RC24108 (C0611-002) November 13, 2006
Computer Science

IBM Research Report

Provenance-Aware Collaboration Developing Environment

Daming Hao, Lin Luo, Yabin Dang, Bo Hou*, Shunxiang Yang
IBM Research Division
China Research Laboratory
HaoHai Building, No. 7, 5th Street
ShangDi, Beijing 100085
China

*State Key Laboratory of Networking and Switching
Beijing University of Post and Telecommunication

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Provenance-aware Collaboration Developing Environment

Daming Hao
Service Asset TechnologiesIBM China Research Laboratory
haodm@cn.ibm.com

Lin Luo
Service Asset TechnologiesIBM China Research Laboratory
luolin@cn.ibm.com

Yabin Dang
Service Asset TechnologiesIBM China Research Laboratory
dangyb@cn.ibm.com

Bo Hou
State Key Laboratory of Networking and SwitchingBeijing University of Post and Telecommunication
polluxplus@gmail.com

Shunxiang Yang
Service Asset TechnologiesIBM China Research Laboratory
yangsx@cn.ibm.com

ABSTRACT

Open-source reuse has been advocated by many open source communities or even some commercial companies. Each coding artifact may be customized and integrated into new offering. Therefore, how to resolve the copyright issue when delivering the new open-source offering will be a serious problem. In our paper, we present a new document structure enabling trusted partition of each developer or company's contribution and thus provide a thorough way to avoid copyright issues. We develop a system called Ariadne based on Eclipse JDT which using proposed document structure instead of traditional ".java" source file to justify our approach. The experimental result shows the data processing efficiency and storage efficiency are both in an acceptable range.

Keywords

Open source, Copyright, Document Structure, Provenance

INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'04, Month 1–2, 2004, City, State, Country.
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

The prosperity of open source developing and reuse has exceeded the wildest expectations of many observers. In just a few short years, Open Source technologies like Apache, Linux, JBoss and MySQL have not only become viable

alternatives, but market leading and preferred solutions for a growing number of critical enterprise IT systems. More and more companies adopt open-source to save development effort and reduce the development lifecycle. Delivering high quality products in less time may help a company win the mounting competitive market. Besides reuse the outside open source, many large companies assetize their existing coding artifact, and hope these assets could be internally shared and reused for new projects. These activities can be regarded as open-source reuse of another type. These open-source including internal assets may be customized (e.g., trim, modify, etc.) and integrated into new offering throughout its evolvement lifecycle. When delivering the new open-source offering, the customized open-source, the new-attached code and the glue code have to be delivered to customer together, so the offering provider should get the open-source provider's agreement that allow the open source to be used in the new offering. A consequent headache is that such an agreement may not be available each time and an ambiguous agreement may cause many legal issues. Firstly, to review or to ask for such an agreement each adopting a piece of open-source may be burdensome and will bring great effort to the software developing lifecycle; secondly, some ambiguous agreements may cause potential legal issues and thus make the open-source adopter hesitate to adopt it (e.g., some agreements may require the offering that integrate it not to just wrap it, however, how to define wrap may be a confused problem); thirdly, some open-source may not have corresponding agreement for some specific integration or delivering pattern; finally, the business model will be restrict, for the open-source provider can get profit only from open-source adopter, but not the end user. It's difficult for the open-source provider to know the correct end user number and thus only some simple billing models are supported (e.g., one time deal), while some other billing models, such as "by final copy", will not be practical.

Furthermore, from the angle of maintenance, the originality-related information (e.g., author, owner, contract constraints, open source, rights, etc) of coding artifacts may be dropped and can hardly be retrieved after several rounds of modifications and collaborative work, which brings risks of maintenance and quality verification to the asset adopters. When team members attempt to share their code across projects, loss of originality-related information may make the adopter inadvertently involve improper open-source and cause code contamination, which is a high-risk issue that most companies try to avoid. Although there are many guidance in company to regulate proper reuse, people often omit them to avoid the effort of manual input originality information.

Currently there is no effective tool that can address such copyright related concerns throughout open-source reuse lifecycle in a thorough way. Embedding comments in source code can differentiate each developer's contribution and thus can clarify the copyright in some extent [1], but the deficiencies are obvious: firstly, since each coding artifact may be reused and modified for several rounds, to write and maintain the well-formatted comments is bothersome for most developers, and some of them will neglect it; secondly, such comments can be deleted or modified by following developers, thus it lacks for the warranty; thirdly, content and format of the comments is determined arbitrarily by the developers themselves so it is difficult to automatically manage them; finally, although the copyright has been clarified by comments, since the customized open-source code still have to be delivered to customer by the offering integrator, some copyright-related risk still can't be avoided..

Version control systems (e.g., CVS), allow one to keep track of different versions of a given document, including who created this version and when [2]. However, it still cannot solve the above issues, because it only maintains the code in a project-centric lifecycle. When artifacts are reused in some new project, the centrally-controlled records may be dropped during the transmission, thus it is difficult for the new team to retrieve the corresponding originality-related information from merely the snapshot of the code; Besides, this tools only record originality-related information during check-in/check-out, but cannot know whether the checked-in artifact is purely created by the author or whether parts of it were copied from others' artifacts.

