

# A Hybrid Approach to Concurrency Control for Object-Oriented Database Systems

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

## ABSTRACT

In advanced database applications, traditional serializability theory is not enough to meet the need of 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 systems. Object-oriented databases provide a promising alternative for advanced applications such as computer-aided design (CAD) and multimedia database (MMDB). In this paper, to increase the degree of concurrency control for advanced database applications with long-duration transactions, we propose a *donation-based* concurrency control protocol for object-oriented database systems. Basically, our proposed protocol can be considered as a hybrid approach which combines the locking method in ORION object-oriented database system and altruistic locking (which is proposed for supporting long transactions). However, we have a different implementation for the *donate* operation used in altruistic locking, which can decrease the number of cascading abortions. That, in turns, can improve the system performance. From our simulation study, we show that our proposed approach can have a higher throughput and a shorter response time than altruistic locking.

**Key Words:** concurrency control, data consistency, database management systems, serializability, transactions

## I. Introduction

The new generation of computer-based applications, such as computer-aided design 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. It is desirable to base these kinds of advanced applications to behave like the traditional database applications. These applications are termed *advanced* to distinguish them from traditional database applications, such as banking systems and airline reservations systems. In the traditional applications, the nature of the data and the operations performance on the data are amenable to concurrency control mechanisms that enforce the classic transactions. But transactions in these advanced applications differ from those in conventional applications in many respects. Some of these differences include the duration of transactions, granularity of transaction management features, the cooperation nature and the consistency constraint (Barghouti and Kaiser, 1991). These differences make the advanced applications have roughly performance and even cannot work in 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, where a *transaction* is a partially ordered sequence of read and write operations that are executed atomically on the objects. Such executions are called *serializable*. *Serializability theory* provides a framework for ensuring the correctness of concurrency control protocols. Almost conventional concurrency control algorithms are constructed by using serializability theory, and then they will obey the consistency of the database systems (Bernstein *et al.*, 1987).

The theory is quite simple. It abstracts the semantics of concurrent computations by a small set of syntactic concepts: computations are partially ordered sets of operations; interference between operations is described by *conflicts*; and interference between program executions in a computation is described by a *directed* graph. Serializability is guaranteed if the graph is acyclic (Bernstein *et al.*, 1987).

But in advanced database applications, traditional serializability theory is not enough to meet the need of advanced database applications. In addition to the need for the conventional database applications, the advanced database systems also need to support long transactions, user control, and synergistic cooperation for their specific characteristics (Farrag and Ozsu, 1989; Garcia-Molina, 1983; Kim *et al.*, 1984; Klahold *et al.*, 1985; Korch and Speegle, 1988; Lee and Liou, 1996; Pu *et al.*, 1988). 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 constraint to them. For example, concurrency control mechanisms like locking and timestamps, which are used in conventional data bases, are inadequate for handling long-duration transactions. If locking is used for concurrency control, we have the problem of long-duration waits. Thus, a long-duration transaction may delay other transactions. Such long delays are unacceptable. If timestamps are used, aborting a long-duration transaction implies heavy penalties in terms of wasted computations and other critical resources. Thus, there is a need for a different mechanism to handle long-duration transactions.

Moreover, the newer generation of database applications requires modeling techniques more powerful than the ones offered by relational systems. Object-oriented databases provide a promising alternative for advanced applications such as computer-aided design (CAD) and multimedia database (MMDB) (Hurson and Pakzad, 1993; Woelk and Kim, 1987).

In this paper, to increase the degree of concurrency control for advanced database applications with long-duration transactions, we propose a *donation-based* concurrency control protocol for object-oriented database systems. Basically, our proposed protocol can be considered as a hybrid approach which combines the locking method in ORION object-oriented database system and altruistic locking (Salem *et al.*, 1994) (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, we have a different implementation for the *donate* operation

used in altruistic locking, which can decrease the number of cascading abortions. That, in turns, can improve the system performance in terms of throughput and response time. Note that 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. Second, when one of the transactions  $T_k$  which participates in the donation of a certain data object aborts, all of those transactions  $T_i$  which participate in the donation of the same data object must abort, too, no matter transaction  $T_i$  occurs before or after transaction  $T_k$  in the donation relationship. 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. From our simulation study, we show that our proposed approach can have a higher throughput and a shorter response time than altruistic locking.

The rest of the paper is organized as follows. Section II presents the object-oriented data model concepts and introduces the ORION object-oriented database system. Section III describes altruistic locking protocol. Section IV presents the proposed donation-based strategy for advanced database applications and the proof of correctness. Section V shows the simulation results. Finally, Section VI gives the conclusions.

## II. The Object-Oriented Data Model

In this Section, we introduce object-oriented concepts and the locking method in ORION object-oriented database systems (Garza and Kim, 1988). The core object-oriented concepts include the following items (Shah and Wong, 1994):

1. *Objects and object identifiers*. In an object-oriented system, all real-world entities are represented as objects and each object has a unique identifier. An object may be a simple object or it may contain other objects.
2. *Attributes and methods*. An object can have one or more attributes and methods, which operate on the values of these attributes.

3. *Encapsulation and message passing.* External entities cannot directly access the attributes of the object. To access the values of these attributes, messages have to be sent to the object.
4. *Classes and Instance.* Classes provide a means to group objects that share the same set of attributes and methods. Objects that belong to a class are called *instances* of that class. A class describes the form (attributes) of its instances, and the operations (methods) applicable to its instances
5. *Class hierarchy and inheritance.* The classes in an object-oriented systems form a hierarchy (the class hierarchy) where a subclass inherits all the attributes and methods of its superclass(es). Inheritance provides an important means for sharing behavior among related objects.

In the Advanced Computer Architecture Program at MCC, they have built a Prototype object-oriented database system called ORION. ORION applications impose locking requirements on three orthogonal types of hierarchy, and one of them is the well-known granularity hierarchy for logical entities, devised to minimize the number of locks to be set. The other two types of hierarchy are consequences of the rich data model that ORION supports, and necessitate extensions to the current theory of locking (Garza and Kim, 1988).

In the granularity locking, ORION supports five lock modes: *IS*, *IX*, *S*, *SIX* and *X* (Garza and Kim, 1988). Instance objects are locked only in *S* or *X* mode to indicate whether they are to be read or updated, respectively. However, class objects may be locked in any of the five modes. An *IS* (Intention Shared) lock on a class means that instances of the class are to be explicitly locked in *S* mode as necessary. An *IX* (Intention Exclusive) lock on a class means instances of the class will be explicitly locked in *S* or *X* mode as necessary. An *S* (Shared) lock on a class means that the class definition is locked in *S* mode, and all instances of the class are implicitly locked in *S* mode, and thus 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

	<b>IS</b>	<b>IX</b>	<b>S</b>	<b>SIX</b>	<b>X</b>
<b>IS</b>	✓	✓	✓	✓	<b>NO</b>
<b>IX</b>	✓	✓	<b>NO</b>	<b>NO</b>	<b>NO</b>
<b>S</b>	✓	<b>NO</b>	✓	<b>NO</b>	<b>NO</b>
<b>SIX</b>	✓	<b>NO</b>	<b>NO</b>	<b>NO</b>	<b>NO</b>
<b>X</b>	<b>NO</b>	<b>NO</b>	<b>NO</b>	<b>NO</b>	<b>NO</b>

Figure 1: Compatibility matrix for granularity locking.

*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 (Garza and Kim, 1988).

The compatibility matrix of Fig. 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  $T_1$ . For example, in Fig. 1, we see that when a transaction  $T_1$  holds an *X* lock, no lock of any mode may be granted to any other transaction. However, when transaction  $T_1$  holds an *S* lock, another transaction  $T_2$  may be granted an *IS* or *S* lock (Garza and Kim, 1988).

### III. 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 (Salem *et al.*, 1994). For the example shown in Fig. 2, altruistic locking will allow transaction  $T_2$  to acquire locks on objects *A* and *B*, although transaction  $T_1$  still holds locks on these objects. (Under 2PL, transaction  $T_1$  could simply Unlock objects *A* and *B* so that transaction  $T_2$  could acquire locks. However, because of the two-phase rule, transaction  $T_1$  would be unable to acquire

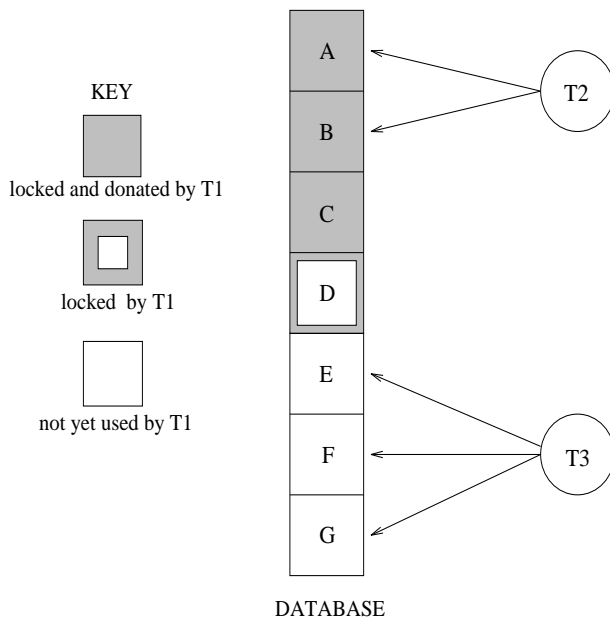


Figure 2: An altruistic locking example.

future locks on objects  $E$ ,  $F$ , and  $G$ .) 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.

