미세 단위 소프트웨어 객체를 위한 연산 기반 버전 및 일관성 관리 모델
(An Operation-Based Model of Version Storage and Consistency Management for Fine-Grained Software Objects)

노 정 규[†] 우 치 수[††]
(Jungkyu Rho) (Chisu Wu)

요 약 소프트웨어 문서는 수많은 논리적인 객체와 객체간의 관계로 이루어진 구조를 가지고 있으며 그 구조가 빈번하게 변경될 수 있다. 본 논문에서는 소프트웨어 편집 과정에서 적용되는 연산에 기반한 미세 단위 소프트웨어 객체의 버전 및 일관성 관리 모델을 제안하였다. 모든 소프트웨어 객체는 연산으로 구성된 인터페이스를 가지고 객체의 편집은 연산을 통해서 이루어진다. 편집기를 통하여 객체에 적용된 연산은 연산 히스토리에 기록되고 버전 관리와 소프트웨어 구성 요소간의 일관성 관리에 이용된다. 객체의 버전은 연산 히스토리를 이용한 델타를 이용하여 저장되고 검색되므로 델타 추출을 위한 비교 과정이 필요 없으며 버전간의 변경 내용을 쉽게 파악하여 버전 전파 여부를 결정지을 수 있다는 장점이 있다. 일관성은 객체간의 종속성과 객체에 적용된 연산의 종류에 의해 관리되므로 불필요한 변경 전파를 피할 수 있다. 본 논문에서는 객체에 적용된 연산을 기반으로 하여 버전 검색 및 미세 단위 일관성에 대한 정형적인 모델을 제시하였다.

Abstract Software documents consists of a number of objects and relationships between them, and structure of documents can be changed frequently. In this paper, we propose a version storage and consistency management model for fine-grained software objects based on operations applied to edit software objects. An object has an interface and can be updated only through operations defined in its interface. Operations applied to objects are recorded in the operation history, which is used to retrieve versions of a document and manage consistency between documents. Because versions of an object are stored and retrieved using the operation delta, it is not needed to compare versions of a document to extract delta and it is easy to identify the changes between versions in order to propagate the changes. Consistencies between documents are managed using dependencies between objects and kinds of the operations applied to the objects. Therefore unnecessary version propagation can be avoided. This paper presents a formal model of version retrieval and consistency management at the fine-grained level based on operations applied to the objects.

1. 서 론

Software documents have logical structure and a lot of objects and relationships between them, and both individual objects and structure of documents can be changed frequently. For example, a document may have classes, functions, and relationships between classes and functions, and new classes can be added, some existing classes can be removed, or some relations can be modified. However, most of the existing software configuration management systems maintain a software document as a single file. In those environments adopting coarse-grained object management, dependencies between software objects cannot be represented. When an object is

† 정 회 원 : 삼성전자 통신연구소 연구원
jkrho@telecom.samsung.co.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
wuchisu@selab.snu.ac.kr

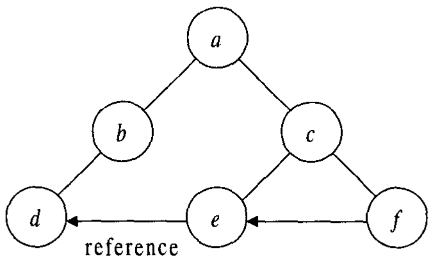
논문접수 : 1999년 4월 9일
심사완료 : 2000년 5월 30일

changed, it is desirable to easily identify the change and the range of change propagation for efficient software development and maintenance. However, it is difficult to identify those in coarse-grained object management models. Traditional coarse-grained configuration management models are adequate for managing software documents which have comparably a few dependencies between components.

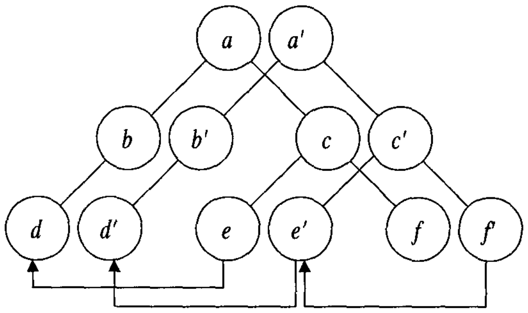
In contrast, fine-grained configuration management maintains structure information contained in software documents. The advantages of fine-grained configuration management are to preserve inter-document dependencies and to control change propagation at fine-grained level [1]. Moreover it can avoid the mismatch between logical structure and physical structure of software objects [2]. Fine-grained configuration management is useful for software diagram, like class diagram as well as the application which requires sophisticated dependency management. But the existing fine-grained object management models are not suitable for managing software objects when the structure of a document is frequently changed.

Fine-grained software configurations consist of component objects, and may have composition and reference relations between objects. When the same relation refers all versions of an object, the relation is said to be *generic*. On the other hands, if each version of an object is bound to a specific relation, the relation is said to be *bound*. A bound configuration is a composite object that has bound relations to its components. When an object was updated in a bound configuration, a new version of the object should be created and existing references should be updated to point the new version. For example, Figure 1 shows a bound configuration. When the object *d* was updated, *d*, a new version of *d*, was created and the change was propagated through not only composition relations but also reference relations. Due to this version proliferation problem, bound configurations are not suitable for storage management of fine-grained configurations.

