Ariadne: An Eclipse-based system for tracking the originality of source code

L. Luo
D. M. Hao
Z. Tian
Y. B. Dang
B. Hou
P. Malkin
S. X. Yang

In this paper we introduce Ariadne, an Eclipse-based system for tracking the originality of source code in collaborative software development environments in which software reuse is a common practice. We describe its architecture within the Eclipse framework, the originality metadata of which it keeps track, and the history clue—the data structure used to implement the tracking mechanism. We also discuss the implementation of the Ariadne client, the main component of the system, and show how digital signatures are used to validate the integrity of the metadata-handling process. We demonstrate the functions of Ariadne in two typical scenarios: tracking of software bugs and generating originality claims for Certificate of Originality reports. Although our Eclipse-based prototype is designed to handle Java source code, our approach can be extended to other kinds of artifacts.

INTRODUCTION

The reuse of software artifacts plays an important part in improving the quality of software and reducing development costs. To save duplicate efforts during software development, software development organizations harvest their existing artifacts, such as source code, requirement documents, and design model files, as component assets and provide support for their reuse. In order to minimize the business risks resulting from the illegal use of software artifacts, the originality information associated with these software artifacts has to be carefully and reliably tracked. We consider originality information all information that pertains to questions such as: "Who authored this snippet of source code?" "Who has ownership rights for this software artifact?" "What contractual restrictions apply to this software artifact? "Does this snippet of source code involve another party's patent?" and "Is this snippet of source code open-source software?"

A typical example of the problems that could arise when originality information is not monitored is code contamination by open-source software.[1] When members of a software development team share their code, they may inadvertently embed open-source code into a commercial software product. To control the code contamination risk,

some companies choose the "closed source" strategy, which rules out the use of any open-source or third-party code during software development. Other companies, including IBM, require that every development project for software delivered to customers has to undergo a Certificate of Originality (CoO) process whose purpose is to determine the originality information associated with all software artifacts in the product. The experience shows, however, that due to originality information that is often incomplete or uncertain, this process is often costly (in terms of manpower) and unreliable. Open-source communities also need to manage code originality information. In this environment, developers who contribute source code to the community also face the potential risk of lawsuits if the code infringes on certain patents.[2]

Aside from its role in intellectual property (IP) issues, originality information is also important for managing aspects of the software-development life cycle, such as detecting code cloning, tracing the source of software bugs, and evaluating asset reuse.

If we consider other software artifacts, such as documents, design models, images, and audio and video clips, we find the same imperative to maintain originality-related information about each artifact. Only based on such information can we determine whether the artifact contains content whose access is either prohibited (e.g., illegally copied music) or restricted (e.g., using the artifact after one's access rights have expired).

There are two basic approaches to managing the IP issues related to originality information associated with source code. In the first approach, we search all repositories of source code available for items that are similar to the segment of source code under study. We refer to this approach as *after-the-fact retrieval*. In the second approach, we enhance the tools used for developing source code and record all events that are significant from the IP perspective. Moreover, the historical record of all IP-significant events becomes the metadata associated with the source code. We refer to this approach as *in-process recording,* or *tracking*.

After-the-fact retrieval approach

Techniques that have been developed for detecting code cloning and plagiarism can be used for retrieving originality information associated with a given segment of source code.[3–8] Krugle** from Krugle, Inc. and Google** Code Search from Google, Inc. are commercial tools that detect code cloning.[5,6] By adapting traditional search algorithms to the source-code domain, they seek to locate keywords or source-code segments in known open-source repositories, such as SourceForge**. Their focus, however, is limited to reusing existing open-source software, not managing IP concerns in software development.

