The execution of this transaction block creates a new local view with its local scheduler (Figure 6). The object which invoked the transaction, in this case O, is put in the local view. Then, the local scheduler of the subtransaction schedules O, the statements inside the transaction block are executed and two messages are sent to O_1 and O_2 asynchronously. The destination objects of the messages are copied into the subtransaction's local view and their OIDs are inserted into the read set (Figure 7). In the presence of asynchronous message passings, the parent transaction and the subtransaction could access the same objects concurrently, violating serializability. To avoid such situation, the objects in the parent's view are made inaccessible by parent transaction while they are accessed by its subtransactions.

Now O has completed the execution of all the statements inside the transaction block. However, the local scheduler finds other schedulable objects (i.e., O_1 and O_2) in the local view and the execution of the transaction does not completes (Section 3.1). Suppose O_2 is scheduled first. We assume that the message `[Forward` O_3`]` has no side effect on O_2 but sends an `Update` message to O_3. The object O_3 is copied from the parent transaction's view to the subtransaction's view, the OID of O_3 is inserted into the read set, and the `Update` message is passed to the copy of O_3.

Then O_1 and O_3 are scheduled (the order is irrelevant to the result). O_1 and O_3 are modified their internal state by the `Update` messages and their OIDs are inserted into the write set of the subtransaction's view (Figure 9). Now the local scheduler finds no schedulable object in the local view and it detects the termination of the transaction execution.

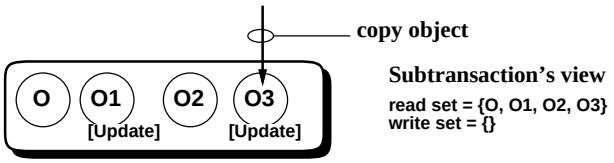

Figure 8: Schedule O_2

Figure 9: Schedule O_1 and O_3

Next, the transaction enters the serializability validation phase. Serializability is checked by comparing the read set of the subtransaction with the write sets of already committed sibling subtransactions. If the same object is shared by them, the validation fails; otherwise it succeeds. Note that read sets and write sets do not contain total objects nor ordinary objects, and hence validation is not performed on them.

Finally, the transaction enters the commit phase. If the validation succeeds, the transaction commits; objects (both total atomic and total objects) that are updated are copied into the parent's view and the elements in the write set are included in the write set of the parent's view (Figure 10). Otherwise, the subtransaction aborts; the subtransaction's view is cleared and no visible effect of its execution remains.

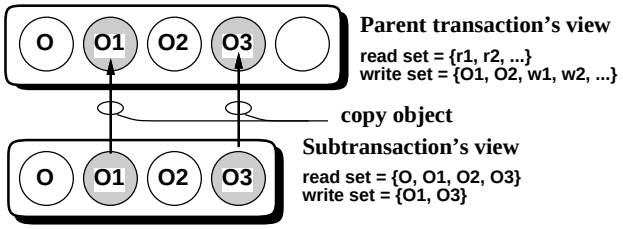

Figure 10: Commit transaction

[NZ90] Marian H. Nodine and Stanley B. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In *Proceedings of the 16th VLDB Conference*, pages 83–94, Brisbane, Australia, 1990. ACM SIGMOD.

[Wak91] K. Wakita. A synchronization scheme using transactions in object-oriented concurrent programming languages. Research report, February 1991.

[Weg87] P. Wegner. Dimensions of object-oriented language design. In *Proceedings of the 1987 Conference on OOPSLA, SIGPLAN Notices*, volume 22(12), pages 168–182, Orlando, Florida, October 1987.

[Yon90] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* MIT Press, Cambridge, Mass., 1990.

Appendix: Optimistic Nested Concurrency Control Algorithm

Although our transaction facilities are independent of a particular runtime algorithm for concurrency control, an efficient algorithm applicable to a wide range of distributed applications is nevertheless desirable. In this appendix, we present a nested concurrency control algorithm based on the optimistic approach [KR81] tailored for wide-range applicability in OOCP. We illustrate the algorithm with an example transaction session. The complete description of the algorithm appears in [Wak91].

The heart of the algorithm is the treatment of asynchronous message passings: detection of the transaction termination and avoidance of concurrent accesses by the parent transaction and its subtransactions. The *local view* of each transaction plays an important role in our algorithm. It is a set of objects that are accessed during the transaction execution. Objects in a local view is scheduled by a transaction specific *local scheduler*. Besides scheduling objects in the local view, its important task is to detect the termination of the transaction execution. To check serializability, the runtime system maintains a read set and a write set. A read set is a set of total atomic objects that have been sent messages and a write set is a set of total atomic objects that have been modified during the transaction execution.

The algorithm is divided into three phases: the *transaction execution phase*, the *validation phase*, and the *commit phase*. In the transaction execution phase, the transaction is executed preparing updates to the parent transaction's view in its local view. In the validation phase, serializability check is performed with respect to the previously committed sibling subtransactions, and if it succeeds, the transaction commits in the commit phase.

