to the actors.

Because Fondue is designed for developing reactive systems, every transformation of system state, executed in form of a system operation, must be triggered by an input event sent by some actor. The only exception is time-triggered events. The associated system operations are executed by the system in an automated way, triggered by elapsed time. One can however, for the sake of uniformity, imagine that they are triggered by a fictitious external clock actor. Figure 8 shows all the input and output messages that involve User actors (including registered and unregistered users) and Credit Institution actors in the Auction System.

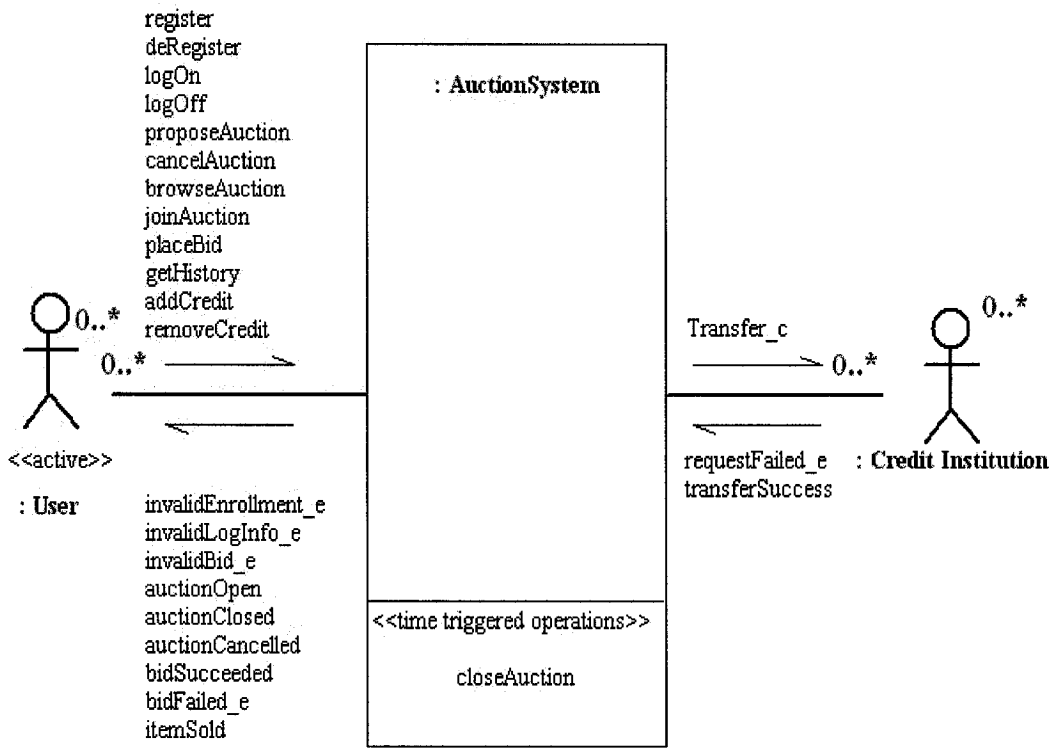

Figure 8. Environment Model of the Auction System

In general, input and output events are asynchronous. As a result, a system operation that is triggered by an input event coming from an actor most of the time sends back an output event in order to inform the actor of the outcome of the operation. Output events that notify an actor of exceptional outcomes use the naming convention _e . In the Auction System, for instance, placing a bid by sending the `placeBid` event might trigger the following output events:

- *bidSucceeded*, in case the user wins the auction
- *bidFailed_e*, in case some other user places a higher bid
- *invalidBid_e*, in case the proposed bid is not valid, e.g. bid amount is lower than the current bid

To illustrate that we have considered concurrency in the environment model, we added the multiplicity notation "0..*" to the actors and the input events. Adding multiplicity to user actors means multiple users can interact with the system at

any one time, and several actors can spontaneously send input messages to the system at a given time. Similarly, multiple Credit Institutions can interact with the system at the same time. Adding multiplicity to input events means several users can send input messages to the system at the same time. We also added the <<active>> stereotype to the user actor, stating that when the users are in a spontaneous and active state that they can send input messages to the system as they wish. The reactive system must be able to handle concurrent requests sent by different users.

5.3 Concept Model

The *Concept Model* is a subset of the *Domain Model*. The *Domain Model* captures all concepts within the domain of the problem, such as all classes and their relationships, external actors and their communication paths with the system. The *Concept Model* offers insight into the problem domain, and excluded those objects, classes and relationships that belong to the environment. In our case, the concept model provides a description of the conceptual system state of the Auction System represented as classes, attributes and associations between classes.

5.3.1 Building the Model

To construct the concept model, one must first brainstorm a list of candidate classes. Real entities such as people, organizations, places, and physical objects can be considered as candidate classes. It is also possible to use abstract concepts as candidate classes. In the Auction System, key concepts or objects can be identified by going through all use cases and highlighting all nouns. From the Buy Item on Auction use case (figure 7), we extracted the classes `Customer`, `Auction`, `Bid`, `Account`, and relationships such as `Makes` and `JoinedTo`.

The complete concept model for the Auction System is shown in Figure 9. The system class name is called *AuctionSystem.*

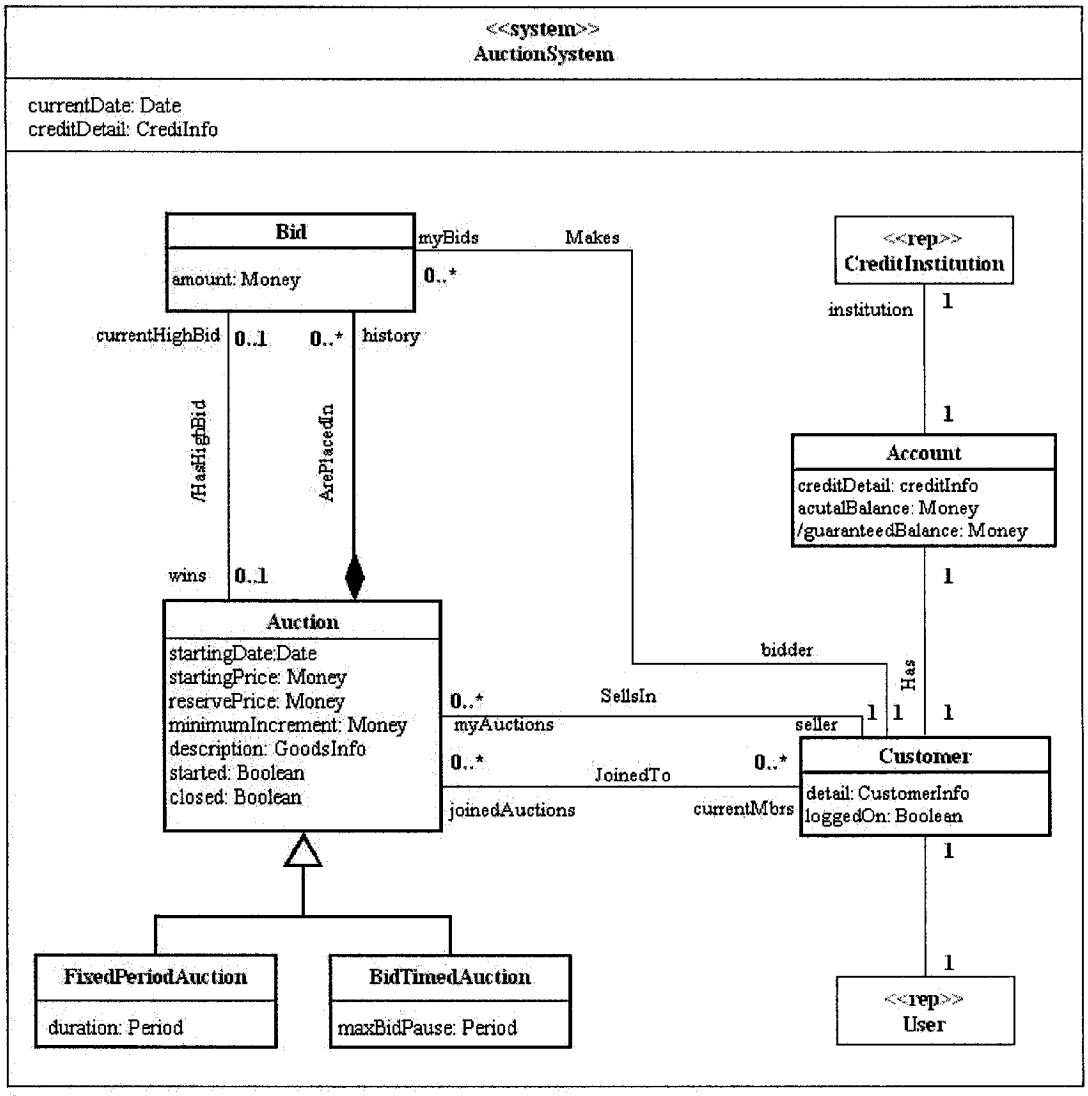

Figure 9. Concept Model of the *Auction System*

Note the concept model itself is not extended to address concurrency. However, it provides the base to identify possible shared concepts at a later stage. The concept model of the *AuctionSystem* system consists of six (normal) classes: *Auction*, *FixedPeriodAuction*, *BidTimedAuction*, *Bid*, *Customer* and *Account*, two <<rep>> classes: *User* and *CreditInstitution* that represent external actors, and several (non-composition) associations.

- Auction and Bid:

The *Auction* class contains bids that are made for the goods on sale in an auction. The *HasHighBid* association is derived: it stands for a link between an auction and its highest bid. *Auction* is an abstract class that has two subclasses: *FixedPeriodAuction* and *BidTimedAuction*. These two classes inherit all attributes and associations of the *Auction* class.

- Customer:

The *SellsIn* and *JoinedTo* associations link customers to the auctions that they sell goods in and that they are joined to, respectively. The *Makes* association links customers to their bids. The *Has* association links customers to their accounts. As indicated by the multiplicity for customer and for account, one customer has one account.

- Account:

The *Account* class contains two attributes and one derived attribute: *creditDetails* represents the information needed by a credit institution to perform a transfer into or out of the bank account of the associated customer. *actualBalance* represents the amount of credit that the associated customer has with the Auction System. The *guaranteedBalance* is a derived attribute that will be discussed in the section 5.3.2.

The concept model also shows two system-wide attributes, *currentDate* and *creditDetail*, which hold the information needed by a credit institution to perform a transfer into or out of the bank account of the enterprise owning the Auction System. This information will be needed when depositing the commissions of the auctions.

5.3.2 Derived Constraints and Attributes

Some concepts that are relevant cannot be expressed in UML. We can, however, specify them by using additional OCL constraints.

For instance, the invariant *allPositiveBalancesForCusts* states that account balances must not drop below zero:

context: Account **inv** allPositiveBalancesForCusts:
self.actualBalance ≥ 0;

The invariant *onlyActiveAuctsHaveMbrs* states that only auctions that have been started but are not closed can have links to customers via the *JoinedTo* association.

context: Auction **inv** onlyActiveAuctsHaveMbrs:
not self.started **or** self.closed **implies**
self.currentMbrs→isEmpty ();

A customer can only be joined to an auction if he or she is logged on:

context: AuctionManager **inv** loggedOutCustsAreNotJoinedToAnyAuct:
self.customer→
forall (c | not c.loggedOn **implies** c.joinedAuction→isEmpty ());

The concept model of the Auction System defines a derived attribute and a derived association. The association *HasHighBid*, which links an auction to its current highest bid, is defined in OCL as follows:

context: Auction **inv:**
self.currentHighBid =
self.history → any(b| b.amount = self.history → max())

The textual form translation of the association is: The current high bid (named *currentHighBid*) of an auction is equivalent to the bid that has the maximum amount of all the placed bids on the auction.

