An Efficient Version Model of Software Diagrams

Jungkyu Rho and Chisu Wu
Department of Computer Science
Seoul National University, Seoul, 151-742, Korea
{jkrho, wuchisu}@selab.snu.ac.kr

Abstract

Various types of diagrams are used to represent the design of software systems. During the design phase, versions of a diagram may be created like other design documents and source code, and it is necessary to manage them efficiently. However, traditional configuration management systems and some object-oriented database management systems that provide object versioning are not suitable for managing versions of a diagram. In this paper, we propose an efficient version model of software diagrams. This model reflects the common characteristics and structure of software diagrams, and revisions of a diagram are managed by operation delta and object visibility. A merge model for versions of a diagram is also presented at the end of this paper.

1 Introduction

During the analysis and design phase of software development, various types of diagrams are used. For example, use-case diagrams and class diagrams of UML are commonly used. Generally, software diagrams consist of nodes, edges, and attributes. Nodes are conceptual software design entities and edges represent relationships between nodes. Attributes are used to represent the properties of nodes and edges. For example, a class diagram may have class and package nodes and association and generalization edges. Nodes and edges may have attributes, such as name, role name, and cardinality.

During the software design phase, versions of a diagram are created like other software engineering documents. For efficient version management of diagrams, the following requirements should be satisfied. First, fine-grained data model where nodes and edges are individual objects should be used. With fine-grained data, tool construction becomes easier, and detailed relationships between different documents and different parts of the same document are available [5]. Second, efficient diagram delta management should be supported. During the revision history of a diagram, nodes and edges are created and deleted frequently and deleting a node causes deletion of the edges connected to the node. Therefore, those characteristics should be considered in delta management. Third, diagrams contain graphic information, such as position and size of a node, to represent software design information. Therefore, it is desirable to distinguish the changes of design information from those of graphic information. Finally, it is necessary to provide a merging tool for software diagrams because structure-oriented merging tools [6] [10] do not reflect the characteristics of diagrams.

Traditional configuration management systems and some of the OODBMSs support version management of composite objects. If all versions of an object are referred by the same relationship, then the relationship is *generic*. On the other hands, if each version is bound to a specific relationship, then the relationship is *bound*. A bound configuration is a composite object that has bound relationships to its components. When an object is updated in a bound configuration, a new version of the object and a new reference to the version are created. For example, a typical diagram is presented in Figure 1(a) and its fine-grained data model representation is illustrated in Figure 1(b). Diagram, node, and edge objects have bound relationships to each other. Suppose an attribute of n_1 was modified. Then n_1', a new version of n_1, should be created and D', a new version of D, should have n_1' instead of n_1. In addition, unchanged edge e_1 should be also modified to have a reference to n_1' in D'. Therefore e_1' should be created and n_2', e_2', and n_3' should be created subsequently. As a result, a change in one object was propagated to the edge-connected subdiagram (n_1, n_2, n_3, and their edges). Therefore bound configurations are not suitable for fine-grained diagram version management.

Existing software configuration management systems and software engineering environments are inadequate for fine-grained diagram version management. RevisionTree server [6] supports revision control for fine-grained hierarchically structured documents, such as programs that consist of classes and functions. However, it supports only bound configurations. CoMa [11] manages configuration of engineering design documents, but CoMa also supports only bound configurations. PCTE [3] provides version management of composite objects. However, objects in PCTE are supposed to be coarse-grained objects and it supports only bound configurations.

