oodOPT: A Semantics-Based Concurrency Control Framework for Fully-Replicated Architecture

YANG Guangxin (杨光信)[1] and SHI Meilin (史美林)[2]

[1]*Bell-Labs Research China, Lucent Technologies, Beijing 100080, P.R. China*

[2]*Department of Computer Science and Technology, Tsinghua University, Beijing 100084, P.R. China*

E-mail: gxyang@acm.org; shi@csnet4.cs.tsinghua.edu.cn

Received October 22, 1999; revised September 20, 2000.

Abstract Concurrency control has always been one of the most important issues in the design of synchronous groupware systems with fully-replicated architecture. An ideal strategy should be able to support natural and flexible human-to-computer and human-to-human interactions while maintaining the consistency of the system. This paper summarizes previous researches on this topic and points out the deficiencies of the existing results. A novel semantics-based concurrency control framework, *oodOPT*, is proposed. The main idea of the framework is to resolve conflicts by utilizing semantics of the operations and the accessed data objects. With this approach, complexities in concurrency control are shifted completely from application developers to the framework. Conflicts among operations on objects with different semantics and the strategies resolving these conflicts are analyzed. After describing the algorithm in full detail, the discussion ends up with a comparison with other related work and some considerations for open problems.

Keywords computer supported cooperative work, groupware, concurrency control, oodOPT, COFFEE, Cova

1 Introduction

To support natural, flexible and reliable human-to-computer as well as human-to-human interactions, synchronous groupware systems often adopt the so-called 'fully-replicated' architecture, where data objects and user operations are equally replicated at all cooperative sites. While responsiveness and reliability can be greatly improved with this approach, consistency maintenance becomes more complex than that in centralized and distributed architectures[1], e.g., DBMSs and OSs. The complexities originate largely from the differences between the concurrency control frameworks (CCF), as shown in Table 1.

Table 1. Differences between CCFs in Centralized and Fully-Replicated Architectures

	Centralized	Fully-replicated
Architecture	• One site as the coordinator • Multiple threads	• Multiple peer sites, with no coordinator • One thread at each site
Objects being operated	• Simple in structural and operational semantics, e.g., w/r on a relation or file	• Complex and diverse in their semantics • e.g., set, bag, list, array, tree, graph, etc.
User interface vs CCF	• Loosely coupled • Operations are often collected first and submitted to CCF in batch mode as a transaction	• Tightly coupled • Operations are submitted to CCF immediately after they are generated
Goals of CCF	• Throughput • Atomic, consistency, isolation, and duration properties of transactions	• Responsiveness, naturalness, reliability • Consistency of causal dependence, operation results, and final object states

From the comparisons, we can see that issues in maintaining system consistency would be quite different in the two distinct architectures. Traditional strategies, such as lock and timestamp-based ones,

The work was supported by the National Natural Science Foundation of China under grant No.69773029 and the National '863' High-Tech Programme of China (Nos.863-306-ZD-10-2B, 863-306-ZD-02-03-1).

often lead to sticky and/or unnatural user interfaces when they are explored in real-time groupware systems[2]. Therefore, it is natural and a great challenge to develop more advanced strategies.

The challenge has attracted many research efforts and interests on this topic in the last decade. Many approaches were proposed and implemented. The most promising one would be *dOPT* implemented in GROVE, a cooperative text editor by Ellis and Gibbs[3]. *dOPT* increases greatly the naturalness and flexibility of interactions by exploring the semantics of the data objects and the operations defined on them.

This paper describes *oodOPT*, a semantics-based concurrency control framework for fully-replicated architectures. *oodOPT* follows *dOPT* in the sense that they both take a semantics-based approach. It generalizes *dOPT* based on the object model of *Cova*, a programming language for developing cooperative applications[4,5]. Firstly, related work on this topic and their deficiencies are discussed. Then it comes to our approach to address this problem. The concurrency control framework will then be discussed in detail. Finally we compare *oodOPT* with other related work and present the conclusions.

2 Related Work

Research on advanced CCF for fully-replicated architecture was motivated by various types of co-authoring systems. Due to the differences in application areas and user requirements, strategies for controlling concurrent operations vary widely in their complexity and efficiency. What these strategies achieve, such as the granularity of concurrency operations, the naturalness and flexibility of interactions, etc., also vary widely. Greenberg classifies these strategies into two categories: optimistic and pessimistic[2]. Generally, pessimistic algorithms lead to sticky and limited user interfaces, while optimistic algorithms have the user interfaces changed unnaturally. These make them far from supporting natural and flexible interactions.

dOPT executes operations immediately at the generating site so that the interface responds quickly. They will then be multicast to other cooperative sites so that the cooperators are aware of each other's activities. This leads to two problems. The first one is *causal violation*. Due to unpredictable network latency, an operation o_2, whose generation depends on another operation o_1, may arrive at a site before the arrival of o_1. If o_2 (the result) is executed prior to o_1 (the cause), then the causal dependency will be violated.

