Multiple Concurrency Control Policies in an Object-Oriented Programming System

Gail E. Kaiser
Wenwey Hseush
Steven S. Popovich
Shyhtsun F. Wu

Columbia University
Department of Computer Science
New York, NY 10027
212-939-7000/Fax: 212-666-0140
Kaiser@cs.columbia.edu

22 September 1990 (revised 28 January 1992)
CUCS-020-90

Abstract

Object-oriented systems have been generally recognized as a good choice for approaching a range of parallel and distributed programming problems. But different parallel and distributed applications have different consistency models, so *multiple concurrency control policies* are needed. When objects are shared among applications with different policies, multiple policies must operate simultaneously and compatibly. We have investigated two consistency models, atomic blocks and classical serializable transactions, and the interactions both between these models and with ''uncontrolled'' concurrency, in the context of the MELD distributed object-oriented programming language. We present the relevant language constructs and discuss certain programming problems and implementation issues.

Keywords: Applications, database systems, programming languages, transaction processing.

Multiple Concurrency Control Policies in an Object-Oriented Programming System

Gail E. Kaiser *Wenwey Hseush*
Steven S. Popovich *Shyhtsun F. Wu*

Columbia University
Department of Computer Science
New York, NY 10027

January 28, 1992

Abstract

Object-oriented systems have been generally recognized as a good choice for approaching a range of parallel and distributed programming problems. But different parallel and distributed applications have different consistency models, so *multiple concurrency control policies* are needed. When objects are shared among applications with different policies, multiple policies must operate simultaneously and compatibly. We have investigated two consistency models, critical sections on individual objects and classical serializable transactions, and the interactions both between these models and with "uncontrolled" concurrency, in the context of the MELD distributed object-oriented programming language. We present the relevant language constructs and discuss programming problems and implementation issues.

Keywords: Applications, database systems, distributed systems, parallel programming, programming languages, transaction processing.

1 Introduction

Object-oriented programming languages and data management systems have become very popular [Mey90, Pae91]. Concurrent object-oriented programming systems (COOPS) are particularly attractive for intrinsically parallel and distributed applications [Agh90], such as discrete event simulation and network management [(AN89]. COOPS thus require support for *concurrency control*, to enforce data consistency within and across these applications.

It is sometimes suggested that the classical *transaction* model be integrated directly into COOPS facilities. This model permits applications to be defined in terms of serializable transactions: either all the operations carried out during a transaction complete or none of them do, and the effects of the transactions committed over the lifetime of the system appear to have occurred in some serial order [BHG87].

The classical transaction model provides a good solution to many programming problems, but has been recognized as too strict for many applications [Ne89]. Different parallel and distributed applications have different consistency requirements, so *multiple* concurrency control policies are needed. When data is shared among applications with different policies, then the policies must operate simultaneously and compatibly.

We have investigated the *interoperability* of classical transactions and a less strict consistency model, which we call *atomic blocks*, that provides exclusive access to an individual object during a sequence of statements. We have also explored the implications of "uncontrolled" concurrency within the same COOPS. Our work has been in the context of the MELD object-oriented programming language [KPHW89].

After briefly discussing the motivation for this work, we introduce MELD's classes and the basic concurrency-related facilities, which might execute without any control. We explain both atomic blocks and transactions, together with the corresponding language constructs and run-time support. We then discuss the insights obtained through our integration of the three policies. We describe our implementation and compare to other concurrent object systems, and conclude by summarizing the contributions of this work.

2 Motivation

There are many applications where classical serializable transactions unnecessarily restrict concurrency. One area is computer-supported cooperative work [Gre88]. Collaboration involves long "transactions" carried out over hours or days, where participants *share* knowledge and data while their "transactions" are in progress, so the isolation enforced by serializability is inappropriate.

Real-time systems [Sta88] are another domain where the classical transaction model does not work. For example, in an on-line stock trading system, the stock exchange feed writes the new values of prices as they are received and cannot be blocked even though there may be portfolio management computations concurrently reading the prices. Further, portfolio managers want to see the latest prices as they become available, so reading an old snapshot of the data (obtained, for example, using a multiversion concurrency control protocol) is unacceptable, as is blocking price updates (for example, using a locking concurrency control protocol).

We have previously proposed some specific approaches for both the cooperative work and real-time domains [PKH88, Kai90, KH92], but our goal in sketching these domains here is to motivate our claim that the classical transaction model alone is insufficient, and to demonstrate the need for different concurrency control policies specific to application functionality. Thus, a COOPS should provide language constructs for several distinct concurrency control policies, supporting different consistency requirements. Since applications with different consistency requirements sometimes share data, the policies must operate compatibly within an objectbase.

Such interoperability is related to the problem of executing transactions in heterogeneous database systems [Ram91]. Global transactions consist of subtransactions, each of which executes on a different database, where each database may have a different scheme for implementing serializable transactions. One might use two-phase locking, another timestamps with multiple versions, and a third optimistic concurrency control, but the global transactions must be serialized. This might be enforced by requiring each database to support a particular predefined protocol, as in Superdatabase [Pu88]. Elmagarmid and Du describe a methodology to construct local concurrency control schemes given a particular global mechanism and application-specific tradeoffs [ED90].

This problem is subtly different than ours, since each of the federated databases supports only a single implementation of transactions, so it is never the case that the *same* object is accessed by different threads of control operating according to different implementation schemes. In contrast, the Papyrus architecture does allow multiple threads employing different concurrency control schemes to access the same objects, with the interactions among schemes specified in a conflict matrix [NW90]. But again the divergence is with respect to implementation schemes, not the consistency models themselves, since they <u>all</u> enforce serializability. We are concerned with interoperability among implementations of different consistency models.

There has been much theoretical work on consistency models other than serializability. Lynch [Lyn83] and Garcia-Molina [GM83] have proposed consistency models in terms of multiple abstraction levels, where interleavings between steps at one level of abstraction depend on the semantics of the lower-level operations carried out by the steps. Weihl [Wei89] describes three local atomicity properties for abstract data types, which exploit of the algebraic properties of the type's operations to ensure global consistency. Herlihy and Wing [HW90] describe a fine grain correctness condition called linearizability, which requires that each operation appear to "take effect" instantaneously, with the start order of non-concurrent operations preserved. Barghouti and Kaiser [BK91] survey the literature on transaction models for cooperative work. None of this work addresses the problems of interoperability among multiple consistency models applied to the same objects and possibly even the same operations.

3 Overview of MELD

In the MELD COOPS, each class defines instance variables and methods for every object of the class. Instance variables are strongly typed, where the type is primitive (integer, string, *etc.*), a programmer-defined class, or an array of variables of the same type (either primitive or programmer-defined). Instance variables and methods may be inherited from ancestor classes, using a multiple inheritance policy described in previous papers [KG87b, KG87a]. MELD methods may call arbitrary C functions, for example, to support mathematical and/or graphic applications or reuse existing code.

