Supporting Update Propagation in Object-Oriented Databases*

Chengfei Liu  Hui Li  Maria E Orlowska

CRC for Distributed Systems Technology

and

School of Information Technology

The University of Queensland

Qld 4072, Australia

Email: \{liu,huil,maria\}@it.uq.edu.au

Abstract

Objects in a database are interrelated. When an update operation is applied to an object, it may also impact on its related objects, depending on the semantics of their relationships. Current OODBMSs provide no support for update propagation but hard-coding. In this paper, we study update propagation support for generic update operations in object-oriented databases. We take a declarative approach, specifying propagation policies for each identified reference attribute in classes of an object-oriented database schema. Prop-

* The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.
agation policies for generic update propagation are well defined. However, we also discover that potential conflicts among propagation policies may occur if the policies can be arbitrarily specified by a designer. Therefore, we promote the update propagation problem to a higher level, investigating possible dependencies between objects. As such, the designer only needs to specify the dependency property for each reference attribute. Propagation policies are predefined for each type of dependency. By introducing some restrictions on an object-oriented database schema, conflict-free propagation policies can be achieved. Implementation issues for update propagation support in object-oriented database systems are also addressed.

**Keywords:** Object-oriented databases; Update propagation; Propagation policy; Object dependency.

## 1 Introduction

Objects in a database are interrelated. When an update operation is applied to an object, it may also impact on its related objects, depending on the semantics of their relationships. Let us consider an example for electronic document processing [12]. Suppose that a *document* consists of a title, an author and a number of *sections*. A section in turn is composed of *paragraphs*. A document may *share* sections with other documents. A section exists only if it belongs to at least one document and can not be deleted as long as there exists a document using it. Similarly, a paragraph may be shared among different sections. For a paragraph to exist, there must be at least one section containing it and thus a document containing it. A paragraph can not be deleted individually either. Annotation paragraphs may be added to documents, however, they are not shared among different documents. For an annotation paragraph to exist, the document which it annotates must also exist.
Further, documents may contain *images* that are extracted from other files. The existence of images, however, does not depend on the documents using them. Due to these different relationships, when an update operation, say *delete*, is applied to a document object, the propagations to its related sections, annotation paragraphs and images are also different.

(i). For each of its related sections, the section is deleted if it is no longer shared by other documents. The propagated deletion of a section object may impact on its related paragraphs in the same way.

(ii). All of its annotation paragraphs are deleted unconditionally.

(iii). None of its related images are deleted.

Current object-oriented database systems provide no support for update propagation. Usually, a designer has to implement an ad hoc method in a class which does the update to the objects of the class as well as the propagation to all the related objects for each update operation. This hard-coding approach is obviously cumbersome, where the propagation rules are buried in the code, making them hard to understand and control. In addition, propagation rules can not be changed dynamically. Also, similar forms of propagation defined on different object types can not be reused.

Ideally we shall be able to support update propagation in a declarative way, such that the designer only needs to specify what to do, not how to do it. In relational database systems, efforts have been put into support for referential integrity constraints [3, 4, 15, 16]. This can be regarded as a restricted form of update propagation applying to the *insert*, *update* and *delete* operations by specifying referential integrity rules. However, referential integrity rules are not sufficient to specify all update propagation rules. For instance, in the above electronic documents example, the propagation rule of deletion on document objects to section objects across the relationship *Share* is hard to express using referential integrity rules. This propagation is somehow restricted, i.e., only the last document object which
shares a section object will propagate the deletion to the section object. Since a document may contain a set of sections, and a section may be shared by many documents (i.e., not dependent exclusively to a document), the relationship Share between document objects and section objects is a $m:n$ relationship. In relational database systems, an extra relation must be introduced to represent a $m:n$ relationship. Therefore, in the example, a relation called Doc_Share_Sec is defined to reflect the relationship Share between document objects (modelled as Document relation) and section objects (modelled as Section relation).

\[ \text{Document(title, author, \cdots)} \]
\[ \text{Section(section#,\cdots)} \]
\[ \text{Doc\_Share\_Sec(title, section#, \cdots)} \]

The primary key of each relation is underlined. There are two referential integrity constraints:

\[ \text{Doc\_Share\_Sec.title} \subseteq \text{Document.title} \]
\[ \text{Doc\_Share\_Sec.section#} \subseteq \text{Section.section#} \]

As we may observe, what referential integrity rules can specify in this scenario is limited to the impact on the tuples of the relation Doc_Share_Sec when an update operation is applied to either Document tuples or Section tuples. They can not specify what should be done to Section tuples when the update operation is applied to Document tuples across the relation Doc_Share_Sec.

Much work has been done in representing and enforcing integrity constraints in object-oriented database systems [7, 9, 6, 10, 5]. Although update propagation could be considered as a case of integrity constraint maintenance, the specification of update propagation rules, however, is slightly different from specification of integrity rules. An update propa-
gation rule emphasises how the database can evolve from one valid state to another, while an integrity rule emphasises what are valid states of databases.