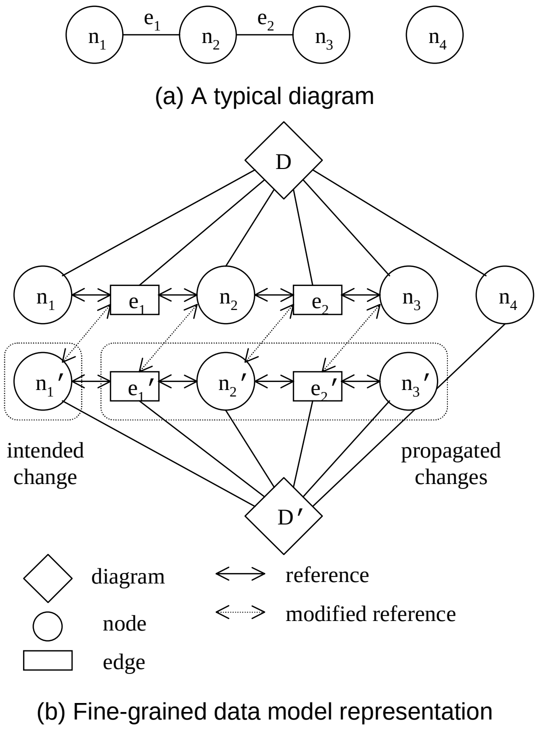


Figure 1 Problem of bound configuration

Some OODBMSs support generic configuration. However, those are not suitable for configuration management of fine-grained objects where components are created and deleted frequently. In ORION model, generic objects have high storage overhead. Therefore coarse-grained objects should be used in order to limit storage overhead of generic object [1]. Database version of O_2 [2] relives storage overhead of generic objects, but it is not suitable for managing frequent creation and deletion of component objects. GOODSTEP [4] extended O_2 with the facilities for software development environments. However, it does not provide improved storage management for versioning of fine-grained data.

In this paper, we propose an efficient diagram version management model that reflects the facts that software design information is represented by means of graphic information and the structure of software diagrams consists of nodes and edges. It provides efficient storage management and retrieval of revisions of a diagram through *operation delta* and *object visibility*. Furthermore, our model has an advantage that it is not necessary to compare a new revision with the previous one to extract delta because editing operations are used as delta. We provide a merge model for versions of a diagram and implemented a prototype of Diagram Version Management System (DIVERS).

This paper is organized as follows. Section 2 introduces our fine-grained diagram model and version model. Section 3 describes the storage model of DIVERS in detail. Section 4 briefly explains our diagram merge model and Section 5 presents a conclusion.

2 Version Model

2.1 Diagram Model

Diagrams are important ways to represent the software design. Diagrams contain not only design information related to the software design, but also auxiliary information, like diagram layouts. We classified the information contained in diagrams to *design information, quasi-design information* and *graphic information*. Design information is directly related to the software design. For example, class definitions and associations between classes are the design information of class diagrams. Quasi-design information does not represent the software design itself, but it describes the design. For example, description of class definition is quasi-design information. Graphic information is a means to represent design information in a diagram. For example, graphic information includes shapes, positions, and sizes of nodes and edges.

Our diagram model consists of *nodes*, *graphic nodes*, *edges*, *attributes*, and *diagrams*. A diagram has nodes as its children, a node has graphic nodes and set-valued attributes as its children, and a graphic node has edges as its children. A node may have atomic attributes and set-valued attributes. A graphic node is the graphic representation of a node, and has graphic attributes, such as position, size, and color. Two or more graphic nodes representing one node may appear in a diagram to enhance readability. Existence of a graphic node depends on the existence of its node. An edge may have atomic attributes. An edge connects its source and destination graphic nodes. Existence of an edge depends on the existence of its parent graphic nodes. Attributes are classified into set-valued attributes and atomic attributes. Existence of a set-valued attribute depends on the existence of its parent node. A diagram is a composite object that consists of nodes, graphic nodes, edges, and set-valued objects. A node can expand to another diagram. Our diagram model is defined as follows.

$$\begin{aligned}
&\textit{diagram} = N_0\\
&\textit{node} = (n, \textit{node\_type}, \textit{node\_name}, A_n, A_{0n}, QA_n, G_{0n})\\
&\textit{graphic node} = (g, GA_g, E_{0g})\\
&\textit{edge} = (e, \textit{edge\_type}, A_e, QA_e, GA_e)
\end{aligned}$$

set-valued attribute = (*a*, *attr_type*, *value*)

where N_0 is a set of nodes, and *n* is a node identifier within a diagram. Node has its type, *node_type*, and name, *node_name*. A_n, A_{0n}, QA_n, and G_{0n} are sets of atomic attributes, set-valued attributes, quasi-design attributes, and graphic nodes, respectively. $g = (n, GN)$ is a graphic node identifier and *GN* distinguishes a graphic node from those belong to the same node. GA_g is a set of graphic attributes, and E_{0g} is a set of edges. $e = (g_1, g_2, EN)$ is an edge identifier and *EN* distinguishes an edge from those belong to the same parent graphic nodes. Edge has its type, *edge_type*, and A_e, QA_e, and GA_e are sets of atomic attributes, quasi-design attributes, and graphic attributes, respectively. $a = (n, AN)$ is a set-valued attribute identifier and *AN* distinguishes a set-valued attribute from those belong to the same node. Set-valued attribute has its type, *attr_type*, and value, *value*.

