A Change Management Framework: Dependency Maintenance and Change Notification

Soon-Young Huh
Department of Management Science, Korea Advanced Institute of Science and Technology, Taejon, South Korea

David A. Rosenberg
West Coast Development Laboratory, Object Design, Inc., San Mateo, California

In this article, we present an object-oriented database system (ODBMS)-based “change management framework” that manages dependency relationships between shared objects and dependent user views in a collaborative system and coordinates change and propagation activities between the two. First, it provides a set of abstract object classes that constitute the core constructs of the change management framework such as a dependent dictionary, a supporter, and a dependent. Second, it extends the framework in two directions: persistent shared objects and distributed computing. For persistent shared objects, a delayed change notification mechanism is additionally introduced to support a transaction management environment. For distributed computing, a client-server computing model is incorporated into the change management mechanisms. At the highest level of this framework, change manager classes are provided to encapsulate all the complex structures and dynamic behavioral schemes of the mechanisms. The framework is developed under a commercial ODBMS called OBJECTSTORE using the C++ programming language.

1. INTRODUCTION

Software systems called collaborative or groupware systems recently have emerged that support groups of people engaged in common tasks such as collaborative document editing, group decision making, group scheduling, concurrent engineering design, and software development. Since multiple users can modify the shared objects concurrently in these systems, and the change may immediately affect the tasks that other users are simultaneously engaged in, change management mechanisms are important for the provision of synchronized, consistent views of the evolving objects. To the users who are simultaneously engaged in the group work, such change management mechanisms usually include coordination of concurrent modification activities and correct propagation of change results.

Structurally, the change management mechanisms in a collaborative computing architecture involve a rich mix of several technology components such as an interactive user interface, distributed computing, a real-time system, and persistent object storage. Because those components have developed as independent research fields, many of them have become technically mature thanks to long development and wide research efforts, both in industry and academia. For instance, the model-view-controller (MVC) framework was developed as a standard user interface facilitating change management in SMALLTALK user communities (Goldberg and Robinson, 1983; Krasner and Pope, 1988) and was simulated in other development environments, such as an object-oriented program support class library written in C++ (Gurlen, 1987). Similarly, distributed computing has developed over the past decade, and techniques required to build distributed systems, such as a client-server paradigm, are well established. Recently, research efforts have been made to combine distributed computing with the object-based paradigm to support large and complex distributed applications (Chin and Chanson, 1991). For persis-
tent object management, several object-oriented
database management systems (ODBMS) have been
actively researched and are now commercially avail-
able (Kim and Lochovsky, 1989; Zdonik and Maier,
1990). Compared with the development of these
individual technologies, attempts to combine them
to facilitate change management in the collaborative
computing architecture are yet to be satisfactorily
proven because integration of the component tech-
nologies has been only partially achieved. For in-
stance, an integrated architecture such as the MVC
deals with only transient objects, not persistent
shared objects or multiple concurrent users and
distributed systems. In contrast, distributed object-
based programming systems and real-time operating
systems support persistent objects and multiple users.
However, they provide little for change management
to support concurrent access to shared objects and
automatic propagation of the change results. From
these studies, it is noticeable that one of the major
obstacles hampering integration of these component
technologies is that individual modeling paradigms
for representing distributed computing mechanisms,
data management, and change management mecha-
nisms are not compatible with one another.

Recognizing the challenges in integrating and
extending the individual technologies into a general-
ized modeling framework, we present an object-
oriented model for change management in the col-
laborative computing architecture. It specifically
adopts an ODBMS as an underlying platform; this is
largely because the ODBMS can facilitate seamless
accommodation of transient and persistent shared
objects and flexible capture of dynamic behavioral
mechanisms of the change management on a single
formalism, and because the object-oriented ap-
proach is generally known to be effective in dis-
tributed systems (Chin and Chanson, 1991) and
structurally complex systems (Booch, 1990; Rumba-
ugh et al., 1991). Current work focuses on devel-
opment of a change management framework that
facilitates maintenance of a dependency relationship
between user views and shared objects and coordi-
nation of change and propagation activities between
the two in distributed computing environments. In
doing so, it specifically integrates three individual
technologies: a change management mechanism for
transient objects, persistent object management, and
distributed computing. First, to embody the change
management mechanism, it establishes the depen-
dency relationship between the transient shared ob-
jects and user views and implements a change noti-
fication mechanism from a modified object to other
user views depending on the object. Second, it ex-
tends the change management mechanism to deal
with persistent shared objects. Since multiple users
can access the shared objects simultaneously,
database mechanisms for transaction management
and concurrency control are also considered. De-
layed change management is introduced as an exten-
sion to the change notification mechanism in a per-
sistent object management environment. Third, it
extends the change management mechanism to dis-
tributed computing environments. As a distributed
computing paradigm, a client-server model is used.
Using a layered approach, the dependency mainte-
nance of distributed objects on the client-server
model and change notification among processes are
implemented. For concrete and solid representation
of the change management mechanisms, class defi-
nitions are provided for all the major constructs with
simplified C++ class versions. A prototype system
is also developed based on a commercial ODBMS
called OBJECTSTORE with the C++ program-
ing language.

The article is organized as follows. In Section 2,
we review the literature on the object-oriented
paradigm and collaborative computing architecture.
In Section 3, we examine basic concepts and seman-
tics involved in the change management mecha-
nisms. In Section 4, we discuss change management
with transient objects with generic class definitions
capturing the constructs associated with dependency
maintenance and change notification mechanisms.
In Section 5, we extend the change management
mechanisms by introducing persistent shared objects
and a database transaction management mechanism.
In Section 6, we add distributed computing features
to the change management framework and provide
change manager class definitions to facilitate inter-
process communication over a client-server model.
Finally, in Section 7, we summarize the work and
provide research conclusions of the proposed change
management framework.

