Object-Oriented Programming Language Facilities for Concurrency Control

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

April 1989

CUCS-439-89

Abstract

Concurrent object-oriented programming systems require support for concurrency control, to enforce consistent commitment of changes and to support program-initiated rollback after application-specific failures. We have explored three different concurrency control models — atomic blocks, serializable transactions, and commit-serializable transactions — as part of the MELD programming language. We present our designs, discuss certain programming problems and implementation issues, and compare our work on MELD to other concurrent object-based systems.

Copyright © 1989 Gail E. Kaiser

Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

1. Introduction

Concurrent object-oriented programming systems (COOPS) require support for concurrency control, to enforce consistent commitment of changes to collections of objects and to support program-initiated rollback after application-specific failures involving updates to shared objects. It is sometimes suggested that the classical transaction model, successfully applied in databases and operating systems, be integrated directly into COOPS facilities. This is clearly desirable, but by itself too limiting. COOPS applications require *several granularities* of transaction-like facilities.

The classical transaction model [5] permits activities to be defined as atomic and serializable transactions: either all the operations carried out during a transaction complete and commit or none of them do, and from the viewpoint of the objects the set of transactions committed over the lifetime of the system appear to have been executed in some serial order. Nested transactions [19] permit concurrency among subactivities and failure of subactivities without forcing the top-level transaction to abort.

This model is unfortunately insufficient for many applications suitable for COOPS implementation, and unnecessarily inefficient for others. Some applications require strict serializability, that is, not only do transactions appear to have been executed in some serial order, but they must appear to have been executed in exactly the serial order in which they were initiated. This is necessary to satisfy any first-come/first-served policy. Some applications must appear to have been executed in some, perhaps strict, serial order with respect to an external device such as a printer or an external agent such as a human user, as well as with respect to the objects within the system. Real-time applications [25] add special timing constraints and require predictable concurrent behavior.

Some applications are able to accept a degree of inconsistency in exchange for greater concurrency: for example, in statistical systems, small inconsistencies may have negligible effects on computations. Some applications permit forking of versions to allow greater concurrency, that is, multiple attempts to update the same object cause the creation of multiple copies of the object each with a distinct version identifier. In some cases the versions may persist, while others require application-specific merge operations. Audit requirements for other applications may make any form of version forking unthinkable.

To be truly general purpose, a COOPS must be able to meet the requirements of all of these very different kinds of applications. This can be accomplished by providing only very low-level primitives, such as semaphores, and burden the programmers with constructing the necessary mechanisms and building on top of them. But the whole point of first OOPS, and then COOPS, was to unburden the programmer from low-level details and allow him to work at a level closer to the problem domain. So maybe COOPS cannot, or should not attempt to be, truly general purpose. We do not intend to argue on either side of this question here. Instead, we would like to demonstrate the application of several transaction models to the same COOPS, and describe their advantages and disadvantages, and then discuss the problems of allowing multiple transaction models to coexist within the same COOPS.

In the MELD programming language, we have explored three different concurrency control models — atomic blocks, serializable transactions, and commit-serializable transactions — and developed corresponding programming language facilities for each model. We use the term "transaction" rather loosely, since two of the three concurrency control schemes explored relax the serializability requirement. Note also that we are concerned only with concurrency control and have so far ignored crash recovery, although this is clearly necessary for persistent objects and/or long-running applications.

Atomic blocks are critical sections with respect to a particular object, so they can be used to enforce strict serializability locally, but without global consistency. Serializable transactions follow the classical transaction model, although we are using a relatively unusual optimistic technique for enforcing serializability. One interesting aspect of our application of this technique to a COOPS is representation of the transaction itself as an object and the create, commit and abort operations as messages. Commit-serializable transactions are a flexible transaction mechanism allowing cooperation among distinct transactions and greater concurrency than would otherwise be possible.

First we introduce the relevant MELD facilities necessary as background for transaction processing. Then we describe each of the three concurrency control models in turn, together with the corresponding language constructs and their semantics. We briefly describe the status of the implementation effort, and this is followed by a comparison to concurrency control facilities in other COOPS. The appendix gives a small example program using atomic blocks.

2. Overview of MELD

MELD is a concurrent object-oriented programming language, whose primary motif is providing a wide range of programming facilities by supporting multiple granularities of a small number of fundamental programming concepts. MELD supports two granularities of encapsulation and reusability [8]: classes and modules. There is actually a third granularity, from our solution to the "multiple inheritance problem", discussed elsewhere [9]. Inheritance would complicate the later discussion of transactions, so is ignored here.

Classes provide medium grain encapsulation. Each class is essentially an abstract data type that defines instance variables (private data), methods (operations), and *constraints*. Constraints are statements not associated with any named operation, and are automatically executed as needed to maintain integrity constraints among the instance variables; this is a simple form of active values [26] and distinct from more general constraint programming languages [12]. Instance variables are strongly typed, where the type is a built-in class (integer, string, *etc.*), a built-in constructor (array, sequence, set, table), or a programmer-defined class or union (a set of alternative classes).

```
CLASS PrintSpooler ::=
    Q: FileQueue := FileQueue.create;
    P: Printer; {* Printer managed by this PrintSpooler *}
    L: AcctLog; {* Log for usage of Printer *}

METHODS:

    {* constraint *}
    if (not Q.Empty()) then [
        P.PrintFile(Q.First());
        Q := Q.RemoveFirst(); ]

    {* method *}
    QueueFile (F: File; U: User) -->
    [ Q := Q.Enter(F);
      L.LogUsage(F, User); ]

    ...

END CLASS PrintSpooler
```