To distinguish design information from other information, the design view of a diagram is defined as follows.

design node = $(d, A_n, A_{0n},)$
design edge = (d_1, d_2, A_e)
set-valued attribute = (*d* , *attr_type*, *value*)

where *d* = (*node_name*, *node_type*) is a design key of node. We say that two diagrams are *design equivalent* if the design views of the two diagrams are equivalent.

We imposed two integrity rules on the diagram data model. First, a child object should be deleted when its parent object is deleted. For example, an edge should be deleted if the source or destination graphic node of the edge is deleted. Second, node names are unique within the nodes that belong to the same diagram and have the same type. However, it is possible to change node name unless the uniqueness is violated in the current state of a diagram.

2.2 Version Model

When a user edits a diagram, a number of operations are applied to change the state of the diagram. Those are divided into structure operations and attribute operations listed in Table 1. Structure operations include creation and deletion of nodes, graphic nodes, and edges. Attribute operations are divided into atomic and set-valued attribute operations. Atomic attribute operations update the attributes contained in nodes, graphic nodes, and edges. Set-valued attribute operations include creation, deletion, and update.

As described above, diagrams contain design and graphic information, and optionally have quasi-design information. Often, a new version of a diagram may differ from the previous version only in graphic or quasi-design information. For example, users may update only graphic attributes to create a new version. In this case, the new version is design equivalent to the previous version. Later, when they retrieve the version, it is desirable to inform them that the version is design equivalent to the predecessor version.

Table 1 Edit operations

node	create(*n*, *node_type*, *node_name*), delete(*n*)
graphic node	create(*g*, initial values of graphic attributes), delete(*g*)
edge	create(*e*, *edge_type*, initial values of graphic attributes), delete(*e*)
atomic attribute	update(*n*, *attr_name*, *attr_value*), update(*g*, *attr_name*, *attr_value*), update(*e*, *attr_name*, *attr_value*)
set-valued attribute	create(*a*, *attr_type*, *attr_value*), delete(*a*), update(*a*, *attr_value*)

In our work, diagram delta Δ = (ΔD, ΔQ, ΔG) is provided as documentation, where ΔD, ΔQ, and ΔG denote design delta, quasi-design delta, and graphic delta, respectively. Design delta consists of the operations that change the design view of a diagram, e.g. creation and deletion of nodes and edges, change of node name, and change of design attributes. Quasi-design delta contains the operations that change quasi-design attributes, such as descriptions and comments on design information. Graphic delta contains the operations that change graphic attributes, such as creation of a new graphic node, change of graphic node size, and change of edge position. Some operations change both design and graphic information. For example, creation of an edge changes both design and graphic attributes.

Relationship between an expanding node and its expanded diagram is either bound or generic. When a new version of a diagram is created, the change is propagated to its expanding node only if bound relationship exists and the design delta of the diagram is not nil. In this case, a new version of the higher level diagram should be created.

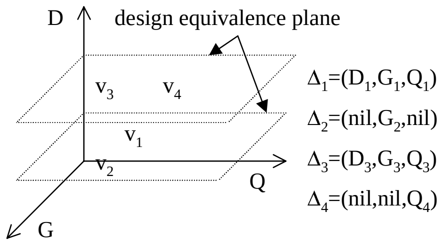


Figure 2 Design equivalence plane

Figure 2 shows an example of diagram delta and design equivalence planes. If the design delta of a new version is nil, then the version and the predecessor version belong to the same design equivalence plane. In Figure 2, v_1 and v_3 are design equivalent to v_2 and v_4, respectively. Design Version [8] introduced equivalent design version of composite objects. However, it does not explicitly provide delta. In EXTRA-V [9] model, an attribute of a class can be declared as a dimension of the version space. But it is not possible to declare a group of attributes from distinct classes as one dimension (e.g. graphic). The advantage of our approach is that users are able to know the changes occurred in derivation of a version. Moreover, design equivalent versions can be easily identified from the delta.

3 Version Storage Management

In this section, we present our version storage management model. Generally, version storage maintains either the copies of all versions or one version with delta. DIVERS is the latter. It maintains the latest revision and the *operation delta* applied to derive each revision. Updating an object does not cause creation of a new version of the object. In contrast, objects are shared between revisions. As a result, all references and composition relationships between objects are considered as generic relationships. Each object maintains its latest state and the operations applied to the object are recorded as backward delta. To retrieve an old revision, operation delta is applied to the latest revision in reverse order. Objects are not shared between branches whereas those are shared between revisions. When a new branch is created, all objects in the branch are copied. Each branch maintains the state of its latest revision and operation delta.