The *guaranteedBalance* attribute of the *Account* class is a derived attribute. It can be regarded as an invariant throughout the operation. Its definition in OCL is as follows:

context: Account **inv:**
self.guaranteedBalance = self.actualBalance – self.myBids →
select (b| b.wins → exists(a| **not** a.closed)).amount →sum()

The *guaranteedBalance* attribute stands for the maximum amount of money that the customer has available in his/her account for bidding. It is calculated by subtracting from the customer's actual balance all outstanding high bids he/she has in all active auctions.

5.4 Protocol Model

The *Protocol Model* uses the state diagrams of UML. It specifies the sequence that the events are to be sent to the system under development. It can also be extended by adding the <<concurrent>> stereotype to the model in order to record the inherent concurrency of the system.

The Auction System is a highly dynamic system, featuring competitive and collaborative concurrency. It stems from the fact that a customer can participate in multiple auctions simultaneously, and that the system must be able to serve multiple customers. In order to describe the concurrency of such a system, two partitioning techniques are adopted. One is called *divide-by-actor*, and the other is called *divide-by-collaboration.*

The divide-by-actor technique describes the interaction protocol between the system and each actor type separately by using a composite state, which is referred to as an actor-activity-state. For the purpose of naming convention, such states take an ...Activity suffix. After identifying all actor types, each actor-activity-state will be given a multiplicity that matches the number of possible concurrent instances for each actor type. The partitioning results in *actor-activity-states.* In the concurrent sense, the states are conjoined with each other to form the protocol model for the system. As a result, this way ensures establishing all inherent concurrency for the system, and capturing all events generated on the reception of messages from actors.

The *divide-by-collaboration* technique is used in some systems where the interaction between the system and each actor is fairly simple, but the collaborative behavior is complex. The technique specifies the interaction protocol between the system and its actors in terms of distinct types of collaboration between them. Again, as a naming convention, the collaboration is represented by a state named with a ...*View* suffix, which stands for a *view-state.* The *view-states* do not necessarily represent inherent concurrency of the system. Instead, they restrict the concurrent behavior of collaborating independent actors. This is different from the *actor-activity-states.*

The protocol model for the *AuctionSystem* system is shown in Figure 10. It uses the partitioning techniques we mentioned above and therefore it consists of two concurrent states: *UserActivity* and *AuctionView.*

The *UserActivity* state models the protocol that represents the interactions between the system and the *User* actors. As we mentioned in the concept model, *User* actor here includes User of the system and the Credit Institution. The *AuctionView* state models the protocol for the auctions, representing the collaboration between various parties during the auctions. Since the number of opening auctions is dynamic, the Auction System is auto-concurrent, meaning that

a dynamic number of auctions are opening concurrently. Similarly, the number of customers participating in an auction is dynamic. Thus the *UserActivity* and *AuctionView* states are modeled as *auto-concurrent* states, as marked by the multiplicity notation "0..*". Each concurrent state represents the interaction between the system and an individual *User* actor.

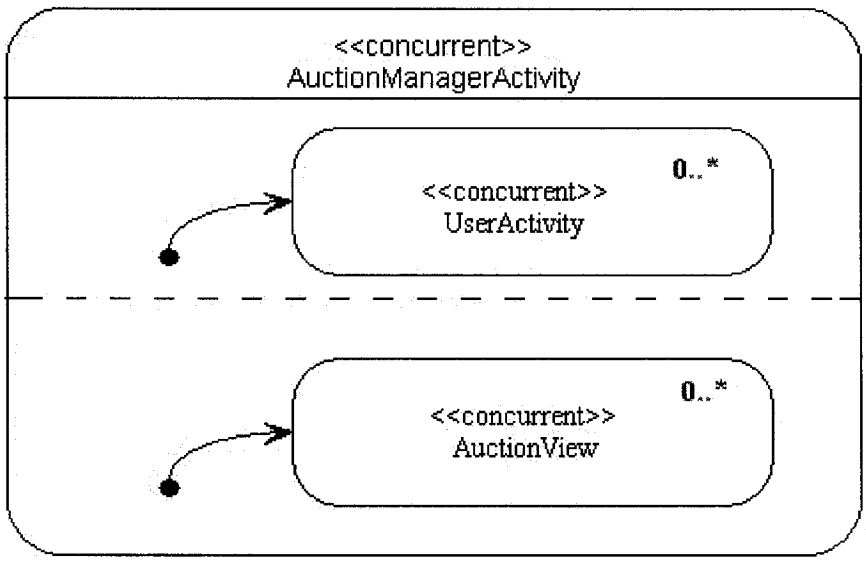

Figure 10. Protocol Model of the Auction System

Note the model and its two sub-states, *UserActivity* and *AuctionView*, are marked by the <<concurrent>> stereotype. The <<concurrent>> stereotype notation is an extension to the Fondue protocol model. It highlights the fact that events in the model can be invoked concurrently.

Figure 11 shows the sub-states of the *UserActivity* auto-concurrent state.

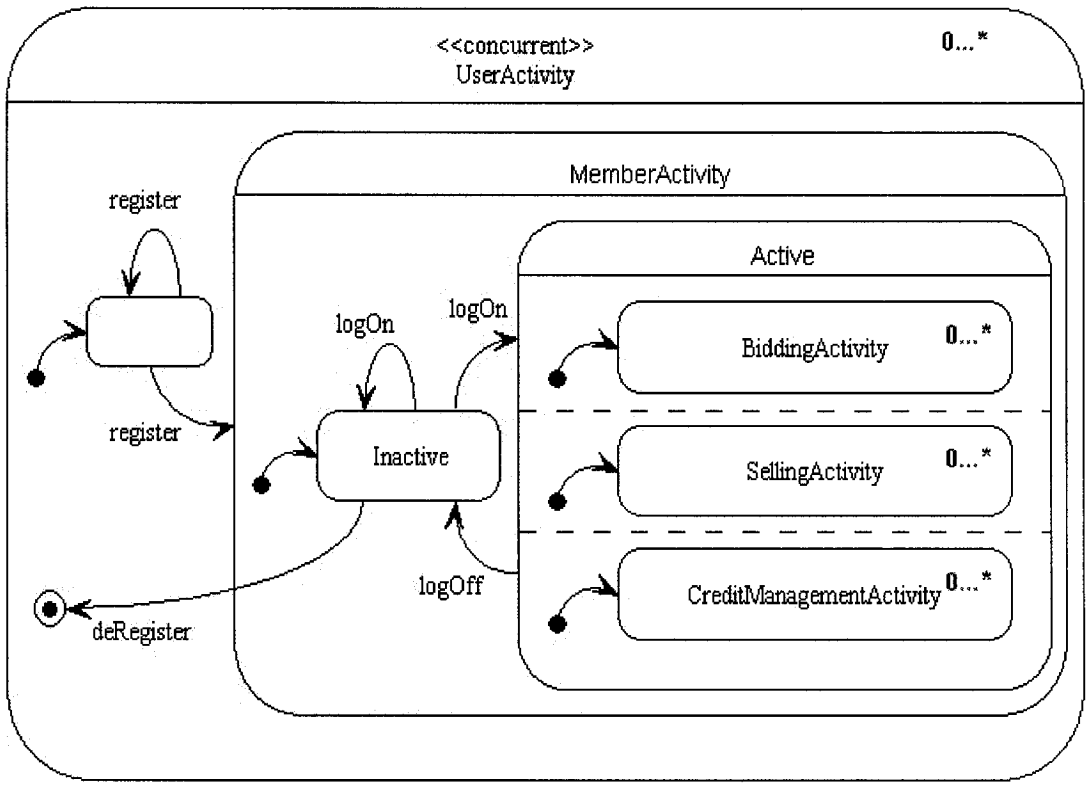

Figure 11. The UserActivity Auto-concurrent State

The above figure indicates that a user must register before he/she start to use the system. Once registered, a user (becomes a customer) also needs to log on before he/she is able to participate the bidding, selling or credit management activities. In the real life, because a customer cannot physically perform multiple activities in parallel, the Active state is not concurrent. Likewise, although a customer is allowed to log off the system at any time after he/she has logged on, he/she cannot physically log off while placing a bid (or perform any other operation) at exactly the same time.

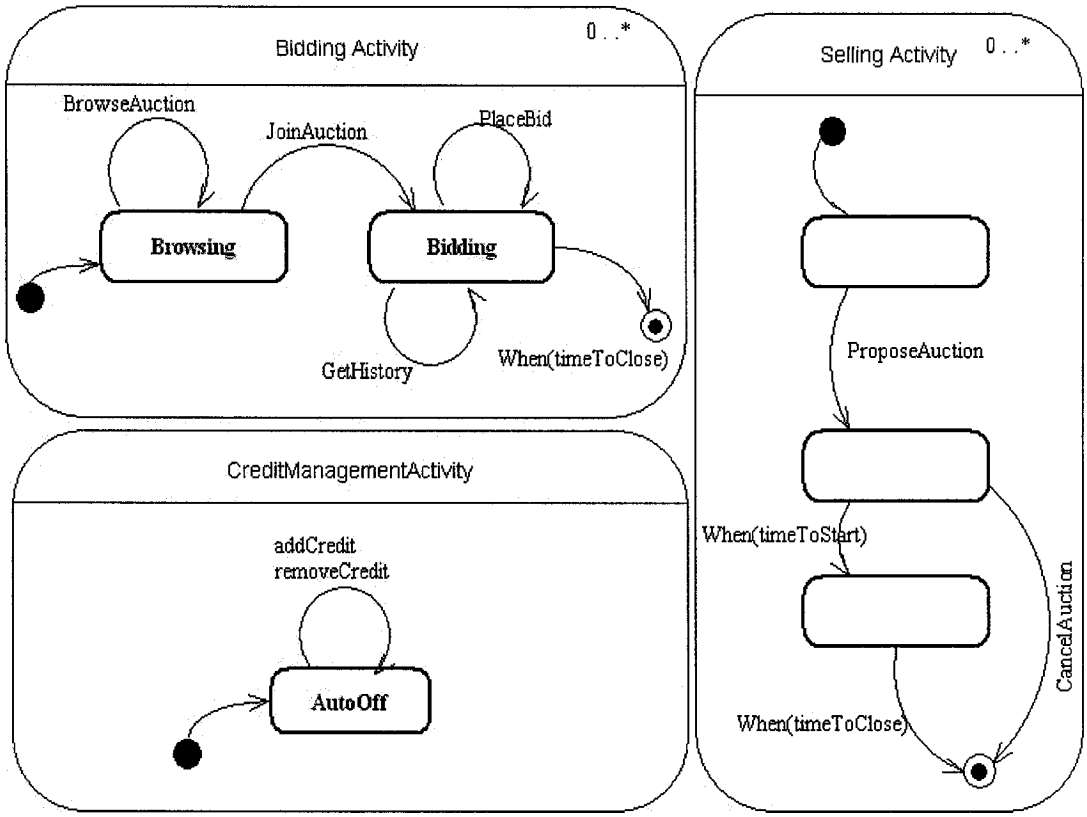

Figure 12. BiddingActivity, SellingActivity and CreditManagementActivity

Figure 12 shows the sub-states of the *BiddingActivity*, the *SellingActivity* and the *CreditManagementActivity*. The *BiddingActivity* is auto-orthogonal. This means that the customer can be participating in possibly many different auctions at any one time. The *SellingActivity* state is also auto-orthogonal. Similarly to *BiddingActivity*, this means that a customer can be selling in possibly many auctions at any one time. The *CreditManagementActivity* state is orthogonal, for a Customer has only one account to manage during the auction.