Rumbaugh [17] proposed a simple mechanism for controlling operation propagation across relationships for general object-oriented programming languages. It is based on associating propagation attributes for particular operations with the relationships. However, this work has not addressed update propagation in object-oriented database systems specifically. There is no concern about potential conflict in propagation attribute declarations.

In this paper, we focus on update propagation in object-oriented databases. We realize that automatic system support for application-specific update propagation is hard to achieve. Therefore, our discussion is concentrated on propagation of generic update operations. Similar to [17], propagation policies are specified for each generic update operation on reference attributes in all classes of an object-oriented database schema. Possible update propagation policies are carefully studied. However, we recognize the potential policy conflict problem if policies can be arbitrarily specified by a designer. Therefore, we promote the update propagation problem to a higher level. We investigate possible dependencies between arbitrary pairs of objects such that the designer needs only to specify the dependency property for each reference attribute. Propagation policies for each type of dependencies are predefined. With some restrictions on an object-oriented database schema, the system will generate propagation policies automatically and guarantees conflict-free propagation.
2 Update Propagation in OODBs

2.1 Object-Oriented Databases

Object-oriented databases [11, 2] were proposed to meet the needs of advanced database applications, such as CAD/CAM, software engineering, spatial databases and multimedia databases, where applications require more complex structures for objects, new data types and the need to define nonstandard application-specific operations. In object-oriented databases, any real-world entity is uniformly modelled as an object. Each object has a state and a behaviour associated with it. The state of an object is defined by values of its attributes and the behaviour of an object is specified by the methods that operate on the object state. They are encapsulated together and accessed or invoked from outside the object through explicit message passing or function calls. An object is uniquely identified by a system-generated object identifier (OID) which, unlike the primary key in relational databases, can not be modified by applications. The value of an attribute of an object is also an object in its own right, which can be both primitive and non-primitive objects. A non-primitive object in turn consists of a set of attribute values. Furthermore, an attribute of an object may take on a single value or a set of values. In contrast, the relational data model only allows an attribute to take on a single primitive object as value. Objects with the same set of attributes and methods are grouped in classes. The intentional notion of a class corresponds to an Abstract Data Type definition.

**Definition 1** A class named as $C$ is defined as

$$ C = \langle C, \{C_{sup}\}, \{A : C_A\}, \{M : \{P : C_P\}\} \rangle $$

(a) $C$ is the extent of $C$, $C = \{o\}$ where $o$ is an object (instance) of $C$.

(b) $\{C_{sup}\}$ is the set of super-classes of $C$. 

6
(c) \( \{ A : C_A \} \) is the set of attribute definitions of \( C \), where \( A \) and \( C_A \) represent name and domain type of an attribute, respectively. The value of attribute \( A \) of an object \( o \) is expressed by \( o.A \).

(d) \( \{ M : \{ P : C_P \} \} \) is the set of method definitions of \( C \), where \( M \), \( P \) and \( C_P \) are method name, parameter name and parameter domain type, respectively.

An object-oriented database schema \( S \) is defined by a set of classes

\[
S = \{ C_1, C_2, \ldots, C_n \}
\]

where \( C_i (1 \leq i \leq n) \) is a class name.

For the discussion of this paper, we are only concerned with the extent and attributes of a class. Attributes can be classified as two types: literal attributes and reference attributes. A literal attribute takes immutable object(s) as value, while a reference attribute takes mutable object(s) as value. As defined in ODMG-93 object model [2], a relationship between objects of two classes \( C_i \) and \( C_j \) in object-oriented databases can be expressed as a pair of reference attributes (traversal paths as in ODMG-93) \( A_{ref} \) and \( A'_{ref} \), where \( A_{ref} \) is a reference attribute of \( C_i \) with \( C_j \) as its domain type, and \( A'_{ref} \) is a reference attribute of \( C_j \) with \( C_i \) as its domain type. \( A'_{ref} \) is called the inverse reference attribute of \( A_{ref} \) and vice versa. To emphasise the reference relationship, we use the following notation to represent a reference attribute \( A_{ref} \) of class \( C_i \) with class \( C_j \) as its domain type,

\[
A_{ref} : \mathcal{C}_i \rightarrow \mathcal{C}_j
\]

where \( \mathcal{C}_i \) and \( \mathcal{C}_j \) are extents of \( C_i \) and \( C_j \) respectively. If \( A_{ref} \) is a single-valued reference attribute, \( A_{ref} \) of every object in \( \mathcal{C}_i \) takes one object (i.e., its OID) in \( \mathcal{C}_j \) as its value. If \( A_{ref} \) is set-valued, \( A_{ref} \) of every object in \( \mathcal{C}_i \) can take a set of objects (OIDs) in \( \mathcal{C}_j \) as its value. For the purpose of this discussion, we treat a single-valued reference attribute as also taking a set as value, but the set can hold at most one OID.
The directional representation contrasts to the in-directional key value based representation of the relational model. In relational databases, a relationship is usually represented as a referential integrity constraint of primary key values in a relation \( R_i \) and foreign key values in another relation \( R_j \). If the relationship is of \( m:n \), an extra artificial relation \( R_e \) must be created to represent the relationship, with referential integrity constraints between \( R_i \) and \( R_e \), and between \( R_j \) and \( R_e \). There is no direct constraint between \( R_i \) and \( R_j \). This is why an update operation fails to be propagated either from tuples of \( R_i \) to \( R_j \) or from tuples of \( R_j \) to \( R_i \). However, the \( m:n \) relationship can be expressed as a pair of set-valued reference attributes in object-oriented attributes, without introducing an extra class.