2. LITERATURE ON THE OBJECT-ORIENTED
PARADIGM AND COLLABORATIVE
COMPUTING ARCHITECTURE

With the advent of object-oriented programming
languages (OOPLs) such as SMALLTALK (Gold-
berg and Robson, 1983), C++ (Stevens, 1990),
Objective-C (Cox, 1980), and EIFFEL (Meyer, 1988),
the object-oriented paradigm emerges as a strong
technology for software development. The object-
oriented paradigm has been more enriched and es-
ablished by the efforts of the software engineering
communities with recent development of object-
oriented analysis and design methodologies (Booch, 1990; Coad and Yourdon, 1990; Rumbaugh et al., 1991). With the help of such a rich development basis, the paradigm has been applied to diverse application areas such as artificial intelligence, databases, real-time systems, distributed systems, and operating system design. Specifically, a fruitful extension has been made in the database area, resulting in ODBMS (Kim and Lochoisky, 1989; Zdonik and Maier, 1990). Aiming at next-generation database management systems (DBMS), the ODBMS provides a system in which the basic modeling primitives such as persistent data types can be arbitrarily defined and extended by users rather than constrained to certain system-provided types; the data types are object-oriented and extensible rather than record-oriented. On the basis of the ODBMS, the current research attempts to extend the object-oriented model in the area of collaborative computing.

Diverse research efforts have developed that can be categorized as studies on collaborative computing architecture in a broader perspective. Such studies are prominent in the areas of SMALLTALK programming environment, computer-supported cooperative work (CSCW), groupware systems, and distributed object-based operating systems.

In the SMALLTALK environment, the MVC architecture has been successfully adopted as a user interface paradigm (Goldberg and Robson, 1983; Krasner and Pope, 1988; Shan, 1989) because of its usefulness in window management. The architecture is mostly credited for its original provision of the change management framework, but has limitations in implementing a collaborative system since it provides only a single-user-oriented transient model and view-based interactive user interface paradigm.

CSCW and groupware systems support a group of people engaged in group work by promoting communication among them and dealing with structured sets of data objects including design drawings, software modules, documents, and complex relationships among data, people, and schedules (CSCW, 1988, 1990). Groupware systems are highly interactive with real-time change notification, distributed, focused on shared data with concurrent accesses, ad-hoc in the way of accessing and modifying data, and have external channels such as an audio or video link (Ellis and Gibbs, 1989). In terms of system features, groupware systems are supposed to support inheritance, object-linking, associative access, views, histories and versions, triggers, access control, concurrency control, and change notification (Ellis and Gibbs, 1989; Greif and Sarin, 1988), which generally reflect extensive functionality of database management systems (DBMS). However, since the groupware systems are usually developed to support specific application systems with dedicated system architectures, their framework would have limitations for general purpose use.

Recent work on distributed systems has emphasized incorporation of abstraction mechanisms in the operating system as the distributed applications tend to be large and complex. Typical development in this direction leads to various distributed, object-based programming systems (Chin and Chanson, 1991) such as Argus (Liskov, 1988), Emerald (Jul et al., 1988), and Guide (Krakowiak et al., 1990). By combining features of distributed operating systems and those of object-based or object-oriented programming languages, these systems provide an environment in which programs consisting of a set of interacting autonomous objects may execute concurrently on multiple processes that are distributed over a set of workstations or personal computers. In terms of collaborative computing, however, these systems provide little change management and notification capability among processes and weak persistence and transaction management functionality for shared concurrent multiuser environments.

As a whole, information sharing and change management are essential to collaborative computing systems. Accordingly, DBMS can be considered as a natural candidate for an underlying platform of the systems. However, relational DBMS can be less effective for the collaborative computing because of enforcement of flat tables on the complex objects through a normalization process. Most groupware systems use proprietary file-based systems and have added features found in DBMS, such as persistence, concurrency control, locking, and transaction management to the application systems. However, providing such data-sharing primitives may have other weaknesses: 1) it is burdensome for the system developers to program those data-sharing features in proprietary ways; 2) it might not be a robust system, nor a flexible system; and 3) the data-sharing program modules, if they are not well organized, could cause inefficiency and add complexity to the application system. We adopt an ODBMS as a general system platform for collaborative computing since it can facilitate direct capture of the complex objects and provide the whole set of DBMS features. We further add the change management capabilities to the ODBMS and develop a general change management framework that can support transient, persistent, and distributed objects on a single formalism, an object-oriented paradigm.
3. BASIC CONCEPTS AND CONSTRUCTS OF CHANGE MANAGEMENT

In this section, we describe the concepts and constructs involved in change management using an engineering design example. Consider a tree with a structure comprising a bicycle, as shown in Figure 1a. One possible representation of the tree that illustrates a bicycle shape might be made graphically, as shown in Figure 1b. Alternatively, the tree structure might display the tree textually as an indented list representation (Figure 1c) or diagrammatically as a tree representation (Figure 1d) to indicate the hierarchical bill of material relationships. The tree could be further partitioned into multiple subviews, which are individually designed to display different details of the same tree.

If the different views of the tree are to visualize the same tree simultaneously, then they must be consistent with each other and with the tree itself. In particular, if a part of the tree is modified through interaction with one view or by a separate process unknown to the views, then the rest of the views should be notified of that change. It is specifically noticeable that between the tree and its three views, there emerge two types of connections: a dependency relationship and change notification. The dependency relationship specifies that the views are dependent on the tree and are meaningful only in the presence of the tree. If a view is derived from a single tree, then the view will disappear immediately after the tree is removed. The change notification ensures that the dependent views must consistently mirror the current valid image of the tree by continuously reflecting the change of the tree. Conversely, the tree is also obliged to notify its dependent views of its dynamic changes. The concepts used in Figure 1 can be more generically defined as follows:

- **A supporter** is an object to be referenced and/or modified as a primary shared source. It can be a persistent object residing at a database server or a transient object located at the memory. In Figure 1, the tree structure configuring a bicycle in Figure 1a becomes a supporter.