Some existing fine-grained configuration management models, such as [1] and [3], were implemented on O_2 [4], an OODBMS supporting generic configurations. However, O_2 may have some inefficiencies with fine-grained configuration where component objects are created and deleted frequently. Another issue of fine-grained configuration management is how to extract delta between versions of a structured document. In general, it is necessary to compare every object in two versions, and the comparison time is proportional to the average number of objects in a version.



(a) Initial state



(b) After updating the object d

Fig. 1 Version proliferation in a bound configuration

As stated above, there are a number of relationships and dependencies between software components. To maintain consistency at fine-grained level, we need to know existence of dependencies between versions of documents, occurrence of changes, the contents of the changes, which documents the changes should be propagated to, and which parts of the dependent documents should be changed. In coarse-grained software development environments, dependencies are represented in terms of files. In such environments, when contents of a file were changed, it is difficult to know which parts of the dependent documents should be changed. Fine-grained configuration models, such as [1] and [2], can manage dependencies at fine-grained level, but changes are propagated regardless of the type. The existing configuration management tools can distinguish only between the interface change and the implementation change. However, it is desirable to distinguish changes of information, like of comment or graphical information, which is not directly related to the design of a software system, with changes of the design information. In addition, the complexity of dependency management cannot be ignored because fine-grained configurations may have a great number of dependencies between objects.

We propose an operation-based version storage and consistency management model for fine-grained software configurations. A configuration consists of documents, modules, and elementary objects. Based on the encapsulation property of software modules, we imposed a restriction, that is the destination of an inter-document dependency should be coarser than a module, in order to reduce the complexity of fine-grained dependency management. Operations applied to edit software objects, such as creation, deletion, and update, are recorded in an operation history, which is used to store the result of editing. Accordingly, it is needless to compare two versions to extract delta. Operation histories are used as backward delta between versions as well as inconsistency information. Inter-document consistency at fine-grained level is managed by operation histories and dependencies.

This paper is organized as follows. Section 2 presents an operation model of fine-grained objects management. Section 3 describes a version storage and consistency management model based on operation histories. Section 4 reviews the related work, and Section 5 gives a conclusion.

2. An Operation Model of Object Management

Our software configuration model consists of documents, modules, elementary objects, and relations between them. All objects should be created, deleted, and updated by the operations defined in their interface. Figure 2 shows an example of objects and relations between them.

An *elementary object* is a unit of software components. For example, statements, while loops, variables are elementary objects. A *module* is a unit of encapsulation. Objects residing outside a module are independent of the internals of the module. For example, classes, functions, and global variables are modules. The destination of an inter-module dependency must be coarser than a module. A *document* is a unit of editing and consists of modules, objects, and relations. A document may have several versions. Relations between objects are categorized into composition relations and dependencies, and relations can be created or deleted through the version history of a document. Object type definition has an interface that consists of operations that are used to change objects of the type. A *configuration* is a set of documents related by dependencies. Composition hierarchies of objects can be constructed from a number of documents, modules, and elementary objects. Not only a document but also a module

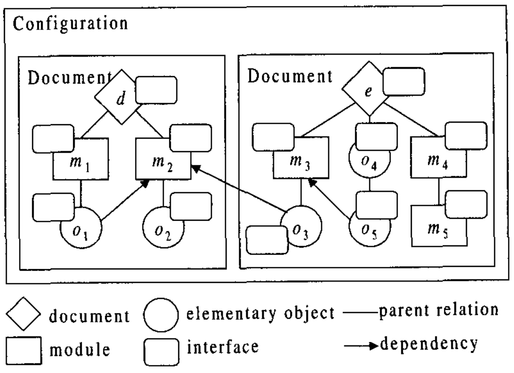


Fig. 2 Software objects and relations

can have elementary objects and modules as its children. In contrast, a composite elementary object can have only elementary objects as its children.

Generally, software documents are edited by people. In our model, every object has an interface and can be changed only by the operations defined in the interface. Operations applied by editing are recorded in a transient operation history, which is used to store the result of editing. Figure 3 shows an example of an editing and check-in process. Transient objects are manipulated by editors and reside in volatile storage. Operations applied to transient objects are recorded in a transient operation history. During the check-in process, the operations in the transient operation history are applied to persistent objects. Persistent objects are stored in non-volatile storage, such as file systems and databases. Operations applied to persistent objects are stored in a persistent operation history and used for version storage and consistency management.