In order to simplify the representation of the update propagation problem in object-oriented databases, we assume the so-called relational integrity constraints [9] always hold for any relationship, i.e., if a reference attribute \( A_{ref} : C_i \rightarrow C_j \) exists in class \( C_i \), the inverse reference attribute \( A'_{ref} \) of \( A_{ref} \) also exists in the class \( C_j \).

In the electronic document processing example, four classes \( \text{Document}, \text{Section}, \text{Paragraph}, \text{Image} \) can be defined for the object-oriented database schema \( S_{doc} \), i.e.,

\[
S_{doc} = \{ \text{Document, Section, Paragraph, Image} \}.
\]

In the following, we give type definitions of these four classes. Every reference attribute has an inverse reference attribute specified by an inverse clause.

class Document

type {
    title: String;
    author: String;
    content: Set<Section> inverse Section: shared_by;
}
2.2 Problem Specification

The update propagation problem can be stated as follows:

When an update operation is applied to an object, the update is not just applied to the object only, it may also propagate to its related objects across its relationships to these objects according to
application-specific rules, until no new objects can be reached.

Usually update propagation is not symmetric, propagation on an object at one end of a relationship may not be the same as that at the other end.

In most OODBMSs, an application designer has to implement a method for each update operation as well as its propagation for each class, no matter how similar the operations are. The semantics of an update operation on individual objects is mixed with the semantics of propagation along the relationships. As we may observe, relationships between objects are the main reason of propagation of a generic update operation. The selection of which objects the operation should propagate to can be separated as a property of the relationship between objects, rather than a property of the operation as a whole. As discussed above, a relationship in OODBs can always be represented as a pair of reference attributes. So it is possible for designers to simply declare a propagation policy on each reference attribute for each type of update operation. Given an object-oriented database schema, we define a reference graph to reflect all relationships between classes in the schema. Based on a reference graph, we further define a propagation graph for specifying propagation policies for a generic update operation.

**Definition 2** Given an object-oriented database schema $S = \{C_1, C_2, \ldots, C_n\}$, a reference graph of the schema $S$ is defined as a labelled directed graph $RG = (V_{RG}, E_{RG}, L_{RG})$ where

(a) $V_{RG} = S$,

(b) $E_{RG}$ is a finite set of arcs. If a reference attribute $A_{ref} : C_i \rightarrow C_j$ is defined in class $C_i$ of $S$, then an arc $e = \langle C_i, C_j \rangle \in E_{RG}$,

(c) $L_{RG}$ is a set of labels for edges by applying a labelling function $f : E_{RG} \rightarrow A \times A$, where $A$ is a set of reference attribute names. The function $f$ maps each arc $e = \langle C_i, C_j \rangle \in E_{RG}$
Figure 1: Reference graph for electronic document example

to a pair \( < A_{ref}, A'_{ref} > \in A \times A \), where \( A_{ref} \) is the name of a reference attribute \( A_{ref} : C_i \rightarrow C_j \), and \( A'_{ref} \) is the name of the inverse reference attribute of \( A_{ref} \).

Figure 1 shows the reference graph for the electronic document database schema \( S_{edoc} \).

**Definition 3** Given an object-oriented database schema \( S = \{C_1, C_2, \cdots, C_n\} \), a propagation graph of the schema \( S \) for a generic update operation \( U \) is defined by \( PG = (V_{PG}, E_{PG}, L_{PG}) \) where

(a) \( V_{PG} = S \),

(b) \( E_{PG} \) is a finite set of arcs. If a reference attribute \( A_{ref} : C_i \rightarrow C_j \) is defined in class \( C_i \) of \( S \), then an arc \( e = < C_i, C_j > \in E_{RG} \).

(c) \( L_{PG} \) is a set of labels for edges by applying a labelling function \( g : E_{PG} \rightarrow A \times A \times \mathcal{P} \), where \( A \) is a set of reference attribute names and \( \mathcal{P} \) is a set of propagation policies for the operation \( U \). The function \( g \) maps each arc \( e = < C_i, C_j > \in E_{PG} \) to a triple \( < A_{ref}, A'_{ref}, P > \in A \times A \times \mathcal{P} \), where \( A_{ref} \) is the name of a reference attribute \( A_{ref} : C_i \rightarrow C_j \); \( A'_{ref} \) is the name of the inverse reference attribute of \( A_{ref} \), and \( P \) is a propagation policy.
defined on $e$ for the operation $U$. Propagation policies for each generic update operation are defined later in section 2.4.