3.1 Operation Lists

Each revision has the *operation list* that was applied to the previous revision to produce that revision, and each branch has a set of operation lists for all of its revisions. Operation lists must be *valid* and *minimal* in order to be used as operation delta. The valid operation list satisfies the following rules.

1. Object identifiers are unique throughout the revision history. Even if an object was removed, reusing its identifier is not allowed.
2. Neither operations on the objects removed in the previous revision nor operations on their descendants are allowed.
3. If object deletion exists in the operation list, neither operations on the object nor operations on its descendants can appear after the deletion operation.

The minimal operation list can be constructed from the valid operation list by applying the following rules.

1. If object deletion exists, the preceding operations on that object or its descendant objects can be removed.
2. Repeated update operations on an attribute can be reduced to the last operation.

In other words, the valid and minimal operation list does not contain operations on the objects that were deleted either explicitly or implicitly.

Create operations are recorded in operation lists without attribute value specifications, e.g. c(*n*), and update operations are recorded in inverse form, For example, when the value of attribute *a* of node *n* is v_1 and update(*n*, *a*, v_2) is applied, the value of *a* will be set to v_2 and u(*n*, *a*, v_1) will be recorded in the operation list.

Operation lists are separately stored in order to reduce the searching time for operation delta to retrieve previous versions. The operation list of the *k*-th revision consists of OP_k and sets of OP_{kn}, OP_{kg}, and OP_{ke}. OP_k contains creation and deletion of nodes and edges. OP_{kn} contains creation and deletion of the graphic nodes and the set-valued attributes of node *n*. It also contains update operations on the attributes of node *n*. OP_{kg} contains update operations on the attributes of graphic node *g*. OP_{ke} contains update operations on the attributes of edge *e*.

3.2 Object Visibility

Component objects are included or excluded in a revision according to *visibility*. After an object was created, it is visible from its parent object. Later, after the object is deleted, it will be invisible from its parent object. An object is included in a revision only if it is visible from the root of composition hierarchy, the diagram object. Therefore a revision of a diagram is composed of the nodes that are visible from the diagram object, the graphic nodes and the set-valued attributes that are visible from the visible nodes, and the edges that are visible from the visible graphic nodes. Object visibility differs from delta visibility of Change-Oriented Versioning [7], where a desired version is constructed by applying visible delta to the baseline.

To define object visibility, the following terms must be defined first.

- *N* is a set of the nodes that were created throughout the revision history of a diagram, regardless of visibility.
- N_0 is a set of the nodes that are visible from the diagram object of the latest revision.
- G_0 is a set of the graphic nodes that are visible from the visible nodes, which are the elements of N_0. G_{0n} is a set of the graphic nodes that are visible from node *n*.
- E_0 is a set of the edges that are visible from the visible graphic nodes, which are the elements of G_0. E_{0g} is a set of the edges that are visible from graphic node *g*.
- A_0 is a set of the set-valued attributes that are visible from the visible nodes, which are the elements of N_0. A_{0n} is a set of the set-valued attributes that are visible from node

n.

To delete objects, the delete algorithms in Figure 3 are applied. The algorithms make objects invisible from their parents. To retrieve the i-th revision, the following visibility definitions are used.

```
delete_node(n: node)
begin
   $N_0 := N_0 - \{n\}$;
   for $\forall g \in G_{0n}$ delete_edge_of(g);
end

delete_graphic_node(g: graphic node)
begin
   $n := g.n$;
   $G_{0n} := G_{0n} - \{g\}$;
   delete_edge_of(g);
end

delete_attribute(a: set-valued attribute)
begin
   $n := a.n$;
   $A_{0n} := A_{0n} - \{a\}$;
end

delete_edge_of(g: graphic node)
begin
   for $\forall e \in E_{0g}$
   begin
      if $g = e.g_1$ then $h := e.g_2$
      else $h := e.g_1$;
      $E_{0h} := E_{0h} - \{e\}$;
   end
end

delete_edge(e: edge)
begin
   $g := e.g_1$;
   $E_{0g} := E_{0g} - \{e\}$;
   $h := e.g_2$;
   $E_{0h} := E_{0h} - \{e\}$;
end
```