There are transitions in systems that are triggered by time events. In the Auction System, the *BiddingActivity* and *SellingActivity* states contain transitions with `when(timeToStart)` and `when(timeToClose)` time events. They are defined as Boolean expressions that evaluate to true when the auction in question is started or closed. The OCL definition of `timeToStart` is presented

below. In this context, `a` is the auction object in question and `self` is the system object. `timeToStart` is true if the start date of the auction has arrived.

declares: timeToStart: Boolean **Is** a.startingDate >= self.currentDate;

Figure 13 shows the *AuctionView* auto-concurrent state. It precisely captures the concurrency that concentrates on the collaborations between actors who send input events to the system. The figure shows the accepted input events from the auction point of view. Joining, bidding and reading (getting history) are only permitted when the auction has started and until the auction closes.

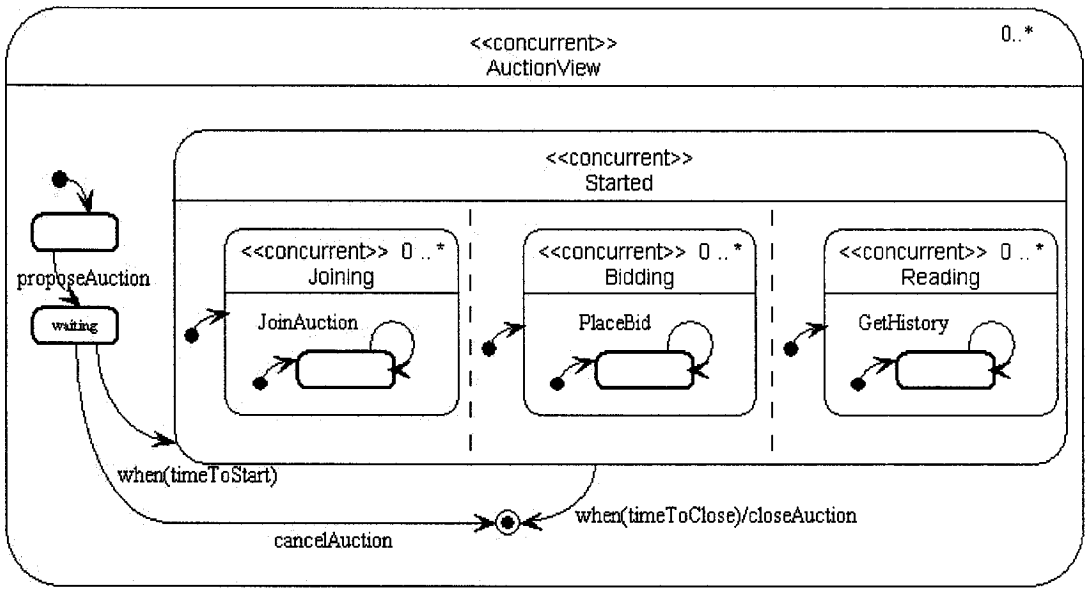

Figure 13. The AuctionView Auto-concurrent State

The closing time of an auction depends on the kind of auction. For this reason, the OCL expression for `timeToClose` contains an if-then-else construct. In the case of fixed-period auctions, `timeToClose` is true if the starting date plus the fixed duration has arrived. In the case of bid-timed auctions, `timeToClose` is true if the state Bidding of *AuctionView* has been active for longer than the maximum pause allowed in bidding. In order to express this part of the condition, the definition makes use of `actTime`, a predefined attribute of

every state, which measures the time that has elapsed since last entering the state. In the model, the auto-concurrent state Bidding is reentered (and `actTime` is restarted) every time a bid is placed.

declares:

```
timeToClose: Boolean Is
    if a.oclIsTypeOf (FixedPeriodAuction) then
        a.startingDate + a.duration <= self.currentDate
    else
        AuctionView::Bidding.actTime > a.maxBidPause
    endif;
```

5.5 Reference Table

The protocol model has shown us the sequence of the operations. Since we have extended the protocol model by adding the <<concurrent>> stereotype, we can also extract essential information about what operations might be executed concurrently. According to this, we can create a reference table to give the developer a clear view of those concurrent operations.

General Reference Table The protocol model of a dynamic system contains information about concurrency. Any *Activity* or *View* sub-state of the protocol model indicates the possible concurrency of input events. In our case, we need to look at *AuctionView* and *UserActivity*.

The purpose of the general reference table is to extract the concurrency information from the protocol model, and then lists potential concurrent input events in a table by grouping them according to the *Activity* and *View* sub-states. At a later stage, the developer will be able to use the table as a reference to look up possible concurrency information when he/she is creating the concurrent operation schema (see section 5.6.3). Table 2 is the general reference table for the Auction System. The input events are grouped by *AuctionView* and *UserActivity*.

Note that all input events for the *AuctionView,* including events that belong to its sub-state *Started,* focus on collaborations on the auction itself, so we do not need to divide them into smaller groups to further examine their potential concurrency status respectively.

The *UserActivity* has sub-states, namely, bidding activity, selling activity and credit management activity. All input events concentrate on different types of activities. They might not affect each other even in the case when they are invoked concurrently. Hence, to examine their possible concurrency status, we would be better to divide them into smaller groups by the nature of the activities.

<table>
<tr><th>Protocol Model</th><th colspan="2">Concurrent Input Events</th></tr>
<tr><td rowspan="5">AuctionView</td><td colspan="2">proposeAuction</td></tr>
<tr><td colspan="2">cancelAuction</td></tr>
<tr><td colspan="2">joinAuction</td></tr>
<tr><td colspan="2">closeAuction</td></tr>
<tr><td colspan="2">getHistory</td></tr>
<tr><td></td><td colspan="2">placeBid</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td rowspan="10">UserActivity</td><td rowspan="2">Registration</td><td>register</td></tr>
<tr><td>deRegister</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td rowspan="2">Logging</td><td>logOn</td></tr>
<tr><td>logOff</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td rowspan="2">Credit</td><td>addCredit</td></tr>
<tr><td>removeCredit</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">browseAuction</td></tr>
</table>

Table 2. General reference table for Auction System

Table 2 shows that concurrent input events of auctions include `proposeAuction`, `cancelAuction`, `joinAuction`, `closeAuction`, `getHistory` and `placeBid`. Concurrent input events of user activity include registration related (`register` and `deRegister`), logging related (`logOn` and `logOff`), credit related (`addCredit` and `removeCredit`) and browsing (`browseAuction`). The input events that belong to both *UserActivity* and *AuctionView* are not repeated in the *UserActivity* part of the table.

Specific Reference Table The general reference table groups potential concurrent input events, but it is not specific enough to indicate possible concurrent invocations between every two input events. For instance, although all operations in *AuctionView* are grouped together, suggesting they have the possibility to run concurrently, there could be two of these operations that will never be executed concurrently. Obviously, such information is needed when the developer is working on the concurrent operation schema (see section 5.6.3).

To further specify the occurrence of concurrent invocations of the input events, we can create separate reference tables for the *View* part and the *Activity* part of the general reference table, respectively. The concurrency information provided in the separate reference tables will be more specific.

The specific reference table can be created with the help of the protocol model. On one hand, the protocol model naturally specifies the execution sequence of the input events. On the other hand, with our extension to add <<concurrent>> stereotype to the model, concurrency information for the input events can be extracted.

In our case, we should create separate specific reference tables for *AuctionView* and *UserActivity*. For *UserActivity*, smaller specific tables can be created according to different types of activities. We will take the table for *AuctionView* as the example.

The specific reference table focuses on input events to a specific auction. For all the input events in *AuctionView*, we want to see if one event could be invoked concurrently with other events. An input event leads to a system operation.

For each auction, there is only one item for sell, therefore only one `proposeAuction` operation is needed, i.e. it is not possible to have two `proposeAuction` operations running concurrently. From the protocol model for *AuctionView*, the sequence indicates the `proposeAuction` operation is executed before all other operations (see figure 13 in section 5.4). Therefore, `proposeAuction` will never run concurrently with any other operation.

The <<concurrent>> stereotype for the *Started* sub-state of *AuctionView* in figure 13 indicates that the *Started* sub-state is auto-concurrent. Thus, the three operations that belong to the *Started* sub-state, namely, `joinAuction` and `placeBid` and `getHistory`, could be executed concurrently.

In addition, the joining, bidding, reading (getting history) activities themselves are auto-concurrent, respectively (see figure 13). For instance, multiple customers could join the auction at the same time. Hence, one `joinAuction` operation could be executed concurrently with another `joinAuction` operation invoked by a different customer. Similarly, multiple `placeBid` operations could be executed concurrently, and multiple `getHistory` operations could be executed concurrently.

According to the auction rules, once an auction is started, it can not be cancelled. As indicated in figure 13, a seller can either start an auction or cancel an auction after proposing the auction. Hence, a *cancelAuction* operation cannot be executed concurrently with any other operation. Since only the seller of the auction can cancel it, it is not possible to have multiple `cancelAuction` operations run concurrently.

The `closeAuction` operation is a time-triggered event. It could fire anytime during the auction, so it could run concurrently with `joinAuction`, `placeBid` and `getHistory`.

Once the analysis for potential occurrences of concurrent input events is finished, the specific reference table can be created easily. Table 3 is a specific reference table for concurrent input events of the auction, i.e. the *AuctionView*. For space reason we cannot list the full names of all the operations. The abbreviations and their corresponding operations are:

Propose: `proposeAuction` Join: `joinAuction`

Bid: `placeBid` History: `getHistory`

Cancel: `cancelAuction` Close: `closeAuction`

	Propose	**Join**	**Bid**	**History**	**Cancel**	**Close**
Propose	-	N	N	N	N	N
Join	N	Y	Y	Y	N	Y
Bid	N	Y	Y	Y	N	Y
History	N	Y	Y	Y	N	Y
Cancel	N	N	N	N	-	N
Close	N	Y	Y	Y	N	-

Table 3. Specific Reference table for the AuctionView

In table 3, the symbols Y, N and – are used to indicate whether concurrency is possible between two input events in a certain auction. An occurrence of Y means two operations could happen concurrently. An occurrence of N means two operations cannot happen concurrently. An occurrence of – means concurrency is not applicable here.

5.6 Operation Model

The *Operation Model* specifies effects of the system operations on the conceptual state specified in the concept model. The generated output messages are also specified. To build the operation model, a separate operation schema has to be written for each operation. The operations for the *AuctionSystem* system include: `register`, `deRegister`, `logOn`, `logOff`, `joinAuction`, `proposeAuction`, `cancelAuction`, `placeBid`, `browseAuction`, `getHistory`, `addCredit`, `removeCredit`, and the time-triggered `closeAuction`. To illustrate how to create an operation schema, we will use the `placeBid` operation as an example.

