Office of Graduate Studies
University of South Florida
Tampa, Florida

CERTIFICATE OF APPROVAL

This is to certify that the thesis of

JENNY YING HU

in the graduate degree program of
Computer Science
was approved on March 27, 2002
for the Master of Science in Computer Science degree.

Examining Committee:

Major Professor: Eugene Fink, Ph.D.

Member: Dmitry B. Goldgof, Ph.D.

Member: Sudeep Sarkar, Ph.D.

EXCHANGES FOR COMPLEX COMMODITIES: REPRESENTATION AND INDEXING OF ORDERS

by

JENNY YING HU

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

May 2002

Major Professor: Eugene Fink, Ph.D.

Acknowledgements

I appreciated the help of my colleagues, family, and friends, who has greatly contributed to my work. I am grateful to Eugene Fink, who has supervised my thesis work and provided guidance in all aspects of my research. I thank Dmitry Goldgof and Sudeep Sarkar for their comments and suggestions, Josh Johnson for helping me understand the code, and Savvas Nikiforou for his assistance with software problems.

I am grateful to my husband, Guiwen Cheng, who has always been here to help me, and to my daughter Annie, who has constantly contributed troubles and made me laugh. I also thank my parents, Zhiqiong Yang and Chaoyuan Hu, who supported me coming to the United States and have always given me their great love.

Table of Contents

EXCHANGES FOR COMPLEX COMMODITIES:
REPRESENTATION AND INDEXING OF ORDERS

by

JENNY YING HU

An Abstract

of a thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

May 2002

Major Professor: Eugene Fink, Ph.D.

The modern economy includes a variety of markets with millions of participants, and the Internet has opened opportunities for the development of new marketplaces. Researchers have extensively studied various auction algorithms, which allow on-line trading of complex goods. They have also developed automated exchange-based markets for standardized commodities, such as stocks and bonds. On the other hand, they have done little work on exchange systems for nonstandard goods.

The purpose of the reported work is to develop an automated exchange for complex nonstandard commodities, such as used cars and collectible stamps. First, we formalize the concept of complex commodities and related trading rules. Second, we propose a distributed exchange architecture, which consists of a central server for matching buyers and sellers, and remote user interfaces. Third, we describe indexing structures for fast identification of matches between buyers and sellers.

The developed system supports complex constraints in the description of commodities and their prices. For example, a car buyer can specify a set of desirable vehicles and their features, and a dependency of the acceptable price on the features. The system also supports constraints on combinations of transactions; it allows a trader to specify mutually exclusive transactions, as well as simultaneous purchase or sale of several commodities.

Abstract Approved: _____

Major Professor: Eugene Fink, Ph.D.
Assistant Professor, Department of Computer Science and Engineering

Date Approved: _____

Chapter 1 Introduction

The modern economy includes a variety of markets, from cars to software to office space, and most markets involve middlemen [Rust and Hall, 2001]. For example, customers usually buy cars through dealerships, which in turn acquire cars from manufacturers; the sale of used vehicles may also involve dealers. These resales increase the cost of goods since they include commissions for the middlemen. The Internet has opened opportunities for reducing the number of middlemen [Klein, 1997; Turban 1997; Wrigley, 1997; Bakos, 2001], and many companies have experimented with direct sales over the web. The on-line marketplaces include bulletin boards, auctions, and exchanges.

Electronic bulletin boards are similar to traditional newspaper classifieds. They vary from sale catalogs to newsgroup postings, which help buyers and sellers to find each other; however, they often require a customer to invest significant time into searching among multiple ads. For this reason, many buyers prefer on-line auctions, such as eBay (www.ebay.com). Auctions have their own problems, including significant computational costs and asymmetry between buyers and sellers. For example, a traditional auction requires a buyer to bid on a specific item; it helps sellers to obtain the highest price, but it limits buyers' flexibility. Furthermore, most auctions do not allow fast sales; for instance, if a seller posts an item on eBay, she can sell it in three or more days, but not sooner.

An exchange-based market does not have these problems; it ensures symmetry between buyers and sellers, and supports fast-paced trading. Examples of liquid markets include the traditional stock and commodity exchanges, as well as currency and bond markets. For instance, a trader can buy or sell any public stock in seconds, at the best available price. The main limitation of these exchanges is rigid standardization of tradable items. For instance, the New York Stock Exchange allows trading of about 3,100 securities, and the buyer or seller has to indicate a specific item, such as IBM stock.

For most goods and services, the description of a desirable trade is more complex. For instance, a car buyer needs to specify a make, model, options, color, and other

features. She usually has a certain flexibility and may accept any car that satisfies her constraints, rather than looking for one specific vehicle. She may also need to include preferences into her description; for example, she may indicate that a white Mustang is acceptable but less desirable than a red Mustang. An effective on-line exchange for nonstandard commodities should satisfy the following requirements:

- Allow complex constraints in specifications of buy and sell orders
- Support fast-paced trading for markets with millions of orders
- Include optimization techniques that maximize traders' satisfaction
- Allow a user to select preferred trades among matches for her order

The purpose of the reported work is to develop an automated exchange for complex goods and services. We begin with a motivating example, review of the previous work on automated auctions and exchanges, and summary of our main contributions.

1.1 Example

We give an example of a car exchange that would allow trading new and used vehicles. To simplify this example, we assume that a trader can describe a car by four attributes: model, color, year, and mileage. For instance, a seller may offer a red Mustang, made in 1999, with 35,000 miles.

In Figure 1.1(a), we illustrate a traditional car market, which includes manufacturers, car dealers, and customers. Dealers get cars from manufacturers and resell them to individual clients. In Figure 1.1(b), we show an alternative trading model with a centralized exchange, similar to a stock market. Car dealers may participate in this exchange by acquiring large quantities of cars and reselling them to customers; however, they are no longer an essential link, and manufacturers can sell directly to end users.

The exchange allows placing buy and sell orders, analogous to the orders in a stock market. A prospective buyer can place a *buy order,* which includes a description of the desired vehicle and a maximal acceptable price. For instance, she may indicate that she wants a red Mustang, made after 1999, with less than 20,000 miles, and she is willing to pay $19,000. Similarly, a seller can place a *sell order;* for instance, a manufacturer may offer a brand-new Mustang of any color for $18,000.

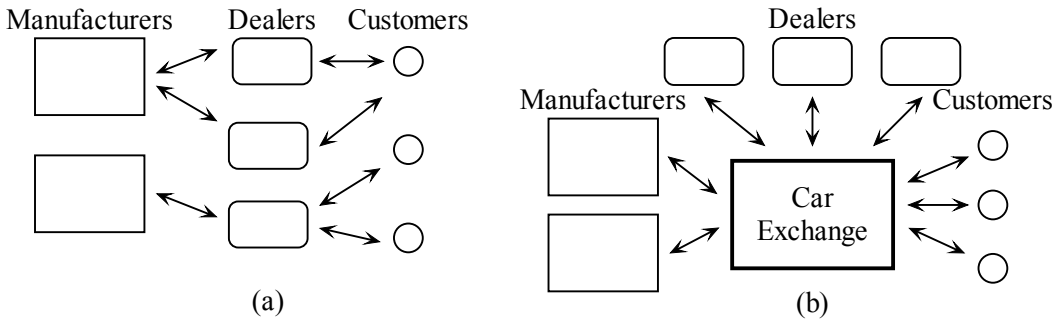


Figure 1.1: The traditional car market versus a centralized exchange. The usual market includes manufacturers, dealers, and customers (a). An alternative scheme is to allow trading through the central exchange (b).

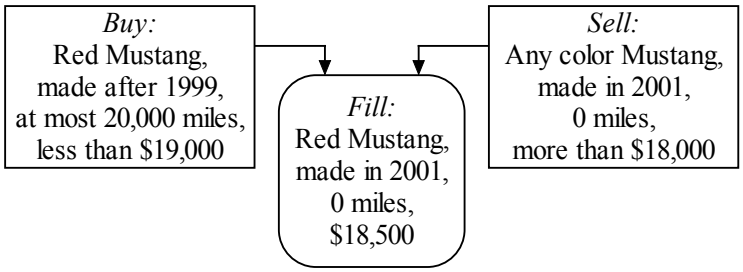


Figure 1.2: Matching orders and the resulting trade. When the system finds a match between two orders, it generates a fill, which is a trade that satisfies both parties.

The exchange system searches for matches between buy and sell orders, and generates corresponding *fills,* that is, transactions that satisfy both buyers and sellers. In the previous example, it will determine that a brand-new red Mustang for $18,500 satisfies both the buyer and the seller (see Figure 1.2); thus, the customer will acquire a red Mustang from the manufacturer for $18,500.

If the system does not find fills for some orders, it keeps them in memory and tries to match them with incoming orders. We illustrate it in Figure 1.3, where the system keeps the buy order for two hours, until it receives a matching sell order.

If the system finds several matches for an order, it chooses the match with the best price. For example, the buy order in Figure 1.4(a) will trade with the cheaper of the two sell orders. If two matching orders have the same price, the system gives preference to the earlier one, as shown in Figure 1.4(b). These rules correspond to the standard "fairness" requirements of the financial industry.

The system allows a user to trade several identical items by specifying a size for an order. For example, a dealer can place an order to sell four Mustangs, which may trade

with several buy orders; that is, the system can match it with a smaller buy order (see Figure 1.5) and later find a match for the remaining cars. In addition, the user can specify a minimal acceptable size of a transaction. For instance, the dealer may place an order to sell four Mustangs, and indicate that she wants to trade at least two cars. In this case, her order will match with buy orders whose size is at least two (see Figure 1.6a). If an order includes a minimal size, a user can indicate that she will accept a simultaneous trade with several orders, if their total size is above the specified minimum (see Figure 1.6b).

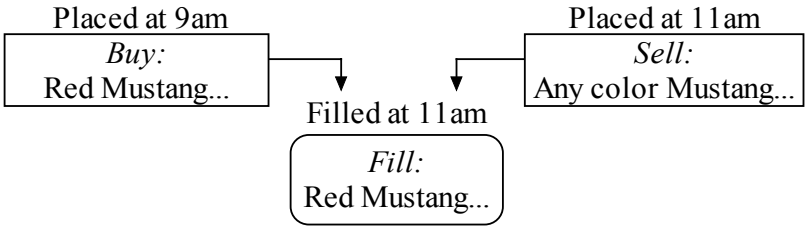


Figure 1.3: Matching an old order with a new one. If the system cannot find an immediate match for an order, it keeps looking for matches among new orders. In this example, it gets a buy order at 9am and finds a match two hours later.

Furthermore, a user can indicate that a transaction size must be divisible by a certain number, called a *size step*. For example, a wholesale agent may specify that she is selling fifteen cars, and the transaction size must be divisible by five. In this case, she may sell five, ten, or fifteen cars (see Figure 1.7).

A user can specify that she is willing to trade any of several items. For example, a customer can place an order to buy either a Mustang or a Camaro. As another example, a dealer can offer ten cars, which may be Mustangs, Camaroes, and Vipers; then, she may end up selling ten Mustangs, or ten Camaroes, or three Mustangs and seven Vipers.

If a user describes a set of items, she can indicate that the price depends on an item. For example, if a customer wants a Mustang or Camaro, she may offer $18,500 for a Mustang and $17,500 for a Camaro. Furthermore, she may offer an extra $500 if a car is red, and subtract $1 for every ten miles on its odometer.

A user can also specify her preferences for choosing among potential trades; for instance, she may indicate that a red Mustang is better than a white Mustang, and that a Mustang for $19,000 is better than a Camaro for $18,000 (see Figure 1.8). She specifies her preferences by providing some measure of transaction quality, that is, assigning a numeric quality to each acceptable transaction.

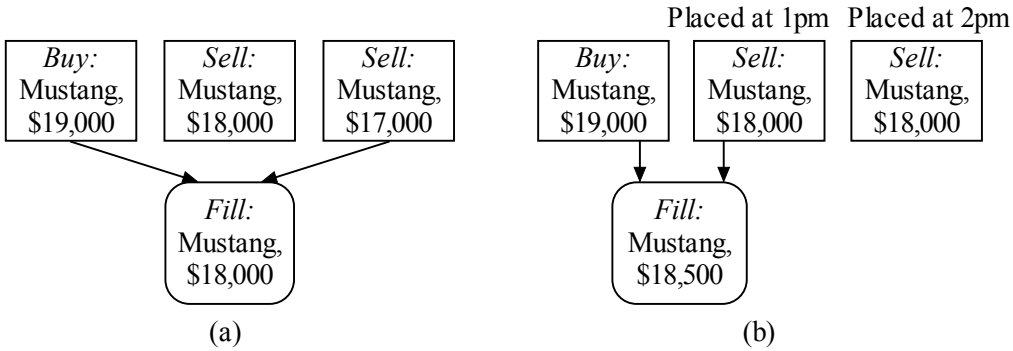


Figure 1.4: Fairness rules. When the system finds several matches for an order, it chooses the best-price match (a). If two matches have the same price, it prefers the earlier order (b).

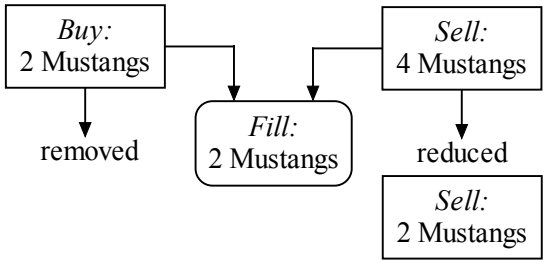


Figure 1.5: Example of order sizes. Buyers and sellers can trade several items at once by specifying an order size. When the system finds a match, it completely fills the smaller order and reduces the size of the larger one.

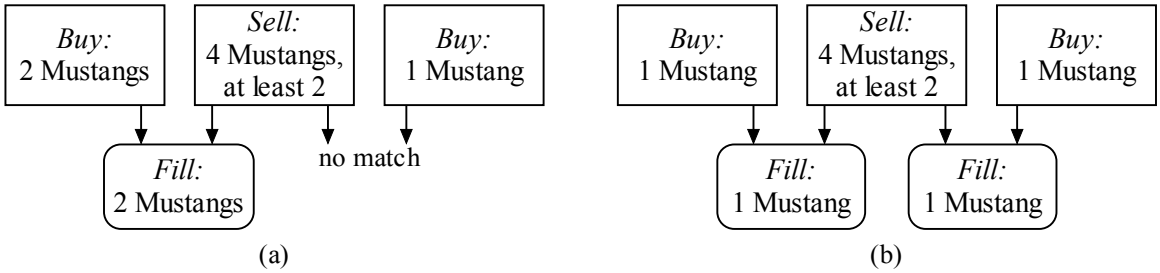


Figure 1.6: Order with a minimal acceptable size. It matches with orders whose size is no smaller than the minimal size (a). The trader may optionally allow matching with multiple small orders that have appropriate total size (b).

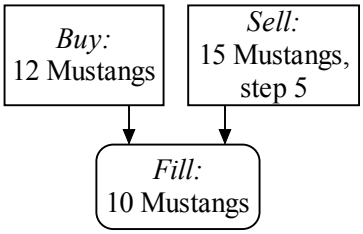


Figure 1.7: Specifying a size step. The fill size must be divisible by this step.

For example, suppose that a buyer is looking for a Toyota Echo made after 1998. She has $10,000 for the purchase, and she tentatively plans to stay within this limit; however, she may consider paying up to $12,000 if it allows buying a much better car. She can specify her quality estimate as shown in Figure 1.9. In this example, she assigns a hundred-point quality to a 1999 model for $10,000, adds extra points if a car is newer, subtracts points for miles, and adds or subtracts points depending on the price. Based on this quality measure, she prefers the first of the three sell orders in Figure 1.9.

If a trader is interested in several alternative transactions, she can specify that she wants to execute one of several orders. For example, suppose that she wants to buy a Camry for $20,000, or sell an Echo for $12,000, or sell a Mustang for $18,000. After filling one of these orders, the system will cancel the other two (see Figure 1.10). A trader can also indicate that she wants to complete several transactions at once. For example, she may place an order to sell a Tercel for $10,000 and buy a Camry for $20,000, and then the system must complete both transactions together (see Figure 1.11).

Furthermore, a trader can indicate that several orders should be executed in a sequence. For example, she may first sell a Tercel for $10,000, then buy a Sequoia for $30,000, and finally buy a trailer for $2,000 (see Figure 1.12). The system will match the second order only after finding a fill for the first one, and it will match the last order after filling the first two.

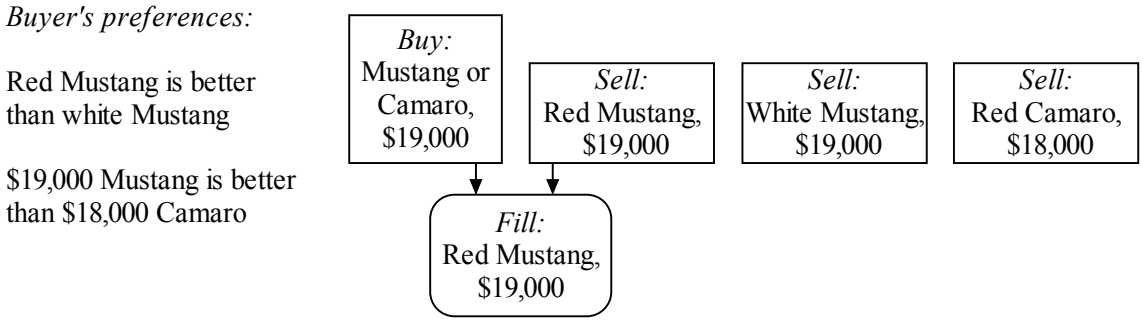


Figure 1.8: Example of preferences. A customer wants to buy a Mustang or Camaro, and she specifies her preferences; the system uses them to choose among matching orders.

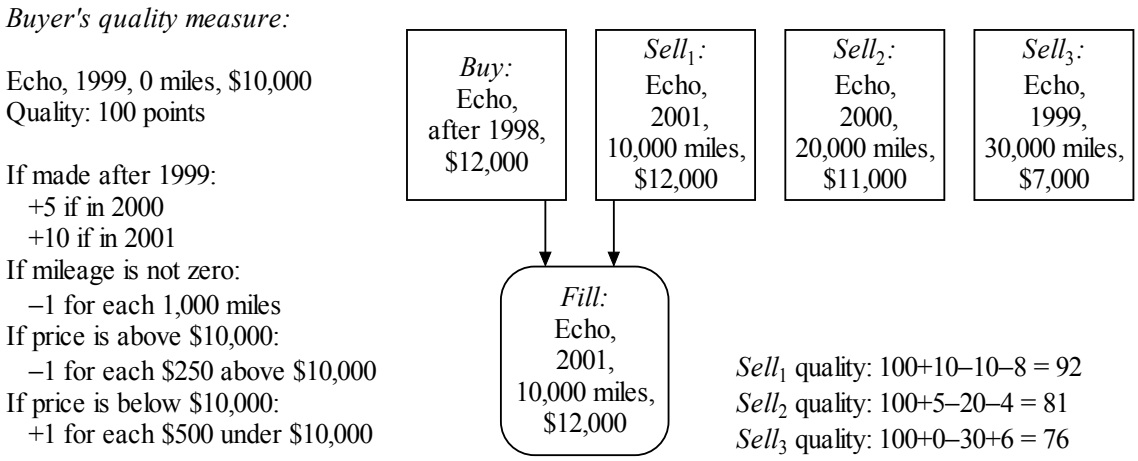


Figure 1.9: Example of a quality measure. A customer encodes her preferences by a numeric quality, and the system uses it to compare matching orders.

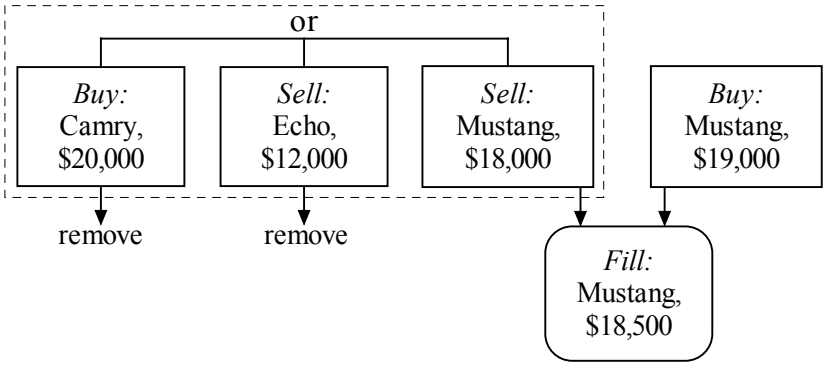


Figure 1.10: Executing one of several alternative transactions. The system fills one of the alternative orders and then removes the others.

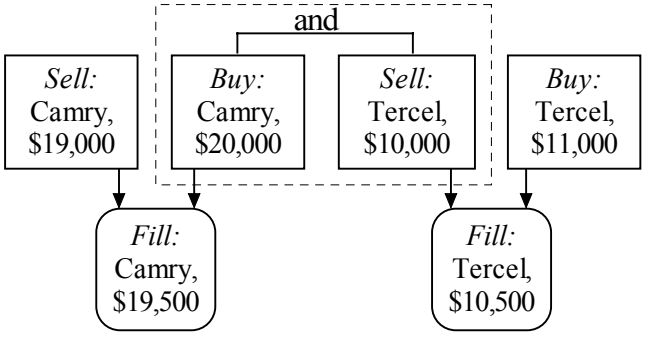


Figure 1.11: Executing two orders at once.

Figure 1.12: Executing several orders in a sequence.

A customer can request additional information about potential trades before committing to one of them. For instance, she can view the pictures of matching cars, along with their technical descriptions, and manually select the best match. After placing an order, the customer can modify its description without removing it from the market; in particular, she can change its price and size, and modify the item description.

1.2 Previous Work

Economists and computer scientists have long realized the importance of auctions and exchanges, and studied a variety of trading models. The related computer science research has been focused on optimal matches in various auction scenarios, and on general-purpose systems for auctions and exchanges. It has led to successful systems for Internet auctions, such as eBay (www.ebay.com), Bid.Com (www.bid.com), and Yahoo Auctions (auctions.yahoo.com). Some companies have also built on-line auctions for business-to-business transactions; for example, FreeMarkets (www.freemarkets.com) has deployed a reverse auction that allows suppliers to bid for a large contract with a business customer.

