Proceedings of International Conference on Distributed
Systems, Software Engineering and Database Systems

# A Donation-Based Approach to Concurrency Control for Object-Oriented Database Systems*

Ye-In Chang and Chung-Che Wu
Dept. of Applied Mathematics
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C

**Abstract**

*Traditional serializability theory is not efficient enough to meet the needs of the advanced database applications, for example, supporting long transactions. Moreover, the new generation of database applications requires modeling techniques more powerful than the ones offered by relational database systems. Object-oriented databases provide a promising alternative for advanced applications such as computer-aided design and multimedia databases. In this paper, to increase the degree of concurrency control and improve the performance of advanced database applications, we propose a donation-based concurrency control protocol for object-oriented database systems. Basically, our proposed protocol is based on the granularity locking method in the ORION object-oriented database system and altruistic locking (which is proposed for supporting long transactions).*

## 1 Introduction

The new generation of computer-based applications, such as computer-aided designed and manufacturing (CAD/CAM), multimedia databases (MMDB), office automation, and software development environments (SDEs), requires more powerful techniques to generate and manipulate large amounts of data. These applications are termed *advanced* to distinguish them from *traditional* database applications, such as banking systems and airline reservations systems [2]. Transactions in these *advanced* applications differ from those in conventional applications in many respects, where a *transaction* is a partially ordered sequence of read and write operations that are executed atomically on the objects [3]. Some of these differences include the duration of transactions, granularity of transaction management features [9], the cooperation nature and the consistency constraint. These differences make the advanced applications have roughly performance and even cannot work under traditional database techniques.

In the database systems, maintaining the consistency of the shared data needs the *concurrency control* algorithms to controlling accesses to these data. There is a theory which has been developed for proving the correctness of database concurrency control algorithms. In this theory, a concurrency control algorithm is regarded as correct if it ensures that any interleaved execution of transactions is equivalent to a serial one. Such executions are called *serializable*. *Serializability theory* provides a framework for ensuring the correctness of concurrency control protocols. All most conventional concurrency control algorithms are constructed by using serializability theory. But in the advanced database applications, traditional serializability theory is not efficient enough to meet the needs of the advanced database applications, which need to support long transactions, user control, and synergistic cooperation for their specific characteristics [2, 4, 5]. But these new needs in the advanced database applications usually conflict to the serializability theory which cannot support more semantic information and more relaxing consistency constraints to them. Thus, there is a need for a different mechanism to handle long-duration transactions [9].

Moreover, the newer generation of database applications requires modeling techniques more powerful than the ones offered by relational database systems. Object-oriented databases (OODB) provide a promising alternative for advanced applications such as computer-aided design and multimedia databases [7, 9, 10]. An object-oriented data model not only provides great expressive power to decide data and to define complex relationships among data, it but also provides mechanisms for behavioral abstraction.

In this paper, to increase the degree of concurrency control and improve the performance of advanced database applications, we propose a *donation-based* concurrency control protocol for object-oriented database systems. Basically, our proposed protocol is based on the granularity locking method in the ORION object-oriented database system and altruistic locking [8] (which is proposed for supporting long transactions). When a transaction requests a lock on an object, it should use the *granularity* locking method to get the right of accessing the object. If there is any conflict in the *granularity* locking process, the protocol will check whether the conflict object has been *donated*. If the object has been *donated*, the transaction still can lock the conflicting object. However, in altruistic locking, they only keep one copy of donated object no matter the number of donations. In this way, two problems occur. First, there is no way to

*This research was supported by the National Science Council of the Republic of China, Grant No. NSC 85-2213-E-110-034

ımmit the right version of data. Second, when one of .e transactions which participates in the donation of certain data object aborts, all of those transactions hich participate in the donation of the same data )ject must abort. To solve the above two problems, ɜ have a different approach to the implementation of .e *donate* operation, called *improved altruistic lock- g.* In our approach, the *donate* operation will create new private object space and copy the instances of .e donated object to the new created object, then turn the new object address.

The rest of the paper is organized as follows. ɛction 2 presents the object-oriented data model ıncepts and introduces the ORION object-oriented ıtabase system. Section 3 describes altruistic locking ·otocol. Section 4 presents the proposed donation- ısed strategy for advanced database applications. Fi- ılly, Section 5 gives the conclusions.