- **A dependent** is an object that provides a visual representation of the supporter or determines the precise fashion in which the supporter is to be manifested. Dependents are created within a process that is executed as a single program in the operating system. In Figure 1, the three views (Figure 1, a–c) become the dependents of the tree supporter. In a multitasking computing environment, the dependents of a supporter can be made internally (locally) within the same process or externally (remotely) in the other processes.

- **A depending object** is a generic construct designating either a dependent or a process that contains a dependent. In the example, if Figure 1b is made by one process, say A, and Figures 1, c and d are made by another process, B, then the processes A and B become depending objects of the tree. In such a capacity, a dependent set identifies a set of the depending objects of a supporter.

- **A dependency pair** consists of a supporter and its dependent set (including the set of dependents and the set of depending objects), and dependency relationship designates the whole set of the dependency pairs.

In the dependency relationship, the nature of the depending objects can be more autonomous than the views of the tree example because they can be made to adapt discretionarily to changes made in the supporter. A depending object can reference multiple supporters, whereas a supporter can have multiple depending objects.

In terms of distribution of computing, the depending objects can be located in a single system or distributed in multiple systems; in both cases, the processes perform individual tasks concurrently by maintaining multiple independent threads of control. In such a concurrent multitasking environment, a supporter is located in one process of a certain system (i.e., a server system), whereas the dependents are distributed across processes in the other systems (i.e., client systems). Similarly, in terms of the object lifetime expectancy, a supporter can be either persistent or transient, whereas dependents are all transient. Thus, a persistent supporter survives beyond the lifetime of the process in which it was created, whereas its transient dependents disappear with the termination of the process.

![Figure 1. A binary tree and its views.](image-url)
Concerning the change notification, the supporter has an operation to notify changes made to itself to dependent sets. Such an operation may be invoked by either the supporter itself or the object that initiates the change. Thus, regardless of how the dependency relationship is constructed, that is, whether the supporter is persistent or dependents are distributed over the network, changes made in the supporter by one depending object must be quickly propagated to the rest of the dependents so that the dependents are modified accordingly.

In the following sections, change management mechanisms in different computing environments are discussed. Specifically, each mechanism is described in terms of constituent object classes because the class definitions provide a well-defined user interface and communication protocol consisting of data structures and their associated operations. In particular, because the underlying system platform is based on an ODBMS that usually supports a C++ programming language, the class definitions provided here are written in C++. Simplified versions of the classes are used to delineate the core functionality of the main object classes.

4. CHANGE MANAGEMENT WITH TRANSIENT OBJECTS

This section describes the change management mechanism in a simplistic computing environment in which all objects are transient in the memory and created within a single process. The change management mechanism has two primary tasks: management of dependency relationships and notification of change of the supporter to the dependents. First, a two-layered approach is provided to manage the dependency relationship; a dependent dictionary exists as a general dependency maintenance tool on the lower layer, and a supporter incorporates the dependent dictionary on the higher layer to support dependency relationships. Development of the change notification mechanism then follows.

4.1. Dependent Dictionary

Dependents of a supporter are generally very volatile and dynamically registered and removed at the users' convenience. The Dependent Dictionary class is a dedicated tool set for managing such dynamically changing dependency relationships in a general and efficient way (Figure 2). By accommodating an abstract superclass as a dependent class, the dictionary does not place any restrictions on the object type of the dependents. By doing so, the dictionary class makes a general tool in which dependents can take on a wide variety of object types, ranging from textual documents to multimedia data.

In the class, the relationship between the supporter and dependents is established with a one-to-many relationship. The keys of the dictionary become the supporters, and the values for a key are the dependent sets of the supporter. For implementing the dependency relationship between the supporters and dependents, the ODBMS allows developers to choose a search method out of several options including linked list, array, set, and hash table; to obtain high performance in the dependency relationship with a large amount of dependency

```
typedef reference<Supporter> Supporter_ref;
template<class DependingObject>
class DependentDictionary
{
  public:
    DependentDictionary();
    ~DependentDictionary();
    Supporter_ref* add(Supporter&, DependingObject);
    set<DependingObject*>* dependents(Supporter&);
    Supporter_ref* removeDependent(Supporter&, DependingObject);
    Supporter_ref* removeAll(Supporter&);
    // remove all dependents
    Supporter_ref* removeAllDependents(DependingObject);
    // remove the given dependent from all supporters

  protected:
    hash<Supporter_ref*, set<DependingObject>*> hashTable;
}
```

Figure 2. A class definition of dependent dictionary.
pairs, a hash table is chosen. In support of dynamic dependency management based on the hash table, three types of primary data manipulation operations are provided: addition, deletion, and retrieval of dependents for a specific supporter. These are embedded in the operations of a supporter class and serve the dependency maintenance requests from the supporter as the lowest level operations.

4.2. Supporter and Management of Dependents

On the basis of the Dependent Dictionary class, two classes are defined at a higher level: Supporter and Dependent (Figure 3). In this section, the dependency relationship aspect of the two classes is further discussed; the change notification aspect is discussed in the following section.

The Supporter class incorporates the Dependent Dictionary class as a tool by containing dependences as an object-valued attribute. Specifically, the dependences attribute is defined to handle only dependents by parameterizing the Dictionary class with the Dependent class. Thus, through the dependences attribute, a supporter is able to contain both its dependents and the basic dependency maintenance operations that are provided by the Dependent Dictionary. We name the dependency attribute internal dependent dictionary because for the current process, it enables a supporter to directly refer to its local dependents and provides a well-defined basis for maintaining them.