The propagation graph $PG$ of a schema $S$ for an update operation $U$ can be constructed from the reference graph $RG$ of $S$ by simply augmenting the label with a propagation policy, i.e., associating a propagation policy with each reference attribute for the operation $U$. We will show the propagation graph of $S_{edoc}$ later in this section after propagation policies for each generic update operation are introduced.

2.3 Generic Update Operations

In object-oriented databases, a generic update operation consists of deleting an object, inserting an object, and modifying an attribute value. Usually, modification of literal attribute values of an object will not impact on other objects, provided the object-oriented database is well designed and update anomalies will not occur during updating [13]. Furthermore, since a reference attribute value is an OID or a set of OIDs, OIDs can only be added or removed (i.e., adding a reference or removing a reference), but can not be modified. Therefore, the generic update operations which will result in propagation are restricted as follows:

- delete an object
- remove a reference
- insert an object
- add a reference
2.4 Propagation Policies

Each reference attribute \( A_{ref} : C_i \rightarrow C_j \) has a propagation policy defined for each generic update operation \( U \). The propagation policy specifies how to propagate the operation \( U \) from an object in \( C_i \) to its related objects in \( C_j \). Actually, a propagation rule can be either generic or application specific. Realising that automatic support for application specific update propagation is very hard, possibly nothing but triggering or hardcoding, therefore, we only address generic update propagation. After careful observation between arbitrary pairs of objects, we summarise the following generic propagation policies for the four types of generic update operations.

The propagation policies for delete object operation:

- **IndependentlyPropagate(IP)**

  If \( IP \) policy is defined on reference attribute \( A_{ref} : C_i \rightarrow C_j \), and \( \exists o_k^i \in C_i, \exists o_l^j \in C_j \) such that \( OID(o_l^j) \in o_k^i \cdot A_{ref} \), then deleting \( o_k^i \) will propagate the delete operation to \( o_l^j \).

- **DependentlyPropagate(DP)**

  If \( DP \) policy is defined on reference attribute \( A_{ref} : C_i \rightarrow C_j \), and \( \exists o_k^i \in C_i, \exists o_l^j \in C_j \) such that \( OID(o_l^j) \in o_k^i \cdot A_{ref} \), then attempt to propagate the delete operation on \( o_k^i \) to \( o_l^j \). However, if the deletion of \( o_l^j \) fails, \( o_k^i \) is not allowed to delete. This implies that the deletion of \( o_k^i \) is dependent on the deletion of \( o_l^j \). An object can not be directly deleted if the object is reserved for multiple objects. This is defined by the RV policy later in this section. An object \( o_k \) can not be deleted if either \( o_k \) can not be directly deleted or there exists another object \( o_l \) such that \( DP \) policy is defined from \( o_k \) to \( o_l \) and \( o_l \) can not be deleted.
• **RestrictedlyPropagate**(RP)

If RP policy is defined on reference attribute \( A_{ref} : C_i \rightarrow C_j \), and \( \exists o_k^i \in C_i, \exists o'_k^j \in C_j \) such that \( OID(o_k^i) \in o_k^i.A_{ref} \), then deleting \( o_k^i \) will propagate the delete operation to \( o_k^j \) only if \( \neg \exists o_h^j \in C_j \) such that \( OID(o_h^j) \in o_h^j.A'_{ref} \), where \( A'_{ref} : C_j \rightarrow C_i \) is the inverse reference attribute of \( A_{ref} \).

• **Inverse**(IV)

If IV policy is defined on reference attribute \( A_{ref} : C_i \rightarrow C_j \), and \( \exists o_k^i \in C_i, \exists o'_k^j \in C_j \) such that \( OID(o_k^j) \in o_k^j.A_{ref} \), then deleting \( o_k^j \) only causes removing \( OID(o_k^j) \) from \( o_k^j.A'_{ref} \), where \( A'_{ref} : C_j \rightarrow C_i \) is the inverse reference attribute of \( A_{ref} \).

• **Reserve**(RV)

If RV policy is defined on reference attribute \( A_{ref} : C_i \rightarrow C_j \), and \( \exists o_k^i \in C_i, \exists o'_k^j \in C_j \) such that \( OID(o_k^j) \in o_k^j.A_{ref} \), then \( o_k^i \) is simply not allowed to be deleted. This is the only policy that can cause a deletion fail directly.

The propagation policies for remove reference operation:

• **IndependentlyPropagate**(IP)

If IP policy is defined on reference attribute \( A_{ref} : C_i \rightarrow C_j \), and \( \exists o_k^i \in C_i, \exists o'_k^j \in C_j \) such that \( OID(o_k^j) \in o_k^j.A_{ref} \), then removing \( OID(o_k^j) \) from \( o_k^j.A_{ref} \) will result in removing \( OID(o_h^j) \) from \( o_h^j.A'_{ref} \) and propagating a delete operation to \( o_h^j \).