Suppose the following transaction block is executed at an object O. It asynchronously sends two messages Update and Forward to O_1 and O_2, respectively. The Update message modifies the state of O_1. Upon receiving the Forward message, O_2 sends Update message to O_3, asynchronously.

```
Transaction
  begin O1 <= [Update v];
        O2 <= [Forward O3]
  end
```

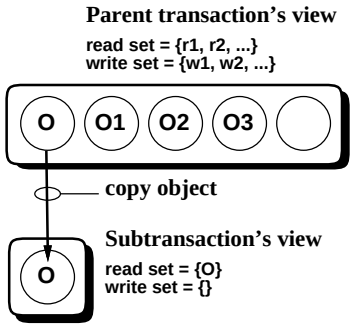

Figure 6: Start transaction

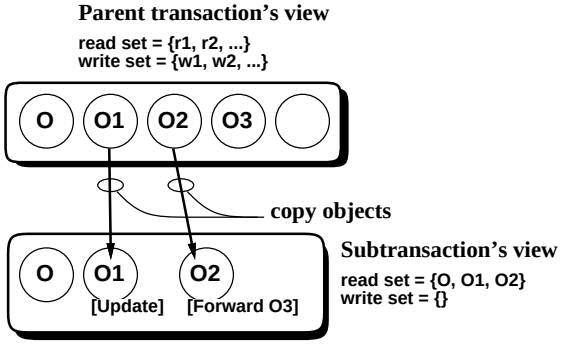

Figure 7: Send messages

To make such facilities available to the programmer, we mainly introduced two linguistic constructs called *transaction blocks* for nested transactions and *indirect instance variable declarations* for fine-grained control of concurrency. Furthermore, declarations of object atomicity level (total atomic, total, and ordinary) allow the programmer to customize concurrency control mechanisms.

A prototype implementation of our language HARMONY which realizes the transaction facilities proposed in this paper is now being completed.

Acknowledgments

We would like to thank Satoshi Matsuoka for his helpful discussions and comments on many versions of this paper. The first author is also grateful to Professor Takashi Masuda for his generous supports.

References

[Agh90] G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, 1990.

[Ame87] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, Cambridge, Mass., 1987.

[B+90] Beard et al. A visual calendar for scheduling group meetings. In *Proceeding of the Conference on Computer-Supported Cooperative Work*, pages 279–290, Los Angeles, California, October 1990. ACM SIGCHI and SIGOIS.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM computing surveys*, 21(3):261–322, September 1989.

[DHW88] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, 21(12):57–69, 1988.

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, SIGMOD RECORD*, volume 18(2), pages 399–407, Portland, Oregon, June 1989.

[Hew77] C. E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Inteligence*, 8(3):323–364, June 1977.

[Hew86] C. E. Hewitt. Offices are open systems. *ACM Transactions on Office Information Systems*, 4(3):271–287, July 1986.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981.

[Lis88] B. Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3):300–312, 1988.

[LW85] R. J. LeBlanc and C. T. Wilkes. Systems programming with objects and actions. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 132–138, Berlin, West Germany, 1985. IEEE.

[Mos85] J. E. B. Moss. *Nested Transaction: An Approach to Reliable Distributed Computing.* The MIT Press, Cambridge, Massachusetts, 1985.

[MR90] B. Martin and K. Ramamritham, editors. *Workshop on Transaction and Objects*, Ottawa, Canada, October 1990.

[Nie87] O. M. Nierstrasz. Active objects in Hybrid. In *Proceedings of the 1987 Conference on OOPSLA, SIGPLAN Notices*, volume 22(12), pages 243–253, Orland, Florida, December 1987.

Table 1: Conflicting operation table for a bounded buffer

	Put	Get
Put	Conflict	OK
Get	OK	Conflict

The above scheme is not specific to the bounded buffer. In general, data type specific concurrency control can be derived from a *conflicting operation table* such as Table 1. The conflicting operation table is implemented in terms of a set of indirect instance variables that are total atomic. In the example of Buffer_Revised, two total atomic indirect instance variables p and g are introduced to detect the Put/Put conflict and the Get/Get conflict, respectively.

6 Related Works

Several efforts have been made to incorporate atomic transaction facilities in concurrent programming languages (for example, Argus [Lis88], Aeolus [LW85], and Avalon/C++ [DHW88]). Their approaches are all based on RPC. In contrast our approach is based on non-RPC protocols.

Among them, in Avalon/C++, the inheritance hierarchy is accommodated to support both the level of atomicity and the customization of concurrency control algorithm — (1) by choosing an appropriate class for superclass, the level of atomicity are specified for all the instances of a new class and (2) by redefining two hook methods commit and abort provided in the subatomic class and handling short term locks properly, the programmer can add a class specific concurrency control protocol. The problem in this approach is its dependence on the inheritance hierarchy and the underlying implementation of the concurrency control algorithm. In contrast, our approach does not use the inheritance hierarchy for these purposes. Instead, the level of atomicity are specified on a per-object basis for customization. Therefore, our approach does not depend on the inheritance hierarchy nor the implementations of concurrency control mechanism thus achieves better encapsulation.

Recently, for the development of cooperative design applications, several models of *cooperative transactions* have been proposed to relax the restriction of serializability. [NZ90] defines a cooperative transaction hierarchy and allows the programmer to specify the constraints on the access patterns to objects shared by multiple transactions in a *transaction group*. For the development of real-time groupwares, [EG89] proposes the *transformation matrix* which is used to resolve the conflicting operation sequences in a distributed environment. These approaches are very flexible in expressing cooperation among transactions. (The approach of [EG89] does not seem to deal with nested transactions.) And these approaches specify permissible operation sequences based on the observation of the external operations performed on shared objects. In contrast, we restrict the description of constraints to contain only the local information of the atomicity levels of the subcomponents.

