Real-Time Object-Oriented Database Support For Program Stock Trading

Victor Fay Wolfe
Assistant Professor of Computer Science

Kam Fui Lau
Doctoral Student in Managment Science

Stu Westin
Associate Professor of Information Systems

The University of Rhode Island
Kingston, RI 02881

Real-Time Object-Oriented Database Support For Program Stock Trading

ABSTRACT

Many computer systems that automatically control financial applications are required to handle large amounts of data with timing constraints imposed on the data and on the transactions that access the data. Traditional database technology, however, is not designed to manage perishable (time-constrained) data of financial applications, such as "current" stock prices and trading volumes in index arbitrage transactions. Furthermore, traditional database systems are not capable of scheduling the time-constrained transactions in financial applications, such as transactions that must be performed within time bounds of guaranteed price quotations. This paper presents a schema model of real-time object-oriented database that addresses these weaknesses and shows how this model can support a form of intelligent program stock trading called index arbitrage. This paper also illustrates the use of a protocol called timed atomic commitment, which is an extension of traditional distributed database atomic commitment protocol, to perform index arbitrage transactions using our real-time object-oriented database model.

1 Introduction

In many applications computer systems manage large amounts of data where there are timing constraints on the data and on the transactions that access the data. For instance, in database systems that support intelligent program stock trading [Miller, Miller, & Brennan, 1991], "current" stock prices and volumes of stocks can only be considered valid for short periods of time and are thus time-constrainted data. Similarly, database transactions initiated by a computer program for stock trading may be constrained to be completed before the deadline imposed by guaranteed price quotations from brokers. Such applications require a *real-time database management system.* Traditional database management systems (DBMS) are concerned with maintaining logical consistency of persistent data and providing good average response time for transactions. Although real-time DBMSs may still have persistent data, some of their data may also be *perishable* because it may become invalid after a certain amount of time (*e.g.*, stock prices). Consequently, real-time DBMSs must be concerned with both logical consistency and *temporal consistency* of data. Also, since the success of a transaction in a real-time DBMS can depend on meeting its timing constraints, the real-time DBMS must be concerned with scheduling transactions to meet timing constraints instead of simply optimizing average transaction response time.

Recently, there has been limited research directed towards developing relational real-time databases that can support such real-time applications [Ramamritham, 1993; Son, Yannopoulos, Kim & Iannacone, 1992]. Although the relational data model is useful for many applications, we believe that it is not as well-suited as an *object-oriented data* model [Zdonik & Maier, 1990] for many applications that require complex data, complex relationships among data, and first-class support for timing constraints. As we show in Section 2.2, the encapsulation and complex data representation of the object-oriented model facilitates the expression and enforcement of complex constraints found in financial databases. Unfortunately, although there has been significant recent work done in developing object-oriented databases, there has been little work done on real-time object-oriented databases. This paper describes the schema model of a real-time object-oriented database that incorporates the ability to express absolute temporal consistency constraints on a data object attribute, relative temporal consistency among attributes of a data object, and relative temporal consistency among attributes of different data objects. In addition to expressing temporal consistency of data, our real-time object-oriented database model supports expression and enforement of transaction timing constraints.

As an example of an application with time-constrained transactions, consider a form of program trading called *index arbitrage* where an investor uses program trading to take a position in a stock index futures contract on an exchange, such as the Chicago Mercantile Exchange (CME), and simultaneously takes an opposite position in a basket of stocks that replicates the underlying index of the futures contract on a stock exchange, such as the New York Stock Exchange (NYSE). We assume that the system's database is tied directly to price information from the stock market and to the futures market through a quotation services company's reporting system [Spectrum Staff, 1987]. We further assume that the system is tied to an automatic order execution system[1] that would accept orders in electronic form for the exchange-listed stocks.

Using the data stored in the database, the system computes the difference between a futures contract at the futures exchange and the actual stocks at the stock exchange. If the price difference is more than the carrying cost (the cost of short term money less the dividends received during the holding period) plus the transaction cost (broker's fees and the cost of conducting the exchange), the computer will initiate a transaction with brokers at respective exchanges. However, when the orders are actually placed, the prices for the stocks and the futures may have already changed; if the transaction is not executed while the price data is valid, the investor may turn a profit opportunity into a loss. In order to reduce the risk posed by perishable data, the investor must ensure that he can buy and sell the stocks and futures at the prices he gets from his database. One way to facilitate ensuring a guaranteed profit is for the investor to obtain a guaranteed quotation from his brokers. A guaranteed quotation is a typical price quotation along with a guarantee that the quotation will be good for a certain time interval (*e.g.* from the time of the quotation for 30 minutes).

The data's timing constraints from the guaranteed quotations in the index arbitrage application, as well as external timing constraints that may be imposed for other reasons (*e.g.* a deadline to obtain capital for another purchase, or to complete by the close of the exchange), must be used to schedule the index arbitrage transaction. There have been several techniques developed to support real-time scheduling of database transactions [Abbott & Garcia-Molina, 1988; Harista, Livny & Carey, 1990]. Most of these techniques use a deadline-driven scheduling hueristic. Unfortunately, determining how to decompose execution into transactions and how to establish the deadline for each transaction is non-trivial.

Our solution to performing such an index arbitrage uses a traditional database protocol called atomic commitment [Berstein, Hadzilacos, & Goodman, 1986]. and a recent real-time extension called timed atomic commitment [Davidson, Lee, & Wolfe, 1991]. Traditional atomic commitment requires that either all of a set of actions, such as committing writes to data objects, is performed, or none are performed. In timed atomic commitment, co-ordinating all-or-nothing behavior must be done under timing constraints. Index arbitrage

[1]DOT and Super DOT are examples of automated order execution systems in use at the NYSE.

can be considered a timed atomic commitment: Before carrying out the buying and selling of futures and stocks, the investor's program must determine from the database that a suitable profit is guaranteed and that the buy/sell actions can be performed within timing constraints. If these conditions can not be met, the investor does not initiate a purchase and loses nothing. Using the terminology of traditional atomic commitment: if both the buying and selling of futures and stocks occur, the system has committed; if neither is carried out, then the system has aborted. In an timed environment, there is another outcome: If one or both parts of the transaction miss their deadline (perhaps due to a communication failure), a loss may occur and the system is in an exception state calling for recovery. A centralized two-phase commit protocol has been developed for timed atomic commitment [Davidson, et al., 1991]; in this paper, we adopt this protocol to express a timed-constrained transaction for index arbitrage in our real-time object-oriented database schema model.

Section 2 describes related work in the fields of program stock trading, databases for financial applications, real-time databases, and real-time protocols. Section 3 presents our model of an object-oriented real-time database and illustrates its use for the index arbitrage application. Section 4 presents the timed atomic commit protocol and its application to the index arbitrage example in our real-time object-oriented database model. This section also describes our real-time object-oriented database testbed. Section 5 summarizes.

2 Application and Related Work

To motivate the need for real-time object-oriented database support for program stock trading, we first discuss the requirements of an example index arbitrage application. We then review related work on databases for financial applications, real-time databases, and object-oriented real-time databases. This review shows that current financial applications do not take advantage of new database technology like real-time databases and that current real-time databases can better support financial applications through the use of the object-oriented data model.

2.1 Index Arbitrage

Index arbitrage is one of seventeen strategies implemented using program stock trading [Market Volitility and Investor Confidence Panel, 1990]. It involves taking a position in a derivative index product, such as stock index futures, and simultaneously taking an opposite position in a basket of stocks that replicates the underlying index of the futures contract. This transaction is carried out in order to profit from the price difference between the basket and the derivative product. Suppose, for example, a "current" S&P 500 futures contract is priced lower than the underlying basket of actual stocks. An investor could make profit by "simultaneously" buying the futures contract and selling the underlying stocks.

Stock index arbitrage can use the S&P 500 as the futures leg (S&P 500 futures contract is traded at the Chicago Mercantile Exchange). The S&P 500 futures contract is based on the S&P 500 Composite Stock Price Index. It allows the investor to buy or sell, in a convenient and economical way, the equivalent of a portfolio of the 500 stocks that comprise the index. The contracts are priced at 500 times the level of the S&P 500 index. For instance, if the current S&P index level is 252, the price of the contract will be \$126,000. However, a contract can be purchased with a small down payment, with no interest charged on the outstanding balance. Theoretically, the price of the futures contract should be set at the same level as the underlying stock index (cash market). However, under certain circumstances, price differences in the markets occur and provide arbitrage opportunities to investors.

Index arbitrage between the cash and the futures markets becomes profitable when the difference between the futures price (F) and the futures "fair" value exceeds transaction costs (t). The fair value equals the cash price (S) plus expected "carrying costs" (c). There is a profitable index arbitrage opportunity whenever:

$$|F - (S + c + t)| > 0$$

If the futures price is low compared to cash prices for the stocks, the investor buys the futures contract and simultaneously sells the underlying basket of stocks. He then invests the proceeds from the transaction in risk-free securities and holds the position until the futures settlement date. If the relative futures price is high, the investor buys the stock and sells the futures contract.