• **DependentlyPropagate**(DP)

If DP policy is defined on reference attribute \( A_{ref} : C_i \rightarrow C_j \), and \( \exists o_k^i \in C_i, \exists o'_k^j \in C_j \) such that \( OID(o_k^j) \in o_k^j.A_{ref} \), then attempt to propagate the delete operation to \( o_k^j \). However, if the deletion of \( o_k^j \) fails, \( OID(o_k^j) \) is not allowed to be removed from \( o_k^j.A_{ref} \). Otherwise, remove \( OID(o_k^j) \) from \( o_k^j.A_{ref} \) and propagate the delete
operation to $o^j_i$.

- **RestrictedlyPropagate(RP)**

  If RP policy is defined on reference attribute $A_{ref} : \mathcal{C}_i \rightarrow \mathcal{C}_j$, and $\exists o^i_k \in \mathcal{C}_i, \exists o^j_i \in \mathcal{C}_j$ such that $OID(o^i_k) \in o^j_1.A_{ref}$, then removing $OID(o^i_k)$ from $o^j_1.A_{ref}$ will result in removing $OID(o^j_i)$ from $o^j_1.A'_{ref}$ and propagating a delete operation to $o^j_i$ only if $\neg\exists o^i_h \in \mathcal{C}_i$ such that $OID(o^i_h) \in o^j_1.A'_{ref}$, where $A'_{ref} : \mathcal{C}_j \rightarrow \mathcal{C}_i$ is the inverse reference attribute of $A_{ref}$.

- **Inverse(IV)**

  If IV policy is defined on reference attribute $A_{ref} : \mathcal{C}_i \rightarrow \mathcal{C}_j$, and $\exists o^i_k \in \mathcal{C}_i, \exists o^j_i \in \mathcal{C}_j$ such that $OID(o^i_k) \in o^j_1.A_{ref}$, then removing $OID(o^j_i)$ from $o^j_1.A_{ref}$ will result in removing $OID(o^i_k)$ from $o^j_1.A'_{ref}$ only.

The propagation policies for insert object operation:

- **None(NO)**

  The insertion of an object $o^j_i$ in $\mathcal{C}_i$ is not restricted to the existence of any object in $\mathcal{C}_j$ if policy NO is specified on a reference attribute $A_{ref} : \mathcal{C}_i \rightarrow \mathcal{C}_j$ for insert operation.

- **Bound(BD)**

  The insertion of an object $o^j_i$ in $\mathcal{C}_i$ is bound to the existence of an object in $\mathcal{C}_j$ if policy BD is specified on a reference attribute $A_{ref} : \mathcal{C}_i \rightarrow \mathcal{C}_j$ for insert operation.

The propagation policies for add reference operation:

- **Inverse(IV)**

  If IV policy is defined on reference attribute $A_{ref} : \mathcal{C}_i \rightarrow \mathcal{C}_j$, then adding $OID(o^j_i)$...
in $o_i^j.A_{ref}$ will cause adding $OID(o_i^j)$ in $o_i^j.A_{ref}'$, where $A_{ref}' : C_j \rightarrow C_i$ is the inverse reference attribute of $A_{ref}$.

- **NotAllowed(NA)**

  If NA policy is defined on reference attribute $A_{ref} : C_i \rightarrow C_j$, then it is not allowed to add $OID(o_i^j)$ into $o_i^j.A_{ref}$.

  The propagation graph of the electronic document database schema $S_{doc}$ for delete operation is shown in figure 2. From the figure, we can see, when deleting a document object, its propagations to objects via different reference attributes are different.

  (i). It will independently propagate to its related section objects, the document object will definitely be deleted. A section object can not be deleted individually, it will be deleted by the last document object which references it.

  (ii). It will dependently propagate to its annotated paragraph objects. An annotated object can be deleted individually, resulting in removing the inverse reference of the document for which it annotates.

  (iii). It will just remove the inverse references of referred figure objects, not causing their deletion.

3 Policy Conflict and Prevention

3.1 Potential Policy Conflict

Specifying propagation policies for update propagation is much easier than hard-coding individual methods. However specifying a propagation policy depends on designer’s perception towards the application. It is possible that the policies declared by a designer are incompatible or conflicting.
An object may reserve(RV) references for other objects. If there exist object $o_p$ in $C_i$ reserves reference for object $o_q$ in $C_j$, $o_q$ reserves reference for object $o_k$ in $C_k$, and $o_k$ reserves reference for $o_p$, where $C_i$, $C_j$ and $C_k$ are extents of classes $C_i$, $C_j$ and $C_k$ respectively, then it will not be allowed to delete any object in the cycle. As shown in figure 3(a), such a cyclic RV policy declaration may cause potential propagation policy conflict.