Although these auctions are more effective than traditional bulletin boards, they have the typical drawbacks of auction markets, including significant computational requirements and asymmetric treatment of buyers and sellers.

Recently, researchers have developed several efficient systems for *combinatorial auctions,* which allow buying and selling sets of commodities rather than individual items. They have considered not only auctions with completely specified commodities, but also markets that enable a user to negotiate desirable features of merchandise.

1.2.1 Combinatorial Auctions

A traditional combinatorial auction allows bidding on a set of fully specified items. For example, Katie may bid on a red Mustang, black Corvette, and silver BMW, for a total

price of $80,000. In this case, she will get all three cars together or nothing; that is, the system will not generate a partial fill. An advanced auction may allow disjunctions; for instance, Katie may specify that she wants either a red Mustang and black Corvette or, alternatively, two silver BMWs. On the other hand, standard combinatorial auctions do not allow incompletely specified items, such as a Mustang of any color.

Rothkopf *et al.* [1998] gave a detailed analysis of combinatorial auctions and described semantics of combinatorial bids that allowed fast matching, but did not develop a matching algorithm. Nisan discussed alternative semantics for combinatorial bids, formalized the problem of searching for optimal and near-optimal matches, and proposed a linear-programming solution, but did not test its effectiveness [Nisan, 2000; Lavi and Nisan, 2000].

Sandholm [1999] developed several efficient algorithms for one-seller combinatorial auctions, and showed that they scaled to a market with about one thousand bids. Sandholm and his colleagues later improved the original algorithms and implemented a system that processed several thousand bids [Sandholm, 2000a; Sandholm and Suri, 2000; Sandholm *et al.,* 2001a]. They developed a mechanism for determining a trader's preferences and converting them into a compact representation of combinatorial bids [Conen and Sandholm, 2001]. They also described several special cases of bid processing that allowed polynomial solutions, proved the NP-completeness of more general cases, and tested various heuristics for NP-complete cases [Sandholm and Suri, 2001; Sandholm *et al.,* 2001b].

Fujishima proposed an approach for enhancing standard auction rules, analyzed trade-offs between optimality and running time, and presented two related algorithms [Fujishima *et al.,* 1999a; Fujishima *et al.,* 1999b]. The first algorithm ensured optimal matching and scaled to about one thousand bids, whereas the second found near-optimal matches for a market with ten thousand bids.

Lehmann *et al.* [1999] investigated heuristic algorithms for combinatorial auctions and identified cases that allowed *truthful bidding,* which meant that users did not benefit from providing incorrect information about their intended maximal bids. Gonen and Lehmann [2000, 2001] studied branch-and-bound heuristics for processing combinatorial bids and integrated them with linear programming.

Yokoo *et al.* [2001a, 2001b] considered a problem of *false-name bids,* that is, manipulation of prices by creating fictitious users and submitting bids without intention to buy; they proposed auction rules that discouraged such bids.

Andersson *et al.* [2000] compared the main techniques for combinatorial auctions and proposed an integer-programming representation that allowed richer bid semantics. In particular, they removed some of the restrictions imposed by Rothkopf *et al.* [1998].

Wurman *et al.* [2001] have compared a variety of previously developed auctions and identified the main components of an automated auction, including bid semantics, clearing mechanism, rules for pricing and canceling bids, and policies for hiding information from other users. They proposed a standardized format for describing the components of each specific auction.

Although the developed systems can efficiently process several thousand bids, their running time is superlinear in the number of bids, and they do not scale to larger markets.

1.2.2 Advanced Semantics

Several researchers have studied techniques for specifying the dependency of an item price on the number and quality of items. They have also investigated techniques for processing "flexible" bids, specified by hard and soft constraints, similar to buy orders in Figures 1.2, 1.3, and 1.8.

Che [1993] analyzed auctions that allowed negotiating not only the price, but also the quality of a commodity. A bid in these auctions is a function that specifies a desired trade-off between the price and quality. Cripps and Ireland [1994] considered a similar setting and suggested several different strategies for bidding on the price and quality.

Sandholm and Suri [2001] studied combinatorial auctions that allowed bulk discounts; that is, they enabled a bidder to specify a dependency between item price and order size. For example, a car dealer could be willing to buy five Toyota Echoes for $10,000 each, or ten Echoes for $9,500 each. Lehmann *et al.* [2001] also considered the dependency of price on order size, showed that the corresponding problem of finding best matches was NP-hard, and developed a greedy approximation algorithm.

Bichler discussed a market that would allow negotiations on any attributes of a commodity [Bichler *et al.,* 1999; Bichler, 2000]; for instance, a car buyer could set a fixed price and negotiate the options and service plan. He analyzed several alternative versions of this model, and concluded that it would greatly increase the economic utility of auctions; however, he pointed out the difficulty of implementing it and did not propose any computational solution.

Jones [2000] extended the semantics of combinatorial auctions and allowed buyers to use complex constraints; for instance, a car buyer could bid on a vehicle that was less than three-year old, or on the fastest available vehicle. She suggested an advanced semantics for these constraints, which allowed compact description of complex bids; however, she did not allow complex constraints in sell orders. She implemented an algorithm that found near-optimal matches, but it scaled only to one thousand bids.

This initial work leaves many open problems, which include the use of complex constraints with general preference functions, symmetric treatment of buy and sell orders, and design of efficient matching algorithms for advanced semantics.

1.2.3 Exchanges

Economists have extensively studied traditional stock exchanges; for example, see the historical review by Bernstein [1993] and the textbook by Hull [1999]. They have focused on exchange dynamics and related mathematics, rather than on efficient algorithms [Cason and Friedman, 1999; Bapna *et al.,* 2000]. Several computer scientists have also studied trading dynamics and proposed algorithms for finding the market equilibrium [Reiter and Simon, 1992; Cheng and Wellman, 1998; Andersson and Ygge, 1998].

Successful on-line exchanges include electronic communication networks, such as REDI (www.redibook.com), Island (www.island.com), NexTrade (www.nextrade.org), Archipelago (www.tradearca.com), and Instinet (www.instinet.com). The directors of large stock and commodity exchanges are also considering electronic means of trading. For example, the Chicago Mercantile Exchange has deployed the Globex system, which supports trading around the clock.

Some auction researchers have investigated the related theoretical issues; they have viewed exchanges as a variety of auction markets, called *continuous double auctions.* In particular, Wurman *et al.* [1998a] proposed a theory of exchange markets and implemented a general-purpose system for auctions and exchanges, which processed traditional fully specified orders. Sandholm and Suri [2000] developed an exchange for combinatorial orders, but it could not support markets with more than one thousand orders. Blum *et al.* [2002] explored methods for improving liquidity of standardized exchanges. Kalagnanam *et al.* [2000] investigated techniques for placing orders with complex constraints and identifying matches between them. They developed network-flow algorithms for finding optimal matches in simple cases, and showed that more complex cases were NP-complete. The complexity of their algorithms was superlinear in the number of orders, and the resulting system did not scale beyond a few thousand orders.

The related open problems include development of a scalable system for large combinatorial markets, as well as support for flexible orders with complex constraints.

1.2.4 General-Purpose Systems

Computer scientists have developed several systems for different types of auctions and exchanges, which vary from specialized markets to general-purpose tools for building new markets. The reader may find a survey of most systems in the review articles by Guttman *et al.* [1998a, 1998b] and Maes *et al.* [1999].

For example, Chavez and his colleagues designed an on-line agent-based auction; they built intelligent agents that negotiated on behalf of buyers and sellers [Chavez and Maes, 1996; Chavez *et al.,* 1997]. Vetter and Pitsch [1999] constructed a more flexible agent-based system that supported several types of auctions. Preist [1999a; 1999b] developed a similar distributed system for exchange markets. Bichler designed an electronic brokerage service that helped buyers and sellers to find each other and to negotiate through auction mechanisms [Bichler *et al.,* 1998; Bichler and Segev, 1999].
Wurman and Wellman built a general-purpose system, called the Michigan Internet AuctionBot, that could run a variety of different auctions [Wellman, 1993; Wellman and Wurman, 1998; Wurman *et al.,* 1998b; Wurman and Wellman, 1999a]; however, they

restricted users to simple fully specified bids. Their system included scheduler and auctioneer procedures, related databases, and advanced interfaces. Hu *et al.* [1999] created agents for bidding in the Michigan AuctionBot; they used regression and learning techniques to predict the behavior of other bidders. Later, Hu *et al.* [2000] designed three types of agents and showed that their relative performance depended on the strategies of other auction participants. Hu and Wellman [2001] developed an agent that learned the behavior of its competitors and adjusted its strategy accordingly. Wurman [2001] considered a problem of building general-purpose agents that simultaneously bid in multiple auctions.

Parkes built a fast system for combinatorial auctions, but it worked only for markets with up to one hundred users [Parkes, 1999; Parkes and Ungar, 2000a]. Sandholm created a more powerful auction server, configurable for a variety of markets, and showed its ability to process several thousand bids [Sandholm, 2000a; Sandholm, 2000b; Sandholm and Suri, 2000].

All these systems have the same key limitation as commercial on-line exchanges: they require fully specified bids and do not support the use of constraints.

1.3 Contributions

The review of previous work has shown that techniques for trading of complex commodities are still limited. Researchers have investigated several auction models, as well as exchanges for standardized securities, but they have not developed exchanges for complex goods. The main open problems include design of an automated exchange for complex securities, and development of a rigorous theory of complex exchanges.

A recent project at the University of South Florida has been aimed at addressing these problems. Johnson [2001] has defined complex orders and related trading semantics, which are applicable to a variety of markets. He has developed an exchange system that supports a market with 300,000 orders and processes 200 to 500 new orders per second. We have continued his work, extended the order semantics, and developed indexing structures for fast identification of matches between buy and sell orders.

We view a market as a set of tradable items, and a specific order as its subset. We allow price functions that show the dependency of the order price on a specific item, as

well as utility functions for encoding preferences among potential trades. We consider several types of combinatorial orders, which include mutually exclusive transactions, simultaneous purchase or sale of several commodities, and chains of consecutive trades.

First, we analyze a general trading problem, give a formal model of an exchange for combinatorial orders, and define the related trading rules (Chapter 2). Then, we explain the representation of orders in the implemented system and the main operations supported by the system, such as placement of new orders, modification and cancellation of orders, and manual selection among potential transactions (Chapter 3). Finally, we propose a distributed exchange architecture, which consists of a central matcher and remote user interfaces, and describe the data structures and algorithms that allow fast search for matches between buy and sell orders (Chapter 4). We conclude with a summary of results and discussion of future challenges (Chapter 5).

Chapter 2 General Exchange Model

We describe a general model of trading complex commodities, using a car market as an example. We formalize the concept of buy and sell orders, consider a trading environment that allows hard and soft constraints in the order specification, and discuss methods for representing combinations of purchases and sales. In Chapters 3 and 4, we will present an automated exchange that supports a limited version of this general model.

2.1 Orders

We begin by defining buy and sell orders, which include descriptions of commodities, price and size specifications, and traders' preferences among acceptable transactions. We then state conditions of a match between a buy order and sell order.

2.1.1 Buyers and Sellers

When a buyer looks for a certain item, she usually has some flexibility; that is, she may buy any of several acceptable items. For example, if Katie needs a sports car, she may be willing to buy any of several models, such as a Mustang, Camaro, and Viper. For each model, Katie has to determine the maximal acceptable price. Furthermore, she may have preferences among these models; for instance, she may prefer Mustangs to other models, and she may prefer red cars to black ones.

Similarly, when a dealer sells a vehicle, she has to decide on the minimal acceptable price. For instance, Laura may be selling a Mustang for no less than $19,000 and a Camaro for no less than $18,000. If the dealer offers multiple items, she may prefer some sales to others; for example, Laura may prefer to sell the Mustang for $19,000 rather than the Camaro for $18,000.

If a buyer's constraints match a seller's constraints, they may trade; that is, the buyer may purchase an item from the seller. If a buyer finds several acceptable items, she

usually buys the best among them; similarly, a seller may be able to choose the most attractive deal among several offers.

We use the term *buy order* to refer to a buyer's set of constraints; for example, Katie's desire to purchase a sports car can be expressed as an order for a sports car, and her price limits and preferences will be part of this order. When a buyer announces her desire to trade, we say that she has *placed an order*. Similarly, a *sell order* is a seller's constraint set that defines the offered merchandise. For instance, Laura may place an order to sell a Mustang or Camaro, and her order may also include price limits and preferences.

2.1.2 Concept of an Order

A specific market includes a certain set of items that can potentially be bought and sold; we denote it by M, which stands for *market set*. This set may be very large or even infinite; in the car market, it includes all vehicles that have ever been made, as well as the cars that can be made in the future.

When a trader makes a purchase or sale, she has to specify a set of acceptable items, denoted I, which stands for *item set*; it must be a subset of M, that is, $I \subseteq M$. For example, if Katie shops for a brand-new sports car, her set I includes all new sports vehicles.

In addition, a trader should specify a limit on the acceptable price; for instance, Katie may be willing to pay $19,000 for a red Mustang, but only $18,500 for a black Mustang, and even less for a Camaro. Formally, a price limit is a real-valued function defined on the set I; for each item $i \in I$, it gives a certain limit $Price(i)$. For a buyer, $Price(i)$ is the maximal acceptable price; for a seller, it is the minimal acceptable price. To summarize, a buy or sell order must include two elements (see Figure 2.1a):

- A set of items, $I \subseteq M$
- A price function, $Price: I \rightarrow \mathbf{R}$,
  where $\mathbf{R}$ is a set of real-valued prices

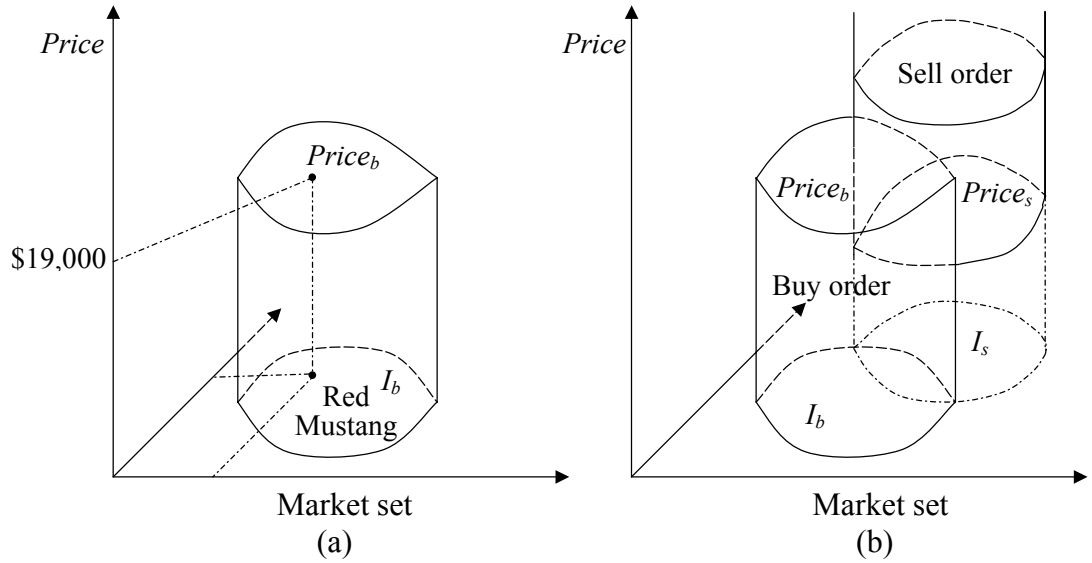


Figure 2.1: Example of a buy order (a) and a match between a buy and sell order (b). The horizontal plane represents the market set M, and the vertical axis is the price **R**. The buyer is interested in a certain set I_b of cars with different price limits; in particular, she would buy a red Mustang for $19,000. Her order matches the sell order on the right.

The prices in consumer markets are usually in dollars or other currencies; however, traders in some specialized markets may use different price measures. For example, mortgage brokers often view the interest rate as the "price" of a mortgage. The properties of such price measures may differ from those of dollar prices. In particular, the price may not be additive; for instance, if a customer takes a 5% loan and a 6% loan, the overall interest is not 11%.

We allow such price measures and do not require the use of dollar prices. The only requirement is that a price decrease always benefits a buyer, and a price increase benefits a seller. In other words, the buyer is interested in finding a given item with the lowest available price, whereas the seller tries to get the highest possible price. For instance, bank customers look for low-interest loans, whereas bankers try to get high interests.

We say that a buy order *matches* a sell order if the buyer's constraints are consistent with the seller's constraints, thus allowing a mutually acceptable trade (see Figure 2.1b). For instance, if Katie is willing to pay $19,000 for a red Mustang, and Laura offers a red Mustang for $18,000, then their orders match. Formally, a buy order

(I_b, $Price_b$) matches a sell order (I_s, $Price_s$) if some item i satisfies both buyer and seller, at a mutually acceptable price:

there exists $i \in I_b \cap I_s$ such that $Price_s(i) \leq Price_b(i)$.

2.1.3 Quality Functions

Buyers and sellers may have preferences among acceptable trades, which depend on a specific item i and its price p. For instance, Katie may prefer a red Mustang for $19,000 to a black Camaro for $18,000.

We represent preferences by a real-valued function $Qual(i, p)$ that assigns a numeric quality to each pair of an item and price. Larger values correspond to better transactions; that is, if $Qual(i_1, p_1) > Qual(i_2, p_2)$, then the user would rather trade i_1 at price p_1 than i_2 at p_2. For example, Katie's quality function satisfies the following condition:

$Qual$(red-Mustang, $19,000) > $Qual$(black-Camaro, $18,000).

Furthermore, we assume that negative quality values correspond to unacceptable trades; that is, if $Qual(i, p) < 0$, the user will not trade item i at price p.

Each trader may use her own quality functions and specify different functions for different orders. Note that we define quality as a totally ordered function, which is a simplification, because traders may reason in terms of partially ordered preferences. For instance, Katie may believe that a $19,000 Mustang is better than an $18,000 Camaro, but she may be undecided between a $19,000 Mustang and a $17,000 Camaro. Also note that buyers look for low prices, whereas sellers prefer to get as much money as possible, which means that quality functions must be monotonic on price:

- *Buy monotonicity:* If $Qual_b$ is a quality function for a buy order, and $p_1 \leq p_2$, then, for every item i, we have $Qual_b(i, p_1) \geq Qual_b(i, p_2)$.
- *Sell monotonicity:* If $Qual_s$ is a quality function for a sell order, and $p_1 \leq p_2$, then, for every item i, we have $Qual_s(i, p_1) \leq Qual_s(i, p_2)$.

We do not require a user to specify a quality function for each order; by default, quality is defined through price. This default quality is a function of a transaction price and its difference from the user's price limit. For example, buying a Toyota Echo for $11,000 is better than buying it for $12,000; as another example, if a user has specified a

$12,000 price limit for an Echo and a $19,000 limit for a Mustang, then buying a Mustang for $11,000 is better than buying an Echo for $11,000.

To formalize this rule, we denote the user's price function by *Price*, and the price of an actual purchase or sale of an item i by p. The default quality function must satisfy the following conditions for every item i and price p:

- *For buy orders:* If $Price_1(i) \le Price_2(i)$, then $Qual_1(i, p) \le Qual_2(i, p)$.
- *For sell orders:* If $Price_1(i) \le Price_2(i)$, then $Qual_1(i, p) \ge Qual_2(i, p)$.

Intuitively, the larger the gap between the price limit and actual price, the better the deal; that is, the more the user saves, the more she likes the transaction.

We have considered two default functions, and a user can choose either of them. The first function is the difference between the price limit and actual price:

- *For buy orders:* $Qual_b(i, p) = Price(i) - p$.
- *For sell orders:* $Qual_s(i, p) = p - Price(i)$.

This default is typical for financial and wholesale markets; intuitively, the quality of a transaction depends on a user's savings. For example, suppose that a car dealer wants to purchase either ten Mustangs for $19,000 each or ten Echoes for $12,000 each. Suppose further that she finds Mustangs for $17,500 and Echoes for $11,000. If she buys Mustangs, she saves ($19,000 – $17,500) · 10 = $15,000. On the other hand, if she acquires Echoes, her savings are only ($12,000 – $11,000) · 10 = $10,000. Thus, the first transaction is more attractive.

The other default function is the ratio of the price difference to the price limit:

- *For buy orders:* $Qual_b(i, p) = \frac{Price(i) - p}{Price(i)}$.
- *For sell orders:* $Qual_s(i, p) = \frac{p - Price(i)}{Price(i)}$.

This default is traditional for consumer markets; it shows a user's percentage savings. For instance, if a customer is willing to pay $19,000 for a Mustang, and she gets an opportunity to buy it for $17,500, then the transaction quality is $\frac{\$19,000 - \$17,500}{\$19,000} = 0.08$. If she is also willing to pay $12,000 for an Echo and finds that it is available for $11,000, the quality of buying it is $\frac{\$12,000 - \$11,000}{\$12,000} = 0.09$, which is preferable to the Mustang.

2.1.4 Order Sizes

If a user wants to trade several identical items, she can include their quantity in the order specification; for example, Katie can place an order to buy two sports cars, and Laura can announce a sale of fifty Camaroes. We assume that an order size is a natural number; thus, we enforce discretization of continuous commodities, such as orange juice.

The user can specify not only an overall order size but also a minimal acceptable size. For instance, suppose that a Toyota wholesale agent is selling one thousand cars, and that she works only with dealerships that are buying at least twenty vehicles. Then, she may specify that the overall size of her order is one thousand, and the minimal size is twenty. If the minimal size equals the overall size, we say that the order is *all-or-none*. For example, the agent may offer twenty cars and specify that her minimal size is also twenty; then, she will sell either nothing or twenty cars at once.