Firstly, we will have a simple, sequential version of operation schema for `placeBid` ignoring concurrency issues and focusing on the functionality of the operation only. Based on the sequential version, we will then develop a more complex, concurrent operation schema for `placeBid`. The concurrent version shows the execution effects on the system when concurrency has been taken into account.

5.6.1 The Sequential Version

To create a sequential operation schema for `placeBid` (see Figure 14 at the end of this section), we need to apply the operation schema template to this operation. The template has been introduced in the Fondue overview (see Figure 2 in section 3.3.4). Let's look into details about how to create the schema.

The first line of the schema with the key word *Operation* specifies the context of the operation. In our case, the system class name is *AuctionSystem*, which can be found in the concept model. Then the system class name is followed by the name of the operation, i.e. `placeBid`, and the parameters. The parameters should indicate the concepts that are directly involved in the operation. In the case

of `placeBid`, a customer specifies a bid amount to place a bid in an auction. Hence, we take *customer*, *bid amount* and *auction* as the parameters. It is possible for us to obtain a customer's account information from *customer*, so we do not need *account* to be a separate parameter.

The parameter types are usually OCL types, including the base types of most programming languages such as integers, strings and self-defined OCL types. In any case, we assume either the GUI of the actor is powerful enough to gather the parameter values directly and send them with the event, or, alternatively, the actor sends parameter values that can be interpreted by our system and mapped to the appropriate types. For example, an operation that takes a customer object as a parameter might be invoked from an actor, passing the name of the customer in the form of a String instead of a customer object.

The *Description* clause briefly describes the content of the `placeBid` operation.

The *Scope* clause lists all the classes and associations that are used in the precondition and post condition. These classes and associations come from the concept model of the system. In our case, the classes *Auction*, *Bid*, *Customer* and *Account* are involved. They are connected through the associations *ArePlacedIn*, *Makes*, *Has*, and *HasHighBid*.

The *Message* clause declares the possible output events resulting from the execution of the operation. For each message, its type and destination actor must be specified. In our case, if `placeBid` is not successful during the operation, an *invalidBid_e* message will be propagated to the calling customer.

The *New* clause declares all the names that refer to the new objects that are potentially created by the operation. These objects will be instantiated in the *Post* clause using the predefined operation *oclIsNew*. In our example, if the

`placeBid` operation is successful, a new bid object will be created, so we declare *newBid* as the potential new object.

The *Pre* clause declares the precondition that has been assumed for the operation. From the rules of the auction, we know that in order to place a bid, the auction must have started and not been closed yet, and the customer that wants to place the bid must have joined the auction. Therefore we consider this as the precondition and translate it into OCL expressions.

The *Post* clause defines the required state of the system at the end of the operation, also using OCL syntax. Only conceptual system state *changes* must be mentioned here, any unmentioned state remains the same. This is called the *minimum set principle* by Sendall [16].

Since we have done the requirement analysis (see section 4.2), we know that a successful bid has to be valid first (condition 1, a valid bid means the bid amount is no less than the current high bid plus the minimum increment), then the customer that is placing the bid must have enough money (condition 2, the customer's guaranteed account balance is no less than his/her bid amount for the item in the auction). If the two conditions can be satisfied, the bid is made. Otherwise, if any of these two conditions is not satisfied, the bid is unsuccessful and the system will propagate a message informing the customer that the bid is invalid. We translate these concepts into OCL expressions and they become the post condition in the operation schema.

Figure 14 presents the complete operation schema for the sequential `placeBid` operation.

Operation: AuctionSystem::placeBid (a:Auction, c:Customer,
bidAmount:Money);

Description: A customer requests to place a bid in the given auction: the system must decide whether the bid is valid and if so make the bid the current high bid for the auction;

Scope: Auction; Bid; Customer; Account; ArePlacedIn; Makes; Has; HasHighBid; JoinedTo;

Messages: Customer::{InvalidBid_e};

New: newBid: Bid;

Pre: a.currentMbrs –> **includes**(c) & a.started & **not** a.closed;

Post: **if** bidAmount ≥ a.currentHighBid.amount + a.minimumIncrement **then**

if c.account.guarranteedBalance ≥ bidAmount **then**

newBid.oclIsNew(amount => bidAmount) &

a.bid→ **includes**(newBid) &

c.myBids→ **includes**(newBid)

else

sender^invalidBid(Reason::insufficientFunds)

endif

else

sender^invalidBid(Reason::invalidBid)

endif

Figure 14. Sequential Operation Schema for the `placeBid` Operation

5.6.2 Identifying Shared Concepts

The sequential operation schema for an operation can present us a simple and straightforward view of the functionality of the operation. However, if the protocol model[5] or the reference tables indicate that the operation might be executed concurrently with other operations, a sequential operation schema is not enough to elaborate on our concern for concurrency. In this case, a concurrent

[5] The protocol model here is the one that has been extended with <<concurrent>> stereotype

version of the operation schema must be created. However, before we proceed to create the concurrent operation schema, we must first identify exactly what are the shared concepts that are involved in the concurrent operations.

In our case, the *AuctionView* state of the protocol model (see figure 13) and the reference table for *AuctionView* (table 3 in section 5.4) have indicated that `placeBid` could potentially be executed concurrently with `placeBid`, `joinAuction`, `getHistory` that are issued by other users. Thus, we must create a concurrent operation schema for `placeBid` to address concurrency.

From the *Scope* clause of the sequential operation schema, we are able to see all conceptual states and relations that are accessed by the operation and therefore might be shared. In our case of `placeBid` (see figure14), these accessed concepts include *Auction, Bid, Customer, Account,* and these accessed relations include *ArePlacedIn, Makes, Has, HasHighBid* and *JoinedTo.* Through analysis, we can find that:

(1). Since different customers issue concurrent `placeBid` operations, the *Customer* concept is not shared.

(2). Since each bid is made by a different customer, the *Makes* concept is not a shared.

(3). Since each `placeBid` operation creates a new bid, the *Bid* concept is not shared.

(4). For each new high bid, the *HasHighBid* relation is updated concurrently, so *HasHighBid* is shared.

(5). Since each new bid is inserted into a list of bids of an auction, the *ArePlacedIn* relation is modified and hence it is shared.

(6). As the result of the `placeBid`'s modifying the *HasHighBid* relation, the *guaranteedBalance* of the customer who was previously holding the highest bid is modified. Therefore, the *Account.guaranteedBalance* concept is shared.

(7). When `placeBid` runs concurrently with `joinAuction`, the `joinAuction` operation modifies the *JoinedTo* relation and `placeBid` consults this relation, so *JoinedTo* is shared.

(8). Since there is a time-triggered event `closeAuction`, which can result in closing the auction at any time, the *close* attribute of *Auction* is shared.

Once all the shared concepts have been identified, they will be recorded in a new *Shared* clause of the concurrent operation schema, as illustrated in figure 15.

5.6.3 The Concurrent Version

After identifying shared concepts for potential concurrent operations, we are now ready to transform the sequential operation schema to its concurrent version.

In order to create the concurrent operation schema for an operation, essential changes (especially changes to the **Pre** and **Post** conditions) must be made based on its sequential version. The following paragraphs (identified by **change 1** to **change 4**) explain why we make the changes and how to make the changes.

Change 1. A new clause called *Shared* will be added after the *Scope* clause. The *Shared* clause is needed because it records all shared concepts of the concurrent operations.

Change 2. Some conditions that have been originally stated as a precondition in the sequential operation schema may have to be changed. When operations are executed concurrently, it might not be enough to check a condition at the beginning of an operation. We want to be able to rely on the fact that the condition remains satisfied while performing certain changes.

To emphasize such a constraint, we propose to use the **rely** statement in the concurrent operation schema. The structure of the **rely** statement is **rely** A **then** B **fails** C **endre**, where the words **rely**, **then**, **fails** and **endre** are keywords, and A is a condition and B, C are state changes. The statement asserts that either condition A keeps being true when all state changes specified in B are realized, or, all state changes specified in C will be realized.

Change 3. Since we want to guarantee atomic execution results, an **if** statement (with the structure **if** A **then** B **else** C **endif**) in the sequential operation schema might have to be transformed into a corresponding **rely** statement. This situation arises when the condition is based on shared concepts that might change due to a concurrently executing operation.

Change 4. Because of introducing concurrency and using the **rely** A **then** B **fail** C **endre** statement, extra output messages may need to be added to the post condition, in order to inform the environment about abnormal outcomes due to interference with other operations.

Now let's look at the concurrent operation schema for `placeBid`, shown in figure 15.

(1). A *Shared* clause has been added. This clause records all concepts that `placeBid` shares with operations that execute concurrently. (See section 5.6.2)

(2). The **not** a.closed precondition in the sequential version has been removed and replaced by a **rely** statement in post condition of the concurrent version.

In the sequential version, the **not** a.closed condition is considered as part of the precondition because the sequential operation schema has instantaneous semantics. Since every operation is executed atomically, specifying **not** a.closed in the precondition is sufficient to guarantee that the customer places a bid while

the auction is still active. In the concurrent version, however, the **not** a.closed condition must be ensured while the `placeBid` operation is in progress.

When the **rely** statement is used to emphasize the **not** a.closed condition in the post condition of the concurrent version (marked by ‹*› in figure 15), it means either a bid is successfully placed (all related changes and updates are made) and during all this time the auction is not closed, or an error message is sent to the customer who requested to place the bid.

(3). The two nested **if** statements in the sequential version will be replaced by two nested **rely** statements (‹1› and ‹2› in figure 15) in the concurrent version. The first **rely** statement states that there must be no other `placeBid` operation to modify the current high bid while the auction is accepting a new high bid. The second **rely** statement states that the current high bidder must continuously have sufficient funds according to his/her guaranteed balance.

(4). A new error message called *auctionClosed* will be added to the post condition because of introducing concurrency. This error message is not necessary in the sequential operation schema for `placeBid`. In the concurrent version, however, it might happen that the auction closes while a bid is placed.

Figure 15 shows the concurrent operation schema for `placeBid`.

Operation: AuctionSystem::placeBid(a:Auction, c:Customer, bidAmount:Money);

Description: A customer requests to place a bid in the given auction: the system must decide whether the bid is valid and if so make the bid the current high bid for the auction;

Scope: Auction; Bid; Customer; Account; ArePlacedIn; Makes; Has; HasHighBid; JoinedTo;

Shared: Account.guaranteedBalance; HasHighBid; ArePlacedIn; JoinedTo; Auction.closed;

Messages: Customer::{invalidBid_e};

New: newBid: Bid;

Pre: a.currentMbrs –> **includes**(c) & a.started

Post:

‹*›**rely not** a.closed **then**
‹1› **rely** bidAmount ≥ a.currentHighBid.amount +
a.minimumIncrement **then**
‹2› **rely** c.account.guarranteedBalance ≥ bidAmount **then**
newBid.**oclIsNew**(amount => bidAmount) &
a.bid→ **includes**(newBid) &
c.myBids→ **includes**(newBid)
fail
sender^invalidBid(Reason::insufficientFunds)
endre
fail
sender^invalidBid(Reason::invalidBid)
endre
fail
sender^invalidBid(Reason::auctionClosed)
endre;

Figure 15. Concurrent Operation Schema for the `placeBid` Operation

In summary, after all the changes made from sequential version to concurrent version, the execution effects of the concurrent `placeBid` are:

If the auction stays open (marked by ‹*›), if the bid stays over the current high bid plus the minimum increment (marked by ‹1›), and if the customer has sufficient account balance (marked by ‹2›), then the bid is successful and the bid is made by the system (asserted by the expressions showing the new bid is made).