Figure 1 depicts part of a class definition for simple printer spoolers. Keywords are given in **boldface** font. There are three instance variables, `Q`, `P` and `L`; in each case the type is another programmer-defined class, `FileQueue`, `Printer` and `AcctLog`, respectively. The `Q` variable is initialized by sending the **create** message to the `FileQueue` class,[1] while the other two variables are implicitly initialized to **nil**. A new `PrintSpooler` object is created by executing "PrintSpooler.**create**()".

Three methods are defined in the figure. The `QueueFile` method adds a file to the print queue, logs the user's request for accounting purposes, and then uses **send** to invoke the `Daemon` method. If the queue is non-empty, `Daemon` prints the first element of the queue and removes it from the queue; if the queue is empty, it does nothing. `CancelFile` cancels a print job when requested by the user.

MELD supports two granularities of parallelism, within an object and among objects (which may reside on the same or different machines). A method may send a message to its own object with "**send** message **to $self**" or to another object with "**send** message **to** receiver", where the "message" includes the name of a method and its actual arguments and the "receiver" is the name of a variable that references the other object. The original method continues execution without concern for whether its message has been received and does not wait for an acknowledgment, so **send** is asynchronous.

A method may also make a procedure call to its own object using "**$self**.message" or to another object using the "receiver.message" construct. In either case, the calling

[1] MELD does not actually represent classes as full-fledged objects, so this "message" is interpreted specially by the run-time support.

```
CLASS PrintSpooler ::=
        {* FIFO queue of Files *}
        Q: FileQueue := FileQueue.create();
        {* Printer managed by instance *}
        P: Printer;
        {* Log for usage of Printer *}
        L: AcctLog;
METHODS:
        Daemon () ⟶
           if (not Q.Empty())
           then begin
                P.PrintFile(Q.First());
                Q.RemoveFirst();
           end
        QueueFile (F: File; U: User) ⟶
           begin
                Q.Enter(F);
                L.LogUsage(F, U);
                send Daemon() to $self;
           end
        CancelFile (F: File; U: User) ⟶
           begin
                Q.RemoveEntry(F, U);
                L.LogCancellation(F, U);
           end
        ...
END CLASS PrintSpooler
```

Figure 1: Printer Spooler Class

method blocks until the return, so procedure call is synchronous. A synchronously called method returns to its caller using the "**return** expression" construct, if a function, or by reaching the end of the method, if a procedure.[2]

When an object receives a message or a call, it (conceptually) spawns a new control thread to execute the method named in the message, and thus may execute multiple methods in parallel (some of which may be blocked awaiting returns from synchronous calls). The MELD implementation does not represent threads physically, but the effect achieved is the same as generating a new thread of control. When multiple methods execute in parallel, they may access the same instance variables.

Concurrency is thus maximized at the cost of nondeterminism and potential inconsistency since one method may view, or even overwrite, the intermediate values of instance variables written by another concurrently executing method. If the programmer knows that such read-write or write-write conflicts with never arise, due to the logic of the program, then this is not a problem and maximum concurrency is desirable. But if the logic of the program permits race conditions, then the programmer can use either atomic blocks or transactions, as described in the next two sections, respectively. Both atomic blocks and transactions effectively isolate the protected code sequence, so such conflicts cannot arise.

When strict isolation is not desirable, the programmer can explicitly program synchronization and information exchange between multiple on-going control threads. A method may send a message using the "**respond** message **to** receiver" construct, which is received when the recipient object (which may be the same object if the "receiver" is "**$self**") invokes a matching "**delayuntil** message". Alternatively, the **delayuntil** may happen first, and block until the corresponding **respond**. The **delayuntil** primitive will wait indefinitely if there is never a matching response (deadlocks are detected, and broken by halting the system). The names of responses must not conflict with method names, and are declared by "**response** name (arguments)". Figure 2 shows how these facilities could be used to implement a counting semaphore.

4 Atomic Blocks

An *atomic block* is a sequence of statements whose effects appear to be atomic with respect to the current object, that is, the entire object is locked for the duration of the atomic block. Any other code already executing within the object is suspended, and any arriving messages are queued until the atomic block terminates. Thus, no other code can read or update the instance variables of the object while the atomic block is executing. Atomic blocks combine the advantages of the critical sections common in

[2] This is the only distinction between functions and procedures, except that the type of the return value of a function is declared in its method header.

```
{* Semaphore for mutual exclusion *}
CLASS Semaphore ::=
        {* no instance variables needed *}
METHODS:
        {* declaration *}
        response Available();
        {* initialize counting semaphore by
           generating N responses *}
        Init (N: integer) ⟶
           if (N > 0)
           then begin
                respond Available() to $self;
                send Init(N-1) to $self;
           end
        {* wait until count is positive, and effectively decrement count by matching Available() response *}
        P() ⟶ delayuntil Available();
        {* effectively increment count by generating another response *}
        V() ⟶ respond Available() to $self;
END CLASS Semaphore
```

Figure 2: Semaphore Class

operating systems to protect sequences of code, and the locking common in database systems to protect pieces of data. Atomic blocks represent a minimal concurrency control policy that protects both code and data.[3]

Figure 3 illustrates how atomic blocks might be used for the classical producer-consumer problem. The `Init` method invokes `Spacing`, which signals the `N` spaces available in the buffer using the `SpaceAvailable` response. The `Produce` method blocks according to the **delayuntil** construct until a space is available, and then enters its atomic block – the code surrounded by the **begin atomic** and **end atomic** keywords (uncontrolled sequential code is enclosed between simple **begin** and **end**, and is known as a *sequential block*).[4]

The atomic block manipulates the queue representation, and then signals that a space in the buffer has been filled using the `EntryAvailable` response. The `Consume` method blocks until a space has been filled, and then enters its atomic block. Although it responds `SpaceAvailable` before returning, intended to signal a waiting `Produce` method, `Consume` continues to hold a lock on the `ProducerConsumer` object until after the atomic block terminates with the **return**.[5] A waiting `Produce` method's **delayuntil** on `SpaceAvailable` is not matched until after the lock is released.

Thus an atomic block cannot be corrupted, since no other threads can read or write any instance variable of an object while an atomic block is executing on that object. However, an atomic block can begin executing on an object in the middle of one or more concurrent sequential blocks, suspending them, and thus can view the partial results of the sequential blocks. If atomicity of a sequential block is necessary for the application, then the programmer should write it as an atomic block.

The degree of consistency guaranteed by atomic blocks is insufficient in some cases. Consider Figure 1 again, and now assume that each operation on a `FileQueue` is implemented by an atomic block. Say the printer queue is empty, and a user invokes `QueueFile` to queue a file for printing. `Daemon` executes `Empty`, which returns **false**. The **if** statement then enters its **then** part, preparing to invoke `First`. But say that in the meantime, the same user executes `CancelFile`. Then it might be possible that the `RemoveEntry` called by `Cancelfile` is scheduled before the call to `First` executed by `QueueFile`, in which case `First` would find an empty queue and some erroneous condition would result.