In addition, the user can indicate that a transaction size must be divisible by a certain number, called a *size step*. For example, stock traders often buy and sell stocks in blocks of hundred. As another example, a wholesale agent may specify that she is selling cars in blocks of ten; in this case, she would be willing to sell twenty or thirty cars, but not twenty-five.

To summarize, an order may include six elements:

- Item set, I
- Price function, *Price*: $I \to \mathbf{R}$
- Quality function, *Qual*: $I \times \mathbf{R} \to \mathbf{R}$
- Overall order size, *Max*
- Minimal acceptable size, *Min*
- Size step, *Step*

The item set, price limit, and size specification are hard constraints that determine whether a buy order matches a sell order, whereas the quality function serves as both hard and soft constraints. Rejection of a negative quality is a hard constraint, whereas choice of large values among positive-quality transactions is a soft constraint.

To define the matching conditions, we denote the item set of a buy order by I_b, its price function by $Price_b$, its quality function by $Qual_b$, and its size parameters by Max_b,

Min_b, and $Step_b$. Similarly, we denote the parameters of a sell order by I_s, $Price_s$, $Qual_s$, Max_s, Min_s, and $Step_s$. The two orders match if they satisfy the following constraints.

Conditions 2.1

- There is an item $i \in I_b \cap I_s$, such that $Price_s(i) \leq Price_b(i)$.
- There is a price p, such that
  - $Price_s(i) \leq p \leq Price_b(i)$, and
  - $Qual_b(i, p) \geq 0$ and $Qual_s(i, p) \geq 0$
- There is a mutually acceptable size value *size*, such that
  - $Min_b \leq size \leq Max_b$,
  - $Min_s \leq size \leq Max_s$, and
  - *size* is divisible by $Step_b$ and $Step_s$

The price and quality functions in this model do not depend on a transaction size, which is a simplification, because sellers sometimes offer discounts for bulk orders. For example, a car dealer may give a discount to a customer who purchases two cars at once, and an even larger discount to a buyer of five cars. In such cases, a seller can place several orders with different price limits and minimal sizes, as illustrated in Figure 2.2. If a seller wants to complete only one of these orders, she can use the disjunctive-order mechanism described in Section 2.3.1.

Figure 2.2: Example of a bulk discount. If a dealer is offering a lower price for bulk purchases, she has to place several orders with different prices and minimal sizes.

2.1.5 Market Attributes

The set M of all possible items may be very large, which means that we cannot explicitly represent all items. For instance, we cannot make a catalog of all feasible cars because it would include a separate entry for each combination of model, color, features, year, and mileage. To avoid this problem, we define the set M by a list of attributes and possible

values for each attribute. As a simplified example, we describe a car by four attributes: *Model, Color, Year,* and *Mileage.*

Formally, every attribute is a set of values; for instance, the *Model* set may include all car models, *Color* may include standard colors, *Year* may include the integers from 1896 to 2001, and *Mileage* may include the real values from 0 to 500,000. The market set M is a Cartesian product of the attribute sets; in this example, $M = Model \times Color \times Year \times Mileage$. If the market includes n attributes, each item is an n-tuple; in the car example, it is a quadruple that specifies the model, color, year, and mileage.

The Cartesian-product representation is a simplification based on the assumption that all items have the same attributes. Some markets do not satisfy this assumption; for instance, if we trade cars and bicycles on the same market, we may need two different sets of attributes. We further limit the model by assuming that every attribute is one of the three types:

- Set of explicitly listed values, such as the car model
- Interval of integer numbers, such as the year
- Interval of real values, such as the mileage

The value of a commodity may monotonically depend on some of its attributes. For example, the quality of a car decreases with an increase in mileage. If a customer is willing to buy a certain car with 20,000 miles, she will agree to accept an identical vehicle with 10,000 miles for the same price. That is, a buyer will always accept smaller mileage if it does not affect other aspects of the transaction.

When a market attribute has this property, we say that it is *monotonically decreasing*. To formalize this concept, suppose that a market has n attributes, and we consider the kth attribute. We denote attribute values of a given item by $i_1,\ldots, i_k,\ldots, i_n$, and a transaction price by p. The kth attribute is monotonically decreasing if all price and quality functions satisfy the following constraints:

- *Price monotonicity:* If *Price* is a price function for a buy or sell order, and $i_k \le i'_k$, then, for every two items $(i_1,\ldots, i_{k-1}, i_k, i_{k+1},\ldots, i_n)$ and $(i_1,\ldots, i_{k-1}, i'_k, i_{k+1},\ldots, i_n)$, we have $Price(i_1,\ldots, i_k,\ldots, i_n) \ge Price(i_1,\ldots, i'_k,\ldots, i_n)$.

- *Buy monotonicity:* If $Qual_b$ is a quality function for a buy order, and $i_k \leq i'_k$, then, for every two items $(i_1,\ldots, i_{k-1}, i_k, i_{k+1},\ldots, i_n)$ and $(i_1,\ldots, i_{k-1}, i'_k, i_{k+1},\ldots, i_n)$, and every price p, we have $Qual_b(i_1,\ldots, i_k,\ldots, i_n, p) \geq Qual_b(i_1,\ldots, i'_k,\ldots, i_n, p)$.
- *Sell monotonicity:* If $Qual_s$ is a quality function for a sell order, and $i_k \leq i'_k$, then, for every two items $(i_1,\ldots, i_{k-1}, i_k, i_{k+1},\ldots, i_n)$ and $(i_1,\ldots, i_{k-1}, i'_k, i_{k+1},\ldots, i_n)$, and every price p, we have $Qual_s(i_1,\ldots, i_k,\ldots, i_n, p) \leq Qual_s(i_1,\ldots, i'_k,\ldots, i_n, p)$.

Similarly, if the quality of commodities grows with an increase in an attribute value, we say that the attribute is *monotonically increasing*. For example, the quality of a car increases with the year of making.

Note that monotonic attributes are numeric, and we cannot apply this notion to an unordered set of values, such as car models. Also note that we do not consider partially ordered attribute sets, which is a simplification, because some attributes may be "partially monotonic." For example, Camry LX (a deluxe model) is definitely better than Camry CE (a basic model), whereas the choice between Camry CE and Sienna CE depends on a specific customer.

Theoretically, we can view the price as one of the monotonic attributes; however, its use in the implemented system is different from the other attributes.

2.2 Order Execution

We introduce the notion of a *fill*, which is a specific transaction between buyers and sellers. We first consider a trade between one buyer and one seller, and then define fills for transactions that involve multiple buyers and sellers. We use this notion to define conditions of an acceptable multi-order transaction.

2.2.1 Fills

When a buy order matches a sell order, the corresponding parties can complete a trade, which involves the delivery of appropriate items to the buyer for an appropriate price. We use the term *fill* to refer to the traded items and their price. For example, suppose that Katie has placed an order for two sports cars, and Laura is selling three red Mustangs. If the prices of these orders match, Katie may purchase two red Mustangs from Laura; in

this case, we say that two red Mustangs is a fill for her order. Formally, a fill consists of three parts: a specific item i, its price p, and the number of purchased items, denoted *size*.

If (I_b, $Price_b$, $Qual_b$, Max_b, Min_b, $Step_b$) is a buy order, and (I_s, $Price_s$, $Qual_s$, Max_s, Min_s, $Step_s$) is a matching sell order, then a fill (i, p, *size*) must satisfy the following conditions:

- $i \in I_b \cap I_s$
- $Price_s(i) \le p \le Price_b(i)$
- $Qual_b(i, p) \ge 0$ and $Qual_s(i, p) \ge 0$
- $\max(Min_b, Min_s) \le size \le \min(Max_b, Max_s)$
- *size* is divisible by $Step_b$ and $Step_s$

Note that a fill consists of a specific item, price, and size; unlike an order, it cannot include a set of items or a range of sizes. Furthermore, all items in a fill have the same price; for instance, a fill (`red-Mustang`, \$18,000, 2) means that Katie has purchased two red Mustangs for \$18,000 each. If she had bought these cars for different prices, we would represent them as two different fills for the same order.

If both buyer and seller specify a set of items, the resulting fill can contain any item $i \in I_b \cap I_s$. Similarly, we may have some freedom in selecting the price and size of the fill; the heuristics for making these choices depend on a specific implementation.

- *Item choice:* If $I_b \cap I_s$ includes several items, we may choose an item to maximize either the buyer's quality or the seller's quality. A more complex heuristic may search for an item that maximizes the overall satisfaction of the buyer and seller.
- *Price choice:* The default strategy is to split the price difference between a buyer and seller, which means that $p = \frac{Price_b(i) + Price_s(i)}{2}$. Another standard option is to favor either the buyer or the seller; that is, we may always use $p = Price_b(i)$ or, alternatively, we may always use $p = Price_s(i)$.
- *Size choice:* We assume that buyers and sellers are interested in trading at the maximal size, or as close to the maximum as possible; thus, the fill has the largest possible size. This default is the same as in financial markets.

In Figure 2.3, we give an algorithm that finds the maximal fill size for two

matching orders. The GCD function determines the greatest common divisor of $Step_b$ and $Step_s$ using Euclid's algorithm. The main procedure finds the least common multiple of $Step_b$ and $Step_s$, denoted *step*, which equals $\frac{Step_b \cdot Step_s}{\text{GCD}(Step_b, Step_s)}$. Then, it computes the greatest size, divisible by *step*, that is between max(Min_b, Min_s) and min(Max_b, Max_s). If no fill size satisfies these constraints, the algorithm returns zero, which means that the size specification of the buy order does not match that of the sell order.

After getting a fill, the trader may keep the initial order, reduce its size, or remove the order; the default option is the size reduction. For example, if Laura has ordered a sale of three cars and gotten a two-car fill, the size of her order becomes one. If the reduced size is zero, we remove the order from the market. If the size remains positive but drops below the minimal acceptable size *Min*, the order is also removed. The process of generating a fill and then reducing the buy and sell order is called the *execution* of these orders. In Figure 2.4, we illustrate four different scenarios of order execution.

2.2.2 Multi-Fills

If a user specifies a minimal order size, she may indicate that she will accept a trade with multiple matching orders if their *total* size is no smaller than her minimal size. For example, the buy order in Figure 2.5(a) does not match either of the sell orders; however, if the user allows trades with multiple matching orders, we can generate the transaction shown in Figure 2.5(a). If the user specifies the size step, then the total size of a multi-order transaction must be divisible by this step (see Figure 2.5b).

To formalize this concept, suppose that a buyer has placed an order (I_b, $Price_b$, $Qual_b$, Max_b, Min_b, $Step_b$), and she is willing to trade with multiple sell orders. Suppose further that sellers have placed k orders, denoted as follows:

(I_1, $Price_1$, $Qual_1$, Max_1, Min_1, $Step_1$)

(I_2, $Price_2$, $Qual_2$, Max_2, Min_2, $Step_2$)

…

(I_k, $Price_k$, $Qual_k$, Max_k, Min_k, $Step_k$)

Then, the buy order matches these k sell orders if they satisfy the following conditions.

FILL-SIZE($Max_b, Min_b, Step_b; Max_s, Min_s, Step_s$)
The algorithm inputs the size specification of a buy order, Max_b, Min_b, and $Step_b$, and the size specification of a matching sell order, Max_s, Min_s, and $Step_s$.

Find the least common multiple of $Step_b$ and $Step_s$:

$$step := \frac{Step_b \cdot Step_s}{\text{GCD}(Step_b, Step_s)}$$

Find the maximal acceptable size, divisible by *step*:

$$size := \left\lfloor \frac{\min(Max_b, Max_s)}{step} \right\rfloor \cdot step$$

Verify that it is not smaller than the minimal acceptable sizes:
If $size \geq Min_b$ and $size \geq Min_s$, then return *size*
Else, return 0 (no acceptable size)

GCD($Step_b$, $Step_s$)
$small := \min(Step_b, Step_s)$
$large := \max(Step_b, Step_s)$
Repeat while $small \neq 0$:
$rem := large \bmod small$
$large := small$
$small := rem$
Return *large*

Figure 2.3: Computing the fill size for two matching orders; if there is no acceptable size, the algorithm returns zero.

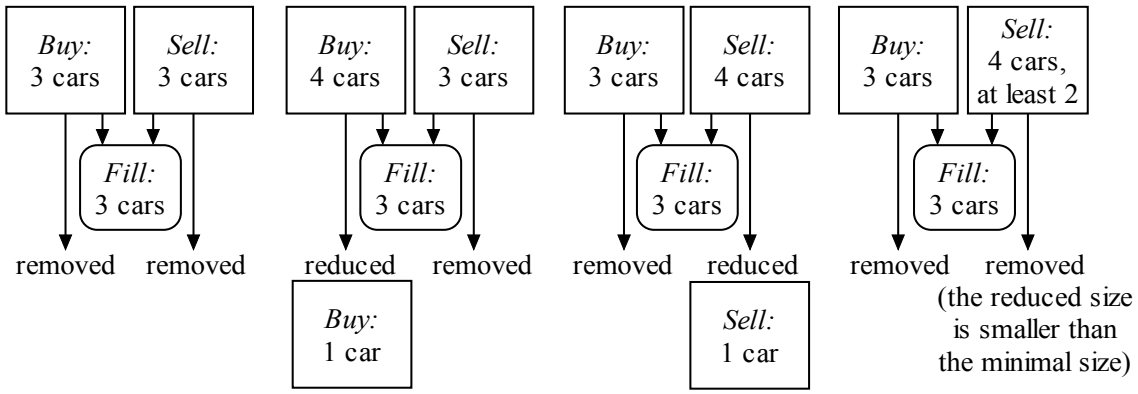


Figure 2.4: Examples of order execution.

Conditions 2.2

- For every $j \in [1..k]$, there is an item $i_j \in I_b \cap I_j$, such that $Price_j(i_j) \leq Price_b(i_j)$
- For every $j \in [1..k]$, there is a price p_j, such that
  - $Price_j(i_j) \leq p_j \leq Price_b(i_j)$
  - $Qual_b(i_j, p_j) \geq 0$ and $Qual_j(i_j, p_j) \geq 0$
- There are acceptable sizes, $size_1$, $size_2$,…, $size_k$, such that
  - For every $j \in [1..k]$, $Min_j \leq size_j \leq Max_j$
  - For every $j \in [1..k]$, $size_j$ is divisible by $Step_j$
  - $Min_b \leq size_1 + size_2 + \ldots + size_k \leq Max_b$
  - $size_1 + size_2 + \ldots + size_k$ is divisible by $Step_b$

Similarly, we can define a match between a sell order and multiple buy orders. Furthermore, we can allow transactions that involve multiple buy orders and multiple sell orders, as shown in Figure 2.6. We will define the conditions for such transactions in Section 2.2.3.

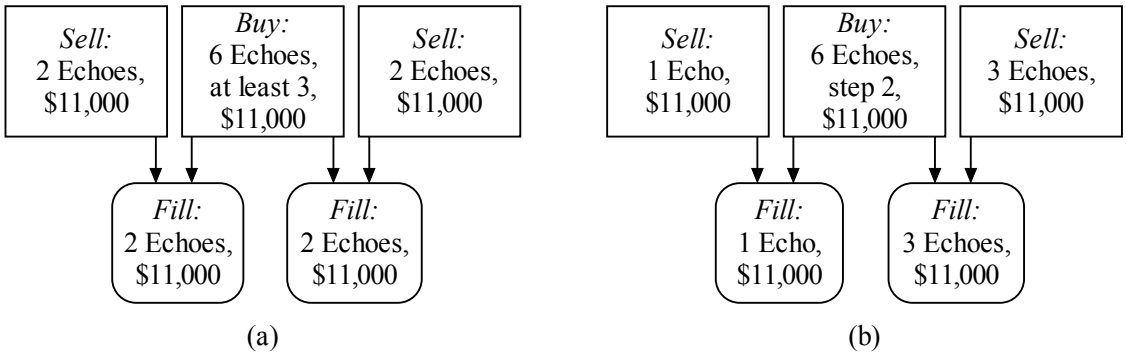


Figure 2.5: Examples of multi-order transactions.

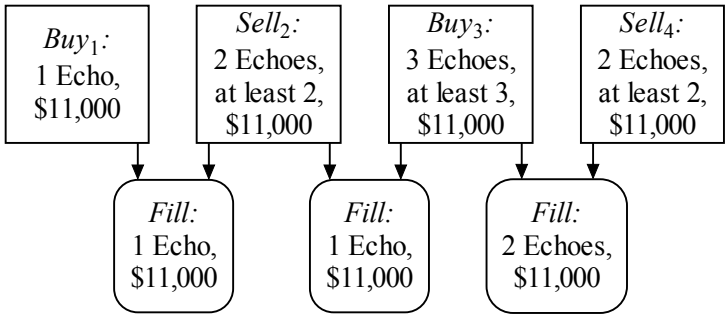


Figure 2.6: Example of a transaction that involves multiple buy and sell orders.

We refer to the result of a multi-order transaction as a *multi-fill,* which is a set of several fills for a given order. Since a multi-fill can include both purchases and sales, we denote the purchase sizes by positive integers, and the sale sizes by negative integers. For instance, the orders in Figure 2.6 get the following multi-fills:

Buy_1: {(Echo, \$11,000, 1)}

$Sell_2$: {(Echo, \$11,000, –1), (Echo, \$11,000, –1)}

Buy_3: {(Echo, \$11,000, 1), (Echo, \$11,000, 2)}

$Sell_4$: {(Echo, \$11,000, –2)}

As another example, the multi-fill {(Camry, \$20,000, 1), (Echo, \$11,000, –2)} means that a trader has bought a Camry for \$20,000 and sold two Echoes for \$11,000 each.

We say that two multi-fills have the same item set if they include the same commodities, not necessarily at the same price. For example, the multi-fill {(Echo, \$11,000, 1), (Echo, \$12,000, 1)} has the same item set as {(Echo, \$11,000, 2)}; in this example, both multi-fills represent the purchase of two Echoes. As another example, {(Camry, \$20,000, 2), (Echo, \$11,000, 1), (Echo, \$12,000, –2)} includes the same item set as {(Camry, \$20,000, 3), (Camry, \$21,000, –1), (Echo, \$11,000, –1)}; both multi-fills represent a purchase of two Camries and sale of an Echo. Finally, we define the *empty multi-fill,* denoted Ø, as the empty set of fills.

2.2.3 Equivalence of Multi-Fills

We next observe that different multi-fills may be equivalent from a trader's point of view. For instance, buying two Echoes for \$10,000 each and immediately selling one of them for the same price is equivalent to buying one car; that is, the multi-fill {(Echo, \$10,000, 2), (Echo, \$10,000, –1)} is equivalent to {(Echo, \$10,000, 1)}. As another example, if two fills include the same set of items and the same total price, most traders would consider them identical; thus, {(Echo, \$10,000, 1), (Tercel, \$12,000, 1)} is equivalent to {(Echo, \$11,000, 1), (Tercel, \$11,000, 1)}. If a multi-fill $M\text{-}Fill_1$ is equivalent to $M\text{-}Fill_2$, we write "$M\text{-}Fill_1 \equiv M\text{-}Fill_2$."

An exact definition of equivalence may vary across markets. For example, if the price is in dollars, buying two identical items for prices p_1 and p_2 is equivalent to buying each item for $\frac{p_1 + p_2}{2}$. On the other hand, if we consider the sale of mortgages and view

the interest rate as a price, this averaging rule may not work because of nonlinear growth of compound interests.

To formalize the concept of equivalence, we first define the *union of multi-fills,* which is the set of all transactions contained in these multi-fills. For example, the union of {(Echo, $10,000, 1)} and {(Echo, $10,000, 1), (Tercel, $12,000, 1)} is a three-element multi-fill {(Echo, $10,000, 1), (Echo, $10,000, 1), (Tercel, $12,000, 1)}. This definition is different from the standard union of sets since a multi-fill may include multiple identical elements. We denote the multi-fill union by "+" to distinguish it from the set union:

$$\{(i1_1, p1_1, size1_1),\ldots, (i1_m, p1_m, size1_m)\} + \{(i2_1, p2_1, size2_1),\ldots, (i2_k, p2_k, size2_k)\}$$
$$= \{(i1_1, p1_1, size1_1),\ldots, (i1_m, p1_m, size1_m), (i2_1, p2_1, size2_1),\ldots, (i2_k, p2_k, size2_k)\}.$$

A multi-fill equivalence is defined for a specific market, and it may be different for different markets. Formally, it is a relation between multi-fills that satisfies the following properties:

- Standard properties of equivalence:
  - $M\text{-}Fill \equiv M\text{-}Fill$ (reflexivity)
  - If $M\text{-}Fill_1 \equiv M\text{-}Fill_2$, then $M\text{-}Fill_2 \equiv M\text{-}Fill_1$ (symmetry).
  - If $M\text{-}Fill_1 \equiv M\text{-}Fill_2$ and $M\text{-}Fill_2 \equiv M\text{-}Fill_3$,
    then $M\text{-}Fill_1 \equiv M\text{-}Fill_3$ (transitivity).
- A transaction that involves zero items is equivalent to the empty multi-fill:
  $\{(i, p, 0)\} \equiv \varnothing$
- Buying or selling identical items separately, at the same price,
  is equivalent to buying or selling them together:
  $\{(i, p, size_1), (i, p, size_2)\} \equiv \{(i, p, size_1 + size_2)\}$.
- The union operation preserves the equivalence:
  If $M\text{-}Fill_1 \equiv M\text{-}Fill_2$, then $M\text{-}Fill_1 + M\text{-}Fill_3 \equiv M\text{-}Fill_2 + M\text{-}Fill_3$.

These conditions are the required properties of the multi-fill equivalence in all markets; in a specific market, the equivalence may have additional properties. For example, $\{(i, p, 1), (i, p, 1)\}$ is always equivalent to $\{(i, p, 2)\}$. On the other hand, $\{(i, p_1, 1), (i, p_2, 1)\}$ may be equivalent to $\{(i, \frac{p_1 + p_2}{2}, 2)\}$ in some markets, such as car trading, but not in other markets, such as mortgage sales.