"Track changes" tool in Word can record modification by a user, but just like the mechanism of CVS, such tracking features only record the current author of a document and the editing-related information [3]. For example, modification by user A in a document is copied by user B to another document, the information of user A will be lost in later document. In a word, the purpose of version control system and Word is mainly versioning, not to provide maintenance of originality-related information.

In contrast with previous traditional approaches, we try to address these copyright-related issues in a new way. Instead of marking the author or owner on each piece of a coding artifact to clarify the copyright, we partition each developer's contribution into different artifacts in an automatic way. Through selecting proper artifact to deliver, these copyright-related issues can be addressed in a thorough way. For example, when programmer A adopts a piece of open-source (the open-source can be regarded as artifact B) written by programmer B and makes some modification, then programmer A's modification is wrapped as artifact A. In order to avoid copyright-related risk, programmer A only need to delivery artifact A to customer. Once customer receives it, with the help of an automatic tool, the corresponding artifact B will be found through a link embedded in artifact A and gotten from programmer B's site with an agreement, then the tool will integrate these two artifacts into the original source code automatically. Through this approach, the copyright issues will be addressed and the risk of code contamination will be avoided.

However, traditionally, since the code pieces developed by different developers may be interwoven and the version of a source file may be updated frequently, to partition each developer's contribution and make the partition stable are very difficult. In proposed approach, we partition the provenance of a coding artifact in its developing lifecycle but not

partition its content. Provenance is a type of metadata that describes the operation history of a coding artifact, including the records of creating, modifying, deleting, referring, etc [4]. Since the provenance is flat incremental and the operation records performed by some one are usually successive, each developer's contribution is easy to be partitioned and the partition is stable.

In this paper we propose a new document structure enabling trusted partition of each developer or company's contribution and thus provide a thorough way to address copyright issues. The main concept is to record provenance information of a coding unit and partition each developer's successive records into different artifacts. We develop a system called Ariadne based on Eclipse JDT which using proposed document structure instead of traditional ".java" source file to justify our approach. The system is designed for coding artifact because it is a widely reused artifact type, but the concept can be easily extended to other artifacts.

The rest of the paper is organized as follows. The next section analyzes the requirements for designing such a document structure. Section 3 describes the proposed approach which can trustily partition each developer or company's contribution. Section 4 will show the experimental results and discuss the storage efficiency and data processing efficiency with proposed document structure.

REQUIREMENT ANALYSIS

We consider the functional and non-functional requirements for designing such a document structure through analyzing a scenario.

Scenario:

Alice and Bob are developers of Company COM, they are both in an open-source project ABC. When developing a module, in order to shorten the developing lifecycle, Alice adopted an open-source "Openlib" library from another open-source company DOT and modified the library to fit his interface. When this module has been finished, Alice sent it to Bob. Bob made some modification and integrate it with his modules. Finally, the entire source code of project ABC was published on an open-source community

Now let's suppose some possible circumstances and analyze them.

If the agreement of "Openlib" library is ambiguous, adopting it in project ABC and delivering it to customer may cause potential risk and thus make Alice hesitate to adopt it. Besides, review or ask for such an agreement may be burdensome. However, if the original "Openlib" library and Alice and Bob's modification are partitioned into different artifacts and only the artifacts that describe Alice and Bob's modification are published on open-source community, it's the customer's responsibility to decide if adopting "Openlib" library has any risk, so for company COM, the potential legal risk will be avoided.

If a bug is found in the source code of project ABC, it is difficult for the project manager to judge whether this bug is embedded in the original "Openlib" library, or it is imported by Alice or Bob's modification when the comments are neglected. However, if provenance of ABC project in the developing lifecycle is recorded, from the operation history, who imported this bug can be easily found.

Even well-formatted comments are embedded in source code, since they can be tampered easily, it can't be use as forensic. For example, if the project manager finds a bug and concludes that it was caused by Alice, Alice may contest that this bug was caused by Bob's modification for the comments in corresponding to this modification was neglect by Bob. However, if each developer's modification is confirmed by their signature, it will be incontestable.

Based on the analysis of the above scenario, the basic features of the proposed document structure will be:

Each developer or organization's contribution throughout the whole developing lifecycle should be partitioned into different artifacts.

The subject of each operation can be identified easily in order to find the bug maker.

Artifact that describes a developer's contribution can't be tampered by other developers or by himself, otherwise, the artifact can't be used as forensic.

The document structure should be generated and parsed automatically with least manual interfere.

Besides these functional requirements, there are some non-functional requirements:

Isomorphic structure. If a document is described with proposed document structure, no matter what the document type is, all the artifacts should be with the same fundamental structure in order to facilitate standard module to parse them.

Extensible. Proposed document structure can be extended easily to support more document types.