To maintain causal dependencies, a state vector $\boldsymbol{V} = \langle v_1, v_2, \ldots, v_n \rangle$ is introduced, where v_i is the number of executed operations generated at the ith site. For two state vectors $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$, $\boldsymbol{V}_1$ is *less than* $\boldsymbol{V}_2$ (denoted as $\boldsymbol{V}_1 < \boldsymbol{V}_2$) if each element of $\boldsymbol{V}_1$ is not greater than the corresponding element of $\boldsymbol{V}_2$ and there exists at least one *unequal* element. Every site maintains a local state vector (LSV) whose initial value is all zeros. After executing an operation from the i-th site, the i-th element of the LSV will be increased by one. An operation is multicast together with the LSV of the generating site. When an operation is to be executed at a site, the state vector associated with it will be checked against the site's LSV. Only operations whose state vector is *less than* or *equal to* the LSV can be executed. It can be proved that causal dependency can be achieved when operations are executed in this way[6].

For two operations, if neither one is causally dependent on the other, they are said to be *concurrent*. According to the above scheduling schema, concurrent operations can be executed in any order. This leads to the second consistency problem. For example, suppose the initial string is '*abcd*'. o_1 and o_2 are concurrent. o_1 inserts a '1' at the 2nd position and o_2 deletes the character at the 3rd position. If o_1 is executed first, the final string becomes '*a*1*cd*'. If o_2 is executed first, the final string becomes '*a*1*bd*'. Since both execution orders are possible, there is no guarantee that the final objects at two sites are identical.

The problem comes from the fact that an operation may make the site object's state different from the state depending on which the concurrent operation was generated. By executing an operation in its original form, the effects caused by executed concurrent operations are completely neglected. This is exactly the root of the second type of inconsistency.

To maintain the consistency, *dOPT* transforms the operation to be executed against executed concurrent operations with the so-called transformation functions (TF). The inputs to a TF are two concurrent operations, one to be executed and one executed. The return value is a new operation, which is obtained by utilizing the semantics of the two input operations and the objects being operated. For example, in the case described above, if o_1 is executed first, then the position parameter of o_2 will be adjusted to 4, reflecting the fact that o_1 has moved the characters following 'a' one position rearwards.

dOPT transforms the operation to be executed against each concurrent operation in the operation log one by one. However, it fails in the case of partial concurrency, where the operation to be transformed and the operation used for transformation are not generated according to the same object state, which violates the precondition required by the transformation functions.

To solve the partial concurrency puzzle, several enhancements of *dOPT* were published in the last decade. The adOPTed[7] algorithm proposed by Reseel *et al.* replaces the linear operation log of *dOPT* with an *interaction model*, which is a directed graph with the state vectors being its vertices and the operations being its edges. Suleiman uses the "forward and backward" transformation functions[6], Sun *et al.* utilizes the "inclusion and exclusion" transformation functions[8,9] respectively to ensure that the sequence of operations for transformation has the same context as the operation to be transformed.

These enhancements differ in time and space complexities. However, all of them are based on similar ideas, i.e., to satisfy the pre-conditions required by the transformation functions that the operation pair supplied to the TF should be generated from the same object state.

2.1 Deficiencies

The most promising aspect of *dOPT* lies in its semantics-based approach, which resolves the conflicts among concurrent operations with the semantics of the operations and the object being operated. Although it is declared that the algorithms mentioned above could be used as a general purpose CCF for fully-replicated architecture, there is much work to approach this goal. The cause lies in the specialization of the semantics of the object being operated and the operations defined on it. The structure of the object handled by these algorithms is a linear list and there are only two operations, i.e., *insert* and *delete* defined on them. The reason why these algorithms could not be used as a general purpose CCF lies in the following aspects.

First, a linear list consisting of only simple characters is not enough for practical real-time systems. Data structures with much richer semantics are often required to model the object to be shared in the real-time session. Even for a practical real-time text editor, other structures are required to describe the hierarchy of *documents*, *paragraphs*, *sentences*, *words*, and *characters* as well as the formats of the objects at different levels.

The second problem rises from the fact that transformation functions are application-specific and should be designed and implemented from scratch for each system. As the complexities of the objects and operations increase, this work is by no means an easy task[10]. At the same time, the diversity of programming languages and platforms makes it hard to reuse the design and implementation, which leads to lots of duplicated work and low efficiency.

The third one comes from the fact that *user operations* are directly mapped to the *primitive operations* defined on the object. This is rarely the case in practical systems, where user operations are often the combination of primitive operations on objects at different levels. For example, a *move* operation may be implemented by deleting the characters first and then inserting them at the new position. Since the combinations are usually more complex than primitive operations themselves, it is often hard to define TFs among combined operations. The problem becomes even more serious when users are allowed to interact with the system by combining primitive operations in an *ad hoc* manner.

These problems urge us to seek a solution able to shift the difficulties in concurrency control from groupware developers to the system. Our attempts lead to *oodOPT*, a semantics-based CCF, which we will discuss in the following sections.

