An Object-Oriented Database Model for a Change Management Framework in Workgroup Computing Systems

Soon-Young Huh
Graduate School of Management
Korea Advanced Institute of Science and Technology
207-43 Cheongryangri-Dong, Dongdaemoon-Ku, Seoul, Korea 130-012
syhuh@green.kaist.ac.kr, fax: 82-2-958-3604

Abstract

Workgroup computing systems are emerging to support a group of users engaged in common tasks such as group decision making, engineering design, group scheduling, or collaborative writing. Since shared objects in the workgroup systems are constantly evolving, providing the users with synchronized and consistent views of the objects is important for achieving improved team productivity. This paper proposes an object-oriented model for a change management framework supporting workgroup systems to facilitate managing dependency relationship between shared objects and dependent user views and to coordinate change and propagation activities between the two in client-server computing environments. In developing the framework, this study firstly singles out the requirements of change management in workgroup systems and conceptualizes the change management mechanisms with transient shared objects. It secondly extends the mechanisms into a persistent transactional database computing and client-server computing environment. In so doing, it specifically employs an object-oriented database paradigm to seamlessly capture transient and persistent transactional objects in a single formalism and presents a set of well-modularized abstract object constructs facilitating dependency management and change notification mechanisms for workgroup computing systems. The proposed framework is developed on the basis of a commercial Object-oriented Database Management System (ODBMS) called OBJECTSTORE using the C++ programming language.
1. Introduction

Workgroup computing systems are gradually adopted in organizations to support a group of people performing collaborative tasks in a shared environment. Recently, as the Internet and intranets are emerging, workgroup computing systems gain wider acceptance from the collaborative workgroups [6]. Such systems help group users to perform tasks individually in a distributed environment while achieving enhanced productivity in group work by facilitating shared information about whole tasks. Examples of such systems include concurrent engineering design, group decision making, multi-user document editing, group scheduling, and medical diagnosis [8, 9]. In such workgroup systems, group users interact with the shared objects (i.e., shared information resources) on the basis of user views by creating, deleting, examining, or modifying the objects. In such capacity, when some users make changes on the objects concurrently, other users may be left with unupdated views. As well, they may experience conflicts in their subsequent updates of the objects since the contents of the objects shown in the views might be inconsistent with the actual contents stored in the database. Thus, providing users with robust object management mechanisms as well as synchronized and consistent views of the objects becomes important for accomplishing the goal of workgroup computing. The role of a change management framework, in this sense, is crucial to facilitate sharing of group objects in the central object management servers and maintaining consistency between a base shared object and its dependent user views [10, 14, 15].

With respect to a change management framework in workgroup systems, research has been done in a wide variety of disciplines: user interfaces, object-oriented programming language (OOPL), persistent object management, and client-server systems. OOPL has already been adopted as a strong user interface component technology of workgroup computing on the basis of programming languages such as Smalltalk [18], C++ [28], and Objective-C [7]. For instance, the Model-View-Controller (MVC) framework has been developed as a standard user interface facilitating task sharing and transient change management in the Smalltalk user communities [18, 26]. Similar development framework for object sharing was made in the C++ community through the Object-oriented Program Support
class library [13]. Meanwhile, since workgroup systems involve a group of users who demand synchronous interaction, robust support for concurrent task access and transactional object management are requested and thus database management systems serve as the main storage system for the object repositories [8, 9, 10, 17, 21]. Because the structures and compositions of the shared objects of the workgroup systems are complicated, proprietary object-based file systems provided with some DBMS features such as concurrency control and locking are also adopted as persistent object repositories [10, 14]. Finally, client-server systems have developed over the past decade as an independent research area to provide task sharing and distributed computing over inter-connected networks, which has resulted in well-established techniques such as client-server paradigms [25, 27]. Recently, the client-server systems are extended to distributed computing systems to support large and complex distributed applications. Accordingly, the object-oriented paradigm is being increasingly incorporated into distributed computing systems [4, 12, 29], thereby providing an implementation basis of commercialized distributed computing architectures such as CORBA and OLE [2, 29].

Compared with the successful progress made in individual component technologies, however, research on change management for workgroup computing systems has not yet constructed an integrated framework that merges the individual technologies under a single formalism or that provides generic dependency maintenance constructs extendible to diverse application systems. For instance, the MVC architecture is a transient object-based, single user oriented system and cannot deal with persistent shared objects, nor multiple concurrent users and client-server systems. Contrastively, though distributed object-based programming systems can support persistent objects and multiple users, they provide little for change management that supports both concurrent access to shared objects and automatic propagation of the change results. Similarly, though file system-based object repository approaches [8, 9, 10, 14] attempt to accommodate complicated persistent objects through transaction mechanisms, they have limitations of proprietary systems including instability, inflexibility, complexity of the mechanisms and idiosyncrasies involved in their data management function calls. To facilitate development of workgroup systems, a generalized
and flexible change management framework is required to integrate different component technologies effectively within a single formalism and to make the mechanisms conceptually understandable as well as adaptable to diverse application systems.