We use the concept of equivalence to define conditions for a multi-order

transaction, such as the trade in Figure 2.6. Specifically, we can execute a transaction that involves k orders, denoted $Order_1$, $Order_2$,…, $Order_k$, if there exist multi-fills $M\text{-}Fill_1$, $M\text{-}Fill_2$,…, $M\text{-}Fill_k$, such that

- For every $j \in [1..k]$, $M\text{-}Fill_j$ matches $Order_j$
- $M\text{-}Fill_1 + M\text{-}Fill_2 + \ldots + M\text{-}Fill_k \equiv \emptyset$

For example, we can select the following multi-fills for orders in Figure 2.6:

- {(Echo, \$11,000, 1)} matches Buy_1
- {(Echo, \$11,000, −1), (Echo, \$11,000, −1)} matches $Sell_2$
- {(Echo, \$11,000, 1), (Echo, \$11,000, 2)} matches Buy_3
- {(Echo, \$11,000, −2)} matches $Sell_4$

The union of these multi-fills is equivalent to the empty multi-fill:

{(Echo, \$11,000, 1)} + {(Echo, \$11,000, −1), (Echo, \$11,000, −1)}
+ {(Echo, \$11,000, 1), (Echo, \$11,000, 2)} + {(Echo, \$11,000, −2)} ≡ Ø.

2.2.4 Price Averaging

A trader may sometimes accept a multi-fill even if it does not satisfy Conditions 2.2. For example, consider the transaction in Figure 2.7. The price of the second fill does not match the buy order, but the overall price of the two fills is acceptable. The buyer pays \$22,000 for two cars; thus, their average price matches the buyer's price limit. When placing an order, the trader has to specify whether she will accept such price averaging.

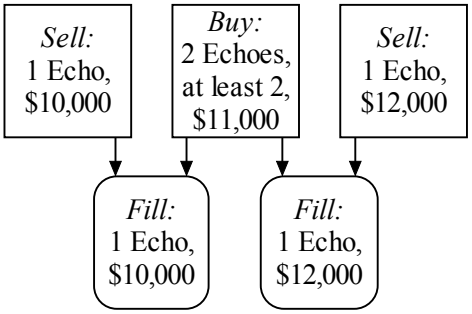


Figure 2.7: Example of price averaging.

Since the price may not be in dollars, we cannot directly compute the total price of a multi-fill. For example, if the price of a mortgage is the interest rate, the overall interest of a multi-fill is not the sum of its elements' rates. To allow price averaging, we

define a *payment* for a multi-fill. Intuitively, it represents a dollar amount delivered by a buyer or received by a seller, and the units of payment may differ from price units. For example, when a homebuyer negotiates a mortgage, she may use interest as a price measure; after receiving the mortgage, she will repay it in dollars. Formally, a payment is a real-valued function *Pay* on multi-fills that has the following properties:

- If a trader does not buy or sell any items, the payment is zero:

  $Pay(\emptyset) = 0$.
- The payment is proportional to the number of items:

  $Pay(\{(i, p, size)\}) = size \cdot Pay(\{(i, p, 1)\})$.
- The payment for multiple fills equals the sum of respective payments:

  $Pay(\{(i_1, p_1, size_1), \ldots, (i_k, p_k, size_k)\})$

  $= Pay(\{(i_1, p_1, size_1)\}) + \ldots + Pay(\{(i_k, p_k, size_k)\})$.
- Equivalent multi-fills incur the same payment:

  If $M\text{-}Fill_1 \equiv M\text{-}Fill_2$, then $Pay(M\text{-}Fill_1) = Pay(M\text{-}Fill_2)$.
- A buyer's payment is monotonically increasing on price:

  If $p_1 \leq p_2$, then $Pay(\{(i, p_1, 1)\}) \leq Pay(\{(i, p_2, 1)\})$.

Since a payment is monotonic on price, both buyers and sellers want to reduce their payments. For buyers, this reduction means paying less money; for sellers, it means getting more money, which is represented by a smaller negative value. For example, a car seller would rather get the −$12,000 payment than the −$11,000 payment, which means that she prefers selling her vehicle for $12,000 rather than for $11,000. A buyer's payment may be negative, which means that a seller pays the buyer for accepting an undesirable item. For example, if the seller wants to dispose of a broken car, she may pay $100 for pulling it away; in this case, the buyer's payment is –$100.

Note that a payment depends not only on price but also on specific items; that is, $Pay(\{(i_1, p, 1)\})$ may be different from $Pay(\{(i_2, p, 1)\})$. For example, the payment for a 6% fifteen-year mortgage is different from the payment for a 6% thirty-year mortgage. Also note that the total payment of all transaction participants is zero. For example, consider the trade in Figure 2.7. The buyer's payment is $22,000, the first seller's payment is −$10,000, and the second seller's payment is −$12,000; thus, the overall payment is $22,000 − $10,000 − $12,000 = 0.

We can decompose the payment for a multi-fill into the payments for its elements:

$Pay(\{(i_1, p_1, size_1),\ldots, (i_k, p_k, size_k)\})$

$= size_1 \cdot Pay(\{(i_1, p_1,1)\}) + \ldots + size_k \cdot Pay(\{(i_k, p_k,1)\})$.

To simplify this notation, we will usually write $Pay(i, p)$ instead of $Pay(\{(i, p, 1)\})$.

A user can also define a quality function for multi-fills. Formally, it is a real-valued function $Qual_m$ on multi-fills that satisfies the following constraints:

- If the user does not trade any items, the quality is zero:

  $Qual_m(Ø) = 0$.
- The quality function is consistent with the quality of simple fills:
  - If $size > 0$, then $Qual_m(\{(i, p, size)\}) = Qual_b(i, p)$.
  - If $size < 0$, then $Qual_m(\{(i, p, size)\}) = Qual_s(i, p)$.
- Equivalent fills have the same quality:

  If $M\text{-}Fill_1 \equiv M\text{-}Fill_2$, then $Qual_m(M\text{-}Fill_1) = Qual_m(M\text{-}Fill_2)$.
- The multi-fill union preserves relative quality of multi-fills:

  If $Qual_m(M\text{-}Fill_1) \le Qual_m(M\text{-}Fill_2)$,

  then $Qual_m(M\text{-}Fill_1 + M\text{-}Fill_3) \le Qual_m(M\text{-}Fill_2 + M\text{-}Fill_3)$.

Recall that the quality of simple fills is monotonic on price (see Section 2.1.3), which implies that the multi-fill quality is also monotonic on price:

- If $size > 0$ and $p_1 \le p_2$, then $Qual_m(\{(i, p_1, size)\}) \ge Qual_m(\{(i, p_2, size)\})$.
- If $size < 0$ and $p_1 \le p_2$, then $Qual_m(\{(i, p_1, size)\}) \le Qual_m(\{(i, p_2, size)\})$.

If the user does not provide a multi-fill quality function, we define it as the weighted mean quality of a multi-fill's elements. If the multi-fill includes purchases $\{(i_1, p_1, size_1),\ldots, (i_j, p_j, size_j)\}$ and sales $\{(i_{j+1}, p_{j+1}, -size_{j+1}),\ldots, (i_k, p_k, -size_k)\}$, the default quality is

$$Qual_m(\{(i_1, p_1, size_1),\ldots, (i_j, p_j, size_j), (i_{j+1}, p_{j+1}, -size_{j+1}),\ldots, (i_k, p_k, -size_k)\})$$

$$= \frac{size_1 \cdot Qual_b(i_1, p_1) + \ldots + size_j \cdot Qual_b(i_j, p_j) + size_{j+1} \cdot Qual_s(i_{j+1}, p_{j+1}) + \ldots + size_k \cdot Qual_s(i_k, p_k)}{size_1 + \ldots + size_j + size_{j+1} + \ldots + size_k}.$$

Now suppose that a trader has placed an order (I, $Price$, $Qual_m$, Max, Min, $Step$), and that she accepts price averaging. Then, a multi-fill $\{(i_1, p_1, size_1),\ldots, (i_k, p_k, size_k)\}$ is acceptable if it satisfies the following conditions.

Conditions 2.3

- $i_1, \ldots, i_k \in I$
- $size_1 \cdot Pay(i_1, p_1) + \ldots + size_k \cdot Pay(i_k, p_k)$
  $\leq size_1 \cdot Pay(i_1, Price(i_1)) + \ldots + size_k \cdot Pay(i_k, Price(i_k))$
- $Qual_m(\{(i_1, p_1, size_1), \ldots, (i_k, p_k, size_k)\}) \geq 0$
- $Min \leq size_1 + \ldots + size_k \leq Max$
- $size_1 + \ldots + size_k$ is divisible by *Step*

2.3 Combinatorial Orders

A combinatorial order is a collection of several orders with constraints on their execution. A simple example is a *spread,* often used in futures trading, which consists of a buy order and sell order that must be executed at the same time; for instance, a trader may place an order to buy gold futures and simultaneously sell silver futures.

Combinatorial auctions allow larger combinations of bids; for example, a trader can order a simultaneous purchase of a sport utility vehicle, trailer, boat, and two bicycles. Some auctions also support mutually exclusive bids; for instance, a user can indicate that she needs either a boat or two bicycles.

We describe combinatorial orders in the proposed exchange model, which include mutually exclusive orders, simultaneous transactions, and chains of consecutive trades.

2.3.1 Disjunctive Orders

A *disjunctive-order* mechanism is for traders who want to execute one of several alternative transactions. For example, if Katie wants to sell one of her three cars, she can place the order in Figure 2.8(a). As another example, if Katie has a trailer, she can either buy an old sport utility vehicle (SUV) for pulling it or sell the trailer (see Figure 2.8b). We have to guarantee that a trader does not get fills for two different elements of a disjunctive order. For example, if Katie places the order in Figure 2.8(a), she will sell at most one of her cars.

If a trader specifies a size for some elements of a disjunctive order, these elements must be all-or-none orders; that is, their minimal sizes must be the same as the overall

sizes. For example, Katie may place an order to sell a Camry or two Tercels, as shown in Figure 2.8(c).

A disjunctive order as a whole can also have a size, which is equivalent to placing several identical orders. For example, suppose that Katie has specified size five for the order in Figure 2.8(a). Then, she will sell five cars, and each car will be a Mustang, Tercel, or Camry. As another example, if she specifies size five for the order in Figure 2.8(b), she will complete five transactions, and each transaction will be either a purchase of a sport utility vehicle or a sale of a trailer; for instance, she may end up buying two sport utility vehicles and selling three trailers.

In addition, a disjunctive order can have a minimal size and size step. For example, suppose that a dealer is buying Camries for $18,000 and reselling them for $20,000, and she is interested in bulk transactions that involve at least ten cars. She may place the order in Figure 2.8(d); its minimal size is ten, and its step is five. If the minimal size of a disjunctive order is the same as the maximal size, it is an all-or-none order. In this case, it may be an element of another disjunctive order; it may also be an element of a conjunctive order, described in Section 2.3.2.

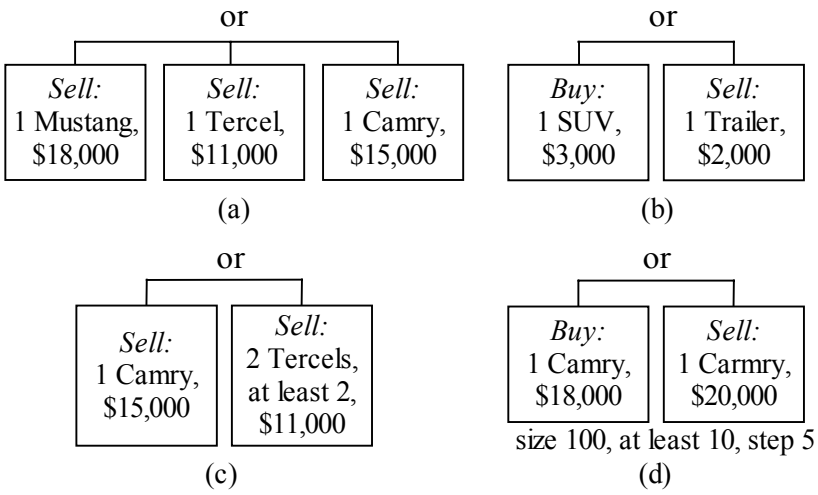


Figure 2.8: Examples of disjunctive orders.

If a trader uses quality functions in a disjunctive order, she must specify a function for every element of a disjunction. If the trader does not specify quality functions, we use the same default as for simple orders. We utilize quality functions not only for selecting the best fill for each element of a disjunction, but also for selecting

among fills for different elements. For example, suppose that a trader has placed the disjunction in Figure 2.8(b), and that she has specified a quality function $Qual_b$ for the buy element and $Qual_s$ for the sell element. Suppose further that she has found an old Explorer for $2,500, and that she can sell the trailer for $2,200. If $Qual_b$(Explorer, $2,500) > $Qual_s$(Trailer, $2,200), the trader prefers the purchase of the Explorer to the sale of the trailer.

To summarize, a disjunctive order consists of five parts:

- Set of all-or-none orders, $Order_1$, $Order_2$,…, $Order_k$
- Optional permission for price averaging
- Overall order size, *Max*
- Minimal acceptable size, *Min*
- Size step, *Step*

A multi-fill *M-Fill* matches a disjunctive order if we can decompose it into *m* multi-fills, denoted $Sub\text{-}Fill_1$, $Sub\text{-}Fill_2$,…, $Sub\text{-}Fill_m$, that match elements of the disjunction and satisfy the following constraints.

Conditions 2.4

- $Min \le m \le Max$, and *m* is divisible by *Step*
- Every multi-fill $Sub\text{-}Fill_j$ matches some element of the disjunction; that is, for every $j \in [1..m]$, there is $l \in [1..k]$ such that $Sub\text{-}Fill_j$ matches $Order_l$.
- If the order does not allow price averaging, then

  $M\text{-}Fill \equiv Sub\text{-}Fill_1 + Sub\text{-}Fill_2 + \ldots + Sub\text{-}Fill_m$.

  If the order allows price averaging, then

  M-Fill includes the same items as $Sub\text{-}Fill_1 + Sub\text{-}Fill_2 + \ldots + Sub\text{-}Fill_m$,

  and $Pay(M\text{-}Fill) = Pay(Sub\text{-}Fill_1 + Sub\text{-}Fill_2 + \ldots + Sub\text{-}Fill_m)$.

For example, suppose that a trader has placed the disjunctive order in Figure 2.8(c), and specified that its overall size is six and its minimal acceptable size is three. Then, the multi-fill {(Camry, $16,000, –2), (Tercel, $11,500, –2)} matches the order since we can decompose this multi-fill into three parts:

{(Camry, $16,000, –1)} + {(Camry, $16,000, –1)} + {(Tercel, $11,500, –2)}.

The first and second parts match the left element of the disjunction, and the third part matches the right element. After completing this transaction, we reduce the size of the disjunctive order, as shown in Figure 2.9.

2.3.2 Conjunctive Orders

A trader places a *conjunctive order* if she needs to complete several transactions together. For example, if a customer wants to sell her old Tercel and buy a new Echo, she may place the order in Figure 2.10(a). As another example, if a trader plans to buy a sport utility vehicle, trailer, and boat, she may place the order in Figure 2.10(b).

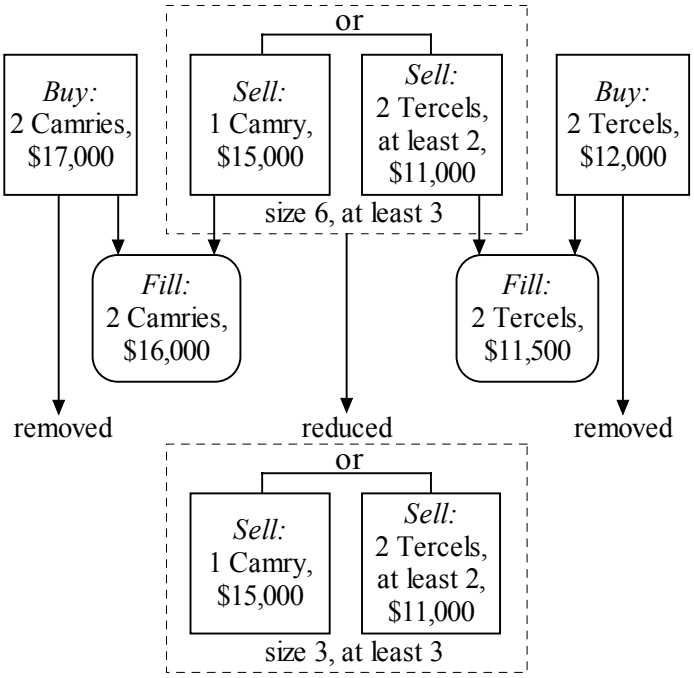


Figure 2.9: Example of a transaction that involves a disjunctive order.

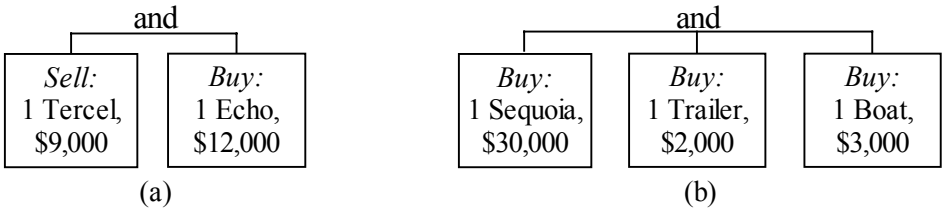


Figure 2.10: Examples of conjunctive orders.

We have to guarantee that the trader gets a fill for all elements of a conjunction at the same time. For example, the conjunction in Figure 2.10(a) can simultaneously trade with two simple orders (see Figure 2.11a). As a more complex example, it can be a part of a transaction that involves several conjunctive orders (see Figures 2.11b and 2.11c).

A disjunctive order may be an element of a conjunction. For example, if a customer wants to buy a trailer, boat, and one of several alternative vehicles, she can place the order in Figure 2.12(a). Furthermore, a conjunctive order may be an element of a disjunction, and a trader may nest several conjunctions and disjunctions (see Figure 2.12b).

If a trader specifies sizes for some elements of a conjunctive order, these elements must be all-or-none. For example, a customer may place an order to sell an old Tercel and buy two new Echoes (see Figure 2.13a). As another example, she may sell a Tercel and buy two new cars, where each new car is either an Echo or a Civic (see Figure 2.13b).

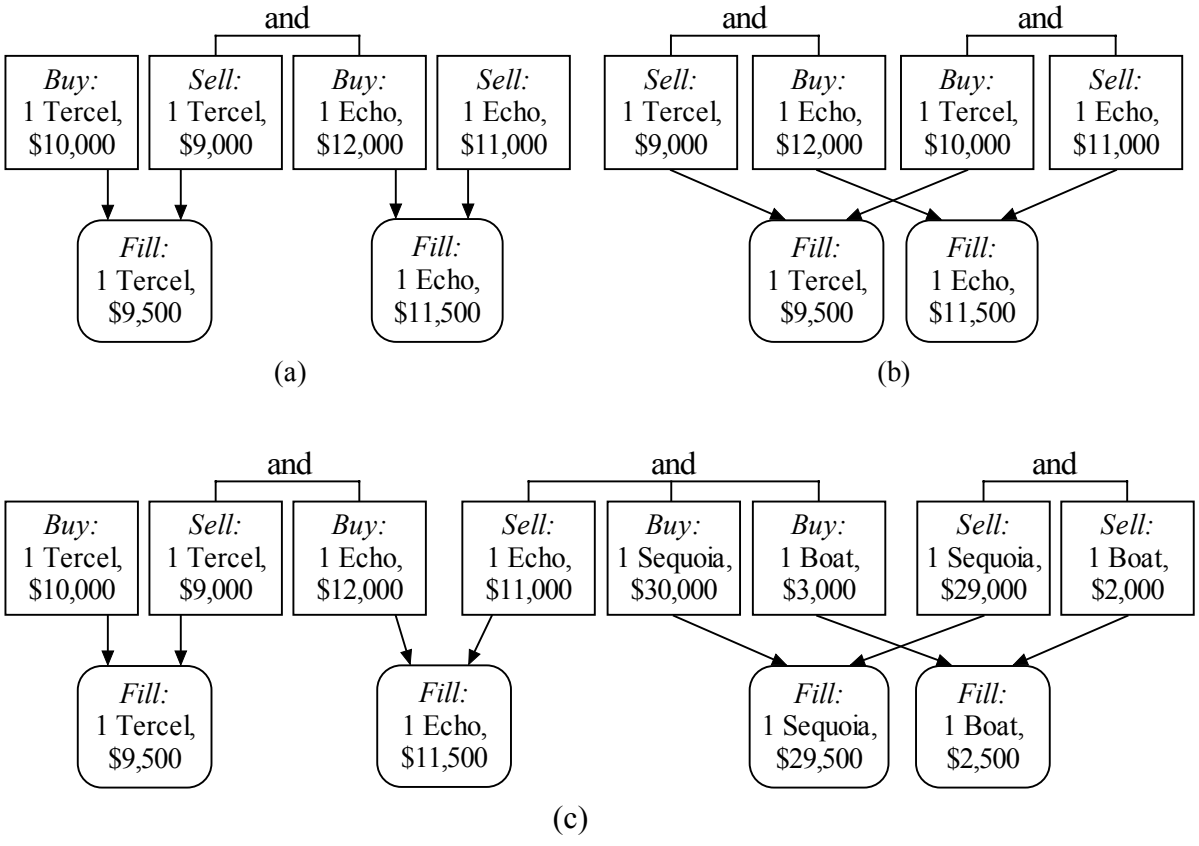


Figure 2.11: Example transactions that involve conjunctive orders.