Though several primitive data manipulation operations are provided by the Dependent Dictionary class embedded in the dependences, they are not adequate to effectively meet the particular needs arising in the management of dependency relationships such as object referential integrity between supporters and dependents. For more customized service, both the Supporter and Dependent classes provide their own set of operations. First, in the Supporter class, addDependent and removeDependent facilitate registration and deletion of a certain dependent in association with the current supporter when the dependent is to be connected or disconnected. Since a dependent can have multiple supporters, when the dependent is to be associated with another supporter, addDependent needs to be executed again for the dependent in the second supporter. As such, these operations are geared to maintain the basic one-to-one binary dependency relationship between a dependent and its supporter.

~ Supporter facilitates removal of a supporter. If there are some dependents associated with the supporter in the dependences, it also releases the dependents to ensure the referential integrity in the dependency pair. Additionally, to facilitate the management of a persistent supporter in a database system, initialize lets the database system be ready for the dependency maintenance mechanism. Further discussion about the persistent supporter is provided in Section 5.2. Second, in the Dependent class, ~ Dependent helps to delete a single depen-

![Figure 3. Definition of supporter and dependent classes.](image-url)
dent and releases the dependency relationship from its supporters. Internally, \texttt{Dependent} first identifies all its supporters, and executes \texttt{deleteDependent} in the individual supporters. Thus, whenever a dependent is created or removed from a supporter, the dependency pair is continuously maintained in the dictionary.

Note that the class definitions for \texttt{Supporter} and \texttt{Dependent} are generic, minimal, and dedicated to support management of dependency relationships and change notification (change notification is addressed in the following section) so that they can be easily used in other applications. As superclasses, they will be inherited into derived supporter and dependent classes and facilitate the addition of application-specific attributes and operations to the subclass definitions. Such class inheritance also leads to effective division of tasks and responsibilities between the superclasses and subclasses. The superclasses take care of the generic change management tasks, and the subclasses focus only on application-dependent tasks.

4.3. Change Notification

When a supporter is changed, by invoking \texttt{changed}, the \texttt{Supporter} class activates the change notification mechanism and propagates the change information to all the dependents of the supporter. Specifically, when the supporter is a transient object, \texttt{sendUpdate} is internally triggered inside \texttt{changed} to send an update request message to all the registered dependents. In case of a persistent supporter, the internal behavior of \texttt{changed} differs (as is elaborated in Section 5.2). Upon receiving the requests, the dependents execute \texttt{update} to modify their status accordingly. If necessary, more elaborated update requests can be delivered from the supporter to the dependents to help dependents determine how to update themselves, using an argument associated with \texttt{changed} and \texttt{update}. Instead of the current approach of immediate invocation of the change notification mechanism everytime the supporter changes, the timing for invoking \texttt{changed} can also be discretionarily controlled by the modifying process to avoid performance degradation due to frequent change propagation. In either case, the supporter is made to signal itself immediately to confirm the change.

As mentioned earlier, the benefits of the change notification mechanism are realized in the subclasses of the \texttt{Supporter}. At the subclass layer, when the supporter is changed, the communication protocol more likely becomes a message-broadcasting protocol than a one-to-one communication protocol because the supporter only needs to signal a change to itself without caring about who should be notified of this change. The rest of the tasks are taken care of by the superclass, \texttt{Supporter}. First, the dependency relationship mechanism is delegated for identification of the dependents, and second, the change notification mechanism handles the propagation of the change to the dependents. To users, such a change notification process would look even simpler. When the user changes a supporter, all the dependents automatically receive update request messages, and thus adjust themselves correspondingly.

Just as the dependency relationship mechanism was defined to be generic and minimal in the class definition of \texttt{Supporter} and \texttt{Dependent}, so is the change notification mechanism. The classes are free from application-specific operation. For instance, the \texttt{Supporter} class does not provide any operation to perform changes to a supporter. Similarly, the \texttt{update} operation of the \texttt{Dependent} class is left undefined. This is because \texttt{Supporter} and \texttt{Dependent} are supposed to be inherited to subclasses. The change operations in a supporter and the corresponding update operations in a dependent can be added or solidified in individual subclasses. For instance, in the implementation of the change operation, the \texttt{changed} operation may be included as a finishing statement to trigger the communication protocol of change notification. By doing so, when a change is made by the change operation, the change notification mechanism is automatically activated immediately.

The change management mechanism can now be applied to the example of the bicycle tree structure introduced in Figure 1. In terms of dependency relationships, the tree structure (Figure 1a) and the rest of three individual views (Figure 1, b–d) would be registered respectively as instances of subclasses of \texttt{Supporter} and \texttt{Dependent}, that is, a supporter and its dependents. The tree structure, as a supporter, may additionally have change operations such as adding and deleting a part, or moving a part from one place to another. In contrast, the individual views would specialize the generic \texttt{update} operation of the \texttt{Dependent} class with different implementations to perform a graphical or textual update task in response to changes made in the tree structure. Now, the dynamics of the change notification between the two would become fairly straightforward. For instance, when the front wheel of the bicycle tree structure is replaced with a different style wheel,
the tree structure would immediately trigger changed with an appropriate argument reflecting the content of the change; the dependent views would then be notified of the change, execute individual updates so that they can adjust individual representation accordingly, and thus the change is propagated throughout. In brief, the proposed change management mechanism can be applied to a variety of application systems without requiring additional significant programming effort.

The following sections explore the extensions of the change management mechanism to a persistent and distributed computing environment.

5. CHANGE MANAGEMENT WITH PERSISTENT SUPPORTERS

A supporter is called persistent if it is stored in a database and outlives the program session in which it was created. Thus, persistence of the supporter is achieved not be simply taking complete snapshots of the supporter to be stored in a filing system but by making the supporter sharable by multiple users. In addition, different from traditional DBMS, the change notification mechanism ought to be provided. Instead of waiting passively for the user's intervention to informing the dependents of changes in the supporter, the mechanism should monitor the dynamic changes of the persistent supporters and actively notify the dependents of the significant changes.