Figure 3 Delete algorithms

Node visibility

Suppose Ci is a set of the nodes created and Di is a set of the nodes deleted between the i+1th revision and the latest revision. Then Ci and Di are obtained by

$$C_i = \{n \mid c(n) \in \textstyle\bigcup_{k=i+1}^{l} OP_k\}$$

$$D_i = \{n \mid d(n) \in \textstyle\bigcup_{k=i+1}^{l} OP_k\}$$

where l is the latest revision number, $c(n)$ is creation of node n, $d(n)$ is deletion of node n, and OP_k is the operation list of the k-th revision. Node visibility in the i-th revision, $V_i(n)$ and a set of the nodes included in the i-th revision, N_i are

$$V_i(n) = \begin{cases} true & if \quad n \in (N_0 \bigcup D_i) - C_i \\ false & otherwise \end{cases}$$

$$N_i = \{n \mid V_i(n)\}$$

respectively, where $1 \le i < l$.

Graphic node visibility

Suppose CG_{in} is a set of the graphic nodes created and DG_{in} is a set of the graphic nodes deleted between the i+1th revision and the latest revision among the graphic nodes contained in node n. Then CG_{in} and DG_{in} are obtained by

$$CG_{in} = \{g \mid c(g) \in \textstyle\bigcup_{k=i+1}^{l} OP_{kn}\}$$

$$DG_{in} = \{g \mid d(g) \in \textstyle\bigcup_{k=i+1}^{l} OP_{kn}\}$$

where $c(g)$ is creation of graphic node g, $d(g)$ is deletion of graphic node g, and OP_{kn} is the operation list of the k-th revision contained in node n. In the i-th revision, visibility of the graphic nodes contained in node n is

$$V_{in}(g) = \begin{cases} true & if \quad g \in (G_{0n} \bigcup DG_{in}) - CG_{in} \\ false & otherwise \end{cases}$$

and a set of the visible graphic nodes contained in node n is

$$G_{in} = \{g \mid V_{in}(g)\}$$

And a set of the graphic nodes included in the i-th revision is

$$G_i = \textstyle\bigcup_{n \in N_i} G_{in}$$

Edge visibility

Suppose E'_i is a set of the edges that are visible from the graphic nodes included in the i-th revision, and CE_i is a set of the edges created and DE_i is a set of the edges deleted between the i+1th revision and the latest revision. Then E'_i, CE_i, and DE_i are obtained by

$$E'_i = \textstyle\bigcup_{p \in G_i} E_{0g}$$

$$CE_i = \{e \mid c(e) \in \textstyle\bigcup_{k=i+1}^{l} OP_k\}$$

$$DE_i = \{e \mid d(e) \in \textstyle\bigcup_{k=i+1}^{l} OP_k\}$$

where $c(e)$ and $d(e)$ are creation and deletion of edge e, respectively. Edge visibility in the the i-th revision is

$$V_i(e) = \begin{cases} true & if \quad e \in (E'_i \bigcup DE_i) - CE_i \\ false & otherwise \end{cases}$$

and a set of the edges included in the i-th revision is

$$E_i = \{e \mid V_i(e)\}$$

Set-valued attribute visibility

Suppose CA_{in} is a set of the attributes created and DA_{in} is a set of the attributes deleted between the i+1th revision and the latest revision among the set-valued attributes contained in node *n*. Then CA_{in} and DA_{in} are obtained by

$$CA_{in} = \{a \mid c(a) \in \bigcup_{k=i+1}^{l} OP_{kn}\}$$

$$DA_{in} = \{a \mid d(a) \in \bigcup_{k=i+1}^{l} OP_{kn}\}$$

where *c*(*a*) and *d*(*a*) are creation and deletion of set-valued attribute *a*, respectively. In the *i*-th revision, visibility of the set-valued attributes contained in node *n* is

$$V_{in}(a) = \begin{cases} true & if \quad a \in (A_{0n} \bigcup DA_{in}) - CA_{in} \\ false & otherwise \end{cases}$$

and a set of the set-valued attributes contained in node *n* is

$$A_{in} = \{a \mid V_{in}(a)\}$$

And a set of the set-valued attributes included in the *i*-th revision is

$$A_i = \bigcup_{n \in N_i} A_{in}$$

In addition, attribute value in the *i*-th revision is decided by the earliest (inverse) update of the attribute between the *i*+1th revision and the latest revision.