7 Concluding Remarks

In this paper, we have presented novel transaction facilities and their linguistic constructs which are powerful for modeling and implementing distributed organizational information systems in object-oriented concurrent languages. Our transaction facilities are characterized by:

- Accommodation of nested transactions to non-RPC message passing protocols
- An explicit way of grouping multiple message passings into a single transaction
- Object-wise (class independent) specification of atomicity level
- Customizability of concurrency control mechanism and its independence of underlying concurrency control algorithms of the runtime system

```
class Buffer_Split superclass Object
    var size;
    indirectvar in : unstructured, out : unstructured, vec : vector;
  begin
    method init(n)
      begin size := n; in := 0; out := 0; vec := new_vector(size) end;
    method Put(item) when (in < out + size)
      begin vec[in mod size] := item; in := in + 1 end;
    method Get(item) when (in > out)
      begin !vec[out mod size]; out := out + 1 end
  end
```

In this implementation, the first two requirements are guaranteed by the guard expressions[7] attached to the methods and the atomic property of the buffer object. However, the third requirement is not satisfied because this implementation prohibits the concurrent execution of a transaction, T_1, containing the invocation of Put method and another transaction, T_2, containing the Get method. The reason is: T_1 attempts to update the value of the container object **IN**[8] and T_2 attempts to update the value of the container object **OUT**. An important point here is that both transactions read the values of **IN** and **OUT** in their guards in order to check the invocation condition of Put and Get, causing a read/write conflict thereby requiring one of the transactions to abort.

Our approach to customization of concurrency control is to use the level of atomicity through the specification of object category. We use the indirect variable declaration to specify the category of objects that implement the indirect variables:

indirectvar $\langle variable_1 \rangle : \langle type_1 \rangle : \langle category_1 \rangle$, $\vdots$, $\langle variable_m \rangle : \langle type_m \rangle : \langle category_m \rangle$;

where, $\langle category_i \rangle$ specifies the category of the container which represents the indirect instance variable. In the definition of the Buffer_Revised class below, indirect instance variables *in*, *out*, and *vec* are declared as total; as a result, serializability validation for these objects is omitted (Figure 5). Instead, serializability validation is performed on new indirect instance variables *p* and *g*, which are declared as total atomic — they are touched in the body of Put and Get, respectively. Note that *p* and *g* represents the conflicting invocation of Put and Get by multiple transactions.

The guard expressions in the Buffer_Revised example ensure the first requirement for a concurrent buffer. The second requirement is guaranteed because concurrent invocations of Put (or Get) cause a write/write conflict at the Container object **P** (or **G**). Buffer_Revised also satisfies the third requirement, because **P** and **G** are the only total atomic objects that comprise the buffer, and concurrent invocations of Put and Get do not cause conflicts with respect to serializability.

```
class Buffer_Revised superclass Object
    var size;
    indirectvar in : unstructured : total, out : unstructured : total,
          vec : vector : total,
          p : unstructured : total atomic, g : unstructured : total atomic;
  begin
    method init(n)
      begin size := n; in := 0; out := 0; vec := new_vector(size) end;
    method Put(item) when (in < out + size)
      begin vec[in mod size] := item; in := in + 1;
            p := any value      { Touch the indirect instance variable p }
      end;
    method Get(item) when (in > out)
      begin !vec[out mod size]; out := out + 1;
            g := any value      { Touch the indirect instance variable g }
      end
  end
```

[7] The `when` (*predicate*) attached to a method declares that the method is invoked only when the predicate evaluates to true.

[8] **IN**, **OUT**, **P**, and **G** in the following discussion refer to the objects in the figures 4 and 5.

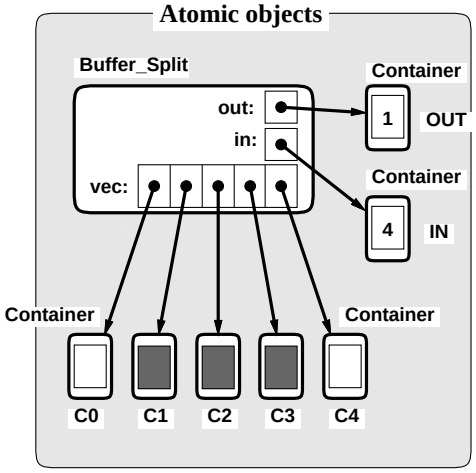

Figure 4: A naive implementation of a buffer

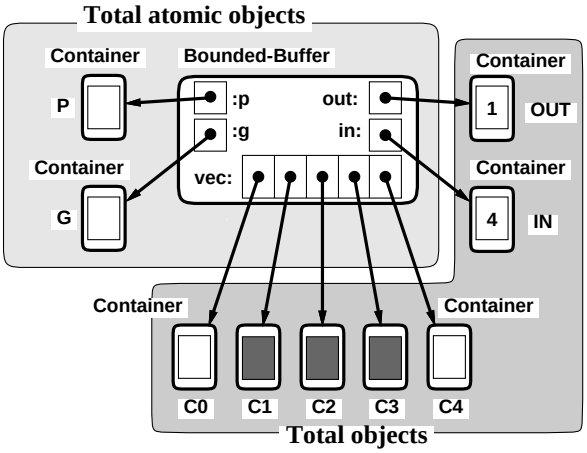

Figure 5: A Concurrent Buffer

wise correct operation sequences. We often like to weaken the correctness condition by customizing concurrency control algorithms that are performed at runtime.