Figure 3(b) shows another potential conflict that may occur. If an object $o_p$ may propagate a delete to another object $o_q$ through one path of references, while $o_p$ also reserves for $o_q$, then the propagation is self-contradictory.

In addition, it is also possible that the policies defined for one update operation are not compatible with the policies defined for another update operation. As shown in figure 4, the policy defined on reference attribute $A_{ref}$ for delete operation is incompatible with the policy defined on the inverse reference attribute $A'_{ref}$ of $A_{ref}$ for insert operation. The policy $BD$ defined for insertion says an object in $C_i$ can not exist without the existence of an object in $C_i$. While the policy $IV$ defined for deletion says there is no requirement for
propagation to objects in \( C_j \) if an object in \( C_i \) is deleted.

\[
\begin{array}{ccc}
\text{Ci} & \xrightarrow{BD} & \text{Cj} \\
\text{Ck} & \xrightarrow{RV} & \text{Cj} \\
\end{array}
\quad
\begin{array}{ccc}
\text{Ci} & \xrightarrow{IV} & \text{Cj} \\
\text{Ck} & \xrightarrow{IP} & \text{Cj} \\
\end{array}
\]

(a) insert \hspace{2cm} (b) delete

Figure 4: Incompatible policy declaration

Obviously, arbitrary policy definitions on reference attributes for same or different operations can easily cause policy conflict and incompatibility. Bearing the problems in mind, we study the semantics of relationships and propose an approach for resolving the incompatible and conflict policy declaration at a higher level.

3.2 Types of Object Dependencies

Kim et al. [12] presented a model of composite objects and studied four types of composite references. Halper et al. [8] and Artale et al. [1] elevated the part-whole relation to the status of a first-class citizen and further studied the semantics and functionality of a variety of part relations. In this paper, we emphasise update propagation, therefore, we consider dependencies between arbitrary pairs of objects, not limited to the part-whole relation. In this light, we classify types of dependencies between objects as follows.
Definition 4 Class $C_i$ is said to be exclusively dependent on class $C_j$ via reference attribute $A_{\ref} : C_i \rightarrow C_j$ iff
\[ \forall o_k^i \in C_i, \exists o_j^i \in C_j (\text{OID}(o_j^i) \in o_k^i.A_{\ref}^i) \]
holds for every state of the database. We also call the reference attribute $A_{\ref}$ to be exclusively dependent and its inverse reference attribute $A_{\ref}^i : C_j \rightarrow C_i$ to be exclusively determinant.

Definition 5 Class $C_i$ is said to be sharedly dependent on class $C_j$ via reference attribute $A_{\ref} : C_i \rightarrow C_j$ iff
\begin{enumerate}
\item[(a).] $\forall o_k^i \in C_i, \exists o_j^i \in C_j (\text{OID}(o_j^i) \in o_k^i.A_{\ref}^i)$ holds for every state of the database,
\item[(b).] $o_k^i$ is not allowed to be deleted if $\exists o_j^i \in C_j (\text{OID}(o_j^i) \in o_k^i.A_{\ref}^i)$, where $A_{\ref}^i : C_j \rightarrow C_i$ is the inverse reference attribute of $A_{\ref}$.
\end{enumerate}
We also call the reference attribute $A_{\ref}$ to be sharedly dependent and $A_{\ref}^i$ to be sharedly determinant.

Definition 6 Class $C_i$ is said to be multiply dependent on a set of classes $C_{j_1}, \ldots, C_{j_m}$ via reference attributes $A_{\ref_{j_1}}, \ldots, A_{\ref_{j_m}}$ iff
\[ \forall o_k^i \in C_i, \exists o_j^{i_1} \in C_{j_1}, \ldots, \exists o_j^{i_m} \in C_{j_m} (\text{OID}(o_j^{i_1}) \in o_k^i.A_{\ref_{j_1}} \land \cdots \land \text{OID}(o_j^{i_m}) \in o_k^i.A_{\ref_{j_m}}) \]
holds for every state of the database. We also call the reference attributes $A_{\ref_{j_1}}, \ldots, A_{\ref_{j_m}}$ to be multiply dependent and their inverse reference attributes $A_{\ref_{j_1}}^i, \ldots, A_{\ref_{j_m}}^i$ to be multiply determinant.

Definition 7 Class $C_i$ and class $C_j$ is said independent via reference attribute $A_{\ref} : C_i \rightarrow C_j$ iff there is no restriction on their objects. We also call the reference attribute $A_{\ref}$ and its inverse reference attribute $A_{\ref}^i : C_j \rightarrow C_i$ to be independent.
A reference attribute is said to be dependent if it is exclusively dependent, sharedly dependent or multiply dependent. Likewise, a reference attribute is said to be determinant if it is exclusively determinant, sharedly determinant or multiply determinant. Usually, real update propagation happens only to the determinant reference attributes. In the next, we study the propagation policies for each type of these object dependencies.