Figure 4(a) shows three revisions of a typical diagram. The operations between two revisions are applied to derive the successor revision. The stages of the internal representation after each revision are illustrated in Figure 4(b). In derivation of revision 2, deletion of n_1 makes n_1 invisible and also makes e_1 invisible from g_2, and deletion of e_2 makes e_2 invisible from its parents, g_2 and g_3. In derivation of revision 3, deletion of e_3 makes e_3 invisible from its parents, and deletion of a_1 makes a_1 invisible from n_3. Suppose a user wants to retrieve revision 1 after creation of revision 3. Then the internal representation of revision 3 is used to obtain revision 1. The result is as follows.

$N_0 = \{ n_2, n_3, n_4, n_5 \}$, $C_1 = \{ n_4, n_5 \}$, $D_1 = \{ n_1 \}$, $N_1 = \{ n_1, n_2, n_3 \}$,
$G_0n_1 = \{ g_1 \}$, $G_0n_2 = \{ g_2 \}$, $G_0n_3 = \{ g_3 \}$,
$CG_1n = \varnothing$ and $DG_1n = \varnothing$ for $\forall n \in N_1$,
$G_1n_1 = \{ g_1 \}$, $G_1n_2 = \{ g_2 \}$, $G_1n_3 = \{ g_3 \}$, $G_1 = \{ g_1, g_2, g_3 \}$,
$E_0g_1 = \{ e_1 \}$, $E_0g_2 = \varnothing$, $E_0g_3 = \varnothing$, $E'_1 = \{ e_1 \}$,
$CE_1 = \{ e_3, e_4 \}$, $DE_1 = \{ e_2, e_3 \}$, $E_1 = \{ e_1, e_2 \}$,
$A_0n = \varnothing$ for $\forall n \in N_1$, $CA_1n_1 = \varnothing$, $DA_1n_1 = \varnothing$,
$CA_1n_2 = \varnothing$, $DA_1n_2 = \varnothing$, $CA_1n_3 = \varnothing$, $DA_1n_3 = \{ a_1 \}$,
$A_1n_1 = \varnothing$, $A_1n_2 = \varnothing$, $A_1n_3 = \{ a_1 \}$, and $A_1 = \{ a_1 \}$.

3.3 Analysis

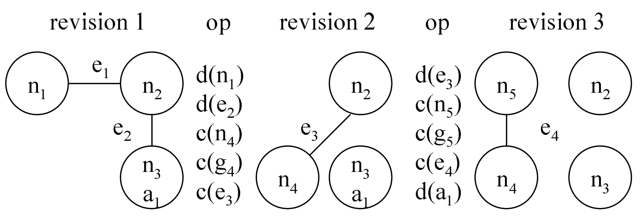
In this section, we demonstrate the correctness of the visibility definitions and compare our model with database version of O_2 [2].

Correctness of the visibility definitions

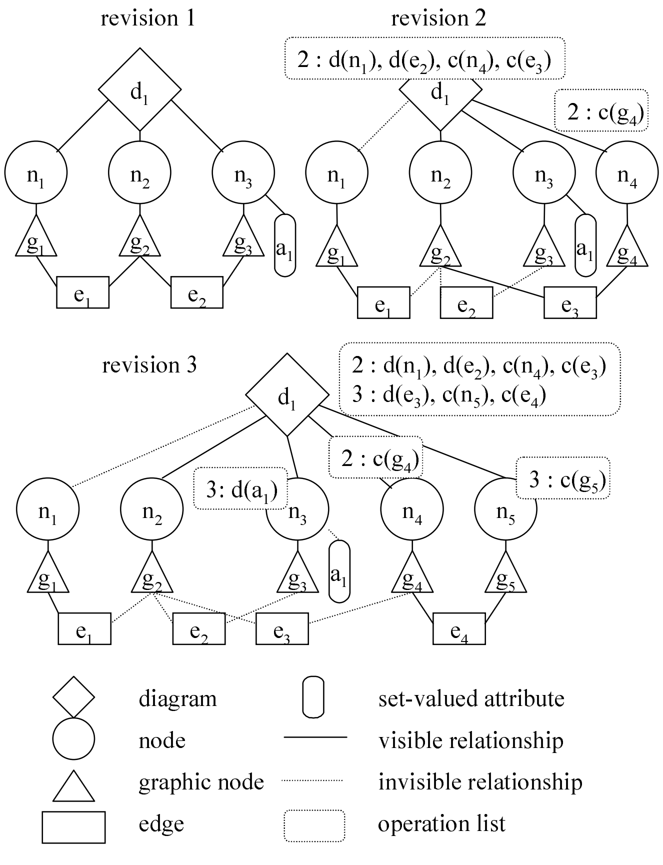
We examine the correctness of node and edge visibility in this section. The visibility of graphic node and set-valued attribute is similar to node visibility. We assume that all operation lists are valid.