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. Unlike other early-release approaches, the altruistic locking strategy guarantees serializable executions and places no restrictions on the way data must be accessed (e.g., transactions are not forced to scan the database sequentially or to traverse it down a tree).

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. Transactions are never required to donate any object; donations are always optional. 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 arbitrary access to an object that has been donated but not unlocked. In the example of Fig. 2, say transaction  $T_3$  reads the value of  $C$  donated by transaction  $T_1$  and then reads the value of  $E$  before transaction  $T_1$  gets a chance to lock and update it. The history would not be serializable. To avoid this problem, an altruistic scheduler places restrictions on transactions that accept donations; i.e., those transactions 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 transactions should not simultaneously hold locks on any object. The first of these rules is as follows:

*1. 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  $Y$ . (Note that both the object, and the transaction locking it, may be described as being in the wake.) A transaction is *completely* in the wake of another transaction if all objects it locks are in the other's wake.

*2. 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 Fig. 3, transaction  $T_1$  locks 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

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 3: 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  $b$ .

object  $b$  for the same reason, since transaction  $T_5$  locks object  $e$  which is not donated by transactions  $T_1$  and  $T_2$ .

## IV. A Donation-Based Concurrency Control Protocol for OODB

In this section, to increase the degree of concurrency control for advanced database applications with long-duration transactions, we present a *donation-based* concurrency control protocol for object-oriented database. 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, we have a different implementation for the *donate* operation used in altruistic locking, which can decrease the number of cascading abortions.

That, in turns, can improve the system performance.

To simplify our design, we apply the query model of (Banerjee *et al.*, 1988) in our method. The query model has the following syntax: (**Receiver Selector Arg<sub>1</sub> Arg<sub>2</sub> Arg<sub>3</sub> ...**), 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, and the arguments, *Arg<sub>1</sub>*, *Arg<sub>2</sub>*, 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 Fig. 4, 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. (Note that in Fig. 4, the *Reference Relationship* means the reference to a non-primitive attribute, and the *Structure Relationship* means the class-superclass relationship, which follows the structure graph used in Pang and Yang's object-oriented database system (1994). In this section, we will see how this query model works with our method.

## 1. Select the 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 Fig. 5. According to the model stated in (Garza and Kim, 1988), 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. An example of *selecting* all persons who live in T city and salary < 25000 from the *Person* department is as follows:

(name select :V (:V salary < 25000 and (:V live name = 'T'))).

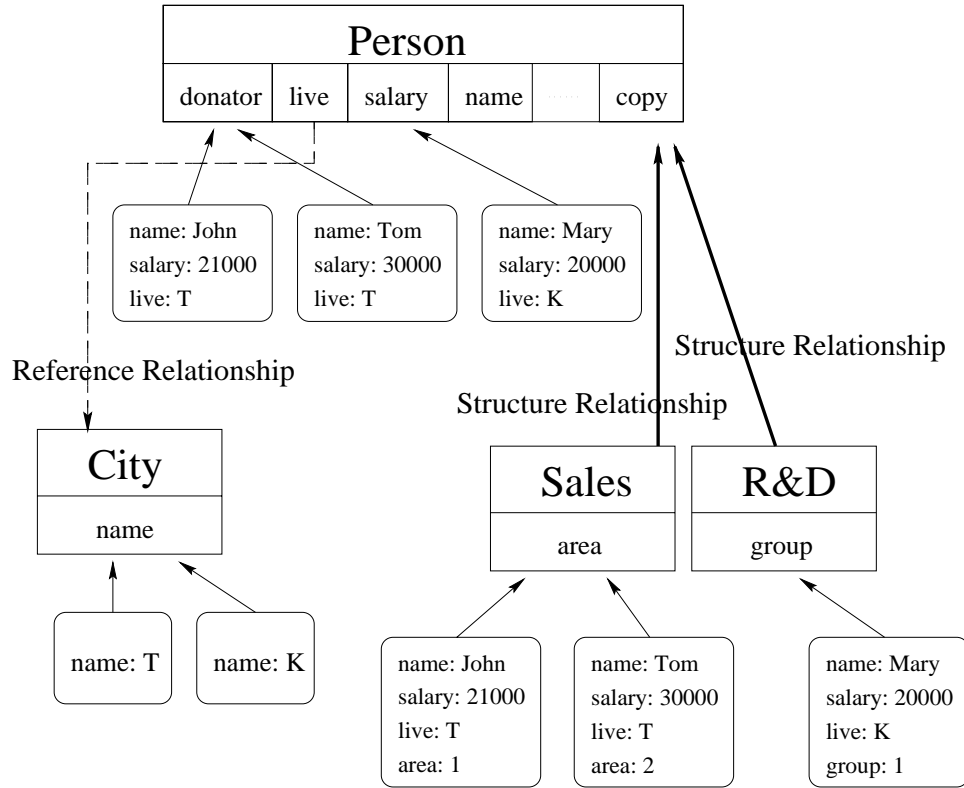


Figure 4: The structure graph of the *Person* department.

```

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

```

Figure 5: The algorithm for selecting instances.

```

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

```

Figure 6: The algorithm for changing instances.

## 2. Change Instances of Classes

When we change the instances of a certain 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 Fig. 6. An example of *changing* the person whose name is Mary to live in T city is shown as follows:

(live Change :V (:V name = Mary) 'T').

## 3. 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 definitions of a class will also change the definitions 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 Fig. 7 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. For example, in the altruistic locking method, consider that transaction  $T_1$  has locked class *Sales* in Fig. 4, changed the attribute *area* of instance John and donated class *Sales*. Now transaction  $T_2$  locks class *Sales* and changes the attribute *area* of *Sales* to attribute *client* which is the client records belonging to that salesman, and then transaction  $T_2$  commits. Later, when transaction  $T_1$  wants to commit its results, it will find that class *Sales* does not have the

```

(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 definition of wanted attribute to Arg;
  ⋮
  return value;
end;

```

Figure 7: The algorithm for changing the definition of a class.

attribute *area* as before, which causes an inconsistent database state. An example of *adding* an attribute *telephone* to class *Person* is shown as follows:

(person Add telephone).

#### 4. 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  $x$ , writes  $x = 1$ , and then donates the data object  $x$ . Transaction  $T_2$  then locks  $x$ , 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  $T_k$  which participates in the donation of a certain data object aborts, all of those transactions  $T_i$  which participate in the donation of the same data object must abort, too, no matter transaction  $T_i$  occurs before or after transaction  $T_k$  in the donation relationship. 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.

#### A. 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  $WakeSet_a$ .

#### B. 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. In general, Fig. 8 shows the system state after an object is donated by one transaction and is then locked by another transaction. 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 object, instead of the donated object. Fig. 4 shows the system state of class *Person* initially. Fig. 9 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 new added dotting lines from *Sales R&D* to *Person.Shadow* 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

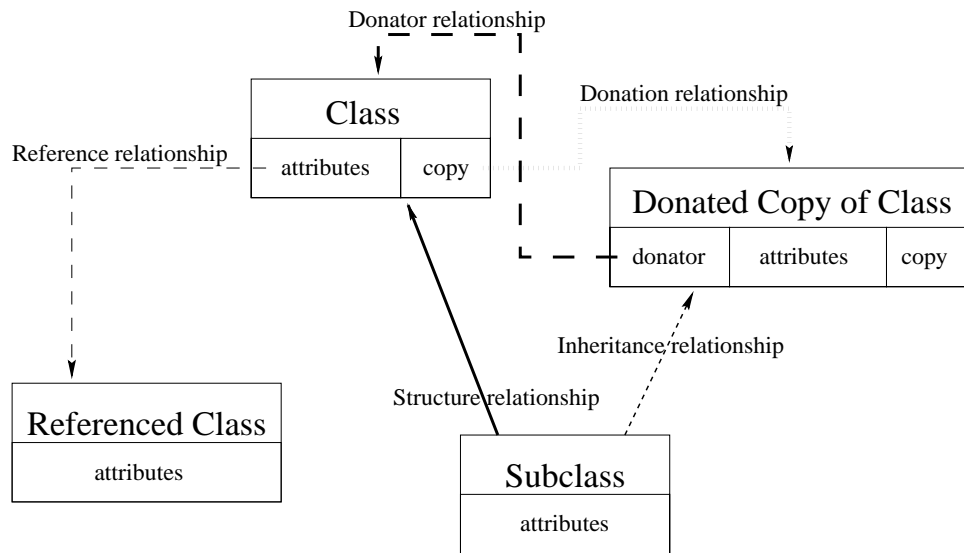


Figure 8: The system state after an object is donated by one transaction and is then locked by another transaction.