## The object-oriented data model

In this Section, we introduce object-oriented con- pts and the granularity locking method in the RION object-oriented database system [6]. The core )ject-oriented concepts include the following items ]: (1) *Objects and object identifiers.* In an object- iented system, all real-word entities are represented ı objects and each object has a unique identifier. An )ject may be a simple object or it may contain other )jects. (2) *Attributes and methods.* An object can ıve one or more attributes and methods, which op- ate on the values of these attributes. (3) *Encapsu- tion and message passing.* External entities cannot rectly access the attributes of the object. To access .e values of these attributes, messages have to be nt to the object. (4) *Classes and Instance.* Classes ·ovide a means to group objects that share the same t of attributes and methods. Objects that belong ı a class are called *instances* of that class. A class ɛscribes the form (attributes) of its instances, and ıe operations (methods) applicable to its instances ) *Class hierarchy and inheritance.* The classes in ı object-oriented systems form a hierarchy (the class erarchy) where a subclass inherits all the attributes ıd methods of its superclass(es). Inheritance pro- des an important means for sharing behavior among lated objects. In ORION object-oriented database ·stem, applications impose locking requirements on ıree orthogonal types of hierarchy, and one of them the well-known granularity hierarchy for logical en- ties, devised to minimize the number of locks to be t [6]. In the granularity locking, ORION supports /e lock modes: *IS*, *IX*, *S*, *SIX* and *X* [6]. Instance )jects are locked only in *S* or *X* mode to indicate hether they are to be read or updated, respectively. owever, class objects may be locked in any of the /e modes. An *IS* (Intention Shared) lock on a class .eans that instances of the class are to be explicitly cked in *S* mode as necessary. An *IX* (Intention Ex- usive) lock on a class means instances of the class ill be explicitly locked in *S* or *X* mode as necessary. n *S* (Shared) lock on a class means that the class ɜfinition is locked in *S* mode, and all instances of ıe class are implicitly locked in *S* mode, and thus

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | ✓ | ✓ | ✓ | ✓ | NO |
| IX | ✓ | ✓ | NO | NO | NO |
| S | ✓ | NO | ✓ | NO | NO |
| SIX | ✓ | NO | NO | NO | NO |
| X | NO | NO | NO | NO | NO |

Figure 1: Compatibility matrix for granularity locking

are protected from any attempt to update them. An *SIX* (Shared Intention Exclusive) lock on a class implies that the class definition is locked in *S* mode, and all instances of the class are implicitly locked in *S* mode and instances to be updated (by the transaction holding the *SIX* lock) will be explicitly locked in *X* mode. An *X* (Exclusive) lock on a class means that the class definition and all instances of the class may be read or updated. An *IS*, *IX*, *S*, or *SIX* lock on a class implicitly prevents the definition of the class from being updated [6]. The compatibility matrix of Figure 1 defines the semantics of the lock modes. A compatibility matrix indicates whether a lock of mode $M_2$ may be granted to a transaction $T_2$, when a lock of mode $M_1$ is presently held by a transaction $T_1$.

## 3 Altruistic locking

Altruistic locking is a modification to two-phase locking (2PL) in which several transactions may hold locks on an object simultaneously, under certain conditions [8]. In altruistic locking, the basic idea is to allow long transactions to release their locks early, once it is determined that the data which the locks protect will no longer be accessed. Therefore, altruistic locking provides a third concurrency control operation, called *Donate*, in addition to Lock and Unlock. Like Unlock, Donate is used to inform the scheduler that access to an object is no longer required by the locking transaction. However, when Donate is used, the donating transaction is free to continue to acquire new locks; i.e., Donate and Lock operations need not be two-phase.

Several rules govern the use of the Donate operation by well-formed transactions. Transactions may only donate objects which they currently have locked. They can not access any objects that they have donated. Moreover, a donation is not a substitute for unlocking. A well-formed transaction must eventually unlock every object that it locks, regardless of whether it donated any of those objects. Donate operations are beneficial because they may permit other transactions to lock the donated object before it is unlocked.

Clearly, an altruistic scheduler should not allow an arbitrary access to an object that has been donated but not unlocked. To avoid this problem, an altruistic scheduler places restrictions on transactions that accept donations, i.e., those transactions which access donated objects. These restrictions are embodied in two rules which are observed by an altruistic scheduler, much as a 2PL scheduler observes a rule that

| Time | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|---|---|---|---|---|---|
| (1) | WLock(a) | | | | |
| (2) | WLock(b) | | | | |
| (3) | WLock(c) | | | | |
| (4) | Donate(a) | | | | |
| (5) | Donate(b) | | | | |
| (6) | Donate(c) | | | | |
| (7) | | WLock(a) | | | |
| (8) | | | WLock(a) | | |
| (9) | | WLock(b) | | | |
| (10) | | WLock(c) | | | |
| (11) | | Donate(a) | | | |
| (12) | | Donate(b) | | | |
| (13) | | | | WLock(a) | |
| (14) | | | | WLock(d) | |
| (15) | | | | | WLock(e) |
| (16) | | | | | WLock(b) |