3 Towards a General Purpose Framework

The first step towards a general-purpose framework is to provide a solid *data description mechanism* that can be used to model the structural and operational semantics of the artifacts shared in a real-time session. *oodOPT* is based on the *Cova* Object Description Language (CODL), which implements an extended version of the Object Model[11] proposed by ODMG and a self-contained programming language for implementing the operations of objects. CODL is a pure object-oriented language with its syntax similar to Java. Besides, four additional commands, *foreach*, *insert*, *delete*, and *update*, are implemented for manipulating collection objects.

The second step would be identifying the primitive operations defined on different types of objects. These operations would be the only operations that could be used to change or retrieve the state of objects. In CODL, there are *atomic* objects and *collection* objects, i.e., *set*, *bag*, *list*, *array*, *dictionary*, *tree*, *graph*. Operations defined on these objects could be classified into four categories, as listed below.

- A *Reference* returns a value based on the current state of the referenced objects, or navigates through a collection, etc.
- An *Update* changes the state of an object, e.g., by changing the attribute of an atomic object, or replacing the element with a new one at a specific position in a collection.
- An *Insert* inserts a new element into a collection object.
- A *Delete* removes a specific element from a collection object.

Primitive operations may have different forms in CODL. For example, an *update* may be an *assignment*, or an explicit *update* command on a collection. Operations on objects are implemented by combining primitive operations with flow control statements and other language constructs. User operations are translated into operations on objects by user interface modules. Therefore, there are few limitations on the operations available to users. New operations can even be defined dynamically.

An instance of Cova virtual machine (CovaVM) runs at each cooperative site. It maintains an internal object space that contains a copy of the artifact being shared. The UI module passes the names of operations along with the actual parameters to CovaVM, which will decompose the commands of the corresponding methods into primitive operations and apply them to the object. Similar to *dOPT*, operations will be transmitted to and executed at other cooperative sites to achieve awareness.

The third step is then to find a way to execute these primitive operations so that the effects and results of a user operation are identical at all sites. This is because that upon the execution of an operation, the state of objects might have been changed by other concurrent operations. If operations are executed in their original forms, they may produce different results at different sites, as shown in Section 2.

oodOPT is implemented in the Cova virtual machine. Before a primitive operation is executed, it will be transformed against other primitive operations that have been executed on the same object. The transformation is based on the semantics of the primitive operation itself and the semantics of the object it operates. This is feasible because this semantic information is available to CovaVM. If each primitive operation produces an identical result at different sites, the results by their combination at different sites will also be identical.

3.1 Formal Descriptions

Before going into details on how *oodOPT* works, we first give several definitions as the formal descriptions of a synchronous groupware system and its correctness.

Definition 1. *A synchronous groupware system G can be formalized as a tuple $\langle O, S\rangle$, where $O = \langle D, M\rangle$ and $S = \{s_1, s_2, \ldots, s_n\}$. O is the definition of the shared object, D describes its structural semantics and $M = \{m_1, m_2, \ldots, m_k\}$ is a set of methods defined on D. User operations on the object will be translated into calls (denoted as c) to the methods defined on O. The notation c*

will also be used to refer to the user operation thereinafter. S describes the dynamic properties of G. Each s_j in S is a quintuple $\langle o, i, p, \boldsymbol{V}, Q\rangle$, representing a cooperative site. i and p are the identifier and the priority numbers assigned to s_j respectively. They are unique within the scope of S_G. The site object o is an instance of O_G. The state vector $\boldsymbol{V}$ represents the current state of site s_j. The request queue Q contains all unexecuted requests received by site s_j. Each request r is a quaternion $\langle i, \boldsymbol{V}, c, p\rangle$, where c is the operation, i and p are the identifier and the priority number of the source site where c was generated, and $\boldsymbol{V}$ is the state vector of the source site when c was generated.

For a groupware system G, the period during which an object is opened for sharing is called a *session*. A session may be divided into multiple *stages*. A session goes into a different stage when the object definition O or the number of sites in S changes.

Definition 2 (Consistency Model of a Groupware System). *If the following three conditions are always satisfied during each stage of G, G is said to be consistent.*

1. *Consistency of Causal Dependency C_A: given two operations c_1 and c_2, if c_2 is generated based on the state produced by c_1, then the execution order of c_1 must be before that of c_2 at any site.*

2. *Consistency of Operation Results C_B: for each operation c, the actual results produced by its execution at other sites must always be identical to that produced by its local execution.*

3. *Consistency of Final States C_C: after the operations generated by all sites are executed at every site in G, all site objects must be logically equivalent (see Definition 3 for a formal description).*

We name this consistency model as *COFFEE*, which can be regarded as a coordinator to the *ACID* model that should be followed by a centralized transaction manager. The *COFFEE* model is defined here to set a goal for *oodOPT*. In fact, some of the conditions can be relaxed in some cases. For example, in a free style brain storming, users may not care whether their final documents are identical. However, in some other cases, application specific constraints may be imposed on the shared object. These constraints may be violated by concurrent operations, although any operation alone does not violate them. This is another source leading to inconsistency. However, we will not address this type of inconsistency in this paper.