Example Index Arbitrage. We now show a hypothetical instance of a successful index arbitrage. In late January of a certain year, the S&P 500 index was at 386; the T-bill rate was at 5%; the annual dividend yield of the S&P 500 was 4%; and the S&P 500 futures contract with a March 20 expiration date was at 388. If the index suddenly dropped to 380 while the futures contract remained at 388, an index arbitrage program would determine that a transaction should be carried out. The program would have computed that, since the difference had widened to 8 points (388 - 380), it would become profitable to sell the futures and buy the stocks that replicate the index. The program would determine that the transaction would require \$10,192,000. Of that figure, \$9,880,000 would buy the stock shares with a market value equivalent to 52 contracts at \$500 $*$ 380 per S&P 500 contract. An amount of \$312,000 would be used to sell an equal number of contracts (52 contracts at the \$6,000 down payment price per contract). On March 20, when the contract expired, the investor must then sell the stock involved and settle the futures contracts. No matter where the S&P 500 index closes on this date, profit is the same because it has been locked in since the time the index arbitrage program performed the transaction. Consider the computation of the investor's profit on March 20 under the following two scenarios:

Scenario 1:S&P 500 Index closes at 375		**Scenario 2**: S&P 500 Index closes at 392	
Profit from settlement of futures (388-375) * $500 * 52	$338,000	Profit from sale of $9,880,000 stock (S&P 500 rose to 392 from 380) (1 - 380/392) * $9,880,000	$312,000
Plus cash dividend received on $9,880,000 worth of stocks ($9,880,000 * 4% * 1/4)	$98,800	Plus cash dividend received on $9,880,000 worth of stocks ($9,880,000 * 4% * 1/4)	$98,800
Less loss on sale of $9,880,000 stock (S&P 500 dropped from 380 to 375) (1 - 375/380) * $9,880,000	$130,000	Less loss on settlement of futures (388-392) * $500 * 52	$104,000
Total Profit (before transaction costs)	$306,800	Total Profit (before transaction costs)	$306,800
Annual rate of return on $10,192,000 (306,800/$10,192,000) * 4 * 100%)	12.04%	Annual rate of return on $10,192,000 (306,800/$10,192,000) * 4 * 100%)	12.04%

The annual return (before transaction costs) in both instances was 12.04%, which exceeds the investor's alternative return of 5% on T-bills, making the transaction worthwhile [Amling, 1989].

Determining these price differences is greatly facilitated by the use of program stock trading. This program trading relies on "current" data from many sources as illustrated above, and on execution occurring within timing constraints imposed by the transaction. Consequently, a database for such applications should support time-constrained data, time-constrained execution, and complex relationships among data.

2.2 Related Work

Current Databases Used For Program Trading. Most database systems currently in use in financial applications are commercial products that do not provide direct support for the unique requirements of program trading. Thus, instead of being treated as a property of the database, these requirements are often either neglected or treated in an ad hoc manner in the system. Traditional index arbitrage transactions are mostly done by human experts that typically use spreadsheet programs (*e.g.* Lotus 123, Excel) and traditional commercial database programs (*e.g.* dBase, FoxPro, Oracle, etc.) to store and extract their data. Some recent work in expert systems for stock trading uses sophisticated computer technology, but still uses ordinary database support. Viau [1993] uses seven modules in a rule-based expert system called *Igor* to pre-digest relevant information for deciding whether to buy or sell stocks. Each module in *Igor* has its own database, but no special database technology is applied. Lirov, Aloni, and Grinfeld [1993] use embedded artificial intelligence for trading floor support. Their system uses heterogeneous databases for the integration of projects and resources, but their databases provide no direct support for real-time data. Jang and Lai [1993] use large scale databases of time series in stock market prediction. Their databases contain time information, but only as historical information; they do not use time as a database constraint.

Real-Time Databases. Recently there has been research to develop *real-time databases* that directly support time-constrained transactions and time-constrained data. Ramamritham [1993] provides an excellent introduction into the issues of real-time databases. One of the differences between a real-time and a non-real-time database is that the real-time database must store timestamps indicating when temporally sensitive data (like "current" stock prices) was written to the database. Using these timestamps, a real-time database must also extend the data constraint mechanism of non-real-time databases to include *absolute temporal consistency* constraints where data must be updated by a specific time, and *relative temporal consistency* constraints where timestamps of two data objects must be within a given interval of each other.

Another difference between real-time and non-real-time database management systems is that transaction scheduling for real-time should seek to optimize meeting transaction timing constraints and priorities, not necessarily average transaction throughput as is often done in non-real-time systems. Techniques to provide real-time transaction scheduling include priority-driven techniques, earliest-deadline-first techniques [Harista, et al, 1993], and hybrid scheduling techniques [Abbott & Garcia-Molina, 1988; Huang, Stankovic, Towsley, & Ramamritham, 1989]. Concurrency control techniques must also be adjusted for the requirements of real-time. Typically these techniques relax serializability as the concurrency control correctness criteria through lock-based techniques [Sha, Rajkumar, Son, & Chang, 1991], optimistic techniques [Huang, Stankovic, Towsley, & Ramaritham, 1991], and semantic concurrency control [Garcia-Molina, 1983]. Real-time database testbeds have been developed for military applications, and in academic settings, but are not currently used widely in commercial applications.

Object-Oriented Databases. Most current work on real-time databases uses the relational data model. Recently, however, there has been significant work in designing *object-oriented databases* that introduce encapsulation of complex data types with user-defined functions to manipulate the data and inheritance as a data definition tool [Zdonik and Maier, 1990]. We believe that these capabilities of object-oriented databases provide several advantages over relational databases for supporting real-time applications like program stock trading:

1. *An object-oriented data model has better modeling capability for complex real-world data.* In a relational model, fields are simple, statically typed data structures. Object-oriented data models typically allow structured data fields, as well as polymorphic and variant fields in objects of the same type. Also, the inheritance mechanism of object-oriented data models naturally provides abstraction in modeling real-world data in many applications. For example, the representation of various financial commodoties as variations on previously-defined commodities, as discussed in Section 3, is facilitated by the object-oriented inheritance mechanism and are naturally supported by an object-oriented data model's inheritance capabilities.

2. *If there are many complex inter-related classes with few instances of each class, as there would be in a financial database, the conception of and implementation of object-oriented data models are simpler.* In a relational model, this scenario would be represented by tables with a great deal of redundant indirect information used to represent the complex relationships. In an object-oriented data model, classes and relationships are expressed in the schema typing; the actual data have a simple structure.

3. *An object-oriented data model typically makes it easier to integrate constraint expression and checking as well as eliminate update anomalies as compared to relational models.* This better constraint handling in an object-oriented data model is particularly important when *timing constraints* present in real-time databases are added.

4. *The encapsulation mechanisms of an object oriented data model facilitate the enforcement of different concurrency control techniques within individual objects.* Most current relational models, including those in real-time databases, use a form of serializability as the concurrency control correctness criteria. However, serializability is used to maintain functional consistency of data (data integrity); it is not concerned with temporal consistency of the data. Moreover, associating different concurrency control correctness criteria with different data objects, which we believe is effective in real-time databases, is not easily supported in a relational model.

5. *Methods of the object-oriented data model provide better support for manipulating complex data objects.* In an object-oriented data model, it is possible to add new operations on data as a method of an object type. On the other hand, relational models have a fixed set of operations on atomic data (*e.g.* read, write, comparison); it is typically not possible to add new operations.

Other advantages of an object-oriented data model include better modularity, facilitation of software re-use, portability, support for generic programming, program access to the schema, and a uniform design paradigm from specification to implementation [Gupta, 1991]. Despite all of these advantages, the authors are unaware of any work done on developing real-time object-oriented databases for financial applications.

3 Model Description

Our approach to providing real-time database support for index arbitrage combines the permanence of data, sharing of data, and arbitrary size of data provided by typical databases; the complex data representation, user-defined operations, and modularity of the object-oriented data model, and the support for time-constrained data and time-constrained transactions of a real-time database, into a model for *real-time object-oriented databases*. We use three components to model the properties of a real-time object-oriented database schema: *objects*, *relationships* and *transactions*. *Objects* represent database entities, such as individual stocks, futures, and exchanges. *Relationships* represent associations among the database objects, such a portfolio that relates a collection of stocks. *Transactions* are executable programs which access the objects and relationships in the database. The following discussion defines our model and applies its components to index arbitrage.

Objects. An *object* is defined by $\langle N, A, M, C\rangle$. The component N is a unique name or identifier for the object. The component A is set of attributes, each of which is characterized by $\langle V, T\rangle$. V is a complex data type that represents some characteristic value of the object, and T is the time of the last update to V.