The techniques for code clone detection and plagiarism detection are based on the concept of calculating fingerprints (checksum) for both the source file and the target file and then comparing them.[3,4] The checksum calculation algorithms are designed to be robust enough to resist potential code obfuscation or other changes. Products ProtexIP**/ development from Black Duck Software, Inc. and IP Amplifier from Palamida, Inc. combine techniques for code cloning and plagiarism detection with a source-code fingerprint registry, functions for source-code license management and IP policy configuration and compliance reporting, and thus provide a complete solution for managing IP concerns.[7,8]

The advantage of the after-the-fact approach is that it does not depend on the software development process. The code fingerprint embodies the intrinsic properties of a code segment and is the vehicle used to locate the code segment in the code fingerprint registry. The after-the-fact approach is especially useful for source code copied or inherited from legacy code. Access to the code fingerprint repository is required; otherwise, the originality information is not available. There are, however, limitations to this approach. The performance and coverage of the fingerprint comparison are highly dependent on the project size. In addition, certain levels of code modification or transformation may diminish or even destroy the retrieval accuracy, as in the case when source code is compiled into binary code. Finally, the after-the-fact approach is not applicable to certain kinds of originality information, such as source code extracted from a textbook, an algorithm whose use might infringe on a patent, or source code under contractual constraints.

In-process recording approach

Currently there are no effective tools for fully tracking originality information throughout the

software life cycle. Although comments embedded in source code can be used to record originality information, their creation and maintenance rely on the diligence of developers, and the arbitrary nature of their content and format makes them unsuitable for automated processing.

Software configuration management (SCM) tools (also known as version control or source control tools), such as ClearCase* and ClearQuest* from IBM Rational and the open source CVS (Concurrent Versioning System), can automatically record author and revision information during check-in/check-out events. Their design, however, is geared toward project-centric software development, and when a software component is reused by a new project, the historical record of the previous project may be dropped. Furthermore, SCM tools only record originality information at check-in and check-out, and some originality information, such as whether the checked-in component incorporates code copied from another module, is not available.

Some word-processing applications, such as Microsoft Word, have a tracking function that records the document editing history. However, just like the SCM mechanism, it is mainly used for version control. When users copy-and-paste a segment of text from another document, the originality information of the source document is lost.

In order to find out how source code evolves during the software-development life cycle, Kim et al. investigated the situations in which developers perform copy-and-paste.[9] To carry out their research, they constructed an Eclipse**-based IDE (integrated development environment) client that logs all copy-and-paste events.[10] Although their purpose was not tracking originality information, the tool can be regarded as an in-process recording tool.

The in-process approach requires the handling of additional information, namely originality information, which we refer to as metadata. To ensure metadata integrity and non-repudiation during software development, digital signature technology can be used, such as calculating the checksum (digest) of the content and encrypting it with the developer's private key. Even if only one bit of the content is changed, the signature verification process will detect that the metadata or the corresponding source codes have been tampered with.[11]

Ariadne

In this paper we describe Ariadne, an Eclipse-based system for tracking the originality information of source code in collaborative software development. We named our system Ariadne, after the character in Greek mythology. Ariadne, the daughter of king Minos of Crete, who gave Theseus the ball of yarn (the clue) with which he was able to escape the labyrinth after killing the Minotaur (Ariadne's clue is thus a metaphor for traceability). Ariadne is based on Eclipse and provides an enhanced IDE that manages originality information during software development. The originality information is automatically generated whenever possible; in other cases, such as when legacy code is imported, the developer is prompted to enter the originality data. The integrity of the originality information is ensured through the use of digital signature technology.[11] Although our prototype is designed to handle source code, our approach can be extended to other software artifacts.

The rest of the paper is organized as follows. In the next section we describe Ariadne's architecture within the Eclipse framework, the originality metadata of which it keeps track, and the history clue—the data structure used to implement the tracking mechanism. We also discuss the implementation of the Ariadne client, the main component of the system, and show how digital signatures are used to validate the integrity of the metadata-handling process. In the following section we show experimental results. We demonstrate the functions of Ariadne in two typical scenarios: tracking of software bugs and generating CoO reports. We also discuss preliminary performance results of the Ariadne prototype. In the last section we summarize our results and discuss future work.

DESIGN AND IMPLEMENTATION

In this section we describe the overall architecture of Ariadne, discuss the history-clue data structure, and describe the structure and some details of the implementation of the Ariadne client.