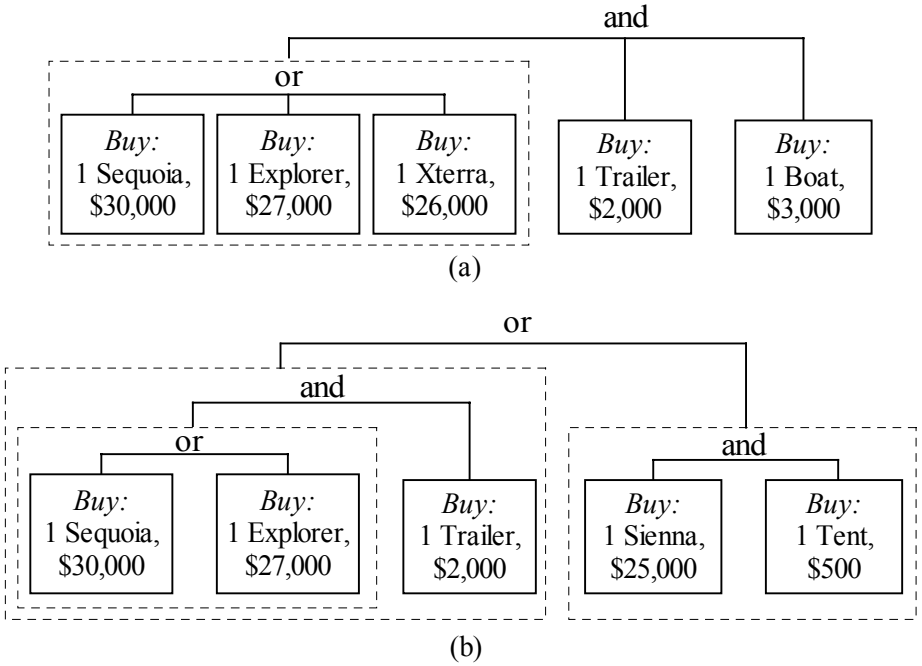


Figure 2.12: Examples of nested disjunctions and conjunctions.

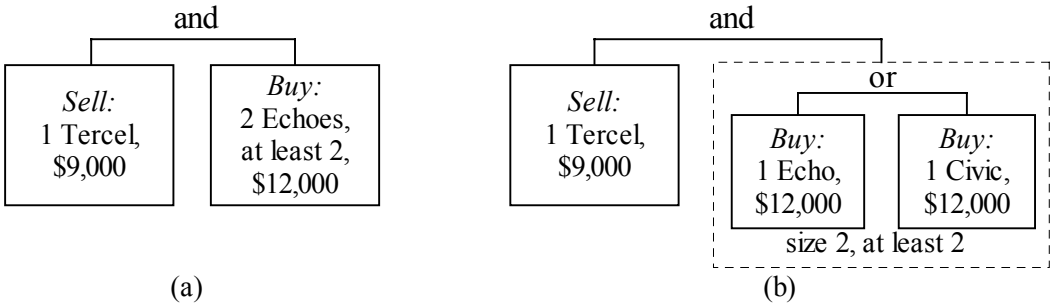


Figure 2.13: Examples of size specifications in conjunctive orders.

A conjunctive order as a whole may have a size, which is equivalent to placing several identical orders; in addition, it may have a minimal size and size step. For instance, if a trader places the order in Figure 2.14(a), she may complete two, four, or six conjunctive transactions; each transaction will involve selling a Tercel and buying an Echo. If the minimal size of a conjunctive order is the same as the overall size, then it is an all-or-none order, and it can be an element of a disjunction or another conjunction.

When a trader places a conjunctive order, she is usually interested in the price of the overall transaction rather than the prices of its elements. For example, suppose that a

customer is selling her old Tercel and buying an Echo, and she is willing to spend $3,000 for this transaction. She may sell the Tercel for $9,000 and buy an Echo for $12,000; alternatively, she may sell her old car for $8,000 and buy a new one for $11,000.

We allow two mechanisms for specifying a price limit for the overall transaction. First, a trader can set a payment limit for a conjunctive order, along with price limits for its elements. For instance, she may place the order shown in Figure 2.14(b); in this case, she wants to get at least $5,000 for her old Tercel and pay at most $15,000 for a new Echo, and her total cash spending must be at most $3,000. Thus, she is willing to sell her Tercel for $5,000 and buy an Echo for $8,000, and she is also willing to sell her car for $12,000 and buy a new one for $15,000. Recall that the units of payment may differ from price (see Section 2.2.3); for example, mortgage traders may express the price as an interest rate, and the overall payment for a conjunctive order as a dollar amount.

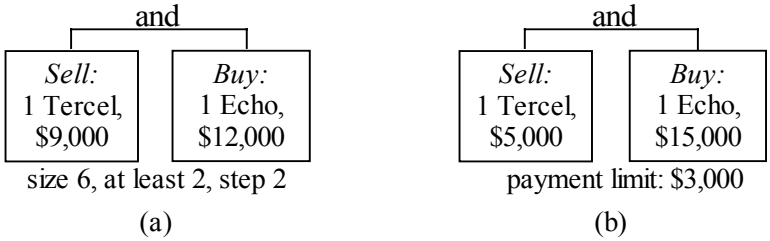


Figure 2.14: Conjunctive orders with a size specification (a) and payment limit (b).

Second, a trader can specify a price limit for each element of a conjunction, and indicate that she will accept any multi-fill that leads to the same total payment, even if the prices of individual elements do not satisfy the price limits. This option is similar to price averaging for simple orders, described in Section 2.2.4. For example, suppose that a trader uses this option for the order in Figure 2.12(a). If she gets a Sequoia with a trailer and boat, the total payment must be at most $30,000 + $2,000 + $3,000 = $35,000. If she gets an Explorer instead of a Sequoia, the total payment must be at most $27,000 + $2,000 + $3,000 = $32,000.

In addition, a trader can specify a multi-fill quality function for a conjunctive order. For instance, suppose that a trader has placed the order in Figure 2.12(a), and she prefers Sequoia to Explorer. Then, her quality function must satisfy the following constraint:

$Qual_m$({(Explorer, \$27,000, 1), (Trailer, \$2,000, 1), (Boat, \$3,000, 1)})

< $Qual_m$({(Sequoia, \$30,000, 1), (Trailer, \$2,000, 1), (Boat, \$3,000, 1)}).

A trader can also specify quality functions for elements of a conjunction, but we do not use them for selecting the best fill; their only use is to reject matches with negative quality.

To summarize, a conjunctive order consists of seven parts:

- Set of all-or-none orders, $Order_1$, $Order_2$,…, $Order_k$
- Overall payment limit, *Pay-Max*
- Multi-fill quality function, $Qual_m$
- Optional permission for price averaging
- Overall order size, *Max*
- Minimal acceptable size, *Min*
- Size step, *Step*

We next define a multi-fill that matches a conjunctive order. We first consider a conjunction of size one and then generalize the definition to larger sizes. A multi-fill *M-Fill* matches a conjunctive order of size one if it can be decomposed into multi-fills for the elements of the conjunction, denoted $Sub\text{-}Fill_1$, $Sub\text{-}Fill_2$,…, $Sub\text{-}Fill_k$, that satisfy the following conditions.

Conditions 2.5

- For every $j \in [1..k]$, $Sub\text{-}Fill_j$ matches $Order_j$
- If the order does not allow price averaging, then

  $M\text{-}Fill \equiv Sub\text{-}Fill_1 + Sub\text{-}Fill_2 + \ldots + Sub\text{-}Fill_k$.

  If the order allows price averaging, then

  M-Fill includes the same items as $Sub\text{-}Fill_1 + Sub\text{-}Fill_2 + \ldots + Sub\text{-}Fill_k$,

  and $Pay(M\text{-}Fill) = Pay(Sub\text{-}Fill_1 + Sub\text{-}Fill_2 + \ldots + Sub\text{-}Fill_k)$.
- $Pay(M\text{-}Fill) \leq Pay\text{-}Max$
- $Qual_m(M\text{-}Fill) \geq 0$

For example, the multi-fill {(Tercel, \$9,000, −1), (Echo, \$12,000, 1), (Civic, \$12,000, 1)} matches the conjunctive order in Figure 2.13(b). To show the match, we decompose it into two parts:

{(Tercel, $9,000, −1)} + {(Echo, $12,000, 1), (Civic, $12,000, 1)}.

The first part matches the sell element of the conjunction, and the second part matches the disjunctive buy.

If a conjunctive order includes a size specification, then a multi-fill *M-Fills* matches the order if it can be decomposed into multi-fills $M\text{-}Fill_1$, $M\text{-}Fill_2$,…, $M\text{-}Fill_{size}$ that satisfy the following conditions.

Conditions 2.6

- $Min \le size \le Max$
- *size* is divisible by *Step*
- For every $l \in [1..size]$, $M\text{-}Fill_l$ satisfies Conditions 2.5
- If the order does not allow price averaging, then

  $M\text{-}Fills \equiv M\text{-}Fill_1 + M\text{-}Fill_2 + \ldots + M\text{-}Fill_{size.}$

  If the order allows price averaging, then

  M-Fills includes the same items as $M\text{-}Fill_1 + M\text{-}Fill_2 + \ldots + M\text{-}Fill_{size}$,

  and $Pay(M\text{-}Fills) = Pay(M\text{-}Fill_1 + M\text{-}Fill_2 + \ldots + M\text{-}Fill_{size})$.

For instance, the conjunctive order in Figure 2.14(a) matches the multi-fill {(Tercel, $9,000, −2), (Echo, $12,000, 2)}, which can be decomposed into two parts:

{(Tercel, $9,000, −1), (Echo, $12,000, 1)}

+ {(Tercel, $9,000, −1), (Echo, $12,000, 1)}.

2.3.3 Chain Orders

The *chain-order* mechanism allows execution of several transactions in a sequence. To illustrate it, suppose that Katie plans to sell two Tercels and a Rio, and to purchase a Sequoia. Because of budgetary constraints, she wants to sell all three cars before buying a new one. Suppose further that Katie wishes to acquire a trailer after buying a Sequoia. In Figure 2.15, we show the sequence of Katie's transactions, which form a chain order.

Formally, a chain order is a directed acyclic graph; its nodes are orders, and edges are temporal constraints. If the graph includes an edge from $order_1$ to $order_2$, we can execute $order_2$ only after we have completely filled $order_1$. For instance, we cannot execute Katie's buy orders before she sells her Rio and both Tercels.

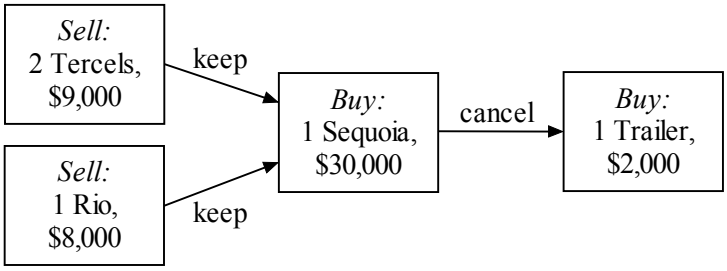


Figure 2.15: Example of a chain order. The trader first sells two Tercels and a Rio, then purchases a Sequoia, and finally acquires a trailer.

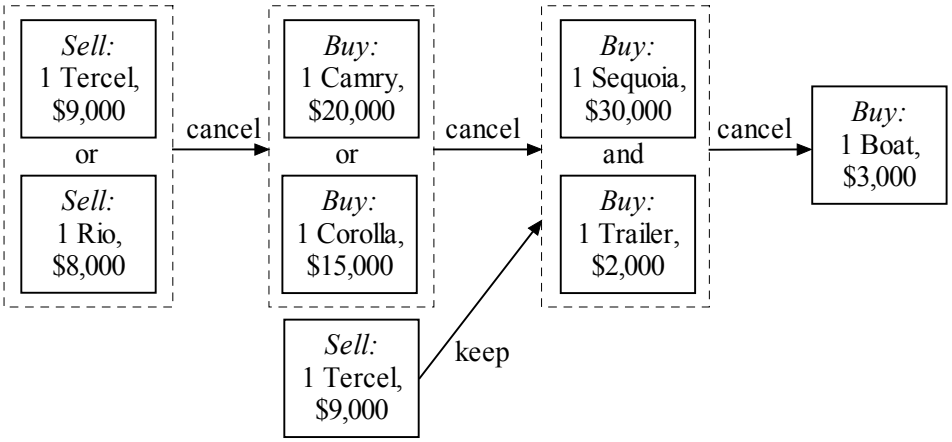


Figure 2.16: Chain order with two simple orders, two disjunctions, and a conjunction.

The elements of a chain order may be combinatorial orders; that is, the chain may include disjunctive orders, conjunctive orders, and even other chains. We do not impose any restrictions on the elements of a chain; in particular, they may not be all-or-none. In Figure 2.16, we show a chain that includes two simple orders, two disjunctions, and a conjunction.

If a trader cancels an element of a chain without getting a fill, she may want to execute the following orders; alternatively, she may want to cancel them. For each edge in the chain, the trader has to specify whether the cancellation of the earlier order causes the cancellation of the later one; in Figure 2.15, we show such specifications. In this example, if Katie cancels either sale, she is still interested in buying a Sequoia. On the other hand, if she cancels the purchase of a Sequoia, she will not buy a trailer. As another example, the removal of the leftmost disjunction in Figure 2.16 will cause the cancellation of all buy orders.

When placing a chain order, a trader may specify its size, which is equivalent to placing several identical orders. For example, if Katie specifies that the size of her order in Figure 2.15 is two, she may end up selling four Tercels and two Rios, and buying two Sequoias and two trailers. On the other hand, a chain cannot have a minimal size or size step. To summarize, a chain order consists of the following parts:

- Set of orders
- Temporal constraints that form a directed acyclic graph
- "Keep" or "cancel" specification for each constraint
- Overall order size

Since the execution of a chain includes several steps, it cannot be an all-or-none order; hence, it cannot be an element of a disjunctive or conjunctive order.

Intuitively, some elements of a chain are *inactive;* that is, they cannot lead to a trade. An element becomes *active* after the execution of all preceding elements. We illustrate this concept in Figure 2.17, where thick boxes mark active orders. The use of chain orders is a special case of activating an order upon certain conditions. We have considered three types of activation conditions in the implemented system: completion of the preceding orders in a chain, reaching a pre-set time, and a request from the user. A related problem is to develop a more general activation mechanism.

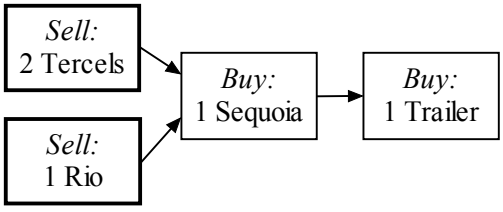


Figure 2.17: Active and inactive elements of a chain order. Thick boxes mark active orders, which can lead to immediate trades.

Chapter 3 Order Representation

We have built an exchange system for a special case of the automated trading problem. We describe the representation of orders in this system, and explain the supported operations for placing new orders, and modifying and canceling old orders.

3.1 Item Sets

We explain the use of Cartesian products for representing simple sets of items, and then introduce the notion of filter functions, which allow encoding of more complex sets.

Cartesian products. When a trader places an order, she has to specify a set of acceptable values for each market attribute, which is called an *attribute set.* Thus, if a market includes n attributes, the order description contains n attribute sets. For example, Katie may indicate that she is buying an Echo or Tercel, the acceptable colors are white, silver, and gold, the car should be made after 1998, and it should have at most 30,000 miles.

To give a formal definition, suppose that the set of all possible values for the first attribute is M_1, the set of all values for the second attribute is M_2, and so on, which means that the market set is $M = M_1 \times M_2 \times \ldots \times M_n$. The trader has to specify a set $I_1 \subseteq M_1$ of values for the first attribute, a set $I_2 \subseteq M_2$ of values for the second attribute, and so on. The resulting set I of acceptable items is the Cartesian product of the attribute sets:

$$I = I_1 \times I_2 \times \ldots \times I_n.$$

For instance, Katie may specify the following item set:

$$I = \{\mathsf{Echo}, \mathsf{Tercel}\} \times \{\mathsf{white}, \mathsf{silver}, \mathsf{gold}\} \times [1999..2001] \times [0..30{,}000].$$

Attribute sets. A trader can use specific values or ranges for each attribute; for instance, she can specify a desired year as 2001 or as a range from 1999 to 2001. Note that ranges work only for numeric attributes, such as year and mileage.

A market specification may include certain *standard sets* of values, such as "all sports cars" and "all American cars," and a trader can use them in her orders. We have to

include all standard sets in the market description, and traders cannot define new sets. We can specify a standard set by a list of values, numeric range, or union of several ranges. For example, we can define a set of American cars as a list of specific models:

{Corvette, Mustang, Saturn, …}.

As another example, we can specify "antique-car years" as [1896..1950].

A trader can use unions and intersections in the specification of attribute sets. For instance, suppose that Katie is interested in Mustangs, Corvettes, and European sports cars. Suppose further that we have defined a standard set of all European cars, and another standard set of all sports cars. Then, Katie can represent the desired set of models as follows:

{Mustang, Corvette} ∪ (European-cars ∩ Sports-cars).

We allow an arbitrary nesting of unions and intersections. To summarize, an attribute set is one of the following constructs:

- Specific value, such as Mustang or 1998
- Range of values, such as [1999..2001]
- Standard set of values, such as all European cars
- Intersection of several attribute sets
- Union of several sets

Unions and filters. A trader can define an item set *I* as the union of several Cartesian products. For example, if she wants to buy either a used Camry or a new Echo, she can specify the following set:

$$I = \{\text{Camry}\} \times \{\text{white, silver, gold}\} \times [1995..2001] \times [0..30{,}000]$$
$$\cup\ \{\text{Echo}\} \times \{\text{white, silver, gold}\} \times \{2001\} \times [0..200].$$

Furthermore, the trader can indicate that she wants to avoid certain items; for instance, a superstitious user may want to avoid black cars with 13 miles on the odometer. In this case, the user has to provide a *filter function* that prunes undesirable items. Formally, it is a Boolean function on the set *I* that gives FALSE for unwanted items. We implement it by an arbitrary C++ procedure that inputs an item description and returns TRUE or FALSE. To summarize, the representation of an item set consists of two parts:

- A union of Cartesian products,
  $I = I1_1 \times I1_2 \times \ldots \times I1_n \cup I2_1 \times I2_2 \times \ldots \times I2_n \cup \ldots \cup Ik_1 \times Ik_2 \times \ldots \times Ik_n$
- A filter function, *Filter*: $I \rightarrow \{\text{TRUE, FALSE}\}$,
  implemented by a C++ procedure.

We do not impose restrictions on filter functions; however, if a filter prunes too many items, the system may miss some matches. To avoid this problem, a user should choose Cartesian products that tightly bound the set of acceptable items.

3.2 Price, Quality, and Size

We now explain the representation of price functions, quality functions, and order sizes.

Price threshold. If a price function is a constant, a trader specifies it by a numeric value, called a *price threshold*. If an item set is the union of several Cartesian products, the trader can specify a separate threshold for each product. For instance, if Katie's item set is the union of used Camries and new Echoes, she can indicate that she is paying $15,000 for a Camry and $12,000 for an Echo. If several Cartesian products overlap, and the trader has defined different thresholds for these products, then we use the tightest threshold for their intersection; that is, we use the lowest threshold for buy orders, and the highest threshold for sell orders.

Price function. We specify a price function by an arbitrary C++ procedure that inputs an item and outputs the corresponding price limit. Note that a trader can specify different functions for different orders. If an order includes both a threshold and price function, the system uses the tighter of the two. For example, if Katie's threshold for buying an Echo is $12,000, and her price function returns $12,500 for a specific vehicle, then the resulting price limit is $12,000. Price thresholds help to prune unacceptable items, whereas C++ price functions allow more accurate evaluation of the remaining items. If the market includes monotonic attributes, the price functions must satisfy the monotonicity condition in Section 2.1.5.

Quality function. The representation of a quality function is also an arbitrary C++ procedure; it inputs an item description and price, and outputs a numeric quality value.

The system includes several standard functions, and a trader can select among them and adjust the corresponding parameters. The system allows specifying different quality functions for different orders. If a user does not provide any quality function, the system uses a default quality measure defined through the price function (see Section 2.1.3). All quality functions must be monotonic on price (see Section 2.1.3); furthermore, if some attributes are monotonic, the quality functions must also satisfy the monotonicity condition of Section 2.1.5.

Additional data. An order description may include additional data for the use by filter, price, and quality functions. For example, we can include the warranty and previous-owner information even if we do not use it as market attributes. In this example, a price may depend on warranty, and a filter may reject the cars pre-owned by a smoker.

Size. The implemented size specification is the same as in the general model; it includes the overall size, minimal acceptable size, and size step. A trader can specify whether the system should preserve the minimal size in the case of a partial fill; if not, the system removes the minimal size after a partial fill (see Figure 3.1). The trader can also indicate whether she accepts multi-fills and allows price averaging.

3.3 Cancellations and Inactive Orders

We describe three mechanisms for removing an order from the market: immediate cancellation, expiration time, and temporary inactivation.

Cancellation. If a trader places an order and does not get a fill, she can later cancel it. If a trader has placed a combinatorial order, she can cancel the entire order or some of its elements. If she deletes an element of a disjunctive or conjunctive order, the other elements remain on market. On the other hand, a cancellation of a chain-order element may cause the deletion of other elements if the chain includes “cancel” constraints. For instance, the removal of the middle element in Figure 2.15 causes the cancellation of the rightmost element.

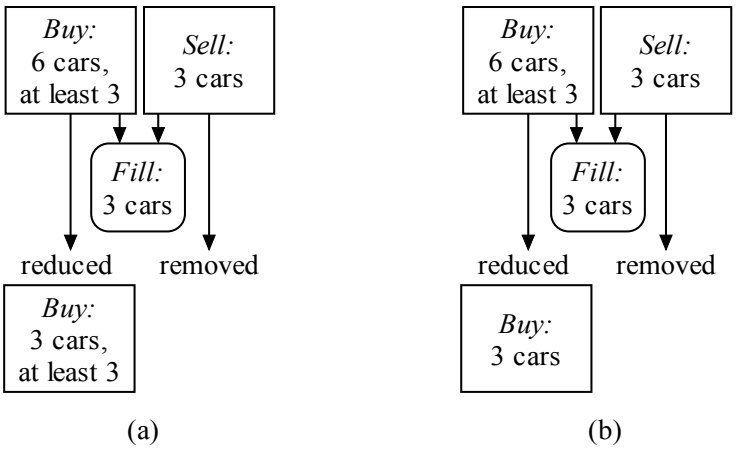


Figure 3.1: Examples of partial fills. If a trader specifies a minimal size, she indicates whether the system should preserve it after a partial fill (a), or remove it after the first fill (b).