M is a set of methods which are the only means of accessing the attributes in the object. A method is defined by $\langle O, ATC\rangle$, where O is a sequence of programming language statements including: conditional branching, looping, I/O, and reads and writes to the object's attributes. Since our objects have both a value and a time field, read and write operations on attributes must read and write the time as well as the value. ATC is a set of *absolute timing constraints* on the execution of the statements in O; these constraints are formally defined later in this section.

C is a set of constraints which define the correctness of the object with respect to the system specification. A constraint is defined by $\langle Pr, ER\rangle$. Pr is a predicate represented in a boolean algebra with special atoms defined to signify the changing of an attribute ($change(a)$), the start time of an execution ($start(e)$), and the completion time of an execution ($complete(e)$). Execution e is a single executable entity such as a method invocation, or a

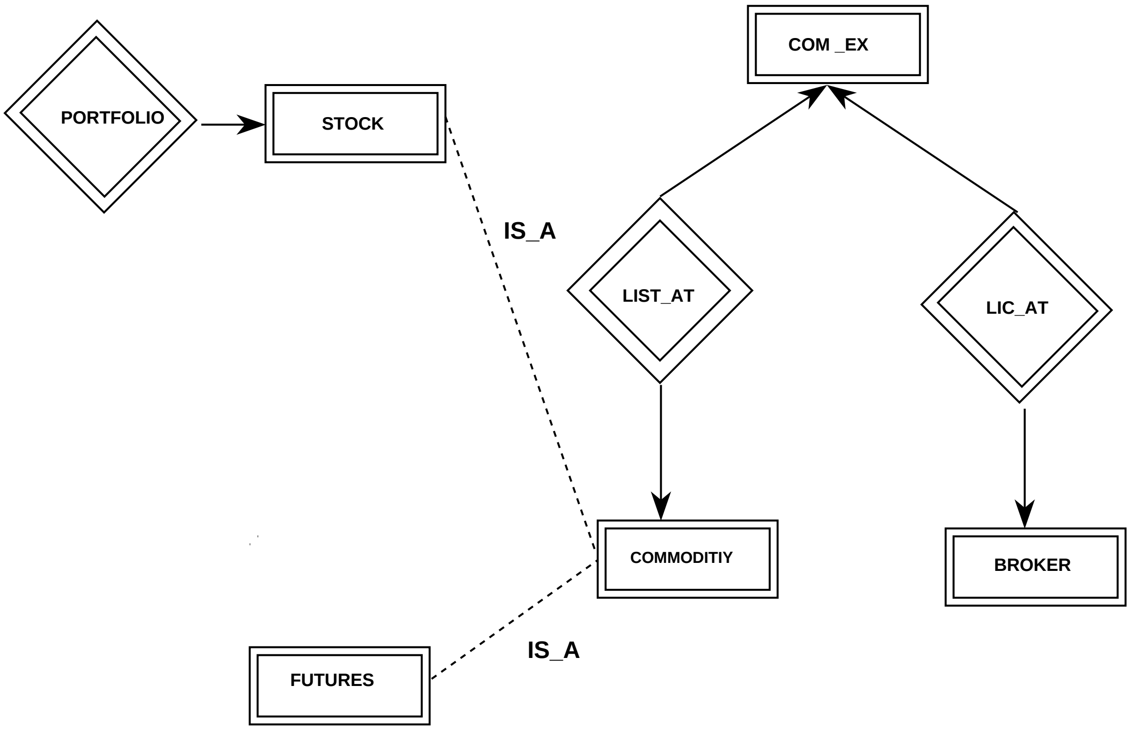

Figure 1: Type-level Model of Database for Index Arbitrage

simple read operation. Violating a constraint amounts to making the constraint's predicate false. The ER component of a constraint is an *enforcement rule* which is defined by $\langle O, ATC \rangle$. O is again a sequence of programming language statements including reads and writes on the object's attributes that is executed when the constraint is violated; and ATC is again a set of absolute timing constraints on the execution of these statements.

Figure 1 shows the types used in the database. Double rectangles are object types, double diamonds are relationship types.The object types for index arbitrage are COM_EX (Commodity Exchange), $COMMODITY$, $STOCK$, $FUTURES$ and $BROKER$. Objects are instances of the types shown in Figure 1. Figure 2 shows an example of an object representing a stock commodity in our database model. The attributes include a *price* which has a *value* and *time* field. The price can be updated or retrieved using the methods *UpdatePrice* and *GetPrice* respectively, the object and return values. The stock objects also contains constraints. Traditional data integrity constraints, such as constraining the price to be non-negative are supported; as are *temporal consistency* constraints, such as constraining the price of the stock to be valid for five minutes from when it is recorded. Detailed descriptions of each of the object types in Figure 1 are provided in Appendix A.

In Figure 3 we show an object instance-level model representing brokers, commodities and exchanges in our local database to provide information for the index arbitrage. Ellipsoids

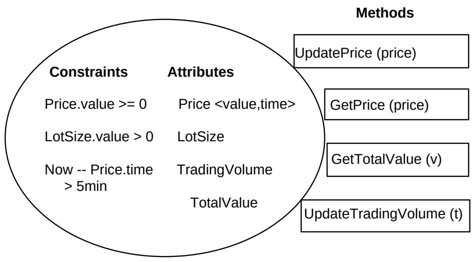

Figure 2: Example of a *Stock* Object

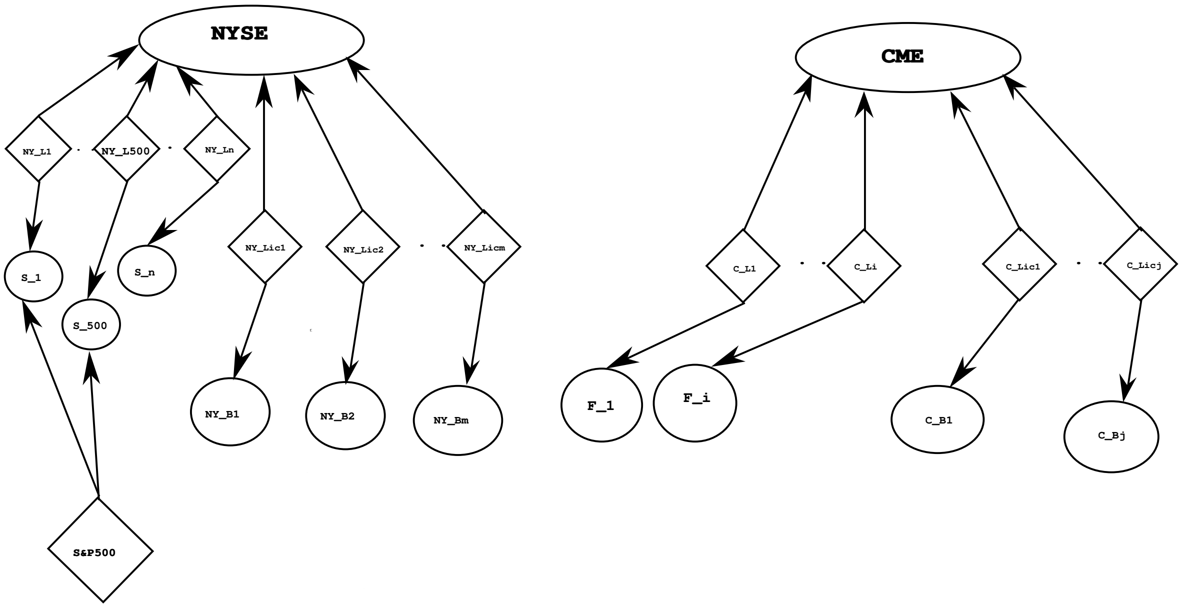

Figure 3: Object Instance Model For Index Arbitrage

represent object instances such as stocks: $S\_1 \ldots S\_n$, brokers: $NY\_B1 \ldots NY\_Bm$ and $C\_B1 \ldots C\_Bj$, futures: $F\_1 \ldots F\_i$, and the $NYSE$ and CME exchanges.

Relationships. A Relationship is an object that expresses associations among other objects; it is defined by $\langle N, A, M, C, P, IC\rangle$. The first four components are identical to the same components in the definition of an object. P is the set of objects participating in the relationship. IC is a set of inter-object constraints placed on objects in the participant set. These constraints are defined similar to object constraints by $\langle Pr, IER\rangle$, but the predicate Pr is allowed to span multiple objects in P. IER is a set of inter-object enforcement rules used to maintain the inter-object constraints. These enforcement rules are defined as before by $\langle O, ATC\rangle$, however the programming language statements in O can now include invocations of methods of the objects in P. If any changes to the elements of P cause a constraint in IC to be violated, that constraint's enforcement rule is triggered.

In Figure 1 we show that a relationship type links one or more object types together (shown in the figure by using solid arrow pointing from the relationship type to the object type(s)). For example, a $BROKER$ object type is related to a commodity exchange (COM_EX) object type by a licensed at (LIC_AT) relationship type. In addition to linking the two object types together, the relationship also stores information pertaining to the relationship, such as the broker's license number on the exchange. A relationship may exist between several instances of the same object type. For example, $PORTFOLIO$ relationship type has just one link to the $STOCK$ object type, indicating that it is a relationship among $STOCK$ instances. Detailed descriptions of each of the relationship types in Figure 1 are provided in Appendix A.