Overall architecture

As illustrated in *Figure 1*, Ariadne consists of the Ariadne client, the Ariadne compliance server, and some peripheral components. The peripheral components include a support services component, which consists of an identity server and a certificate authority, and an artifact repository. The Ariadne

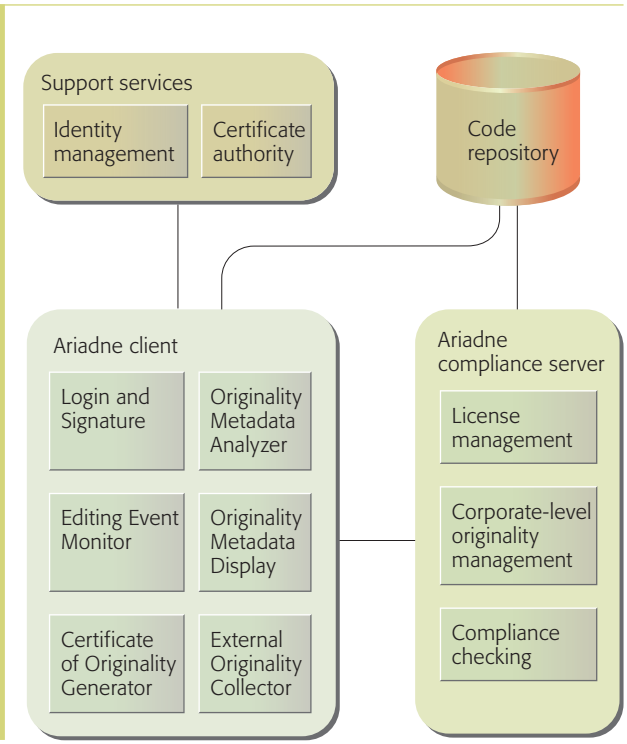

Figure 1
Ariadne architecture

client implements an IDE, whereas the compliance server covers license management, corporate-level originality data management, and compliance checking according to company policies.

In the current implementation of the Ariadne client, its originality-tracking capability is demonstrated on Java** source code (for other kinds of artifacts, similar modifications to the corresponding artifact editors are required). The Ariadne client dynamically tracks the developer's editing events (such as insert, delete, and copy-and-paste) in the Ariadne Java editor, identifies the currently edited artifact and the associated originality-information metadata, and updates the originality-information metadata. It also supports the creation of the CoO report.

As discussed in the Introduction, originality information is of two types: editing history and IP-related information. The editing history can be automatically generated by client monitoring. The types of editing events we track include insert a line, delete a line, modify a line, and copy-and-paste an object. IP-related information includes open-source claims, applicable patents, licensing terms, and contractual requirements. When it is first encountered, this information is entered by the developer through manual input, possibly after searching through source-code repositories. Upon reuse, the information is automatically combined with editing history information.

The originality metadata could be managed in several ways. It could be embedded into the source code as comments, stored in a separate metadata file, or centrally managed by the Ariadne compliance server. In our implementation the metadata is handled as a separate metadata file whose name is the same as that of the source Java file (file of type `*.java`) but with the special postfix `*.orimeta` (short for "originality metadata").

The key logical functional components of the Ariadne client are shown in Figure 1. The *Login and Signature* module enables the Ariadne user to log in and then verifies the user's identity (through password authentication). Following authentication, the source code created by the developer, together with the associated metadata, are digitally signed by using the developer's private key whenever the file is saved. For verification at the project level, the metadata are transferred to the Ariadne compliance server and submitted to the project leader to be signed again by using the team key. The Login and Signature module also performs an integrity check on the metadata when necessary, for example, when the source-code file is first opened.

The *Editing Event Monitor* inherits the monitoring capability of the Eclipse development environment. It monitors in real time two kinds of editing events: basic operations, such as insert, delete, modify, and replace, and cross-file operations, such as copy-and-paste. These editing events are mapped to the corresponding line-oriented Ariadne basic operations. When copy-and-paste operations are monitored, the *Originality Metadata Analyzer* is triggered to extract the originality metadata associated with the copied code segment from the second originality metadata file.