Definition 3 (Logic Equivalence between Data Objects). *Given two Cova objects, o_1 and o_2, they are said to be logically equivalent iff:*

1. *Both o_1 and o_2 are of the same data type and have an equal value; or,*

2. *Both o_1 and o_2 are of the same collection type and each element in one collection has a logically equivalent element with a logically equivalent index (a position or a key, if possible) in another collection; or,*

3. *Both o_1 and o_2 are of the same Cova class type and each attribute of one object is logically equivalent with the corresponding attribute of another object.*

In Definition 3, literal values, attributes of objects, elements and indexes of collections are uniformly regarded as objects. To simplify the discussion, an equal-sign (=) is used to denote the logic equivalence relationship. When o_1 is not logically equivalent to o_2, we denote it as $o_1 \neq o_2$.

According to this definition, it is obvious that '*a*' = '*a*', list⟨ '*a*', '*b*', '*c*'⟩ = list⟨'*a*', '*b*', '*c*'⟩, while list⟨'*a*', '*b*', '*c*'⟩ ≠ set⟨'*a*', '*b*', '*c*'⟩ because they are not of the same collection type. Similarly, list⟨'*a*', '*b*', '*c*'⟩ ≠ list⟨'*a*', '*c*', '*b*'⟩, for the second element of the first list '*b*', is not logically equivalent to the element '*c*', whose index is equivalent to that of '*b*' in the first list.

When an operation c on an object o is executed, it may cause two different types of effects:

1. An object is returned to the caller. We use $o.c$ to denote the returned object.

2. The state of o is changed. o:c is used to denote the modified object.

Definition 4 (Conflict between Operations). *Given two concurrent operations c_1 and c_2, generated by two different sites in the same real-time groupware system G, if at any site, when c_1 and c_2 are executed serially on the site object o in different orders, at least one of the three conditions ① $(o : c_1).c_2 \neq o.c_2$, or ② $(o : c_2).c_1 \neq o.c_1$, or ③ $(o : c_1) : c_2 \neq (o : c_2) : c_1$ is satisfied, then c_1 and c_2 are said to be conflicting.*

For two conflicting operations, executing them in their original forms violates the 2nd and 3rd consistency conditions given in Definition 2. For example, suppose o = list⟨'*a*', '*b*'⟩. Two concurrent operations c_1 and c_2 insert a 'c' at the third position and a 'd' at the first position respectively. Condition ③ will be satisfied when they are executed serially in different orders. Therefore, the two operations conflict. This is exactly one of the cases handled by *dOPT* and its derivations.

Based on these definitions, we will now discuss in detail the *oodOPT* framework.

4 The oodOPT Framework

The name, *oodOPT*, stands for the combination of the object-oriented (*oo*) and transformation-based (*dOPT*) natures of the framework. In *oodOPT* operations are scheduled in a way similar to *dOPT*. It is implemented in CovaVM, which executes the operations by interpreting the statements of the corresponding method definition. For each statement, it will be decomposed into a series of *Cova* instructions, which are the minimum executable unit. Primitive operations on objects are implemented as *Cova* instructions.

This section illustrates how the framework works by discussing the conflicts among primitive operations on different types of objects as well as how these conflicts could be resolved with the semantics of the operations and the objects.

4.1 Atomic Objects

In *Cova*, atomic objects are implemented with user-defined classes. Currently, *Cova* class supports only one type of *properties*, i.e., *attributes*. Attributes of atomic objects can be of literal types, collection object types, or atomic object types. Each attribute is assigned a unique order number as its identifier.

For atomic objects, two primitive operations, i.e., *reference* and *update*, are provided. Both of them operate on a single attribute of an object. For an attribute of object types, the value returned by a reference operation is the identifier of the object referred to by this attribute. Thus, conflict of operations on attributes of literal or non-literal types can be handled in the same way.

Table 1. Conflicts among *Atomic* Operations

	$R_1(i)$	$U_1(i)$
$R_2(i)$	$\times$	$\surd$
$U_2(i)$	$\surd$	$\surd$

Table 1 shows the conflicts among two concurrent operations on an atomic object, where $\surd$ states the two operations conflict, $\times$ states there is no conflict. The parameter i is the order number of the attribute. According to Definition 4, it is easy to verify that an *update* will conflict with concurrent reference and update operations.

To resolve these conflicts, *oodOPT* maintains an operation log (L) for each atomic object. Each item in the log is a sextuple $\langle t, n, v, s, p, \boldsymbol{V}\rangle$, where t is the type of the operation; n is the order number of the attribute operated; v is the value set by this operation; s is the identifier of the site generating the request that contains this operation; p and $\boldsymbol{V}$ are the priority number and the state vector associated with the request respectively.