To meet this need, this paper presents an object-oriented database model for a change management framework that can manage dependency relationship between transient user views and persistent shared objects in databases and coordinate change propagation between the two in workgroup computing systems. More precisely, by adopting an ODBMS as an underlying platform, the proposed framework attempts to present a single formalism in which existing component technologies are seamlessly integrated and, thus, structural and behavioral constructs of the framework are coherently provided. From a structural perspective, both transient and persistent transactional objects shared in workgroup systems are uniformly accommodated in the framework without requiring any conceptual transformation processes. Additionally, a two-tiered dependent dictionary mechanism is introduced to manage the dependency relationship between shared objects and their local dependent user views as well as external user views that are distributed over the network. From a behavioral perspective, the dynamic operations of the change management mechanisms, involving dependency management and change notification, are embedded into both the shared objects and dependent user views. They are also extended to client-server based workgroup computing environments by accommodating interprocess communication facilities and remote procedure calls. Finally, to facilitate reuse of the framework, the change management constructs are made generically with easily inheritable properties so that it can be adapted to a wide variety of workgroup application domains and computing architectures. A prototype system is developed based on a commercial ODBMS called OBJECTSTORE [20] with the C++ programming language.

This paper is organized as follows: Section 2 identifies the requirements of workgroup systems and focuses on the change management aspects of these systems. Section 3 describes the basic constructs and core mechanisms of a change management framework, including dependency management and change notification mechanisms. Section 4 describes the integration strategy of these constructs and mechanisms with client-server computing
components using an object-oriented approach. It then implements change management framework in a client-server computing environment. Finally, Section 5 summarizes and provides conclusions regarding the proposed change management framework.

2. Change Management Requirements in Workgroup Systems

Core components of workgroup systems are frequently associated with common tasks and a shared environment regardless of the application domain of the systems. Common tasks usually consist of a set of shared objects while a shared environment consists of multiple user views that are created to represent and visualize the shared objects. In general, common tasks and a shared environment in workgroup systems can be taxonomized by several dimensions including time (synchronous vs. asynchronous), space (same place or different places), size of the group, and application-level [11, 12, 22, 23]. Under such circumstances, a number of requirements can be highlighted with respect to change management.

- Support for multi-user views

In workgroup systems, multiple users cooperatively access shared objects through user views at individual workstations. Systems often can enrich cooperation among users by facilitating sharing of identical views across different users. Cooperating users also want to view identical objects in different ways. For example, in concurrent engineering design systems, designers' view of a product could be different than that of an account concerned with controlling the overall cost of products. Thus, the systems should allow tailoring of views on the basis of personalized preference and working practices. Users also want to manipulate shared objects at different workstations and to dynamically change views in those workstations.

- Support for coordinated modifications of shared objects

In workgroup systems, cooperating users may wish to simultaneously modify shared objects. This requires the system to allow changes of objects in a concurrent way and to maintain object integrity throughout the modification process. Thus, the system should support
database transaction management mechanisms that allow the concurrent modifications and ensure that shared objects remain consistent over the lifecycle of the workgroup computing sessions.

- Support for client-server computing

  The needs of client-server computing also arise in workgroup systems due to distributed decision making inherent to the workgroup activity (such as group scheduling) or due to the distribution of expertise and problem solving capabilities within an organization (such as concurrent engineering design or medical diagnosis). In such client-server computing environments, user views are distributed over multiple client systems and are connected to a set of shared objects stored in a single server. For improved productivity, the change management framework also needs to be operational in a client-server computing environment.

- Support for different but synchronized views

  In workgroup systems, since the shared objects are kept consistent, users want their views to be synchronized and consistent with the shared objects all the time. To enable this, the system should allow user views to be notified immediately if the underlying objects change and to facilitate automatic update on the views following the changed objects. Moreover, since user views differ in their components and formats, the system should allow individualized update mechanisms on user views and ensure that different user views are automatically synchronized according to their individual styles.
3. Conceptual Constructs and Mechanisms of the Change Management Framework

To accommodate the requirements outlined in the previous section, this section presents the design of the change management framework. Firstly, the basic constructs and mechanisms involved in the framework are described from a logical viewpoint. Secondly, an object-oriented database model of the framework is presented in order to describe the physical details of the logical constructs and their internal mechanisms.

3.1. Basic Constructs of Change Management

The operational aspects of workgroup systems (including creation, deletion, update, and synchronization of user views) and of coordinated modification in workgroup tasks usually have two key components: shared objects and their dependent user views. To adopt the two components in a generalized way, the change management framework uses two primary structural constructs: supporter and dependent. A **supporter** is an object to be referenced and modified as a basic shared source. 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 runs as a single program in the operating system. Since the dependent and the process containing the dependent are usually located together, we also call them **depending objects** in order to designate the two universally. When a process creates a dependent in association with a supporter, both the process and the dependent are registered as depending objects of the supporter. A **dependency pair** designates an individual pair of a supporter and the set of depending objects that rely on the supporter, whereas a **dependency relationship** means the whole set of the dependency pairs. Since the individual dependency pairs dynamically change during the sessions of the workgroup computing, data integrities within the dependency relationship are important for securing consistency. Thus two kinds of integrity constraints are enforced, namely, referential integrity and object integrity.