It may seem that we could solve this problem by implementing the `Daemon` method

[3]We are not currently concerned with atomicity in the sense of crash recovery, although MELD does support a persistent objectbase, and selected classes may be declared persistent – instances of all other classes are by default transient and exist only in memory.

[4]The implementation actually encloses atomic blocks in parentheses "(...)" and uncontrolled sequential blocks in square brackets "[...]", following a minimalist style borrowed from C, but we employ keywords in this paper for improved readability.

[5]It is possible that the scheduler may switch contexts immediately after the **return**, in which case the lock will actually be released only when the scheduler next selects this (conceptual) thread of control.

```
CLASS ProducerConsumer ::=
      Q: array[1..N] of T;
      P, C: integer := 0;
METHODS:
      {* declaration *}
      response SpaceAvailable(), EntryAvailable();
      {* initialization *}
      Init () ⟶ send Spacing(N) to $self;
      {* create S spaces *}
      Spacing (S : integer) ⟶
         if (S > 0)
         then begin
              respond SpaceAvailable() to self;
              send spacing(S-1) to $self;
         end
      Produce (X : T) ⟶
         begin
              delayuntil SpaceAvailable();
              begin atomic
                    P := P % N + 1;
                    Q[P] := X;
                    respond EntryAvailable() to self;
              end atomic
         end
      Consume () :  T ⟶
         begin
              delayuntil EntryAvailable();
              begin atomic
                    C := C % N + 1;
                    respond SpaceAvailable() to self;
                    return(Q[C]);
              end atomic
         end
END CLASS ProducerConsumer
```

Figure 3: Producer-Consumer Program

```
Daemon () ⟶
        begin atomic
              if (not Q.Empty())
              then begin
                    P.PrintFile(Q.First());
                    Q.RemoveFirst();
        end
        end atomic
```

Figure 4: Atomic Daemon Method

as an atomic block surrounding all its procedure calls to the object referenced by its `Q` variable, as shown in Figure 4. But this would not work as desired, since the atomic block would lock only the `PrintSpooler` object and not any of the objects referenced by its instance variables. If there were any other access paths to these objects, such as a second `PrintSpooler` object sending files to the same printer (for instance, the "don't care" queue illustrated in the next section), then the inconsistency described above could still occur. The basic problem is that an atomic block only protects one object (**$self**), not any of the other objects that may be sent messages from within the atomic block.

Another problem with atomic blocks is that there is a high potential for deadlock. The simplest case is a recursive procedure call. Since all calls and messages for the same object are queued until an atomic block terminates, the atomic block would wait for itself forever.

5 Serializable Transactions

Transaction blocks solve these problems with atomic blocks, and should be used instead of atomic blocks for any applications that require consistency across multiple objects and/or where the logic of the program may lead to deadlock. A transaction block is a sequence of statements executed as a serializable transaction with respect to all other MELD code ever executed.

Returning to the `PrintSpooler` example, the inconsistency problem could be solved by defining the `Daemon` method as a transaction rather than an atomic block, as shown in Figure 5. Transaction blocks appear between the keywords **begin transaction** and **end transaction**.[6]. Transaction blocks guarantee serializability, so it could never

[6]Double angle brackets "$\ll \ldots \gg$" are used in the implementation.

```
Daemon () ⟶
        begin transaction
              if (not Q.Empty())
              then begin
                   P.PrintFile(Q.First());
                   Q.RemoveFirst();
              end
        end transaction
```

Figure 5: Transaction Daemon Method

happen that `Empty` returns **false** but the subsequent call to `First` finds an empty queue.

MELD implements transactions using a distributed optimistic concurrency control mechanism based on multiple versions, developed by Agrawal *et al.* [ABGS87]. Transactions read the versions of objects that existed at the time they started, and update shadow copies of these objects. After all its work has completed, the transaction is validated by checking that its reads and writes are serializable with respect to all previously validated transactions. In the case of distributed transactions involving objects residing in multiple processes on the same or different machines, two-phase commit is used. Only if validation succeeds are the shadow copies made public. If validation fails, the shadow copies are forgotten and the transaction may be restarted.[7]

Validation of transactions is done at the granularity of individual instance variables and array elements, rather than the entire object as protected by atomic blocks, in order to increase concurrency within an object.[8] In the original `PrintSpooler` example, this has no useful effect since the sets of variables accessed by the three methods overlap. However, in the extended example of Figure 6, a single `PrintSpooler` object supports `N` file queues and `M` printers, say with `N` one more than `M` to allow a "don't care" queue for as fast as possible printing by the first available printer. Now multiple copies of `QueueFile` and `CancelFile` can execute simultaneously for different users, and at the same time as `Daemon`, as long as these methods do not update the same file queue or spool to the same printer.

[7]Restart of aborted transactions is treated as optional, with user confirmation required.

[8]Locking of individual instance variables and array elements by atomic blocks seemed highly likely to result in deadlock situations, so we decided against this alternative.

```
CLASS PrintSpooler ::=
       {* N = M+1 for first available *}
       Q: array[1..N] of FileQueue;
       P: array[1..M] of Printer;
       {* per queue *}
       L: array[1..N] of AcctLog;
METHODS:
       Init () ⟶ send BufferSpacing(N) to $self;
       BufferSpacing (B : integer) ⟶
          if (B > 0)
          then begin
               Q[B] := FileQueue.create();
               send BufferSpace(B-1) to $self;
          end
       Daemon () ⟶
          begintransaction
               if (not Q.Empty())
               then begin
                    P.PrintFile(Q.First());
                    Q.RemoveFirst();
               end
          end transaction
       QueueFile (F: File; U: User; myQ: integer) ⟶
          begin
               begintransaction
                    Q[myQ].Enter(F);
                    send LogUsage(F, U) to L[myQ];
               end transaction
               send Daemon() to $self;
          end
       CancelFile (F: File; U: User; myQ: integer) ⟶
          begintransaction
               Q[myQ].RemoveEntry(F, U);
               send LogCancellation(F, U) to L[myQ];
          end transaction
END CLASS PrintSpooler
```

Figure 6: Enforcing Serializability on Instance Variables

6 Interactions among Multiple Concurrency Control Policies

The integration of atomic blocks and serializable transactions in the same system with uncontrolled concurrency raises several interesting issues. First, in order to guarantee serializability, each transaction must be validated with respect to all atomic blocks and other MELD code executing over the same objectbase, not just all other transactions. We handle this by treating sequential and atomic blocks as sequences of "fake" transactions that appear, for the purposes of validation of real transactions, to have already committed. All statements in uncontrolled code and atomic blocks are immediately committed step-by-step as they execute, using fake transaction numbers, since transaction numbers are required by the optimistic concurrency control scheme.[9] If there is a conflict between non-transaction code and a transaction, the transaction is rolled back and may be restarted.