Figure 2-1: Print Spooler Class

Figure 2-1 depicts part of the class definition for simple print spoolers. Comments are given between curly braces "{...}". There are three instance variables, Q, P and L; in each case the type

is another programmer-defined class. The Q variable is initialized by sending the `create` message to the `FileQueue` class, while the other two variables are implicitly initialized to nil. One constraint and one method are defined. Whenever the queue is non-empty, the constraint automatically prints the first element of the queue and removes it from the queue. (In a concurrent system, the constraint would have to be an atomic block, as would the `RemoveFirst` method, not shown.) The method, `QueueFile`, adds a file to the print queue and logs the user's request for accounting purposes. (`QueueFile` would not have to be atomic unless the accounting system required correct ordering of print requests; the `Enter` method would have to be atomic in any case.)

Features provide large grain encapsulation. Classes are the standard unit of reusability in object-oriented languages, but we believe classes are too small — the cost of retrieving, understanding and specializing the reusable unit may be larger than the cost of developing it from scratch, and there's also the cost of organizing the library of reusable units [14]. A practical reusable unit consists of related classes, unions and global (to the feature) objects bundled together to provide a coherent functionality.

MELD's feature construct is a modular unit with a public interface consisting of an export clause and an import clause, and a private implementation. The import clause lists other features whose exported classes, unions and global objects may be used for types of instance variables or as superclasses. The export clause may optionally indicate for each class the subset of its instance variables and methods available to externally defined subclasses and clients, respectively. This supports finer control over visibility than available in most data abstraction languages, along the lines of TrellisOwl [22] and CommonObjects [23], where interfaces are defined at the class granularity.

The `PrintServer` feature depicted in Figure 2-2 imports the `Files` feature, which provides the `File` and `FileQueue` classes. Only the `PrintSpooler` class is exported, and the `Printer`, `AcctLog` and other classes are internal to the implementation. No global objects are defined for this feature, since distinct printer and log objects are created for each print spooler by the `Initialize` method, not shown. The imports list could have been written "IMPORTS Files[File, FileQueue]", to indicate the specific *view* of the feature that is imported; similarly, the exports list could have been written "EXPORTS PrintSpooler[QueueFile]", to make only this one method visible to external clients.

```
FEATURE PrintServer

INTERFACE:
    IMPORTS Files;
    EXPORTS PrintSpooler;

IMPLEMENTATION:

CLASS PrintSpooler ::= ...

CLASS Printer ::= ...

CLASS AcctLog ::= ...

...

END FEATURE PrintServer
```

Figure 2-2: Print Server Feature

MELD supports three granularities of concurrency [10]: multiple threads, message passing, and transactions. Multiple control threads within an object permit multiple methods to execute at the same time. When multiple methods accessing the same instance variables execute simultaneously, however, concurrency is maximized at the cost of nondeterminism. Atomic blocks solve this problem by enforcing critical sections with respect to the object.

```
obj: class; (* declaration of object *)
name: string; (* declaration, holds name of object *)

obj := class.get(name); (* gets handle on remote or persistent object *)
```

Figure 2-3: Referencing a Remote or Persistent Object

Objects provide medium grain concurrency. MELD supports both synchronous and asynchronous message passing among local or remote objects. The "receiver.message" notation is used for sending messages synchronously, where the message might return a result, and the "send message to receiver" notation is used for sending messages asynchronously. An object can wait for a particular message using the "delayuntil message from sender", where the "from sender" part is optional. The distinction between local and remote receiver and sender objects is transparent at the point of message passing, but remote objects must be treated specially to first obtain references to the objects. In particular, the same sequence of declarations and code used to reference persistent objects is also used for remote (either transient or

persistent) objects, as shown in Figure 2-3.

Transactions provide large grain concurrency. Atomic blocks, serializable transactions and commit-serializable transactions are discussed in the following three sections.

3. Atomic Blocks

An *atomic block* is a list of statements executed atomically with respect to the current object. The atomic block may be the top-level of a method or nested within a method. Nesting an atomic block within another atomic block is permitted, but meaningless.

```
CLASS C ::= instance variables

METHODS: constraints

    M (arguments) -->
    ( body )

    N (arguments) -->
    ( body )

    O (arguments) -->
    [ body ]

    P (arguments) -->
    [ body ]

END CLASS C
```

Figure 3-1: Atomic Block

Consider the example in Figure 3-1. Atomic blocks are indicated by parentheses "(...)" and sequential blocks (conventional compound statements) by square brackets "[...]" (alternatively, keywords such as "begin atomic ... end atomic" and "begin ... end" could be used, respectively). M and N are both atomic, while O and P are non-atomic sequential blocks. If an object of class C first receives a message O and later while O is still executing receives a message P, the newly invoked method P begins execution even though O has not completed. The two methods can arbitrarily read and update the same instance variables in arbitrary order determined by the race conditions. However, if an object of class C first receives a message invoking M and later while M is still executing receives a message invoking N (or O), the newly invoked method has to wait until M completes. If instead the object first receives a message O and later while O is still executing receives a message M, then O's sequential block is suspended while M completes its

atomic operation, and then O is resumed.

```
CLASS PCQueue ::=
    Q: array[1..N] of T;
    P, C: integer := 0;
    Used: integer := 0;

METHODS:

    Produce(X: T) -->
    if (Used = N) then [
        delay until SpaceAvailable();
        $self.Produce(X);
    ] else (
        P := P % N + 1;
        Q[P] := X;
        Used := Used + 1;
        send EntryAvailable() to $self;
    )

    Consume(): T -->
    if (Used = 0) then [
        delay until EntryAvailable();
        return ($self.Consume());
    ] else (
        C := C % N + 1;
        Used := Used - 1;
        send SpaceAvailable() to $self;
        return (Q[C]);
    )
END CLASS PCQueue
```