Referential integrity enforces a dependent to be existent only in the presence of its supporter. If the supporter is removed, the dependent disappears immediately. Similarly, if a dependent is removed, the supporter will not refer to the dependent any longer. Meanwhile,
the object integrity enforces dependent user views to remain synchronized with the current status of the supporter. If the supporter is changed, its dependents will update themselves accordingly. Basically a change management framework addresses the mechanisms that are geared to maintain the dependency relationship as well as to support these two kinds of integrity constraints. Specifically, two mechanisms are provided: a dependency management mechanism and a change notification mechanism. The former primarily maintains the dependency relationship and the referential integrity while the latter preserves the object integrity. For example, consider the engineering design shown in Figure 1.

INSERT FIGURE 1 HERE.

On the left-hand side of Figure 1-a, the tree data structure configuring a bicycle is a supporter as a shared object. On the right-hand side, several representations of the tree structure exemplify different users' views of the same supporter: a product view with a graphic of a bicycle at the top, an indented textual list view in the middle, and a cost management accounting view calculating the overall cost of the product's components at the bottom. These three views become the dependents of the tree structure supporter. Considering that the three views can be maintained in multiple processes, the supporter will have more than three depending objects including the dependents and their associated processes. Conversely, a depending object could reference multiple supporters. As a whole, the cardinality semantics between supporters and depending objects make a many-to-many association. Once the dependency relationship is established as such, the dependency management mechanism administers the registration of the current dependent views in regard to the tree structure. As well, the change notification mechanism ensures that the dependent views consistently mirror the current valid image of the tree structure by continuously reflecting the changes made to it.

3.2. Core Mechanisms For Change Management
This section describes an object-oriented database model developed with mechanisms required for a change management framework. Specifically, three mechanisms, dependent dictionary, dependency management, and change notification, are described in detail regarding their physical implementation. In order to give a concise description of the data model, a very simplified version of class definitions rather than an actual implemented version are provided but their functional specifications cover all the required core details, ranging from a transient computing environment to a database transactional computing environment. In delineating the data model, we use graphic notations borrowed from the Object Model Technique (OMT) [24] and consider an ODBMS as an implementation platform. Brief descriptions of the OMT notations are as follows: The notations use boxes and lines to respectively depict classes of objects and relationships between the objects. In a class box, there are three rows: the top row denotes the name of the class, the middle row displays the attributes of the class, and the bottom row represents all the operations. Specifically, abstract operations that are more customizable in the subclasses are declared within braces. A subclass relationship between two classes is depicted as a line of triangles. Cardinality constraints are pictorially represented as the endpoints of a line depicting a relationship. As a presentation convention, the name of the class is capitalized (e.g., Dependent) while the name of the operation is in boldface (e.g., addDependent).

INSERT FIGURE 2 HERE.

First, as outlined in section 2, user views are very volatile over computing sessions by being registered and removed dynamically at the users’ convenience. Dependent dictionary, defined by Dependent Dictionary, is a dedicated tool set for managing such dynamically changing dependency relationship in a general way (see Figure 2-a). Developed on the many-to-many cardinality semantics of the dependency pairs between the supporter and dependents, it allows dynamic registration, retrieval, and deletion of both supporters and dependents. Since it receives an abstract superclass as the Dependent class, it can accommodate diverse types of dependents, ranging from textual documents to multimedia
data. Meanwhile, to avoid performance degradation in the presence of a large amount of dependency pairs, a hash table taking the supporter as a key object is specifically embedded as a primary search method. In brief, the dictionary serves as a low-level but self-contained general tool that makes the development of high-level dependency management and change notification mechanisms more simplistic.

Second, the Supporter class incorporates the dependent dictionary as an intrinsic tool and provides various dependency management operations which are internally undertaken by the dictionary. Figure 2-b shows the many-to-many dependency relationship and its associated dependency management operations. For instance, registering of a new dependent for a supporter is performed by executing the addDep of the supporter. Retrieval of the dependent set of a specific supporter is performed by simply executing getDep while retrieval of the supporter set of a current dependent is conversely done by executing getSup of the Dependent. The remove of Dependent performs deletion of the dependent from all of its supporters. Specifically, the dependent dictionary employed here is termed an internal dependent dictionary since the dependency relationship is concerned with only the dependents created in the local process.

Third, the remaining operations in the Supporter and Dependent are concerned with the change notification mechanism between a supporter and its dependent set. Since the change notification mechanism involves several interactions among supporters, dependent dictionary, and dependents, we illustrate the mechanism using an event trace diagram as shown in Figure 3. The event trace diagram has been used extensively in the world of telecommunications because it effectively describes the sequence of events and the objects exchanging events [21, 24]. In the diagram, an individual participating object and event are represented respectively by a vertical line and a line of horizontal arrows from the sender object to the receiver object. Time goes downwards, but the spacing is irrelevant. Only the sequence of events is shown. To change a user view, the user may invoke a change operation to the supporter which sends a change request to it. When the supporter finishes the change operation, it subsequently broadcasts its changed status to all dependents by invoking the changed of the Supporter and sending a "changed" message to the dependent
dictionary. The aspect argument in the `changed` can be optionally passed to specify the details of the change made in the supporter. Upon receiving the message, the dictionary inside the Supporter looks up the dependency pair with respect to the current supporter and returns the dependent set. When the corresponding dependent set is identified, the supporter sends update request messages to all the dependent set by executing `sendUpdate`. Upon receiving the update requests from the supporter, the dependents individually execute `update` to modify their status accordingly, and the change notification process is completed.