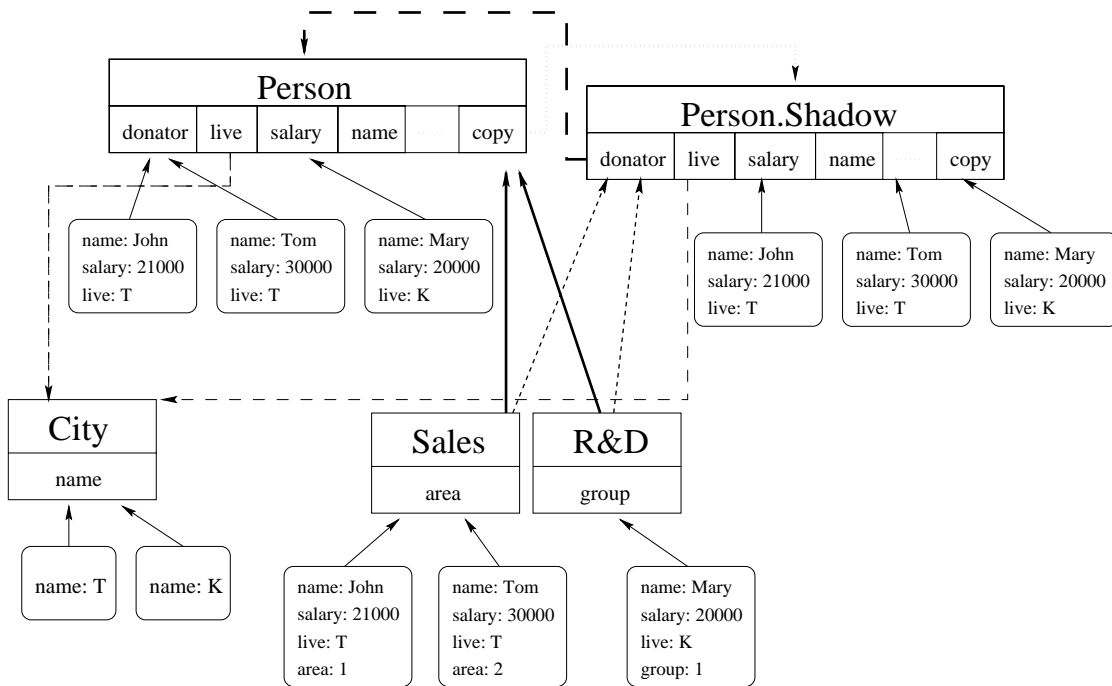


Figure 9: The system state after class *Person* is donated by transaction  $T_1$  and locked by transaction  $T_2$ .

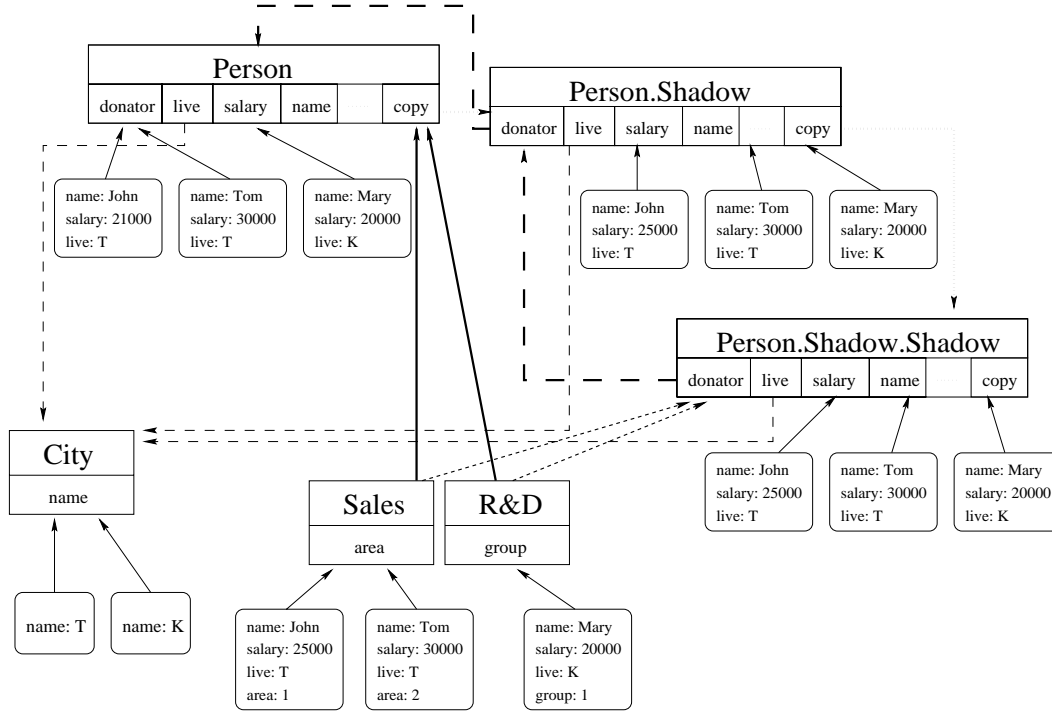


Figure 10: The system state after class *Person.Shadow* is donated by transaction  $T_2$  and locked by transaction  $T_3$ .

donation relationship. For example, in Fig. 9, after transaction  $T_2$  updates John’s salary from 21000 to 25000 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 Fig. 10. 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* operation, for each object, we have to add four more attributes: *public*, *realObject*, *donator* and *copy* as shown in Fig. 11. When a transaction donates an object  $x$ ,  $x.public$  is set to true, which is done in Function *Donate* as shown in Fig. 12. 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

```

struct Transaction_type
{
    :
    set of obj_id_type locked_obj = { };
    set of transaction_id_type donator = { };
}
class Class_type
{
    private:
        :
    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 11: The data structure of objects and transactions.

```

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

```

Figure 12: Function *Donate*.

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, 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 done in 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 definition 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_ptr.donator := d_object;
  d_ptr.real_object := 'False';
  d_ptr.public := 'False';
  d_ptr.commit := 'False';
  d_ptr.copy := nil;
  d_ptr.lock_mode := 'NULL';
  d_ptr.locked_tran := { };
end;

```

Figure 13: Function *Make\_Donate*.

## 5. 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, to ensure that the serializability property is satisfied, 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 Fig. 11.) 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 Fig. 14.

There are three cases for each of those objects locked by the committing transaction. For Case 1, an object is an original one; that is, it is not a donated one. In this case, in addition to writing the data to the permanent space and setting *object.commit* = 'True' and *object.lock\_mode* = 'NULL', we will reset *object.public* = 'False' if there is no extra copy of this object. For Case 2, an object is a donated one and its donator has been committed. In this case, two more situations must be considered. For Case 2(a), this object is donated again and the new copy of this object exists. In this case, in addition to copying the data to the donator, and writing the data to the permanent space, we will update the donation

```

Commit(tran)
begin
for each c_object in tran.locked_obj do
  if (c_object.real_object = 'True') then (* Case 1 *)
    begin
      write the 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') then (* Case 2 *)
      if (c_object.public = true and c_object.copy ≠ nil) then (* Case 2(a) *)
        begin
          copy all the changed instances to the related instances of c_object.donator;
          write the c_object to permanent space;
          Unlock(c_object, tran);
          c_object.donator.copy := c_object.copy;
          c_object.copy.donator := c_object.donator;
          destroy c_object;
        end
      else
        begin (* Case 2(b) *)
          copy all the changed instances to the related instances of c_object.donator;
          write the c_object to permanent space;
          Unlock(c_object, tran);
          c_object.donator.copy := nil;
          if c_object.donator.real_object = 'True' then
            c_object.donator.public := 'False';
          destroy c_object;
        end
      else
        waiting for some time period and Committing again; (* Case 3 *)
      end;
    end;
  end;
end;

```

Figure 14: The *Commit* algorithm.

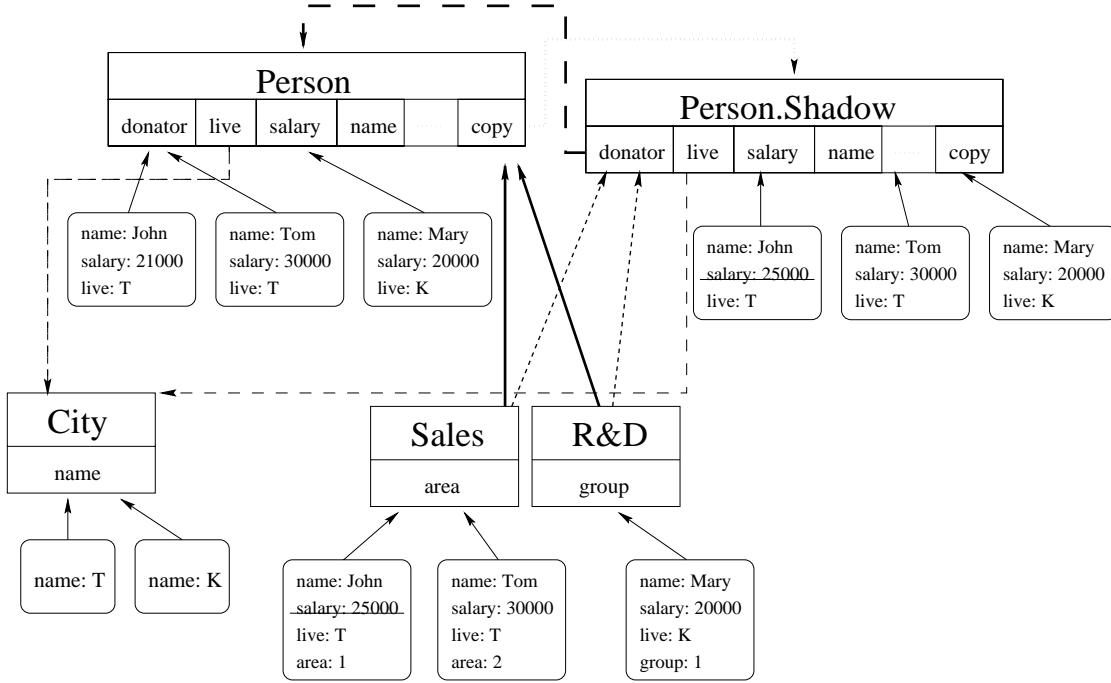


Figure 15: The system state after an instance of class *Person.Shadow* is updated.

relationship between the previous donator and the copy after the current object, and then unlock the object and destroy the object. For Case 2(b), this object is the last one in the donation relationship. For this case, in addition to copying the data to the permanent space, we have to update its *donator.copy* = nil. Moreover, if its donator is the original object, then we set its *donator.public* = 'False' to denote that there is no donation now. Finally, we unlock the object and destroy the object. For Case 3, it is a donated object but its donator has not been committed, then the committing transaction must wait for some time period and commit again.