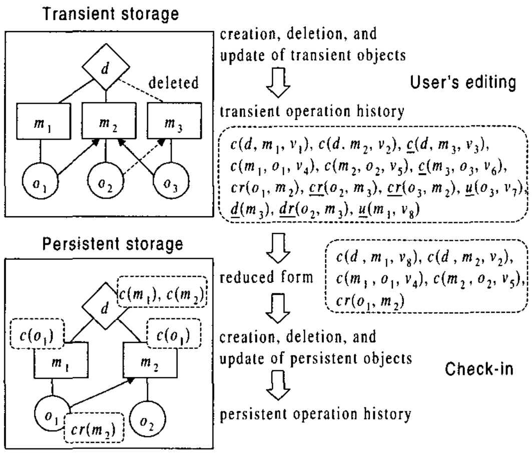

Transient storage
deleted
d
m_1
m_2
m_3
o_1
o_2
o_3
creation, deletion, and update of transient objects
User's editing
transient operation history
$c(d, m_1, v_1)$, $c(d, m_2, v_2)$, $\underline{c}(d, m_3, v_3)$, $c(m_1, o_1, v_4)$, $c(m_2, o_2, v_5)$, $\underline{c}(m_3, o_3, v_6)$, $cr(o_1, m_2)$, $\underline{cr}(o_2, m_3)$, $\underline{cr}(o_3, m_2)$, $\underline{u}(o_3, v_7)$, $\underline{d}(m_3)$, $\underline{dr}(o_2, m_3)$, $\underline{u}(m_1, v_8)$
Persistent storage
$c(m_1)$, $c(m_2)$
$c(o_1)$
$c(o_1)$
$cr(m_2)$
reduced form
$c(d, m_1, v_8)$, $c(d, m_2, v_2)$, $c(m_1, o_1, v_4)$, $c(m_2, o_2, v_5)$, $cr(o_1, m_2)$
creation, deletion, and update of persistent objects
Check-in
persistent operation history

Fig. 3 Operation-based editing and check-in

A transient operation history consists of $c(y, x, v)$, $d(x)$, $u(x, v)$, $cr(x, y)$, and $dr(x, y)$, which are creation of x as a child of y with the value v, deletion of x, update of x with the value v, creation of a dependency from x to y, and deletion of a dependency from x to y, respectively. Transient operation histories can also be used to support undo and redo in the editing process. A persistent operation history consists of $c(x)$, $d(x)$, $u(v)$, $cr(y)$, and $dr(y)$, which are creation of x, deletion of x, update with the value v, creation of a dependency to y, and deletion of a dependency to y, respectively. Create operations without attribute value specifications and inverse update operations are stored in a persistent operation history. For example, when the value of the object x is v_1 and $u(x, v_2)$ is applied, the value of x will be set to v_2 and $u(v_1)$ will be recorded. Persistent operation histories are stored separately in the corresponding objects to reduce searching time to retrieve previous versions. So $u(v)$, $cr(y)$, and $dr(y)$ are stored in the applied object, whereas $c(x)$ and $d(x)$ are stored in the parent of x.

We defined *object visibility* to manage hierarchies of objects. When an object was created, it is visible from its parent object. Later, when the object is deleted, it will be invisible from the parent. An object is visible if and only if its parent is visible and it is also visible from the parent. Before defining object visibility, existence and parent relation of objects should be defined.

Definition 1 *Existence* of an object x, $E(x)$, is true if and only if x has been created ever.

Definition 2 Let O be a set of objects. *Parent* relation P is a binary relation on O, and $<x, y>$ is an element of P if y is the parent of x. P' is a binary relation on O, and it represents y had been the parent of x.

Definition 3 *Ancestor* relation P^+ is a quasi order on O. *Historic ancestor* relation P^* is a quasi order on O, which includes the past ancestor relations as well as the present relations.

$$P^+ = \{ <x, y> \mid <x, y> \in P \vee \exists z (<x, z> \in P^+ \wedge <z, y> \in P^+) \}$$

$$P^* = \{ <x, y> \mid <x, y> \in (P \cup P') \vee \exists z (<x, z> \in P^* \wedge <z, y> \in P^*) \}$$

Definition 4 Let O and M be sets of objects and modules, respectively. *Dependency* relation R is a subset of $O \times M$, and $<x, y>$ is an element of R if x depends on y.

Definition 5 *Visibility* of the object x, $V(x)$, is true if and if only there exists an object y such that $V(y)$ is true and $<x, y>$ is a element of P. Visibility of a dependency $<x, y>$, $V(<x, y>)$, is true if and only if $V(x)$ and $V(y)$ are true and $<x, y>$ is an element of R.

$V(x) \Leftrightarrow \exists y \ (V(y) \wedge <x, y> \in P)$

$V(<x, y>) \Leftrightarrow [V(x) \wedge V(y) \wedge <x, y> \in R]$

Note that existence of an object is true regardless of the visibility if the object was created once. We defined the *virtual root* r, of which visibility is always true. Since $V(r)$ is true, the actual root of a document can be created as a child of r.

Each operation has its pre-condition and post-condition. Table 1 shows pre-conditions and post-conditions of operations in terms of visibility and relations. A transient operation history should be valid in order to be used for object management. Informally, a valid operation history has no operations on invisible objects and no repeated creation of an object. We assumed that editors always produce valid operation histories.