At the instance level, in Figure 3, we use diamond shapes to represent relationships. For example, the $NYSE$ object is related to broker object NY_B1 by a Licensed_At relationship called NY_Lic1. Other relationships are: Listed_At which has instances $NY\_L1 \ldots NY\_Ln$ and $C\_L1 \ldots C\_Li$ between commodities and their exchange, and $S\&P500$ which is an instance of the $PORTFOLIO$ relationship type. Since relationships are objects, they may contain: attributes, such as the license number; methods, that may invoke methods of the related objects; and constraints that may span the related objects, such as a constraint that the broker may only trade on exchanges at which he is licensed.

Transactions. Updates and queries to the database are supported by transactions. In a traditional database system, a transaction is defined as a partial order of database operations terminated by a *commit* or *abort* operation. Typically, two desired properties are *atomicity*, which requires that either the effect of all actions in the transaction are reflected in the database or none are, and *exclusivity* which ensures that no other concurrent transaction interferes. Our model of transactions is patterned after the *Real-Time Concurrency* (*RTC*) process model presented by Wolfe, Davidson, and Lee [1993], in that atomicity and exclusivity can be enforced through the use of constraints on parts of the transaction, but they are not required on the entire transaction. This relaxation of typical transaction properties allows more flexibility in creating real-time transactions [Wolfe, et al., 1993].

In our model, a *transaction* is characterized by $\langle MI, C\rangle$. *MI* is a set of method invocations on one or more objects and/or relationships in the database. C is a set of constraints placed on the execution of these method invocations defined again by $\langle Pr, ER\rangle$. Some of the types of constraints that may be placed on a transaction follow.

- *Precedence* constraints define an irreflexive partial ordering on *MI*. That is, if $m_i, m_j \in$

MI such that $m_i \prec m_j$ then either m_i completes before m_j starts, m_i starts executing but m_j does not start executing, or neither method invocation is executed. The predicate is expressed as: $(complete(m_i) < start(m_j)) \vee (complete(m_i) = start(m_j) = \infty)$.

- *Exclusive* constraints specify that a set of method invocations, $M \subseteq MI$, of the transaction must be executed without interruption from any conflicting methods of other transactions. Let m be any method invocation in M, and n be any method invocation not in M that conflicts with a method invocation in M. The predicate of an exclusive constraint is expressed as: $(start(n) = \infty) \vee (complete(n) < start(m)) \vee (complete(m) < start(n))$

- *Atomic* constraints on a set of method invocations, $M \subseteq MI$, specify that every method invocation in M is executed or no method invocation in M is executed. This requirement is expressed with the predicate: $(\forall_{m \in M} complete(m) \neq \infty) \vee (\forall_{m \in M}, start(m) = \infty)$.

- *Absolute timing* constraints specify that execution is to be performed within a certain time frame. We use a *temporal scope* to define absolute timing constraints: $\langle E, sa, sb, d\rangle$ where E is a set of executions, sa is an absolute earliest start time, sb is an absolute latest start time and d is an absolute latest complete time (deadline). The predicate for an absolute timing constraint is: $\forall_{e \in E}((sa \leq start(e) \leq sb) \wedge (complete(e) \leq d))$. In the case of absolute timing constraints on transactions or inter-object enforcement rules, E is a set of method invocations. For object methods and enforcement rules, E is a set of programming language statements including reads and writes of attributes.

- *Simultaneous* constraints specify that a set of method invocations, $M \subseteq MI$, must start executing within the same time interval. This constraint predicate is expressed as: $\forall_{m_i, m_j \in M}(start(m_i) = start(m_j) \pm \delta)$, there δ is a small time value.

Any collection of method calls generated by the investor's program or by the automated update facilities tied to the quotation company's reporting system can be realized in a transaction. In particular, Section 4 describes how the index arbitrage with guaranteed quotations can be supported by a transaction using a timed atomic commitment protocol in this real-time object-oriented database.

Inheritance. In the traditional object-oriented paradigm, *inheritance* is the mechanism used to define objects as extensions of previously defined objects. A sub-type in the inheritance hierarchy inherits all of the properties of all of its super-types and may add its own properties as well. Traditionally, inheritance applies only to attributes and methods. In our model, we extend this notion to include constraints, enforcement rules, and the added features of a relationship. We also support another feature of inheritance known as *substitutability* [Zdonik & Maier, 1990], which allows us to treat an instance of an object as an instance of any of its super-types. In other words, an instance of a sub-type may be used in contexts where one of its super-types is expected. Referring to Figure 1, a dotted line represents that one object type is derived from another object type using inheritance. for instance, a $STOCK$ object type is derived (indicated by IS_A) from a $COMMODITY$ object type. The child object type ($STOCK$) inherits all of the attributes, methods, and constraints from the parent object type ($COMMODITY$); it also adds its own special properties.

4 Timed Atomic Commitment For Index Arbitrage

We now show how a technique called timed atomic commitment can be realized using a transaction of our real-time object-oriented database model to perform index arbitrage. We first summarize the definition of timed atomic commitment and describe a set of real-time programming language constructs that we use to express psuedo-code for the transaction. These constructs support capabilities of transactions in our model by expressing timing, exclusive, atomic, and precedence constraints on object method invocations. We use the psuedo-code to express a transaction that performs a timed atomic commitment for index arbitrage and discuss its correctness. The full transaction psuedo-code appears in Appendix B.

4.1 Definition of Timed Atomic Commitment

Atomic commitment requires that a set of participants (transactions) either all decide to commit or all decide to abort. It is a problem that has been studied extensively, has a clean definition, and has a range of provably correct protocols for its implementation [Bernstein et al., 1986]. However, it is impossible to realize a correct atomic commitment under a deadline. If a processor fails before a decision has been reached and remains down until after the deadline, it may be impossible for any processor to reach a decision. Furthermore, if a processor fails before completing the decided upon action, it may be down until after the deadline and obviously cannot complete the action. Even if processors don't fail, message loss alone causes timed atomic commitment to be impossible. This fact follows easily from the "Two General's Paradox" [Gray, 1979; pp394–481], which states that there can be no fixed length protocol for non-trivial agreement between two or more processes if messages can be lost. Since reasonable operating environments include message loss and processor failure, traditional atomic commitment cannot be extended to observe a deadline. Therefore, the definition of timed atomic commitment (TAC) modified the definition of traditional atomic

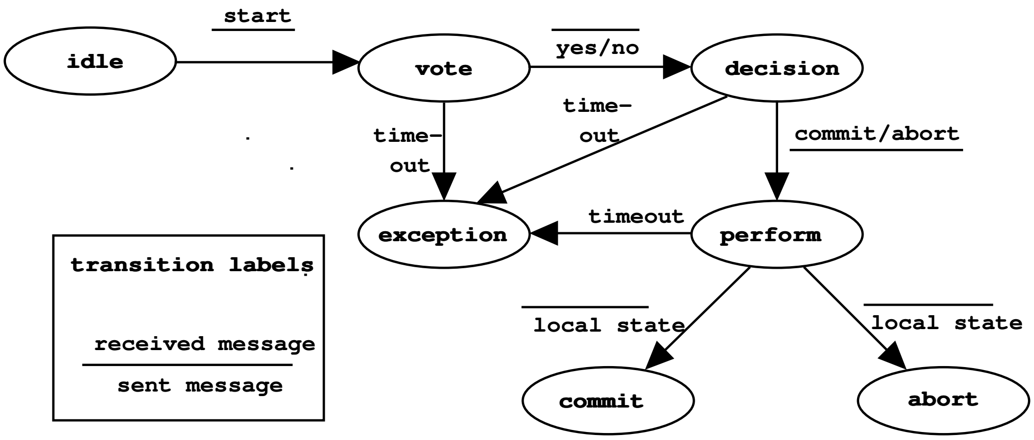

Figure 4: Model of a Participant in TAC

commitment to include a deadline and an exception outcome in [Davidson, et al., 1991]; we summarize that definition of TAC here.

Figure 4 shows a model of the process followed by participants in a timed atomic commitment. When the TAC commences, a global clock is initiated to measure the deadline for completion, D. Each participant goes through three phases: a *vote* phase, at the end of which it produces a vote of YES or NO; a *decision* phase, at the end of which it produces the decision, COMMIT or ABORT; and a *performance* phase, during which it performs the decided upon action and records the outcome in its local state. The vote indicates the participant's perception of its ability to commit: a YES vote is a promise to commit if the decision is made to commit; a NO vote means it cannot promise to commit. The local state of a participant is initially EXCEPTION, and cannot be altered after the TAC ends at deadline D.