This adds a significant performance penalty to uncontrolled code and atomic blocks executing in the same system as transactions. In particular, each statement must first obtain a transaction number and then commit its read and write sets. The implementation recognizes the special case of non-transaction code and these processes are streamlined for the fake transactions (since no commit protocol is needed), but there does not seem to be any way to avoid some additional overhead. The compiler might check whether transactions are actually used in a given application, and generate optimized executables accordingly, but this could cause serious run-time errors with respect to separately compiled code operating on the same objectbase.

In addition to faking a transaction, each statement in a sequential block must also check the lock of the current object, to avoid conflicts with atomic blocks, but this is cheap by comparison. Each statement in a transaction block must check locks for the same reason. The code within atomic blocks does not need to check locks, since by definition if an atomic block is executing it holds the lock.[10]

Another cost imposed specifically by our optimistic implementation technique for implementing transactions is the maintenance of multiple shadow versions of instance variables. Each instance variable is represented as a sequence of public versions and a sequence of shadow versions. The public versions are ordered with the most recently committed first, and older versions following. The performance penalty here is negligible, except for the indirect effects of the additional memory space consumed.

Atomic blocks and other code not nested within a transaction access instance variables exactly as they would in a non-transactional system, except it is necessary to check for

[9] It would be more efficient for an atomic block to execute as a single fake transaction, but this is not supported in the implementation.

[10] We should note that neither transactions nor atomic blocks prevent corruption by concurrent C code embedded within MELD programs, since C statements do not respect the protection conventions.

each statement whether its transaction number matches the first public version. It will, and that will be the end of the search. In the case of transaction code, including the code of an atomic block executed *within* a transaction, the transaction number is also used. Access to an instance variable involves a search through the sequence of shadow versions if this instance variable has been updated by that transaction, and through the sequence of public versions if not, since a corresponding shadow version will not exist if this instance variable has not been updated by that transaction.

MELD's support for asynchronous message passing means that multiple threads may execute on behalf of the same transaction. This is a problem independent of the need to interoperate with non-transaction code. Since all threads generated from within a transaction carry the same transaction identifier, there is no synchronization among these threads, and the same nondeterminism and potential inconsistency arises as with uncontrolled code. The conventional approach to multi-threaded transactions is to support nested transactions [Mos85], with each thread initiated as a subtransaction.

We did not do this in MELD because we found some ambiguity in the notion of nested transactions when an optimistic concurrency control implementation is used. For example, it is not clear whether a subtransaction should validate only against its siblings, only against other subtransactions in its "family" rooted by a single top-level transaction, or against all transactions in the system. Further, for any of these alternatives, the required overhead would be unnecessarily high when there is only one level of subtransactions, and the subtransactions are short and execute only on a single object. As a compromise, MELD allows atomic blocks to be executed by threads directly or indirectly spawned from transaction blocks, as a minimalist subtransaction facility. The atomic block (in effect) views the shadow version of the object belonging to the transaction, rather than the latest committed version of the object. However, the atomic block still locks the entire object independent of its context.

7 Implementation Status

MELD is translated into C and runs on SunOS 4.0.3 on Sun 3s and Sun 4s, Ultrix 3.1 on DecStation 3100s, and AIX 2.2.1 and Mach 2.5 on IBM RT 135s. The MELD implementation is organized into five subsystems: (1) mpp, the MELD preprocessor, which implements multiple inheritance; (2) sc, which supports static semantic checking across separately compiled MELD modules, called *features*; (3) mc, the MELD compiler, which translates preprocessed Meld programs into C; (4) ml, the MELD linker, which combines separately compiled MELD features and links them together with the MELD run-time kernel and existing C code called from within MELD methods; and (5) the run-time kernel, which supports concurrency control policies, persistent data, the distributed name service, and other facilities. The total is about 24,000 lines of C code and `yacc` and `lex` rules.

Several small example systems have been implemented in MELD, including a multi-user stock trading application. We have also developed a small information model for distributed network management, involving throughput as reported by (simulated) data sensors.

8 Related Work

In Section 2, we considered related work on multiple concurrency control schemes for the same concurrency control policy, serializability. Here we compare MELD to a few other systems concerned with multiple concurrency control policies.

Argus [LCJS87] has both atomic and non-atomic objects, binding concurrency control to one level of abstraction. In contrast, MELD supports concurrency control both within and among objects, and gives the programmer the freedom to combine different concurrency control policies on the same object.

Camelot [SBD$^+$86] and Mach [JR86] together provide a transaction facility for objects, with a C language extension for programming [Blo89]. Avalon, which is implemented on top of Camelot, provides more complete language support for transactions, concentrating on fault tolerance rather than concurrency control, as extensions of C++ [DHW88] and CommonLisp [CLNW90]. Avalon's notion of consistency was influenced by Argus, and also binds atomicity to the object.

Hybrid [Nie87] has an atomic block construct that provides atomicity across multiple objects, but blocks other code from executing within any of those objects until the atomic block terminates. MELD's atomic blocks work similarly, but on single objects only; MELD's transactions provide atomicity across multiple objects, but permit more concurrency because serializability is enforced at the granularity of individual instance variables rather than entire objects.

In Clouds, concurrency control is not bound to the object level, and *standard* (uncontrolled), *local consistency preserving* (analogous to atomic blocks) and *global consistency preserving* (analogous to transactions) threads may be executed across multiple objects [CD89]. However, when threads with different consistency levels access the same object, the *lowest* level of consistency is supported (that is, standard < local < global), exactly the opposite of MELD – and exactly the opposite of what we believe most programmers would expect. Thus no guarantees can be made for consistency actually being preserved by global consistency preserving threads!

In the *transaction group* paradigm, transactions are collected into groups, and concurrency control within each group of transactions may employ a different policy although serializability is maintained among transactions in different groups [KSUW85, FZ89]. But this work does not make clear whether it is necessary to statically determine "groups" and/or the objects they may access *a priori*. This is not possible in our context, since the same code may execute either uncontrolled or as part of a transaction and the set of objects shared among control threads with the same or different policies is determined dynamically as the applications execute at run-time.

9 Conclusions

We began by motivating the need for multiple consistency models, and thus multiple concurrency control policies, in concurrent object-oriented programming systems. We described our experimentation with three concurrency control policies in the MELD COOPS. Uncontrolled sequential blocks are unsafe, but relatively fast. Atomic blocks are lightweight and guarantee consistency within an object, but are not sufficient for applications requiring consistency among multiple objects and/or processes. Serializable transactions are relatively expensive.

To integrate the three policies, we extended the run-time support so all other code checks the locks manipulated by atomic blocks and also extended atomic blocks to operate on the shadow versions of instance variables maintained by transactions. The most significant performance penalty seems to be due to the necessity to execute all statements of uncontrolled code and atomic blocks as if they were fake transactions in order to detect conflicts between transactions and non-transaction code.