First, all nodes in *N* were created once and might be deleted or not, regardless of their visibility in the latest revision. Therefore all nodes included in *N* are divided into the five categories listed in Table 2.

n_1 : created and deleted before the *i*-th revision.
n_2 : created before the *i*-th revision and deleted after the *i*-th revision.
n_3 : created before the *i*-th revision and not deleted.
n_4 : created and deleted after the *i*-th revision.
n_5 : created after the *i*-th revision and not deleted.



(a) Revisions of a diagram



(b) Internal representation

Figure 4 An example of revision history

By the delete algorithms in Figure 3, n_1, n_2, and n_4 are not included in the latest revision. According to the above definitions, N_0 contains n_3 and n_5, D_i contains n_2 and n_4, C_i contains n_4 and n_5. Therefore N_i contains n_2 and n_3, and the definition of N_i produces the nodes included in the i-th revision.

Second, all child edges of the graphic nodes included in the i-th revision are divided into the 7 categories listed in Table 3. Since the edge categories from e_1 to e_5 are similar to the node categories, it is sufficient to define e_6 and e_7.

e_6 : created before the i-th revision and at least one of its ancestors was deleted after the i-th revision.

e_7 : created after the i-th revision and at least one of its ancestors was deleted after the i-th revision.

By the delete algorithms in Figure 3, e_1, e_2, e_4, e_6, e_7, and the parent graphic nodes of e_6 and e_7 are not included in the latest revision. Note that e_6 and e_7 are visible from one of their parent graphic nodes. According to the above definitions, E'_i contains e_3, e_5, e_6 and e_7, CE_i contains e_4, e_5, and e_7, DE_i contains e_2 and e_4. Therefore E_i contains and e_2, e_3, and e_6, and the definition of E_i produces the edges included in the i-th revision.

Table 2 Node categories

	n_1	n_2	n_3	n_4	n_5
$OP_1 \sim OP_i$	$c(n_1)$ $d(n_1)$	$c(n_2)$	$c(n_3)$		
i-th revision	×	o	o	×	×
$OP_{i+1} \sim OP_l$		$d(n_2)$		$c(n_4)$ $d(n_4)$	$c(n_5)$
latest revision	×	×	o	×	o

o : included × : not included

Table 3 Edge categories

	e_1	e_2	e_3	e_4	e_5	e_6	e_7
$OP_1 \sim OP_i$	$c(e_1)$ $d(e_1)$	$c(e_2)$	$c(e_3)$			$c(e_6)$	
i-th revision	×	o	o	×	×	o	×
$OP_{i+1} \sim OP_l$		$d(e_2)$		$c(e_4)$ $d(e_4)$	$c(e_5)$	$d(p_6)$	$c(e_7)$ $d(p_7)$
latest revision	×	×	o	×	o	•	•

o : included × : not included

• : not included, but visible from one of its parent graphic nodes

p_i : one of the parent graphic nodes (or the ancestor nodes) of e_i

Comparison to O_2 version management

We compare our model, DIVERS with database version of O_2 [2] because O_2 is considered as the best candidate for versioning of the fine-grained diagram model among existing OODBMSs. Comparisons are made in three aspects, operation processing, the storage requirements, and version retrieval.

Creation of a new version of a diagram includes update, create, and delete operations. First, when an attribute of an object is updated, DIVERS adds the update operation to the operation list. In contrast, O_2 creates a new version of the object and updates the version stamp for the object unless the object is newly created in the same revision. Second, when a new object is created, the references of its parent objects are updated. For example, node creation causes update of the diagram object, and edge creation causes update of the parent graphic nodes. O_2 updates the version stamp and carries out the above update process for the parent objects. Therefore additional object creation and version stamp update may be required. However, DIVERS just updates the parent objects and adds the create operation to the operation list. Third, when an object is deleted, its parent objects are also updated. Deletion of an edge is similar to creation of an edge. But deletion of a node causes deletion of its edges and update of the nodes connected with the edges. In this case, O_2 updates the version stamp and carries out the update process for the diagram object and the connected nodes. But DIVERS adds the delete operation to the operation list and just updates the diagram object and the nodes. In DIVERS, the time to add an operation to a operation list is $O(1)$ regardless of the number of revisions. O_2 searches the version stamps to update them. The searching time is proportional to the number of changes, which is proportional to the number of revisions in average.