5.1. Change Operations and Transaction Management

As was described in a previous section, because the transient supporter resides in the memory, a change made to it is always valid, and change propagation to the dependents should follow immediately after the supporter is changed. In contrast, because a persistent supporter is stored in a database system and there are multiple concurrent users accessing it, operations intended to change the supporter might conflict with one another and thus not be able to propagate change immediately to dependents. More precisely, the change operation upon the persistent supporter needs the following basic features:

- Object Integrity. The system must ensure that a persistent supporter is always in valid states before and after a change operation is made to the supporter. In other words, an operation either entirely completes change in the supporter (a commit state) or has no effect at all (an abort state).
- Serializability. Multiple operations that execute concurrently against the same supporter should be scheduled in a sequential order and not interfere with one another so that object integrity of the supporter is maintained.
- Permanence. Once an operation commits, its effect cannot be backed out of the database. The effect can be altered by running another operation.
- Transaction Management. Transaction management is a mechanism used by the DBMS to ensure the aforementioned properties on the change operation. Transaction management has a boundary with beginning and ending points; all the changes made to a supporter within the transaction boundary are private and invisible to other processes until the entire transaction terminates successfully. Inside the boundary, a submitted typical transaction goes through several states including begin, precommit, commit, abort. Outside the transaction boundary, the state of the supporter is valid while the supporter's status inside the boundary cannot be affirmed valid yet until a transaction terminates.

OODBSs usually support these features since they have adopted the classic DBMS notion of transactions as a foundation for ensuring object integrity. In such a capacity, dependents are supposed to reflect the valid images of the supporters consistently. In maintaining the consistency, the internal states of the transaction boundary are utilized to facilitate the change notification mechanism. Specifically, object operations the change notification mechanism are flexibly associated and triggered with the states to facilitate saving, removing, or propagating changes made to supporters.

5.2. Delayed Changes

When a change operation is performed upon a persistent supporter, the dependents can be notified only if the operation inside a transaction boundary has committed. Otherwise, dependents need not to be aware of the change attempt. A more practical case occurs when a series of change operations are performed on persistent supporters in a single transaction boundary. In case of a “moving object” operation that consists of deleting and adding operations, though the first deleting operation is performed well, the partial change cannot be notified but should be delayed until the second adding operation is also executed successfully. If the second operation gets entangled, the transaction will abort, and the de-
A Change Management Framework

J. SYSTEMS SOFTWARE
1996; 34:231-246

A change should be removed. As such, in general, because a transaction can be aborted in the midst of change operations, change made by individual operations cannot be propagated immediately to the dependents. Rather, the changes are accumulated and delayed until the whole transaction is finished, and are to be propagated only when the transaction reaches the commit state.

To facilitate change notification under the transaction boundary and to support the delayed change, the Delayed Change class is provided (Figure 4). A delayed change is an abstract messenger object of a supporter, which serves to deliver some direction about how or what to change to the dependents. The delayed change is created when a change operation is performed on a persistent supporter in a transaction boundary. More precisely, when a change operation of the supporter is executed and the \textit{changed} operation is triggered inside the change operation (see Section 4.3), the persistence of the supporter is first checked in the \textit{changed} operation. If the supporter is persistent, then a corresponding delayed change is automatically created with a reference to the supporter and a direction, if needed. Later, if the transaction commits successfully, then the delayed change is released and propagated to the dependents with the help of the internal dependent dictionary. When multiple change operations are performed on the supporters in a transaction, the corresponding delayed changes are created and sequentially accumulated in a list. By handling multiple delayed changes as a collection, management of the delayed changes can be more compact and easier.

The operations of the Delayed Change class can be classified into two categories. The first type of operations helps with creation, addition, and removal of a delayed change from the chained list of delayed changes. When a persistent supporter is changed inside a transaction, a delayed change is created to hold the content of the change and added to the chained list of delayed changes. Such delayed changes last during a particular transaction and are all removed when the transaction is finished. Because a delayed change is created for each change operation for a specific persistent supporter, and multiple changes can be made on the same supporter in a single transaction, the chained list consequently maintains a one-to-many relationship between persistent supporters and delayed changes.

The second type of operations is involved in the change notification mechanism. When the transaction finishes with a commit state, the change notification mechanism is activated by propagating the delayed changes to all the concerned dependents. Many of the operations of both types are executed as special procedures that are embedded in the transaction states.

To better understand the notification of delayed changes under transaction management, suppose an example transaction that involves a series of change operations as shown in Figure 5. At time $T_0$, a

```cpp
class DelayedChange
{
    public:
        DelayedChange(Supporter&, int, DelayedChange*);
        DelayedChange();
        void add(Supporter&, int);
        void removeAll();
        void remove(DelayedChange*);
        DelayedChange* readFrom(istream&);
        void propagateAll();
        void propagateInternal();
        Supporter& getSupporter();

    protected:
        Supporter& supporter; // the supporter
        int aspect; // the change aspect
        DelayedChange* first;
        DelayedChange* next;
}
```

// add a delayed change for the specified supporter next
// to the existing delayed change
// initialize a delayed change
// add a new delayed change for the persistent supporter
// remove the list of delayed changes
// remove the specified delayed change
// read a delayed change from an input stream
// propagate changes to the dependents
// propagate changes to internal dependents
// return a supporter from the current delayed change

// the supporter for the delayed change

// the head of a list of delayed changes associated with
// the current transaction
// next delayed change in the list

Figure 4. Definition of delayed change class.
transaction begins, and at T1, T2, and T3, individual change operations are performed against three different supporters, S1, S2, and S3, where S1 and S3 are persistent supporters. At time T4, the transaction would commit successfully, or abort in some moment in between T0 and T4.