Figure 3-2: Producer/Consumer Queue

A non-atomic block can be arbitrarily corrupted. There is no notion of consistency with respect to non-atomic blocks that can be detected or enforced by the system. An atomic block cannot be corrupted. It exclusively locks the entire object for its duration. However, it is possible for atomic blocks to view the partial results of non-atomic blocks. An alternative semantics would prevent atomic blocks from viewing the partial results of other non-atomic code as well as preventing other code from viewing partial results of atomic blocks. This does not seem necessary, since if the application required consistency of data updated by the non-atomic code, the programmer should have written it as an atomic block. Figure 3-2 illustrates using atomic blocks for the classical producer/consumer problem.

Atomic blocks have one serious flaw. Consider the example in Figure 3-3, where R is an atomic block but P is not, and R invokes P synchronously. The programmer presumably expects

```
CLASS E ::= c: integer;

METHODS:

    P (a, b: integer): integer -->
    [ c := a + b; return c; ]

    Q (x: integer) -->
    [ c := 2 * x; ]

END CLASS E

CLASS F ::= d: E;
            e: integer;

METHODS:

    R () --> ( e := d.P(1, 2); )

END CLASS F
```

Figure 3-3: Problem with Atomic Blocks

that e will be assigned to 3, but this need not be the case. Consider the scenario where Q is invoked immediately after "c := a + b" executes, but before "return c" executes. The atomicity of a method R does not imply anything about the other methods it invokes!

The problem is that an atomic block is a critical section only with respect to the current object. It locks only that object, not any of the objects to which it sends messages. There is no notion of complex objects in MELD, in the sense that locking an object does not have the effect of locking component objects. If complex objects were added, then messages sent to any objects reachable *via* instance variable links from the original would still be effectively part of the same atomic block. But that would not solve the problem since MELD atomic blocks could still send messages to method arguments and global objects.

4. Serializable Transactions

Transaction blocks solve this problem with atomic blocks. A transaction block is a list of statements executed as a serializable transaction with respect to all other code ever executed by the MELD system. A transaction block may be an entire method or nested within a method. Nesting a transaction block within another transaction block permits nested transactions [18].

Consider the example in Figure 4-1, where R is now a transaction block rather than an atomic

```
CLASS E ::= c: integer;

METHODS:
    P (a, b: integer): integer -->
    [ c := a + b; return c; ]

    Q (x: integer) -->
    [ c := 2 * x; ]

END CLASS E

CLASS F ::= d: E;
            e: integer;

METHODS:

    R () --> << e := d.P(1, 2); >>

END CLASS F
```

Figure 4-1: Transaction Block

block. Transaction blocks appear between double angle brackets "<<>>", although begin transaction, end transaction keywords could be substituted; we do not do so for sequential and atomic blocks as well as transaction blocks, even though we realize the examples would be much easier to read, to make clear the distinction between transaction blocks and the free-form Create and Commit operations described later. P and Q are still sequential blocks. R invokes P synchronously, and Q is invoked immediately after "c := a + b" executes, but before "return c". In this case, P is considered part of the R transaction and cannot be corrupted, so e is guaranteed to be set to 3.

Transaction blocks are implemented in MELD using a distributed optimistic concurrency control mechanism based on multiple versions developed by Agrawal *et al.* [2]. This scheme permits read-only transactions to commit with no concurrency control overhead. Initiating a transaction creates a "coordinator" object, a remote "cohort" object is created each time a thread from this transaction first executes on another machine to keep track of the ReadSet and WriteSet of the transaction with respect to that machine, and a two-phase commit protocol is employed. In the example above, serializability would be enforced by rolling back the transaction initiated by R and restarting it after Q terminated. This scheme can unfortunately lead to starvation, since repeated calls to Q can effectively prevent R from ever committing.

This is not quite as severe a problem as it may seem, because the serializability is enforced at

```
CLASS E ::= c: integer;
            d: integer;

METHODS:

    P (a, b: integer): integer -->
    [ c := a + b; return c; ]

    Q (x: integer) -->
    [ d := 2 * x; ]

END CLASS E

CLASS F ::= d: E; e: integer;

METHODS:

    R () --> << e := d.P(1, 2); >>

END CLASS F
```