Figure 2: An example which shows how altruistic locking rules work: transactions $T_1$, $T_2$ work correctly, transaction $T_3$ fails to lock object $a$, transaction $T_4$ fails to lock object $d$, transaction $T_5$ fails to lock object $b$.

transactions should not simultaneously hold locks on any object. The first of these rules is as follows:

**Altruistic Locking Rule 1.** *Two transactions may not simultaneously hold locks on the same object unless one of the transactions donates the object first.*

If a transaction $X$ locks an object that has been donated (and not yet unlocked) by another transaction $Y$, we say that transaction $X$ is *in the wake of* the donating transaction $Y$. A transaction is *completely* in the wake of another transaction if all objects it locks are in the other's wake.

**Altruistic Locking Rule 2.** *If a transaction $T_a$ is in the wake of another transaction $T_b$, then $T_a$ must be completely in the wake of $T_b$ until $T_b$ performs $T_b$'s first Unlock operation.*

For example, in Figure 2, transaction $T_1$ locks the objects $a$, $b$, $c$ and donates them. When transaction $T_2$ wants to lock objects $a$, $b$, $c$ , transaction $T_2$ can enter the wake of transaction $T_1$ and lock them. When transaction $T_3$ wants to lock object $a$, transaction $T_3$ will be rejected because object $a$ is locked by transaction $T_2$. Transaction $T_4$ locks object $a$ successively after transaction $T_2$ donates object $a$. However, transaction $T_4$ can not lock object $d$, since transaction $T_4$ must be completely in the wake of transaction $T_2$, just like transaction $T_2$ is completely in the wake of transaction $T_1$. Transaction $T_5$ can not lock object $b$ for the same reason, since transaction $T_5$ locks object $e$ which is not donated by transactions $T_1$ and $T_2$.

## 4 A donation-based concurrency control protocol for OODB

In this section, to increase the degree of concurrency control and improve the performance of advanced database applications, we present a *donation-based* concurrency control protocol for object-oriented databases. To simplify our design, we apply the query model of [1] in our method. The query model has the following syntax: **(Receiver Selector Arg$_1$ Arg$_2$ Arg$_3$ ...)**, where *Receiver* is the object, or a message which can be evaluated to an object, to which the message is sent, *Selector* is the name of the method,

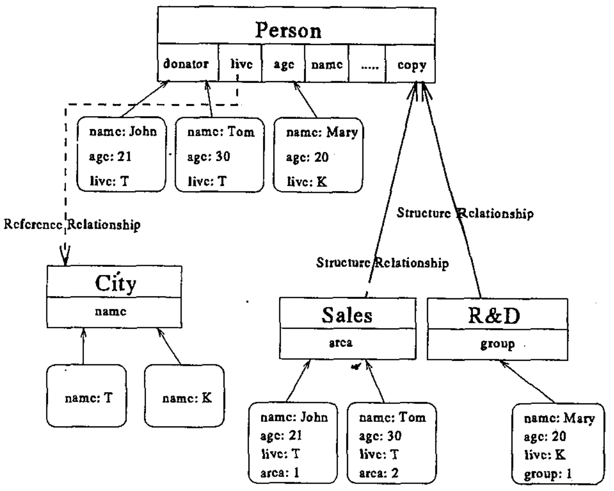

Figure 3: The structure graph of the *Person* department

```
(Receiver Select Arg₁ Arg₂ Arg₃ ...)
begin
  for each superclass of Receiver do
    Request(superclass of Receiver, IS, Tran, { });
  Request(Receiver, S, Tran, { });
  Select the wanted instance with conditions of
  Arg₁ Arg₂ Arg₃ ...;
  return the value;
end;
```

Figure 4: The algorithm for selecting instances

and the arguments, *Arg$_1$*, *Arg$_2$*, etc., are objects or blocks of code which can be evaluated to objects. In this query model, if we have a query on one or more instances of a class, the class and all classes specified as non-primitive domains of the attributes of the class must be recursively traversed. For example, in Figure 3, when we select the instances of class *Person*, we also need to traverse the attribute *live* of class *Person*, which takes the values of instances of class *City*, as well as the domains of non-primitive attributes of these classes. In this section, we will see how this query model works with our method.

### 4.1 Select and change instances of classes

When a transaction *Tran* wants to read some objects under some conditions, we use *Select* to be the Selector in the query model, which is shown in Figure 4. According to the model stated in [6], when we want to select some instances of certain class, we need to request a lock on its all ancestors and then lock itself before return its selected result. The function *request* in the algorithm is to use the altruistic locking method to check whether any conflicting lock happens. If no conflict occurs, the algorithm will go ahead and do the operation with wanted arguments. If a conflict occurs, the algorithm will wait in function *request* until the condition of conflicting lock is not happened. The details of function *request* will be described later.