Informally, in a "perfect" operating environment, the goal of TAC is to guarantee that, at deadline D, either all participants have local states of COMMIT, or all participants have local states of ABORT. Furthermore, a COMMIT outcome is preferable to an ABORT outcome. To reach a COMMIT outcome, every participant must vote YES and decide to COMMIT. Additionally, the commit actions must be successfully performed by deadline D. To reach an ABORT outcome, some participant must vote NO, and thus all participants decide to ABORT; aborting (which may include performing restoring actions) must also be successfully performed by D.

Unfortunately, actual operating environments are not perfect and include faults. For example, local clocks may be skewed (*e.g.* a variation between the investor's clock and the broker's clock), messages may be delayed or even lost, processes may not be able to execute on a processor when they need to, and execution may take longer than expected. Any of these factors may cause some participant to have a local state of EXCEPTION after the TAC, *i.e.*, be unable to vote, decide, or perform the decided upon action by deadline D. However, most operating environments offer "guarantees": for example, local clocks are synchronized to within a constant ϵ, and delivery time of messages has an upper bound δ. If the operating environment does not maintain a stated guarantee, we say that a *fault* has occurred. When faults occur, we allow the TAC to indicate an EXCEPTION outcome.

We now specify what it means to perform *correct* timed atomic commitment.

1. All participants that reach a decision reach the same one.
2. The decision is to commit only if all participants vote YES.
3. At D, a participant's local state either reflects the participant's completed action or is EXCEPTION.
4. If there are no faults, then
   (a) all participants reach a decision;
   (b) if all participants vote YES, then the decision is to commit;
   (c) all participants complete the decided upon action by D; and
   (d) at D, a participant's local state reflects the participant's completed action.

Criteria 1 and 2 define logical consistency of TAC, while Criterion 3 requires the local state to be determined at D. Criterion 4 defines minimal "success" requirements: Criterion 4b requires the decision to be COMMIT if there are no faults and all participants vote YES; this invalidates trivial protocols that arbitrarily force the decision to be ABORT. Criteria 2 and 4a together imply that a decision must be made to ABORT rather than remaining EXCEPTION if there are no faults and some participant votes NO; this eliminates trivial protocols that allow a process to remain undecided. Criteria 4c and 4d require that, in the absence of faults, the decided upon action must be successfully completed and recorded in the local state by D.

Note that in addition to the "all-commit" or "all-abort" outcomes of traditional atomic commitment, there are three other combinations of local states in a TAC: 1) all exceptional; 2) some committed, some exceptional; and 3) some aborted, some exceptional. This increased number of outcomes is due to the distinction between the EXCEPTION state and the ABORT state. In the example, an ABORT state implies that the index arbitrage was not carried out and the investor has not lost money. In an EXCEPTION state, the participant may have *partially* performed commit or abort actions; *e.g.*, one part of the buy/sell may have been completed by the deadline while the other was not. The EXCEPTION state indicates that the system may be inconsistent, and that recovery should be performed.

In the index arbitrage example, it is not enough that the participants establish their own local states by deadline D; the investor's transaction must know all of the local states by D so that it can determine if the arbitrage was successful. We therefore extend the definition of timed atomic commitment with a *calling transaction* that initiates the TAC by sending out the start messages, measures D on its clock, and establishes the outcome of the TAC by D. The outcome of the TAC is represented by a *global state vector*. The global state vector entry for each participant is initially EXCEPTION and is changed when the caller determines each participant's local state. To ensure that the caller correctly establishes the outcome of the TAC by deadline D, we replace TAC Criteria 3 and 4d in the timed correctness criteria with:

> **3′** At D, a participant's local state either reflects the participant's completed action or is EXCEPTION. Furthermore, the participant's global state vector entry is either its local state or is EXCEPTION.
>
> **4d′** at D, a participant's local state reflects the participant's completed action. Furthermore, the participant's global state vector entry is the same as its local state.

The TAC protocol that we employ in our index arbitrage example is based on this extended definition.

4.2 Transaction Programming Language

We express the timed atomic commitment in the index arbitrage example with psuedo-code that utilizes the *RTC* real-time programming language constructs [Wolfe, et al., 1993]. These constructs consist of a small set of orthogonal primitives that express timing and consistency constraints naturally and explicitly by specifying a set of method invocation statements. *Block* statements are used to express precedence orderings, concurrency constraints, and timing constraints.

Method Invocation Statements. An *RTC* method invocation statement may either be *synchronous* (denoted by **action**) or *asynchronous* (denoted by **action&**), and must indicate the object and method being invoked. With a synchronous method invocation statement, the calling transaction waits for the invoked method to complete. A calling transaction does not wait for an asynchronously invoked method to complete. Completion of an asynchronously invoked method can be indicated by using an *event variable* (discussed below); the run-time system sets the event variable's value to be the completion time of the invoked method. For example, in the investor's transaction of Appendix B, all method invocations are asynchronous. The statement "**action&** C_SP500.GetPrice(Price1)" invokes the method to get a price from the representation of the SP500 future on the Chicago Mercantile Exchange; the price is returned in parameter *Price1*.

Timing Constraints. Timing constraints are specified using the *RTC timing block* construct, which constrains explicitly the earliest start time, latest start time, maximum execution time, and completion time of statements in the block. Timing expressions can be formed using arithmetic operations, maximum functions, and minimum functions involving time and event types. An event has absolute time values that are established in three ways: 1) a **signal** statement by a transaction or method, which causes the current absolute time to be assigned to the event variable; 2) a **clear** statement by a transaction or method, which sets the event variable to an infinite absolute time; or 3) the run-time system signaling an event associated with the completion of an asynchronous method invocation.

The timing block construct **after** constrains the earliest start time, while **before** constrains the latest start time; **by** is used to indicate the deadline. All of these constructs include an expression of absolute time. If some statement in the timing block has not started by the time specified in the **before** constraint, the statement is not started and an E_START exception handler becomes ready. If the timing block has not completed by the specified deadline, then the current execution is aborted and an E_DEADLINE exception handler becomes ready. Most parts of a timing block are optional.

Method Invocation Precedence Ordering. Flexible expression of precedence constraints (concurrency) within transactions and between transactions is supported using a combination of events, synchronous and asynchronous method invocations, along with earliest start time constraints. For example, concurrency within a transaction is expressed using asynchronous method invocations to "fork" off concurrent method invocations. Then, later in the transaction, the earliest start time constraint (**after**) can be used to "join" execution by requiring that the transaction wait for a subset of the spawned method invocations to complete. For example, the investor's transaction in Appendix B. expresses the following precedence constraints:

C_SP500.GetPrice → *C_B1.GetGuaranteedQuote* → *C_B1.Buy (or sell)*
NY_SP500.GetPrice → *NY_B2.GetGuaranteedQuote* → *NY_B2.Sell (or buy)*

which state that the two *GetPrice* method invocations must be performed before the two *GetGuaranteedQuote* method invocations, which themselves must be performed before the *Buy* or *Sell* method invocations. The two *GetPrice* method invocations have no precedence constraints relative to each other and therefore may execute concurrently as may the two *GetGuaranteedQuote* method invocations and also the *Buy* and *Sell* method invocations. Precedence orderings between methods in different transactions can be expressed using events that are global to the transactions.

Guaranteed Constraints. We define a method invocation a to be *ready* at time t if and only if executing it at t meets all precedence and absolute timing constraints. Using this notion, a *guaranteed block* (denoted by **guaranteed – end guaranteed**) specifies that each statement and method invocation in the block must execute at the earliest time that it is ready, and continue to execute without interruption until it is completed. This means that no delays due to contention for resources may occur in the transaction or the methods that the transaction invokes while the transaction is in the guaranteed block.

Exclusive Execution Constraints. To ensure that no other transaction in the database manipulates an object involved in the index arbitrage while it is being performed, the investor's transaction uses an *exclusive block* (denoted by **exclusive – end exclusive**). The meaning of the block is that no method that is incompatible with any method invocation in the exclusive block may be executed by another transaction after the exclusive block starts and before it completes.

Multiple Constraints. Multiple constraints are expressed by nesting blocks. The semantics of nested blocks are a composition of the semantics of the individual blocks, thus allowing the expression of multiple constraints on parts of transactions. If exceptions are raised simultaneously in several nested blocks, only the "outermost" violated block handles the exception.