Figure 4-2: Enforcing Serializability On Instance Variables

the granularity of individual instance variables rather than entire objects. Figure 4-2 shows essentially the same example, but here Q updates a new instance variable d rather than the conflicting instance variable c. In this case, Q and R can proceed concurrently with no conflict and R commits.

```
FEATURE Bank
INTERFACE: ...
IMPLEMENTATION:

CLASS teller ::=
  account : SavingsAccount;

METHODS:

(* Constraints *)
     account := SavingsAccount.getObjectIdName(accountName);
     if (account = nil) then
        send "Account Not Available!!" to $stdout;

(* Methods                          *)
(* << ... >> Transaction Blocks *)

"withdraw %d, %s"(cash: Integer, accountName: String)-->
     <<
       if(account <> nil) then
       [
         send withdraw(cash) to account;
         send "please wait, your transaction is being processed" to $stdout;
```

```
        ]
      >>

"deposit %d, %s"(cash: Integer, accountName: String)-->
      <<
        if(account <> nil) then
        [
          send deposit(cash) to account;
          send "please wait, your transaction is being processed" to $stdout;
        ]
      >>
END CLASS teller

PERSISTENT CLASS SavingsAccount ::=
  balance : Integer := 0;

METHODS:

{* Constraints *}
send "the balance is %d"(balance) to $stdout;

{* Methods                                                          *}
{*  every persistent class inherits the following methods:          *}
{*          getObjectIdByName(ss: String);                          *}
{*          getNameByObjectId(objid: Object);                       *}
{*          .....                                                   *}

deposit(cash: Integer)-->
    <<
       balance := balance + cash;
    >>

withdraw(cash: Integer)-->
    <<
       if(cash > balance) then send "Insufficient balance" to $stdout;
       else balance := balance - cash;
    >>
END CLASS SavingsAccount
END FEATURE Bank
```

Figure 4-3: Bank Feature with Transaction Blocks

The example in Figure 4-3 uses persistent classes and I/O messages, not discussed here, as well as transaction blocks. Although multiple methods can run within a particular `SavingsAccount` object, the transaction block accesses to `balance` are serialized.

As a supplement to the transaction block, MELD provides the abort statement, typically used with a conditional. Execution of an abort statement immediately forces rollback (without restart) of the enclosing transaction block under program control. The abort statement may appear in this block or in some other method invoked *via* synchronous or asynchronous message passing.

If the transaction block is nested within another, then only the immediately enclosing block is aborted.

Transaction blocks are rather limited, in that the transaction must begin and end in the same method. Although this requirement generally supports good programming practice, there are cases where the necessary programming may become awkward. It is desirable to add Create and Commit operations, and allow the Commit operation as well as Abort to appear in different methods than the matching Create operation.

```
FEATURE Bank
INTERFACE:
  IMPORTS Transaction;

IMPLEMENTATION:

CLASS teller ::=
  account : SavingsAccount;

METHODS:

(* Constraints *)
     account := SavingsAccount.getObjectIdName(accountName);
     if (account = nil) then
         send "Account Not Available!!" to $stdout;

(* Methods                          *)

"withdraw %d, %s"(cash: Integer, accountName: String)-->
(      t : Transaction;

       t := Transaction.Create();
       if(account <> nil) then
       (
         send withdraw(cash, t) to account;
         send "please wait, your transaction is being processed" to $stdout;
       )
       else send Commit to t;
)

"deposit %d, %s"(cash: Integer, accountName: String)-->
(      t : Transaction;

       t := Transaction.Create();
       if(account <> nil) then
       (
         send deposit(cash, t) to account;
         send "please wait, your transaction is being processed" to $stdout;
       )
       else send Commit to t;
)
END CLASS teller
```

```
PERSISTENT CLASS SavingsAccount ::=
  balance : Integer := 0;
METHODS:
(* Constraints *)
send "the balance is %d"(balance) to $stdout;
(* Methods *)

deposit(cash: Integer, td: Transaction)-->
[
       balance := balance + cash;
       send Commit to td;
]

withdraw(cash: Integer, td: Transaction)-->
[
       if(cash > balance) then send "Insufficient balance" to $stdout;
       else balance := balance - cash;
       send Commit to td;
]
END CLASS SavingsAccount
END FEATURE Bank
```

Figure 4-4: Bank Feature with Transaction Objects

In MELD we do this by representing transactions as objects. `Create`, `Commit` and `Abort` are treated as normal messages, where `Create` is sent to the `Transaction` class. Figure 4-4 shows the same example as for transaction blocks, but now transaction objects are used.

One problem with our current design lies in the multi-threaded nature of MELD, since an arbitrary number of threads may operate as part of the same transaction (even without nesting). Our implementation of the abort operation allows auxiliary threads to continue execution until they are ready to terminate, and then rolls back their results if the corresponding transaction has already aborted. This is clearly non-optimal from a performance viewpoint, but it has the fewest complications.

Adding the commit operation would require tracking down all the auxiliary threads and waiting until all of them are ready to terminate. If one or more threads executed abort operations, the transaction would be aborted even though a commit operation had previously been executed on behalf of the same transaction. We do not currently synchronize the threads associated with the same transaction; in particular, serializability is not enforced with respect to such threads unless they are explicitly separated into subtransactions.

5. Commit-Serializable Transactions

Commit-serializability [21] is an extended transaction model we have developed for *open-ended activities*, such as CAD/CAM, VLSI design, office automation and software development. The name "commit-serializability" reflects that our model requires committed transactions to be serializable but permits transactions to divide and merge in ways such that the committed transactions may not bear any simple relationship to the initiated transactions.