During our experience designing and implementing MELD, including the integration of these three concurrency control policies, and the development of several small systems using MELD, we have learned much about COOPS as well as concurrency control. We have recently redesigned MELD from scratch, with the new language named MELDC [GH91]. The primary goal of our new effort is high performance for distributed COOPS applications written in MELDC.

Acknowledgments

Ronald Chacar implemented the toy network management system in MELD. Quoc-Bao Nguyen ported MELD from SunOS to AIX, Mach and Ultrix, fixing several bugs in the concurrency control implementation in the process. David Staub developed MELD's distributed name service. David Garlan collaborated with the first author on the original design for a non-concurrent MELD.

The Programming Systems Laboratory was supported by National Science Foundation grants CCR-9000930, CDA-8920080 and CCR-8858029, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

An extended abstract of this paper appeared in the *Second IEEE Symposium on Parallel and Distributed Processing*, Dallas TX, December 1990, pp. 623-626.

References

[ABGS87] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Journal of Distributed Computing*, 2(1):45–59, April 1987.

[Agh90] *ECOOP-OOPSLA Workshop on Object-Based Concurrent Programming*. ACM Press, October 1990. Special issue of *OOPS Messenger*, 2(2), April 1991.

[(AN89] Secretariat: USA (ANSI). Information retrieval, transfer and management for osi, part 1: Management information model. Technical Report ISO/IEC JTC1/SC21 N, ISO International Organization for Standardization, May 1989.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading MA, 1987.

[BK91] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[Blo89] Joshua J. Bloch. The Camelot library: A C language extension for programming a general purpose distributed transaction system. In *9th International Conference on Distributed Computing Systems*, pages 172–180, Newport Beach CA, June 1989. Computer Society Press.

[CD89] Raymond C. Chen and Partha Dasgupta. Linking consistency with object/thread semantics: An approach to robust computation. In *9th International Conference on Distributed Computing Systems*, pages 121–128, Newport Beach CA, June 1989.

[CLNW90] Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing. Reliable distributed computing with avalon/common lisp. In *International Conference on Computer Languages*, pages 169–179, New Orleans LA, March 1990.

[DHW88] David Detlefs, Maurice Herlihy, and Jeannette Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *Computer*, 21(12):57–69, December 1988.

[ED90] Ahmed K. Elmagarmid and Weimin Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *6th International Conference on Data Engineering*, pages 37–46, Los Angeles CA, February 1990.

[FZ89] Mary F. Fernandez and Stanley B. Zdonik. Transaction groups: A model for controlling cooperative work. In *3rd International Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 128–138, Queensland, Australia, January 1989.

[GH91] Howard Gershen and Erik Hilsdale. Navigating the meldc the meldc user's manual. Technical Report CUCS-031-91, Columbia University Department of Computer Science, October 1991.

[GM83] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[Gre88] Irene Greif, editor. *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufmann, San Mateo CA, 1988.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[JR86] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 67–77, Portland, OR, September 1986. Special issue of *SIGPLAN Notices*, 21(11), November 1986.

[Kai90] Gail E. Kaiser. A flexible transaction model for software engineering. In *6th International Conference on Data Engineering*, pages 560–567, Los Angeles CA, February 1990. IEEE Computer Society.

[KG87a] Gail E. Kaiser and David Garlan. MELDing data flow and object-oriented programming. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 254–267, Orlando FL, October 1987. Special issue of *SIGPLAN Notices*, 22(12), December 1987.

[KG87b] Gail E. Kaiser and David Garlan. Melding software systems from reusable building blocks. *IEEE Software*, 4(4):17–24, July 1987.

[KH92] Gail E. Kaiser and Brent Hailpern. An object-based programming model for shared data. *ACM Transactions on Programming Languages and Systems*, January 1992. In press. Available as IBM Research Report RC 16442 and Columbia University Department of Computer Science CUCS-046-90, December 1990.

[KPHW89] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush, and Shyhtsun Felix Wu. MELDing multiple granularities of parallelism. In Stephen Cook, editor, *3rd European Conference on Object-Oriented Programming*, British Computer Society Workshop Series, pages 147–166, Nottingham, UK, July 1989. Cambridge University Press.

[KSUW85] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes. A transaction model supporting complex applications in integrated information systems. In *ACM-SIGMOD 1985 International Conference on Management of Data*, pages 388–401, Austin TX, May 1985. ACM.

[LCJS87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of argus. In *11th ACM Symposium on Operating Systems Principles*, pages 111–122, Austin TX, November 1987. Special issue of *Operating Systems Review*, 21(5), 1987.

[Lyn83] Nancy A. Lynch. Multilevel atomicity — a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[Mey90] Norman Meyrowitz, editor. *OOPSLA ECOOP '90 Proceedings Object-Oriented Programming: Systems, Languages and Applications Conference European Conference on Object-Oriented Programming*, Ottawa, Canada, October 1990. ACM Press. Special issue of *SIGPLAN Notices*, 25(10), October 1990.

[Mos85] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. Information Systems. The MIT Press, Cambridge MA, 1985. PhD Thesis, MIT LCS TR-260, April 1981.

[Ne89] Erich Neuhold and Michael Stonebraker (editors). Future directions in DBMS research. *SIGMOD Record*, 18(1):17–26, March 1989.

[Nie87] O. M. Nierstrasz. Active objects in hybrid. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 243–253, Orlando FL, October 1987. Special issue of *SIGPLAN Notices*, 22(12), December 1987.

[NW90] Marie-Anne Neimat and Kevin Wilkinson. Extensible transaction management in papyrus. In Bruce Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 503–511, Kona HI, January 1990.

[Pae91] Andreas Paepcke, editor. *OOPSLA '91 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Phoenix AZ, October 1991. ACM Press. Special issue of *SIGPLAN Notices*, 26(11), November 1991.

[PKH88] Calton Pu, Gail E. Kaiser, and Norman Hutchinson. Split-transactions for open-ended activities. In Francois Bancilhon and David J. Dewitt, editors, *14th International Conference on Very Large Data Bases*, pages 26–37, Los Angeles CA, August 1988.

[Pu88] Calton Pu. Superdatabase for composition of heterogeneous databases. In *4th International Conference on Data Engineering*, pages 548–555, Los Angeles CA, February 1988.

[Ram91] S. Ram, editor. *Special Issue on Heterogeneous Distributed Databases Systems*, volume 24:12 of *IEEE Computer Magazine*. IEEE Computer Society, December 1991.

[SBD+86] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson. The camelot project. *Database Engineering*, 9(4), December 1986.

[Sta88] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988.

[Wei89] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.