For example, for the database shown in Fig. 4, let transaction  $T_1$  lock class *Person* and donate the class. Then transaction  $T_2$  locks class *Person.Shadow*, changes John's salary to 25000 (as shown in Fig. 15), and donates class *Person.Shadow* again. Later,  $T_3$  locks class *Person.Shadow.Shadow*, changes John's salary to 30000 (as shown in Fig. 16). From this donation order, we know that the history of donation should be  $T_1 < T_2 < T_3$ .

Assume that transaction  $T_1$  has already committed. Next, if transaction  $T_2$  commits its locked object, class *Person.Shadow* first, the results of class *Person.Shadow* will be returned to class *Person* as shown in Fig. 17, which is Case 2(a) in the Commit algorithm.

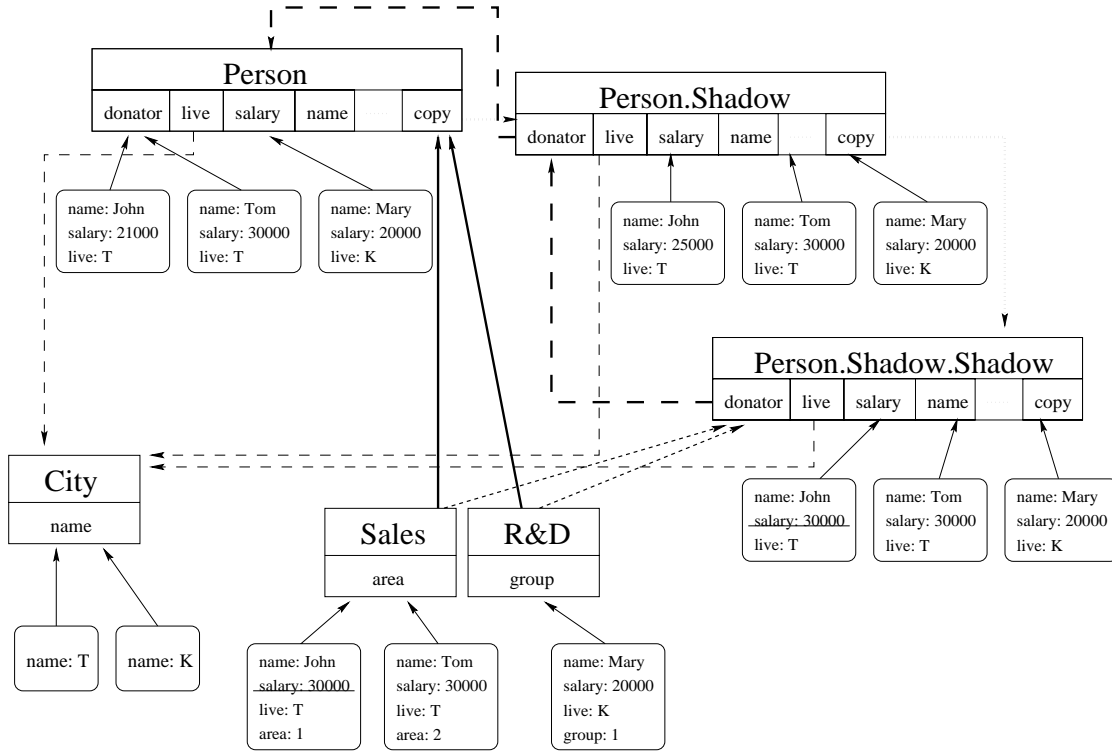


Figure 16: The system state after an instance of class *Person.Shadow.Shadow* is updated.

Next, transaction  $T_3$  commits its locked object, class *Person.Shadow.Shadow*, and the results of class *Person.Shadow.Shadow* will be returned to class *Person* again as shown in Fig. 18, which is Case 2(b) in the Commit algorithm. On the other hand, if transaction  $T_3$  commits class *Person.Shadow.Shadow* before transaction  $T_2$  commits class *Person.Shadow*, the *commit* operation issued by transaction  $T_3$  will not be executed until transaction  $T_2$  has issued the *commit* operation. 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.

## 6. 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

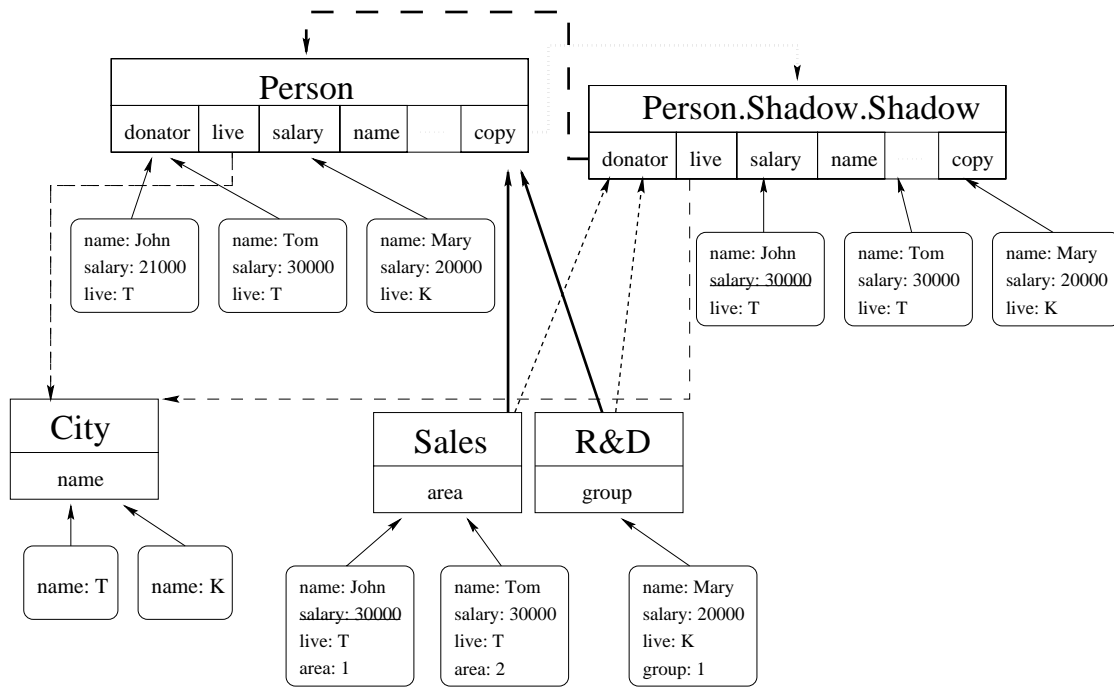


Figure 17: The system state after class *Person* and class *Person.Shadow* are committed.

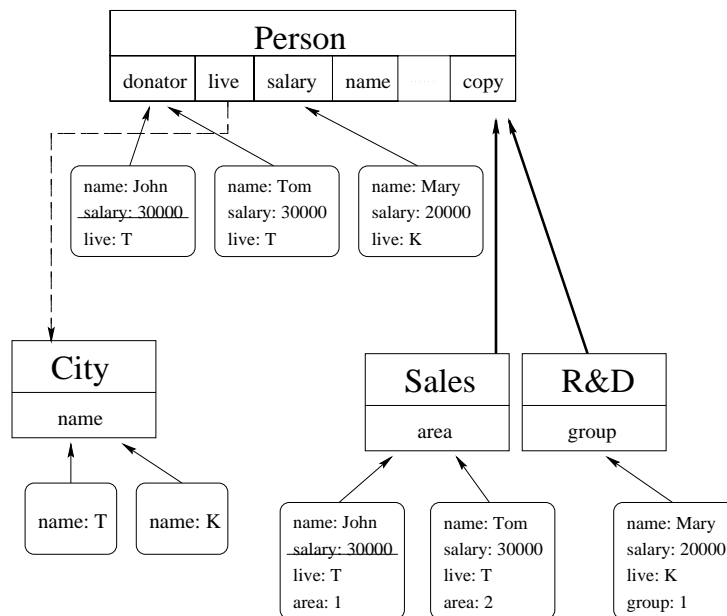


Figure 18: The system state after class *Person*, class *Person.Shadow* and class *Person.Shadow.Shadow* are committed.

$T_1$  should cause the abortion of those transactions which occur after transaction  $T_1$  in the sequence of the donation relationship in our approach, instead of the abortion of all of the transactions participating in the donation relationship in altruistic locking. Moreover, if transaction  $T_1$  is the only one transaction which locks the copy of the data object (ex., with  $X$  lock), 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). Finally, the lock held by transaction  $T_1$  is released. The *abort* algorithm is shown in Fig. 19. For example, in Fig. 16, when transaction  $T_2$  aborts, transaction  $T_3$  should also abort. In this case, an abort message will be sent to transaction  $T_2$  and transaction  $T_3$ , and the copies of the data object which transaction  $T_2$  and transaction  $T_3$  lock will be destroyed. Note that in altruistic locking, the abortion of transaction  $T_2$  in Fig. 16 will cause the cascading abortions of all the transactions which lock the same class *Person*; that is, transactions  $T_1$  and  $T_3$  must abort, too. Therefore, our proposed strategy can decrease the number of cascading abortion in altruistic locking, which in turns, can improve the system performance.