For atomic objects, two types of operations are logged. One is *normal update*, which is actually executed update operations. The other is *pseudo update*, which is not executed due to conflict resolution. Before an *update* operation is executed on an atomic object, the operation log is searched backwards from the end to the top. If a non-preceding update to the same attribute is found, the algorithm will check whether the two updates are from the same request. If so, the log item is refreshed with the new value. The state of the attribute will also be refreshed if the log item is a normal update. If the two updates are not from the same request, their priorities are checked. If the update to be executed has a lower priority than the log item, it will not be executed and only a pseudo update item is appended to the tail of the log.

To retrieve the value of an attribute, the operation log is also searched. If the log is empty, the initial value of the attribute will be returned. If a log item from the same request or a preceding normal update is found, then the value of the log item will be returned. Otherwise, the value of the latest preceding pseudo update will be returned.

The strategies for executing *update* and *reference* operations on atomic objects are depicted by Algorithm 1 and Algorithm 2 respectively.

Algorithm 1. Execute $(o, u, s, p, \boldsymbol{V})$

```
input o: the atomic object
      u: a tuple ⟨n, v⟩; n: the attribute #; v: the new value
      s: identifier of the site that generates the request
      p: priority number of the site
      V: state vector associated with the request
output none
body
{
    bDuplicated=false; bExecute=true;
    foreach l in L_o (from tail to head)
    {
        if (n_l! = n_u)
           continue;
        if (V_l^{s_l} ≥ V^{s_l})
        { //l and u are concurrent
           if (V_l == V && s_l == s)
           { //multiple updates in the same request
              bDuplicated = true;
              break;
           } else if (p_l > p)
           { bExecute = false;
              break;
           }
        }
    }
    if (bDuplicated)
    { //refresh the log item
       v_l = v_u;
       if (t_l == NORMAL_UPDATE)
          set the value of the n_u-th attribute of o to v;
    } else {
       t = PSEUDO_UPDATE;
       if (bExecute)
       {
          set the value of the n_u-th attribute of o to v;
          t = NORMAL_UPDATE;
       }
       append ⟨t, n_u, v_u, s, p, V⟩ to L_o at its tail;
    }
}
```

Algorithm 2. Execute $(o, r, s, p, \boldsymbol{V})$

```
input o: the atomic object
      r: a tuple ⟨n, v⟩ representing the reference
output Value of the attribute to be retrieved. The value will be stored in v_r too.
body
{
    bFound=false; bFoundPseudo=false;
    foreach l in L_o (from tail to head)
    {
        if (n_l! = n_u)
              continue;
```

```
        if (V_l == V && s_l == s)
        {       bFound=true;
                break;
        } else
        if (V_l^{s_l} < V^{s_l})
        {
                if (t_l == NORMAL_UPDATE)
                { bFound=true;
                  break;
                } else if (!bFoundPseudo)
                {  bFoundPseudo=true;
                   pv = v_l;
                }
        }
    }
    if (bFound)
        v_r = v_l;
    else if (bFoundPseudo)
        v_r = pv;
    else
        v_r = the value of the n_u-th attribute of o;
    return v_r;
}
```

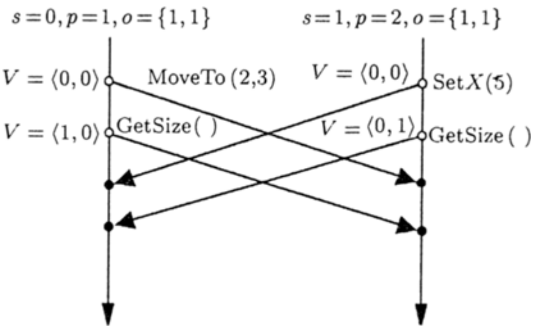

Fig.1. Concurrent operations on an *atomic* object.

We will show how Algorithms 1 and 2 work with an example. Suppose there are two users manipulating concurrently a *point* object with two attributes, i.e., the x and y coordinates. As shown in Fig.1, three methods are used, i.e., *MoveTo*, *SetX*, *GetSize*. The first two are *update* operations, the last one is a *reference* operation.

According to Algorithm 1, after the *MoveTo* is executed at site 0, the operation log of o becomes:

$$\{\langle NORMAL\_UPDATE, 1, 2, 0, 1, \langle 0, 0\rangle\rangle, \langle NORMAL\_UPDATE, 2, 3, 0, 1, \langle 0, 0\rangle\rangle\}.$$

o becomes $\{2, 3\}$. The successive *GetSize* returns 6, which is the product of 2 and 3. At site 1, the operation log becomes $\{\langle NORMAL\_UPDATE, 1, 5, 1, 2, \langle 0, 0\rangle\rangle\}$. o becomes $\{5, 1\}$. A successive *GetSize* returns 5, which is the product of 5 and 1.