Expiration time. When placing an order, a trader can specify its *expiration time* with one-second precision. If the system does not find a match by the specified time, it cancels the order. A trader can also place an "immediate-or-cancel" order, which is removed if there is no immediate match. When placing a combinatorial order, a trader can specify an expiration time for the whole order, as well as different times for its elements. The expiration of the whole order leads to the removal of all its elements, whereas the expiration of individual elements does not affect the other elements.

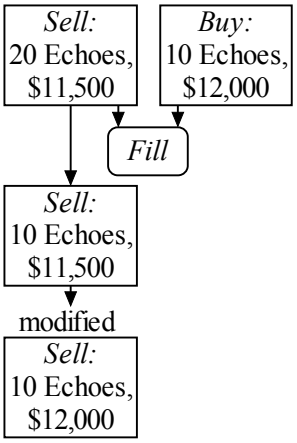
Inactive order. We can mark some orders as *inactive,* which means that they cannot lead to a trade. We have introduced this mechanism for efficiency; it allows temporary removal of an order from the trading process without deleting it from the indexing structure. In particular, we use it to delay trading with inactive elements of a chain. We also enable users to inactivate their orders by hand, and to specify inactivation and reactivation times. The system allows inactivation of combinatorial orders, but it does not support selective inactivation of their elements.

3.4 Modifications

A trader can modify her order without removing it from the market. For example, if Katie has placed an order to buy a gold Toyota Echo for $11,500 and has not gotten a fill, she can increase the price to $12,000. As another example, she can change the item

description from “gold Echo” to “any Echo or Tercel.” A trader can also define conditions that trigger a modification. For instance, suppose that Laura is selling twenty Echoes for $11,500 each. She can indicate that, if she gets a partial fill of at least ten cars, then her price increases to $12,000 (see Figure 3.2a).

We specify a modification request by a C++ procedure that inputs an order and returns its modified version; if an order requires no modification, the procedure returns the “no change” signal. For example, if Laura wants to increase her order price after its size drops to ten, she can use the procedure in Figure 3.2(b). The system includes standard modification functions, and a user can select among them and adjust the appropriate parameters.



(a)

```
CHANGE-ORDER(old-order)
If size[old-order] > 10,
    then return "no change"
new-order := old-order
price[new-order] := $12,000
Return new-order
```

(b)

Figure 3.2: Example of an order modification. If the size of the order drops to ten, the system should increase its price (a). The modification request includes a procedure that checks the size and adjusts the price (b).

A user can specify an activation and expiration time for a modification request. When it becomes active, the system invokes the corresponding procedure. If it returns “no change,” the system re-invokes it after any change to the order, which may be caused by a partial fill or by another modification. If a trader changes her mind, she can manually remove an old request. The system cancels a request in the following cases:

- The processing of the request has resulted in a modification
- The request has expired
- The corresponding order has been removed from the market
- The user has manually removed the request

A trader can also place an "immediate-or-cancel" request; if it does not result in an immediate modification, the system does not re-invoke it later.

3.5 Confirmations

When placing an order, a trader can provide not only a description used in automated matching, but also additional information for human traders; for instance, a car dealer can post a picture of a vehicle. The system enables traders to browse through potential matches and choose the most desirable trade.

When a user places an order, she can indicate the need for *confirmation.* In this case, when the system finds matches, it displays their descriptions; if the user confirms some of the matches, the system executes the corresponding trades. For example, Katie can place an order to buy a silver Corvette and require confirmation; then, she can browse through matching Corvettes and handpick the best match.

When the system finds a match between a buy and sell order, it checks the need for confirmation. If neither order requires confirmation, it immediately executes the trade. If one of the orders needs confirmation, the system notifies the corresponding user and executes the trade upon getting her approval (see Figure 3.3a). If both orders require confirmation, it notifies both sides and completes the trade only after getting both approvals (see Figure 3.3b). For example, if Katie requests confirmation for her Corvette order, and Laura sells a Corvette that also needs confirmation, then the system will complete the transaction only after getting approvals from both Katie and Laura. If the system finds a multi-order trade, it may need more than two confirmations.

A trader can confirm several different matches for her order, which allows the system to execute any of them. For instance, Katie may confirm several Corvettes, and then she will get one of them.

When the system asks users for confirmation, it does *not* remove the matching orders from the trading process, and it can find other matches for them. If a user delays her confirmation, she may miss a trade, and then the system notifies her that the trade is no longer available (see Figure 3.3c). For instance, when Katie confirms the purchase of a specific Corvette, she may find out that someone else has bought it before her. The system tries to fill orders without confirmation before sending requests for confirmation.

This strategy improves the speed of the trading process and reduces the number of "late" confirmations.

3.6 User Actions

To summarize, a trader can perform six main operations: place an order, cancel an old order, activate or inactivate an order, place a modification request, cancel an old request, and confirm a trade. The implemented system does not support changes to modification requests; if a user needs to change her old request, she should cancel it and place a new one. We list the main elements of a simple order in Figure 3.4, the elements of a combinatorial order in Figure 3.5, and the elements of a modification request in Figure 3.6. If a user does not specify some of the elements, the system uses the corresponding default values.

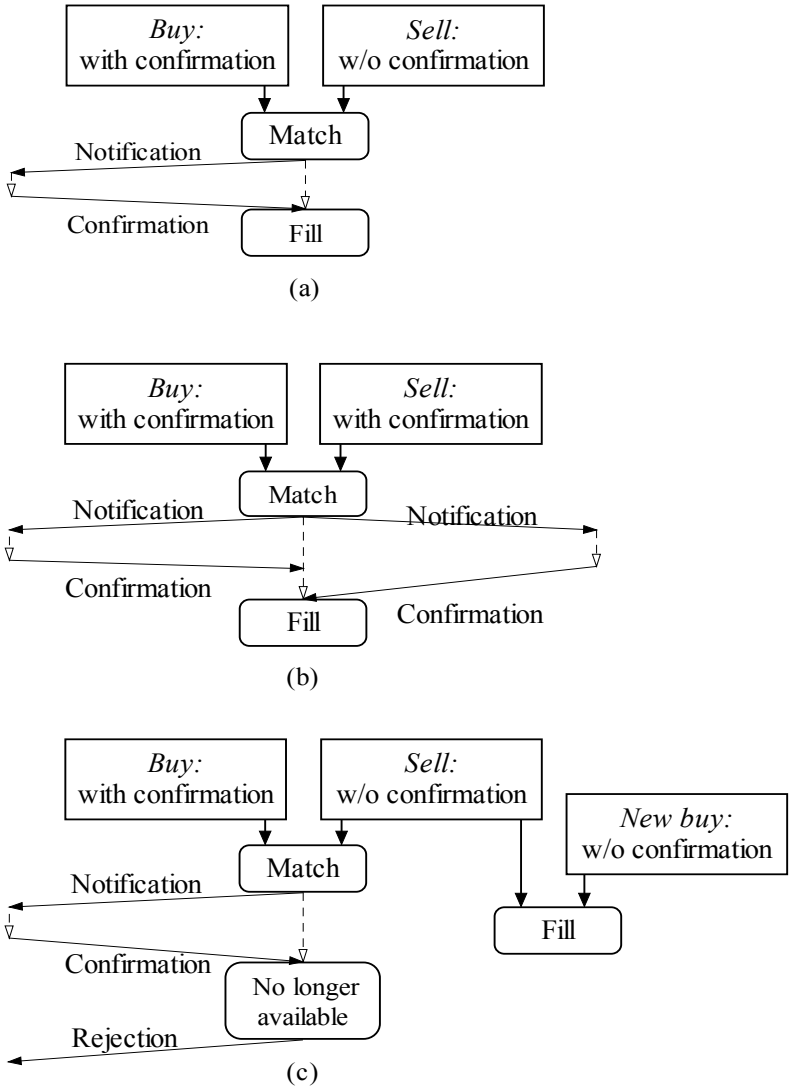


Figure 3.3: Trading with confirmations. If one of the matching orders needs confirmation, the system notifies the corresponding trader and waits for her approval (a). If both orders need confirmation, the system waits for approval from both traders (b). If it finds an alternative match before getting an approval, it executes the corresponding trade and later rejects the confirmation (c).

- *Item set*

  Union of Cartesian products (no default)

  Filter function (by default, no filter)
- *Price*

  Price threshold for each Cartesian product

  (by default, $-\infty$ for sell orders and $+\infty$ for buy orders)

  Price function (by default, equal to the threshold)

  Quality function (by default, based on the price function)
- *Additional data*

  Data for price, quality, and filter functions (by default, no data)

  Information for human traders (by default, no information)
- *Size*

  Overall order size (by default, one)

  Minimal acceptable size (by default, one)

  Size step (by default, one)
- *Activation and expiration*

  Active or inactive status (by default, active)

  Inactivation time (by default, never)

  Reactivation time (by default, never)

  Expiration time (by default, never)
- *Options*

  Acceptance of multi-fills (by default, accept)

  Acceptance of price averaging (by default, accept)

  Confirmation request (by default, no confirmation)

Figure 3.4: Elements of a simple order and their default values. When a trader places an order, she has to specify an item set, and she may optionally specify the other elements.

Disjunctive order:

- Set of all-or-none orders (no default)
- Size (the same as in a simple order)
- Activation and expiration (the same as in a simple order)
- Acceptance of price averaging (by default, accept)

Conjunctive order:

- Set of all-or-none orders (no default)
- Overall payment limit (by default, $+\infty$)
- Multi-fill quality function (by default, no function)
- Size (the same as in a simple order)
- Activation and expiration (the same as in a simple order)
- Acceptance of price averaging (by default, accept)

Chain order:

- Set of orders (no default)
- Ordering constraints (no default)
- "Keep" or "cancel" specification for each constraint (by default, "keep")
- Overall order size (by default, one)

Figure 3.5: Elements of combinatorial orders.

- Reference to a specific order (no default)
- Modification function that inputs an order and returns either a modified order or "no change" signal (no default)
- Activation time (by default, immediate)
- Expiration time (by default, never)

Figure 3.6: Elements of a modification request.

Chapter 4 Indexing Structure

We describe data structures and algorithms for fast identification of matches between buy and sell orders. We first explain the overall architecture and then present the mechanism for fast retrieval of matching orders. We refer to the orders that are currently in the system as *pending orders.*

4.1 Architecture

The system consists of a central matcher and multiple user interfaces, which run on separate machines and communicate over the network using an asynchronous messaging protocol. We outline the distributed architecture and explain the main functions of the matcher.

User interfaces. The traders enter their orders through interface machines, which send the orders to the matcher engine (see Figure 4.1). The central engine serves as a trading pit; it finds matches among orders, generates fills, and sends them to the corresponding interfaces. In Figure 4.2, we list the main types of messages from interfaces, which correspond to the user actions supported by the system (see Sections 3.3–3.6).

Matcher engine. The matcher maintains a description of market attributes, a collection of pending orders, and a queue of scheduled future events (see Figure 4.3). It includes a central structure for indexing of pending orders, implemented by two trees (see Section 4.2). This structure allows indexing of orders with fully specified items; for example, it can include an order to sell a red Mustang made in 1999, but it cannot contain an order to buy any red car made after 1999. If we can put an order into the indexing structure, we call it an *index order*. If an order includes a set of items, rather than a fully specified item, the matcher adds it to an unordered list of *nonindex orders*. In Figure 4.4, we give an example of four index orders and four nonindex orders.

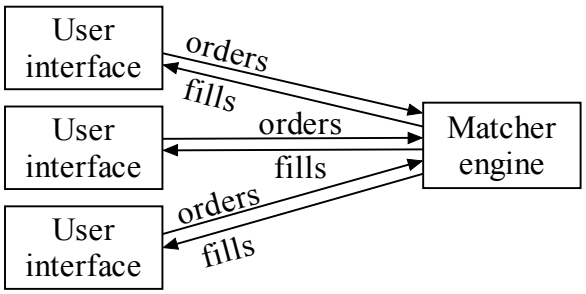


Figure 4.1: The architecture of the trading system.

- Placing an order
- Canceling an order
- Activating or inactivating an order
- Placing a modification request
- Canceling a modification request
- Confirming a trade

Figure 4.2: Main types of messages from a user interface.

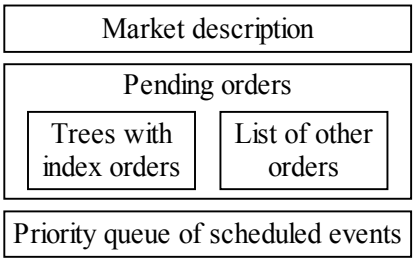


Figure 4.3: Main data structures in the matcher engine.

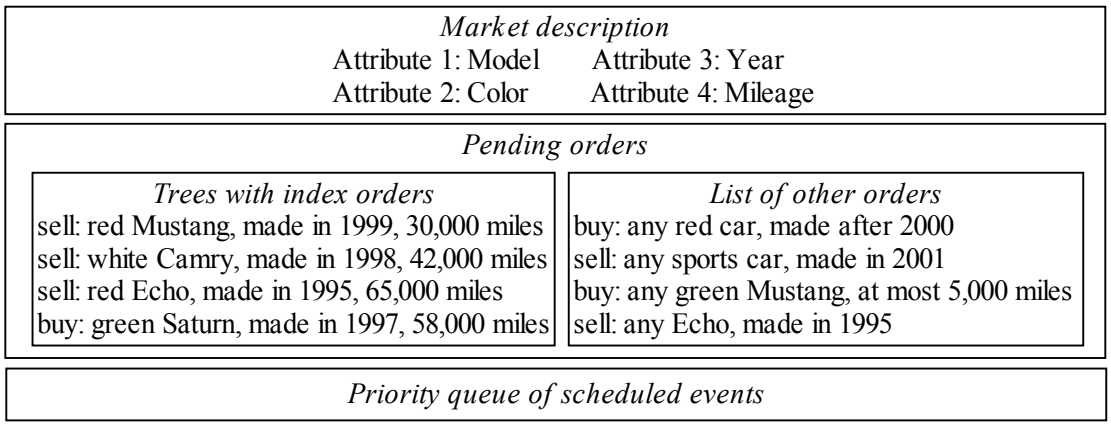


Figure 4.4: Example of index and nonindex orders.

The indexing structure allows fast retrieval of index orders that match a given order. On the other hand, the system does *not* identify matches between two nonindex orders. For example, if the orders are as shown in Figure 4.4, and a trader places an order to buy a car made after 1997, then the system will find two matches: "sell red Mustang made in 1999" and "sell white Camry made in 1998."

Matching cycle. In Figure 4.5, we show the main cycle of the matcher, which alternates between parsing new messages and searching for matches. When it receives a message with a new order, it immediately searches for matching index orders (see Figure 4.6a). If there are no matches, and the new order is an index order, then the system adds it to the indexing structure. Similarly, if the matcher fills only part of a new index order, it stores the remaining part in the indexing structure. If the system gets a nonindex order and does not find a complete fill, it adds the unfilled part to the list of nonindex orders.

For example, suppose that Laura places an order to sell a red Mustang, made in 1999, with 30,000 miles. The system immediately looks for matching index orders; if it does not find a match, it adds the order to the indexing structure. If Katie later places a buy order for a sports car, the system identifies the match with Laura's order, and informs Katie and Laura that they have exchanged a Mustang.

When the system gets a cancellation message, it removes the specified order from the market. When it receives a modification message, it makes the corresponding changes to the specified order (see Figure 4.6b). If the changes can potentially lead to new matches, the system immediately searches for index orders that match the modified order; in Figure 4.8, we list all modifications that can result in new matches. For example, if Laura has placed an order to sell a Mustang for $18,000, and she later reduces its price to $17,500, then the system immediately looks for new matches. On the other hand, if she increases the price to $18,500, the system does not search for matches.

After processing all messages, the system tries to fill pending nonindex orders, which include not only the new arrivals, but also the old unfilled orders. For each nonindex order, it identifies matching index orders, as shown in Figure 4.7. For example, consider the market in Figure 4.4, and suppose that Laura places an order to sell a green Mustang, made in 2001, with zero miles. Since the market has no matching index orders,

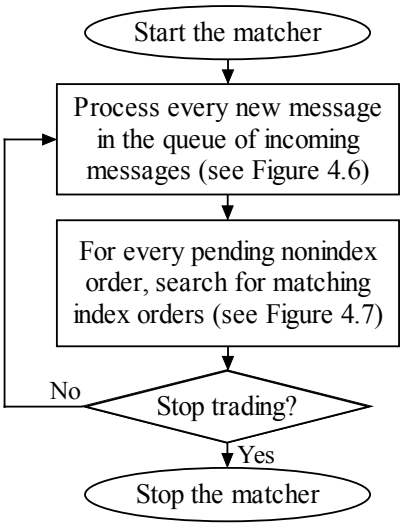
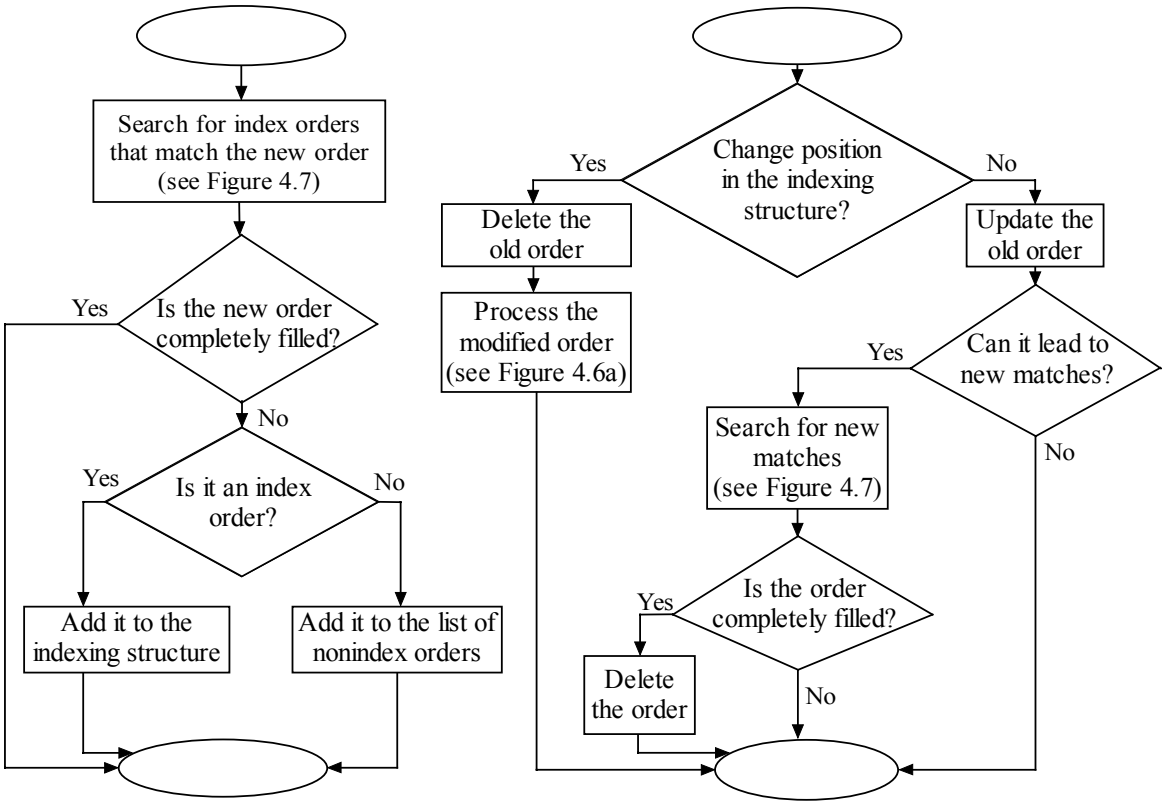


Figure 4.5: Top-level loop of the matcher engine.



(a) Processing a new order. (b) Modifying an order.

Figure 4.6: Addition and modification of an order.

the system adds this new order to the indexing structure. After processing all messages, it tries to fill the nonindex orders, and determines that Laura's order is a match for the old order to buy any green Mustang.

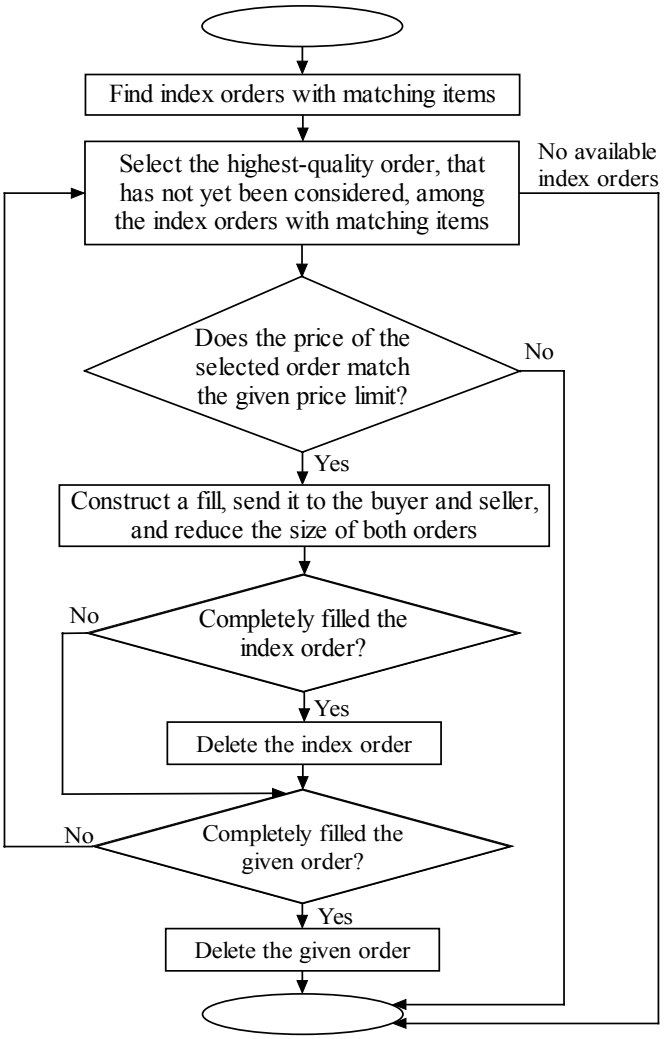


Figure 4.7: Search for index orders that match a given order.