Open-ended activities are characterized by long duration, uncertain developments, and interactions with concurrent activities. Consider, for example, our archetypical open-ended activity, software development. A software development environment might enclose within a single transaction all activities responding to a modification request. These activities — including browsing and editing perhaps overlapping sets of source files, compiling and linking, executing test cases and generating traces, *etc.* — could take days or weeks and require modifying substantial portions of the system. Existing software development tools provide some of the needed facilities: serialized access to individual files and the creation of parallel versions, checkpointing, system build, and undo/redo. The crippling problem is that these mechanisms operate on individual files, rather than on the complete set of resources updated during the activity so consistency cannot be guaranteed. A few environments do publish sets of resources as a unit but use *ad hoc* methods not yet developed into transaction models. On the other hand, serializable transactions are too restrictive, for instance:

- A programmer would be prevented from editing a file simply because another programmer had previously read the file but has not yet finished his programming transaction.
- Programmers would not be able to release certain resources — so that they can be accessed by other programmers cooperating to build the same subsystem — while continuing to use other resources that are part of the same activity.

Commit-serializability provides the advantages of a transaction model without the disadvantages of serializability. The model is supported by two new operations, Split and Join, in addition to the Create, Commit and Abort operations discussed in the previous section.

The Split operation divides an in-progress transaction into two new ones, each of which may later commit or abort independently of the other. Say a user U has read modules M, N and O and updated modules N and O. He has compiled the changed N and O, linked them together with the old object code for M, and is in the process of debugging. The C attribute of an object represents

Transaction U_1	Transaction U_2	Transaction V
initiate		
read(M.c)		
read(N.c)		
write(N.c)		
read(O.c)		
write(O.c)		
write(N.o)		
write(O.o)		initiate
read(M.o)		access(X)
read(N.o)		access(Y)
read(O.o)		access(Z)
		request read(N.c)
corresponding notify(N.c, read)		
split((M,O,resume), (N,commit))		
	initiate	
access(M)	commit(N)	
access(O)		actual read(N.c)
commit(M,O)		write(N.o)
		access(X)
		access(Y)
		access(Z)
		commit(N,X,Y,Z)

Figure 5-1: Example Split Schedule

its source code and the o attribute its object code. Another user V requests access to module N. Since U is done making changes to N but needs to continue work on M and O (the issues of how this is decided are discussed later), the transaction splits and commits a new transaction that updates N. V then reads N, decides to use this new version rather than the old one for testing his own changes to other modules, recompiles N and tests his subsystem. Later V commits N and U commits M and O. The corresponding schedule is depicted in Figure 5-1.

The inverse Join operation merges a completed transaction into an in-progress transaction. Say a user U has read modules M, N and O and updated modules N and O. He has compiled the changed N and O, linked them together with the old object code for M, and completed debugging. Another user V is working on other changes to the same subsystem. Since U is done, his transaction joins M, N and O to V's resources, so all changes to the subsystem will be published together. U then goes on to his next task, as part of a new transaction. The schedule for this example is displayed in Figure 5-2.

Transaction U_1	Transaction U_2	Transaction V
initiate		
read(M.c)		
read(N.c)		
write(N.c)		
read(O.c)		
write(O.c)		
write(N.o)		
write(O.o)		initiate
read(M.o)		access(X)
read(N.o)		access(Y)
read(O.o)		access(Z)
join(V)		
	initiate	access(M)
	access(P)	access(X)
	access(Q)	access(N)
	access(R)	access(Y)
	commit(P,Q,R)	access(O)
		access(Z)
		commit(M,N,O,X,Y,Z)

Figure 5-2: Example Join Schedule

MELD's Split and Join operations are illustrated in Figure 5-3. In order to later use the Split operation, the initiating Create must be associated with a handler as shown. A handler is not necessary to use Join, since the joined transaction can always ignore its new resources. Handlers are explained later on.

```
Split(A: ( AReadSet, AWriteSet, AMessage ),
      B: ( BReadSet, BWriteSet, BMessage ))

Join(S: TID)

TID := Transaction.Create(HandlerMessage)
```

Figure 5-3: Split and Join Operations

When the Split operation is invoked during a transaction T, there is a TReadSet consisting of all objects read by T but not updated and a TWriteSet consisting of all objects updated by T. TReadSet is divided into AReadSet and BReadSet, and TWriteSet into AWriteSet and BWriteSet. AMessage and BMessage are sent to `$self`, to indicate what to do next for each of

the transactions.

For example, transaction T1 has read objects M and N and updated objects N and O. Another transaction T2 requests access to object N. T1's request handler is invoked, and in this case the handler decides that the transaction is done making changes to N, but needs to continue work on M and O. The handler executes the Split operation and commits a transaction T3 that updates N. T2 then accesses N. Later T2 commits N and T1 commits M and O.

In the special case where AMessage is the Commit operation, objects in AWriteSet may also appear in either BReadSet or BWriteSet. Objects in AWriteSet can also appear in BWriteSet if A later commits before B. BReadSet need not be disjoint with AWriteSet, provided that A does not update any of these objects *after* the split, since B is serialized after A. This can be enforced by not allowing B to commit until after A does, and aborting B if A aborts.