When we change the instances of a certain class,

```
(Receiver Change Arg1 Arg2 Arg3 ...Argn)
begin
 for each superclass of Receiver do
  Request(superclass, IX(or SIX), Tran, { });
 Request(Receiver, X, Tran, { });
  Change the wanted instance with conditions of Arg1
  Arg2 Arg3 ... Argn-1 to the value of Argn
 return the value;
end;
```

Figure 5: The algorithm for changing instances

```
(Receiver Add Arg)
(Receiver Sub Arg)
(Receiver Update Arg)
begin
 for each subclass of Receiver do
  Lock(subclass of Receiver, X, Tran);
 Lock(Receiver, X, Tran);
 Change the definition of the wanted attribute to Arg;
 return the value;
end;
```

Figure 6: The algorithm for changing the definition of a class

we need to lock the class in X mode and to lock its ancestors in IX or SIX mode. The algorithm is shown in Figure 5.

## 4.2 Change definitions of classes

When we want to change the definition of a class, we have to lock all its subclasses in the inheritance relationship. The reason is that changing the definition of a class will also change the definition of its all subclasses, which inherit from it. Therefore, in this case, there is no way to donate this class object, i.e., to share an unlocked class object among transactions. The function *lock* in our algorithm as shown in Figure 6 is the same as the *lock* function in 2PL. In this way, it avoids the problem that changing the definition of a class will conflict with any change in the class lattice rooted at the class.

## 4.3 Donate an object

In altruistic locking, they only keep one copy of donated object no matter the number of donations. In this way, two problems occur. First, there is no way to commit the right version of data. For example, transaction $T_1$ locks and writes $x = 1$ and then donates the data object $x$. Transaction $T_2$ then locks and writes $x = 2$. Next, transaction $T_1$ commits. Since there is only one copy of data, the value of data object $x$ which transaction $T_1$ commits will be 2, instead of 1. Although serializability is maintained in altruistic locking, a transaction may commit a data which is not what it intends. Second, when one of the transactions which participates in the donation of a certain data object aborts, all of those transactions which participate in the donation of the same data object must abort, too, no matter how long those transactions have been executed. This really violates the motivation of altruistic locking, since altruistic locking is proposed for supporting long transactions. Therefore, to solve the above two problems, we have a different approach to the implementation of the *donate* operation, called *improved altruistic locking.* In our approach, the *donate* operation will create a new private object space and copy the instances of the donated object to the new created object, then return the new object address, which will be described in details later. If the donated object is an instance, the copy of the new object should include the contents of the donated object. If the donated object is a class, the copy of the new object should include the definition and the instances of the donated object. Therefore, the following two rules must be followed, where the first rule is the same as *Altruistic Locking Rule 1,* and the semantics of the second rule will be discussed in details in the *Request* function later.

**Improved Altruistic Locking Rule 1.** *Two transactions may not simultaneously hold locks on the same object unless one of the transactions donates the object first.*

Let the set of transactions which a transaction $T_a$ is in their wakes (when transaction $T_a$ starts its first read/write operation) be called $WakeSet_a$.

**Improved Altruistic Locking Rule 2.** *If a transaction $T_a$ is in the wakes of a set of transactions, $WakeSet_a$, then for every data object $x$ which transaction $T_a$ accesses, $x$ must be donated by a transaction $T_b$, where $T_b \in WakeSet_a$. Moreover, when transaction $T_b$ unlocks the data object which is accessed by transaction $T_a$, we let $WakeSet_a = WakeSet_a - \{T_b\}$.*

We also have to change the class inheritance relationship at the runtime. Since the locking conflict will be detected at the new copy of the donated object, the locking path should include the new copy of the donated objected, instead of the donated object. Figure 3 shows the system state of class *Person* initially. Figure 7 shows the system state after transaction $T_2$ locks class *Person.Shadow* which is donated by transaction $T_1$, where *Person.shadow* is the new copy of the donated object *Person.* The dotting lines shows the class inheritance relationship between the class and its superclass. We see that the superclass of class *Sales* and class *R&D* has been changed to class *Person.Shadow.*

To simplify the implementation, we do not directly change the class-superclass relationship when a donation occurs; instead, we apply a *run-time* approach. That is, when an object searches for its superclass, it will look for its newest superclass by tracing the donation relationship. For example, in Figure 7, after transaction $T_2$ updates John's age from 21 to 25 and donates class *Person.Shadow,* and then transaction $T_3$ locks class *Person.Shadow.Shadow,* the new superclass of class *Sales* and class *R&D* is changed to class *Person.Shadow.Shadow* as shown in Figure 8. Next, if transaction $T_3$ wants to search the superclass of class *Sales,* it will go to class *Person* by the original structure relationship between class *Person* and class *Sales,* then go to the newest superclass, class *Person.Shadow.Shadow,* by following the donation relationship between class *Person,* class *Person.Shadow* and class *Person.Shadow.Shadow.*