Even if a few attributes are updated, O_2 copies the whole object. Moreover, creation and deletion of objects cause implicit update, and it may require copying the objects. However, DIVERS uses delta for changed objects and does not require additional storage overhead for implicit update.

DIVERS maintains the latest revision of each branch. To retrieve an old revision, the definition of visibility is used. The time complexity of the visibility definition is proportional to the number of the operations from the revision to the latest revision. DIVERS stores operation lists separately to localize the scope of operation searching. In contrast, O_2 searches the version stamps to obtain a specific database version. The searching time for version stamps is also proportional to the number of the operations, but it is larger than that of DIVERS because creation and deletion of objects in O_2 may cause another update operations.

4 Merge Model

Our merge model differs from structure-oriented merge [10] in two aspects. First, graphic information is ignored in merging two diagrams because it generally makes too much conflict. Second, assuming that deletion of the useless objects is easier than creation of the objects that are not included, we chose a passive method that includes all possible objects. Instead, DIVERS informs a user that some useless objects are included.

The 3-way merge rules for merging the two versions originated from the base version are presented in Table 4. Nodes are changed by update of their attributes, creation of new edges, or changes of their edges, while edges are changed by update of their attributes. The nodes of two versions are merged by the rules in Table 4. When conflict occurs, the rules for edges and set-valued attributes are used to resolve the conflict between the two nodes. But conflict in atomic attributes cannot be resolved without user interaction.

Table 4 3-way merge rules

v1	v2	node	edge	set-valued attribute
changed	not changed	conflict	v1 (v2)	v1 (v2)
changed	changed	conflict	v1 v2	v1 v2
not changed	deleted	(v1)	(v1)	(v1)
changed	deleted	v1	v1	v1
created	created	conflict	v1 v2	v1 v2

v1	: includes the object of v1
v1 v2	: includes both the objects of v1 and v2
(v1)	: includes the object of v1, but advises users to exclude it
v1 (v2)	: includes both the objects of v1 and v2, but advises users to exclude the object of v2

5 Conclusion

In this paper, we proposed an efficient diagram version model, which reflects the structure of software diagrams and uses editing operations as delta. Based on the model, a prototype of DIVERS was implemented on top of ObjectStore. It can be implemented on top of any DBMSs that do not provide object versioning since it does not use the version management facilities of OODBMS. We plan to implement a diagram editor that provides operations lists to DIVERS.

References

[1] E. Bertino and L. Martino, *Object-Oriented Database Systems*, Addison-Wesley, 1993.

[2] W. Cellary and G. Jomier, “Consistency of Versions in Object-Oriented Databases”, *Building an Object-Oriented Database System, The Story of O2*, Morgan Kaufmann, 1992.

[3] ECMA, *Portable Common Tool Environment* (*PCTE*) *Abstract Specification*, Standard ECMA-149, March 1995.

[4] The GOODSTEP Team, “The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes”, *Proceedings of the APSEC`94*, 1994.

[5] P. Lindsay, Y. Liu and O. Traynor, “A Generic Model for Fine Grained Configuration Management Including Version Control and Traceability”, *Proceedings of the Australian Software Engineering Conference* (*ASWEC`97*), September 1997.

[6] B. Magnusson, U. Asklund, and S. Minor, “Fine-Grained Revision Control for Collaborative Software Development”, *ACM SIGSOFT'93 - Symposium on the Foundations of Software Engineering*, December 1993.

[7] B. P. Munch, Jens-Otto Larsen, B. Gulla, R. Conradi, and Even-Andre Karlsson, “Uniform Versioning: The Change-Oriented Model”, *SCM-4: 4th International Workshop on Software Configuration Management*, May 1993.

[8] R. Ramakrishnan and D. J. Ram, “Modeling Design Versions”, *Proceedings of the twenty-second international Conference on Very Large Data Bases*, September 1996.

[9] E. Sciore, “Version and Configuration Management in an Object-Oriented Data Model”, *VLDB Journal*, Vol.3-1, 1994.

[10] B. Westfechtel, “Structure-Oriented Merging of Revisions of Software Documents”, *SCM-3: 3rd International Workshop on Software Configuration Management*, 1991.

[11] B. Westfechtel, “A Graph-Based System for Managing Configurations of Engineering Design Documents”, *International Journal of Software Engineering & Knowledge Engineering*, vol. 6-4, 1996.