One possible approach would be to make some functionalities (such as commitment and abort) of the runtime system *user definable* as hook functions. The problem of this approach is that it is much dependent on the runtime concurrency control algorithm such as the two-phase lock protocol. Instead, our approach to customization of concurrency control is object splitting technique (explained in Section 3.2) supplemented with the category specification, which is independent of underlying concurrency control algorithms. Three benefits of the algorithm independence are: 1) the programmer can customize concurrency control mechanisms without the knowledge of implementation details of runtime systems, 2) a single program runs on different implementations of *the same language*, and as a result, the user of the applications can choose one that best fits to the application and the architecture, 3) concurrency control algorithms of runtime systems can be changed or tuned without affecting the application programs.

Let us illustrate the problem of the usual serializability validation using an example of defining a highly concurrent bounded FIFO buffer, which must satisfy the requirements:

1. Put (Get) method should not be invoked when the buffer is full (empty).
2. The execution of two transactions, each of which contains an invocation of Put (Get) method, cannot overlap.
3. The execution of a transaction containing an invocation of Put method and another transaction containing an invocation of Get method may overlap.

A naive implementation using the object splitting technique is given below and its structure is illustrated in Figure 4. The data is stored in the Container objects whose OIDs are stored in *vec*. Two indirect instance variables *in* and *out* count the total number of items inserted and removed; the number of items in the buffer can be calculated by `(in - out)`. They also serve as indices to the *vec*. Namely, `(in mod size)` is the location where the next item should be inserted by Put and `(out mod size)` is the location from which the next invocation of Get removes an item.

```
class Observer superclass Person
  begin
    method negotiate(manager)
        var time;
      begin Transaction
              begin manager <== [remove_me, Me];
                    for time := 0 to 23 do
                          if (timetable[time] = manager) then
                                timetable[time] := nil
              end
      end
  end
```

5 Transaction Facilities (Continued)

This section presents two advanced features of our transaction facilities. They are used to reduce the runtime overhead of concurrency control mechanisms for serializability validation and also increase concurrency by customization of concurrency control mechanisms.

5.1 Object-wise Specification of Atomicity Level

In the previous approaches, every object is usually ensured atomicity and concurrency control mechanisms of runtime systems allow only serializable operation sequences on an object. We think this is often unnecessarily inefficient. For example, objects that are temporarily created during a transaction and are not shared by its subtransactions should not be subject to concurrency control. Thus in our approach, we do not impose concurrency control on all the objects. Rather, we support three categories of objects: *total atomic objects*, *total objects*, and *ordinary objects*, to specify the level of atomicity at a per-object basis.

- **Total atomic object:** Both totality and atomicity are ensured on operations performed on total atomic objects. Updates made on them during a transaction execution are localized to the transaction's view, and upon its commitment the updates are reflected to the parent transaction's view. Atomicity guarantees that only serializable operation sequences can be performed on total atomic objects.
- **Total object:** Only totality is ensured on total objects, and atomicity is not ensured. The consequence is that operations performed on total objects may not be serializable and hence in the presence of concurrency, non-serializable operation sequences may introduce inconsistency such as lost operations, non-reproducible reads, etc. This unique, seemingly unusual feature of total objects is introduced for the purpose of the *concurrency control customization* we will discuss in the next subsection.
- **Ordinary object:** Ordinary objects are unaffected by concurrency control, and hence exhibit the maximum execution efficiency due to the lack of system overhead. During transaction execution, **message passing to ordinary objects that are not created by the transaction is prohibited**; if this rule is violated, the system signals an error and the transaction aborts.

The category of each object is specified statically at its creation by the instance creation form. The general form of instance creation statement is as follows:

create_instance ⟨*class name*⟩(⟨*arguments*⟩) : ⟨*category*⟩;

When a category is not specified, the category of the new instance becomes total atomic by default. We refer to our approach as the *object-wise atomicity specification* (Comparison with the *class-wise atomicity specification* of Avalon/C++ will be discussed in Section 6).

5.2 Customization of Concurrency Control

Serializability is a clean and well-understood correctness condition to maintain consistency of distributed systems. However, it is sometimes too restrictive in that they do not permit commitment of other-

```
{ code for deciding the time of the meeting }
Transaction
  begin manager := create_instance Meeting_manager(start, end, organizer
                                                   participants title);
      foreach person in participants do    { announce the meeting }
          person <= [new_meeting start, end, manager];
      for time := start to end do    { fill in the electric reservation board }
          if ((rsv_board <== [Ref time]) <> nil) then
              rsv_board <= [Set time, manager]
          else abort end    { abort the transaction }
  end
```

The following is the definition of the Meeting_Manager class which holds the information of the meeting. We could add more methods and give other useful functionalities to the Meeting_Manager class such as report the topics of the meeting, distribute documents to participants, forward comments to the organizer, etc.

```
class Meeting_Manager superclass Object;
    var start, end, organizer, participants, title;
  begin
    method Initialize(s, e, o, m, t)
      begin start := s; end := e; organizer := o;
            participants := m; title := t end;
    method Remove_me(person)
      begin remove_from_set(participants, person) end;
    ⋮
  end
```