More precisely, at time T0 when the begin state occurs, the begin procedure is triggered, and the ODBMS would read all persistent supporters and their relationship to maintain the initial states of the supporters and to use for roll back in the case of the abort state. When a supporter, say S1, is changed at time T1, because the transaction has not committed yet, a delayed change, D1, is created. At time T2, another change is made to S2, but no delayed change is generated because it is a transient supporter. After time T3, when the third change is attempted to the remaining supporter, S3, another delayed change, D3, would be added to D1 and chained in an order by which the associated supporters were changed in the transaction. Note that serializing the delayed changes in this sequence is a cost-effective way to guarantee the object integrity of the dependents later in the change propagation. At time T4, when the commit state occurs, the commit procedure is triggered and executes the embedded propagation operation, and individual delayed changes in the chain, D1 and D3, are all propagated to their dependents. In contrast, if an abort state occurs during the transaction, because the changes made to the supporters by that moment are all nullified inside the transaction, the abort procedure is triggered and the current chain of delayed changes would be removed by embedded remove operation.

By adopting the transaction management mechanism from ODBMSs and implementing the delayed change concept, a set of transparencies are achieved that hide complexity involved in persistent supporters. Second, access transparency is provided so that the user can use uniform access operations on the supporters regardless of whether the supporters are located in memory or secondary storage. Third, concurrence transparency is automatically obtained because users do not have to cope with problems of concurrent access.

So far, we assume that the supporters and dependents are all in the same system. However, because the users of a supporter are likely to be distributed, we extend the change notification mechanism to a distributed computing environment in the following section.

6. CHANGE MANAGEMENT UNDER DISTRIBUTED COMPUTING

A collaborative computing environment can consist of a set of participant systems connected by a network. We are particularly concerned with a distributed computing environment in which supporters are located in a host system and are accessed by multiple user processes that are created in independent users' workstations on the network. To develop a change management framework on such a distributed computing environment, we adopt a client-server model and encapsulate the necessary mechanisms into two change manager objects called Change Manager Server and Change Manager Client. As principal change management agents in distributed computing, the two manager objects encapsulate all the change management mechanisms discussed previously, as well as the necessary communication techniques, and cause all interaction between the client and server processes to be regarded as operations of the two objects.

6.1. Communication Techniques in Distributed Computing

The area of distributed computing is now reasonably mature in its techniques. To build a change management framework in distributed systems, we specifically adopt the following techniques as underlying principles.

- Client server model. As the standard model for network applications, a client (process) invokes an operation performed by a server (process) by sending a message to the server, and the server responds by performing the requested operation and replying a message to the client.
Interprocess Communication (IPC) facility. As a tool to support the client-server model, the IPC facility is used to facilitate communication between two processes [21]. As an actual network communication protocol, we employ the UNIX TCP/IP protocol that is based on 4.3 BSD UNIX socket interface. In particular, a connection-oriented protocol is adopted for the server to flexibly create and assign multiple processes to clients. Under such a protocol, a server is first started and waits for a connection request from a client. When a connection is requested by a client, the server creates a new process to handle this request, establishes a connection channel, and finally returns to the waiting status for another request from the client. In the change management framework, the server and client processes are substituted by the Change Manager Server and Change Manager Client objects, and the IPC facility is embodied inside the two primary change manager classes. Therefore, all interaction between the client and server processes can be thought of as operations of the two types of objects. From now on, whenever convenient, Change Manager Server and Change Manager Client are alternatively termed as the server and client respectively.

Remote Procedure Calls (RPCs). RPCs are used when a process needs to use of remote services from other processes. The RPCs use the message-passing IPC facility and embed the remote service requests inside the ordinary local operations. By enabling a local operation to invoke an operation of a remote process, the RPCs help to extend the semantics of execution of local operations to the distributed computing environment. Thus, the server and client are allowed to deliver the messages in a bilateral way so that the client can send a message requesting an operation to the server or the server can respond to the client by replying to a message. Because both the client and server use stream-based communication operations, Change Manager Base class is additionally defined as a superclass of the two change manager classes to provide a dedicated and consolidated support for a set of the communication operations.

6.2. Design of Change Manager Server and Client

In a typical client-server computing environment, persistent supporters are stored in the server system and their dependents are distributed in multiple client systems. Figure 6 shows an example of the client-server model, where one server system interacts with three client processes. In the figure, persistent supporters A, B and C are stored in the ODBMS of the server, and their dependents are created and dispersed in the clients P1, P2, and P3. From the viewpoint of a particular client, the dependents of a persistent supporter can be classified into two types depending on where they are located. Internal dependents designate the dependents located in the concerned client, whereas external dependents designate ones found in other processes. For instance, from client P1, the internal dependents of supporter A are a1 and a2, whereas the external dependent of the same supporter is a3, found in client P2. Conversely, from the viewpoint of another client, P2, the dependents a1 and a2 in P1 are external dependents, whereas a3 becomes an internal dependent. A transient supporter cannot have external dependents, because it is not designed to be sharable by multiple processes.

To manage both the external and internal dependents effectively, a two-tiered dependency relationship is established by placing the lower tier at the client (process) level and the upper tier at the server (process) level. At the lower tier, to manage the dependency relationship between a persistent supporter and its internal dependents, each client contains an internal dependent dictionary, which is identical to the dependent dictionary defined previously in Section 4.2 and Figure 3. Figure 6 shows that the clients P1, P2, and P3 individually have their own internal dependent dictionaries. However, the internal dependents contained in the individual clients are made invisible to the server, and thus the server is not required to remember all the dependents but the clients containing them. Therefore, at the upper tier, to handle the dependency relationship between a persistent supporter...
supporter and its depending clients, the server is provided with an external dependent dictionary. The dictionary uses a client as a depending object of the Dependent Dictionary and maintains dependency pairs between a persistent supporter and its set of depending clients. Again, the external dependent dictionary primarily releases the clients from the burden of remembering all the external dependents. By keeping the dependency pairs at the upper tier, the external dictionary identifies the dependent clients for a given changed supporter, and can help those clients to trace further and return their internal dependents. At the lower tier, if the depending clients are identified for a supporter, their internal dependents can be traced further by the internal dependent dictionaries of the individual clients. In brief, the two-tiered dependency relationship is designed to divide and abstract the management of dependency relationship into two independent modules so that each tier can focus on a specialized task and handle it with autonomy and efficiency. At the same time, both tiers are made to cooperate with each other like a cascade, and thus, the overall management of dependency relationships is smoothly conducted in a distributed computing environment.