Otherwise, if any of the conditions change during the execution of the state changes, the system will send an *invalidBid* message to the current customer.

5.7 Summary of Fondue Analysis

We have stepped through the Fondue analysis by using the Auction System as our example. The example demonstrated that the analysis phase should be conducted by building the analysis models one after another. Each model focuses on a different aspect of the system.

The *Environment Model* identifies the system and its external actors. It also shows the message passing between the system and the actors. The *Concept Model* then extracts all classes of the system and connects them by their association relationships. Afterwards, the *Protocol Model* shows the sequence of operations in the system. Finally, operation schemas for each of the operations are created in the *Operation Model*, specifying their execution effects to the system.

In order to address concurrency in the analysis phase, we have extended the models by adding new notations that specifically address concurrency. The multiplicity notations added to external actors and input messages in the environment model means multiple actors can interact with the system concurrently, and several actors can spontaneously send input messages to the system at a given time. A <<concurrent>> stereotype added to a protocol model emphasizes real concurrency in state diagrams. Reference tables listing potential concurrent operations can be created based on the protocol model and are used to help creating concurrent operation schemas in the operation model. A concurrent operation schema can *only* be created after its sequential version has been built and all related shared concepts have been identified.

Chapter 6. Design

This chapter covers design and implementation. Moving from object-oriented analysis to design means we have to map the conceptual state to objects. In other words, in the design phase we have to determine how the functionality specified during analysis is to be provided by the system, by means of interacting objects. The output of the design phase is like a devised blueprint satisfying the requirements defined in the analysis phase.

6.1 Identifying objects

In general, migrating from analysis to design results in that some concepts may be implemented using several objects, or, alternatively, some concepts may be implemented as attributes of classes. The system efficiency is affected by how we identify the objects. A well designed system should have a proper decomposition on the granularity of objects. A system with too fine-grained decomposition might become hard to analyze because it could have thousands of objects with high coupling. Such a system could also generate huge communication overhead. On the other hand, a coarse decomposition will generate objects with unclear responsibilities and thus unavoidably create bulky architectures. Therefore, a well-designed system should have maximized object coherence and minimized object coupling [17], which lead to proper architecture size and high efficiency. In

our case of the Auction System, we have extracted the initial candidate objects including *Auction, Account, Customer* and *Bid* from analysis. They will be used as design objects to hold the application state.

An additional but critical issue that must be considered in the design of concurrent systems is the *shared state*. Some object-oriented programming languages such as Java [18] and Ada [19], support concurrency by providing monitor objects for consistent access to shared data. This is because mutual exclusion guarantees that state updates that are encapsulated inside a monitor will not be preformed concurrently.

In the Auction System, we have identified the shared concepts when `placeBid` is executed concurrently with other operations. These concepts include: *Account.guaranteedBalance*, *Auction.closed*, *BidHistory*, *HasHighBid*, *ArePlacedIn*, and *JoinedTo* (see Fig. 15). Since one of our concerns in the design phase is to allow maximum concurrent execution (for optimization reason), each shared concept must be represented by at least one object.

6.2 Sequential Interaction Model

Just as we did during analysis, we suggest developing a sequential design for each operation first, focusing on the functionality. In a second step, concurrency issues will be addressed. (See section 6.3)

The *Interaction Model* shows how the design objects interact at run-time to provide the behavior specified in the operation schema. It is usually presented in form of an object interaction diagram. In the diagram, the object that receives the external stimulus is called the *controller*. It is responsible for executing the required state changes, or to further delegate responsibilities to additional *collaborator* objects. Since the interaction is conducted by way of communication between objects, the messages for communication and their parameters are chosen. Then they are added to the interfaces of the corresponding objects. In addition to

the objects representing the application state, if there are new objects that represent abstractions of computational mechanisms but are not identified during analysis, it is often necessary to introduce them into the design.

Before we construct the interaction diagram for the Auction System, several decisions about how to implement some concepts that are identified in analysis have to be made.

In terms of the guaranteed balance, we have decided to actually withdraw the money from a customers account when he/she places the bid, and to deposit the money back to the account if ever someone else places a higher bid later on. Using this technique, the actual balance of the account corresponds to the guaranteed balance.

Also, we decided to pass the customer information as a parameter in the initial `placeBid` call to the controller. Because the customer knows his/her account information, we can get reference of the current customer's account from the customer information, instead of passing account as an additional but redundant parameter.

The concept of the bid history is also realized in the design. In analysis, the bid history is reflected as an association from *Auction* to *Bid*. In the design, we realized the concept by using an *insertBid* method, which is a method of the *BidHistory* class. Every time a successful bid is made, the *Auction* calls the *BidHistory* to add a new bid to its list. The *BidHistory* accomplishes the task by calling the *Bid* class to generate a new *Bid* object with details (as we have mentioned before) and then insert the new bid into the bid history.

The sequential interaction diagram of the `placeBid` operation is shown in Figure 16.

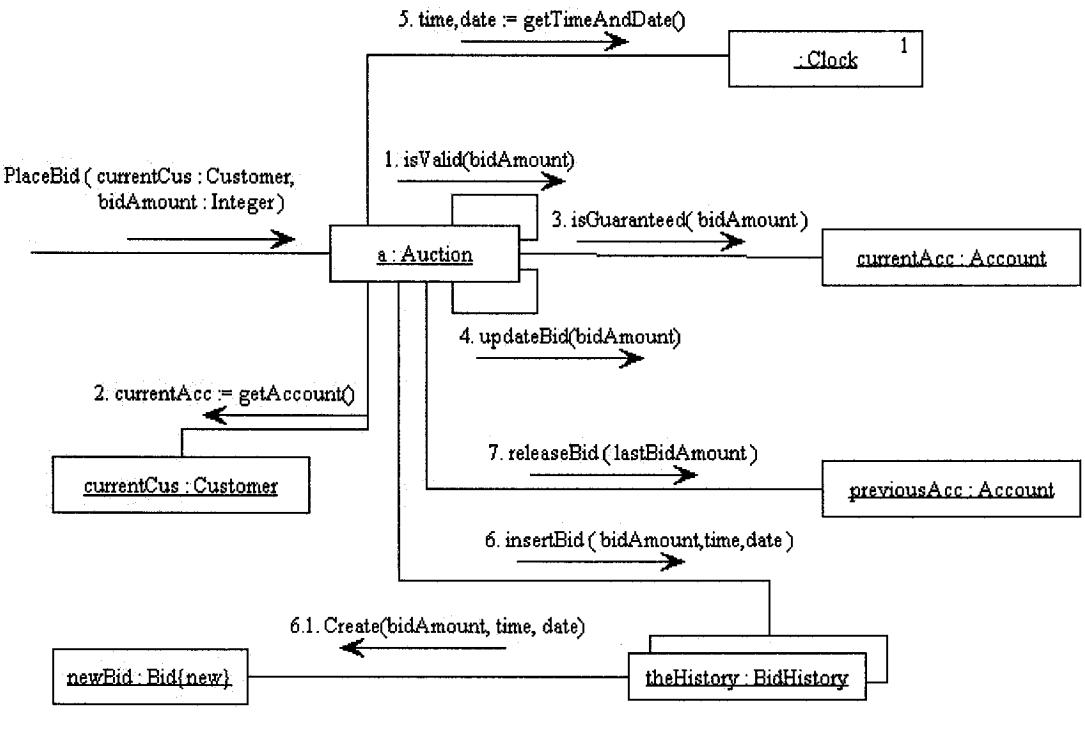

Figure 16. Sequential interaction diagram for placeBid

We can see from the figure that the auction object is the controller, for it receives the initial method call. The initial method call also passes the parameters *currentCus* and *bidAmount* to the controller. The parameter *currentCus* is of type Customer that stands for the current customer object. The current customer object contains all information of a customer, such as user id and the account information of the customer that is placing the bid. The parameter *bidAmount* is of type Integer, which stands for the amount the customer bids on the auction. (For simplicity purpose, we assume the bids have no decimal part.) The ordered numbers in the figure represent the execution sequence of the operations.

Firstly, we check if the bid is valid, i.e. if the bid is higher than or at least equal to the current bid plus the minimum increment. If this fails, an exception is propagated back to the caller.

Secondly, we acquire the customer's account information. The auction object makes a method call to the current customer, then the *currentAcc* object is returned. This object stands for current account, which is the account of the customer that is placing the bid.

Thirdly, we check that whether or not the user has enough money in his/her account (see if the user's actual balance is greater than or equal to the user's bid amount). If yes, the amount is withdrawn (the actual balance is modified by deducting the bid amount from the user's actual balance). In case of any failure, an exception is propagated to the caller.

Fourthly, the current high bid is updated.

Afterwards, the controller calls the *Clock* (which stands for the system clock) to get time and date of the bid. Note there is only one clock in the entire system.

In order to keep a record of all bids associated with an auction, every execution of `placeBid` instantiates a *Bid* object, initializes the state with the amount, time and date of the bid, and inserts the new bid into an ordered list associated with the auction. As stated in the above figure, the list is *BidHistory*.

Finally, if there has been a previous bid, we deposit the amount of the now obsolete bid back to the account of the previous bidder.

For some methods it makes sense to give more details in form of pseudo code. This is the case for the controller method `placeBid` and its related methods `isGuaranteed` and `insertBid`:

(1). Pseudo code for `placeBid`

```
Operation Auction :: placeBid( currentCus: Customer, bidAmount : integer )
        currentAcc := currentCus.getAccount();
begin
        if  isValid(bidAmount) then
                if currentAcc.isGuaranteed(bidAmount) then
                        updateBid(bidAmount);
                        bidNumber ++;
                        theHistory. insertBid(bidAmount);
                        if  bidNumber > 1  then                 //if not first bid
                                previousAcc.releaseBid(lastBidAmount);
                        endif
                        previousAcc := currentAcc;
                        lastBidAmount := bidAmount;
                else
                        Exception("invalidBid: insufficientFunds");
                endif
        else
                Exception("invalidBid");
        endif
end placeBid;
```

Figure 17. Pseudo code for placeBid

(2). Pseudo code for `isGuaranteed`

```
Operation Account::isGuaranteed( bidAmount : Integer)
        OK : boolean;
begin
        if currentAcc.actualBalance – bidAmount ≥ 0 then
                currentAcc.acutalBalance =
                        currentAcc.actualBalance - bidAmount;
                OK = true;
```

```
        else
                OK = false;
        endif
        return OK;
end isGuaranteed
```

Figure 18. Pseudo code for isGuaranteed

(3) Pseudo code for `insertBid`

```
Operation BidHistory::insertBid(bidAmount : Integer, time : Time, date : Date)
        bidList : Vector;                    //the Vector is like the Vector in Java
begin
        newbid = new Bid(time, date, bidAmount);  //create new bid object
        bidList.add(bidAmount);                   //insert
end insertBid
```

Figure 19. Pseudo code for insertBid

The `placeBid` example demonstrated how the sequential design is conducted in form of an interaction diagram that implements conceptual state changes by interacting objects at run-time. A complete design practice, however, should include design for all of the system operations.