When executing *MoveTo*(2, 3) at site 1, two primitive update operations will be generated. One of them is $\langle 1, 2\rangle$. It is concurrent with the one in the log and has a lower priority. Therefore, it is not executed. Only a pseudo update is added to the log. For the second update $\langle 2, 3\rangle$, there is no concurrent operation in the log. Therefore it is executed. The log for o at site 1 now becomes:

$$\{\langle NORMAL\_UPDATE, 1, 5, 1, 2, \langle 0, 0\rangle\rangle, \langle PSEUDO\_UPDATE, 1, 2, 0, 1, \langle 0, 0\rangle\rangle,$$
$$\langle NORMAL\_UPDATE, 2, 3, 0, 1, \langle 0, 0\rangle\rangle\}.$$

o now becomes $\{5, 3\}$. When the request *GetSize*() generated by site 0 is executed at site 1, according to Algorithm 2, the log will be searched. The *reference* to x returns 2, not the current value 5. The *reference* to y returns 3. Therefore, the result returned by this request is 2×3, which is 6. This is exactly the value produced by executing it at site 0. It is easy to verify that when the two operations generated at site 1 are executed at site 0, the consistency conditions C_B and C_C will also be satisfied.

4.2 Set Objects

A set object in *Cova* is an unordered collection that contains multiple member objects of compatible data types. A set object does not allow duplicated members. ODMG defines fourteen methods in the interface of *set* object. All these methods can be expressed by a combination of 3 primitive operations: *navigate, insert*, and *delete*. A *navigate* tries to access each element in the set and does some other calculations according to the current state of these elements. The state of *set* objects may be changed by *inserts* and *deletes*. Therefore, they conflict with *navigate* operations. Conflicts among operations on *set* objects are shown in Table 2.

Table 2. Conflicts among Set Operations

	n_1	i_1	d_1
n_2	×	√	√
i_2	√	×	×
d_2	√	×	×

To resolve these conflicts, an operation log is also maintained for each *set* object. Only *inserts* and *deletes* will be recorded in the log. Each item in the log is a quintuple $\langle t, v, s, p, \boldsymbol{V}\rangle$, where v is now the element inserted or deleted. It can be either a literal value or an object identifier. Other elements in the log item have the same meaning as those of the log item for atomic objects. The strategies to execute *navigates*, *inserts*, and *deletes* on a *set* object are shown as Algorithms 3, 4, and 5.

Algorithm 3. Execute $(o, n, s, p, \boldsymbol{V})$

input o: the *set* object
    n: the *navigate* operation
output A new *set* object whose members can be used for further calculation
body
{
    $o_{\text{new}} = o$;
    foreach l **in** L_o
    {
        if $(\boldsymbol{V}_l^{s_l} \geq \boldsymbol{V}^{s_l}$ && $s_l! = s)$
        { //l and u are concurrent
            if $(t_l ==$ INSERT)
                remove v_l from o_{new};
            else
                insert v_l into o_{new};
        }
    }
    return o_{new};
}

Algorithm 4. Execute $(o, i, s, p, \boldsymbol{V})$

input o: the *set* object
    i: the *insert* operation $\langle I, v\rangle$, where v is the element to be inserted
output *none*
body
{
    foreach e **in** o
    {
        if $(e == v_i)$
            //v_i is already in the set
            return;
    }
    $o+=v_i$;
    append $\langle INSERT, v_i, s, p, \boldsymbol{V}\rangle$ to L_o at its tail;
    return;
}

Algorithm 5. Execute $(o, d, s, p, \boldsymbol{V})$

input o: the *set* object
    d: the *delete* operation $\langle D, v\rangle$, where v is the element to be deleted
output *none*

```
body
{
   foreach e in o
   {
      if (e == v_d)
      {
         remove e from o;
         append ⟨DELETE, v_d, s, p, V⟩
         to L_o at its tail;
         return;
      }
   }
}
```

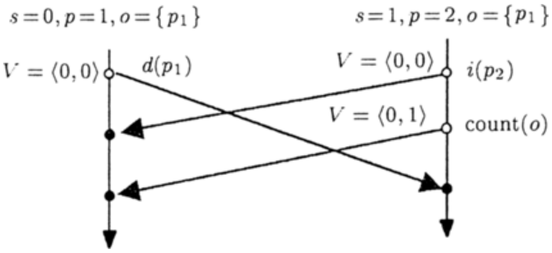

Fig.2. Concurrent operations on a *set* object.

According to Algorithms 4 and 5, it can be seen that *inserts* and *deletes* can be executed in normal ways, for they will never conflict with other concurrent operations recorded in the log. When a *set* object is navigated through, e.g., to count the members, elements inserted by concurrent *inserts* will be omitted, while elements removed by concurrent *deletes* will be included. The resulting *set* object used for navigation will then become logically equivalent to the *set* object navigated by local execution.

This can be further explained by a simple example. As shown in Fig.2, suppose a *set* object o contains only one point p_1 initially. $count(o)$ will return 2 at its local execution. Upon its execution at site 0, the *set* object o contains only p_2 inserted by $i(p_2)$. According to Algorithm 3, p_1 will be inserted back into the *set* object to be navigated, for $d(p_1)$ is concurrent with $count(o)$. Therefore, the resulting object is $\{p_1, p_2\}$, and $count(o)$ at site 0 will return 2 too.