With respect to the update request, two strategies can exist: immediate update and deferred update. The immediate update strategy implemented in the current prototype system reflects every change made to a supporter to its dependent, one at a time, right after the change is made. Benefits of the immediate update strategy include ease of implementing the strategy and real-time based synchronization of the dependents views with their supporters. However, the strategy has several shortcomings, the most serious being the communication control overhead associated with update broadcast and inefficiency of recurring individual updates enforced to the dependents. On the other hand, the deferred update strategy defers the request until a direct or indirect order is given to the dictionary and, thus, can avoid communication control overhead. This strategy is effective when high communication overhead is expected as in a workgroup computing environment where a large number of dependents are manipulated in association with supporters that are changed frequently. The other significant benefit of this strategy is the efficiency of batching and processing the updates when needed, instead of one at a time. Since the appropriate strategy is, thus, application system-dependent, actual implementation of the strategy in the change notification mechanism can be later chosen by the system designer’s needs.

INSERT FIGURE 3 HERE.

So far, we have assumed that the supporters in the change management framework are transiently created and maintained in the memory. When the supporters are to persistently outlive the program sessions and are to be concurrently accessed by multiple users, the
A database transaction mechanism should be involved in the change notification mechanism. More precisely, since multiple change operations could be performed simultaneously on the same supporters, causing conflict between one another, or could terminate unsuccessfully, the change operations should be performed inside a transaction to ensure database consistency. Since ODBMSs support such transaction management by ensuring consistency, permanence, serializability, and recovery [20, 21], a change operation on a persistent supporter needs to be envisaged with a transaction and protected with a write-lock so that other processes cannot write to the page on which the supporter resides. Under transaction management, the change notification to its dependents becomes meaningful only if the operation has committed. Otherwise, dependents need not be aware of the change attempt since supporters do not change at all after the transaction. A more practical case occurs when a change operation consists of a series of sub-change operations like moving a part from one place to another. To ensure database consistency, all the change operations are performed in a single transaction with overall write-lock. In this case, changes made by individual operations cannot be propagated immediately to the dependents since some of them may result in unsuccessful changes due to unexpected system failures. Thus, the individual changes need to be accumulated and delayed until the whole transaction is complete and to be propagated only when the transaction reaches a successful commit state.

To complement the change notification mechanism within a transaction management environment and to accommodate such delayed changes accumulated in a transaction, the Delayed Change class is additionally provided (see Figure 2-b). It performs two types of tasks. As an object collection manager, it stacks individual changes accumulated in the transaction; as an individual change object, it serves as an abstract messenger object of a persistent supporter by delivering directions about how, or what, to change to the dependents. A delayed change is created when a change operation is performed on a persistent supporter in a transaction. When a change operation of the supporter is executed, the persistence of the supporter is first checked. If the supporter is persistent, a corresponding delayed change is automatically created in association with the supporter and registered in a list; if the supporter is transient, the dependent set is immediately notified of the change. Thus, a transaction
containing multiple change operations creates a list of delayed changes, which are increased during the transaction and released or removed depending on the final status of the transaction.

3.3. Adoption of Change Management Mechanisms By Inheritance

The Supporter and Dependent definitions illustrated in Figures 2-a and 2-b are made generic and minimal by supporting only dependency maintenance and change notification. As superclasses, they should be inherited into specialized supporter and dependent subclasses. Application-specific attributes and operations can also be added to these subclasses. In addition, the operations in the Supporter and Dependent can be flexibly customized to suit a particular application environment since most operations are declared as abstract operations. Figure 2-c shows an example of inheritance hierarchy of the Support and Dependent. In the figure, the tree structure of Figure-a inherits the property defined in the Supporter while its dependent views, including the product view, indented list view, and cost accounting view, inherit Dependent. Both subclasses will add their own attributes and operations to the superclasses.

Consequently, such class inheritance leads to the 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. Moreover, division of these tasks provides the following benefits to the subclass layers. First, once the Supporter class is inherited to a subclass, every dependent of a subclass is automatically registered in the dictionary whenever it is created. However, the dependency maintenance mechanism will not be remembered since it is totally hidden from the subclass. Second, when the supporter of a subclass is changed, the communication protocol becomes more likely a message broadcasting protocol than a one-to-one communication protocol since 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, Supporter. Thirdly, simplicity achieved by the encapsulated change notification mechanism makes the
whole process transparent. That is, when a user changes a supporter, all the dependents automatically adjust themselves according to the nature of the change.