## 7. Request, Unlock and Lock

The following three operations: *Request* in Fig. 20, *Unlock* in Fig. 21 and *Lock* in Fig. 22, 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 the donated object is reached. The other is to decide whether the requesting transaction is in the wakes of all the transactions that have donated those data objects which are locked by the requesting transaction.

In Function *Request*, there are three cases. For Case 1, the requesting object is a donated one. For Case 2, the requesting object is an original one but is already locked by some other transaction with a conflicting mode. For Case 3, the requesting object is an original one and is free. In Case 1, first, the *Request* operation will check whether attribute *r\_object.public* is 'True'. If the attribute *r\_object.public* is 'True', then we add the transaction identifier which has locked this copy of data object *r\_object* to a set *donate\_set*

```

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

```

Figure 19: The *Abort* algorithm.

that is used to record those transactions which limit that the requesting transaction must be in their wakes. (Note that the initial value of *donate\_set* is an empty set.) 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 has been donated but the new copy of the object has not been created. Up to this point, the *donate\_set* has recorded all the transactions which have locked the original one 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  $tran.donator \supseteq donate\_set$  such that the requesting transaction follows *Improved Altruistic Locking Rule 2*; otherwise, the requesting transaction must wait for some time period and request again. (Note that the variable *tran.donator* is used to implement the *WakeSet* in **Improved Altruistic Locking Rule 2**.) Next, 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.

For example, if transaction  $T_1$  has been in the wake of transactions  $T_2$  and  $T_3$  due

```

Request(r_object, requestmode, tran, donate_set)
begin
  if r_object.public = 'True' then (* Case 1 *)
    begin
      donate_set := donate_set  $\cup$  r_object.locked_tran;
      if r_object.copy = nil
        begin
          if tran.donator  $\neq$  { } then
            begin
              if tran.donator  $\not\subseteq$  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 (* Case 2 *)
        waiting for some time period and Requesting again
      else
        begin (* Case 3 *)
          r_object.lock_mode := requestmode;
          tran.donator := tran.donator  $\cup$  donate_set;
          r_object.locked_tran := r_object.locked_tran  $\cup$  {tran};
          tran.locked_obj := tran.locked_obj  $\cup$  {r_object};
          r_object.commit := 'False';
        end;
      end;
    end;
end;