Table 1 Pre-conditions and post-conditions of operations

pre-condition	operation	post-condition
$V(y) \wedge \neg E(x)$	$c(y, x, v)$	$V(y) \wedge <x, y> \in P \wedge x.value = v$
$V(y) \wedge <x, y> \in P$	$d(x)$	$V(y) \wedge <x, y> \notin P \wedge <x, y> \in P'$
$V(x)$	$u(x, v)$	$V(x) \wedge x.value = v$
$V(x) \wedge V(y) \wedge <x, y> \notin R$	$cr(x, y)$	$V(x) \wedge V(y) \wedge <x, y> \in R$
$V(x) \wedge V(y) \wedge <x, y> \in R$	$dr(x, y)$	$V(x) \wedge V(y) \wedge <x, y> \notin R$

Definition 6 Let $OP = <o_1, ..., o_n>$ be a transient operation history. OP is *valid* if the following conditions hold.

For all o_i and o_j belong to OP,

(1) $[(o_i = c(z, y, v) \wedge o_j = c(y, x, v')] \Rightarrow i < j$

(2) $[<x, z> \in P^* \wedge ((o_i = c(y, x, v) \wedge o_j = d(z)) \vee (o_i = d(x) \wedge o_j = d(z)) \vee (o_i = u(x, v) \wedge o_j = d(z)))] \Rightarrow i < j$

(3) $[(o_i = c(y, x, v) \wedge o_j = d(x)) \vee (o_i = u(x, v) \wedge o_j = d(x)) \vee (o_i = c(y, x, v) \wedge o_j = u(x, v))] \Rightarrow i < j$

(4) $[(o_i = c(y, x, v) \wedge o_j = c(y', x, v')] \Rightarrow [i = j \wedge y = y' \wedge v = v']$

(5) $[o_i = d(x) \wedge o_j = d(x)] \Rightarrow i = j$

(6) $[(o_i = c(z, x, v) \vee o_i = c(w, y, v')) \wedge o_j = cr(x, y)] \Rightarrow i < j$

(7) $[(o_i = cr(x, y) \vee o_i = dr(x, y)) \wedge ((o_j = d(z) \wedge <x, z> \in P^*) \vee (o_j = d(w) \wedge <y, w> \in P^*))] \Rightarrow i < j$

(8) $[(o_i = cr(x, y) \vee o_i = dr(x, y)) \wedge (o_j = d(x) \vee o_j = d(y))] \Rightarrow i < j$

(9) $[o_i = cr(x, y) \wedge o_j = dr(x, y)] \Rightarrow i < j$

(10) $[(o_i = cr(x, y) \wedge o_j = cr(x, y)) \vee (o_i = dr(x, y) \wedge o_j = dr(x, y))] \Rightarrow i = j$

For all o_j belong to OP,

(11) $o_j = c(y, x, v) \Rightarrow \exists o_i [o_i = c(z, y, v') \wedge i < j]$

(12) $[o_j = d(x) \vee o_j = u(x, v)] \Rightarrow \exists o_i [o_i = c(y, x, v) \wedge i < j]$

(13) $o_j = cr(x, y) \Rightarrow \exists o_i \exists o_k [o_i = c(z, x, v) \wedge o_k = c(w, y, v') \wedge i < j \wedge k < j]$

(14) $o_j = dr(x, y) \Rightarrow \exists o_i [o_i = cr(x, y) \wedge i < j]$

A transient operation history can be reduced before applying it to persistent storage. In other words, operations on temporarily created and deleted objects, update operations on newly created objects, repeated update operations on the same object, and operations on deleted objects can be removed.

Definition 7 *Operation subsumption rule S* is a quasi order on OP.

$S = \{ <o_i, o_j> \mid [<x, z> \in P^* \wedge o_j = d(z)$
$\wedge (o_i = c(y, x, v) \vee o_i = d(x) \vee o_i = u(x, v)$
$\vee o_i = cr(x, y) \vee o_i = cr(y, x) \vee o_i = dr(x, y)$
$\vee o_i = dr(y, x))]$
$\vee [o_j = d(x) \wedge (o_i = c(y, x, v)$
$\vee o_i = u(x, v) \vee o_i = cr(x, y) \vee o_i = cr(y, x)$
$\vee o_i = dr(x, y) \vee o_i = dr(y, x))]$
$\vee (o_j = c(y, x, v') \wedge o_i = u(x, v))$
$\vee (o_j = u(x, v') \wedge o_i = u(x, v) \wedge i < j)$

$\vee$ ($o_j = dr(x, y) \wedge o_i = cr(x, y)$) }

Definition 8 A reduced operation history can be constructed from a valid operation history by applying the following *operation substitution rule.*

Where $<o_i, o_j> \in S$,

(1) if $o_i = c(y, x, v)$ and $o_j = d(x)$, then substitute $\varnothing$ for o_i, o_j, and all p such that $<p, o_j> \in S$.