Members and *observers* are implemented as instances of Members and Observers, which are subclasses of Person. The method new_meeting defined in the Person class checks to see if the notification of the new meeting conflicts with the individual's schedule and if it doesn't, assigns the OID of the *meeting manager* to the time slots. If it does conflict, it sends negotiate message to itself.

```
class Person superclass Object;
    indirectvar timetable : vector;    { individual's private timetable }
  begin
    method Initialize()
      begin timetable := new_array(24);
            for time := 0 to 23 do timetable[time] := nil end;
    method new_meeting(start, end, manager)
        var time;
      begin for time := start to end do
              begin if (timetable[time] <> nil) then
                        self <== [negotiate timetable[time]];
                    timetable[time] := manager end
      end
  end
```

The classes, Member and Observer, implements the negotiate method. The *member* is required to attend the meeting and if the personal schedule and the notification of meeting conflicts (a fact known by the reception of the self-sent negotiate message), the meeting schedule has to be changed. The `abort` statement explicitly aborts the transaction and all the effects of the meeting set-up are cleaned up.

```
class Member superclass Person
  begin
    method negotiate(manager) begin abort end
  end
```

On the other hand, if an *Observer* receives the negotiate message, he/she cancels the previous appointment.

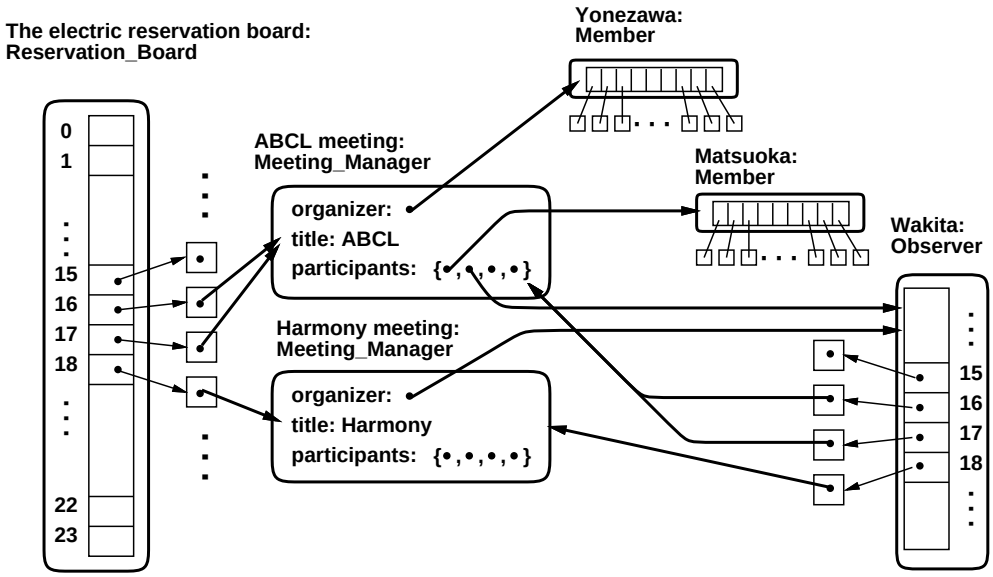

Figure 3: Distributed schedule management

where $\langle variable_i \rangle$ is the name of indirect instance variable and $\langle type_i \rangle$ is either `unstructured` or `vector`. To define a revised version of Vector, Vector_Revised, the programmer only need to replace the instance variable declaration with indirect instance variable declaration:

```
class Vector_Revised superclass Object;
    indirectvar vec : vector;
begin
  ⋮    { The method definitions are the same as that of the Vector class }
end
```

4 Example: Distributed Schedule Management

This section presents our solution to the distributed schedule management. We model the world by three kinds of objects: an *electric reservation board*, *persons*, and *meeting managers* as in Figure 3. The *electric reservation board* object represents the timetable of the conference room and a *person* object represents a project member or an observer of a meeting. A *meeting manager* object represents meeting. The electric reservation table is implemented by a vector whose elements represent time slots. The time slots initially contain `nil`, which means no meeting is set on the time the slot represents. If it contains a reference to a *meeting manager* object, the room is reserved for the meeting the *meeting manager* object represents. A *meeting manager* holds the information pertaining to the meeting, such as the *organizer*, the *title*, and the *potential participants*. In Figure 3, the room is reserved from 16:00 to 17:00 by the ABCL meeting, which is organized by Yonezawa and whose potential participants include Matsuoka and Wakita. Each person has his private schedule. For example, Wakita attends the ABCL meeting from 16:00 to 17:00 and then attends the HARMONY meeting from 17:00.