The role of the handler is to determine the arguments for the Split operation. The HandlerMessage argument to the Create operation is a string that can be sent to an object in the same way as input messages; this is a subterfuge, since in MELD there is no other means for passing procedure parameters or referring to a method symbolically. The following restrictions apply in the case where the reason for the split was the request for some object by some other transactions, and B will immediately commit to make this object available. If the object has been updated during the prefix of T (its history up to now), AWriteSet must contain the requested object. If the request is to update the object, it must not be in BReadSet. If this object has only been read during T, then it must be in AReadSet and not in BWriteSet. This assumes that B does not keep any form of temporary copy of the object, or any value from which the object can be derived.

When the Join operation is invoked during a transaction T, target transaction S must be ongoing. TReadSet and TWriteSet are added to SReadSet and SWriteSet, respectively, and S may continue or commit. For example, transaction T1 has read objects M and N and updated objects N and O. Another transaction T2 is making other changes to a semantically related set of objects. When T1 is ready to commit, it executes Join to join M, N and O to T2's resources, so this set of objects is committed together.

6. Implementation Status

MELD is translated into C and runs on 4.2 and 4.3 Berkeley Unix, on Sun 3's, MicroVax II's and RT 125's. MELD's compiler (including a preprocessor that implements inheritance) consists of 400 lines of Lex input, 1500 lines of Yacc input, and 4000 lines of C code. The run-time environment including the Meld Debugger (MD) [6] has 250 lines of Lex and 650 lines of Yacc, for the Data Path Expression debugging language for specifying high-level concurrent events and the actions to take when such events are recognized, and 6000 lines of C.

Only atomic blocks have been fully implemented in the main-line MELD implementation, which supports a simple name service for sending messages to remote objects (MELD objects currently cannot migrate) and persistent objects using B-trees. Transaction blocks have been implemented in a diverged version, which does not include some of the language facilities added in the past year or so. The largest program attempted in MELD to date has been a toy implementation of "Small Prolog" (which never really worked, but this was not due to a flaw in MELD).

7. Related Work

Herlihy and Wing [1] describe a fine granularity correctness condition for COOPS, *linearizability*. Linearizability requires that each operation appear to "take effect" instantaneously and that the order of non-concurrent operations should be preserved. MELD atomic blocks in effect implement linearizability at the level of blocks, which may encompass an entire method.

Martin [16] describes small grain mechanisms for both *externally serializable* and *semantically verifiable* operations for COOPS. Externally serializable operations enforce serializability among top-level operations but permit non-serializable computations on subobjects. Semantically verifiable operations do not enforce serializability at all, but instead consider the semantics of apparently conflicting operations in preventing inconsistencies from being introduced. Weihl [27] describes a formalism analogous to semantically verifiable operations but restricted to commutative operations. He considers abstract data types, not specifically object-based programming.

Argus [13] has atomic and non-atomic *objects*, binding concurrency control to one level of implementation. In contrast, MELD allows "free-form" concurrency control at any level, and also

gives the programmer the freedom to combine atomic and non-atomic actions on the same object. Camelot [24] and Mach [7] together provide a distributed transaction facility for objects, and Avalon [4] provides some measure of language support as an extension of C++. The Avalon model of concurrency control was heavily influenced by Argus, and, like Argus, binds atomicity to the object rather than allowing it at any smaller level.

In Clouds [3], concurrency control is not bound to the object level, and atomic and non-atomic operations on an object may be mixed. Hybrid [20] 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 commits or aborts. MELD's atomic blocks work similarly, but on single objects only; MELD's transactions provide atomicity across multiple objects, but permit much more concurrency because serializability is enforced at the granularity of instance variables.

Most other COOPS provide only one form of concurrency control; for example, Coral3 [17] uses only two-phase locking, and GemStone [15] uses only an optimistic approach.

8. Conclusions

We have described our experimentation with three types of transaction-like facilities as part of the MELD programming language. Atomic blocks are easy to use, but are not sufficient for applications requiring consistency among multiple objects. Serializable transactions are somewhat more difficult to use, due to interactions with MELD's multiple threads. If asynchronously generated threads are confined to subtransactions, then programming is easier but overhead is increased and concurrency reduced. Commit-serializable transactions should be no more complicated than full serializable transactions except for one crucial point: the request handlers. It is not yet clear how these should be structured, what parameters they should be provided, or even exactly what they should do. We have designed a version of our commit-serializability model for transactions in the Marvel software development environment [11], but there we have taken the easy way out by presenting requests to the human users. This might be a viable option for other applications supporting open-ended activities.

Acknowledgments

David Garlan and the author jointly developed the original, non-concurrent design of MELD. Wenwey Hseush and Steve Popovich participated in the redesign for concurrency. Wenwey and Shyhtsun Felix Wu worked on atomic blocks, Steve and Felix on serializable transactions, and Calton Pu, Norm Hutchinson and the author jointly developed the semantics of commit-serializable transactions. The ideas discussed here were also influenced by discussions with Nasser Barghouti, Dan Duchamp, Brent Hailpern, Maurice Herlihy, Eliot Moss, Bob Schwanke, Soumitra Sengupta, Andrea Skarra, Peter Wegner, Bill Weihl and Stan Zdonik. Nasser and Steve provided extensive critical comments on a draft of this paper. Nicholas Christopher, Jeffrey Gononsky, Nanda S. Kirpekar, Marcelo Nobrega, David Staub, Seth Strump, Kok-Yung Tan, and Jun-Shik Whang contributed to the MELD implementation effort.