3.4. An Example of Collaborative Model Development

Large-scale model development in management science area usually involves various parties and levels of management ranging from the owners of problems to policy makers, and from professional modelers to casual model users who simply execute the models and obtain solutions as needed. These various constituencies of mathematical models work collaboratively by sharing common models. Depending on their individual interests, roles, needs, and modeling skills, however, they differ in their representations of the problems. Figure 4 provides a collaborative model development example for a mathematical transporation model. For the transportaion model structure depicted in Figure 4-a, problem owners and policy makers, for instance, tend to prefer conceptual views such as Figure 4-b, which illustrates the influence relationships among core model constructs, while skilled modelers with extensive operations research and management science knowledge and modeling proficiency use formal and precise modeling languages as represented in Figure 4-c. Still, casual model users employ simplistic institutionalized views of the models they execute daily. Figure 4-d represents such an simplified view in a netform. In order to accommodate such a variety in the representations of the models, a mechanism to allow multiple views of the same model is necessary.

Insert Figure 4 here.

Simultaneously, such a model shared by multiple users undergoes a life cycle. Changes in the model take place as its model structure and the corresponding data sets change due to the dynamic nature of the collaborative model development process, evolutionary operating environment, and the inherent uncertainty associated with the problems. In such a collaborative workgroup environment, the change management mechanisms should convey the changes made to the model to the user views and help to avoid the inconsistency between the revised model and the views of some users and to improve the communication between the
model users, and ultimately, to increase the efficacy of the multi-user modeling environment. Figure 4-e shows an inheritance class diagram between the model structure supporter and its view dependents. Detailed discussion about the model management system is provided in [16]

4. Extension of the Change Management Framework to Client-Server Computing

In this section, we extend the change management framework to a computing environment. We first review the similarity between the Object-Oriented Paradigm (OOP) and the client-server computing environment, and second extend the framework for this environment.

4.1. Merge of Client-server Computing and Object-Oriented Paradigm under the Change Management Framework

Though research concerning client-server computing and OOP have evolved separately over the years and use different terms, they are actually symbiotic in nature and share several principles [25, 27]. Table 1 lists the mapping terms between client-server computing and OOP and shows how they are utilized in the object-oriented database model of the change management framework.

<table>
<thead>
<tr>
<th>Client-Server Computing</th>
<th>Object-Oriented Paradigm</th>
<th>Change Management Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>process (client and server)</td>
<td>object</td>
<td>Change Manager Client and Change Manager Server Object</td>
</tr>
<tr>
<td>protocol</td>
<td>interface</td>
<td>TCP/IP</td>
</tr>
<tr>
<td>communication links</td>
<td>messages</td>
<td>socket programming tool</td>
</tr>
<tr>
<td>function</td>
<td>operation</td>
<td>encapsulated remote procedure call</td>
</tr>
</tbody>
</table>

Table 1. Mapping Terms Between Client-Server Computing and Object-Oriented Paradigm and Their Use in Change Management Framework.
Based on Table 1, we described the mapping concepts between client-server computing and OOP and the methods used to integrate them.

- **Process vs. object.** In client-server computing, a process is an entity which executes a program on a processor; it is, thus, a unit of execution. The processes are usually either clients or servers. Clients request service from servers and servers perform the requested service by executing a single or a set of corresponding functions. From the viewpoint of OOP, a process can be thought of as an object since an object is self-contained and independent, encapsulating its own operations and data. Thus, the client and server processes in an application are implemented by the objects that play different roles. In our change management framework, both processes are respectively implemented as Change Manager Client and Change Manager Server objects. In the following sections, we call client and server interchangeably with client process and server process respectively. In terms of the dependency relationship between the supporter and its depending objects, individual Change Manager Client objects become the depending objects while the Change Manager Server object becomes the supporter.

- **Protocol vs. interface.** In client-server computing, to facilitate communications between net devices, various protocols such as the Open Systems Interconnection model, the Transmission Control Protocol/Internet Protocol (TCP/IP), and the IBM Systems Network Architecture exist. In OOP, the protocol can be expressed as an interface of an object. These interfaces can define both low-level protocols, such as required sets of parameters, and high-level application protocols, such as X.400. The change management framework employs a connection-oriented protocol of UNIX TCP/IP and implements inter-process communication using the interface definition of the Change Manager Client and Server.

- **Communication links vs. messages.** In client-server computing, applications interact by using communication links of varying descriptions including Sockets of TCP/IP, LU 6.2 in
SNA as well as other application program interfaces. In OOP, objects communicate with one another by exchanging messages that contain requests for services and responses to those requests. The message corresponds to one in Smalltalk or invokes an operation in C++ or Eiffel. Messages can be encoded by various communication links and, thus, simplify the interaction mechanisms among the objects. In the change management framework, messages adopting a socket programming tool are utilized to request service between client and server.