4.3 Index Arbitrage Timed Atomic Commitment

When the system in our application decides to pursue an index arbitrage, it issues a transaction to perform a TAC. The transaction follows a protocol based on a centralized timed two-phase commit protocol for TAC presented in [Davidson, et al., 1991]. That protocol uses intermediate timing constraints and *resource reservations* to meet the TAC correctness criteria. Since we make no assumptions about the underlying system being able to provide such capabilities, we will express the index arbitrage transaction's requirements using

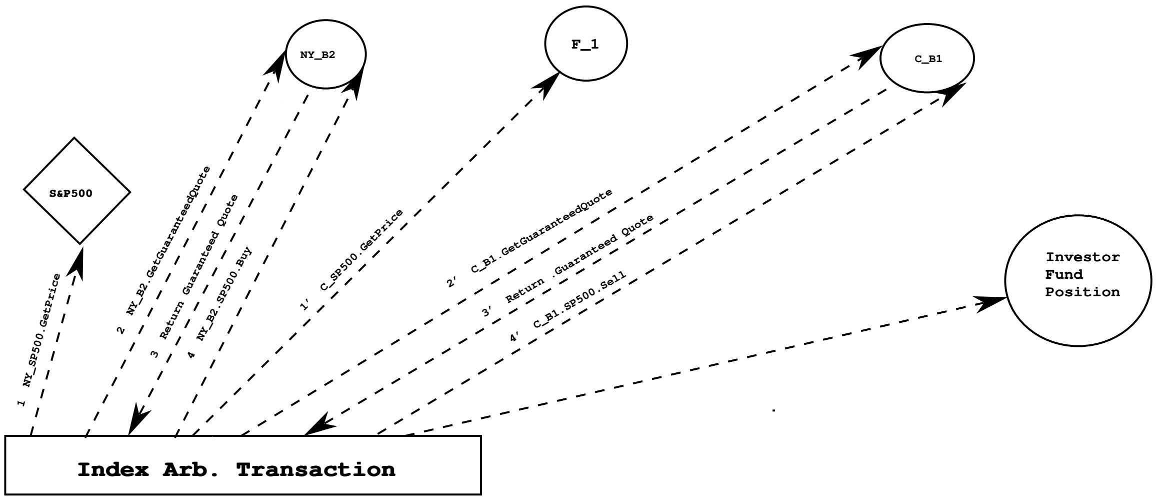

Figure 5: Messages Between an Index Arbitrage Transaction and Object Instances

RTC-based psuedo-code and show that it expresses a correct index arbitrage in our real-time object-oriented database model.

In our solution, the index arbitrage transaction is the *coordinator* that initiates the TAC; the *participants* are the database *BROKER* objects that the transaction uses. The system regularly (*e.g.* every minute) starts the index arbitrage transaction which fetches price information for the futures contract (*C_SP500.GetPrice*) and the underlying stocks (*NY_SP500.GetPrice*) from the respective exchanges. When the prices are collected, the transaction computes the price difference. If the difference is more than the carrying cost plus the transaction cost, the transaction as coordinator requests votes from the participants by calling a vote action for each participant (*C_B1.GetGuaranteedQuote* and *NY_B2.GetGuaranteedQuote*). Each "vote" represents a guaranteed quotation from a broker and includes the price and the volume of the portfolio, the deadline to vote, and the expiration time of the quotation. The transaction waits for the vote actions to return and then decides either to commit by invoking buy and sell actions, or to abort.

Figure 5 shows the investor's transaction as it would appear with the instance-level model of the database from Figure 3 of Section 3. The lines from the transaction to the objects represent method calls, the lines from the objects to the transaction represent return parameters. These messages are numbered to indicate the order in which they are sent.

This timed atomic commitment solution imposes intermediate timing constraints on its phases to ensure that the correctness criteria of Section 4.1 are met. Of particular importance are a latest start time for the transaction to ensure the TAC has a chance of completing correctly, a deadline by which a participant must vote, and a deadline by which the transaction

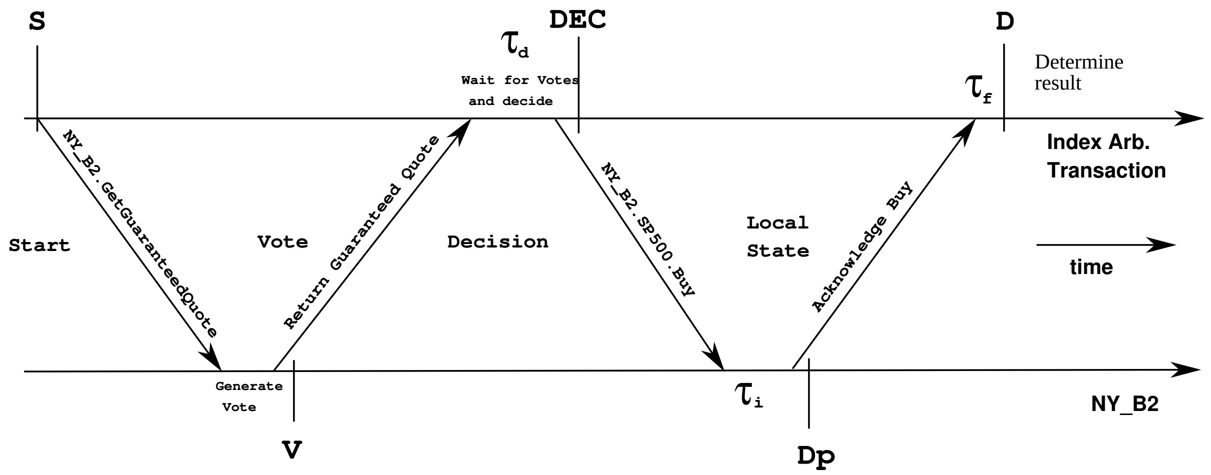

Figure 6: Timeline For TAC Messages

must decide. To derive these timing constraints, we assume that the worst case execution time of all parts of the program are known. Figure 6 shows a timeline with execution times and intermediate deadlines for the investor's interaction with a broker. Appendix B outlines the RTC psuedo-code used to perform a TAC. Let D be the absolute deadline for performing the TAC. We assume that the worst case execution times are as follows:

- τ_i is the maximum execution time needed for a participant P_i (a broker), to receive the method invocation from the transaction, carry out the commit (buy or sell) or abort action, and indicate its completion to the transaction, measured on P_i's clock.
- τ_{amax} is the largest of all of the participants' τ_i's.
- τ_{vmax} is the maximum time it takes any participant to execute a guaranteed quotation (vote) method invocation.
- τ_c is the maximum execution time needed for the transaction to call the two asynchronous method invocations, and initialize the decision variable.
- τ_d is the maximum execution time needed for the transaction to receive notification of the two events being signaled, compute the maximum, resume the statements following the after clause, determine the votes, and test for YES vote.
- τ_s is the maximum execution time needed for the transaction to test or set the value of a variable and then invoke the two asynchronous methods.
- τ_f is the maximum execution time needed for the transaction to receive notification of the two events being signaled, compute the maximum, resume the statements following

the after clause, and set the values for the N entries of the global state vector.

The following intermediate timing constraints are used:

- $D_p = D - \delta - \tau_f - \epsilon$: deadline for sending a return message by a participant's commit/abort method invocation. In the absence of faults, each participant must complete the decided upon action and send the completion message (at most δ time units) so that the transaction has time to process it (at most τ_f time units) before D on the transaction's clock (skewed by at most ϵ).

- $DEC = D_p - \tau_{amax} - \tau_s - \delta - \epsilon$: the deadline for making a decision by the transaction. For a participant with τ_{amax} execution time to guarantee completion of the decided upon action by D_p in the absence of faults, it must start executing the method invocation by $D_p - \tau_{amax}$ on its clock. The transaction must then interpret this time on its own clock using the worst case assumption on clock skew, allowing maximum time to send the method invocation message (τ_s), and the maximum message delay for the decision to arrive at the participant (δ).

- $S = DEC - \tau_c$: the latest start time for the transaction to commence the TAC. The transaction must not start the TAC if it cannot initialize the global vector and send out vote method invocation messages (τ_c) by the time it must decide (DEC).

- $V = DEC - \delta - \tau_d - \epsilon$: deadline for a participant to vote. The participant must vote in time for the vote message to arrive at the transaction and be processed before DEC expires on the transaction's clock.

Although not explicitly shown in the code of Appendix B, before starting a TAC, the transaction will use a **before** clause of a timing block before the code in Appendix B to ensure that it starts the TAC in sufficient time (time S) to invoke vote methods so that all participants reach a decision. After invoking vote methods, the transaction uses an **after** clause in a timing block to wait to receive return arguments from the participants. When it receives all return arguments, it determines the vote from the arguments. If all votes were YES, the transaction decides COMMIT and enters a no-except block to ensure that all commit actions are invoked. However, if intermediate deadline DEC expires before the transaction decides, or if any vote was NO, the transaction decides ABORT and invokes abort actions in the participants. After invoking the decidedupon methods, the transaction uses an **after** clause to wait for events to be signaled indicating that these method invocations have completed. Once all events have been signaled, the transaction updates the corresponding global state vector entries. If deadline D expires before all of these events have been signaled, the global state vector retains its initial EXCEPTION values.

Correctness of TAC-based Index Arbitrage. To show that this index arbitrage solution meets the TAC correctness criteria, we present informal arguments for each criteria.