(2) if $o_i = u(x, v')$ and $o_j = c(y, x, v)$, then substitute $\varnothing$ for o_i, and substitute $c(y, x, v')$ for o_j.

(3) if $o_i = cr(x, y)$ and $o_j = dr(x, y)$, then substitute $\varnothing$ for o_i and o_j.

(4) otherwise, substitute $\varnothing$ for o_i.

Figure 3 shows an example of a transient operation history and the reduced form. The operations applied by the editor are listed in the transient operation history. Deletion of a module enforces automatic deletion of dependent objects in a document. In contrast, deletion of an object does not require deletion of child objects or dependencies. For example, $dr(o_2, m_3)$ follows $d(m_3)$ in the transient operation history, whereas there does not exist $d(o_3)$ or $dr(o_3, m_2)$ after $d(m_3)$. Since m_3 was deleted, $c(d_1, m_3, v_3)$, $c(m_3, o_3, v6)$, $cr(o_2, m_3)$, $cr(o_3, m_2)$, $u(o_3, v_7)$, $d(m_3)$, and $dr(o_2, m_3)$ were removed by the substitution rule. In addition, $c(d_1, m_1, v_1)$ and $u(m_1, v_8)$ were replaced by $c(d_1, m_1, v_8)$.

Theorem 1 The resulting object visibility does not affected by the substitution.

Proof : The proof is in four parts.

(1) Suppose $o_i = c(y, x, v)$ and $o_j = d(x)$. By Definition 6, operations on x or the descendants of x can exist only between o_i and o_j. Since $V(x)$ is false after o_j, visibility of every descendant of x is false. Even after the substitution, $V(x)$ and visibility of the descendants are false since all of the operations on x or the descendants are removed. By Definition 5, $V(x)$ can influence only visibility of the descendants(including the dependencies) of x. Therefore visibility does not affected by the substitution.

(2) Suppose $o_i = u(x, v')$ and $o_j = c(y, x, v)$. By Definition 6, i is greater than j. Since o_i does not influence visibility of any objects, it is sufficient to consider only the value of x. Therefore $c(y, x, v')$ can be substituted for o_j, and o_i can be removed without affecting visibility.

(3) Suppose $o_i = cr(x, y)$ and $o_j = dr(x, y)$. By Definition 6, there does not exist operations on $<x, y>$ or deletion of the ancestors of $<x, y>$ between o_i and o_j. Therefore $V(<x, y>)$ is false before and after substitution. Since o_i and o_j do not influence visibility of any other objects, visibility does not affected by the substitution.

(4) Suppose o_i is an operation on x, and o_j is deletion of x or an ancestor of x, By Definition 6, i is less than j. Since visibility of x and the descendants of x is false after o_j, o_i can be removed without affecting visibility. In addition, where $o_i = u(x, v)$, $o_j = u(x, v')$, and $i < j$, o_i can be removed since any update operations do not affect visibility. *Q.E.D.*

3. Operation-Based Fine-Grained Configuration Management

3.1 Version Storage Management

In our operation-based version storage model, when update operations are applied to a persistent object, the object will be updated and the operations will be recorded as backward delta. Even when deletion of an object is applied, the object will not be visible instead of being removed. Since updated objects are not copied, all relations between objects can be treated as generic relationships. A version has its persistent operation history that was applied to the previous version to produce the version. To retrieve a previous version of a document, persistent operation histories of the document are used. We defined visibility in the i-th version in terms of the latest version of a document and its persistent operation histories. Attribute value of an object in the i-th version is determined by the earliest (inverse) update operation between the i+1th version and the latest version.

Definition 9 Let OP_k be the transient operation history applied to derive the i-th version from the previous version, and let ∪ be the append operator of operation histories. So $\bigcup_{k=n}^{m} OP_k$ is the appended union of the operation histories between the n-th version and the m-th version. If $\bigcup_{k=1}^{l} OP_k$ satisfies the conditions of Definition 6, then OP_k is said to be *valid*.

Definition 10 Let OP_{ky} be the persistent operation history of the k-th version stored in the object y, and let L_y be the set of child objects of y in the latest version. And let C_{iy} and D_{iy} denote the sets of created and deleted child objects of y between the i+1th version and the latest version, respectively. *Object visibility* in the i-th version defined as follows, where l is the latest version number and $1 \le i < l$.

$L_y = \{ x \mid <x, y> \in P \}$

$C_{iy} = \{ x \mid c(x) \in \bigcup_{k=i+1}^{l} OP_{ky} \}$

$D_{iy} = \{ x \mid d(x) \in \bigcup_{k=i+1}^{l} OP_{ky} \}$

$V_i(x) \Leftrightarrow \exists y \, [\, V_i(y) \wedge x \in ((L_y \cup D_{iy}) - C_{iy}) \,]$

Definition 11 Let LR_x be the set of destinations of the dependencies of x. And let CR_{ix} and DR_{ix} denote the sets of destinations of created and deleted dependencies of x between the i+1th version and the latest version, respectively. Then *dependency visibility* in the i-th version is defined as follows, where $1 \le i < l$.