3.3 Predefined Propagation Policies for Different Dependencies

Figure 5 shows propagation policies for different types of object dependencies. As we know, a delete operation can be divided into two parts: one is the operation to the object itself; another is the impact operation, i.e., propagation on all its references to other objects. Whenever an object is deleted, its references to other objects must be removed as well, therefore a remove reference operation sometimes is a sub-operation of delete operation, i.e., the policies for remove reference operation is implied by delete operation and thus is not shown in the figure.

First, let us consider propagation policies for delete operation. If object $o_k^j$ in $C_i$ is exclusively dependent on object $o_j^i$ in $C_j$, deleting $o_j^i$ will dependently propagate (DP) to $o_k^j$, while deleting $o_k^j$ only results in deleting inverse reference. If object $o_k^j$ in $C_i$ is sharedly dependent on a set of objects $O_j^i \subset C_j$, $o_k^j$ is reserved for all objects in $O_j^i$ until they are all deleted. The deletion of every object in $O_j^i$ will independently propagate (IP) to $o_k^j$ but only the last deletion will cause $o_k^j$ to be deleted. If object $o_k^j$ in $C_i$ is multiply dependent to multiple set of objects $O_{i1}^j \subset C_{j1}, \ldots, O_{im}^j \subset C_{jm}$ via different reference attributes. deleting object in any of these sets will restrictedly propagate to $o_k^j$. Note, there is a difference between sharedly dependent and multiply dependent reference attributes, $o_k^j$ is not allowed to delete individually in the former but is allowed in the latter. For independent relationship, deleting an object only cause the removal of its inverse reference.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Delete</th>
<th>Insert</th>
<th>Add Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ci Exclusively</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependent on Cj</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|                                 | CJ  
|                                 |          
|                                 | DP  
|                                 | IV  
|                                 | Cl  | CJ  
|                                 |          
|                                 | NO  
|                                 | BD  
|                                 | Cl  | CJ  
|                                 |          
|                                 | NA  
|                                 | NA  
|                                 | Cl  |
| **Ci Sharedly**                 |                 |                 |                |
| Dependent on Cj                  |                 |                 |                |
|                                 | CJ  
|                                 |          
|                                 | IP  
|                                 | RV  
|                                 | Cl  | CJ  
|                                 |          
|                                 | NO  
|                                 | BD  
|                                 | Cl  | CJ  
|                                 |          
|                                 | IV  
|                                 | IV  
|                                 | Cl  |
| **Ci Multiply**                 |                 |                 |                |
| Dependent on Cj                  |                 |                 |                |
|                                 | CJk  
|                                 |          
|                                 | RP  
|                                 | RP  
|                                 | IV  
|                                 | Cl  | CJ  
|                                 |          
|                                 | NO  
|                                 | BD  
|                                 | Cl  | CJ  
|                                 |          
|                                 | IV  
|                                 | IV  
|                                 | Cl  |
| **Ci Independent**              |                 |                 |                |
| from Cj                         |                 |                 |                |
|                                 | CJ  
|                                 |          
|                                 | IV  
|                                 | Cl  | CJ  
|                                 |          
|                                 | NO  
|                                 | Cl  | CJ  
|                                 |          
|                                 | IV  
|                                 | IV  
|                                 | Cl  |

Figure 5: Predefined propagation policies for different dependencies

The propagation policies for *insert* and *add reference* operations are simple. In fact, there is no propagation problem. When inserting an object, there is no extra operation but restriction if the object being inserted is dependent by definition on some other objects. When adding a reference, the only extra operation is to set the inverse reference.

By categorising reference attributes according to different types of object dependencies, it is easy to see that policies predefined for different operations on relationships (i.e., a pair of reference attributes) with the same type of object dependency are compatible. However, it is permitted in a class definition to have reference attributes with all types of object dependencies. The following rules represent what kind of combination of these dependencies a single instance of a class can have.
**Rule 1** If an object $o$ is exclusively dependent on another object, it will not be allowed to have any dependencies (exclusive, shared, multiple) on other objects.

**Rule 2** If an object $o$ is sharedly dependent on some objects, it will not be allowed to have any exclusive and multiple dependencies on other objects. However, new shared dependencies on other objects will be allowed.

**Rule 3** If an object $o$ is multiply dependent on some objects, it will not be allowed to have any exclusive and shared dependencies on other objects. However, new multiple dependencies on other objects will be allowed.

**Rule 4** An object can have any number of independent reference to it, no matter whether it is dependent on some objects.

### 3.4 Conflict Policy Prevention

As we may notice now, propagation is due to necessary enforcement of object dependencies.

In the following, we first define a *dependency graph* of a given reference graph, then show that if a dependency graph is acyclic, the previous propagation policy conflict problems will no longer exist.

**Definition 8** Given an object-oriented database schema $S = \{C_1, \cdots, C_n\}$, a dependency graph $DG$ of the schema $S$ is a subgraph of its reference graph $RG$, it can be constructed as follows:

(a) $E_{DG} \subseteq E_{RG}$ is derived by keeping those arcs in $E_{RG}$, each of which is labelled with $\langle A_{ref}, A_{ref}' \rangle$, where $A_{ref}$ is a dependent reference attribute name,

(b) $V_{DG} \subseteq V_{RG}$ is therefore reduced to keep only those which are connected by arcs in $E_{DG}$.
Figure 6 shows the dependency graph of electronic documents database.