```

Figure 20: 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 21: 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 := l_object.locked_tran  $\cup$  {tran};
    end;
  end;
end;

```

Figure 22: Function *Lock*.

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  $Y$ . Since in this case,  $T_1.donator = \{T_2, T_3\}$  and  $donate\_set = \{T_2, T_3, T_4\}$ . In this case, if we let transaction  $T_1$  lock data object  $Y$  and if transaction  $T_1$  donates data object  $X$  again and transaction  $T_4$  also wants to lock data object  $X$ , then 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 transaction  $T_3$  wants to lock the same data object  $Y$ , it finds  $T_3.donator = \{T_2\}$  and  $donate\_set = \{T_2, T_1\}$ ; therefore, transaction  $T_3$  must wait. If transaction  $T_2$  commits and transaction  $T_3$  tried to lock data object  $Y$  again, it finds  $T_3.donator = \{ \}$ ,  $T_3.locked\_obj = \{ X \}$  and  $donate\_set = \{T_1\}$ ; Therefore, it waits again. Note that 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.

Note that while in altruistic locking, a transaction can lock a data object only if  $tran.donator = donate\_set$ ; that is why we relax *Altruistic Locking Rule 2* to *Improved Altruistic Locking Rule 2*. 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 allow transaction  $T_1$  to lock data object  $Y$  in altruistic locking, a problem will occur 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. This problem is avoided in our approach, since transaction  $T_1$  cannot commit until transactions  $T_2$  and transaction  $T_3$  are committed.

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*, i.e., it is always in the wake of some other transactions that have donated those data objects which are locked by the requesting transaction, 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 copy of donated object has been reached.

For Cases 2 and 3, when a transaction wants to lock the last object which does not have a new copy, it will then check the compatibility matrix of Fig. 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*; that is, *WakeSet* is decided. Then, we 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* that records those data objects which are locked by transaction *tran*, 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 Fig. 21, to release an object *u\_object*, it removes itself from the *u\_object.locked\_tran* set and the *donator* set (i.e. *WekeSet*) 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 copy of the donated object has been reached. The *Lock* operation, as shown in Fig. 22, is simply a two-phase locking operation. (Note that to avoid deadlocks which can be caused by

the locking strategy, a *priority-based* locking strategy, like *wait-die* or *wound-wait*, can be imposed.)

## 8. Correctness

The correctness of the proposed donation-based approach to concurrency control for object-oriented database systems is proved in Appendix (Wu, 1995). Basically, our strategy can be considered as a hybrid algorithm which combine the improved altruistic locking protocol and granularity locking for objected-oriented database systems. The correctness of altruistic locking has been proved in (Salem *et al.*, 1994) and the correctness of granularity locking has been proved in (Garza and Kim, 1988), which can also be observed from the compatibility matrix as shown in Fig. 1. Since our strategy has extended altruistic locking, to prove the correctness of our strategy, we will follow the proof of the altruistic locking protocol.

In Appendix, for **Property A.1** and **Property A.4**, they are two-phase locking features and are both true in altruistic locking and our improved altruistic locking strategy. For **Property A.2** and **Property A.3**, they are altruistic locking features and are not changed in our strategy. For **Property A.5**, it is the property of the Altruistic Locking Rule 1 (i.e. the Improved Altruistic Locking Rule 1); therefore, the property is true in our strategy. From **Property A.6** to **Property A.14**, they are new properties in our strategy. **Lemma A.2**, **Lemma A.3** and **Lemma A.4** are unique Lemmas in improved altruistic locking, as compared to altruistic locking.

## V. Performance

We compare the performance of our improved altruistic locking with that of altruistic locking using a simulation model which is adapted from the model used in (Agrawal *et al.*, 1987). Specifically, we have used the physical and logical system models from that paper. The simulation model was implemented using the SIMPAS discrete-event simulation package.

The simulation is based on the closed-queueing model illustrated in Figure 23. A set of *num\_terminals* terminals supply transaction requests to a transaction-processing system in which a fixed maximum number of transactions (the multiprogramming level, *mpl*) may

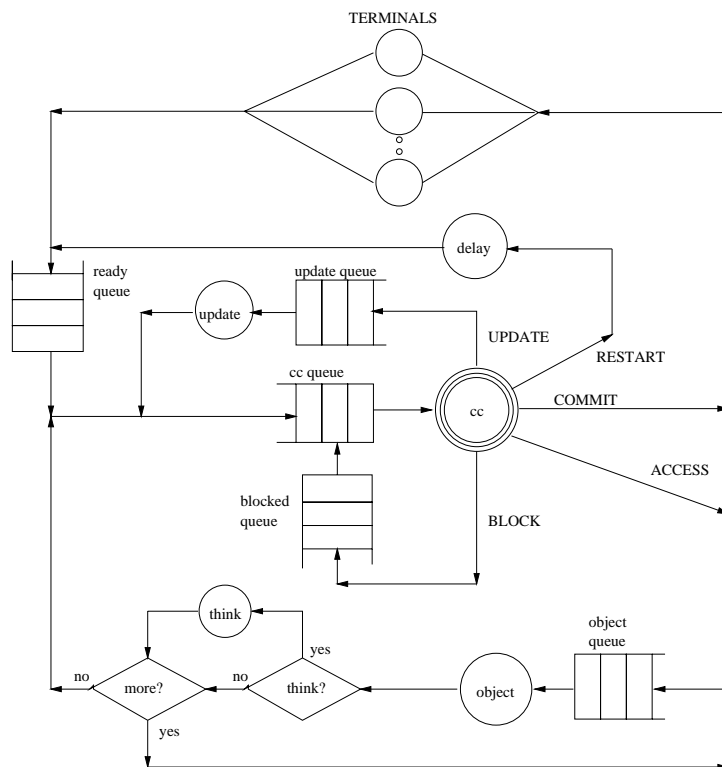


Figure 23: Logical queuing model.

be active. A transaction is considered active if it is either receiving service or waiting for service inside the database system. Requests which cannot be processed immediately are maintained in a FIFO ready queue until enough active transactions have been completed. (Note that transactions in the ready queue are not considered active.)

The transaction then enters the *cc queue* (concurrency control queue) and makes its first concurrency control request. If the concurrency control request is granted, the transaction proceeds to the *object queue* and accesses its first object. When the next concurrency control request is required after an *int\_think\_time* delay, the transaction reenters the concurrency control queue and makes the request. To prevent deadlock, we apply the *Wait-Die approach*. (That is, when a transaction *A* requests a data object that is already locked by another transaction *B*, transaction *A* waits if it is older than transaction *B*; otherwise, it dies, i.e., it is rolled back and automatically retried.) Therefore, transactions may need to be restarted because of concurrency control problems. Such transactions undergo a delay *restart\_delay* before rejoining the ready queue. (Note that a restart does not affect a

transaction's read and write sets.) If the result of a concurrency control request is that the transaction must block, it enters the *blocked queue* until it is once again able to proceed (after a *blocking* delay). Eventually the transaction may complete and the concurrency control algorithm may choose to commit the transaction. If the transaction is read-only, it is finished. If it has written one or more objects during its execution, it first enters the *update queue* and writes its deferred updates into the database. Basically, a deferred-update model is used, meaning that any updates performed by the transaction are actually performed at commit time. Completed transactions undergo a delay of *ext\_think\_time* at the terminals before being resubmitted to the system.

Underlying this logical model is the physical model shown in Figure 24. Each logical operation may require use of one or more of the resources in the underlying physical model. In particular, each object read or write access requires *obj\_io* and *obj\_cpu* time at the disk and CPU servers, respectively. Each write access also requires *obj\_io* when the writing transaction commits (because updates are deferred). Each disk request is served by a randomly selected disk. All of the CPU servers are identical and share a common queue. Jobs in the queue are dispatched to the next available CPU server. Queuing disciplines are FCFS for both the CPU and the disks.

A transaction is modeled according to the number of objects that it reads and writes. The parameter *tran\_size* is the average number of objects read by a transaction, the mean of a uniform distribution between *min\_size* and *max\_size* (inclusive). These objects are randomly chosen (without replacement) from among all of the objects in the database. The probability that an object read by a transaction will also be written is determined by the parameter *write\_prob*. The size of the database is assumed to be *db\_size*. As in (Agrawal et al., 1987), we assume that the cost of concurrency control operations (lock and unlock) is negligible compared to the cost of accessing objects. This means that we are ignoring the direct costs of locking and unlocking objects, as well as related costs, such as conflict-induced context switches.

Our primary performance metrics are *transaction throughput* and *response time*. Transaction throughput rate is the number of transactions completed per second. Transaction response times are measured from the time of entry into the ready queue to the time of

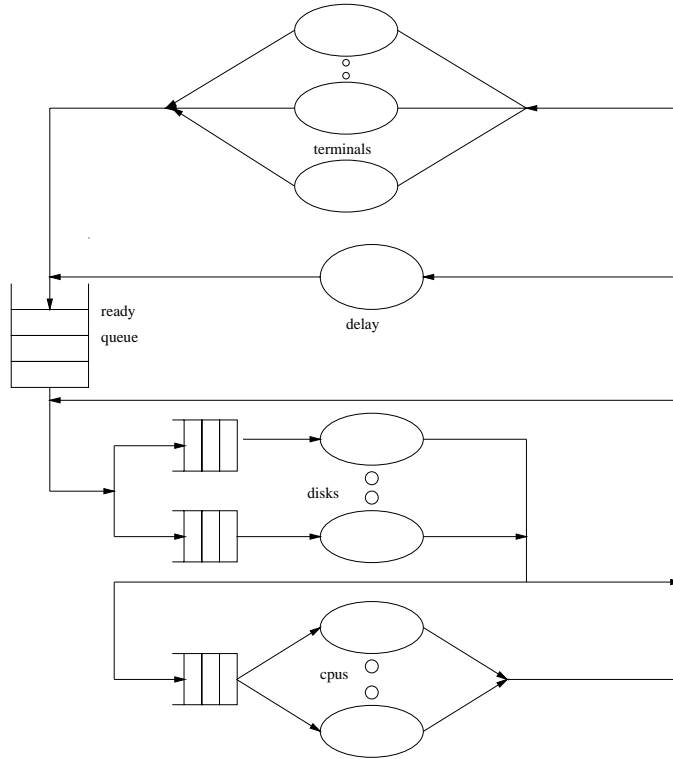


Figure 24: Physical queuing model.

return to the terminals, including any time spent waiting in the ready queue, time spent before (and while) begin restart, and so forth.

Table 1 summarizes the model parameters, the related meaning and the range of per value used in our simulation. Following the altruistic locking simulations, we also assume that each transaction donates each object once each has been locked and accessed. Figure 25 and Figure 26 show the simulation results, where AL denotes the altruistic locking strategy and IAL denotes our improved altruistic locking strategy. From these figures, we show that our improved altruistic locking strategy can have a higher throughput and a shorter response time than the altruistic locking.