$LR_x = \{ y \mid <x, y> \in R \}$

$CR_{ix} = \{ y \mid cr(y) \in \bigcup_{k=i+1}^{l} OP_{kx} \}$

$DR_{ix} = \{ y \mid dr(y) \in \bigcup_{k=i+1}^{l} OP_{kx} \}$

$V_i(<x, y>) \Leftrightarrow [\, V_i(x) \wedge V_i(y) \wedge y \in ((LR_x \cup DR_{ix}) - CR_{ix}) \,]$

For example, Figure 4 shows three versions of a document and persistent objects with operation histories. Version 1 has no operation history because it will not be used anywhere. In derivation of version 2, $c(o_6)$, $d(o_5)$, $c(o_7)$, $c(o_8)$, $cr(o_8, m_2)$ and $u(o_3, v_1)$ were applied. Note that deletion of o_5 made o_5 and $<o_5, m_2>$ invisible, and v_0, the value of o_3 in version 1 was recorded in the update operation. In derivation of version 3, $d(m_1)$, $dr(o_2, m_1)$, $d(o_6)$, $u(o_3, v_2)$, and were actually applied. The underlined operations, $\underline{dr(o_8, m_1)}$ and $\underline{d(o_8)}$, exists in the transient operation history but those are subsumed by $d(o_6)$. Deletion of m_1 made its children invisible and enforced deletion of $<o_2, m_1>$. Suppose a user wants to retrieve version 1 after creation of version 3. Then version 3 and operation histories are used to obtain version 1. First, visible children of the root of the document in version 1 should be obtained from the definition of visibility. Then visible children and dependencies of those can be obtained successively. The value of o_3 is decided by $u(v_0)$.

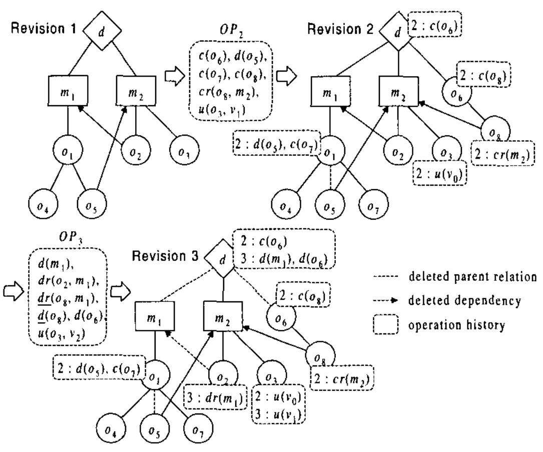


Fig. 4 Objects and operation histories in each version

Theorem 2 shows that the definition of visibility in the i-th version can retrieve the i-th version.

Theorem 2 The definition of visibility in the i-th version always produces correct results, i.e. $V(x)$ in the i-th version is equivalent to $V_i(x)$ in the latest version.

Proof : Let P_i be the parent relation in the i-th version and d denote a document. To show the correctness of object visibility, we first prove that visibility of child objects of d is correct. Note that $V(d)$ and $V_i(d)$ are always true.

$(\Rightarrow)$

$V(x) \Rightarrow V(d) \wedge <x, d> \in P_i$

$\Rightarrow <x, d> \in P_i \wedge (x \in D_{id} \vee x \notin D_{id}) \wedge (x \in C_{id} \vee x \notin C_{id})$

$\Rightarrow <x, d> \in P_i \wedge [(x \notin D_{id} \wedge x \notin C_{id}) \vee (x \notin D_{id} \wedge x \in C_{id}) \vee (x \in D_{id} \wedge x \notin C_{id}) \vee (x \in D_{id} \wedge x \in C_{id})]$

$\Rightarrow$ $<x, d> \in P \vee F \vee (<x, d> \in P_i \wedge x \in D_{id} \wedge x \not\in C_{id}) \vee F$

$\Rightarrow x \in L_d \vee (x \in D_{id} \wedge x \not\in C_{id})$

$\Rightarrow x \in (L_d \cup D_{id}) - C_{id}$

$\Rightarrow V_i(d) \wedge x \in (L_d \cup D_{id}) - C_{id} \Rightarrow V_i(x)$

$(\Leftarrow)$

$V_i(x) \Rightarrow V_i(d) \wedge x \in (L_d \cup D_{id}) - C_{id}$

$\Rightarrow (x \in L_d \vee x \in D_{id}) \wedge x \not\in C_{id}$

$\Rightarrow (<x, d> \in P \wedge x \not\in C_{id}) \vee (x \in D_{id} \wedge x \not\in C_{id})$

$\Rightarrow (<x, d> \in P_i \wedge c(x) \wedge OP_a) \vee (d(x) \in OP_b \wedge c(x) \in OP_c)$

where $a \leq i$, $i < b \leq l$, and $c \leq i$.