- Function vs. operation. In client-server computing, functions are processes that perform particular actions on data such as calculations or database queries. Using remote procedure calls (RPC), a client process can request a service performed by a function stored on a remote server. In OOP, the object operations are the functions. Specifically, by encapsulating the implementation details of the RPC, the object request for remote services can be performed by executing a local operation of the object. In the change management framework, message exchanging facilities that request RPCs between the Change Manager Client and Server objects are all encoded as local operations of the two objects. In addition, functions required to facilitate the dependency relationships and change notification mechanisms are all implemented in the encapsulated operations of the Change Manager Client and Server objects. For convenience, the Change Manager Client and Server objects will be alternately referred to as client and server respectively. More detailed discussion follows later.

4.2. A Two-Tiered Dependency Relationship

In the client-server computing environment, the change management framework becomes more complex since it has to deal with the dependency relationship between client and server in addition to that between supporter and dependent. In a typical client-server computing system, the Change Manager Server resides in the server computer and performs the clients' requests for persistent shared supporters in the server database. Meanwhile, the dependents of a supporter that are distributed over the multiple client systems are incorporated
by Change Manager Clients individually running in the client systems. Figure 5 shows an example of a client-server computing system which consists of one server system and four client processes that are distributed in different systems. In the figure, persistent supports A, B and C are stored in the ODBMS of the server, and their dependents are located in the clients CL_1, CL_2, CL_3, and CL_4. From the viewpoint of a particular client, the dependents of a supporter can be classified into two types depending on where they are located; dependents located in the client of particular interest are called **internal dependents** while those in other clients are called **external dependents**. For the client CL_1, the internal dependent of the supporter B is b_1, whereas the external dependents of the same supporter include b_2 and b_3 in CL_2 and b_4 in CL_3. At the same time, the other clients, CL_1, CL_2 and CL_3 become the depending objects of supporter B since they all contain the dependents of the supporter.

To manage both the external and internal dependents effectively, a **two-tiered dependency relationship** is provided 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 supporter and its internal dependents, each client contains an internal dependent dictionary like that described in Section 3.2. As shown in Figure 5, all the clients from CL_1 to CL_4 have their own internal dependent dictionaries. However, the internal dependents contained in the individual clients are invisible to the server; thus, the server is not required to remember all the dependents, only the clients containing them. In contrast, the server is provided with an **external dependent dictionary** which uses a client as a depending object of the Dependent Dictionary and maintains dependency pairs at the upper tier between a shared supporter and its set of depending clients. In Figure 5, the external dependent dictionary shows how the three supporters A, B, and C are associated with the depending clients CL_1 to CL_4. The external dependent dictionary primarily relieves the clients from having to remember 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 helps those clients to trace further and return their internal dependents. If the
depending clients are identified for a supporter, their internal dependents can be traced further at the lower tier by the internal dependent dictionaries of the individual clients. In brief, the two-tiered dependency relationship is designed to divide, and make abstract, the management of dependency relationship into two independent modules so that each tier can focus on a specialized task and handle it autonomously and efficiently. At the same time, both tiers cooperate with each other like a cascaded system in order to facilitate the smooth management of dependency relationship in a client-server 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 6. The Change Manager Server performs three types of primary operations: communication connections, maintenance of the external dependent dictionary, and change notification and propagation. First, a full communication connection between the server and a client is established by a client that needs service from the server. The client communicates with the server by sending and receiving RPC messages. Second, the server maintains the external dependent dictionary, using the dependency attribute, and stores the upper tier dependency pairs consisting of supporters and their clients. More precisely, when a client creates a dependent for a specific supporter, it sends the server RPC messages requesting corresponding dependency management operations in the server. Upon receiving these messages, the server performs the requested operations such as addSup and removeSup to add or delete the current client as a dependent of the server. 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 change notification. It first identifies all the depending clients with the help of the external dependent dictionary and second propagates the messages notifying the delayed change to the clients.

The Change Manager Client performs counterpart operations to the three types of operations of the Change Manager Server since the two objects are interacting through the change management mechanism. First, the client determines the host system for communication with the server and sends the server a request for connection. Second, concerning the dependency relationship, the client may ask the server to modify its external dependent dictionary as soon as the client is related to a new persistent supporter. In this case, the client requests the server to declare itself as a depending client of the supporters by
sending a RPC. Third, the Change Manager Server invokes the corresponding dependency maintenance operations such as addSup and removeSup to update the external dependent dictionary. Fourth, when a client commits a change on a persistent supporter, it 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 delayed change is notified to the dependent clients. Finally, using readMsg, individual clients receive the change updating message and process it by performing appropriate change update operations on their internal dependents.

INSERT FIGURE 6 HERE.

The Change Manager Server and Client are the two primary constructs of the change management framework needed to facilitate dependency maintenance and change notification within a client-server computing environment. Designed as generic objects, they can also be used as primary building blocks for developing diverse workgroup computing systems, which may include persistent supporters and distributed dependents.

4.3. Change Notification Process Under a Two-Tiered Dependency Relationship

In this section we detail the steps of the change notification process under the client-server computing environment by providing an event trace diagram describing the interactions made by the server and the client with the two-tiered dependent dictionaries. Figure 7 is an extension of the event trace diagram shown in Figure 3, but here it has a two-phased change notification process. In the diagram, we assume all the communication connections between the server and clients and all the dependency relationships between supporters (i.e., transient or persistent supporters) and their depending objects (i.e., both internal and external dependents and depending clients) are already established; thus, both the internal and external dependent dictionaries are established.