6.3 Concurrent Interaction Model

To deal with concurrency in the design, we must ensure multiple readers / single writer access to all shared resources. In a sense, we want to *isolate* accesses from each other. In the meanwhile, we also have to make sure that the *rely* conditions stated in the concurrent operation schema hold during the execution of the respective state changes they belong to. In our example of `placeBid` operation, for instance, we must ensure that while the bidding is ongoing, the auction does not close, the new high bid is higher than the current bid, and the customer has

enough money in his/her account. In other words, the checking and the updating must be made *atomic*.

If we look at the post condition of the concurrent version of the operation schema for `placeBid` again (figure 20), we will find the concepts of atomic checking and updating have already been specified. .

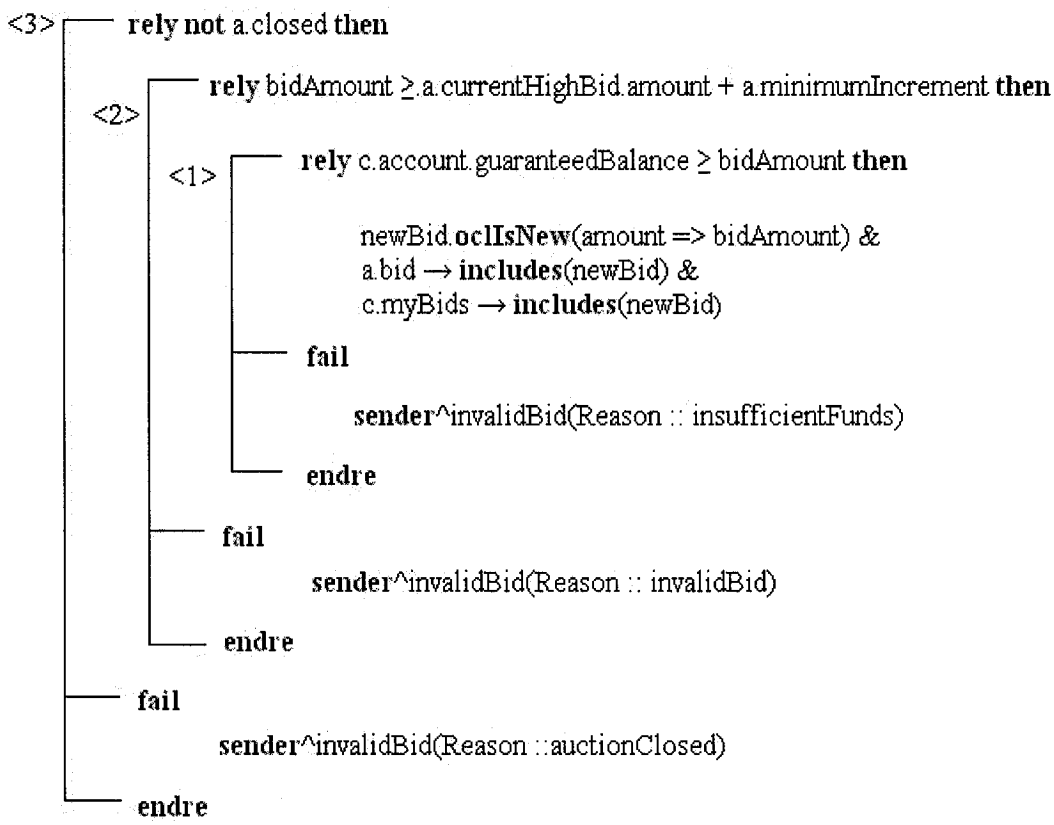

Figure 20. The post condition of concurrent `placeBid` operation schema

It can be noticed that the three **rely** statements in the post condition are structured like nested layers. We indicate this structure by giving numbers to the layers, namely, layer 1, 2 and 3. In each **rely** statement, each pair of keywords **rely** and **endre** can be imagined as a pair of closed brackets.

In the design phase, our solution to ensure the concurrent operation is to set each layer as a *critical region*. In terms of implementation, only one thread can

access the region at any one time. Since the layers are overlapped, we can consider setting the biggest layer as the sole critical region that covers all other smaller critical regions.

There are essentially two different ways of achieving isolation and atomicity with respect to the critical region: using *transactions* or using *monitors and locks.*

6.3.1 Transaction-oriented Design

As we briefly introduced in section 3.4.2, a transaction groups together a set of operations, and gives them the so-called *ACID properties. Atomicity* — either all operations are executed, or none is; *consistency* — transactions move the application from one consistent state to another one; *isolation* — concurrently executing transactions do not see intermediate results of other transactions; and *durability* — state changes made by a transaction are recorded on stable storage. Therefore, if the application modifies sensitive or important data, data that persists, or data that must be kept consistent even in the presence of crash failures, then *transactions* should be used to regulate access to shared objects.

The transaction-oriented design of the `placeBid` operation is shown in Fig. 21. For the sake of providing maximum concurrent execution, the auction state and the current bid have been encapsulated in separate objects.

The entire `placeBid` operation executes from within a transaction. This is shown in the sequence diagram by a gray activation rectangle. At the beginning of the transaction, the *Auction* calls the `isOpen` method to check the auction state. This is a read operation on the auction state, and the atomicity and isolation property of transactions can ensure that the value will not change until the transaction commits. Secondly, the *Auction* validates the bid by the `getBid` method. A valid bid amount must be higher than the current high bid plus the

minimum increment. Thirdly, the bid is deducted from the customer's account by the `isGuaranteed` method if there is enough balance. Fourthly, the current bid of the auction is updated by the `setBid` method. Afterwards, the *Auction* calls the system clock for *time* and *date*, and then passes them together with *bidAmount* as parameters to *BidHistory*. Then the new high bid is created and inserted into the *BidHistory*. At the last step, the account of the previous high bidder is credited. All these operations are executed as part of the transaction.

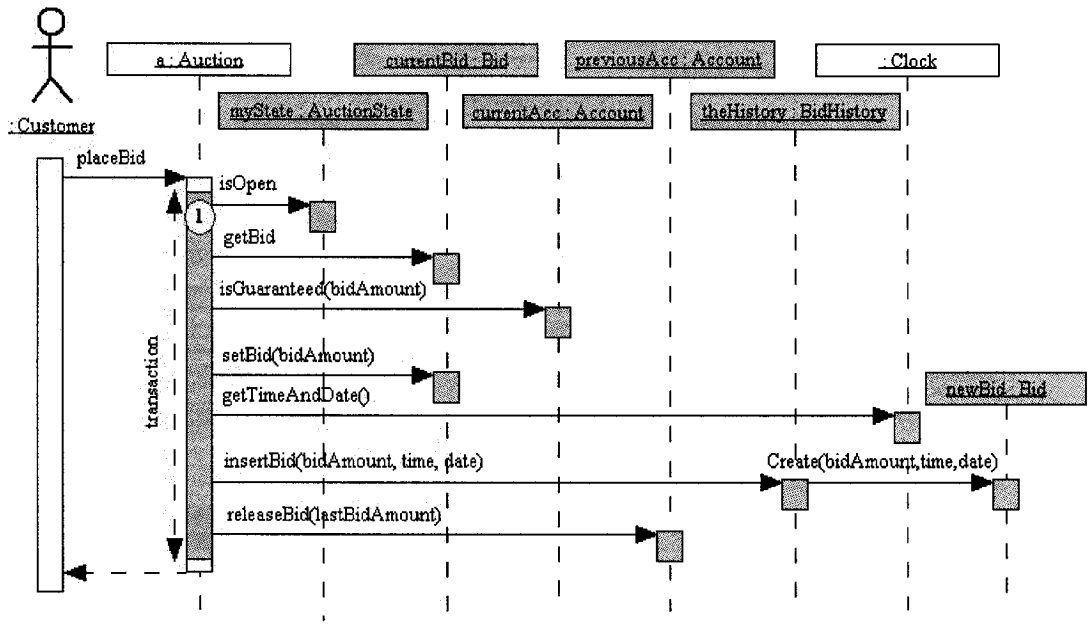

Figure 21. Transaction-oriented Execution of placeBid

During the transaction, *AuctionState*, *Account, Bid* and *BidHistory* are transactional objects, as shown in the diagram by gray object symbols. Since transactional objects have persistent state, they can even survive crash failures. If any one of the conditions is not satisfied, or if any failures occur during the execution of `placeBid`, the transaction will be rolled back, i.e. all state changes made so far will be undone. Using transaction avoids the scenario that a bid is placed without crediting the account of the previous bidder. Because of the isolation property of transaction, no other operations will be affected in case of a

rollback. The *Clock* does not have to be transactional, for the system will just read time and date from the *Clock*.

Interestingly, the actual way of ensuring isolation is still not specified. It depends on the kind of concurrency control that is used by the underlying transaction support. In pessimistic, lock-based concurrency control [20], once shared resources are accessed, they are locked and will not be released until the transaction ends. In our case of Auction System, if a close auction event fires, the `closeAuction` operation would be blocked until all pending `placeBid` operations have terminated and released their locks on the auction state. An alternate means is using optimistic concurrency control [21], such as time-stamp based versioning. In this case, the auction might decide to let the auction close, and abort all concurrently executing `placeBid` operations.

6.3.2 Monitor-based Design

Using transactions requires extensive run-time support, and thus slows down the execution significantly. Alternatively, if an application does not require persistence and tolerance to crash failures, then a simple monitor-based design can provide the same behavior with considerably better performance.

The monitor-based design is very similar to the transaction-oriented design except for few changes. Firstly, the transactional objects are now monitors, i.e. their methods provide multiple readers / single writer semantics (for instance, *synchronized* methods in Java[6], *protected objects* in Ada). Secondly, the atomicity needed for implementing the *rely* conditions is achieved by acquiring *read* or *write locks* when checking the condition (similar to lock-based pessimistic

[6] The current version of Java does not provide monitors, or R/W lock directly. However, by following strict programming conventions, for instance, using classes with synchronized methods and private attributes only, monitors can be programmed. For more detailed information on how to program monitors in Java see [22].

concurrency control[7]). A lock prevents other threads from changing the condition while the operation is being executed. The locks will not be released until after the state changes that *rely* on the condition.

The monitor-based design of `placeBid` is shown in Fig. 22. *AuctionState*, *Account, Bid* and *BidHistory* are now monitors. Again, they are highlighted in the sequence diagram by gray object symbols. This time, the first step would be the system's acquiring a *read lock* (shown in the figure by a dotted gray activation rectangle) when checking the auction status. If there is an attempted concurrent `closeAuction` operation (which would have to acquire a write lock), the acquired *read lock* would block it. Similar to the transaction-oriented design, the *Clock* does not have to be monitor.

By careful analysis, we can find the balance of a customer's account can only grow while the `placeBid` operation is executing[8], because the same customer cannot physically place two bids at the same time, or try to remove credit while placing a bid. In this sense, it is not necessary to acquire a lock to guarantee the balance when accessing the account of the customer that is placing the bid. Consequently, checking and withdrawing the bid amount from the account can be simply performed in one operation, and because the accounts are monitors, the operation itself is atomic. Furthermore, checking and updating the current high bid can be done in a similar way. After the new bid object has been created and inserted into the bid history, we release the read lock on the auction state. Finally, we release the bid of the previous high bidder (if the current bid is successful), or, if the bid is invalid, the money has to be put back on the current bidders account.