In order to implement the above *donate* opera-

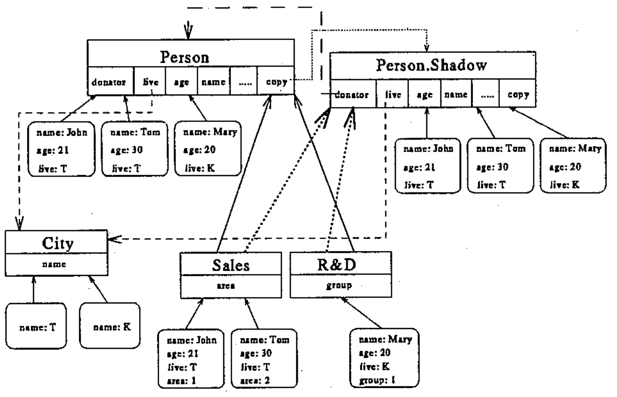

Figure 7: The system state after class *Person* is donated by transaction $T_1$ and locked by transaction $T_2$

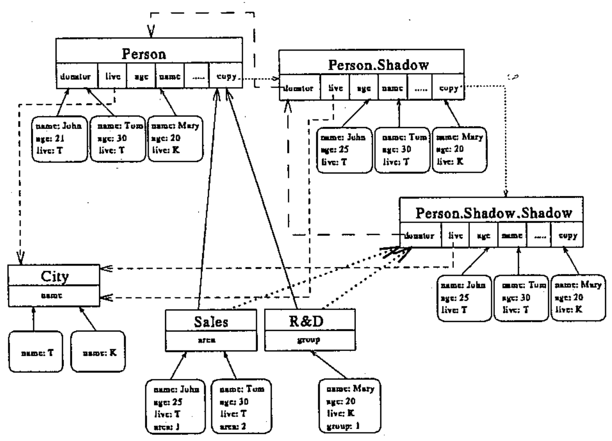

Figure 8: The system state after class *Person.Shadow* is donated by transaction $T_2$ and locked by transaction $T_3$

```
struct Transaction_type
{ set of obj_id_type locked_obj = { };
     set of transaction_id_type donator = { }; }

class Class_type
{ public:
    char* lock_mode = 'NULL';
    char* real_object = 'True';
    char* public = 'False';
    char* commit = 'False';
    obj_id_type copy = nil;
    obj_id_type donator = nil;
    set of transaction_id_type locked_tran = { }; }
```

Figure 9: The data structure of objects and transactions

```
Donate(d_object)
begin
  d_object.public := 'True';
end;
```

Figure 10: Function *Donate*

tion, for each object, we have to add four more attributes: *public*, *real_object*, *donator* and *copy* as shown in Figure 9. When a transaction donates an object $x$, $x.public$ is set to true, which is done in Function *Donate* as shown in Figure 10. After the new copy of object $x$ (called $x.Shadow$) is created, the address of the new copy $x.Shadow$ is recorded in $x.copy$. On the other hand, object $x$'s address is recorded at $x.Shadow.donator$. In this way, the donation relationship is maintained by attributes *donator* and *copy*. This step is finished in Function *Make_Donate*. Moreover, in order to distinguish whether the object is the original one or the donated one, the attribute *real_object* is used. In summary, the Function *Donate* is to set an object's attribute *public* to 'True'. The actions of creating and initializing a new copy of donated object are shown in the function *Make_Donate*. The function *Donate* can be used at any time when a transaction wants to donate its locked object. The function *Make_Donate* is used in function *Request* which will be discussed later to really create a donating space for the transaction which wants to enter other transaction's wake.

```
Make_Donate(d_object)
begin
  Make a copy of all the instances and the defini-
  tion of d_object and return a pointer d_ptr which
  points to the new copy named as
  d_object.shadow;
  d_object.copy := d_ptr;
  d_object.copy.donator := d_object;
  d_object.copy.real_object := 'False';
  d_object.lock_mode := 'NULL';
  d_object.locked_tran := { };
end;
```