Complete Example MELD Program

We include a complete MELD program below. It has an ATM-like interface supporting multiple users, who may share the same bank accounts. The users purchase stocks by expending the funds in their accounts, and their accounts are incremented when they sell stocks. This example system has been used in our recent demos. Note that here we use "[...]", "(...)" and "≪ ... ≫" to bracket uncontrolled sequential code, atomic blocks and transaction blocks, respectively, since this is the syntax supported in the MELD implementation.

```
FEATURE Bank
INTERFACE:
EXPORTS ChkAcct
IMPLEMENTATION:

PERSISTENT CLASS ChkAcct ::=
  balance: real;
METHODS:
  printf("Balance of %s is now %.2f\n",
         object_name($self), balance);

  deposit(money: real; send_obj: name): boolean --> (
    balance := balance + money;
    send deposit_msg(money) to send_obj;
    return(true);
   )

  withdraw(money: real; send_obj: name): boolean --> (
    if (balance < money)
      then [
       send nsf_msg(money) to send_obj;
       return(false);
      ]
    else [
      balance := balance - money;
      send withdraw_msg(money) to send_obj;
      return(true);
    ]
  )

  balance(): integer --> (
```

```
    amount: integer;
    temp_amount: real;
    temp_amount := balance * 100;
    amount := temp_amount;
    return(amount);
  )
END CLASS ChkAcct
END FEATURE Bank

FEATURE BankManager
INTERFACE:
IMPORTS Bank
IMPLEMENTATION:
OBJECT:
  bnkm: BankManager := BankManager.create();

CLASS BankManager ::=
METHODS:
 "create %s"(acctName: string) -->
  [
   acct: ChkAcct;
   acct := ChkAcct.create(acctName);
   send "%s created\n"(acctName) to stdout;
  ]

 "balance %s"(acctName: string) -->
  [
   acct: ChkAcct;
   balance: integer;
   if (strlen(acctName) > 0)
    then [
     acct := ChkAcct.get(acctName);
     if (acct != nil)
      then [
       balance := acct.balance();
       send "%s balance is %d.%02d\n"(acctName,
            balance / 100, balance % 100) to stdout;
      ]
     else send "Can't find %s\n"(acctName) to stdout;
    ]
   else send "Please provide an account name\n"
             to stdout;
```

```
 ]

"deposit %f %s"(amount: real; acctName: string) -->
 [
  acct: ChkAcct;
  if (strlen(acctName) > 0)
   then [
    acct := ChkAcct.get(acctName);
    if (acct != nil)
     then send deposit(amount, $self) to acct;
    else send "Can't find %s\n"(acctName) to stdout;
   ]
  else send "Please provide an account name\n"
            to stdout;
 ]

"withdraw %f %s"(amount: real; acctName: string) -->
 [
  acct: ChkAcct;
  if (strlen(acctName) > 0)
   then [
    acct := ChkAcct.get(acctName);
    if (acct != nil)
     then send withdraw(amount, $self) to acct;
    else send "Can't find %s\n"(acctName) to stdout;
   ]
  else send "Please provide an account name\n"
             to stdout;
 ]

 deposit_msg(money: real) --> [
  send "Deposited %.2f\n"(money) to stdout;
 ]

 nsf_msg(money: real) --> [
  send "Non-sufficient funds to withdraw %.2f\n"(money)
       to stdout;
 ]

 withdraw_msg(money: real) --> [
  send "Withdrew %.2f\n"(money) to stdout;
 ]
```

```
END CLASS BankManager
END FEATURE BankManager


FEATURE Stocks
INTERFACE:
IMPORTS Bank
EXPORTS Stock, Portfolio
IMPLEMENTATION:
%{
   /* directive allowing inclusion of arbitrary C code */
   char *
   object_name(struct object *obj)
   {
     return obj->name;
   }
%}
OBJECT:

PERSISTENT CLASS Stock ::=
  priceDollars, priceEighths: integer;
  stockName: string;
METHODS:
  stockName := object_name($self);

  quote(): integer --> (
   return(priceDollars *  8 + priceEighths);
  )

  update(newprice: real) --> <<
   priceDollars := (integer)newprice;
   priceEighths := (integer)(newprice * 8) - priceDollars * 8;
   if (priceEighths == 0)
    then send "Price of %s is now %d\n"(stockName,
               priceDollars) to stdout;
   else if (priceEighths == 4)
    then send "Price of %s is now %d 1/2\n"(stockName,
              priceDollars) to stdout;
   else if (priceEighths % 2)
    then send "Price of %s is now %d %d/8\n"
      (stockName, priceDollars, priceEighths) to stdout;
   else send "Price of %s is now %d %d/4\n"
```

```
    (stockName, priceDollars, priceEighths / 2) to stdout;
  >>
END CLASS Stock

PERSISTENT CLASS Portfolio ::=
  stockNames: array[0..4] of string;
  stockHoldings: array[0..4] of integer;
METHODS:
  pfInit() --> [
   stockNames[0] := "IBM";
   stockNames[1] := "DEC";
   stockNames[2] := "SUN";
   stockHoldings[0] := 100;
   stockHoldings[1] := 50;
   stockHoldings[2] := 25;
   stockHoldings[3] := 0;
   stockHoldings[4] := 0;
  ]

  printHold(index: integer; send_obj: name) --> [
   if (stockHoldings[index] > 0)
    then send "%d shares of %s\n"(stockHoldings[index],
              stockNames[index]) to stdout;
   if (index < 4)
    then send printHold(index + 1, send_obj) to $self;
   else send printing_done() to send_obj;
  ]

  printpf() --> <<
   send "Your current stock holdings are:\n" to stdout;
   send printHold(0, $sender) to $self;
  >>

  printpf_noxact() --> [
   send "Your current stock holdings are:\n" to stdout;
   send printHold(0, $sender) to $self;
  ]

  buy(BuyName: string; buyNumShares: integer; acct: ChkAcct) -->
  <<
   stk: Stock;
   i: integer;
```

```
   stkPrice: integer;
   stk := Stock.get(BuyName);
   if (stk != nil)
    then [
     stkPrice := stk.quote();
     if (acct.withdraw((real) stkPrice * buyNumShares / 8,
                        $sender))
      then [
       i := $self.findStock(BuyName);
       if (i < 0)
then [
 i := $self.findSpace();
 $self.setHoldings(i, buyNumShares, BuyName);
 delay 5;
 send buy_msg(buyNumShares, BuyName, i) to $sender;
]
       else [
$self.updateHoldings(i, buyNumShares);
delay 5;
send buy_msg(buyNumShares, BuyName, i) to $sender;
       ]
      ]
     else send funds_msg() to $sender;
    ]
   else send no_such(BuyName) to $sender;
  >>

  findStock(StockName: string) -->
   return($self.pfSearchName(StockName, 0));

  pfSearchName(StockName:  string; stockIndex: integer) --> [
   if (stockNames[stockIndex] <> nil &&