4.3 List Objects

Unlike *set* objects, a *list* object is a structured collection whose members can be accessed via continuous indices starting from zero. This results in more complex operations. Therefore, conflicts among these operations are far more complicated than those for a *set* object.

Besides *navigate*, *insert*, and *delete* operations that are similar to those on *set* objects, another primitive operation *update* can be identified. *inserts* and *deletes* now take an additional parameter which specifies the position of the element being operated. An *update* replaces the element at the specified position with a new element. Conflicts among these operations are shown in Table 3.

Table 3. Conflicts among List Operations

	n_1	$i_1(i)$	$d_1(i)$	$u_1(i)$
n_2	×	√	√	√
$i_2(j)$	√	√	√	√
$d_2(j)$	√	√	√	√
$u_2(j)$	√	√	√	$(i=j)?\surd:\times$

To resolve the conflicts, the operation log for *list* objects now contains all the executed *inserts*, *deletes* and *updates*. Each item in the log is a sextuple $\langle t, x, v, s, p, \boldsymbol{V}\rangle$, where x is the index of the element by this operation. Other elements have the same meaning as those of a log item for *atomic* objects. The algorithm to execute *inserts* and *deletes* is shown as Algorithm 6.

The basic ideas of the algorithm are similar to that proposed by Suleiman in [6]. When executing an operation on a *list*, operations in the log that conflict with it are extracted out to form a special sub-log. At the same time, items in the sub-log will be reordered into two parts. The first part contains all the operations that precede the operation to be executed, while all concurrent operations are in the second part. Then the operation to be executed is transformed against the operations in the second part one by one. Finally, the transformed operation is executed.

Algorithm 6. Tfd (t, x, p, t', x', p')

input t, x, p: type, index, and priority of the operation to be transformed
t', x', p': type, index, and priority of the transforming operation
output New index of the operation to be transformed after transposition
body

```
{
    if (t' == UPDATE) return x;
    switch(t){
    case INSERT:
        switch (t)
        { case INSERT:
                return
                (x == x')?((p > p')?x : x + 1) :
                        ((x > x')?(x + 1) : x);
          case DELETE:
                return (x > x')?(x - 1 : x);
        }
    case DELETE or UPDATE:
        switch (t)
        { case INSERT:
                return (x < x')?(x : x + 1);
          case DELETE:
                return (x == x')?(-1):
                              ((x < x')?x : x - 1);
        }
    }
}
```

Algorithm 7 is another implementation of the backward transposition functions in [6]. The input to this algorithm is also a pair of operations (c_1, c_2), where c_1 is executed before c_2. It is provided for reordering the operations in the sub-log, so that the execution order of the two operations can be exchanged without violating C_B and C_C defined in Definition 2. After transformation, a new pair of operations, (c_2', c_1'), will be generated which satisfies $o : c_2' : c_1' = o : c_1 : c_2$.

Algorithm 7. Tbk (t, x, p, t', x', p')

```
input t, x, p: type, index and priority of one operation
      t', x', p': type, index and priority of another operation
output An index pair (x, x')
body
{
    switch (t){
    case INSERT:
        switch (t)
        {  case INSERT:
              return
              ((x' > x)?x : x - 1, (x' > x)?x' - 1 : x');
           case DELETE:
              return (x' == x)?(-1, -1):
               ((x' > x)?x : x - 1, (x' > x)?x' - 1 : x');
        }
    case DELETE
        switch (t)
        {  case INSERT:
              return (x' == x)?(-1, -1) :
              ((x' > x)?x : x + 1, (x' > x)?x' + 1 : x');
           case DELETE:
              return
              ((x > x')?x + 1 : x', (x > x')?x : x' + 1);
        }
    }
}
```

Algorithm 8 provides the steps to extract concurrent conflicting operations from the log and to reorder them into two separate parts with Algorithm 7. The length of the first part is also returned to facilitate the forward transposition in Algorithm 9. Due to limited space, there will be no examples to show how the algorithm works. Interested readers can refer to [6] to find some interesting examples.

Algorithm 8. Separate $(o, s, p, \boldsymbol{V}, L', n)$

input o: the *list* object
$s, p, \boldsymbol{V}$: the same as defined in Algorithm 1
output L': a special sublist of o containing only *inserts* and *deletes*. All items precede $\boldsymbol{V}$ at the head. Operations concurrent with $\boldsymbol{V}$ are placed at the tail.
n: the number of items that precede $\boldsymbol{V}$
body

```
{
    L' = ∅; n = 0;
    foreach l in L_o (from head to tail)
    {
        if (t_l == UPDATE) continue;
        if (V_l^{s_l} ≥ V^{s_l} && s_l! = s)
            //l is concurrent with V
            append l to L' at its tail;
        else
        for (i = length (L') − 1; i ≥ n; i + +)
        {
            l' = L'[i];
            (x_{l'}, x_l) = Tbk(t_{l'}, x_{l'}, p_{l'}, t_l, x_l, p_l);
        }
        insert l into L' at n;
    }
    n++;
}
```

Based on Algorithms 6 – 8, Algorithm 9 depicts how an *insert* or *delete* operation is executed. The algorithm first splits the operation log into two parts. Then the operation to be executed will be transformed against all the concurrent operations. Finally, the transformed operation is executed and added to the operation log.