Below we give the code for an *organizer* to set up a meeting.[5] The *organizer* creates a new *meeting manager*, announces the meeting to its *participants*, and fills in the *electric reservation board*. The *electric reservation board* is implemented by the Vector_Revised class. With the extensive use of asynchronous message passings (indicated by `<=`), the set-up task is highly concurrent. If the *organizer* finds that the room has already been reserved for a certain time slot, he abolishes the meeting set-up by executing the `abort` statement, which aborts the transaction.[6]

[5] asynchronous and synchronous are indicated by `<=` and `<==`, respectively

[6] In general, the `abort` statement aborts the transaction at the deepest level of the nesting, thus the `abort` statement can be used in a method which does not contain a transaction block

Suppose there is an instance of Vector, *the_vector*, to which we want to assign its elements to the values $v_1, v_2, \ldots, v_n$ *atomically* (i.e., without interference from the Set and Ref accesses of other concurrent activities). The HARMONY script below is sufficient for this purpose. In the body of the transaction block, n messages [Set i, v_i] are sent asynchronously (as indicated by "<="). It may be the case that when the body of the transaction block (i.e., for loop) completes its execution, messages sent to *the_vector* remain unprocessed. As we explained above, the commitment of the transaction is postponed until all the messages sent to *the_vector* are processed completely. Totality of the transaction ensures that no other concurrent activities recognize the partially assigned vector and atomicity ensures that the transaction is not interfered by other concurrent activities. In the case of a conflict with other concurrent activities, the runtime system restores the state before the transaction was invoked. A concurrency control algorithm that supports these features is explained in the appendix. (See also Section 5.)

```
the_vector := create_instance Vector(n);   { the_vector is an instance of Vector }
  ⋮
Transaction
  begin for i := 1 to n do
            the_vector <= [Set i, v_i]
                 { messages [Set i, v_i] are sent to the_vector asynchronously }
  end
```

3.2 Indirect Instance Variable

The implementation of Vector class in the Section 3.1 may suffer from its low internal concurrency because concurrent updates to its different elements are prohibited: two concurrent transactions updating different elements of the same vector cause a write/write conflict at the Vector object and hence one of them must abort. When an instance of Vector is shared, it can easily become a performance bottleneck. This is because each object is the unit of concurrency control (e.g., read/write operation on an object is used for serializability validation).

Figure 1: Naive implementation of a vector

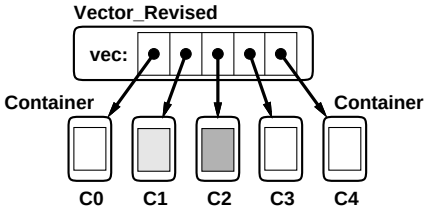

Figure 2: A revised implementation of a vector

This situation can be avoided by splitting the vector object into element pieces so that write/write conflict does not occur (Figure 2). The array structure *vec* of Vector_Revised does not hold the data items directly; rather, it holds the object identifiers (OIDs) of Container objects which in turn hold the data items. In this implementation, the vector becomes highly concurrent because the OIDs of Container objects in Vector_Revised are never changed by the method invocations. Even if the contents of the different elements of the vector are modified by several concurrent transactions, only the contents of Containers objects are modified; all transactions can safely commit because there are no conflicts.

Although this technique of splitting object into several subcomponents for gaining concurrency is powerful, the problem is that the class definition (of, e.g., Vector_Revised) may become somewhat complicated due to the indirect message passing to its subcomponents. Also it is difficult for the compiler to detect and use for optimization the fact that the OID references to the Container objects would remain exclusive to the vector (i.e., not passed to other objects.). We solve these problems by introducing a linguistic construct called *indirect instance variable declaration*, which gives programmers the illusion that the subcomponents indirectly referenced by OIDs are directly accessed in the same manner as ordinary instance variables. The form of indirect instance variable declaration is:

indirectvar $\langle variable_1 \rangle$: $\langle type_1 \rangle$, ..., $\langle variable_m \rangle$: $\langle type_m \rangle$;

control algorithm. In what follows, we assume that objects communicate via asynchronous/synchronous messages and a method defined for an object is executed one at a time (in other words, more than one method cannot be executed simultaneously in the same object even though the messages may have come from different transactions).

In explaining our approach in a concrete manner, we will use the notation borrowed from the syntax of our language HARMONY. HARMONY is a class-based object-oriented concurrent programming language, where objects communicate using asynchronous/synchronous message passing. HARMONY also supports inheritance mechanism for code sharing.

3.1 Transaction Block

A transaction is explicitly specified with a language construct called the *transaction block*. A transaction block is a sequence of statements that is surrounded by "**Transaction begin**" and "**end**":

Transaction
  begin ⟨*statement*⟩; ...; ⟨*statement*⟩ **end**

The ⟨*statement*⟩ is a general statement including an instance variable reference/assignment statement, a message passing form, a reply form, and a transaction block.

A transaction specified by this construct is ensured *totality*; a transaction either completes entirely or have no visible effects. Messages may be sent asynchronously in the same transaction block, and objects receiving the messages may in turn send more messages asynchronously (*cascaded asynchronous messages*). Thus the completion of all the statements in a transaction block does not always mean that the cascaded messages are all completely processed. In order to maintain the totality of a transactions, after the completion of the last statement in the transaction block, the transaction must wait for the completion of all the cascaded messages sent during the its execution (The completion of the last statement of the transaction block and that of the transaction block itself are different!).

A transaction specified by a transaction block is also ensured *atomicity*; a transaction does not logically interfere with other concurrent activities acting upon the same object, because only serializable operation sequences are permitted on an object. By grouping multiple message passings within a transaction block, all the message passings in the block become collectively atomic. As a result, our approach enables us to execute arbitrary combinations of methods atomically, whereas the RPC-based approach needs to introduce an atomic RPC handler which invokes nested transactions. A concurrency control mechanism that realizes this flexible transaction facilities is presented in the appendix. Details of the semantics of the transaction block is found in [Wak91].