Figure 11: Function *Make_Donate*

## 4.4 Commit

When an object is committed by a transaction, we need to write back the results to the permanent space. Since there is a sequence of donation relationship among some transactions, a commitment issued by a transaction $T_1$ can occur only after all the transactions which occur before transaction $T_1$ in the sequence of donation relationship have committed. (Note that to achieve this goal, a variable *commit* associated with each object is needed as shown in Figure 9.) Moreover, a transaction should return the results of the copy of donated objects which it locked to the donator of the new copy. Furthermore, before the copy of a donated object is destroyed, the donation relationship between its donator and the next new copy should be updated. Some variables, including *public* and *commit* and *lock_mode* should also be reset. The *commit* algorithm is shown in Figure 12. Moreover, to avoid the problem that a transaction may unlock modified objects before committing, we follow the *Strict Two-Phase Locking*, in which write locks cannot be unlocked until the locking transaction has committed. To simplify this function, we add an *unlock* operation before the end of the *commit* procedure.

## 4.5 Abort

When a transaction $T_1$ is aborted, all of the data objects locked by transaction $T_1$ must be released and all of the changes to the data objects must be undone. Since there is a sequence of donation relationship among some transactions, the abortion of a transaction $T_1$ should cause the abortion of all the transactions which occur after transaction $T_1$ in the sequence of donation relationship. Moreover, if transaction $T_1$ is the only one transaction which locks the copy of the data object, then either this copy of data object is destroyed (when it is not the original one) or some local variables, including *public*, *lock_mode* and *copy*, are reset (when it is the original one). The *abort* algorithm is shown in Figure 13.

## 4.6 Request, unlock and lock

The following three operations: *Request* in Figure 14, *Unlock* in Figure 15 and *Lock* in Figure 16, are used as essential operations of other operations. In the *Request* operation, there are two functions must be performed. One is to trace the donation relationship until the last new copy of donated object is reached. The other is to decide whether the requesting transaction is in the wakes of all those transactions which have donated data objects and some of those data objects have been locked by the requesting transaction.

First, the *Request* operation will check whether attribute *r_object.public* is 'True'. If the attribute *r_object.public* is 'True', then we have to add the transaction identifier which has locked this copy of data object *r_object* to a set *donate_set* which is used to record those transactions which the requesting transaction is going to be in their wakes. (Note that the initial value of *donate_set* is an empty set as specified in Figure 9.) Next, we will then check whether attribute *r_object.copy* is *nil*. If the attribute *r_object.copy* is *nil*, it means that the object which has been donated but the new copy of the object has not been created.

```
Commit(tran)
begin
for each c_object in tran.locked_obj do
  if c_object.real_object = 'True' then
    begin
      write c_object to permanent space;
      c_object.commit = 'True';
      c_object.lock_mode := 'NULL';
      if c_object.copy = nil then
        c_object.public := 'False';
      Unlock(c_object, tran);
    end
  else
    if (c_object.donator.commit = 'True') and
       (c_object.donator.real_object = 'True') then
      if (c_object.public = true and
          c_object.copy ≠ nil) then
        begin
          copy all the changed instances to the related
          instances of c_object.donator;
          write c_object to permanent space;
          c_object.donator.copy := c_object.copy;
          c_object.copy.donator := c_object.donator;
          Unlock(c_object, tran);
          destroy itself;
        end
      else
        begin
          copy all the changed instances to the related
          instances of c_object.donator;
          write c_object to permanent space;
          c_object.donator.copy := nil;
          if c_object.donator.real_object = 'True' then
            c_object.donator.public := 'False';
          Unlock(c_object, tran);
          destroy itself;
        end
    else
      waiting for some time period and Committing again;
end;
```

Figure 12: The *Commit* algorithm

```
Abort(a_tran)
begin
  for each a_object in tran.locked_obj do
    begin
      send an abort message to transaction a_tran;
      if a_object.copy ≠ nil then
        for each t in a_object.copy.locked_tran
          Abort(t);
      if a_object.locked_tran = { } then
        if a_object.real_object ≠ 'True' then
          begin
            Unlock(a.object, a_tran);
            destroy itself;
          end
        else
          begin
            a_object.public := 'False';
            a_object.lock_mode := 'NULL';
            a_object.copy := nil;
          end;
      Unlock(a.object, a_tran);
    end;
end;
```

Figure 13: The *Abort* algorithm

```
Request(r_object, requestmode, tran, donate_set)
begin
 if r_object.public = 'True' then
 begin
  donate_set :=donate_set ∪ r_object.locked_tran;
  if r_object.copy = nil
  begin
   if tran.donator $\neq$ { } then
   begin
    if tran.donator $\not\supseteq$ donate_set then
      waiting for some time period and requesting again
   end
   else if tran.locked_obj $\neq$ { }
         and donate_set $\neq$ { } then
    waiting for some time period and requesting again;
    Make_Donate(r_object);
  end;
  Request(r_object.copy, requestmode,
          tran, donate_set);
 end
 else if r_object.lock_mode conflicts
         with requestmode then
    waiting for some time period and requesting again
  else
  begin
   r_object.lock_mode := requestmode;
   tran.donator := tran.donator ∪ donate_set;
   r_object.locked_tran:=r_object.locked_tran ∪ {tran};
   tran.locked_obj := tran.locked_obj ∪ {r_object};
   r_object.commit := 'False';
  end;
end;
```