## VI. Conclusion

The mechanisms which solve the problems of long transactions can be divided to two kinds of mechanisms, one kind is extending serializability-based mechanisms, another is relaxing serializability mechanisms. In this paper, we have proposed a donation-based strategy

Parameter	Meaning	Value
db_size	Number of objects in the database	1000
tran_size	Mean size of transaction	8
max_size	Size of largest transaction	12
min_size	Size of smallest transaction	4
write_prob	Pr(write X  read X)	0.25
int_think_time	Mean intratransaction think time	0.1
restart_delay	Mean transaction restart delay	0.1
num_terminals	Number of terminals	100
mpl	Multiprogramming level	10, 20, 30, 40, 50, 60, 70, 80, 90, 100
ext_think_time	Mean time between transactions	0.5
obj_io	IO time for accessing an object	0.1
obj_cpu	CPU time for accessing an object	0.01
blocking	Blocking delay	0.1
abort_rate	Aborting rate	0.2

Table 1: Parameters.

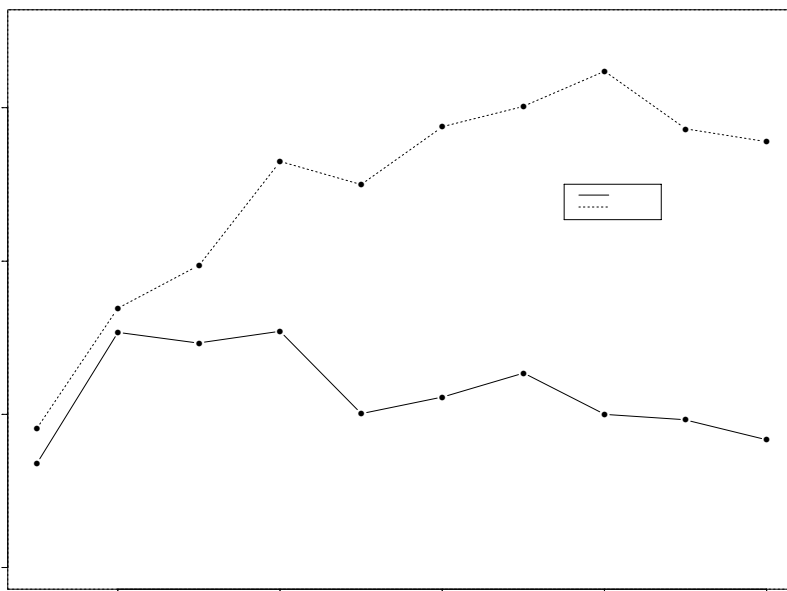


Figure 25: Throughput.

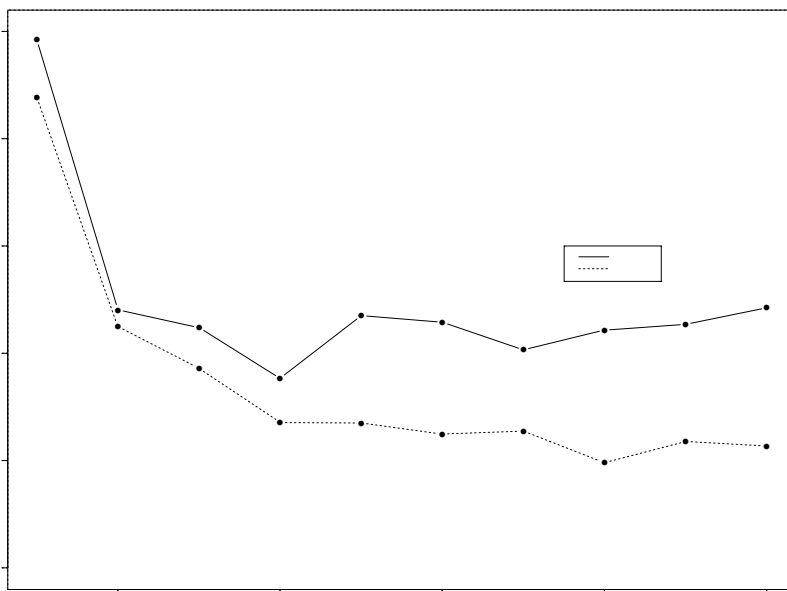


Figure 26: Response time.

which belongs to extending serializability-based mechanisms to solve the problems of long transactions for object-oriented database systems. Basically, our proposed protocol can be considered as a hybrid approach which combines the locking method in ORION object-oriented database system and altruistic locking. 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. Second, when one of the transactions  $T_k$  which participates in the donation of a certain data object aborts, all of those transactions  $T_i$  which participate in the donation of the same data object must abort, too, no matter transaction  $T_i$  occurs before or after transaction  $T_k$  in the donation relationship. 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. From our simulation study, we have shown that our proposed approach can have a higher throughput and a shorter response time than altruistic locking. How to apply the relaxing serializability mechanisms for object-oriented database systems is the future research direction.

## Acknowledgment

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

## References

- Agrawal, R., M. J. Carey and M. Livny (1987) Concurrency control performance modeling: alternatives and implementations. *ACM Trans. on Database Systems*, **12**(4), 609-654.
- Banerjee, J., W. Kim and K. Kim (1988) Queries in object-oriented databases. *Proc. of the 4th Int. Conf. on Data Eng.*, Baltimore, MD, U.S.A.
- Barghouti, N.S. and G.E. Kaiser (1991) Concurrency control in advanced database applications. *ACM Computing Surveys*, **23**(3), 269-317.
- Beeri, C., P.A. Bernstein and N. Goodman (1989) A model for concurrency in nested transactions systems. *Journal of ACM*, **36**(2), 230-260.
- Bernstein, P. A., V. Hadzilacos and N. Goodman (1987) *Concurrency control and recovery in database systems*, Addison-Wesley, Reading, MA, U.S.A.
- Farrag, A.A. and M.T. Ozsu (1989) Using semantic knowledge of transactions to increase concurrency. *ACM Trans. on Database Systems*, **14**(4), 503-525.
- Garcia-Molina, H. (1983) Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. on Database Systems*, **8**(2), 186-213.
- Garcia-Molina, H. and K. Salem (1987) Sagas. *Proc. of ACM SIGMOD Conf.*, Washington, DC, U.S.A.
- Garza, J.F. and W. Kim (1988) Transaction management in an object-oriented database system. *Proc. of ACM SIGMOD Conf.*, Ithaca, NY, U.S.A.
- Hurson, A.R. and S.H. Pakzad (1993) Object-oriented database management systems: evolution and performance issues. *IEEE Computer Magazine*, **26**(2), 48-60.
- Kim, W., R. Lorie, D. McNabb and W. Plouffe (1984) A transaction mechanism for engineering design databases. *Proc. of the 10th Conf. on Very Large Data Base*, Temple, AZ, U.S.A.
- Klahold, P., G. Schlageter, R. Unland and W. Wilkes (1985) A transaction model supporting complex applications in integrated information systems. *Proc. of ACM SIGMOD Conf.*, Ontario, Canada.

- Korth, H.F. and G.D. Speegle (1988) Formal model of correctness without serializability. *Proc. of ACM SIGMOD Conf.*, Minneapolis, MN, U.S.A.
- Lee, S. Y. and R. L. Liou (1996) A multi-granularity locking model for concurrency control in object-oriented database systems. *IEEE Trans. on Knowledge and Data Eng.*, **8**(1), 1996.
- Pan, W. W. and W. P. Yang (1994) The design and implementation of an easy to use and efficient object-oriented database system (EODB). *Proc. of International Computer Symposium*, Hsinchu, Taiwan, R.O.C.
- Pu, C., G. E. Kaiser and N. Hutchinson (1988) Split-transactions for open-ended activities. *Proc. of the 14th Conf. on Very Large Data Base*, Toledo, Spain.
- Salem, K., H. Garcia-Molina and J. Shands (1994) Altruistic locking. *ACM Trans. on Database Systems*, **19**(1), 117-165.
- Shah, P. and J. Wong (1994) Transaction management in an object-oriented data base system. *Journal of Systems Software*, 115-124.
- Woelk, D. and W. Kim (1987) Multimedia information management in an object-orient database system. *Proc. of the 13th Conf. on Very Large Data Base*, Ottawa, Canada.
- Wu, C. C. (1995) A donation-based approach to concurrency control for object-oriented database systems. Master Thesis, Dept. of Applied Math, National Sun Yat-Sen Univ., Kaohsiung, Taiwan, R.O.C.

## APPENDIX

### Correctness of the Proposed Approach

In this Appendix, we first show that schedules of well-formed two-phase transactions that comply with the improved altruistic locking rules are serializable. Then, we show the correctness of the proposed approach to concurrency control for object-oriented database systems. We will adopt the transaction model and notation used in Bernstein *et al.* (1987). We will use  $o_i[x]$  to stand for a "generic" operation (either a read or a write). The subscripts are used to indicate which transaction performs the operation, while the bracketed letters indicate which data item is being operated on. Locking operations are indicated by  $ol_i[x]$ , unlock operations are represented by  $u_i[x]$ , donation operations are represented by  $d_i[x]$ , commit operations are represented by  $c_i$  and abort operations are represented by  $a_i$ .

A transaction  $T_i$  is a partially ordered collection of read, write, and release operations, plus either an abort operation,  $a_i$ , or a commit operation,  $c_i$ , but not both. We will use  $<_i$  to denote the partial order for transaction  $T_i$ . All of  $T_i$ 's read and write operations precede its abort or commit.

A *complete history*,  $H$ , over a set of transactions is a partial ordering,  $<_u$ , of all of the operations of those transactions such that the individual orderings of each of the transactions are preserved. In addition, we require that if two operations,  $o_i[x]$  and  $o_j[x]$  in  $H$  conflict, then either  $o_i[x] <_H o_j[x]$ , or  $o_j[x] <_H o_i[x]$ . Operations conflict if they operate on the same item, and at least one of them is a write. A *history* is a prefix of a complete history.

The *committed projection*,  $C(H)$ , of a history  $H$ , is obtained by deleting all operations of aborted transactions from  $H$ . A *serial history*,  $H_s$ , is a history in which the operations of different transactions are not interleaved. Two histories are equivalent if they are defined over the same transactions and if they order all conflicting pairs of operations in the same way. A history is serializable if its committed projection is equivalent to a serial history.

#### A.1 Definitions

We will rephrase our definitions of well-formedness and of the improved altruistic locking rules in terms of the model. If the transactions in a complete history,  $H$ , are well formed, then  $H$  has the following properties:

**Property A.1.** If  $o_i[x]$  is in  $H$ , then  $ol_i[x]$  and  $u_i[x]$  are also in  $H$ , and  $ol_i[x] < o_i[x] < u_i[x]$  (Salem *et al.*, 1994). (That is, objects must be locked before and unlocked after they are used.)

**Property A.2.** If  $d_i[x]$  and  $o_i[x]$  are in  $H$  then  $o_i[x] < d_i[x]$  (Salem *et al.*, 1994). (That is, objects cannot be read or written by a transaction once it has donated them.)

**Property A.3.** If  $d_i[x]$  is in  $H$  then  $u_i[x]$  is in  $H$  and  $d_i[x] < u_i[x]$  (Salem *et al.*, 1994). (That is, donated objects are eventually unlocked.)

**Property A.4.** If  $ol_i[x]$  and  $u_i[y]$  are in  $H$ , then  $ol_i[x] < u_i[y]$  (Salem *et al.*, 1994). (That is, the two-phase requirement: all of a transaction's locks precede all of its unlocks.)

The first improved altruistic locking rule can be expressed as follows.

**Property A.5.** If  $o_i[x]$  and  $o_j[x] (i \neq j)$  are conflicting operations in  $H$ , and  $o_i[x] < o_j[x]$ , then either  $u_i[x] < ol_j[x]$ , or  $d_i[x]$  exists in  $H$  and  $d_i[x] < ol_j[x]$  (Salem *et al.*, 1994). (That is, transactions cannot simultaneously hold conflicting locks unless one has been altruistically donated.)

Next, we can formalize the notion of wakes.