---

[7] If read and write locks are not provided by the programming language, the programmer can consider using semaphores to implement it.

[8] The guaranteed balance can grow during the operation placeBid if, for instance, a customer *A* bids in auction *a*, and then, while bidding in auction *b*, a customer *B* overbids *A* in *a*.

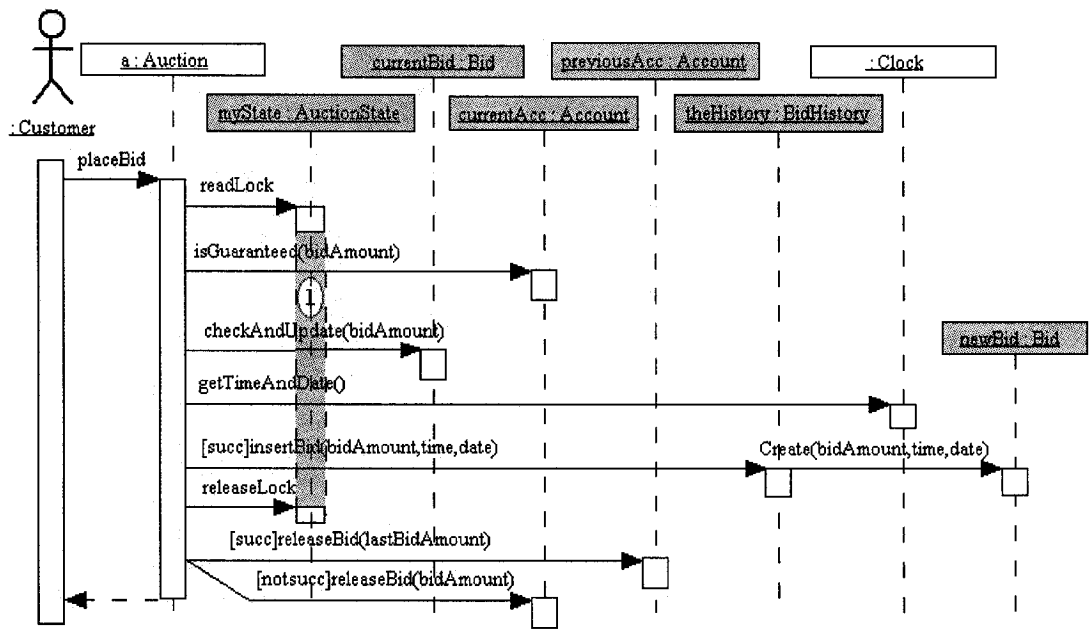

Figure 22. Monitor-based Execution of placeBid

6.3.3 Mapping between Analysis and Design

To present a more straightforward view of how the analysis is mapped to the design, we create table 4 below. The content of operations in the sequential design and the concurrent design are almost the same, but they are implemented in different ways. For example, transaction-oriented design and monitor-based design have different method calls for checking the validity of the bid and different method calls for updating bid.

Here we are discussing the mapping from analysis to the design that refers to both sequential and concurrent versions. Basically, the major concepts that have been identified in the operation schema from the analysis phase are mapped to certain corresponding operations in the design phase. The left column of the table lists the operations in the analysis phase written in OCL expressions. The right column of the table lists the corresponding operations in the design phase.

Analysis – Operation Schema	Design – operations
bidAmount ≥ a.currentHighBid.amount + a.minimumIncrement	**sequential:** isValid() of Auction class
	transaction: getBid() and setBid() of Bid class
	monitor: checkAndUpdate() of Bid class
c.account.guaranteedBalance ≥ bidAmount	**sequential:** isGuaranteed() of Account class
	transaction: isGuaranteed() of Account class
	monitor: isGuaranteed() of Account class
newBid.oclIsNew(amount => bidAmount) &a.bid→ includes(newBid) & c.myBids→includes(newBid)	**sequential:** updateBid() of Auction class, insertBid() of BidHistory class, Create() of Bid class
	transaction: setBid() and Create() of Bid class, insertBid() of BidHistory class
	monitor: checkAndUpdate() and Create() of Bid class insertBid() of BidHistory class

Table 4. Mapping from Analysis to Design

Some operations may not be directly mentioned in an operation schema. For instance, if the current customer's bid is valid and his/her account balance can be guaranteed, and the bid is set as current high bid, then we need to go one step

further to release the previous high bid (if there is any) and return the money to the previous bidder. This is realized by the method `releaseBid` in both of the designs.

6.4 Design Class Model

Once interaction diagrams have been devised for every system operation, it is possible to build the final *Design Class Model.* The design class model depicts the design classes, together with their attributes and their methods. It also includes all the mechanisms to deal with concurrency. In the previous sections we presented two different ways of handling concurrency, one using transactions and the other one using monitors and locks. Monitors / transactional classes can be highlighted using the <<monitor>> or <<transactional>> stereotype. The following sections present the two resulting design class models.

6.4.1 Transaction-oriented Design Class Model

Figure 23 shows the transaction-oriented design class model.

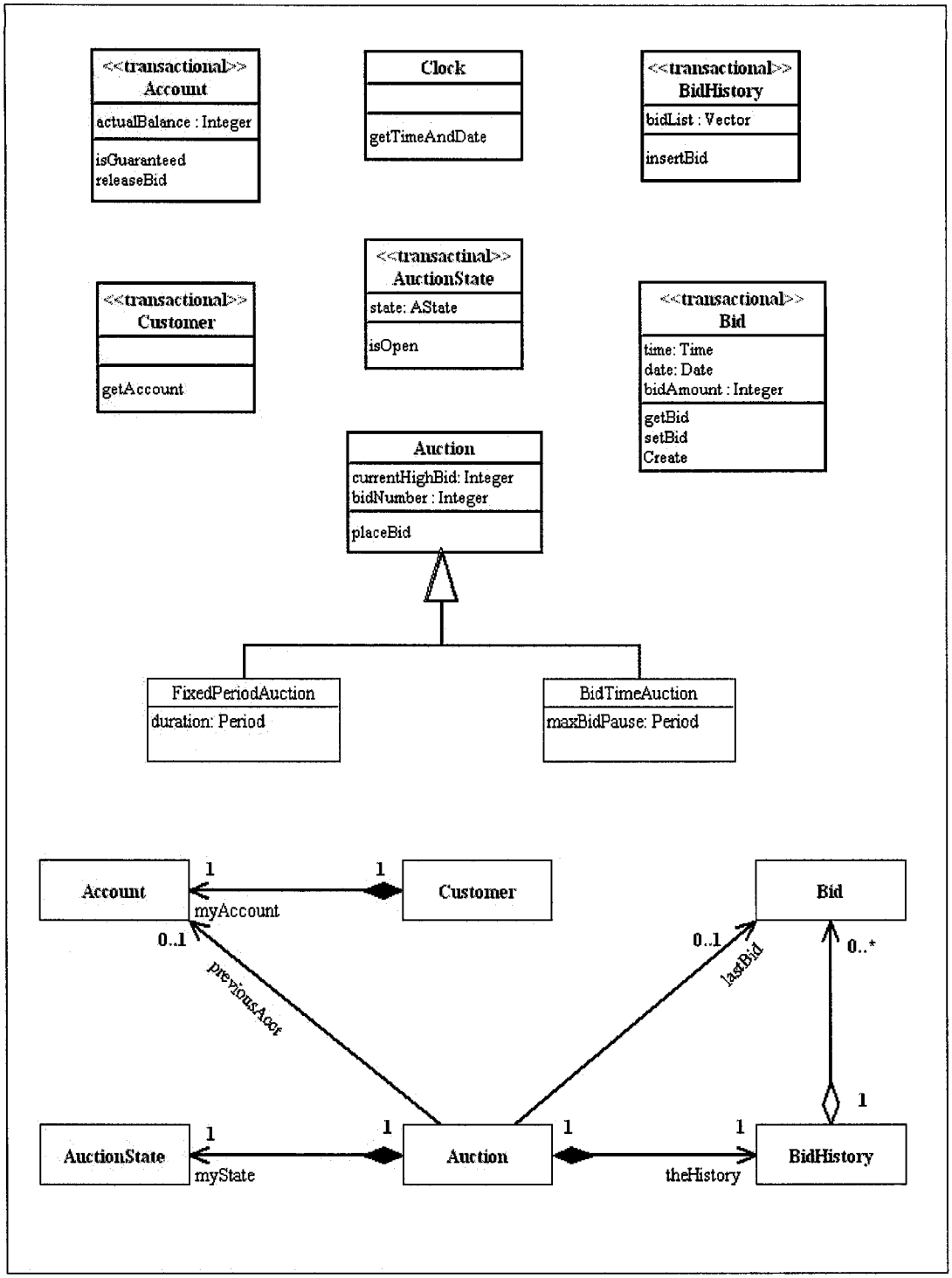

Figure 23. Transaction-oriented Design Class Model

The transaction-oriented design class model shows the static structure of the Auction System, considering only the `placeBid` operation. The upper part

of figure 23 shows all the design classes, their attributes and methods. The lower part of the figure shows the relationships of the classes.

The design classes in the model include *Account, Customer, Auction, AuctionState, Bid, BidHistory* and *Clock.* All the classes, except for the *Auction* class and the *Clock* class, have the transactional stereotype on top of them, which means the objects created from these classes are transactional objects, i.e. their state is stored durably in some database. The attributes and methods of these classes come from the interaction model.

In the *AuctionState* class, the type *AState* is an enumerations type. It's OCL definition is as follows:

type AState **is enum**{open, closed}

The inheritance relationship of the *Auction* class indicates that there are two subclasses of *Auction.* One is *FixedPeriodAuction* and the other is *BidTimeAuction*, representing an auction with fixed period of time length and an auction with predefined maximum pause time length between bids, respectively.

From the lower part of the figure, we can easily see all design classes are connected by navigable associations. A link with an arrowhead in the diagram indicates a navigable association from one class to another. The arrow headed link, together with the role-name at the end of the link, will be implemented as a permanent reference. At the time of implementation, a permanent reference will become an object attribute of the class, which makes a class be able to return a reference to another object. For example, the permanent reference with the name *previousAcct* results in a navigable association from *Auction* to *Account.* Upon implementation, we can use the reference *Auction.previousAcct* to refer to an account object of the *Account* class.

6.4.2 Monitor-based Design Class Model

Figure 24 shows the monitor-based design class model.