Figure 14: Function *Request*

```
Unlock(u_object, tran)
begin
 u_object.locked_tran := u_object.locked_tran - {tran};
 tran.locked_obj := tran.locked_obj - {u_object};
 for each t in u_object.locked_tran
   t.donator := t.donator - {tran};
 if u_object.copy $\neq$ nil
   Unlock(u_object.copy, tran);
end;
```

Figure 15: Function *Unlock*

```
Lock(l_object, requestmode, tran)
begin
 if l_object.lock_mode conflicts
                    with requestmode then
   waiting for some time period and requesting again
 else
  begin
   l_object.lock_mode := requestmode;
   l_object.locked_tran := r_object.locked_tran ∪ {tran};
   tran.locked_obj := tran.locked_obj ∪ {l_object};
   l_object.commit := 'False';
  end;
end;
```

Figure 16: Function *Lock*

Up to this point, the *donate_set* has recorded all the transactions which have locked the original requesting data object or new copies of the requesting data object. If the requesting transaction is already in the wakes of some other transactions, which can be detected by testing the condition *tran.donator* $\neq$ { }, then the requesting transaction must also ensure that those transactions in *donate_set* is a subset of the ones in *tran.donator* (i.e., *tran.donator* $\supseteq$ *donate_set*) such that the requesting transaction follows *Improved Altruistic Locking Rule 2.*

Note that if transaction $T_1$ has been in the wake of transactions $T_2$ and $T_3$ due to the donated data object $X$, then transaction $T_1$ must be completely in the wakes of transactions $T_2$ or $T_3$ or ($T_2$ and $T_3$). That is, at this point, if the data object $Y$ is already locked and donated by transactions $T_2$, $T_3$ and $T_4$, then transaction $T_1$ cannot lock the data object $Y$. In this case, $T_1$.*donator* = $\{T_2, T_3\}$ and *donate_set* = $\{T_2, T_3, T_4\}$. If we let transaction $T_1$ lock data object $Y$ in this case, then an unserializable schedule can occur as follows: transaction $T_1$ donates data object $X$ again and transaction $T_4$ also wants to lock data object $X$; a cycle between transactions $T_1$ and $T_4$ in the serialization graph will occur. So does the case of $T_1$.*donator* = $\{T_2, T_3\}$ and *donate_set* = $\{T_2, T_4\}$ (or *donate_set* = $\{T_3, T_4\}$).

Consider the case in which $T_1$.*donator* = $\{T_2, T_3\}$ (due to the donated data object $X$) and *donate_set* = $\{T_2\}$ for locking data object $Y$ in our approach. Transaction $T_1$ can lock data object $Y$ in this case. Next, if $T_3$.*donator* = $\{T_2\}$ (due to the donated data object $X$) and transaction $T_3$ wants to lock the same data object $Y$, it finds *donate_set* = $\{T_2, T_1\}$; therefore, transaction $T_3$ must wait. If transaction $T_2$ commits and transaction $T_3$ tries to lock data object $Y$ again, it finds $T_3$.*donator* = { } and *donate_set* = $\{T_1\}$. According to our implementation, if a transaction is not in any wake (tran.donator = { }), but has already locked data object $X$ (tran.locked_obj $\neq$ { }) and now must be in the wake of another transaction (donate_set $\neq$ { }), it must wait again. Moreover, according to our approach, transaction $T_1$ cannot commit until transactions $T_2$ and $T_3$ have committed

Note that while in altruistic locking, a transaction can lock a data object only if *tran.donator* = *donate_set*, since there is no restriction on the order of commitment in altruistic locking. Consider the same case in which $T_1$.donator = $\{T_2, T_3\}$ (due to the donated data object $X$) and *donate_set* = $\{T_2\}$ for locking data object $Y$ in altruistic locking. If we let transaction $T_1$ lock data object $Y$ in altruistic locking, a problem occurs as follows. Assume that transactions $T_2$ and $T_1$ commit. At this point, transaction $T_3$ is not in the wake of any transaction; therefore, transaction $T_3$ can lock data object $Y$, resulting in a directed edge from $T_1$ to $T_3$ in the serialization graph. However, in the serialization graph, there is already a directed edge from $T_3$ to $T_1$ for locking data object $X$. That is, an unserializable schedule occurs. Therefore, to avoid this case, a transaction can lock a data object only if *tran.donator* = *donate_set* in altruistic locking; while in our approach, to avoid this case, a transaction can lock a data object only if *tran.donator* $\supseteq$ *donate_set*, since transaction $T_1$ cannot commit until transactions $T_2$ and transaction $T_3$ have committed. Therefore, we have to relax *Altruistic Locking Rule 2* to *Improved Altruistic Locking Rule 2*.