We can summarize the notion of our transaction as *a computing process that guarantees the totality and atomicity of all the cascading effects caused by executing the statements inside a transaction block.*

Let us illustrate this idea with a simple example: Consider an instance of the Vector class which accepts two types of messages [Ref *i*] and [Set *i*, *val*] and behaves like a vector. An instance of the Vector class is created by executing the *instance creation form*, "**create_instance** Vector(*n*)". Upon creation, the method Initialize is invoked with an argument *n*, which specifies the size of the vector.[3] The Vector class provides two methods: Set assigns the *i*th element of the vector to *value* and Ref returns the value of the *i*th element[4].

```
class Vector superclass Object;
    var vec;
  begin
    method Initialize(size)
      begin vec := new_array(size) end;
    method Set(i, val)    { assign the ith element to val }
      begin vec[i] := val end
    method Ref(i)    { reply to the sender with the ith element }
      begin !vec[i] end;
  end
```

[3] A method named Initialize is a special method for initialization and is automatically invoked at the instance creation with the arguments of instance creation form as its actual arguments.

[4] "! *value*" sends the sender a reply message, which contains *value*

summarizes our work and report the current status of development. The appendix presents a concurrency control algorithm which realizes our nested transaction facilities.

2 Design Considerations of Transaction Facilities in OOCP

Several programming languages for the development of distributed reliable systems have been proposed, which support nested transaction facilities (e.g., Argus [Lis88], Aeolus [LW85], and Avalon/C++ [DHW88]). The main feature of such languages is the support for atomic transaction that (1) copes with system failure due to unreliable network communication and/or system crashes, and (2) establishes atomicity of operations in order to avoid interference between concurrent activities. In these languages, each Remote Procedure Call (RPC) is an invocation of an atomic transaction and nested RPCs naturally correspond to nested transactions [Mos85]. For OOCP languages, we propose more powerful transaction facilities to facilitate the development of sophisticated organizational information systems. The following two subsections discuss the basic features of our transaction facilities and more advanced features will be given in Section 5.

2.1 Support for Non-RPC Message Passing Protocols

Communication protocols provided in the previous languages which support transaction facilities are all based on RPC. However, for the development of sophisticated distributed organizational systems that require both reliability and execution efficiency, more flexible communication protocols are required (e.g., rendezvous, asynchronous message passing, future mode message passing, etc.). For example, multicasting new_meeting messages to the participants can naturally be programmed by simultaneous asynchronous message passings. In order to integrate the transaction mechanism and non-RPC message passing protocols, we have to generalize the notion of nested transaction.

2.2 Compound Message Passing in a Single Transaction

In the RPC-based approach to nested transaction, each RPC automatically invokes a transaction and hence explicit specification of transactions does not appear in the code. However, this approach has two drawbacks. First, the system performance degrades due to the runtime overhead of managing *unnecessarily nested transactions*. The nesting of transactions increases both space and execution complexity because each transaction requires duplication of information for possible future rollbacks and runtime overhead for serializability validation. It is preferable to keep the nesting level as small as possible.

Second, the RPC-based approach often lacks linguistic constructs for making a group of message passings atomic. For example, when an organizer sets up a meeting, slots of the electric reservation board and the timetables of the participants should be filled in *atomically*. In the RPC-based approach, a new remote procedure must be created to make grouping of message passings to the electric reservation board and participants, because defining a remote procedure is the only way to make activities collectively atomic. This fact implies that RPC-based approach forces the programmer to cut logical sequence of code into different remote procedures and thus modularity of code decreases significantly. To improve the modularity of distributed software, we need an explicit way of grouping several message passings into a single transaction. Our approach to this problem is to introduce a language construct called a *transaction block* as will be explained in the following section.

3 Transaction Facilities

We designed our transaction facilities for OOCP languages which take into account considerations presented in Section 2. The characteristics of our approach are: (1) a transaction is explicitly specified with a linguistic construct called a *transaction block* which forms a group of message passings into a single transaction and also supports non-RPC message passing protocols, (2) we use the *object splitting technique* with *indirect instance variable declaration* that provides a convenient means for fine-grained control of customization of concurrency, and (3) we also specify atomicity level for each object, which enables the programmer to customize concurrency control independent of the choice of the concurrency

may be a member or an observer of other projects; thus scheduling for different meetings may conflict with individuals' schedules.[1] To model the problem in the frameworks of object-oriented computation, entities such as people, timetables, and slots in a timetable can be modeled as autonomous objects. Suppose an organizer tries to set up a meeting: he looks up the timetable of the conference room to find the convenient time slots that meet his/her personal schedule (sends a query message to the *electric reservation board* object), fills in the time slots of the timetable (sends a reserve_room message to the *electric reservation board* object), and announces the schedule for the meeting to the potential participants of the meeting (multicasts new_meeting messages to objects that represent the potential participants). The project members are required to attend the meeting, and when they receive an announcement of a meeting that conflicts with their schedule, they ask the organizer to change the schedule of the meeting. The presence of observers are optional to the meeting: thus, when observers receive a conflicting announcement of a meeting, they can either honor or cancel the previous arrangement.