High storage efficiency and data processing efficiency. Record provenance information should not consume much memory space; generating and processing such document structure should not cause obvious lag.

PROPOSED APPROACH

Record provenance instead of content

In traditional source code file, since the code pieces developed by different developers may be interwoven and the version of a source file may be updated frequently, to partition each developer's contribution and make the partition more stable are very difficult. For example, if the source code needs to be warranted by the author, he should sign each coding artifact once having finished or modified it. If the version updating is very frequent, many redundancies will be generated and the data size will swell. Furthermore, if a coding artifact has been touched by multiple developers in different stage, how to differentiate each one's part and sign them separately will be an intractable problem. Besides this disadvantage, we can hardly find a universal fundamental document structure to describe different kinds of artifact, such as java source file and java project file, for few common attributes can be extracted from these different file types.

Provenance is a type of meta-data that describes the history or ancestry of an object. Provenance meta-data has been recorded and managed in many scenarios, such as database [5], archival community [6], etc. [7] creates a taxonomy of data provenance techniques, and apply the classification to current research efforts in the field. [8] and [9] discuss some of the technical issues that have emerged in an initial exploration of the topic.

It can be regarded that a traditional coding artifact is a set of content objects. For example, a line in a coding artifact and a license item both can be regard as content objects. Provenance of a coding artifact is the operation records of creating, modifying, deleting, referring these objects. In proposed document structure, we record the provenance on the content objects instead of the content objects themselves. Recording information in this approach has following advantages:

The provenance of different document types contains more common attributes than traditional document. For example, author, owner, modify time, reference links all can be regarded as the common attributes of a successive operation segment, even these operations describes documents of different format, such as coding artifact, word style document, etc. However, it is hard to extract so many common attributes in different traditional document types. Therefore, it is easy for us to construct a common fundamental structure for different document type if we record the operations on the content objects instead of the content objects themselves.

Some attributes we concern, such as author, importer, modify time, etc. are associated to the operation, but not the content. For example, a code artifact may be modified by multiple developer, so the modify time is associated to modifying operation, but not the coding line which is modified. Therefore, if we only record the content objects like traditional document type, there is no suitable position to record these attributes. Furthermore, the responsibility is associated to the operation, but not the content. For the code artifact modified by multiple developers, each developer only needs to be responsible to his operations, but not the final document that has modified by him. Therefore, use proposed document structure can make the responsibility more clear.

The operations related to an artifact are flat incremental. If an artifact is modified, we only need to attach some new operations that describe the modifying to the previous operations list and no previous record need to be changed, so the mentioned problem related to signature has been resolved.

If we record the operations on the content objects instead of the content objects themselves, all the documents can be partitioned according to each time of modification and the partition is stable. For example, the successive operations of a same user can be regard as a fundamental unit in proposed document structure.

The provenance contains more information than traditional document. Not only the original source code and other content objects (e.g., license items) can be extracted through processing the operations, but also some statistic can be made and multiform views can be generated after certain transformation. For example, based on the provenance, views can be generated that shows the operation history of a certain coding line or shows which lines has been touched by a certain developer or which lines are copied from open-source project. Therefore, recording provenance enables flexible views.

However, recording the provenance on the content objects instead of the content objects themselves will cause some performance issues. We will discuss these issues in next section.

Fundamental Unit

As mentioned above, each developer or organization's contribution throughout the whole developing lifecycle should be partitioned into different artifacts which can be regarded as the fundamental units of proposed document structure. So how do we define these fundamental units? If we treat the document content as a set of content objects, we can regard the successive operations performed by a same developer on these objects during once modification as such a fundamental unit, and call the fundamental unit "Version Section". The proposed document structure contains nothing except a set of version sections. Since the operations contained in each version section are only subjected to one developer, use version sections as fundamental units can partition each developer's contribution. Besides, the operation records will be fixed once the developer has finished his modification and quit from editor, and this feature will facilitate signing. Furthermore, we can extract some common attributes from every version section, such as author,

owner, modify time, thus we can define a uniform fundamental structure for each version section even they are used to describe documents of different types.

Link related version sections

As mentioned above, a version section describes the successive operations performed by a certain developer on content objects during once modification. Since the version sections describing same document may be subjected to different developers or organizations and may be kept in distributed regions, in order to retrieve the original document, all the related version sections should be collected from these regions. In proposed document structure, we use "Links" to connect all related version sections to facilitate collecting. A link will be embedded in current version section pointing to an early-created version section, which may be inherited, referred or depended on by current version section. Each version section may be connected to multiple early-created version sections through multiple links and all the links will be embedded in a special segment named "Links" in the version section.

Each version section uses its digest value as its ID, and a link is an URI that refers the ID of the corresponding version section. If the linked version section is also in local region, the link may only contain the ID, otherwise, if the linked version section is in a remote region, the URL of this region will be also attached in the link. Therefore, from a latest version section, through going through all the links recursively, all the related version sections will be collected in advance.