Matching frequency. The matcher keeps track of the "age" of each order, and uses it to avoid repetitive search for matches among the same index orders. If it has already tried to find matches for some order, the matching process will involve search only among new index orders.

If a nonindex order has been on market for a long time, the system matches it less frequently than recent orders. We have implemented a mechanism that determines the intervals between searches for matching index orders; by default, the system increases the

length of an interval between consecutive searches in proportion to an order age. If it does not find a match for a new nonindex order, it repeats the search on the next matching cycle, then after two cycles, then after four cycles, and so on; that is, the intervals between searches increase as the powers of two.

- Changing the item set or additional data
- Increasing the price threshold of a buy order,
  or decreasing the threshold of a sell order
- Changing the price function or quality function
- Increasing the overall order size or reducing the minimal acceptable size
- Changing the size step in such a way that the new step is not a multiple of the old step
- Activating an inactive order
- Allowing multi-fills or price averaging
- Adding new elements to a disjunctive order,
  or deleting some elements from a conjunctive order

Figure 4.8: Order modifications that can lead to new matches.

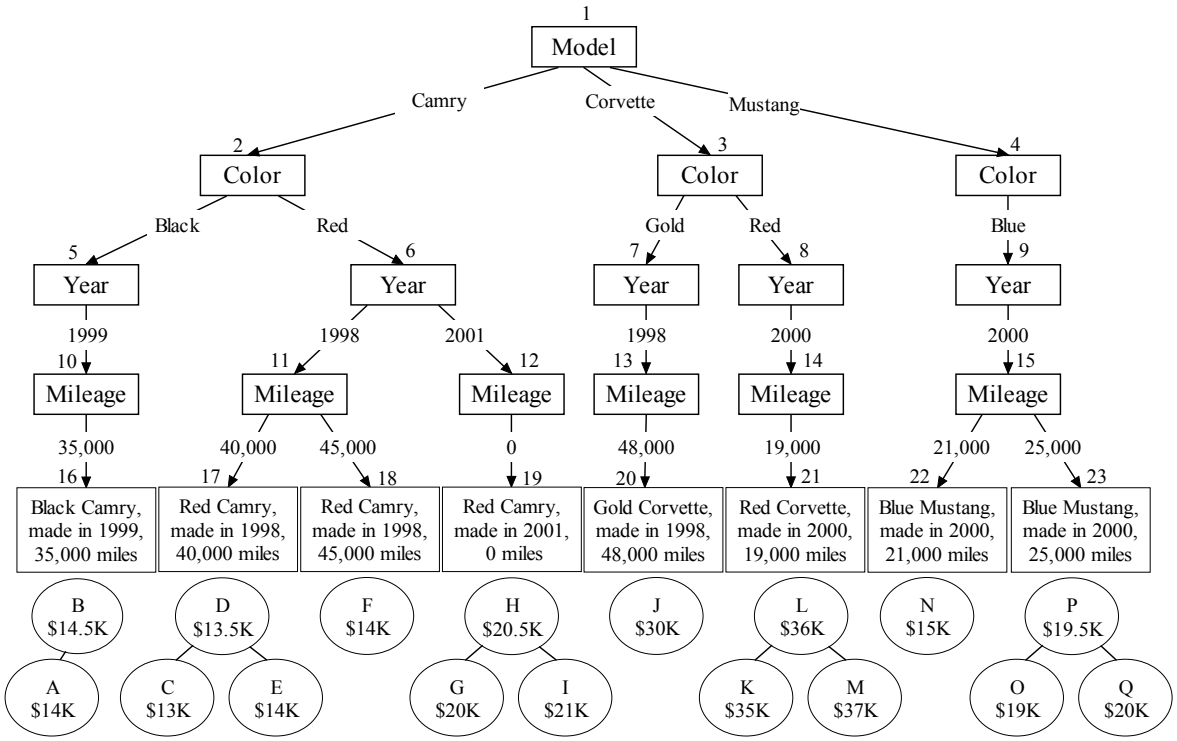
4.2 Indexing Trees

We have implemented an indexing structure for orders with fully specified items, which do not include ranges, standard sets, conjunctions, or disjunctions. The structure consists of two identical trees: one is for buy orders, and the other is for sell orders.

In Figure 4.9, we show an indexing tree for sell orders; its height is equal to the number of market attributes, and each level corresponds to one of the attributes. The root node encodes the first attribute, and its children represent different values of this attribute; in Figure 4.9, each child of the root corresponds to some car model. The nodes at the second level divide the orders by the second attribute, and each node at the third level corresponds to specific values of the first two attributes. In general, a node at level i divides orders by the values of the ith attribute, and each node at the $(i+1)$st level corresponds to all orders with a specific value of the first i attributes. If some items are

	Model	Color	Year	Mileage	Price	Size
A	Camry	Black	1999	35,000	14,000	2
B	Camry	Black	1999	35,000	14,500	1
C	Camry	Red	1998	40,000	13,000	1
D	Camry	Red	1998	40,000	13,500	2
E	Camry	Red	1998	40,000	14,000	2
F	Camry	Red	1998	45,000	14,000	2
G	Camry	Red	2001	0	20,000	2
H	Camry	Red	2001	0	20,500	1
I	Camry	Red	2001	0	21,000	1
J	Corvette	Gold	1998	48,000	30,000	1
K	Corvette	Red	2000	19,000	35,000	2
L	Corvette	Red	2000	19,000	36,000	1
M	Corvette	Red	2000	19,000	37,000	1
N	Mustang	Blue	2000	21,000	15,000	2
O	Mustang	Blue	2000	25,000	19,000	1
P	Mustang	Blue	2000	25,000	19,500	2
Q	Mustang	Blue	2000	25,000	20,000	5

(a) List of index orders.



(b) Indexing tree.

Figure 4.9: Indexing tree with seventeen orders.

not currently on sale, the tree does not include the corresponding nodes; for instance, if nobody is selling an Echo, the root has no child for Echo.

Every nonleaf node includes a red-black tree that allows fast retrieval of its children with specific values. For example, the root in Figure 4.9 includes a red-black tree that indexes its children by model values, as shown in Figure 4.10. A leaf of the indexing tree includes orders with identical items, which may have different prices and sizes. Each leaf includes a red-black tree that indexes the corresponding orders by price.

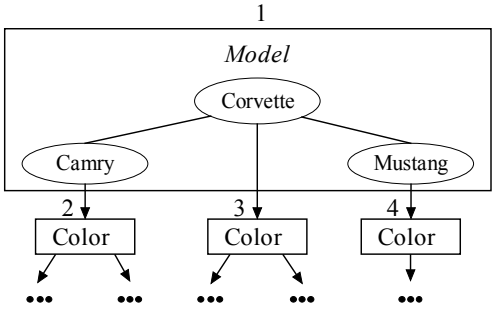


Figure 4.10: Node of an indexing tree. We arrange the attribute values in a red-black tree, and each value points to the corresponding child in the indexing tree.

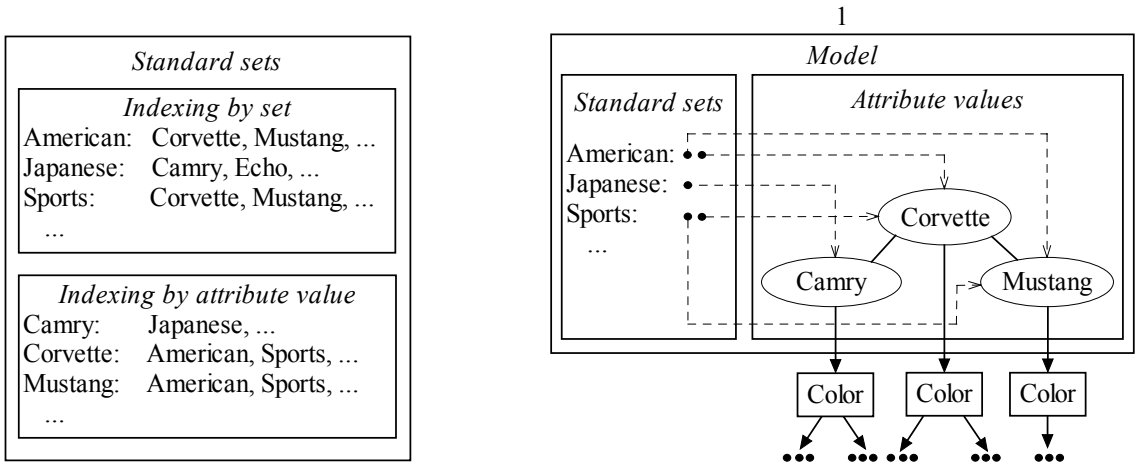
Standard sets. If a market includes standard sets of values, such as "all sports cars" and "all American cars," traders can use them in specifying their orders (see Section 3.1). We define standard sets separately for each attribute; for instance, the set of American cars belongs to the "model" attribute.

For every attribute, the system maintains a central table of standard sets, which consists of two parts (see Figure 4.11a). The first part includes a sorted list of values for every standard set; it allows determining whether a given value belongs to a specific set, by the binary search in the corresponding list. The second part includes all values that belong to at least one set; for each value, we store a sorted list of sets that include it.

Every node of an indexing tree also includes a table of standard sets; for example, the root node in Figure 4.9 includes a table of sets for the first attribute (see Figure 4.11b), and every "color" node includes a separate table of the second-attribute sets. Every set in the table includes a list of pointers to its elements in the red-black tree; for instance, the "American-cars" set points to the "Corvette" and "Mustang" nodes. If the current tree does not contain elements of some sets, we do not add these sets to the

table; for example, if the market does not include any orders to sell European cars, then the "European-cars" set is not in the table.

We have implemented the table of sets by a red-black tree, which allows fast addition and deletion of sets, as well as fast retrieval of all values in a given set. For instance, if a buyer looks for American cars, the system retrieves the appropriate children of the "model" node by finding the "American-cars" set and following its pointers.



(a) Central table of standard sets. (b) Standard sets in a node of the indexing tree.

Figure 4.11: Standard sets of values. The market includes a central table of sets (a), and every node in the indexing tree includes a table of sets for the respective attribute (b).

Summary data. The nodes of an indexing tree include summary data that help to find matching orders. Every node contains the following data about the orders in the corresponding subtree:

- The total number of orders and the total of their sizes
- The minimal and maximal price
- The minimal and maximal value for each numeric attribute
- The time of the latest addition or modification of an order

For example, consider node 2 in Figure 4.9; the subtree rooted in this node includes nine orders. If the newest of these orders was placed at 2pm, the summary data in node 2 are as follows:

- Number of orders: 9
- Total size: 14
- Prices: $13,000..21,000
- Years: 1998..2001
- Mileages: 0..45,000
- Latest addition: 2pm

4.3 Basic Tree Operations

When a user places, removes, or modifies an index order, the system has to update the indexing tree. We first describe addition and deletion algorithms, and then explain the modification procedure.

Adding a new order. When a user places an index order, the system adds it to the corresponding leaf; for example, if Laura places an order to sell a black Camry, made in 1999, with 35,000 miles, the system adds it to node 16 in Figure 4.12. If the leaf is not in the tree, the matcher adds the appropriate new branch; for example, if Laura offers to sell a white Mustang, it adds the dashed branch in Figure 4.12.

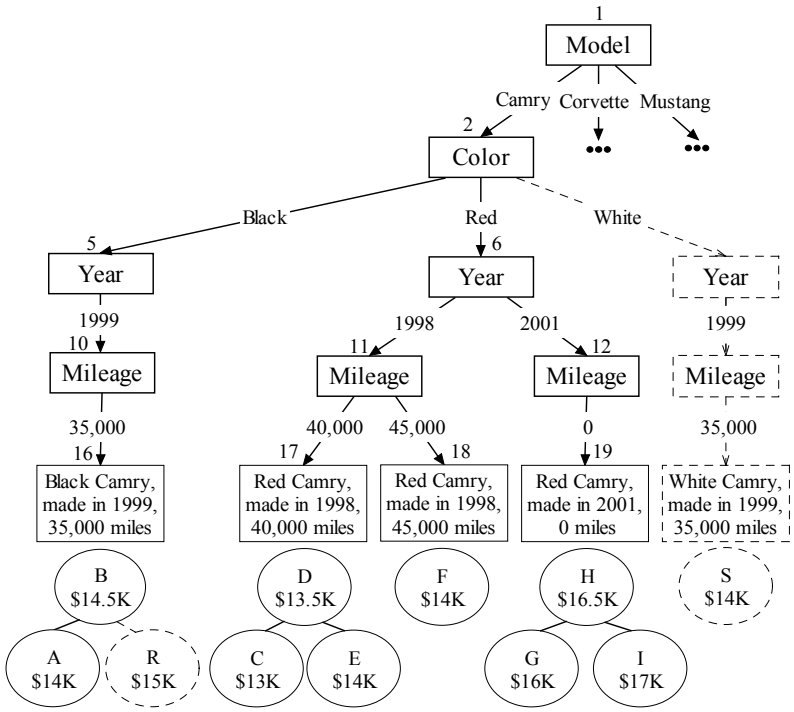
After adding a new order, the system modifies the summary data of the ancestor nodes. Note that every summary value is the minimum, maximum, or sum of the order values. In Figure 4.13, we give the algorithms for updating the number of orders, total size, and minimal price; the update of the other values is similar. These algorithms perform one pass from the leaf to the root, and their running time is proportional to the height of the tree; thus, if the market includes n attributes, the time is $O(n)$.

Deleting an order. When the matcher fills an index order, or a trader cancels her old order, the system removes the order from the corresponding leaf. If the leaf does not include other orders, the system deletes it from the indexing tree; for example, if the matcher fills order F in Figure 4.9, it removes node 18. If the deleted node is the only leaf in some subtree, the system removes this subtree; for instance, the deletion of order J leads to the removal of nodes 7, 13, and 20. We show a procedure for removing an order and the corresponding subtree in Figure 4.14.

After deleting an order, the system updates the summary data in the ancestor nodes. In Figure 4.15, we give procedures for updating the number of orders, total size, and minimal price; the modification of the other data is similar. The update time depends on the number n of market attributes, and on the number of children of the ancestor nodes, $c_1, c_2, \ldots, c_n$. If a summary value is the sum of the order values, the update time is $O(n)$; if it is the minimum or maximum of order values, the time is $O(c_1 + c_2 + \ldots + c_n)$.

	Model	Color	Year	Mileage	Price	Size
R	Camry	Black	1999	35,000	15,000	2
S	Camry	White	1999	35,000	14,000	1

(a) Two new orders.



(b) Indexing tree with new orders.

Figure 4.12: Adding orders to an indexing tree. We show new orders by dashed ovals. If the tree does not include the leaf for a new order, the system adds the proper branch.

Modifying an order. If a trader changes the order size, expiration time, or additional data, the change does not affect the structure of the indexing tree; however, the system needs to update the summary data of the ancestor nodes. If a trader modifies the price of an order, the system changes the position of the order in the red-black tree of the leaf, and propagates the price change to the summary data.

Finally, if a trader changes the item specification, the system treats it as the deletion of an old order and addition of a new one. For example, suppose that Laura has placed an order to sell a black Camry, and the indexing tree is as shown in Figure 4.12. If Laura has entered a wrong color, and she later changes it to white, then the system removes the order from the leftmost leaf in Figure 4.12 and adds it to the rightmost leaf.

ADD-COUNT (*leaf*)
The algorithm inputs the leaf with a newly added order.

node := *leaf*
Repeat while *node* ≠ NIL:
    num-orders[*node*] := *num-orders*[*node*] + 1
    node := *parent*[*node*]

ADD-SIZE (*new-size*, *leaf*)
The algorithm inputs the size of a newly added order
and the corresponding leaf of the indexing tree.

node := *leaf*
Repeat while *node* ≠ NIL:
    total-size[*node*] := *total-size*[*node*] + *new-size*
    node := *parent*[*node*]

ADD-PRICE (*new-price*, *leaf*)
The algorithm inputs the price of a newly added order
and the corresponding leaf of the indexing tree.

node := *leaf*
Repeat while *node* ≠ NIL and *min-price*[*node*] > *new-price*:
    min-price[*node*] := *new-price*
    node := *parent*[*node*]

Figure 4.13: Updating the summary data after addition of an order. We show the update of the order number (ADD-ORDER), total size (ADD-SIZE), and minimal price (ADD-PRICE).

DEL-ORDER (*order*, *leaf*)
The algorithm inputs an old order and the corresponding leaf.

Remove *order* from *leaf*
If *leaf* includes other orders, then terminate
node := *leaf*
Repeat while *parent*[*node*] ≠ NIL and *node* has no children:
    ancestor := *parent*[*node*]
    delete *node*
    node := *ancestor*

Figure 4.14: Deletion of an order. If it has been the only order in some subtree of the indexing tree, the system removes this subtree.

DEL-COUNT (*leaf*)
The algorithm inputs the leaf with a deleted order.

```
node := leaf
Repeat while node ≠ NIL:
    num-orders[node] := num-orders[node] – 1
    node := parent[node]
```

DEL-SIZE (*old-size*, *leaf*)
The algorithm inputs the size of a deleted order,
along with the leaf from which the order is deleted.

```
node := leaf
Repeat while node ≠ NIL:
    total-size[node] := total-size[node] – old-size
    node := parent[node]
```

DEL-PRICE (*old-price*, *leaf*)
The algorithm inputs the price of a deleted order,
along with the leaf from which the order is deleted.

```
If min-price[leaf] < old-price, then terminate
Update the minimal price of the leaf:
    min-price[leaf] := +∞
    For every order in the leaf:
        If min-price[leaf] > price[order],
            then min-price[leaf] := price[order]
Update the minimal prices of its ancestors:
    node := leaf
    Repeat while min-price[node] > old-price
            and parent[node] ≠ NIL and min-price[parent[node]] = old-price:
        node := parent[node]
        min-price[node] := +∞
        For every child of node:
            If min-price[node] > min-price[child],
                then min-price[node] := min-price[child]
```

Figure 4.15: Updating the summary data after deletion of an order. We show the update of the order number (DEL-COUNT), total size (DEL-SIZE), and minimal price (DEL-PRICE).

MATCHING-LEAVES(I, *filter*, *num-leaves*, *root*)
The algorithm inputs an item description I, represented by a union of Cartesian products, a filter function, a limit on the number of retrieved leaves, and the root of an indexing tree. It returns a set of leaves that match the item description.

leaves := Ø (set of matching leaves)
num-left := *num-leaves* (limit on the number of leaves)
For each Cartesian product $I_1 \times I_2 \times \ldots \times I_n$ in the union I:
    Call FIND-LEAVES($I_1 \times I_2 \times \ldots \times I_n$, *filter*, *leaves*, 1, *num-left*, *root*)
    If *num-left* = 0, then return *leaves*
Return *leaves*

FIND-LEAVES($I_1 \times I_2 \times \ldots \times I_n$, *filter*, *leaves*, k, *num-left*, *node*)
The subroutine inputs a Cartesian product $I_1 \times I_2 \times \ldots \times I_n$, a filter function, a set of matching leaves, an attribute number k, a limit on the number of retrieved leaves, and a node of the indexing tree that corresponds to the kth attribute. It finds the matching children of the given node, recursively processes the respective subtrees, and adds matching leaves in these subtrees to the set of leaves.

If $k = n$ and *filter*(*node*) = TRUE, then:
    Add *node* to *leaves*
    num-left := *num-left* – 1
If $k < n$, then:
    Identify all children of *node* that match I_k
    For each matching *child*:
        Call FIND-LEAVES($I_1 \times I_2 \times \ldots \times I_n$, *filter*, *leaves*, k+1, *num-left*, *child*)
        If *num-left* = 0, then terminate

Figure 4.16: Retrieval of matching leaves. The algorithm inputs a set of items, represented by a union of Cartesian products, along with a filter function, a limit on the number of retrieved leaves, and a pointer to the root of an indexing tree. The FIND-LEAVES subroutine retrieves matches for one Cartesian product.

4.4 Search for Matches

We outline a procedure for retrieving the index orders that match a given order, which consists of two main steps. First, it finds the leaf nodes that match the item description of a given order; second, it identifies the highest-quality matches in these leaves. The reader may find a more detailed explanation of retrieval algorithms in the forthcoming thesis of Gong [2002].

Matching leaves. In Figure 4.16, we give an algorithm that retrieves matching leaves for a given item set. The FIND-LEAVES subroutine finds all matching leaves for a Cartesian product using depth-first search in the indexing tree. It identifies all children of the root that match the first element of the Cartesian product, and then recursively processes the corresponding subtrees. For example, suppose that a buyer is looking for a Camry or Mustang made after 1998, with any color and mileage, and the tree of sell orders is as shown in Figure 4.17. The subroutine determines that nodes 2 and 4 match the model, and then recursively processes the two respective subtrees. It identifies three matching nodes for the second attribute, three nodes for the third attribute, and finally four matching leaves; we show these nodes by thick boxes.

If a given order includes a union of several Cartesian products, we call the FIND-LEAVES subroutine for each product. If an order includes a filter function, the subroutine uses it to prune inappropriate leaves. For instance, if the filter rejects the cars made in 1999 with more than 30,000 miles, it prunes node 16 in Figure 4.17.

If an order matches a large number of leaves, the retrieval may take considerable time and slow down the matching process. To prevent this problem, we impose a limit on the number of retrieved leaves. For instance, if we allow at most three matches, and a user places an order to buy any Camry, then the system retrieves the three leftmost leaves in Figure 4.17. We use this limit to control the trade-off between the speed and quality of matches. A small limit ensures the efficiency, but reduces the chance of finding the best match.