When we model this schedule management in object-oriented concurrent computation frameworks, we have to resolve the conflicts of concurrent activities at two places: one at the electric reservation board object and another at personal timetable of each individual:

1. Multiple *organizer* objects might try to fill in the same time slot of the *electric reservation board*. For example, one organizer tries to reserve from 14:00 to 16:00, and another tries to reserve from 15:00 to 17:00 simultaneously. When a conflict occurs, as in this case, at least one of the *organizers* has to cancel his/her reservation, or the system will result in an inconsistent state.

2. A notification of a meeting to a participant may conflict with his/her private schedule (e.g., planning a date with a girl/boy friend). If the proposed schedule of the meeting is not accepted by the participant, the organizer has to restore the previous state where he initiated the setting-up of the meeting: the *organizer* object sends cancel messages to all the objects that are side-effected (e.g., the *electric reservation board* object and *participant* objects who have accepted the notification).

We could avoid these problems by locking the slots of the *electric reservation board* (or the personal timetables). This solution, however, has two drawbacks: (1) degradation of concurrency from pessimistic locking and (2) possible deadlocks. In more general terms, the choice of the synchronization scheme would greatly affect the correctness as well as the performance of the system — the safest scheme would reduce concurrency to nil, while more elaborate schemes not only would risk the correctness but also could result in deadlock or starvation.

We tackled this problem and propose nested transaction facilities that are more powerful than the nested transaction facilities of previous concurrent languages such as Argus [Lis88], Aeolus [LW85], and Avalon/C++ [DHW88]. Our transaction facilities are incorporated into our prototype OOCP language HARMONY, realizing a *transparent view* of shared objects in distributed environments (i.e., an illusion as if objects were not shared but exclusively used by the application) [Weg87, MR90]. The support of nested transactions in asynchronous[2] message passing protocols is one of the novel features of our work in contrast to the previous works which have been based on RPC-style communication. The nested transaction facilities we propose can be easily incorporated into a large family of OOCP languages. Though our transaction features are presented with a simple CSCW application of the schedule management, they are powerful enough to facilitate implementation of more general systems. In addition to the standard notion of atomicity, we introduced the notion of the level of atomicity for each object, which allows customization of concurrency control mechanisms for the purpose of permitting nonserializable operation sequences and improving execution efficiency by increasing concurrency. Our approach to customization achieves good encapsulation owing to the independence of both the class hierarchy and the underlying concurrency control protocol used in the runtime system.

Section 2 discusses several features desirable for transaction facilities for OOCP. In Section 3, we introduce two linguistic constructs, the *transaction block* for transaction specification and *indirect instance variable declaration* for fine-gained control of concurrency. These constructs are used in our solution to the distributed schedule management in Section 4. Section 5 explains the advanced features of our transaction facilities, namely customization of concurrency control using the level of atomicity. Section 7

[1] For simplicity, we assume that a member of one project cannot be an observer of another.

[2] When a message is sent *asynchronously*, the sender does not wait for the message to be received by the destination object. When a message is sent *synchronously*, the sender waits for the destination object to reply back.

Linguistic Supports for Development of Distributed Organizational Information Systems in Object-Oriented Concurrent Computation Frameworks

Ken Wakita and Aki Yonezawa

Department of Information Science, the University of Tokyo*

Keywords and Phrases
Concurrency Control, Nested Transaction, CSCW Development,
Object-Oriented Concurrent Programming

Abstract

For the development of large and sophisticated distributed organizational information systems, one of the most prevalent, yet difficult problems is secure concurrent access to shared objects while preserving collective system consistency. In modeling and implementing such systems in object-oriented concurrent languages, linguistic supports are needed to enable the programmer to have a transparent view of shared objects. For this purpose, we generalized the standard notion of nested transactions to accommodate it to non-RPC message passing protocols, and introduced the notion of object-wise atomicity level. This paper discusses our proposal of linguistic constructs for such transaction facilities.

1 Introduction

The availability of inexpensive yet powerful personal workstations, equipped with large memory and connected to a high performance network, motivated the development of complex distributed organizational information systems such as groupware systems, computer-supported cooperative work (CSCW) systems, and distributed object-oriented database (OODB) systems. However, most of such systems are still programmed using conventional programming languages that lack linguistic supports for distributed computation. With the increase in number and complexity of the distributed organizational information systems in the near future, there will be tremendous demands for concurrent programming languages that can naturally model and implement those distributed systems [BST89].

Past research has shown the object-oriented concurrent computation paradigm to be effective in modeling, describing, and implementing distributed systems. Its underlying computation model is powerful enough to describe various concurrent phenomena in the real world [Hew77, Hew86, Agh90], and proposed linguistic constructs for synchronization are flexible and allow the development of highly concurrent systems [Yon90, Ame87, Nie87]. We found, however, the descriptive power of linguistic constructs in object-oriented concurrent programming (OOCP) languages designed to date is still insufficient for a class of problems found in the development of large, distributed organizational systems. What are lacking are the linguistic supports that guarantee the secure concurrent accesses to shared (concurrent) objects while preserving their collective consistency. We will illustrate below the problem using a simple typical CSCW application, *distributed schedule management* [B+90].

Consider several research projects that share a conference room for their meetings. Each project has its project members and observers, and is led by a project organizer. Each project member and observer

*Physical mail address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. Phone 03-3812-2111 (overseas +81 3-3812-2111) ex. 4095, 4114. E-mail: {ken-w, yonezawa}@is.s.u-tokyo.ac.jp.