An attribute is used to describe the type of these links. Currently, we have pre-defined 3 link types: parent, reference & dependency. More types can be defined within the extension tag in the future.

The parent link means the current section inherits all the content objects contained in its parent section. For example, if we open a document described by version section A and modify it, the newly created version section B will have a parent linkage pointing to version section A. Each version section can only have one parent at most.

The reference link means the current section takes some content objects from its references version section. For example, if we copy some lines from version section A to version section B, version section B will have a reference linkage pointing to version section A. Each version section may have multiple reference version sections.

The dependency link means the current section depends on its dependency version section. For example, the version section describing a ".cpp" file will have a dependency linkage pointing to the version section describing the corresponding ".h" file. Each version section may have multiple dependency version sections.

Differentiating links with these types enables the document parser to decide which version sections should all be collected in advance, while which version sections may be lazy-loaded. For example, some version sections that are linked with "dependency" link type may not be required to be loaded in advance, thus the loading speed will be accelerated. Furthermore, differentiating links with these types also facilitate some statistic and analysis. For example, to found whose version sections are referred most frequently.

Operation Instruction Set

It is natural to regard each line in the source code as an object, for it make a good balance between storage space efficiency, data processing efficiency and object granularity. Besides lines in the source code, the document name and license items all can be regarded as objects, for they may be also modified by different developers. The version section contains operation records on these objects. Each new-created object will be assigned a new object ID. For different object type, different operation instruction set may be defined. For example, for the objects corresponding to lines in the source code, an instruction set that contains 6 atom instructions is defined as following to describe all operations on these lines.

LInsert (objectID, positionID, objectValue): insert a line objectValue to current document after the line indicated by positionID, and assign it a new object ID objected.
LModify (objectID, objectValue): change the content of the line indicated by objectID to objectValue.
LDelete (objectID): delete the line indicated by objected.
LRetrieve (objectID): retrieve the deleted line indicated by objected.
LCopy (objectID, positionID, referenceID, objectID_in_reference): copy the line indicated by objectID_in_reference from version section referenceID to current document after the line indicated by positionID, and assign it a new object ID objected.
LCopyModify (objectID, positionID, referenceID, objectID_in_reference, objectValue): copy the line indicated by objectID_in_reference from version section referenceID to current document after the line indicated by positionID, assign it a new object ID objectID, and change its content to objectValueon.

For the efficiency consideration, the operations recorded in the version section may be not the original operations. For example, even the modification of only a letter will generate a "LModify" operation which replace the entire old line with new value; some original operation, such as "replace", "replace all", will be mapped to the 6 atom instructions.

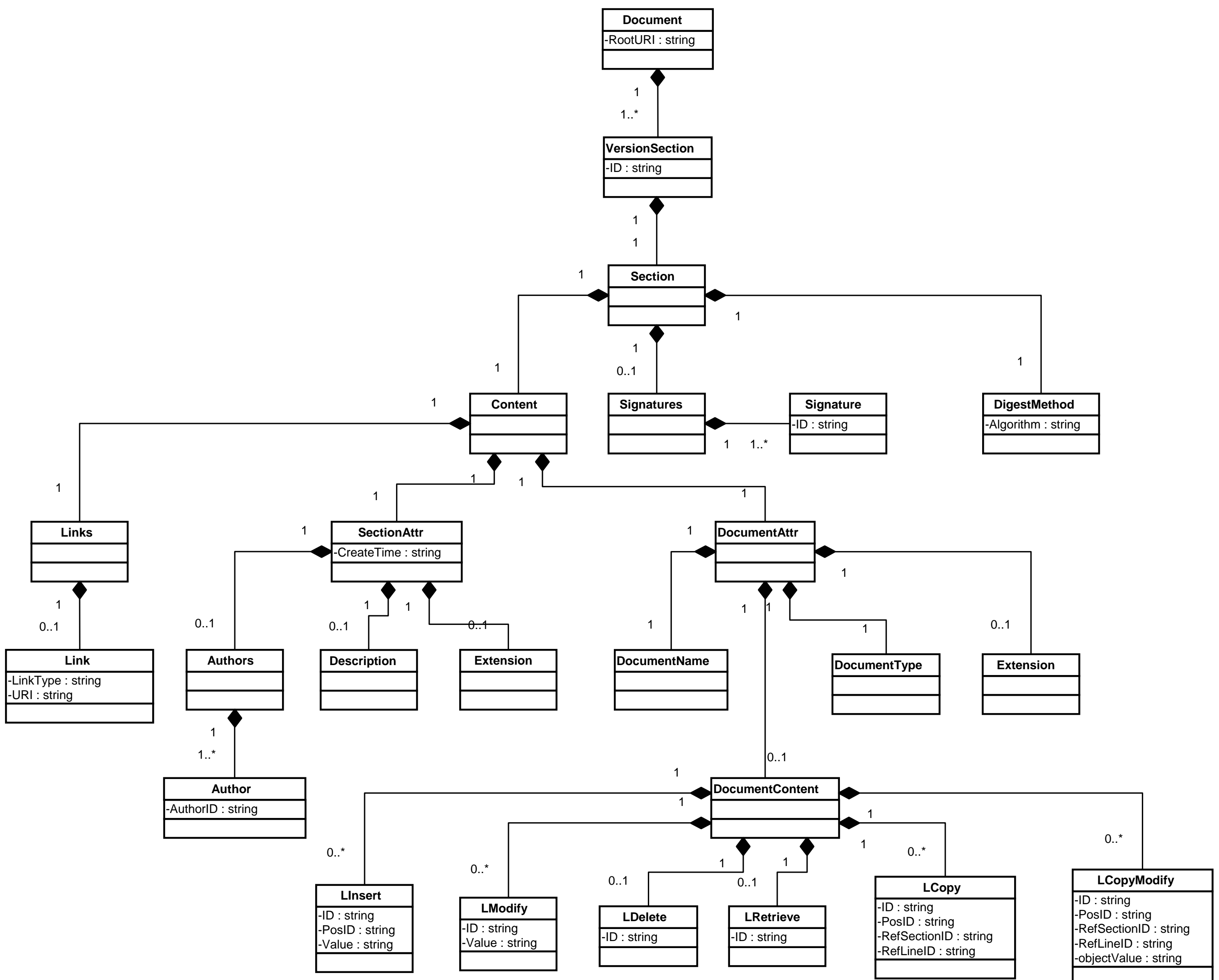
For other object type, the operation instruction set may be different. For example, for the objects corresponding to document name, an instruction set that contains 2 atom instructions is defined as following:

OAssign(objectValue): assign a new content objectValue to current object.

OModify(objectValue): change the content of current object to objectValue.

Structure of version section

As mentioned above, the operations records on the objects of a document during once modification are kept in a version section. A segment named "Document Attribute" is used to contain all these operations, including operations on document name, operations on source code lines, etc. the "Document Attribute" segment only contains the operation on objects but not the objects themselves, so all the objects described in this section can be modified in future version sections. However, some attributes that describe the version section itself, such as who and when created this version section, copyright of this version section, responsibility statement, will not be changed in the future, so the original content of these attributes are recorded, but not the operation on them. Another segment named "Section Attribute" is used to contain all these attributes. Besides, as mentioned, all the link items are embedded in a segment named "Links"; the signature(s) of the whole version section created by a warrantor(s) will be kept in another segment named "Signatures". Thus, each version section has 4 segments.



```
<?xml version="1.0" encoding="UTF-8"?>
<Document RootURI="#ID_bmh1aTEOMAwGA1UEBxMFbGl1">
  <VersionSection Id="ID_bmh1aTEOMAwGA1UEBxMFbGl1">
    <Section>
      <Content>
        <Links>
          <Link LinkType="Parent" URI="#ID_BAMTA2xoMTAeFw0wNTA4MTUw"/>
          <Link LinkType="Reference"
URI="#ID_MQ4wDAYDVQQIEwVhbmh1aTEO"/>
          <Link LinkType="Dependency"
URI="#ID_EwNjd2sxDDAKBgNVBAMTA2xo"/>
        </Links>
        <SectionAttr CreateTime="2006-02-17">
          <Authors><Author AuthorID="haodm@cn.ibm.com"/></Authors>
          <Description>Add a method</Description>
        </SectionAttr>
        <DocumentAttr>
          <DocumentName><OAssign>test.java</OAssign></DocumentName>
          <DocumentType><OAssign>javacode</OAssign></DocumentType>
          <DocumentContent>
              <LInsert ID="1" PosID="0" value="void msleep(long ms)"/>
              <LInsert ID="2" PosID="1" value="{"/>
              <LInsert ID="3" PosID="2" value="    int stTime;"/>
              …
          </DocumentContent>
        </DocumentAttr>
      </Content>
      <Signatures>
        …
      </Signatures>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    </Section>
  </VersionSection>
  <VersionSection Id="ID_BAMTA2xoMTAeFw0wNTA4MTUw">
    …
  </VersionSection>
  …
</Document>
```

The links segment records the links pointing to other version sections.

The section attribute segment records the attributes describing current version section, such as author, updated time, etc.

The document attribute segment records the successive operations records on the document objects..

The signatures segment contains the signature of the warrantor who declares his responsibility to the operations recorded in this version section. Multiple paratactic signature and the related certifications may be included in this segment.

Since all version sections have same structure even they describes different document types, standard software module or hardware can be used to parse the version section (e.g. verify the digest, signature, load sections). There are extension nodes on all the 4 segments for developers to define special attributes and process syntax with namespace. Figure 1 shows the UML model of proposed document structure. When proposed document structure is implemented

with XML language, a sample is shown in Figure 2. In the XML-based implementation, the latest version section is indicated in the root tag with "RootURI" attribute.