       strcmp(stockNames[stockIndex], StockName) == 0)
    then return(stockIndex);
   else [
    if (stockIndex = 4)
     then return(-1);
    else return($self.pfSearchName(StockName, stockIndex + 1));
   ]
  ]

  findSpace() -->
```

```
  return($self.pfSearchOpen(0));

 pfSearchOpen(stockIndex: integer) --> [
  if (stockNames[stockIndex] = nil)
   then return(stockIndex);
  else [
   if (stockIndex = 4)
    then return(-1);
   else return($self.pfSearchOpen(stockIndex + 1));
  ]
 ]

 sell(SellName: string; sellNumShares: integer;
      acct: ChkAcct) --> <<
  stk: Stock;
  i: integer;
  stkPrice: integer;
  stk := Stock.get(SellName);
  if (stk != nil)
   then [
    stkPrice := stk.quote();
    i := $self.findStock(SellName);
    if (i < 0)
     then send no_shares(SellName) to $sender;
    else [
     if (stockHoldings[i] < sellNumShares)
      then send too_few(SellName, i) to $sender;
     else [
      $self.updateHoldings(i, -sellNumShares);
      if (acct.deposit((real)stkPrice * sellNumShares / 8,
          $sender) == false)
then send no_deposit() to $sender;
      delay 5;
      send sell_msg(sellNumShares, SellName, i) to $sender;
     ]
    ]
   ]
  else send no_such(SellName) to $sender;
 >>

 setHoldings(i, SetNumShares: integer; SetName: string) --> [
  stockHoldings[i] := SetNumShares;
```

```
   stockNames[i] := SetName;
  ]

  updateHoldings(i, NewNumShares: integer) -->
   stockHoldings[i] := stockHoldings[i] + NewNumShares;

  remoteNames(i: integer): string -->
   return(stockNames[i]);

  remoteHoldings(i: integer) -->
   return(stockHoldings[i]);

  atomic_buy(StockName: string; stockNumShares: integer;
             acct: ChkAcct) --> [
   stk: Stock;
   i: integer;
   stkPrice: integer;
   stk := Stock.get(StockName);
   if (stk != nil)
    then [
     stkPrice := stk.quote();
     if (acct.withdraw((real) stkPrice * stockNumShares / 8,
                       $sender))
      then [
       i := $self.changeStock(StockName, stockNumShares);
       if (i < 0)
then [
 send "Can't buy a completely new stock atomically\n"
              to stdout;
 send sorry_msg() to $sender;
]
       else [
delay 5;
send buy_msg(stockNumShares, StockName, i)
             to $sender;
       ]
      ]
     else send funds_msg() to $sender;
    ]
   else send no_such(StockName) to $sender;
  ]
```

```
  atomic_sell(StockName: string; stockNumShares: integer;
              acct: ChkAcct) --> [
   stk: Stock;
   i: integer;
   stkPrice: integer;
   stk := Stock.get(StockName);
   if (stk != nil)
    then [
     stkPrice := stk.quote();
     if (acct.deposit((real) stkPrice * stockNumShares / 8,
                      $sender))
      then [
       i := $self.changeStock(StockName, -stockNumShares);
       if (i < 0)
       then [
 i := $self.findStock(StockName);
 send too_few(StockName, i) to $sender;
]
       else [
delay 5;
send sell_msg(stockNumShares, StockName, i)
             to $sender;
       ]
      ]
     else send funds_msg() to $sender;
    ]
   else send no_such(StockName) to $sender;
  ]

  reserveSpace(StockName: string; stockNumShares: integer):
      integer -->
   return($self.pfTakeOpen(0, StockName, stockNumShares));

  pfTakeOpen(stockIndex: integer; StockName: string;
     stockNumShares: integer): integer --> [
   grabbed: boolean;
   grabbed := false;
   (
    if (stockNames[stockIndex] = nil)
     then [
      stockNames[stockIndex] := StockName;
      stockHoldings[stockIndex] := stockNumShares;
```

```
    grabbed := true;
   ]
  )
  if (grabbed)
   then return(stockIndex);
  else [
   if (stockIndex = 4)
    then return(-1);
   else return
        $self.pfTakeOpen(stockIndex + 1, StockName,
                         stockNumShares);
  ]
 ]

 changeStock
 (StockName: string; stockNumShares: integer):
     integer -->
  return
  $self.pfSearchAdd(StockName, 0, stockNumShares);

 pfSearchAdd(StockName: string; stockIndex: integer;
     stockNumShares: integer): integer --> [
  found: boolean;
  ok: boolean;
  found := false;
  (
   if (stockNames[stockIndex] <> nil &&
strcmp(stockNames[stockIndex], StockName) == 0)
   then [
    if (stockNumShares < 0 &&
        stockHoldings[stockIndex] < -stockNumShares)
     then ok := false;
    else [
     stockHoldings[stockIndex] :=
      stockHoldings[stockIndex] + stockNumShares;
     ok := true;
    ]
    found := true;
   ]
  )
  if (found)
   then [
```

```
     if (ok)
      then return(stockIndex);
     else return(-1);
    ]
   else [
    if (stockIndex = 4)
     then return(-1);
    else return
         $self.pfSearchAdd(StockName, stockIndex + 1,
                           stockNumShares);
   ]
  ]
END CLASS Portfolio
END FEATURE Stocks


FEATURE StockManager
INTERFACE:
IMPORTS Bank, Stocks
IMPLEMENTATION:
OBJECT:
  stkm: StockManager := StockManager.create();
  pfm: PortfolioManager := PortfolioManager.create("PM");

CLASS StockManager ::=
METHODS:
 "stock %s"(stockName: string) -->
  [
   stk: Stock;
   stk := Stock.create(stockName);
   send "%s created\n"(stockName) to stdout;
  ]

 "update %d.%d %s"
 (pDols, pCts: integer; stockName: string) -->
  [
   stk: Stock;
   price: real;
   stk := Stock.get(stockName);
   if (stk != nil)
    then [
     price := (real)pDols + (real)pCts / 100;
```

```
     stk.update(price);
    ]
   else send "Can't find %s\n"(stockName) to stdout;
  ]

 "quote %s"(stockName: string) -->
  [
   stk: Stock;
   stkPrice: integer;
   priceDollars: integer;
   priceEighths: integer;
   stk := Stock.get(stockName);
   if (stk != nil)
    then [
     stkPrice := stk.quote();
     priceDollars := stkPrice / 8;
     priceEighths := stkPrice % 8;
     if (priceEighths == 0)
      then send "Price of %s is %d\n"
                 (stockName, priceDollars) to stdout;
     else if (priceEighths == 4)
      then send "Price of %s is %d 1/2\n"
                 (stockName, priceDollars) to stdout;
   else if (priceEighths % 2)
    then send "Price of %s is %d %d/8\n"
               (stockName, priceDollars, priceEighths)
               to stdout;