If the requesting transaction violates *Improved Altruistic Locking Rule 2*, then it has to wait for some time period and requests again. If the requesting transaction really follows *Improved Altruistic Locking Rule 2*, then function *Make_Donate* is called to create the new copy. If attribute *r_object.public* is 'True' and attribute *r_object.copy* is not nil, it means that the object has been donated and there is another transaction which has locked the new copy of the object. In this case, the function *Request* will be called recursively with a new parameter *r_object.copy* until the last new copy of donated object has been reached. When a transaction wants to lock the last object which does not have a new copy, it will then check the compatibility matrix of Figure 1 and decide whether a conflict occurs. If a conflict does not occur, the transaction can lock the object with wanted locking mode. Moreover, we have to add those transactions recorded in *donate_set* to *tran.donator*, add the requesting transaction *tran* to the set *r_object.locked_tran* which records those transactions that have locked the data object *r_object*, add *r_object* to *tran.locked_obj* which records those data objects that *tran* has locked, and reset *r_object.commit* to false. If a conflict occurs, the transaction will wait for a moment and then requests again.

When a transaction uses an *Unlock* operation, as shown in Figure 15, to release an object *u_object*, it removes itself from the *u_object.locked_tran* set and the *donator* set of all other transactions which have a lock on the object *u_object*, since other transactions do not have to be in the wake of this unlocking transaction. If *u_object.copy* is not equal to nil, the *Unlock* operation will be called recursively until the last new copy of the donated object has been reached. The *Lock* operation, as shown in Figure 16, is simply a two-phase locking operation.

The correctness of the proposed donation-based approach to concurrency control for object-oriented database systems is proved in [11].

## 5 Conclusion

The mechanisms which solve the problems of long transactions can be divided to two approaches: one approach is extending serializability-based mechanisms, the other approach is relaxing serializability mechanisms. In this paper, we have proposed a donation-based strategy which belongs to extending serializability-based mechanisms to solve the problems of long transactions for object-oriented database systems. Basically, our proposed protocol is based on the locking method in ORION object-oriented database system and altruistic locking How to apply the relaxing serializability mechanisms for object-oriented database systems is the future research direction.

## References

[1] J. Banerjee, W. Kim and K. Kim, "Queries in Object-Oriented Databases", *Proc. of the 4th Int. Conf. on Data Eng.*, pp. 31-38, 1988.

[2] N. S. Barghouti and G. E. Kaiser, "Concurrency Control in Advanced Database Applications", *ACM Computing Surveys*, Vol. 23, No. 3, pp. 269-317, Sep. 1991.

[3] P. A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass, 1987.

[4] A. A. Farrag and M. T. Ozsu, "Using Semantic Knowledge of Transactions to Increase Concurrency", *ACM Trans. on Database Systems*, Vol. 14, No. 4, pp. 503-525, Dec. 1989.

[5] H. Garcia-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed Database", *ACM Trans. on Database Systems*, Vol. 8, No. 2, pp. 186-213, Jun. 1983.

[6] J. F. Garza and W. Kim, "Transaction Management in an Object-Oriented Database System", *Proc. of the ACM SIGMOD Conf.*, pp. 37-45, 1988.

[7] A. R. Hurson and S. H. Pakzad, "Object-Oriented Database Management Systems: Evolution and Performance Issues", *IEEE Computer Magazine*, pp. 48-60, Feb. 1993.

[8] K. Salem, H. Garcia-Molina and J. Shands, "Altruistic Locking", *ACM Trans. on Database Systems*, Vol. 19, No. 1, pp. 117-165, Mar. 1994.

[9] P. Shah and J. Wong, "Transaction Management in an Object-Oriented Data Base System", *Journal of Systems Software*, pp. 115-124, Jan. 1994.

[10] D. Woelk and W. Kim, "Multimedia Information Management in an Object-Orient Database System", *Proc. of the Conf. on Very Large Data Base*, pp. 319-329, 1987.

[11] C. C. Wu, "A Donation-based Approach to Concurrency Control for Object-Oriented Database Systems", *Master Thesis*, Dept. of Applied Mathematics, National Sun Yat-Sen, University, Kaohsiung, Taiwan, R.O.C .