The mechanism to support the two-tiered dependency relationship is embodied in the change manager objects, Change Manager Server and Change Manager Client, as shown in Figure 7. Specifically, Change Manager Base class is provided as a superclass to these two classes to support UNIX socket-based communication between the two. Briefly, Change Manager Base provides three types of communication operations: 1) getting the port number.

```cpp
class ChangeManagerServer: public ChangeManagerBase
{
public:
    void initialize(); // initialize TCP-based change server
    -ChangeManagerServer(); // remove the supporter
    void addSupporter(Supporter*, int); // add the client as a dependent of the supporter
    void removeSupporter(Supporter*, int); // remove the client
    void listen(); // checks for input on any currently connected socket
    void processMsg(); // dispatch an appropriate operation against incoming message
    void propagate(DelayedChange*, int); // propagate delayed change that is transmitted from initiating client

protected:
    DependentDictionary<int> dependencies; // external dependent dictionary
    set<int> clients; // set of currently active socket number
    int sock; // passive, listening socket
}

class ChangeManagerClient: public ChangeManagerBase
{
public:
    void initialize(); // initialize TCP-based change client
    -ChangeManagerClient(); // remove the supporter
    void addSupporter(Supporter&); // notify the server that the process depends on the supporter
    void removeSupporter(Supporter&); // notify the server that the process no longer depends on the supporter
    void connect(); // initiate the connection
    void propagate(DelayedChange&); // notify the server that the list of delayed changes have committed against their associated supporters

protected:
    void readMessage(); // read and process messages from server
    int sock; // socket for connection
}
```

Figure 7. Simplified definitions of change manager server and change manager client classes.
associated with the connection on the server machine; 2) maintaining input and output stream buffers and returning the input and output streams from the buffers; and 3) notifying clients if the stream connection is lost for any reason.

The Change Manager Server enriches the inherited operations to perform three types of primary operations: communication connection, maintenance of the external dependent dictionary, and change notification and propagation. First, for the communication connection between the server and a client, the server initiates a half-connection and listens for requests on full connection from a client. From the moment of the full connection, the client can communicate with the server, sending and receiving messages that consist of an RPC. When an RPC message is received from a client, it is interpreted and dispatched to execute an appropriate operation. Once the connection is established, the server maintains the external dependent dictionary using the dependencies attribute and stores dependency pairs consisting of a supporter and a set of clients. Being invoked by the RPC messages from a client, the operations such as addSupporter and removeSupporter facilitate addition or deletion of the current client as a dependent of a certain supporter and maintain the upper tier dependency relationship, that is, the relationship between supporters and their dependent clients. Recalling that the upper tier dependency relationship is affected by the alteration of the lower tier dependency relationship, these operations are invoked when a dependent for a specific supporter is created or deleted in the client and the server is to be informed of the change. A client message usually consists of an operator and operand, which, respectively, correspond to a requested operation in the server and the associated supporter. For instance, if a message consists of add_supporter and supporter A, it is interpreted to invoke the addSupporter to add the current client as a depending client of supporter A. Third, when change is attempted to a supporter, the server immediately executes the change notification mechanism and propagates the changes to the depending clients of the supporter. Specifically, propagate is used for the change notification. It first identifies all the depending clients with the help of the external dependent dictionary and then propagates the messages notifying the delayed change to the clients.

The Change Manager Client performs counterpart operations regarding the three types of operations of the Change Manager Server, because the two objects interact through the change management mechanism. First, the client determines the host system for the communication with the server and sends the server a request for connection. Second, concerning the dependency relationship, the client may inform the server to modify its external dependent dictionary as soon as the client is related to a new persistent supporter. In such a case, the client requests the server to add itself as a depending client of the supporters by sending an RPC. Then, the Change Manager Server invokes the corresponding dependency maintenance operations such as addSupporter and removeSupporter to update the external dependent dictionary. Lastly, regarding the change notification, when a client commits a change on a persistent supporter, it additionally notifies the server of a corresponding delayed change by executing propagate so that the committed change can be propagated to the supporter’s external dependent clients. Subsequently, the propagate of the server is invoked, and the dependent clients are notified of the delayed change. Finally, using readMessage, individual clients receive the change update message and process it by performing appropriate change update operations on their internal dependents.

6.3. Change Management Processes between Change Manager Server and Clients

In this section we briefly present a whole picture of the change management process by tracing the interactions between the server and client (i.e., Change Manager Server and Client), which make up the two-tiered dependency relationships and change propagation. The interaction can be segmented into three major phases and is summarized as follows (Figure 8):

The first phase is concerned with the communication connection between the server and client and initialization of two types of dependent dictionaries. In this phase, the server initializes a communication channel and is ready to accept a client request for connection. A client is created and initializes the ODBMS transaction management facility. It also creates internal dependents for some transient or persistent supporters and constructs the internal dependent dictionary. Then it establishes a communication connection to the server by sending the connection request. The client notifies the server of the dependency relationships between the client and the persistent supporters, which the client refers to so that the server manages the external dependent dictionary consistently. In the example shown in Figure 6, all of the internal and external dependent dictionaries can be established in this phase.
The second phase is concerned with the ongoing change management activities inside a client, that is, change operations to supporters and change propagation to internal dependents. When the client makes a change to a supporter, the supporter automatically checks its persistence to decide when to propagate the change by executing change. If the supporter is transient, the change is immediately propagated to the internal dependents. In an opposite case, a corresponding delayed change is created and added to the list of delayed changes until the current transaction is finished. In Figure 6, suppose that the dependents \( a_1 \) and \( b_1 \) are modified in the client \( P_1 \), and thus the supporters \( A \) and \( B \) are changed. Then their delayed changes will be created and accumulated momentarily until the transaction finishes. When the transaction is committed, the whole list of delayed changes will be sent to the server for subsequent change propagation.