$\Rightarrow <x, d> \in P_i \vee <x, d> \in P_i \Rightarrow V(x)$

Now we can prove the correctness of visibility of any descendant objects inductively. We skipped the induction. We also skipped the proof of dependency visibility since it is similar to the of object visibility. *Q.E.D.*

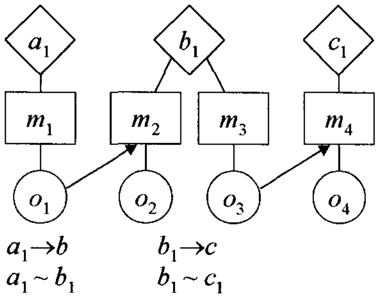
3.2 Consistency Management

This section deals with the issues of inter-document consistency management. Dependencies between objects and modules can exist across document boundaries. If an object x depends on a module m, then a dependency $<x, m>$ exists. So changes of m may require changes of x, or not, according to the change type. We defined *depend on* and *consistent with* relations to represent relationship between versions of documents. Dependencies can be added or removed as documents are being changed.

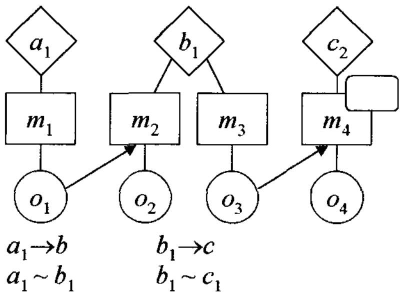
Definition 12 Let a_i be the i-th version of the document a, and R_{ai}, P_{ai}^{+} , and P_b^{*} are the inter-document dependency relation of a_i, the ancestor relation of a_i, and the historic ancestor relation of the document b, respectively. Then a_i *depends on* b, denoted by $a_i \rightarrow b$, if the following condition holds.

Definition 13 If a_i depends on b, and a_i was created based on the contents of b_j or consistency of a_i was checked with respect to b_j, then a_i is *consistent with* b_j, denoted by $a_i \sim b_j$.

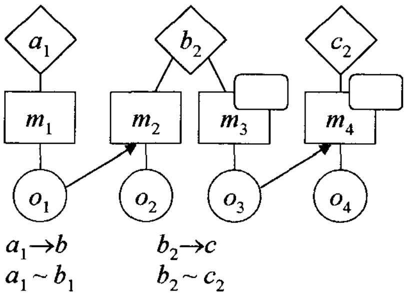
Figure 5 shows an example of versions of documents and dependencies and consistencies between them. When a new version of a document



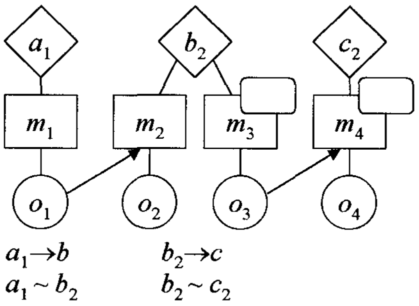
(a) Initial state



(b) Updating c



(c) Propagating the change to b



(d) Checking consistency of a

Fig. 5 Dependencies between documents

is created, public attributes and operation histories of a module are visible to other documents and can be used to propagate changes. A module has a visibility attribute that informs the dependent objects in other documents of deletion of the module. When a module or its ancestor is deleted, its visibility attribute will be set to false. As inconsistencies may occur frequently and repeatedly in the same object, we adopted lazy inconsistency resolution policy to avoid repetitive inconsistency resolutions. A change on a module is propagated to the dependent objects according to the operation type, only when the dependent documents request consistency checking.

To check consistency of a_l, the latest version of the document a, the *depends on* relations, such as $a_l \rightarrow b$, are searched by the depth first manner. If b_m, the latest version of the document b, has no documents which b_m depends on, or b has been visited once, then b_m is consistent. If $a_l \sim b_m$, a_l is consistent with the latest versions of other documents. If a_l is not consistent with b_m, then we investigate whether the change should be propagated to a_l. If so, a_{l+1} is created and $a_{l+1} \sim b_m$ is added. Otherwise, $a_l \sim b_m$ is added. For example, Figure 5(a) shows an initial consistent configuration. In Figure 5(b), c_2 was created and the change was not propagated yet. Suppose we want to know consistency of a_1, then b_1 and c_2 are visited subsequently. Because c_2 is consistent, consistency of b_1 is checked next, and b_2 is created and $b_2 \sim c_2$ is added (Figure 5(c)). In Figure 5(d), a need not to be changed and $a_1 \sim b_2$ is added because b_1 and b_2 are equivalent at the point of the dependency between o_1 and m_2.

Thus, we are able to know occurrence of potential inconsistencies from the latest version number of documents and *consistent with* relations. And the contents of the changes are provided by operation history of the changed modules. The documents which can be influenced by the changes are determined by *consistent with* relations, and we can decide whether the change should be propagated according to the changed module and the applied operation type. The objects influenced by the changes are determined by the fine-grained dependency relations.

4. Related Work

A research on software engineering databases[5] proposed the concept of a configuration, a document, and an object, including intra-document, inter-document, and inter-configuration relationships. A fine-grained configuration management model[1] supports version management of both document and object levels. Relations are categorized into depends_on and based_on relations between documents as well as objects, and consistent_with relations between documents. These models do not have the concept of a module and were implemented on top of O_2.