The Originality Metadata Analyzer records the originality information within an XML-based structure (`.orimeta file`). When a code snippet is copied and pasted from a source file to a target file, the analyzer extracts the associated metadata to the clipboard and then appends it to the target metadata

file. When the developer has to handle IP-related metadata, the analyzer invokes the *External Originality Collector*.

The function of the External Originality Collector complements the function of the Originality Metadata Analyzer. It collects additional IP information when the origin of the source code is unknown or the information available is inadequate. It enables developers to manually enter their own description of the metadata for code snippets without originality metadata that originated in legacy code. To facilitate the manual input, the collector can link to an open-source code registry in order to retrieve after-the-fact originality information.

The *Originality Metadata Display* is a user interface (UI) module that displays the originality metadata to the developer. The developer selects a code snippet, and the associated originality information is displayed for viewing. The *CoO Generator* compiles originality metadata of all the artifacts in a project into the CoO report for the project.

Figure 2 illustrates a typical scenario for processing originality metadata during an editing session in the Ariadne Java editor. The editing session includes a copy-and-paste operation in which a snippet from file `Source.java` is copied into file `Target.java`. The steps in the editing session are on the left; the steps on the right are the corresponding metadata processing steps. When file `Target.java` is loaded, the Originality Metadata Analyzer checks the presence of the file `Target.orimeta`. If the file does not exist, it is created and initialized. Editing events are monitored, and newly created originality information is appended to the existing metadata. For copy-and-paste operations, the corresponding metadata snippets are automatically inserted into the target file. For most editing operations the metadata processing is transparent to the developer and does not affect the normal editing experience. Only when the developer has to handle IP-related information is it necessary to select proper IP claims for the source code created.

At the end of an editing session, when the project is saved, the originality metadata is recorded. Depending on the company policy, the developer may be required to sign his work in order to ensure the integrity and the non-repudiation property of the product. For example, if the code is associated with open-source claims, then the developer's signature is viewed as a declaration of compliance with the company policies.

All clients send originality-related information to the compliance server for analysis. When a particular artifact is reused from other projects (possibly following several additional rounds of reuse), the corresponding originality metadata is collected by the Ariadne compliance server. Companies can use the collected information to audit compliance with the official asset-reuse policies.

It is often the case that software development organizations have implicit or explicit regulations such as "submitted assets should have originality information associated with them, and the user should not delete such information during asset reuse." When lack of originality metadata is detected by Ariadne, the corresponding asset is marked "suspect". The creation of fraudulent metadata can be detected through audit procedures that involve search-and-compare algorithms similar to the after-the-fact approach. In general, company employees do not intentionally remove or forge metadata because of the risks involved in adopting suspect code. In this case, the main function of Ariadne is to facilitate and automate the handling of the originality information.

History clue

History clue is the Ariadne data structure for managing originality metadata consistently and reliably. Because any number of developers may be involved in editing a source-code file, tracking authorship and ownership throughout the software life cycle presents a technical challenge.

Figure 3 provides a conceptual view of the history-clue data structure as a strip of paper of unbounded length with width-wise folds, whose content can be displayed or hidden by opening or closing the folds. With the folds closed, the visible content of the paper is simply the current version of the source code. Each fold contains an item of the editing history of this code, such as a deleted line as shown in Figure 3. The contents of the history clue can only increase over time, because nothing is ever erased from it. The screen capture was taken when developer Luolin was about to save the file and end the session. At that point Ariadne automatically digitally signed all the lines that were affected during the session (these lines are highlighted in the figure).