The third phase is concerned with the ongoing change management activities outside a client, that is, change propagation to external dependents for shared persistent supporters. When a list of committed delayed changes arrives at the server, the server needs to redistribute the delayed changes to appropriate depending clients so that each client is notified of the delayed changes that are directly relevant. For this, the server identifies the depending clients for each delayed change using its external dependent dictionary. Then, from the list of delayed changes notified from the client, the server generates multiple lists of delayed changes, each destined for individual clients, and sends them to the individual clients with a change update request. On the client side, each client receives a list of rearranged delayed changes and performs an appropriate update for individual delayed changes. For each supporter in a delayed change of the list, the client consults the internal dependent dictionary to get the associated dependents and invoke update operations in those dependents. As such, through the change management activities that carry out the second and third phases continuously, consistency in views between supporters and their dependents is maintained.

In the example shown in Figure 6, the changes made in the client \( P_1 \) cause creation of two delayed changes, say, \( d_A \) and \( d_B \), in association with supporters \( A \) and \( B \), respectively. Upon the arrival of the two delayed changes from \( P_1 \), for subsequent change propagation, the server identifies the rest of the depending clients using the external dependent dictionary: \( P_2 \) for \( d_A \) and \( P_2 \) and \( P_3 \) for \( d_B \). Then the server dispatches the delayed changes to individual clients by allocating appropriate delayed changes. Client \( P_2 \) will receive both \( d_A \) and \( d_B \), whereas client \( P_3 \) will receive only one delayed change, \( d_B \). Each client will subsequently perform the appropriate updates for the individual delayed changes. In the case of \( P_2 \), it will subsequently perform change updates on the internal dependents of the two delayed changes, that is, \( a_3 \) for \( d_A \) and \( b_2 \) and \( b_3 \) for \( d_B \).

By developing such a distributed computing environment, location transparency is additionally achieved in both dependents and operations. First, location transparency in dependents occurs because a client is not required to remember all the external dependents dispersed over the network. For change propagation, all the client has to do is to send the server a list of supporters (i.e., delayed changes) that are changed. Change updates are then automatically made in all external dependents. Second, location transparency in operations is achieved through the RPCs used in the client and server (i.e., Change Manager Client and Server). The user is not aware of the location of operations, that is, whether they are local or remote. Thus, the RPCs help users to incorporate distributed applications into a conceptually uniform local view, and significantly contribute to making the whole change management mechanisms simplistic and understandable.
7. CONCLUSION

The provision of a robust and open framework for change management is a significant and important step for collaborative computing. Toward this goal, this research presents an object-oriented change management framework that uses ODBMS as a system platform. Using a library of classes, the framework provides an object-oriented model to support both the dependency relationship and change notification mechanisms. For the management of dependency relationships, the framework uses a dependent dictionary to maintain dependency pairs between the supporters and their dependents; for change notification, it adopts a broadcasting protocol that allows a changed supporter to send a message to its dependents so that the dependents easily have synchronized, consistent views of the supporters that keep evolving.

As a whole, in the object-oriented change management framework, several advantages are attained. Specifically, three benefits are worthy of attention.

First, the object-oriented paradigm provides high abstraction in the change management framework in which major mechanisms including dependent dictionary, supporter, dependent, client, and server are all implemented as self-contained and well-modularized objects. Through the abstraction mechanism, individual objects provide a set of well-defined service protocols on the basis of encapsulated operations and relieve users of the details of a mechanism by hiding complicated data structures and algorithms. Moreover, because a composite object can be built on the previously defined objects as building blocks, those abstract change management mechanisms can be integrated effectively into a higher level object without increasing the complexity of the mechanisms. Most prominent examples are found in the Change Manager Server and Change Manager Client classes. As powerful but still generic composite objects, they consolidate all the primitive change management mechanisms and encapsulate all the necessary communication mechanisms between clients and servers, enhancing simplicity and abstraction of the detailed mechanisms. Thus, the two objects become basic system organizing constructs for change management in collaborative computing and cause all interactions between the client and server processes to be regarded as the operations of the objects. As such, the object-oriented paradigm adopted in this work primarily causes the change management mechanisms and their interactions to be not only conceptually understandable, but also decomposable into smaller modules and expandable to diverse problem domains.

Second, by adopting an ODBMS as an underlying platform, the framework facilitates seamless accommodation of transient and persistent shared supporters and flexible capture of dynamic behavioral mechanisms of change management on a single formalism. The ODBMS approach specifically enriches change management mechanisms with stabilized concurrency handling and transaction management capabilities and ensures that the collaborative computing architecture is more reliable and secure under a multiuser computing environment than a proprietary file system approach. The framework also extends the change management mechanisms to a distributed computing environment by taking advantage of the open architecture of the ODBMS toward a client-server computing model.

Third, a number of transparencies in computing are achieved in the framework, serving a concurrent and distributed computing environment. Persistence, access, and concurrence transparencies are attained in the dependency relationship between the persistent, shared supporters and their dependents. Location transparencies are attained in the internal and external dependents as well as local and remote operations of the clients and servers. Blended with the abstraction mechanism of the object-oriented paradigm, these computing transparencies hide problems and complexities involved in change management processes in collaborative computing so that the conceptual views of major constructs of the change management framework are not distorted by the computational details and complexities.

A prototype system for the change management framework has been developed on a commercial ODBMS called OBJECTSTORE (Lamb et al., 1991) on the SUN-4 system. Further research is required in the application of the change management framework to specialized collaborative computing systems such as office document handling, computer-aided engineering design, model management, group decision support system (GDSS), or CSCW systems.

REFERENCES

Coad, P., and Yourdon, E., Object-Oriented Analysis, Yourdon Press, Prentice-Hall, 1990.