Incontestable

Considering a scenario, Alice writes a function and then this function is invoked in Bob's code, suppose Bob's code has been signed by himself while Alice's code has no warrant. If a bug is found in Bob's code when invoking this function, Bob may contest that is because Alice's code has been modified after he finished his part, thus the responsibility can't be clarified. It can be concluded from this scenario that a developer wouldn't be responsible for his modification if the dependency codes or the previous version has not be warranted by the corresponding authors. Thus, in order to ensure that each developer is with responsibility for has modification, not only their version sections should be warranted by themselves, but also all the version sections inherited, referred and depended by their version sections should be warranted by corresponding authors.

In proposed document structure, there are many mechanisms to make each developer's responsibility incontestable. First, the entire version section may be signed by one or more warrantor(s) to guarantee the facticity and announce his responsibility to the operations recorded in this version section. Not only the author, but also the importer may be the warrantor. The signature covers every segment in a version section, so neither the operation records, nor the links can be tampered. Second, each link contains the digest value of the linked version section. When the linked version section is loaded, the parser may calculate its digest value and compare it with the value given in the link. Thus, this approach prevents the linked version sections from being tampered. Furthermore, using digest value as the ID of a version section makes the ID exclusive at all environments, thus avoid the inaccurate linkage. Therefore, if the signing of each version section is enforced, once all version sections related to a document has been obtained, the responsibility will be clarified and the document can be used as forensic evidence once disaccord appears.

Until a version section has been finished, its digest value can be calculated and its ID can be generated, so only the version sections finished later can link to it. Thus, the circular linkage, which is troublesome in many scenarios, will be avoided automatically.

Generate version sections

New editors can be developed to support proposed document structure. However, a more economical way is to modify some current editors to achieve the same purpose. For example, the Eclipse JDT can be modified to use proposed document structure to manage source code. The enhanced editor will detect user's activities and creates operation records and links automatically

As mentioned, in proposed approach, each document is described by multiple linked version sections, and each version section corresponds to once modification. We call the latest version section as current version section. The method of generating version sections is list as following.

Once a developer log on the editor and create a new document, the editor will generate a new version section and set it as current version section

Once a developer log on the editor and open an old document to modify, the editor will generate a new version section and set it as current version section, and create an parent link pointing to the current version section of the old document

Once the developer copy some objects from document A which is also describes with proposed approach to current document, the editor will add a reference link to current version section, which points to the current version section of document A; Then the editor will add some "LCopy" operation records describing this copy operation. According to user's option, all the version sections describing document A may be copied to the current work space

Once the developer copy some objects from traditional document B to current document, the editor will wrap these objects to a new version section firstly, and prompt the importer to sign this version section. Then the editor will add a reference link to current version section, which points to the new wrapped version section, and add some "LCopy" operation records describing this copy operation. The new wrapped version sections will be kept in the current work space

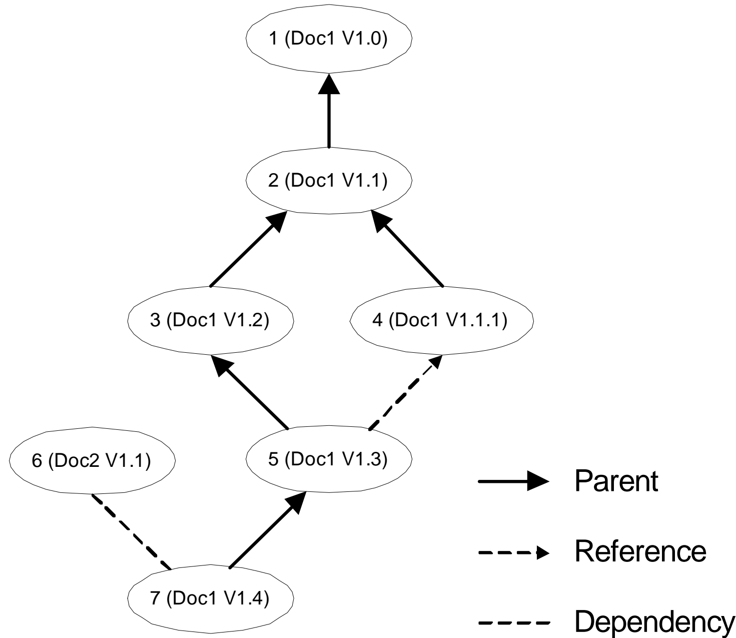
Once the developer made any modification, such as adding a line, modifying a line, corresponding operation records will be added to current version section

Once the developer finish this modification, he will sign the current version section, then the digest value of the signed version section will be calculated and set as its ID and the final version section will be kept in current work space

Since usually we don't care the detail operation sequence related an object by same developer during once modification, the operation records related an object in a version section can be merged to reduce the data size. For example, a LInsert operation and a LModify operation related to same object can be merged to a LInsert operation; a LDelete operation and a LRetrieve operation can be annihilated as a pair. In order to further reduce the data size, multiple

consecutive version sections that belong to same organization may be merged into one version section when delivering the open-source project to customer.