   else send "Price of %s is %d %d/4\n"
              (stockName, priceDollars, priceEighths / 2)
              to stdout;
    ]
   else send "Can't find %s\n"(stockName) to stdout;
  ]
END CLASS StockManager

PERSISTENT CLASS PortfolioManager ::=
pfs: array [0..4] of Portfolio;
npfs: integer;
METHODS:
 "create %s"(pfName: string) -->
  [
   pf: Portfolio;
```

```
  pf := Portfolio.create(pfName);
  send "%s created\n"(pfName) to stdout;
  (
   pfs[npfs] := pf;
   npfs := npfs + 1;
  )
 ]

"print %s"(pfName: string) -->
 [
  pf: Portfolio;
  if (strlen(pfName) > 0)
   then [
    pf := Portfolio.get(pfName);
    if (pf != nil)
     then [
      send printpf() to pf;
     ]
    else send "Can't find %s\n"(pfName) to stdout;
   ]
  else send "Please provide a portfolio name\n"
            to stdout;
 ]

 printAll() --> <<
  send printpfs(0) to $self;
 >>

 printpfs(pfIndex: integer) --> [
  pfName: string;
  if (pfs[pfIndex] != nil)
   then [
    pfName := object_name(pfs[pfIndex]);
    send "For the %s portfolio:\n"(pfName) to stdout;
    send printpf_noxact() to pfs[pfIndex];
    delayuntil printing_done;
    if (pfIndex < 4)
     then send printpfs(pfIndex + 1) to $self;
   ]
 ]

"Print" -->
```

```
  send printAll() to $self;

 printing_done() -->
  [
   i: integer;
   i := 0;
  ]

 "init %s"(pfName: string) -->
  [
   pf: Portfolio;
   if (strlen(pfName) > 0)
    then [
     pf := Portfolio.get(pfName);
     if (pf != nil)
      then [
       send pfInit() to pf;
      ]
     else send "Can't find %s\n"(pfName) to stdout;
    ]
   else send "Please provide a portfolio name\n"
             to stdout;
  ]


END CLASS PortfolioManager
END FEATURE StockManager


FEATURE StockUser
INTERFACE:
IMPORTS Stocks, Bank
IMPLEMENTATION:
OBJECT:
  start: Startup := Startup.create();
  user: User;

CLASS User ::=
  portfolio: Portfolio;
  chkacct: ChkAcct;
  username: string;
METHODS:
```

```
"open %s"(newname: string) --> [
 username := newname;
 portfolio := Portfolio.get(newname);
 if (portfolio != nil)
  then send "Found portfolio for %s\n"
            (newname) to stdout;
 else send "Can't find portfolio for %s\n"
           (newname) to stdout;
 chkacct := ChkAcct.get(newname);
 if (chkacct != nil)
  then send "Found checking account for %s\n"
            (newname) to stdout;
 else send "Can't find checking account for %s\n"
           (newname) to stdout;
]

"buy %d %s"
(stockNumShares: integer; stockName: string) -->
 portfolio.buy(stockName, stockNumShares, chkacct);

"sell %d %s"
(stockNumShares: integer; stockName: string) -->
 portfolio.sell(stockName, stockNumShares, chkacct);

"deposit %f"(money: real) -->
 chkacct.deposit(money, $self);

"withdraw %f"(money: real) -->
 chkacct.withdraw(money, $self);

pfPrintHold(index: integer) --> [
 holdings: integer;
 names: string;
 holdings := portfolio.remoteHoldings(index);
 names := portfolio.remoteNames(index);
 if (holdings > 0)
  then send "%d shares of %s\n"(holdings, names) to stdout;
 if (index < 4)
  then send pfPrintHold(index + 1) to $self;
]

pfPrintpf() --> <<
```

```
   send "Your current stock holdings are:\n" to stdout;
   send pfPrintHold(0) to $self;
  >>

  "print" -->
    send pfPrintpf() to $self;

  "balance" --> [
    balance: integer;
    balance := chkacct.balance();
    send "Your balance is %d.%02d\n"
         (balance / 100, balance % 100) to stdout;
  ]

  "Buy %d %s"
  (stockNumShares: integer; stockName: string) -->
    portfolio.atomic_buy
    (stockName, stockNumShares, chkacct);

  "Sell %d %s"
  (stockNumShares: integer; stockName: string) -->
    portfolio.atomic_sell
    (stockName, stockNumShares, chkacct);

  buy_msg
  (buyNumShares: integer; BuyName: string; i: integer) -->
  [
   currentHoldings: integer;
   currentHoldings := portfolio.remoteHoldings(i);
   send "Bought %d shares of %s; you now hold %d\n"
(buyNumShares, BuyName, currentHoldings)
        to stdout;
  ]

  sell_msg(sellNumShares: integer; SellName: string;
           i: integer) --> [
   currentHoldings: integer;
   currentHoldings := portfolio.remoteHoldings(i);
   send "Sold %d shares of %s; you now hold %d\n"
(sellNumShares, SellName, currentHoldings)
        to stdout;
  ]
```

```
funds_msg() --> [
 send "Insufficient funds to purchase stock\n" to stdout;
]

no_such(stockName: string) --> [
 send "No such stock as %s\n"(stockName) to stdout;
]

no_shares(stockName: string) --> [
 send "You hold no shares of %s\n"
      (stockName) to stdout;
]

too_few(stockName: string; i: integer) --> [
 currentHoldings: integer;
 currentHoldings := portfolio.remoteHoldings(i);
 send "You hold only %d shares of %s\n"
(currentHoldings, stockName) to stdout;
]

no_deposit() --> [
 send "Can't deposit proceeds from sale of stock\n"
      to stdout;
]

"quote %s"(stockName: string) --> [
 stk: Stock;
 stkPrice: integer;
 priceDollars: integer;
 priceEighths: integer;
 stk := Stock.get(stockName);
 if (stk != nil)
  then [
   stkPrice := stk.quote();
   priceDollars := stkPrice / 8;
   priceEighths := stkPrice % 8;
   if (priceEighths == 0)
    then send "Price of %s is %d\n"
              (stockName, priceDollars) to stdout;
   else if (priceEighths == 4)
    then send "Price of %s is %d 1/2\n"
```

```
                (stockName, priceDollars) to stdout;
   else if (priceEighths % 2)
    then send "Price of %s is %d %d/8\n"
     (stockName, priceDollars, priceEighths) to stdout;
   else send "Price of %s is %d %d/4\n"
     (stockName, priceDollars, priceEighths / 2) to stdout;
    ]
   else send "Can't find %s\n"(stockName) to stdout;
  ]

  deposit_msg(money: real) --> [
   send "Deposited %.2f\n"(money) to stdout;
  ]

  nsf_msg(money: real) --> [
   send "Non-sufficient funds to withdraw %.2f\n"
        (money) to stdout;
  ]

  withdraw_msg(money: real) --> [
   send "Withdrew %.2f\n"(money) to stdout;
  ]

  sorry_msg() --> [
   send "Sorry, I can't do that\n" to stdout;
  ]
END CLASS User

CLASS Startup ::=
METHODS:
 "user %s"(username: string) --> [
  user := User.create(username);
  ]
END CLASS Startup
END FEATURE StockUser
```