From the construction of the code in Appendix B , it is obvious that the transaction either invokes both buy/sell methods or neither, so it invokes the same action (COMMIT/ABORT) in all participants (TAC Criterion 1). TAC Criterion 2's requirement that the decision be COMMIT only if all participants vote YES is obvious because the index arbitrage transaction invokes the buy/sell (COMMIT) action only if all "votes" were YES.

In order to reason about meeting intermediate deadlines and the deadline D of Criteria 3 and 4, we must make two observations. First, if there are no faults, any message sent at time t on the sender's clock is guaranteed to arrive by $t + \delta + \epsilon$ on the receiver's clock. Second, if there are no faults, then the method invocation (decision) message arrives at each broker by $D_p - \tau_{amax}$, measured on the broker's clock. This second observation relies on the fact that the transaction has sufficient time to get to where it can make a decision (*i.e.* D does not expire before it can reach a decision). This fact follows from the RTC **before** timing block clause enclosing the TAC and ensuring that the TAC starts by $S = DEC - \tau_c$. By the definition of τ_c, all vote method invocations must be sent by intermediate deadline DEC, so a transaction must get to the point where it can make a decision by DEC. Consequently, it is enough to show that in the absence of faults the decision method is invoked by $DEC + \tau_s$, because the first observation ensures that the invocation message arrives at each broker by $DEC + \tau_s + \delta + \epsilon = D_p - \tau_{amax}$ on that broker's clock. The transaction's RTC timing block with a deadline of DEC ensures that invocations of decided upon methods start by DEC, and, by the definition of τ_s, complete by $DEC + \tau_s$.

The broker is guaranteed to reach a decision (Criterion 4a) because 1) the decision message arrives at a broker by $D_p - \tau_{amax}$, 2) the broker is guaranteed to execute when it is ready, and 3) τ_{amax} includes execution time to receive the decision. To show TAC Criteria 4b and 4c, assume that there are no faults and each broker votes YES. This assumption implies that each broker must have sent its vote message by V measured on its clock. So, every vote message must arrive at the transaction by $V + \delta + \epsilon = DEC - \tau_d$, measured on the transaction's clock. The definition of τ_d guarantees that the transaction can receive all vote return arguments and decide to commit by DEC, which means it will invoke the buy/sell methods, which is committing (Criterion 4a). Also, since 1) the decision message arrives at each broker by $D_p - \tau_{amax}$, 2) each broker is guaranteed to execute when it is ready, and 3) $\tau_i \leq \tau_{amax}$, the definition of τ_i allows the broker to complete the decided upon action and sends a return message by D_p.

TAC Criterion $4d'$ requires that the transaction know by D that the brokers have completed their actions in time. As noted above, each participant sends a return message by D_p

so the return messages must arrive at the transaction by $D_p + \delta + \epsilon = D - \tau_f$. The transaction has time (τ_f) to receive all return messages by D. These return messages establish the transaction's global state vector. Since they arrive by D, TAC Criterion $3'$ is met.

4.4 System Development and Implementation

We have implemented a prototype database that supports objects and relationships such as those described in this paper. The SORAC (**S**emantic **O**bjects **R**elationships **A**nd **C**onstraints) database research project at the University of Rhode Island [Doherty, Peckham, & Wolfe, 1993] captures the objects and relationships of our model by combining features of the object-oriented [Zdonik & Maier, 1990] and semantic data models [Peckham & Maryanski, 1988]. The SORAC data model maintains object-oriented principles: encapsulation of structure and behavior together in data objects; information hiding; and unique, but perhaps complex object identity. It also adds the capability for user-defined relationships among object types [Doherty, et al., 1993]. To capture the model, we developed an object-oriented data definition language for defining objects and relationships. A compiler translates these object and relationship definitions into code that executes on the Ontos commercial object-oriented database system.

Our current real-time object-oriented database (RT-SORAC) testbed is a main-memory database implementation of the SORAC database, which follows our real-time object-oriented database model of Section 3. The RT-SORAC system is implemented on the Lynx real-time operating system [Gallmestier & Lanier, 1991], which is compliant with the proposed IEEE and ISO POSIX 1003.4a real-time operating system standard [IEEE P1003.4a Working Group, 1992]. We have extended Lynx's POSIX 1003.4 (limited) real-time scheduling capabilities to provide dynamic earliest-deadline-first scheduling of transactions and method invocations [Senerchia, 1993].

We are currently developing a transaction system for our prototype real-time object-oriented database. A major focus of this effort is an extension of the standard SQL query language with capabilities to support objects [Gallagher, 1992] and real-time properties [Wolfe & Cingiser, 1992] as the interface to generate transactions. Many of the features of the *RTC* language constructs that we used to describe the index arbitrage example are supported in this real-time SQL extension, such as explicit timing constraints. Once the SQL transaction system is complete, its ability to express and support enforcement of the TAC solution to index arbitrage in a real-time object-oriented database for program stock trading will be an important indication of the effectiveness of the language and the new database technology presented in this paper.

5 Summary and Future Work

This paper has described a new database technology: real-time object-oriented databases, and how it can be used to support an system for program stock trading. We have shown how the object-oriented data model is effective for modeling complex data types, such as individual stocks, and the relationships among data types, such as a portfolio collection. We have also shown how our model can effectively specify real-time constraints of perishable data such as timing constraints on "current" stock prices and volumes. Moreover, we utilized the real-time features of our model to express an illustrative index arbitrage transaction that follows a real-time database protocol called timed atomic commitment. This protocol proved to be an elegant way to handle the time-constrained, all-or-nothing characteristic of an index arbitrage. The real-time object-oriented database support for time-constrained data and the ability to express intermediate deadlines on transactions facilitated this creation of a general algorithm for time-constrained index arbitrage.

Our future work includes integrating the prototype RT-SORAC real-time object-oriented database system to actual quotation services so that price and volume data is automatically updated. Since to our knowledge there are no automated guaranteed quotation services currently available, we must simulate that input into the index arbitrage. With this system, we will be able to accurately perform time-constrained index arbitrage applications.

We are also extending the RT-SORAC system with support for user-defined real-time semantic concurrency control where each object in the system controls concurrency among its own methods [Wolfe & Cingiser, 1993]. In addition, we are developing an SQL-based query interface that supports both the object and real-time properties of RT-SORAC.

References

Abbott, R., & Garcia-Molina, H. (1988). Scheduling real-time transactions: A performance evaluation. Proceedings of the Very Large Database Conference, 14, 203-210.

Amling, F. (1989). Investments: An introduction to analysis & management. Englewood Cliffs, NJ: Prentice Hall.

Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1986). Concurrency Control and Recovery in Database Systems. New York: Addison Wesley.

Davidson, S., Lee, I., & Wolfe, V. (1991). Timed atomic commitment. IEEE Transactions on Computers, 40(5), 573–583.

Doherty, M., Peckham, J., & Wolfe, V. (1993). Implementing relationships and constraints in an object-oriented database. (Technical Report URI-TR-93-218). Rhode Island: The University of Rhode Island Dept. of Computer Science.

Gallagher, L. (1992). Object SQL: Language extensions for object data management. Proceedings of International Society For Mini and Microcomputers, 4, 156-162.

Gallmestier, B., & Lanier, C. (1991). Early experience with POSIX 1003.4 and POSIX 1003.4a. Proceedings of the IEEE Real-Time Systems Symposium, 12, 190-198.

Garcia-Molina, H. (1983). Using Semantic Knowledge For Transaction Processing in a Distributed Database System. ACM Transactions on Database Systems, 8(2), 186-213.

Gray, J. (1979). Notes On Database Operating Systems, Operating Systems. New York: Springer-Verlag.

Gupta, R. (1991). Object-Oriented Databases With Applications To CASE, Networks and VLSI CAD. Englewood Cliffs: Prentice Hall.

Harista,J.R., Livny, M. & Carey, M. J. (1990). Earliest deadline scheduling for real-time database systems. Proceedings of the IEEE Real-Time Systems Symposium, 11, 94–103.

Huang,J., Stankovic, J.,Towsley, D., Ramamritham,K. (1991). Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. Proceedings of the Very Large Database Conference, 17, 110-118.

Huang,J., Stankovic, J.,Towsley, D., Ramamritham,K. (1989). Experimental evaluation of real-time transaction processing. *Proceedings of the IEEE Real-Time Systems Symposium, 10,* 144-154.

IEEE P1003.4a Working Group (1992).. Threads extension for portable operating systems (draft). New Jersey: IEEE Computer Society.

Jang,G. & Lai, F (1993). Intelligent stock market prediction system using dual adaptive-structure neural networks. In R.S. Freedman (Ed.) Proceedings of Second International Conference on Artificial Intelligence Applications on Wall Street, (pp. 88-97). Gaithersburg, MD: Software Engineering Press.

Lirov, Y.. Embedded artificial intelligent for trading floor support. In R.S. Freedman (Ed.) Proceedings of Second International Conference on Artificial Intelligence Applications on Wall Street. (pp. 123-130). Gaithersburg, MD: Software Engineering Press.