INSERT FIGURE 7 HERE.
A client can modify a dependent of a supporter by submitting a change operation to the supporter. The first phase is concerned with change management tasks inside a client and focuses on change notification to internal dependents. When a change request is sent to a supporter ①, the actual change operation is performed inside a database transaction ②. If the supporter is persistent, a corresponding delayed change is created and is cumulatively saved in a list of delayed changes ③ until the current transaction is committed successfully. When the transaction is committed ④, the entire list of delayed changes is sent to both the client itself and the server for change propagation to internal and external dependents ⑤. The change update task in the internal dependents of the current client is almost identical to the case of the transient supporter described in Section 3.2, except that the change update is repeated over the list of supporters that are changed ⑥. The second phase is concerned with change management tasks outside the client, primarily change propagation to the external dependents. The second phase is performed in two steps. In the first step, the list of delayed changes is sent to the server, which identifies clients that need synchronization by checking the external dependent dictionary ⑦. It then generates a set of individualized lists of delayed changes and sends them to these clients ⑧. Each client is, therefore, notified of relevant delayed changes. In the second step, clients individually receive the list of delayed changes, extract changed supporters and repeat the same procedure as that of the change update of the internal dependents ⑨. A detailed change notification algorithm regarding ⑦, ⑧, and ⑨ is provided in Figure 8. It shows the implementation of the propagate operation of the Change Manager Server.

In Figure 5, suppose that the dependents, \(a_1\), \(b_1\), and \(c_1\), are modified in the client \(CL_1\) and thus the supporters, A, B, and C, are changed in a serial order. In the first phase, their delayed changes, \(D_1\), \(D_2\), and \(D_3\), are created cumulatively until the transaction is complete. Then, the accumulated list of delayed changes, \((D_1, D_2, D_3)\), is sent to both the client \(CL_1\) and the server. Upon receiving this list, \(CL_1\) refers to its internal dependent dictionary to identify the internal dependents that need change updates with respect to the individual delayed changes. Consequently, the internal dependents, \(a_2\) and \(c_2\), are retrieved in association with \(D_1\).
and D₃ and are subsequently enforced to execute corresponding updates. In the second phase, change propagation to external dependents is carried out. The server looks up the external dependent dictionary with respect to the supporters, A, B, and C, encapsulated in the list, (D₁, D₂, D₃), and identifies CL₂, CL₃, and CL₄ as affected depending clients. The server then regenerates the multiple lists of the delayed changes. Subsequently, CL₂, CL₃, and CL₄ receive the individualized lists, (D₁, D₂), (D₂, D₃), and (D₁, D₃) respectively. Using the list of the delayed changes, each client looks up its internal dependent dictionary and updates the local dependents in the same way as client CL₁ did with its internal dependents.

In a formal sense, the rearrangement and list generation process of steps ‡ and ¶ in Figure 7 can be assimilated by employing a simplistic matrix handling process. Firstly, the dependency relationship maintained in the external dependent dictionary can be viewed as dependency matrix, E, where the row and column respectively denote all the supporters and depending clients. In the matrix, the element, eᵢⱼ, represents the existence of a dependency relationship between i-th supporter and j-th client; if the value is 1, a dependency relationship exists whereas if the value is 0, a dependency relationship does not exist. Figure 9 provides the matrix representations of the dependency relationships in the external dependent dictionary of Figure 5. For instance, the first column shows that client CL₁ is dependent on all three supporters, A, B, and C, while the first row shows that the supporter has three depending clients, CL₁, CL₂, and CL₄. In such capacity, retrieving depending clients for a specific supporter out of the dependent dictionary is equal to drawing the non-zero valued elements out of the matrix row designating the supporter. Secondly, recall that in the list of delayed changes D(D₁, D₂,..., Dₙ) created after a transaction, each delayed change, Dᵢ, has a supporter, Sᵢ. From a dependency matrix view, retrieving the sublist of delayed changes destined for an affected client is same as taking the non-zero valued elements (i.e., supporters) out of the matrix column designating the client and applying the precedence order of the delayed changes. More precisely, getting the sublist that retains the precedence order in D for an affected client Cₖ is equal to generating a sublist, Dₖ(Dᵢ,..., Dⱼ), by selecting only Dᵢ and Dⱼ where eᵢₖ = 1 and eⱼₖ = 1 in the dependency matrix. After these Dₖ’s are generated for all the affected clients, they are sent to the clients one at a time and locally redispached to
the final affected dependents in the individual clients on the basis of the internal dependent dictionaries.

INSERT FIGURE 8 HERE.