Generated according to these steps, the version sections that describe a document are linked together, and the topology is a directed acyclic graph (DAG). Different from a tree, some version sections may be linked by multiple nodes, so data redundancy can be avoided. Any document can be describes by such a graph, each node in the graph corresponds to once modification. Regarding any node in the graph as the root node, we can get a sub-graph, which corresponds to a certain history version of the current document. Therefore, when document is described with proposed approach, every history version of it can be retrieved. Figure 3 shows such a graph.



Parse version sections

Since only traditional source code documents (e.g. ".java" file, ".c" file, etc.) can be compiled and only content with this form can be read easily by user, when a document described by version sections is loaded, it will be converted to original content. The original content can be retrieved through processing the operations according to following steps:

From the latest version section, through going through all the links recursively, collect all the related version sections. For the version section located at remote space, load them to local space. The integrality of these version sections will be verified through calculate their digest value

From the latest version section, Find the leaf version section through the parent link and process content objects according to each operation record in following sections. If the version section being processed has a reference link, find the leaf version section through the parent link from the referred version section and execute the version sections in this branch firstly and copy the referred content objects to the document that refers others

After the last operation record in the latest version section has been processed, the processed content objects are just what we want

Since some version sections may be referred by multiple version sections, caching the content object set corresponding to them with a pool can enhance the converting speed.

Store version sections

Currently, there are 2 ways to store version sections. One way is like traditional document, all the version sections that describing one document are kept together in a file, with a tag indicating the latest version section (as shown in Table 1). This way uses file to store a document, facilitating transmission and share.

Another way uses a database to store all the version sections that belong to a project or an organization. Each version section is saved as a record, its ID is set as the primary key and its links are set as attributes. Each document described by some of these version sections also has an ID, which is just the ID of its latest version section. When the editor want to retrieve a document from the database, it will find its latest version section according to its document ID, and then get all the related version sections through going through all the links recursively. Due to frequent referring, many version sections may be shared by multiple documents in same project. Therefore, using database to store version sections can avoid many redundancies and thus reduce the total data size.

Applications

Document described by version sections contains more information than traditional document. Based on the information, many applications can be built.

Based on the provenance, applications can be built to show the operation history of every content object. Even if an object is copied from another document, the operation history in the original document can be also retrieved. This application may help project manager find the bug maker.

As mentioned above, if document is describes with version sections, every history version of it can be retrieved. Thus a version management system can be built easily based on proposed document structure. Unlike traditional version management system, the version information is bound to the content and kept locally. Therefore, it will not be lost if the source code is transmitted out of the project lifecycle. Furthermore, if a project is owned by one developer, to manage the versions, no centralized server will be needed, so the enhanced editor with a version management plug-in can be regarded as a light-weight version management system.

Through analyzing the provenance record, some statistics can be made automatically (e.g., to find whose codes are referred most frequently; whose codes are often corrected by senior developer; or which developers like to adopt open-source). According to these statistics, developers can be evaluated or recognized more easily.

Furthermore, if an interface is provided for the developer of each version section to configure the license items with a standard language, the compliance of whole document can be checked with an evaluator, which will help project manager clarify copyright issues.

The proposed document structure shown in this paper is designed for coding artifact because it is a widely reused artifact type. In the future, the mechanism can be easily extended to other artifacts. For example, if ".project" file is recorded with proposed document structure, this method will enable the version of whole project to be managed.

New delivering pattern

Usually, the agreement of many open-source projects may require the offering which integrates it not to only wrap it. However, how to define wrapping may be confused and thus this is a potential risk for developers.

If an open-source project is developed using proposed approach, each developer's contribution will be partitioned into different version sections. Even if a developer adopts some open-source codes and modified them, he may only delivery the version section that describes his modification but not the final combination. Once the customer received his modification, according to the link embedded in the version section, the version sections that describe original open-source will be found. So it is the customer's duty to get an agreement for these version sections. After all the version sections have been collected, the parser will help customer regenerate the final modified code. Thus, on the developer's angle, potential copyright-related risk has been avoided. Furthermore, proposed approach enables the original open-source provider to contact with end customer for the end customer has to ask original open-source provider for agreement, so the original open-source provider can get profit from end customer, thus an efficient billing model "by final copy" is enabled.

Summary

We have introduced a new document structure that records provenance information of a coding unit instead of its content and partitions each developer's contribution into different artifacts. The merit of this document structure can be summarized as following:

More original information has been kept, so based on this information, many analysis and statistics can be made to get valuable conclusions

Recording the operation history instead of content facilitates the clarification of each developer's contribution and responsibilities.