References

[1] Maurice P. Herlihy and Jeannette M. Wing.
Axioms for Concurrent Objects.
In *14th Annual ACM Symposium on Principles of Programming Languages*, pages 13-26. Munich, West Germany, January, 1987.

[2] D. Agrawal, A.J. Bernstein, P. Gupta and S. Sengupta.
Distributed Optimistic Concurrency Control with Reduced Rollback.
Journal of Distributed Computing 2(1):45, April, 1987.

[3] Partha Dasgupta, Richard J. Leblanc Jr. and William F. Appelbe.
The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work.
In *8th International Conference on Distributed Computing Systems*, pages 2-9. San Jose CA, June, 1988.

[4] David Detlefs, Maurice Herlihy and Jeannette Wing.
Inheritance of Synchronization and Recovery Properties in Avalon/C++.
Computer :57-69, December, 1988.

[5] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-632, November, 1976.

[6] Wenwey Hseush and Gail E. Kaiser.
Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language.
In *ACM SIGPlan/SIGOps Workshop on Parallel and Distributed Debugging*, pages 236-246. Madison WI, May, 1988.
Special issue of *SIGPlan Notices*, 24(1), January 1989.

[7] 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.

[8] Gail E. Kaiser and David Garlan.
Melding Software Systems from Reusable Building Blocks.
IEEE Software :17-24, July, 1987.

[9] 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.

[10] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush and Shyhtsun Felix Wu.
Melding Multiple Granularities of Parallelism.
In *European Conference on Object-Oriented Programming*. Nottingham, UK, July, 1989.
In press.

[11] Gail E. Kaiser.
A Marvelous Extended Transaction Processing Model.
In Gerhard Ritter (editor), *11th World Computer Conference IFIP Congress '89*. Elsevier Science Publishers B.V., San Francisco CA, August, 1989.
In press.

[12] Wm Leler.
Constraint Programming Languages Their Specification and Generation.
Addison-Wesley Pub. Co., Reading MA, 1988.

[13] 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.

[14] Yoelle S. Maarek and Gail E. Kaiser.
Using Conceptual Clustering for Classifying Reusable Ada Code.
In *Using Ada: ACM SIGAda International Conference*, pages 208-215. ACM Press, Boston MA, December, 1987.
Special issue of *Ada LETTERS*, December 1987.

[15] David Maier, Jacob Stein, Allen Otis, and Alan Purdy.
Development of an Object-Oriented DBMS.
In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 472-482. October, 1986.
Special issue of *SIGPLAN Notices*, 21(11), November 1986.

[16] Bruce E. Martin.
Modeling Concurrent Activities with Nested Objects.
In *7th International Conference on Distributed Computing Systems*, pages 432-439. West Berlin, West Germany, September, 1987.

[17] Thomas Merrow and Jane Laursen.
A Pragmatic System for Shared Persistent Objects.
In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 103-110. Orlando FL, October, 1987.
Special issue of *SIGPlan Notices*, 22(12), December 1987.

[18] J. Eliot B. Moss.
Nested Transactions and Reliable Distributed Computing.
In *2nd Symposium on Reliability in Distributed Software and Database Systems*, pages 33-39. IEEE Computer Society Press, Pittsburgh PA, July, 1982.

[19] Michael Lesk (editor).
Information Systems: Nested Transactions: An Approach to Reliable Distributed Computing.
The MIT Press, Cambridge MA, 1985.
PhD Thesis, MIT LCS TR-260, April 1981.

[20] 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.

[21] Calton Pu, Gail E. Kaiser and Norman Hutchinson.
Split-Transactions for Open-Ended Activities.
In *14th International Conference on Very Large Data Bases*, pages 26-37. Los Angeles CA, August, 1988.

[22] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian and Carrie Wilpolt.
An Introduction to Trellis/Owl.
In *Object-Oriented Systems, Languages, and Applications Conference*, pages 9-16. Portland, OR, September, 1986.
Special issue of *SIGPlan Notices*, 21(11), November 1986.

[23] Alan Snyder.
CommonObjects: An Overview.
In *Object-Oriented Programming Workshop*, pages 19-29. Yorktown Heights, NY, June, 1986.
Special issue of *SIGPlan Notices*, 21(10), October 1986.

[24] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, Dean S. Thompson.
The Camelot Project.
Database Engineering 9(4), December, 1986.

[25] John A. Stankovic.
Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems.
Computer 21(10):10-19, October, 1988.

[26] Mark J. Stefik, Daniel G. Bobrow and Kenneth M. Kahn.
Integrating Access-Oriented Programming into a Multiparadigm Environment.
IEEE Software 3(1):11-18, January, 1986.

[27] William E. Weihl.
Commutativity-Based Concurrency Control for Abstract Data Types (Preliminary Report).
In Bruce D. Shriver (editor), *21st Annual Hawaii International Conference on System Sciences*, pages 205-214. IEEE Computer Society Press, Kona, HI, January, 1988.