Orm storage model[6] supports version control for fine-grained hierarchically structured documents, such as programs that consist of classes and functions. However, it supports only bound configurations. HiP[7] provides an efficient storage for versions of trees, which have only composition relationships. But HiP cannot manage a graph structure, which has a number of dependency relations, and it also supports only bound configurations.

In POEM[2], a programming environment, a software unit can be changed only by the operations defined in its type definition. But operations are not used for storage and consistency management, and POEM cannot avoid version proliferation as it supports bound configurations. CPRG[8] is a software architecture for consistency management at fine-grained level. CPRG model consists of components, which are modified by graph operations, and relationships between components. When changes occur in a component, the changes are stored in a change description, which can be used to support undo/redo and versioning, and propagated to the related components. However, it does not provide a detailed method to retrieve a consistent version of a composite object and is not suitable for managing fine-grained lightweight objects. And it does not support object management in transient storage.

CoMa[9] is a configuration management model, which is based on graph rewriting, but it cannot perform consistency control at fine-grained level since it follows a coarse-grained approach. A formal model of consistency of configurations vector was presented in [10]. But the model of [10] supports only coarse-grained dependencies, and documents composing a configuration and dependencies are fixed.

O_2[4] is considered as the best candidate for fine-grained version storage among existing OODBMSs because it supports generic configuration. O_2 supports object versioning using the concept of version stamps. But O_2 does not provide structured delta method, which is provided by our model. In O_2, when an object is created or deleted, the parent object should be duplicated, and deletion of an object requires updating version stamps of the descendants. Thus O_2 may have some inefficiency for managing a document which varies its structure frequently.

5. Conclusion

In contrast to other engineering documents, structure of software documents can be changed frequently. We proposed an operation-based fine-grained configuration management model, and a prototype of the transient and version storage model was implemented on top of ObjectStore, an OODBMS. Our model considers transient storage management as well as persistent storage management. To retrieve a version, the latest version and persistent operation histories are used to obtain object visibility. Since operation histories are separately stored to reduce searching time for delta and the space for storing a version is proportional to the number of applied operations, it is suitable for frequent creation of versions. Inter-document consistency can be managed by operation histories and fine-grained dependencies.

References

[1] S. Sachweh and W. Schafer, Version Management for tightly integrated Software Engineering Environments, *Proc. 7th Int'l Conf. Software Engineering Environments*, Apr. 1995.

[2] Yi-Jing Lin and Steven P. Reiss, Configuration Management with Logical Structures, *Proc. 18th Int'l Conf. Software Engineering,* 1996.

[3] P. Lindsay, Y. Liu and O. Traynor, A Generic Model for Fine Grained Configuration Management Including Version Control and Traceability, *Proc. Australian Software Engineering Conf.* Sep. 1997.

[4] F. Bancilhon, C. Delobel, and P. Kanellakis, *Building an Object-Oriented Database System, The Story of O_2*, Morgan Kaufmann, 1992.

[5] W. Emmerich, W. Sch fer and J. Welsh, Databases for Software Engineering Environments - The Goal has not yet been attained, *Proc. 4th European Software Engineering Conf.*, 1993.

[6] B. Magnusson and U. Asklund, Fine Grained Version Control of Configurations in COOP/Orm, *Int'l Workshop on Software Configuration Management*, Mar. 1996.

[7] E. J. Choi and Y. Kwon, An Efficient Method for Version Control of a Tree Data Structure, *Software- Practice and Experience*, vol. 27, no. 7, pp.797-811, Jul. 1997.

[8] J. Grundy, J. Hosking, and W. B. Mugridge, Inconsistency Management for Multiple-View Software Development Environments, *IEEE Trans. Software Engineering*, vol. 24, no. 11, pp.960-981, Nov. 1998.

[9] B. Westfechtel, A Graph-Based System for Managing Configurations of Engineering Design Documents , *Int'l J. Software Engineering and Knowledge Engineering*, vol. 6, no. 4, 1996.

[10] G. Heidenreich, D. Kips, and M. Minas. A New Approach to Consistency Control in Software Engineering, *Proc. 18th Int'l Conf. Software Engineering*, Mar. 1996.

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

[12] B. P. Munch, Jens-Otto Larsen, B. Gulla, R. Conradi, and Even-Andre Karlsson, Uniform Versioning: The Change-Oriented Model, *Proc. 4th Int'l Workshop Software Configuration Management*, 1993.

노 정 규
1991년 서울대학교 계산통계학과 학사. 1993년 서울대학교 계산통계학과 전산과학전공 석사. 1999년 8월 서울대학교 전산과학과 박사. 1999년 9월 ~ 현재 삼성전자 통신연구소 연구원. 관심분야는 소프트웨어 개발 환경, 소프트웨어 형상관리, 데이타베이스

우 치 수
정보과학회논문지 : 소프트웨어 및 응용
제 27 권 제 2 호 참조