The proposed change management framework can be compared with existing distributed computing architectures such as MVC, OLE, and CORBA [2, 18, 26, 31]. All these four architectures employ the object-oriented paradigm. MVC is similar to the proposed change management framework in that MVC supports dependency management and change notification. However, since it cannot support persistent shared objects and multiple concurrent users, its change notification mechanism is limited to only memory resident objects. OLE and CORBA are both middleware systems that can be integrated with the distributed object-based systems such as specific applications, ODBMSs, or file systems. To become de facto middleware systems, both architectures respectively provide cross-form application service protocols and request broker implementations to support the protocols, but do not provide individual application capabilities including the change management mechanism. Thus the change management mechanism is left to third-party object implementors as a kind of external application service. A possible implementation design in CORBA would use the implementation repository to maintain the dependency relationship between supporter object identifiers and dependents object identifiers and let a server process broadcast the change notification when supporters are changed in the external ODBMS. Briefly, in both OLE and CORBA architecture, the change management framework is not a part of their embedded service and needs to be implemented as an individual application system.

5. Summary and Conclusions

A change management framework has long been needed to provide synchronized, consistent views on evolving shared objects in workgroup computing systems. In an effort to attain this goal, this paper describes an object-oriented change management framework developed with dependency management and change notification mechanisms under a client-
server computing environment. The dependency management mechanism, based on a dependent dictionary as an underlying tool, facilitates maintenance of dependency relationship between multiple dependents and supporters (i.e., user views and shared objects, respectively) no matter whether the supporters are transient in memory or persistent in databases. The change notification mechanism is activated when a supporter is changed by broadcasting its corresponding update requests to the dependent sets of the supporter. A delayed change concept is added to adapt these mechanisms to the database transaction environment. The change management framework is extended for a client-server workgroup computing environment on the basis of two-tiered dependent dictionaries.

Several advantages are manifest in the proposed object-oriented change management framework. Firstly, by using an ODBMS as an implementation platform, the framework deals directly with both transient and persistent supporters in a single formalism. This is because the ODBMS facilitates seamless accommodation of persistent transactional objects without requiring additional codes that transforms the in-memory representation of objects to structures that can be stored in any secondary storage device. Furthermore, the ODBMS approach facilitates implementation of the dependency management and change notification mechanisms of the framework with various embedded toolsets. These toolsets include integrity constraint maintenance for managing complex relationships among objects, hash tables, highly optimized addressing schemes for fast access to persistent objects, and open network-based computing utilities. Additionally, the robust concurrency handling and stabilized transaction management capabilities of the ODBMS approach enriches the framework and eventually ensures that the workgroup computing systems adopting it are more reliable and secure under a multi-user computing environment than a proprietary file system approach.

Secondly, by employing OOP as an underlying formalism, the framework provides a set of well-organized abstract objects that can perform complex algorithms associated with dependency management and change notification mechanisms in a highly modularized fashion. Specifically, abstract objects such as Dependent Dictionary, Supporter, Dependent, Delayed Change, Change Manager Server and Client hide complex details and provide well-defined service protocol by encapsulating internal attributes and only legitimate interface operations.
Since defined objects can be the building blocks of composite objects through inheritance and containment, both higher level objects (e.g., Change Manager Server and Client) and their operations are effectively constructed and implemented from lower level objects (e.g., Dependent Dictionary, Supporter and Dependent) without increasing the complexity of the dependency management mechanisms.

Thirdly, the framework provides a set of transparencies in the supporting dependency management and change notification processes. Specifically, persistence and transaction transparencies are provided since users do not need to be concerned with the persistency of the supporters as well as concurrent access to the supporters. Locational transparency is also provided since the two-tiered dependency relationship allows users to manage both internal and external dependents simply and effectively no matter where the dependents are located.

A prototype system for the change management framework is developed on a commercial ODBMS called OBJECTSTORE[20] on a SUN-4 system using C++ programming language. Future research could take two directions. First, the change management framework can be extended into intranet-based workgroup computing systems so that intranet web server can actively provide users with synchronized views of shared tasks. Regarding synchronization of user views, current web technology has limitations since the existing Hypertext Tranfer Protocol (HTTP) is unidirectional, and thus cannot allow a web server to send messages to its web browsers with receiving requests in advance. Second, the application systems of the change management framework could be developed in the areas of office document handling, engineering design, model management, GDSS, or CSCW systems. Currently, a workgroup engineering design system is under consideration.
References


n ← number of clients
delayedChange ← complete list of delayed changes

for i = 1 to n do
begin
clients sublistD[i] ← NULL // initialize delayedChange sublists for all
end

while (delayedChange.next ≠ NULL) do // repeat for all delayed change.
begin
client[1..m] ← delayedChange[j].supporter.getDep() // retrieve the depending clients of current changed
sublistD[k] = sublistD[k] + delayedChange // supporter from the external dependent dictionary.

for k = 1 to m do // add the current delayed change to the ends of
begin
if client[k] ≠ initClient then // the sublists of all affected clients except
sublistD[k] = sublistD[k] + delayedChange // the initiating client where the change was originally made.
end
delayedChange = delayedChange.next // go to next delayed change.
end

for i = 1 to n do
begin
send (sublistD[i]) to client[i] // dispatch individual sublist to the clients
end

Figure 8. A Change Notification Algorithm under the Two-Tiered Dependency Relationship.