All related information is connected with auto-created links. For example, All the referred documents can be found through the reference links embedded in version sections

Since many common attributes can be extracted from the provenance records of many different document types, a general fundamental structure can be constructed for these types, which facilitates the processing and analysis by standard modules

If the policy enforce each version section to be signed by a warrantor before delivering, not only the version section owned by current developer, but also the inherited and referred version sections will all be warranted and thus can't be tampered. Thus, the responsibility of current developer is incontestable

EXPERIMENTS

Implementation

The proposed document structure is implemented in Ariadne, an enhanced Java code editor on Eclipse platform. For a signed-up user, the Ariadne Client dynamically detects and records his editing operations (e.g. add, delete, copy/paste, etc) with XML-based structure, and provides some views for user to retrieve operation history of a certain coding line or to show which lines has been touched by a certain developer. Reference statistics can be performed when necessary. It also provides a mechanism of local version management. The technical detail of Ariadne has been introduced in [11].

The implementation of Ariadne Client follows the Eclipse plug-in developing rules [10]. It is composed of several functional plug-ins, thus Ariadne is indeed an extension component of the traditional Eclipse platform.

A consequent headache is how to deal with code copy. In Ariadne, when some coding objects are copied from document A to document B, the client will put each version section that belongs to document A into clipboard, and then attaching a temporary version section that contains temporary operation(s) indicating the copy operation to the clipboard; When Ariadne detects a pasting operation, if the content in clipboard is a well-structured version section set generated by last step, it will move each version section from clipboard to work space, add a reference link to the referred version section and then converts the temporary operation(s) into formal operation(s) indicating the ID and position in both the source file and the target file.

Based on Ariadne, we test the storage efficiency and data processing efficiency of proposed document structure. Since open-source projects provides a readily accessible and actual data set, we use some coding artifacts in the open-source search engine project "lucene" as the test set for these experiments. The tests were deployed on an Intel Pentium 1.6G machine with a 1024M memory. The machine ran Window XP SP2 and the Ariadne were based on Eclipse Version 3.1.2 with the J2SE JDK 1.4.2 from SUN.

Storage efficiency

The test set for storage efficiency contains 49 ".java" file, which are totally 253925 Bytes. When they are converted into proposed XML-based document structure, suppose each artifact is described by only one version section (It means each artifact is touched only by one person), the total size of destination files is 964849 Bytes, which is the 3.8 times of the original files. Since the network bandwidth and the cost of hard driver space are not serious concerns currently, this rate is acceptable under most circumstances. Some methods can be adopted to reduce the destination file size. Besides the approaches mentioned in last section, we can reduce the data size though encoding the metadata file with compact representation.

Data processing efficiency

The test set for storage efficiency contains 76 coding artifacts, which are totally 393055 Bytes. When they are parsed from version sections into original ".java" files, the total time consumption is 16.204s, the average time consumption for each KBytes is 0.041s, and the average time consumption for each artifact is 0.213s, which is imperceptible for most users. If some version sections have signatures that need to be verified, the verification may also consume some time. In order to optimize the operation efficiency, we have introduced some methods including the content caching with a pool mentioned in last section.

CONCLUSION

In this paper we have proposed a XML-based document structure that records provenance history and enables trusted partition of each developer or company's contribution. Thus potential copyright issues can be avoided. Based on this structure, many applications can be built to track originality information or manage versions. We have developed Ariadne based on Eclipse JDT to justify our approach. The experimental result shows the data processing efficiency and storage efficiency are both in an acceptable range.

REFERENCES

1. Java Coding Standard Version 2.0. http://www.cs.rit.edu/~f2y-grd/java-coding-standard.html
2. CVS - Concurrent Versions System. http://www.nongnu.org/cvs/
3. Collaborative Editing of Electronic Documents. http://training.cssd.pitt.edu/downloads/WordTrackChanges.doc
4. P. Buneman, S. Khanna, and W. Tan. Why and Where: A Characterization of Data Provenance. In International Conference on Database Theory, London, UK, Jan. 2001.
5. Margo Seltzer, Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Jonathan Ledlie. Provenance-Aware Storage Systems. Harvard University Computer Science Technical Report TR-18-05, July 2005

6. Committee on Digital Archiving and the National Archives and Records Administration. R. Sproull and J. Eisenberg, eds. Building an Electronic Records Archive at the National Archives and Records Administration: Recommendations for a Long-Term Strategy. The National Academies Press,Washington, D.C., 2005.
7. Y. L. Simmhan, B. Plale, and D. Gannon. A Survey of Data Provenance Techniques. Technical Report TR-618: Computer Science Department, Indiana University, 2005.
8. P Buneman, S Khanna, W C Tan. Data Provenance: Some Basic Issues. Proc of the Foundations of Software Technology and Theoretical Computer Science Conf (FSTTCS), 2000.87~93
9. Wang-Chiew Tan. Research Problems in Data Provenance. IEEE Data Engineering Bulletin. 2004. electronic version
10. Help Document of Eclipse SDK (http://help.eclipse.org/help31/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/arch.htm)
11. Lin Luo, Da Ming Hao, Zhong Tian, Ya Bin Dang, Bo Hou, Peter Malkin, Shun Xiang Yang. Ariadne - Facilitate Originality Tracking for Artifact Reuse Compliance. Submitted to IBM Systems Journal, Mar. 2006.