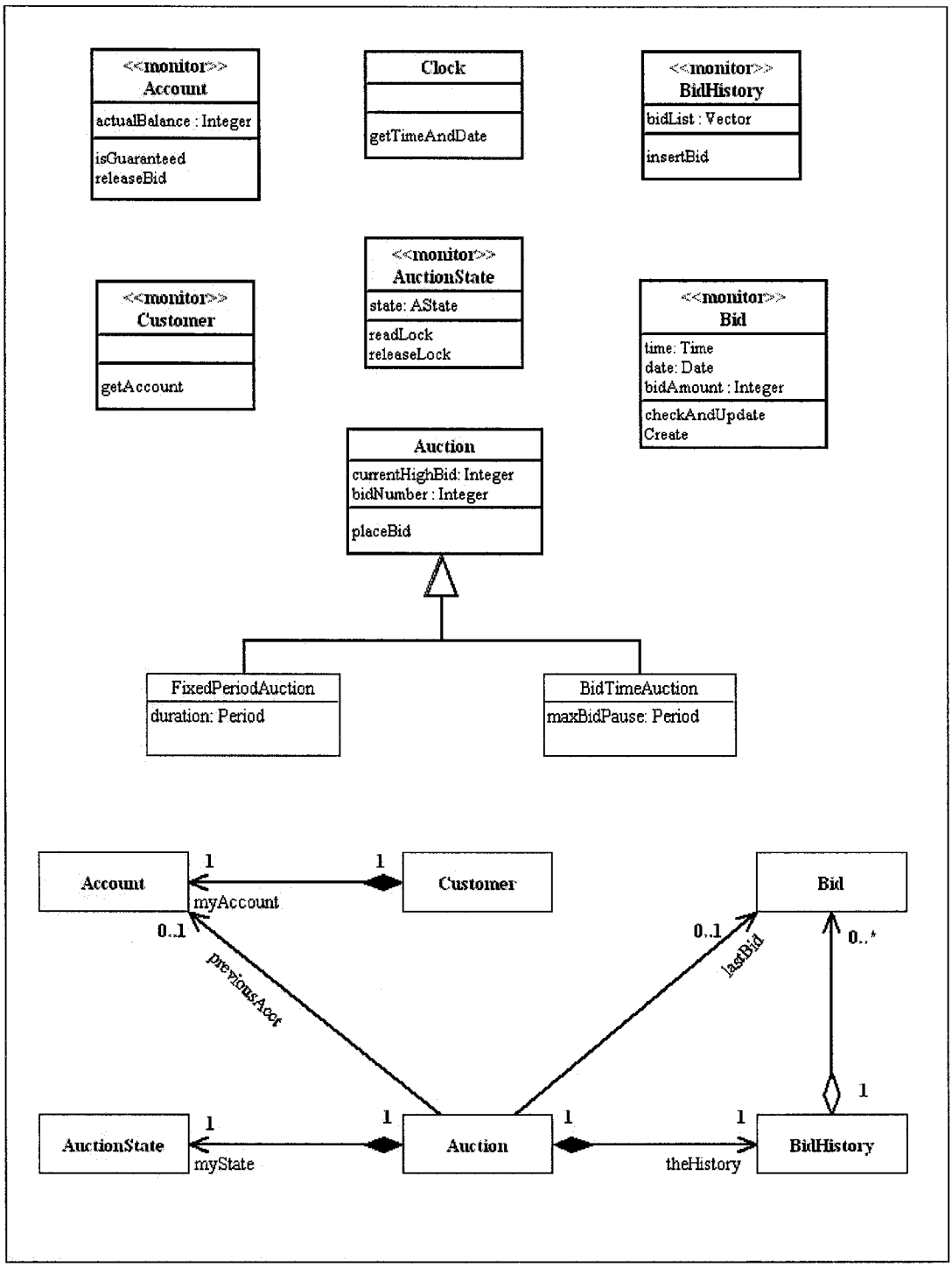

Figure 24. Monitor-based Design Class Model

The monitor-based design class model is very much similar to the transaction-oriented one except some minor differences.

Basically, the navigable associations and permanent references between classes remain the same. Since transactional objects now become monitors in the monitor-based model, all the design classes that used to bear the transactional stereotype now bear the monitor stereotype. Since the atomicity in the monitor-based model is achieved by acquiring *read* or *write locks,* the methods in the *AuctionState* classes have been changed from *isOpen* to *readLock* and *releaseLock.* In addition, the methods in the Bid class have been changed from *getBid* and *setBid* to *checkAndUpdate*

Based on the design class model and the interaction model, the implementation of the system is straightforward.

Chapter 7.
Future Work

The work presented in this thesis shows how concurrency can be integrated into the Fondue development method. The original Fondue abstracts away execution time. Every system operation is assumed to execute instantaneously. The extension presented in this thesis relaxes this assumption, allowing system operations to execute concurrently. In order to still guarantee consistency of the application state, atomic checks and updates can be specified during analysis, and implemented in the design phase using transactions or monitors.

The techniques presented in this thesis work fine as long as atomic checks and updates do not span *multiple* system operations. This is, for example, the case when a logical operation has to be split into several system operations because intermediate information has to be obtained from an external actor.

In the Auction System, for example, we could introduce an *auto-withdraw* feature. In case the customer makes a valid bid but does not have sufficient fund to guarantee the bid, the Auction System automatically contacts the customer's credit institution and requests to transfer funds to the customer's account. At this moment in time, the `placeBid` operation cannot decide yet if the bid is successful. It first has to receive feedback from the credit institution. If the

following return message indicates that the funds have been successfully transferred from the credit institution to the customer's account, the bid can be completed. Otherwise, the bid fails. The main problem is that the Auction System's judgment on the customer's guaranteed balance is based on the information that has to be obtained from an external actor.

During analysis, where we previously were able to specify atomic checks and updates using the *rely* construct in an operation schema, we must now find other means to specify atomicity that spans multiple operations.

In terms of design and implementation, if we use transaction as the solution, the transaction will have to span over the scope of the Auction System. In other words, within the transaction, all state changes that are directly related to the `placeBid` operation inside the Auction System have not committed yet when the operation stops in the middle. Then a transfer fund operation that involves the participating of external actor will be executed. Then the `placeBid` operation continues according to the execution result of transfer fund. To solve this problem, one might consider using more complicated transaction models, such as *chained transactions* or *nested transactions*. If we use monitors with locks as the solution, we might have to use locks that we acquired in one operation, and released in a subsequent one.

The ideas presented in this thesis focused on ensuring that inherent concurrency is discovered during the development of an application, and that shared data structures are accessed in mutual exclusion in order to prevent data corruption. Other issues related to concurrency, such as fairness, scheduling assumptions, memory models, and deadlock situations, have not been addressed directly and are left for future work.

Chapter 8. Conclusion

Concurrency exists in many object-oriented software applications. Any ad hoc solution to address concurrency usually turns out to be unnecessarily complex, inefficient and unstable. We need a systematic approach to deal with concurrency to achieve highly reliable systems.

In this thesis, we stepped through an Auction System case study by following the Fondue method. By adding concurrency notations to the Fondue models, we approached a way to treat inherent concurrency during the early stages of software development.

In the requirement elicitation phase, we added the *Frequency* key word to the use case model, indicating the fact that a user can participate in several auctions simultaneously.

In the analysis phase, we added the multiplicity notations to external actors and communication channels in the environment model, showing the inherent concurrency in the environment. The concept model itself has not been extended, but it provides the base to identify shared concepts at a later stage. The <<concurrent>> stereotype has been added to the protocol model, stating the

auto-concurrent status of a model. The reference tables, which are derived from the protocol model, list potential concurrent operations, and are very helpful to identify shared concepts. In the operation model, we proposed using the *rely* statements in concurrent operation schema to guarantee atomic execution results.

During the design phase, the declarative specifications are refined into an object-oriented design that handles concurrency. We proposed using transactions or monitors to guarantee atomic checks and updates.

Transactions are especially useful to deal with concurrency. Due to the ACID properties, using transactions can ensure atomic execution results in highly concurrent and distributed systems. Examples of such applications are online banking systems, online flight / hotel reservation systems, online shopping systems and online auction systems. In addition, transactions provide tolerance to crash failures.

Monitors can also provide atomicity, but are a lot simpler and hence produce less run-time overhead than transactions. For example, in the object-oriented programming language Java, monitors can be programmed to support concurrency by using synchronization. Thus a Java class, method, or object can be synchronized to ensure atomic operations. This property makes monitors especially suitable when developing small and centralized multithreaded systems.

In the Auction System case study, both design ideas are presented in the form of Fondue interaction models, and finally design class models. The design class models contain all the design classes with attributes and methods. The classes are connected via navigable associations. The transactional objects or monitors are identified using stereotypes. Thus, the implementation based on these models is straightforward.

We believe that our approach helps to better understand the concurrent nature of the problem and the possibilities for addressing the resulting issues in software. Our systematic process leads the developers through the different development stages, focusing on concurrency and providing guidelines on how to transform models when moving from one stage to the other. The approach considerably raises the level of abstraction in which we can describe concurrency and eventually automate software development.

Appendix I: References

[1] OMG Unified Modeling Language Specification, March 2003 version 1.5 formal/03-03-01.

[2] Kruchten P.: *The Rational Unified Process,* Addison-Wesley, 1999.

[3] Sendall, S.; Strohmeier, A.: "UML-based Fusion Analysis". In Proceedings of UML '99, Fort Collins, CO, USA, October 28-30, pp. 278-291, LNCS 1723, Springer Verlag, 1999.

[4] Coleman D.; Arnold P.; Bodoff S.; Dollin C.; Gilchrist H.; Hayes F.; Jeremaes P. : *Object-Oriented Development: The Fusion Method,* Prentice Hall, Englewood Cliffs, NJ, 1994.

[5] Vachon, J.: COALA: A Design Language for Reliable Distributed Systems. Ph.D. Thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, December 2000.

[6] Kienzle, J.; Strohmeier, A.; Romanovsky, A.: "Auction System Design Using Open Multithreaded Transactions". Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable

Systems (WORDS'02), San Diego, CA, USA, January 7th - 9th, 2002, pp. 95 - 104, IEEE Computer Society Press, Los Alamitos, California, USA, 2002.

[7] Strohemier, A.; The Fondue Method, Lecture Notes in Software Engineering Lab, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2003.

[8] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.; *Object-Oriented Modeling and Design,* Prentice Hall, Englewood Cliffs, NJ, 1991.

[9] Jacobson, I.; Christerson, M.; Jonsson, M.; van Overgaard, P.: *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, MA, 1992.

[10] Harel, D.; Politi, M.: *Model Reactive Systems with Statecharts*. McGraw-Hill, 1998.

[11] Warmer, J.; Kleppe, A.: *The Object Constraint Language : Precise Modeling with UML.* Addison-Wesley Pub Co Addison Wesley Longman 1999.

[12] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. Communications of the ACM, vol. 17, no 10, 549-557, October 1974.

[13] Kienzle, J.: Open Multithreaded Transactions: *A Transaction Model for Concurrent Object-Oriented Programming*. Kluwer Academic Publishers, 2003.

[14] Larman, C.: Applying UML and Patterns: *An Introduction to Object-Oriented Analysis and Design*. Second Edition, Prentice Hall, 2001.

[15] Cockburn, A.: *Writing Effective Use Cases*, Addison-Wesley, 2000.

[16] Sendall, S.: Specifying Reactive System Behavior. Ph.D. Thesis, no. 2588, Swiss Federal Institute of Technology, 1015 Lausanne, Switzerland, 2002.

[17] Fenton, N. E.; Pfleeger, S. L.: *Software Metrics: a rigorous and practical approach, 2nd ed.* London; Boston: PWS Publishers., c1997.

[18] Gosling, J.; Joy, B.; Steele, G. L.: *The Java Language Specification*. The Java Series, Addison Wesley, Reading, MA, USA, 1996.

[19] ISO: International Standard ISO/IEC 8652:1995(E): Ada Reference Manual, Lecture Notes in Computer Science 1246, Springer Verlag, 1997; ISO, 1995.

[20] Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[21] Kung, H. T.; Robinson, J. T.: "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems 6(2), June 1981, pp. 213 – 226.

[22] Brinch Hansen, P. : Java's Insecure Parallelism, ACM SIGPLAN Notices, Volume 34, Issue 4, Pages: 38 – 45, April 1999.