Algorithm 9. Execute $(o, m, s, p, \boldsymbol{V})$

input o: the *list* object
m: the *insert* $\langle t, x, v\rangle$, or *delete* $\langle t, x\rangle$, where t is the type of the operation, v is the element to be inserted, x specifies the index of the element
$s, p, \boldsymbol{V}$: the same as defined in Algorithm 1
output *none*
body

```
{
    Separate(o, s, p, V, L', n);
    for (i = n; i < length(L'); i++)
    {
        l' = L'[i];
        x_m = Tfd(t_m, x_i, p, t_{l'}, x_{l'}, p_{l'});
    }
    if (t_m == INSERT)
        insert v_i into o at x_m;
    else
        delete the x_m-th element of o;
    append ⟨t_m, x_m, (t_m = INSERT?v_m : null), s, p, V⟩ to L_o at its tail;
}
```

Algorithms to execute *navigates* and *updates* are similar to Algorithm 3 and Algorithm 1. They are not discussed here due to limited space. Algorithms to resolve conflicts among operations defined on other collection objects can be obtained in similar ways and are not discussed either.

5 Comparison and Conclusion

oodOPT is implemented in the *Cova* runtime system[11], which aims at providing a novel development platform for groupware developers. We have also developed *CovaClient*, a command line tool that can be used to *create*, *open*, and *operate* *Cova* objects. When an object is opened, it will be replicated from *CovaServer* to *CovaClient*. Users operate a *Cova* object by typing the name of one of its methods and required parameters. *oodOPT* functions when an object is being operated by multiple users.

oodOPT outperforms other related work by overcoming deficiencies based on the *Cova* object model and its semantics-based approach. Compared with other CCFs for replicated architecture, such as the lock-based one implemented in Suite[12], *oodOPT* is fully optimistic and avoids the stickyness or unnaturalness. Our semantics-based approach also seems to be applicable to other fields, such as consistency maintenance for data replication in distributed database systems. Future work includes generalizing the concepts of cooperative transactions and introducing rollback facilities into the framework to make it more complete.

References

[1] Ellis C A, Gibbs S J, Rein G L. Groupware: Some issues and experiences. *Communication of ACM*, 1991, 34(1): 39–58.

[2] Greenberg S, Marwood D. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proc. ACM Conf. CSCW*, Chapel Hill, 1994, pp.207–217.

[3] Ellis C A, Gibbs S J. Concurrency control in groupware systems. In *Proc. ACM SIGMOD Conf. Mgmt of Data*, Seattle, 1989, pp.399–407.

[4] Yang Guangxin. Research on meta-groupware — The Cova programming language and system [dissertation]. Tsinghua University, Beijing, 2000.

[5] Yang Guangxin, Shi Meilin. Cova: A programming language for cooperative applications. *Science in China, Series F*, 2001, 44(1): 73–80.

[6] Suleiman M, Cart M, Ferrie J. Serialization of concurrent operations in a distributed collaborative environment. In *Proc. ACM SIGGROUP Conf. Supporting Group Work,* Phoenix, 1997, pp.435–445.

[7] Ressel M, Nitsche-Ruhland D, Gunzenhauser R. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proc. ACM Conf. CSCW*, Cambridge, 1996, pp.288–297.

[8] Sun C Z, Ellis C. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proc. ACM Conf. CSCW*, Seattle, 1998, pp.59–68.

[9] Sun C Z, Jia X H, Zhang Y C *et al.* A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems. In *Proc. ACM SIGGROUP Conf. Supporting Group Work*, Phoenix, 1997, pp.425–434.

[10] Palmer C R, Cormack G V. Operation transforms for a distributed shared spreadsheet. In *Proc. ACM Conf. CSCW*, Seattle, 1998, pp.69–78.

[11] Cattell R G G, Barry D, Bartels D *et al.* The object database standard: ODMG 2.0. San Mateo: Morgan Kaufmann Publishers, 1997.

[12] Muson J, Dewan P. A concurrency control framework for collaborative systems. In *Proc. ACM Conf. CSCW*, Cambridge, 1996, pp.278–287.

YANG Guangxin was born in 1973 and got his B.S., M.S., and Ph.D. degrees in computer science from Tsinghua University in 1996, 1998, 2000 respectively. His major research interests focus on CSCW, groupware, workflow management, etc. He is currently a technical staff member at Bell-Labs Research China.

SHI Meilin was born in 1938 and got his B.S. degree in computer science in 1962 from Tsinghua University. His major research interests focus on computer network and CSCW. He is currently a professor at the Department of Computer Science and Technology of Tsinghua University.