I. Example

This program, called "dining octopi", runs on a Sun 3 workstation. It is essentially a two-dimensional version of dining philosophers. There are eight forks and four "octopi", each with four arms (the other four arms — not shown — are used to sit on since it would be difficult for an octopus to use a chair). An octopus needs four forks in order to eat. Access to forks wraps around from right to left of the screen and from top to bottom. The octopi and forks are objects and all interactions are done by message passing. The graphics hacking is done in C, not shown; any C subroutine call is a legitimate MELD statement and simple variables can be shared between the MELD and C code.

```
FEATURE diners
INTERFACE:
IMPLEMENTATION:

OBJECT:
octopi: array[0..3] of octopus;
forks: array[0..7] of fork;
start: initialization := initialization.create;

CLASS octopus ::=
 left, right, up, down: fork; {* "forks" in various directions *}
 {* display information *}
 x, y: integer;
 index: integer;
METHODS:
 dine() --> [
  rightFirst, upFirst: boolean;
  $self.think();
  rightFirst := (rand() >> 16) % 2 == 1;
  upFirst := (rand() >> 16) % 2 == 1;
  if (rightFirst)
   then right.pickUp($self, true);
  else left.pickUp($self, true);
  if (rightFirst)
   then if (left.pickUp($self, false) == 0)
    then right.putDown(); {* couldn't get left fork *}
   else [
    if (upFirst)
     then if (up.pickUp($self, false) == 0)
      then [
       left.putDown(); {* couldn't get up fork *}
       right.putDown();
      ]
     else [
      if (down.pickUp($self, false) == 0)
       then [
        left.putDown(); {* couldn't get down fork *}
        right.putDown();
        up.putDown();
       ]
      else [
```

```
      $self.eat(); {* got all forks *}
      left.putDown();
      right.putDown();
      up.putDown();
      down.putDown();
     ]
    ]
   else [
    if (down.pickUp($self, false) == 0)
     then [
      left.putDown(); {* couldn't get down fork *}
      right.putDown();
     ]
    else [
     if (up.pickUp($self, false) == 0)
      then [
       left.putDown(); {* couldn't get up fork *}
       right.putDown();
       down.putDown();
      ]
     else [
      $self.eat(); {* got all forks *}
      left.putDown();
      right.putDown();
      up.putDown();
      down.putDown();
     ]
    ]
   ]
  ]
 else if (right.pickUp($self, false) == 0)
  then left.putDown(); {* couldn't get right fork *}
  else [
   if (upFirst)
    then if (up.pickUp($self, false) == 0)
     then [
      left.putDown(); {* couldn't get up fork *}
      right.putDown();
     ]
    else [
     g_pickup_up(index);
     if (down.pickUp($self, false) == 0)
      then [
       left.putDown(); {* couldn't get down fork *}
       right.putDown();
       up.putDown();
      ]
     else [
      $self.eat(); {* got all forks *}
      left.putDown();
      right.putDown();
      up.putDown();
      down.putDown();
     ]
    ]
   else [
    if (down.pickUp($self, false) == 0)
```

```
        then [
         left.putDown(); (* couldn't get down fork *)
         right.putDown();
        ]
       else [
        if (up.pickUp($self, false) == 0)
         then [
          left.putDown(); (* couldn't get up fork *)
          right.putDown();
          down.putDown();
         ]
        else [
         $self.eat(); (* got all forks *)
         left.putDown();
         right.putDown();
         up.putDown();
         down.putDown();
        ]
       ]
      ]
     ]
    send dine() to $self; (* loop, whether successful or not *)
   ]
   (* some methods contained only graphics code *)
   think() --> [
    return(0);
   ]
   eat() --> [
    return(0);
   ]
   init(myIndex: integer) --> [
    upI, downI, rightI, leftI: integer;
    (* init screen display *)
    x := myIndex % 2;
    y := (myIndex / 2) * 2;
    leftI := y * 2 + (x + 1) % 2;
    left := forks[leftI];
    rightI := y * 2 + x % 2;
    right := forks[rightI];
    upI := ((y + 3) % 4) * 2 + x;
    up := forks[upI];
    downI := (y + 1) * 2 + x;
    down := forks[downI];
    $self.setIndex(myIndex);
    send dine() to $self; (* main program for octopus *)
   ]
   setIndex(myIndex:integer) -->
   [
    index := myIndex;
   ]
END CLASS octopus

CLASS fork ::=
 whoHas: octopus := nil; (* octopus that has fork *)
METHODS:
 pickUp(who: octopus; waitFor: boolean) --> [
  willWait: boolean := false;
```

```
    (
     if (whoHas != nil)
      then if (waitFor)
       then willWait := true;
      else return(0);
     else [
      {* fork is free, go ahead and pick it up *}
      whoHas := who;
      return(1);
     ]
    )
    if (willWait)
     then return($self.pickUp(who, waitFor));
  ]
  putDown() --> [
   whoHas := nil;
  ]
END CLASS fork

CLASS initialization ::=
METHODS:
   send startMe(0) to $self;
   startMe(index: integer) --> [
    forks[0] := fork.create("fork 0");
    forks[1] := fork.create("fork 1");
    forks[2] := fork.create("fork 2");
    forks[3] := fork.create("fork 3");
    forks[4] := fork.create("fork 4");
    forks[5] := fork.create("fork 5");
    forks[6] := fork.create("fork 6");
    forks[7] := fork.create("fork 7");
    octopi[0] := octopus.create("octopus 0");
    send init(0) to octopi[0];
    octopi[1] := octopus.create("octopus 1");
    send init(1) to octopi[1];
    octopi[2] := octopus.create("octopus 2");
    send init(2) to octopi[2];
    octopi[3] := octopus.create("octopus 3");
    send init(3) to octopi[3];
   ]
END CLASS initialization
END FEATURE diners
```