Figure 6: Dependency graph for electronic document example

**Proposition 1** If the dependency graph $DG$ of an object-oriented database schema $S$ is acyclic, then the schema $S$ is free of RV-cycle policy conflict.

Proof: From figure 5 we can see, RV policies are only defined on sharedly dependent reference attributes in the propagation graph $PG$ of $S$. Since the dependency graph $DG$ is acyclic and $DG$ is a subgraph of the reference graph $RG$ of $S$, there is no cycle of sharedly dependent reference attributes in $RG$, therefore, there is no RV cycle in $PG$.

**Proposition 2** If the dependency graph $DG$ of an object-oriented database schema $S$ is acyclic, then the schema $S$ is free of self-contradictory policy conflict.

Proof: Suppose a self-contradictory policy conflict exists in the propagation graph $PG$ of $S$. Then from figure 5, we can replace all IP policies with inverse RV policies in $PG$. As a result, a dependency cycle of sharedly dependent reference attributes formed in $PG$ thus $RG$ and $DG$ of $S$. This contradicts to the acyclic dependency graph.
The dependency graph shown in figure 6 is acyclic, therefore, the electronic document processing database schema $S_{edoc}$ is free of both RV-cycle policy conflict and self-contradictory policy conflict.

4 Implementation

In supporting update propagation, we extend the class definition of an object-oriented database schema to include the type of object dependency for each reference attribute appeared in the schema. This is similar to specify referential integrity constraints and rules when creating a table in relational database systems. For example, the class definitions for Section class and Paragraph class are extended as follows.

```java
class Section

type {
    content: Set<Paragraph> inverse Paragraph: shared_by

    shared_determinant:

    shared_by: Set<Document> inverse Document: content

    shared_dependent
}

class Paragraph

type {
    annotated_for: Document inverse Document: annotation

    exclusive_dependent;

    shared_by: Set<Section> inverse Section: content

    shared_dependent
```
Based on the definitions of all classes of an object-oriented database schema, it is easy to construct a dependency graph for the database and to check whether the graph is acyclic.

We can implement propagation policies in an abstract class. With the help of class hierarchy and inheritance facility of an OODBMS, all concrete classes can inherit the propagation methods defined at the high level abstract class. Specialisation may be applied for the application-specific update propagation based on the propagation methods of generic update operation. This can relieve lots of hard-coding work from a designer, while avoiding propagation conflicts and incompatibilities which may occur and hard to check if implemented individually by designers.

In supporting different types of reference attributes, the rules 1–4 should be enforced by the system. This can be accomplished by defining a new insert operation also in the abstract class which refines the insert operation implemented by the OODBMS. The refinement simply checking the preconditions defined by rules 1–4.

It is possible that an update operation to an object may propagate to another object several times from different paths. Rules 1–4 put a restriction on propagation to any object. Since an object can have at most one type of dependency on other objects at a time, the types of multiple propagations to the same object, if occur, should be the same. This guarantees that there is no propagation conflicts along different propagation paths. In addition, we maintain the mutual references for all relationships between objects. When an update propagation happens to an object, say delete, all references of the object to other objects will be removed. Therefore, recursive processing of update propagation is possible, not necessarily restricted to transaction-based processing as adopted in [17].
However, in supporting the DP policy, a two-phase recursive processing strategy is needed. At first phase, check whether all objects which are exclusively dependent on the object which are deletable. At the second phase, really apply the propagation.

5 Conclusion

Update propagation is an important topic in database systems. In most database systems, it is database application designers’ responsibilities to hard-code the propagation semantics in transactions or methods of OODBMSs. In this paper, we discussed update propagation support in object-oriented database systems and tried to relieve the burden from designers as much as possible. Since automatic support of all application-specific update propagation is hard to achieve, we focused on update propagation of generic update operations, especially, on delete operation. As we know, propagation of update is caused by relationships between objects, therefore we took a declarative approach, specifying propagation policies on the reference attributes, rather than hiding propagation semantics in update method of each class. We studied the propagation policies for generic update propagations and discovered that potential conflicts among propagation policy declarations may occur if a designer can arbitrarily declare propagation policies on reference attributes of a class. As such, we promoted the problem to a higher level so that designers only need to specify the dependency property for reference attributes. Propagation policies are predefined for each type of dependency. By introducing some restrictions on class definition, the potential conflicts will disappear. In the paper, we also addressed the implementation issues for update propagation support in object-oriented database systems.

Currently, we are supporting update propagation for a workflow repository [14] using Objectstore OODBMS based on the ideas introduced in this paper.
Acknowledgements

The authors would like to thank Dr Arthur Ter Hofstede for many useful comments that he has provided.

References