**Definition A.1 (In the Wake).** An operation  $o_j[x]$  is *in the wake of* transaction  $T_i$  if  $d_i[x]$  exists in  $H$ , and  $d_i[x] < o_j[x] < u_i[x]$  (Salem *et al.*, 1994).

**Property A.6.** If  $ol_i[x]$  and  $c_i$  are in  $H$ , then  $ol_i[x] < c_i$ . (That is, objects must be locked before they are committed.)

**Property A.7.** If  $ol_i[x]$  and  $a_i$  are in  $H$ , then  $ol_i[x] < a_i$ . (That is, objects must be locked before they are aborted.)

**Property A.8.** If  $d_i[x]$ ,  $ol_j[x]$  and  $c_j$  are in  $H$ , and  $d_i[x] < ol_j[x]$ , then  $c_i$  is in  $H$  and  $c_i < c_j$ . (That is, if transaction  $T_j$  locks the object which has been donated by transaction  $T_i$ , transaction  $T_j$  will not commit until transaction  $T_i$  has committed.)

**Property A.9.** If  $d_i[x]$ ,  $ol_j[x]$  and  $a_i$  are in  $H$ , and  $d_i[x] < ol_j[x]$ , then  $a_j$  is in  $H$  and  $a_i < a_j$ . (That is, if transaction  $T_j$  locks the object which has been donated by transaction  $T_i$ , the abortion of transaction  $T_i$  will cause the abortion of transaction  $T_j$ .)

**Property A.10.** If  $c_i$  is in  $H$ , then for every data object  $x$  or  $y$  which is locked by transaction  $T_i$ ,  $u_i[x]$  is in  $H$  and  $c_i < u_i[x]$  and there is no other operation between  $c_i$  and  $u_i[x]$  or exists some  $u_i[y]$  in  $H$  and  $c_i < u_i[y] < u_i[x]$ . (Note that an unlock operation is added before the end of the commit procedure and the system follows the strict two-phase locking.)

**Property A.11.** If  $a_i$  is in  $H$ , then for every data object  $x$  or  $y$  which is locked by transaction  $T_i$ ,  $u_i[x]$  is in  $H$  and  $a_i < u_i[x]$  and there is no other operation between  $a_i$  and  $u_i[x]$

or exists some  $u_i[y]$  in  $H$  and  $a_i < u_i[y] < u_i[x]$ . (Note that an unlock operation is added before the end of the abort procedure and the system follows the strict two-phase locking.)

By combining **Property A.8** and **Property A.10**, we get the the following property.

**Property A.12.** If  $d_i[x]$ ,  $ol_j[x]$ ,  $c_i$  and  $c_j$  are in  $H$ , and  $d_i[x] < ol_j[x]$ , then  $u_i[x]$  and  $u_j[x]$  are in  $H$  and  $u_i[x] < u_j[x]$ . (That is, if transaction  $T_j$  locks a data object  $x$  which has been donated by transaction  $T_i$ , and transaction  $T_i$  commits, then transaction  $T_i$  will unlock data object  $x$  before transaction  $T_j$  unlocks data object  $x$ .)

By combining **Property A.9** and **Property A.11**, we get the the following property.

**Property A.13.** If  $d_i[x]$ ,  $ol_j[x]$ ,  $a_i$  and  $a_j$  are in  $H$ , and  $d_i[x] < ol_j[x]$ , then  $u_i[x]$  and  $u_j[x]$  are in  $H$  and  $u_i[x] < u_j[x]$ . (That is, if transaction  $T_j$  locks a data object  $x$  which has been donated by transaction  $T_i$ , and transaction  $T_i$  aborts, then transaction  $T_i$  will unlock data object  $x$  before transaction  $T_j$  unlocks data object  $x$ .)

**Definition A.2 (WakeSet).** A transaction  $T_i$  is in a set called  $WakeSet_j$ , if  $ol_i[x]$ ,  $d_i[x]$  and  $ol_j[x]$  exists in  $H$ ,  $d_i[x] < ol_j[x] < u_i[x]$  and there is no data object  $y$  such that  $ol_j[y] < ol_j[x]$ .

**Definition A.3 (Completely in the WakeSet).** A transaction  $T_j$  is completely in  $WakeSet_j$  if each its operations is in the wake of a transaction  $T_i$ , where  $T_i \in WakeSet_j$ .

The second improved altruistic locking rule can be expressed as follows.

**Property A.14.** If  $o_i[x]$ ,  $d_i[x]$  and  $o_j[x]$  exist in  $H$  and  $d_i[x] < o_j[x] < u_i[x]$  and  $o_i[x]$  and  $o_j[x]$  conflicts, then for every  $o_j[y]$  in  $H$ , either:

- $o_j[y]$  is in the wake of  $T_b$ , where  $T_b \in WakeSet_j$  or
- there exists  $u_i[y]$  in  $H$  such that  $u_i[y] < o_j[y]$ .

**Definition A.4 (IALT Histories).** An improved altruistic locking protocol (IALT) history is a history with properties A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8, A.9, A.10, A.11, A.12, A.13 and A.14.

## A.2 Serializability of IALT Histories

We are now ready to prove that **IALT** histories are serializable. To do so, we will make use of the serializability theorem of Bernstein *et al.* (1987). This theorem states that a history  $H$  is serializable iff its *serialization graph*,  $SG(H)$ , is acyclic. The graph  $SG(H)$  is a directed graph with a node for each committed transaction in  $H$ , and an arc  $T_i \rightarrow T_j$  if an operation from  $T_i$  precedes and conflicts with an operation of an  $T_j$  in  $H$ .

In our proof, we will distinguish between different types of arcs in the serialization graph. To this end, we introduce two final bits of notation. First, if a transaction  $T_j$  is completely in  $WakeSet_j$  and exists a transaction  $T_i$ ,  $T_i \in WakeSet_j$ , we will denote this by  $T_i \rightarrow_w T_j$ . Secondly, we define a relation based on transactions' crest points.

**Definition A.5 (Crests Before).** A transaction  $T_i$  *crests before* a transaction  $T_j$  in a schedule  $H$  if  $T_i$  unlocks some object before  $T_j$  locks some object, i.e., if  $u_i[x]$  and  $ol_j[y]$  exist in  $H$  and  $u_i[x] < ol_j[y]$ . (Note that, we will use  $T_i \rightarrow_u T_j$  to indicate that a  $T_i$  crests before  $T_j$ .)

**Lemma A.1.** *For an IALT history  $H$ ,  $\rightarrow_u$  is a partial order.*

**Proof.** Consider three transactions such that  $T_i \rightarrow_u T_j \rightarrow_u T_k$ .  $T_i$  unlocks some object before  $T_j$  locks some object.  $T_j$  unlocks some object before  $T_k$  locks some object. By applying **Property A.4** to  $T_j$ , we conclude that  $T_i \rightarrow_u T_k$  and the precedes relation is transitive. If  $T_i \rightarrow_u T_j$ , then by **Property A.4** all locks of  $T_i$  precede all unlocks of  $T_j$  in  $H$ . If  $T_j \rightarrow_u T_i$ , then some unlock of  $T_j$  precedes some lock of  $T_i$ ; thus the crests-before relation is asymmetric. Finally, because of **Property A.4**, we cannot have  $T_i \rightarrow_u T_i$ ; thus the precedes relationship is not reflexive.

**Lemma A.2.** *If  $T_i \rightarrow_w T_j$ , then for each unlock operation of transaction  $T_i$  and transaction  $T_j$ ,  $u_i[x]$  and  $u_j[x]$ , respectively,  $u_i[x] < u_j[x]$ .*

**Proof.** For all the data objects  $x$  which are accessed by transaction  $T_j$ , they can be divided into two classes: (1)  $o_i[x]$  conflicts with  $o_j[x]$ ; (2)  $o_i[x]$  does not conflict with  $o_j[x]$ . For data objects in class (1), they can be classified into two subclasses according to **Property A.5**: (A)  $u_i[x] < ol_j[x]$ ; (B)  $d_i[x] < ol_j[x]$ . For data objects in class (A), according to **Property A.1**,  $u_i[x] < u_j[x]$ . For data objects in class (B), they can be classified into two subclasses: (a)  $T_i \in WakeSet_j$ ; (b)  $T_i \notin WakeSet_j$ . For data objects in class (a), according to **Definition A.2**,  $d_i[x] < ol_j[x]$ . Moreover, according to **Property A.12** and **Property A.13**, no matter transaction  $T_i$  commits or aborts,  $u_i[x] < u_j[x]$ . For data objects in class (b), according to **Property A.14**,  $u_i[x] < o_j[x]$ , which results in  $u_i[x] < u_j[x]$  based on **Property A.1**. Consequently, if  $T_i \rightarrow_w T_j$ , then all unlocks of  $T_i$  precede all unlocks of  $T_j$  in  $H$ .

**Lemma A.3.** *For an IALT history  $H$ ,  $\rightarrow_w$  is a partial order.*

**Proof.** Consider three transactions such that  $T_i \rightarrow_w T_j \rightarrow_w T_k$ .  $T_i$  donates an object  $x$  before  $T_j$  locks the same object  $x$ .  $T_j$  donates the object  $x$  before  $T_k$  locks the same object  $x$ . By applying **Property A.5** to  $T_j$ , we conclude that  $T_i \rightarrow_w T_k$  and the precedes relation is transitive. If  $T_i \rightarrow_w T_j$ , then by **Lemma A.2**, all unlocks of  $T_i$  precede all unlocks of  $T_j$  in  $H$ . If  $T_j \rightarrow_w T_i$ ,

then some unlock of  $T_j$  precedes some unlock of  $T_i$ ; thus the in the wake relation is asymmetric. Finally, because of **Property A.2**,  $T_1$  cannot access an object after  $T_1$  donates the object, we cannot have  $T_i \rightarrow_w T_i$ ; thus the precedes relationship is not reflexive.

**Lemma A.4.** *If  $T_1 \rightarrow T_2$  is in  $SG(H)$ , then either  $T_1 \rightarrow_u T_2$  or  $T_1 \rightarrow_w T_2$ .*

**Proof.** We assume that  $T_2$  is not in the wake of  $T_1$  and show that this implies  $T_1 \rightarrow_u T_2$ . Because of the arc  $T_1 \rightarrow T_2$ , there must be conflicting operations  $o_1[x] < o_2[x]$  in  $H$ . By **Property A.1**, both transactions lock and unlock  $x$ . By **Property A.5**,  $T_1$  has either donated or unlocked  $x$  before  $T_2$  locks it. In the first case ( $T_1$  unlocks  $x$  before  $T_2$  locks it) we have  $T_1 \rightarrow_u T_2$ . In the second case, object  $x$  is donated (and not unlocked) by  $T_1$  when it is locked by  $T_2$ , so  $T_2$  is in the wake of  $T_1$ . But this means that  $T_1 \rightarrow_w T_2$ .

**Theorem A.1 (Acyclicity Theorem).** *If  $H$  is IALT, then  $SG(H)$  is acyclic.*

**Proof.** We will assume a cycle in  $SG(H)$  and derive a contradiction. Consider a cycle  $T_1 \rightarrow \dots \rightarrow T_{n-1} \rightarrow T_1$  in  $SG(H)$ . There is a partial order exist in  $T_1 \rightarrow T_1$ . By **Lemma A.4**, we know that this partial order must be  $T_1 \rightarrow_u T_1$  or  $T_1 \rightarrow_w T_1$ . But by **Lemma A.1** and **Lemma A.3**, we know that this is impossible, since  $T_1 \rightarrow_u T_1$  and  $T_1 \rightarrow_w T_1$  are partial orders; therefore  $SG(H)$  is acyclic.

**Theorem A.2 (Acyclicity Theorem with Granularity Locking).** *If  $H$  is IALT with granularity locking, then  $SG(H)$  is acyclic.*

Since in our approach to concurrency control for object-oriented database systems, locking an object has to require more locks than the improved altruistic locking protocol, which also will not change the partial order of any pair of operations in the history of improved altruistic locking. Therefore, the partial order of a history will still be pertained in our strategy.