Best matches. After the system identifies all matching leaves, it selects the highest-quality orders from these leaves. It uses the quality function of the given order to evaluate matches, and it processes the matches from the best to the worst. In Figure 4.18, we give an algorithm for identifying best matches, which arranges matching leaves in a priority queue, indexed by the quality of the best order in each leaf. At every step, the algorithm processes the best available match that has not yet been considered.

For example, suppose that a buyer places an order for six Camries or Mustangs made after 1998, and her quality measure depends only on price. The system first retrieves order A, with price $14,000 and size 2, then order B with price $14,500, then

order N with price $15,000 and size 2, and finally order O with price $19,000; we show these orders by thick ovals in Figure 4.17.

Combinatorial orders. If a trader places a disjunctive order, the system finds a match for each element of the disjunction and chooses the best of these matches. If the size of a disjunctive order is greater than one, the system reduces its size after filling one of its elements. When processing a conjunctive order, the system finds a match for each of its elements and completes all corresponding trades. If some elements do not have matches, the system does not perform trades, and repeats the search during the next matching cycle. For example, suppose that a buyer places the conjunctive order shown in Figure 4.19, and the tree of sell orders is as shown in Figure 4.17. The best match for the first element is order A, whereas the best match for the second element includes orders J and K. After finding these matches, the system completes the trade in Figure 4.19.

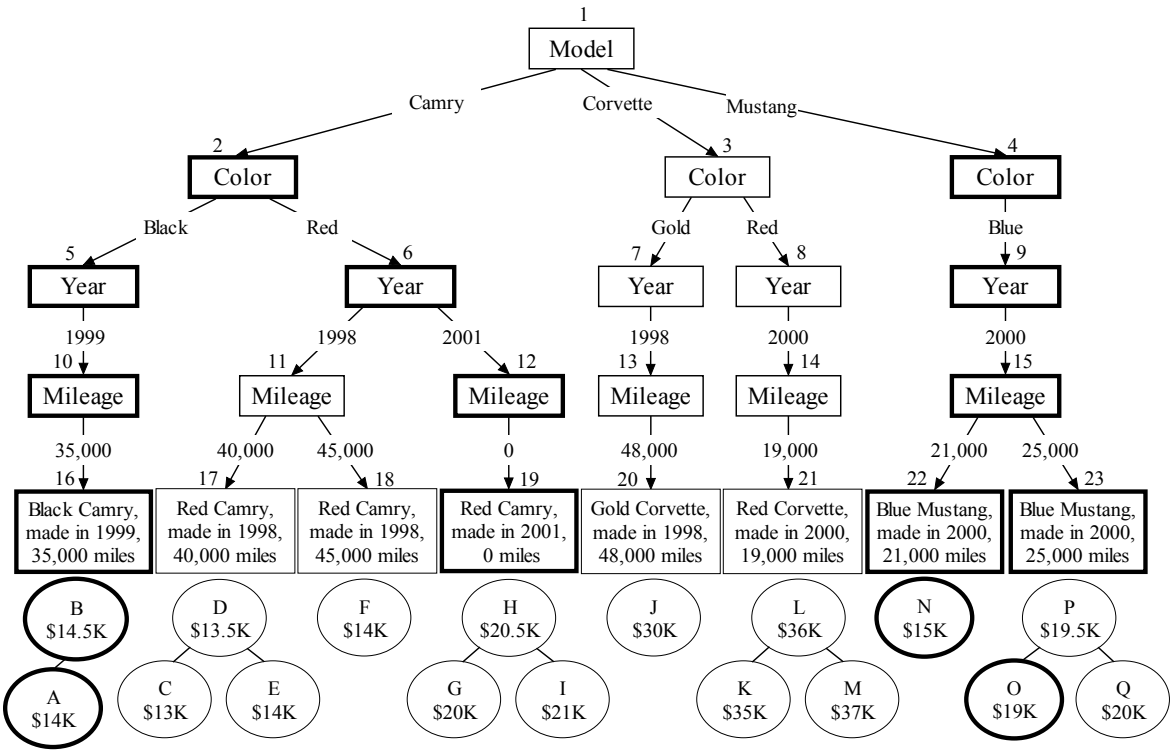


Figure 4.17: Retrieval of matches for an order to buy six Camries or Mustangs made after 1998. We show the matching nodes by thick boxes, and the retrieved orders by thick ovals.

MATCHING-ORDERS(*order*, *leaves*)
The algorithm inputs a given order and the leaves that match the order.
It removes the order after it is completely filled.

For each *leaf* in *leaves*:
    current-order[*leaf*] := *best-price-order*[*leaf*]
    quality[*leaf*] := $Qual_{order}$(*current-order*[*leaf*])
Build a priority *queue* for *leaves*, sorted by *quality*[*leaf*]
Repeat while $Max_{order} > 0$ and $Qual_{order}$(*current-order*[*top-leaf*[*queue*]]) ≥ 0:
    leaf := *top-leaf*[*queue*]
    best-order := *current-order*[*leaf*]
    current-order[*leaf*] := *next*[*current-order*[*leaf*]]
    quality[*leaf*] := $Qual_{order}$(*current-order*[*leaf*])
    Update the position of the *leaf* in the priority queue
    matching-size := FILL-SIZE(Max_{order}, Min_{order}, $Step_{order}$,
                                $Max_{best\text{-}order}$, $Min_{best\text{-}order}$, $Step_{best\text{-}order}$)
    If *matching-size* > 0, then:
        Complete the trade between *order* and *best-order*, with size of *matching-size*
        $Max_{order} := Max_{order} - matching\text{-}size$
        $Max_{best\text{-}order} := Max_{best\text{-}order} - matching\text{-}size$
        If $Max_{best\text{-}order} = 0$, then remove *best-order* from the market
    If $Max_{order} = 0$, then remove *order* from the market

Figure 4.18: Retrieval of matching orders.

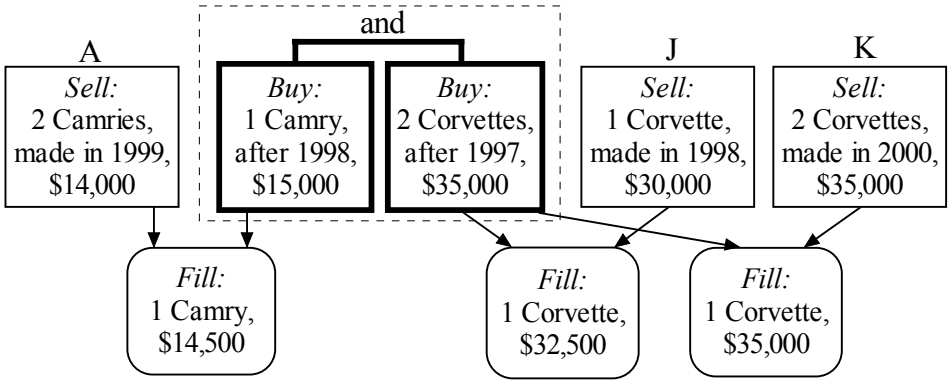


Figure 4.19. Matches for a conjunctive order. We mark the conjunction by thick boxes, and show the matching orders from the tree in Figure 4.17.

Chapter 5 Concluding Remarks

The modern economy includes a variety of electronic marketplaces, such as bulletin boards, auctions, and standardized exchanges; however, it does not yet include exchange markets for complex nonstandard commodities. The reported work is a step toward the development of automated exchanges for nonstandard goods and services. We have proposed a formal model for trading complex commodities, which allows the use of constraints and preference functions in the description of orders. We have then developed data structures for fast identification of matches between buy and sell orders.

On the negative side, the developed structures allow indexing only a certain subclass of complex orders, and we cannot find a match between two orders that do not belong to this subclass. Furthermore, the system does not guarantee finding the best-price or highest-quality matches. We plan to address these problems as part of the future work, which will involve the development of indexing structures for the retrieval of optimal matches. We are also working on a distributed version of the exchange, which will include multiple matchers and allow processing of different commodities on different computers.

References

[Andersson and Ygge, 1998] Arne Andersson and Fredrik Ygge. Managing large scale computational markets. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences,* 7, pages 4–13, 1998.

[Andersson *et al.,* 2000] Arne Andersson, Mattias Tenhunen, and Fredrik Ygge. Integer programming for combinatorial auction winner determination. In *Proceedings of the Fourth International Conference on Multi-Agent Systems,* pages 39–46, 2000.

[Bakos, 2001] Yannis Bakos. The emerging landscape for retail e-commerce. *Journal of Economic Perspectives,* 15(1), pages 69–80, 2001.

[Bapna *et al.,* 2000] Ravi Bapna, Paulo Goes, and Alok Gupta. A theoretical and empirical investigation of multi-item on-line auctions. *Information Technology and Management,* 1(1), pages 1–23, 2000.

[Bernstein, 1993] Peter L. Bernstein. *Capital Ideas: The Improbable Origins of Modern Wall Street.* The Free Press, New York, NY, 1993.

[Bichler, 2000] Martin Bichler. An experimental analysis of multi-attribute auctions. *Decision Support Systems,* 29(3), pages 249–268, 2000.

[Bichler and Segev, 1999] Martin Bichler and Arie Segev. A brokerage framework for Internet commerce. *Distributed and Parallel Databases,* 7(2), pages 133–148, 1999.

[Bichler and Segev, 2001] Martin Bichler and Arie Segev. Methodologies for the design of negotiation protocols on e-markets. *Computer Networks,* 37, pages 137–152, 2001.

[Bichler *et al.*, 1998] Martin Bichler, Arie Segev, and Carrie Beam. An electronic broker for business-to-business electronic commerce on the Internet. *International Journal of Cooperative Information Systems,* 7(4), pages 315–329, 1998.

[Bichler *et al.*, 1999] Martin Bichler, Marion Kaukal, and Arie Segev. Multi-attribute auctions for electronic procurement. In *Proceedings of the First IAC Workshop on Internet Based Negotiation Technologies,* 1999.

[Blum *et al.*, 2002] Avrim Blum, Tuomas W. Sandholm, and Martin Zinkevich. Online algorithms for market clearing. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms,* 2002.

[Cason and Friedman, 1999] Timothy Cason and Dan Friedman. Price formation and exchange in thin markets: A laboratory comparison of institutions. In Peter Howitt, Elisabetta de Antoni, and Axel Leijonhufvud, editors, *Money, Markets and Method: Essays in Honour of Robert W. Clower,* pages 155–179. Edward Elgar, Cheltenham, United Kingdom, 1999.

[Chavez and Maes, 1996] Anthony Chavez and Pattie Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology,* pages 75–90, 1996.

[Chavez *et al.*, 1997] Anthony Chavez, Daniel Dreilinger, Robert Guttman, and Pattie Maes. A real-life experiment in creating an agent marketplace. In *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology,* pages 159–178, 1997.

[Che, 1993] Yeon-Koo Che. Design competition through multidimensional auctions. *RAND Journal of Economics,* 24(4), pages 668–680, 1993.

[Cheng and Wellman, 1998] John Cheng and Michael Wellman. The WALRAS algorithm: A convergent distributed implementation of general equilibrium outcomes. *Computational Economics,* 12(1), pages 1–24, 1998.

[Conen and Sandholm, 2001] Wolfram Conen and Tuomas W. Sandholm. Minimal preference elicitation in combinatorial auctions. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, Workshop on Economic Agents, Models, and Mechanisms,* pages 71–80, 2001.

[Cripps and Ireland, 1994] Martin Cripps and Norman Ireland. The design of auctions and tenders with quality thresholds: The symmetric case. *Economic Journal,* 104(423), pages 316–326, 1994.

[Dou and Chou, 2002] Wenyu Dou and David C. Chou. A structural analysis of business-to-business digital markets. *Industrial Marketing Management,* 31, pages 165–176, 2002.

[Fujishima *et al.*, 1999a] Yuzo Fujishima, Kevin Leyton-Brown, and Yoav Shoham. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence,* 1, pages 548–553, 1999.

[Fujishima *et al.*, 1999b] Yuzo Fujishima, David McAdams, and Yoav Shoham. Speeding up ascending-bid auctions. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence,* 1, pages 554–563, 1999.

[Gonen and Lehmann, 2000] Rica Gonen and Daniel Lehmann. Optimal solutions for multi-unit combinatorial auctions: Branch and bound heuristics. In *Proceedings of the Second ACM Conference on Electronic Commerce,* pages 13–20, 2000.

[Gonen and Lehmann, 2001] Rica Gonen and Daniel Lehmann. Linear programming helps solving large multi-unit combinatorial auctions. In *Proceedings of the Electronic Market Design Workshop,* 2001.

[Gong, 2002] Jianli Gong. *Exchanges for Complex Commodities: Search for Optimal Matches.* Master thesis, Department of Computer Science and Engineering, University of South Florida, 2002. Forthcoming.

[Guttman *et al.*, 1998a] Robert H. Guttman, Alexandros G. Moukas, and Pattie Maes. Agent-mediated electronic commerce: A survey. *Knowledge Engineering Review,* 13(2), pages 147–159, 1998.

[Guttman *et al.*, 1998b] Robert H. Guttman, Alexandros G. Moukas, and Pattie Maes. Agents as mediators in electronic commerce. *International Journal of Electronic Markets,* 8(1), pages 22–27, 1998.

[Hu and Wellman, 2001] Junling Hu and Michael P. Wellman. Learning about other agents in a dynamic multiagent system. *Journal of Cognitive Systems Research,* 1, pages 67–79, 2001.

[Hu *et al.*, 1999] Junling Hu, Daniel Reeves, and Hock-Shan Wong. Agents participating in Internet auctions. In *Proceedings of the AAAI Workshop on Artificial Intelligence for Electronic Commerce,* 1999.

[Hu *et al.*, 2000] Junling Hu, Daniel Reeves, and Hock-Shan Wong. Personalized bidding agents for online auctions. In *Proceedings of the Fifth International Conference on the Practical Application of Intelligent Agents and Multi-Agents,* pages 167–184, 2000.

[Hull, 1999] John C. Hull. *Options, Futures, and Other Derivatives*. Prentice Hall, Upper Saddle River, NJ, fourth edition, 1999.

[Johnson, 2001] Joshua M. Johnson. *Exchanges for Complex Commodities: Theory and Experiments.* Master thesis, Department of Computer Science and Engineering, University of South Florida, 2001.

[Jones, 2000] Joni L. Jones. *Incompletely Specified Combinatorial Auction: An Alternative Allocation Mechanism for Business-to-Business Negotiations*. PhD thesis, Warrington College of Business, University of Florida, 2000.

[Kalagnanam *et al.*, 2000] Jayant R. Kalagnanam, Andrew J. Davenport, and Ho S. Lee. Computational aspects of clearing continuous call double auctions with assignment constraints and indivisible demand. Research Report RC21660(97613), IBM, 2000.

[Klein, 1997] Stefan Klein. Introduction to electronic auctions. *International Journal of Electronic Markets,* 7(4), pages 3–6, 1997.

[Lavi and Nisan, 2000] Ran Lavi and Noam Nisan. Competitive analysis of incentive compatible on-line auctions. In *Proceedings of the Second ACM Conference on Electronic Commerce,* pages 233–241, 2000.

[Lehmann *et al.*, 1999] Daniel Lehmann, Liadan Ita O'Callaghan, and Yoav Shoham. Truth revelation in rapid, approximately efficient combinatorial auctions. In *Proceedings of the First ACM Conference on Electronic Commerce,* pages 96–102, 1999.

[Lehmann *et al.*, 2001] Benny Lehmann, Daniel Lehmann, and Noam Nisan. Combinatorial auctions with decreasing marginal utilities. In *Proceedings of the Third ACM Conference on Electronic Commerce,* pages 18–28, 2001.

[Maes *et al.*, 1999] Pattie Maes, Robert H. Guttman, and Alexandros G. Moukas. Agents that buy and sell. *Communications of the ACM,* 42(3), pages 81–91, 1999.

[Nisan, 2000] Noam Nisan. Bidding and allocation in combinatorial auctions. In *Proceedings of the Second ACM Conference on Electronic Commerce,* pages 1–12, 2000.

[Noussair *et al.*, 1998] Charles Noussair, Stephane Robin, and Bernard Ruffieux. The effect of transaction costs on double auction markets. *Journal of Economic Behavior and Organization,* 36, pages 221–233, 1998.

[Parkes, 1999] David C. Parkes. iBundle: An efficient ascending price bundle auction. In *Proceedings of the First ACM Conference on Electronic Commerce,* pages 148–157, 1999.

[Parkes and Ungar, 2000a] David C. Parkes and Lyle H. Ungar. Iterative combinatorial auctions: Theory and practice. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence,* pages 74–81, 2000.

[Parkes and Ungar, 2000b] David C. Parkes and Lyle H. Ungar. Preventing strategic manipulation in iterative auctions: Proxy agents and price-adjustment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence,* pages 82–89, 2000.

[Parkes *et al.*, 1999] David C. Parkes, Lyle H. Ungar, and Dean P. Foster. Accounting for cognitive costs in on-line auction design. In Pablo Noriega and Carles Sierra, editors, *Agent Mediated Electronic Commerce,* pages 25–40. Springer-Verlag, New York, NY, 1999.

[Preist, 1999a] Chris Preist. Commodity trading using an agent-based iterated double auction. In *Proceedings of the Third Annual Conference on Autonomous Agents,* pages 131–138, 1999.

[Preist, 1999b] Chris Preist. Economic agents for automated trading. In Alex L. G. Hayzelden and John Bigham, editors, *Software Agents for Future Communication Systems,* pages 207–220. Springer-Verlag, Berlin, Germany, 1999.

[Reiter and Simon, 1992] Stanley Reiter and Carl Simon. Decentralized dynamic processes for finding equilibrium. *Journal of Economic Theory,* 56(2), pages 400–425, 1992.

[Rothkopf *et al.*, 1998] Michael H. Rothkopf, Aleksandar Pekec, and Ronald M. Harstad. Computationally manageable combinatorial auctions. *Management Science,* 44(8), pages 1131–1147, 1998.

[Rust and Hall, 2001] John Rust and George Hall. Middle men versus market makers: A theory of competitive exchange. Unpublished manuscript, 2001.

[Sandholm, 1999] Tuomas W. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence,* 1, pages 542–547, 1999.

[Sandholm, 2000a] Tuomas W. Sandholm. Approaches to winner determination in combinatorial auctions. *Decision Support Systems,* 28(1–2), pages 165–176, 2000.

[Sandholm, 2000b] Tuomas W. Sandholm. eMediator: A next generation electronic commerce server. In *Proceedings of the Fourth International Conference on Autonomous Agents,* pages 341–348, 2000.

[Sandholm and Suri, 2000] Tuomas W. Sandholm and Subhash Suri. Improved algorithms for optimal winner determination in combinatorial auctions and generalizations. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence,* pages 90–97, 2000.

[Sandholm and Suri, 2001] Tuomas W. Sandholm and Subhash Suri. Market clearability. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence,* pages 1145–1151, 2001.

[Sandholm *et al.*, 2001a] Tuomas W. Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. CABOB: A fast optimal algorithm for combinatorial auctions. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence,* pages 1102–1108, 2001.

[Sandholm *et al.*, 2001b] Tuomas W. Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. Winner determination in combinatorial auction generalizations. In *Proceedings of the International Conference on Autonomous Agents, Workshop on Agent-Based Approaches to B2B,* pages 35–41, 2001.

[Trenton, 1964] Rudolf W. Trenton. *Basic Economics*. Meredith Publishing Company, New York, NY, 1964.

[Turban, 1997] Efraim Turban. Auctions and bidding on the Internet: An assessment. *International Journal of Electronic Markets,* 7(4), pages 7–11, 1997.

[Vetter and Pitsch, 1999] Michael Vetter and Stefan Pitsch. An agent-based market supporting multiple auction protocols. In *Proceedings of the Workshop on Agents for Electronic Commerce and Managing the Internet-Enabled Supply Chain,* 1999.

[Wellman, 1993] Michael P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research,* 1, pages 1–23, 1993.

[Wellman and Wurman, 1998] Michael P. Wellman and Peter R. Wurman. Real time issues for Internet auctions. In *Proceedings of the First IEEE Workshop on Dependable and Real-Time E-Commerce Systems,* 1998.

[Wrigley, 1997] Clive D. Wrigley. Design criteria for electronic market servers. *International Journal of Electronic Markets,* 7(4), pages 12–16, 1997.

[Wurman, 2001] Peter R. Wurman. Toward flexible trading agents. In *Proceedings of the AAAI Spring Symposium on Game Theoretic and Decision Theoretic Agents,* pages 134–140, 2001.

[Wurman and Wellman, 1999a] Peter R. Wurman and Michael P. Wellman. Equilibrium prices in bundle auctions. In *Proceedings of the AAAI Workshop on Artificial Intelligence for Electronic Commerce,* 1999.

[Wurman and Wellman, 1999b] Peter R. Wurman and Michael P. Wellman. Control architecture for a flexible Internet auction server. In *Proceedings of the First IAC Workshop on Internet Based Negotiation Technologies,* 1999.

[Wurman and Wellman, 2000] Peter R. Wurman and Michael P. Wellman. A*k*BA: A progressive, anonymous-price combinatorial auction. In *Proceedings of the Second ACM Conference on Electronic Commerce,* pages 21–29, 2000.

[Wurman *et al.*, 1998a] Peter R. Wurman, William E. Walsh, and Michael P. Wellman. Flexible double auctions for electronic commerce: Theory and implementation. *Decision Support Systems,* 24(1), pages 17–27, 1998.

[Wurman *et al.*, 1998b] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Proceedings of the Second International Conference on Autonomous Agents,* pages 301–308, 1998.

[Wurman *et al.*, 2001] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. A parameterization of the auction design space. *Games and Economic Behavior,* 35(1–2), pages 304–338, 2001.

[Yokoo *et al.*, 2001a] Makoto Yokoo, Yuko Sakurai, and Shigeo Matsubara. Robust combinatorial auction protocol against false-name bids. *Artificial Intelligence,* 130(2), pages 167–181, 2001.

[Yokoo *et al.*, 2001b] Makoto Yokoo, Yuko Sakurai, and Shigeo Matsubara. Bundle design in robust combinatorial auction protocol against false-name bids. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence,* pages 1095–1101, 2001.