Market Volatility and Investor Confidence Panel (1990). Market volatility and investor confidence - report to the board of directors of the new york stock exchange. New York: New York Stock Exchange.

Miller, J., Miller, M., & Brennan, P. J. (1991). Program Trading – The New Age of Investing.

New York: J.K. Lasser Institute.

Peckham, J. & Maryanski, F. (1988). Semantic data models. ACM Computing Surveys, 20(3), 153–189.

Ramamritham, K. (1993). Real-time databases. International Journal of Distributed and Parallel Databases, 1(2), 123-139.

Senerchia, J. (1993). A dynamic real-time scheduler for POSIX 1003.4a compliant operating systems. Unpublished Master's Thesis, Dept. of Computer Science, The University of Rhode Island. Kingston, RI.

Sha, L., Rajkumar, R., Son, S., & Chang,C. (1991). A Real-Time Locking Protocol. IEEE Transactions on Computers, 40(7), 793-800.

Spectrum Staff. (1987, December). How computers helped stampeded the stock market. IEEE Spectrum.

Son,S. Yannopoulos,S., Kim,Y., Iannacone, C. (1992). Integration of a database system with real-time kernel for time-critical applications. Proceedings of the International Conference on System Integration (pp. 203-212).

Viau, B. (1993). Intelligent broker station of the near future, a rule-based expert system. In R.S. Freedman (Ed.) Proceedings of Second International Conference on Artificial Intelligence Applications on Wall Street. (pp. 123-130). Gaithersburg, MD: Software Engineering Press.

Wolfe, V. & Cingiser, L. (1992). Issues in object-oriented real-time databases. Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software (pp. 180–183). Washington, DC: IEEE Computer Society Press.

Wolfe, V. & Cingiser, L. (1993). Object-based semantic real-time concurrency control. To appear in Proceedings of the IEEE Real-Time Systems Symposium.

Wolfe,V., Davidson,S. & Lee, I. (1993). *RTC*: Language support for real-time concurrency. Real-Time Systems, 5(1), 63-87.

Zdonik, S., & Maier, D. (1990). Readings in Object Oriented Database Systems. San Mateo, CA: Morgan Kauffman.

Appendix A: Detailed Model of Example

This appendix describes types for objects in the index arbitrage example. A full description is impractical due to excessive details, but important features of the example are presented. Key: N = name of type; A = attributes; M = methods; C = constraints; P = set of participant types in a relationship type IC = inter-object constraints in a relationship type (See model of Section 3).

Exchange Object Type. An Exchange Object type represents a commodity exchange, which is a place where investors (individual investors, institutional investors, speculators, arbitrageur etc.) can trade commodities through brokers. At the instance level, a Commodity Exchange can be a market place trading a variety of commodities, like the Chicago Mercantile Exchange (CME). The CME is a futures exchange that trades on agricultural commodities, foreign currencies, interest rates and stock indices, its trading value exceeded \$50 trillion in 1992. A Commodity Exchange can also be an exchange specialized in trading a particular product line - stocks, like New York Stock Exchange (NYSE). The NYSE is an exchange trading over 1,700 leading companies' shares form around the world, the trading value in 1991 was \$1.52 trillion.

The Exchange Object type has attributes Total_Price, Total_Volume, Name_of_Brokers, etc., which should be updated periodically. To specify the temporal consistency of these data items, timing constraints are placed on the object indicating when the attributes must be updated. For instance, the timing constraint on the Total_Volume attribute has a predicate that states Total_Volume.Time < Now - 15 min and the corresponding enforcement rule marks the Total_Volume as invalid if the constraint is violated. The Object also has logical constraints; for instance, we enforce that Total_Volume cannot be negative. This assertion is specified with the predicate Total_Volume.Value $\geq$ 0 and an enforcement rule that denies any update that makes this predicate false. The Total_Price attribute is derived from the Total_Volume, so we express a logical constraint with predicate Change(Volume) => Change (Price), and the enforcement rule that computes a new value of Total_Price when the Total_Volume attribute is changed. The method UpdateTotalPrice(Commodity_Group) writes total price and is called by a transaction generated by an index arbitrage monitor subsystem.

N : Exchange
A : Total_Price
    Total_Volume
M : UpdateTotalPrice(P)
    UpdateTotalVolume(V)
    GetTotalPrice(P)
    GetTotalVolume(V)
C : TotalPrice > 0
    TotalVolume > 0

Portfolio Relationship Type. The Portfolio Relationship type relates commodity types and keeps track of attributes such as the total price and volume of the portfolio. This relationship type also expressed inter-object constraints, such as that the changing of a stock price necessitates changing of the portfolio's price.

N : Portfolio
A : Price_of_Portfolio
    Volume_of_Portfolio
M : GetPortfolioVolume(V)
    UpdateMembership(Member)
    UpdatePortfolioPrice(P)
    GetPortfolioPrice(P)
    UpdatePortfolioVolume(V)
C : PortfolioPrice > 0
    PortfolioVolume > 0
IC : change(Stock, Price) $\Rightarrow$
        UpdatePortfolioPrice(P)
    change(Stock, Volume) $\Rightarrow$
        UpdatePortfolioVolume(V)
P : {Stocks(n)}

Listing Relationship Type. This relationship type relates commodities to their exchange. It also contains information about the listing itself: the Listing_ID.

N : Listing_At_Exchange
A : Listing_ID
M : GetListingID(ID)
    UpdateListingID(ID)
IC : change(Stock Price)
    $\Rightarrow$ Exchange.UpdateTotalPrice(P)
    change(Stock Volume) $\Rightarrow$
    Exchange.UpdateTotalVolume(V)
P : {Exchange(1), Stock(n)}

Commodity Object Type. A Commodity Object type represents a commodity trading at a Commodity Exchange. It has attributes of Price, Label. These attributes should be updated periodically. Logical constraints are also imposed on the Object (Price.Value > 0). The method UpdatePrice(Commodity) writes commodity price and is called by a transaction generated by the index arbitrage monitor subsystem.

N : Commodity
A : Price
    Label
M : UpdatePrice(P)
    GetPrice(P)
    UpdateLabel(L)
C : Price > 0

Futures Object Type. A Futures Object type represents a futures contract trading at a futures market like CME. It is formed through inheritance (IS-A) from the Commodity Object type. It inherits attributes (*e.g.* Price) methods (*e.g.* GetPrice), and constraints (*e.g.* Price > 0) and also adds new attributes, methods and constraints.

N : Futures
A : Price
    Label
    UnitsPerContract
M : UpdatePrice(P)
    GetPrice(P)
    UpdateLabel(L)
    UpdateUnitsPerContract(U)
    GetUnitsPerContract(U)
C : Price > 0
    UnitsPerContract > 0

Stock Object Type. A Stock Object type represents each individual stock trading in a stock market, like NYSE. It is formed through inheritance (IS-A) from the Commodity Object type. It has inherited the basic attributes, methods, and constraints from the Commodity Object type. It has also added new attributes and new methods to the object, including: LotSize, TradingVolume, UpdateLotSize(Stock), GetLotSize(Stock), and UpdateTradingVolume(Stock).

N	:	Stock
A	:	Price
		Label
		LotSize
		TradingVolume
M	:	UpdatePrice(P)
		GetPrice(P)
		GetVolume(V)
		UpdateLabel(L)
		UpdateLotSize(S)
		UpdateTradingVolume(V)
C	:	Price.Value $>$ 0
		LotSize.Value $>$ 0
		Price.Time $<$ Now - 5 min

Broker Object Type. A Broker Object type represents an individual broker trading at a commodity exchange.

N	:	Broker
A	:	Trading_Portfolio
		Trading_Commodity
		Trading_Portfolio_Commission
		Trading_Commodity_Commission
M	:	GetGuaranteedQuote(Commodity,Price,Vol,d_b,d_p,Gtime)
		UpdatePrice(A)
		GetPrice(P)
		Sell(Commodity,Vol,Price)
		Buy(Commodity,Vol,Price)
		ConfirmExecution(Commodity)
C	:	Commission $\geq$ 0

Licensed At Relationship Type. This relationship type relates a broker to the exchange at which he/she is licensed. It also contains information about the licensing: the License_Num.

N : Licensed_At_Exchange
A : License_Num
M : UpdateLicNum(Num)
    GetLicNum(Num)
P : {Exchange(1), Broker(n)}

Investor Object Type. An Investor Object type represents an investor (index arbitrageur). Its main functions are to keep track of the CapitalBalance, Account_at_Broker, Inventory as well as handle LossCover. The temporal and logical consistency implementation, method invocation and constraints imposition are similar to those of the other objects types.

N : Investor
A : Name_of_Investor
    Capital_Balance
    Account_At_Broker
M : UpdateCapitalBalance(Balance)
    GetCapitalBalance()
    UpdateAccount_At_Broker(Broker)
    UpdateInventory(Commodity)
    CoverLoss(Transaction_ID,Fund)

Appendix B: Index Arbitrage Transaction

.

.

.