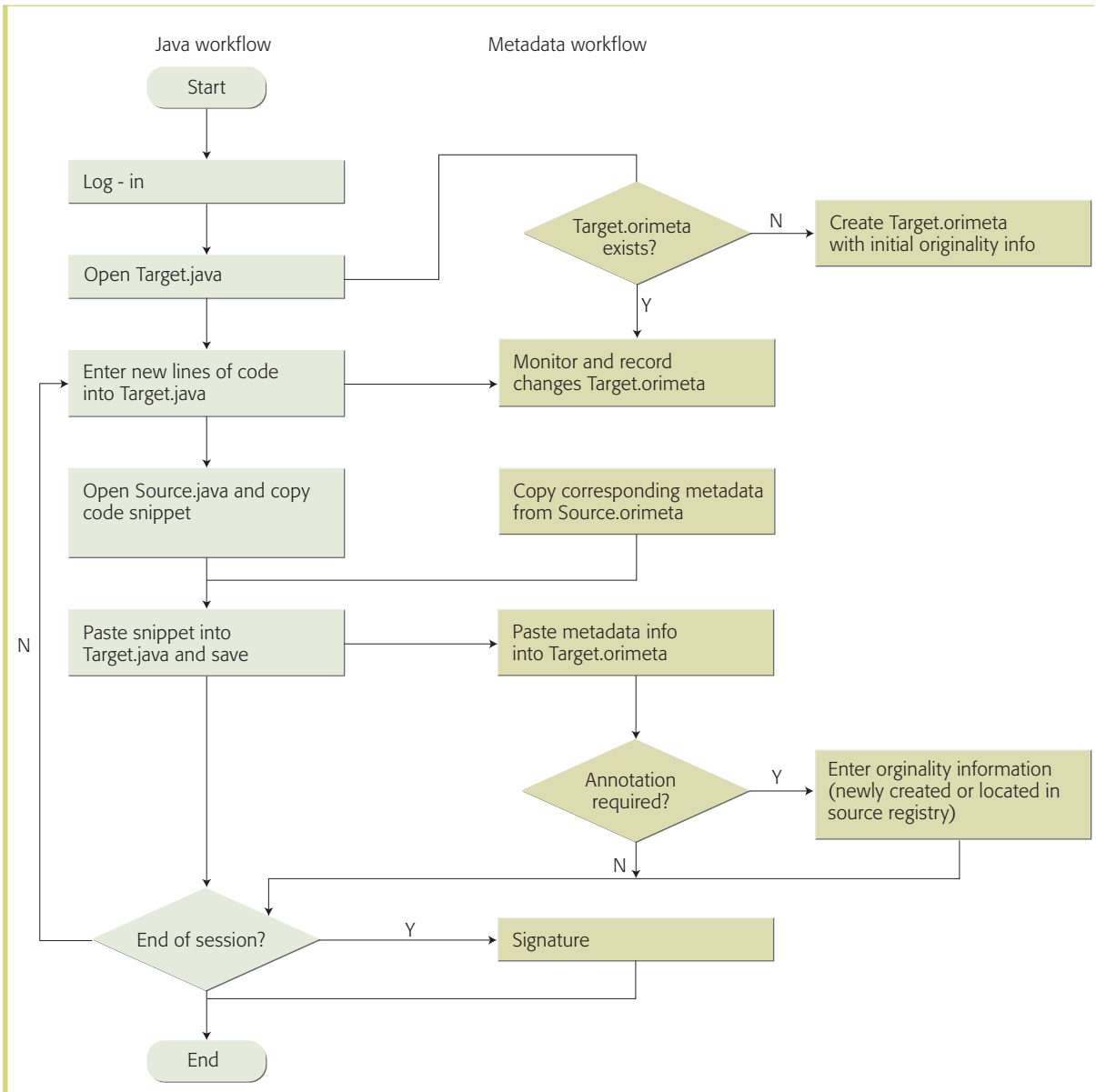

Figure 2
Java and metadata workflows: A typical scenario

Digital signatures validate the integrity of the metadata-handling process.[11] For each editing session, the digest of the source code and the corresponding metadata are calculated and signed by using the developer's private key and then stored in the metadata file. The signature can be verified at any time by calculating the digest and comparing it with the stored digest.

In order to record the editing history in the history clue, possibly including actions by multiple developers, all the editing events are mapped into a number of line-based basic operations. The simplest ones are *add a line* (a line of code is inserted) and *delete a line* (a line of code is deleted); in both of these cases a line is added to the history clue. The operation *modify a line* takes place when some words on a line are modified, and it can be mapped to two simpler operations: delete the old line and add the new line; in this case, two lines are added to the history clue. The operation *paste an object* is applied on an object, which may consist of a line

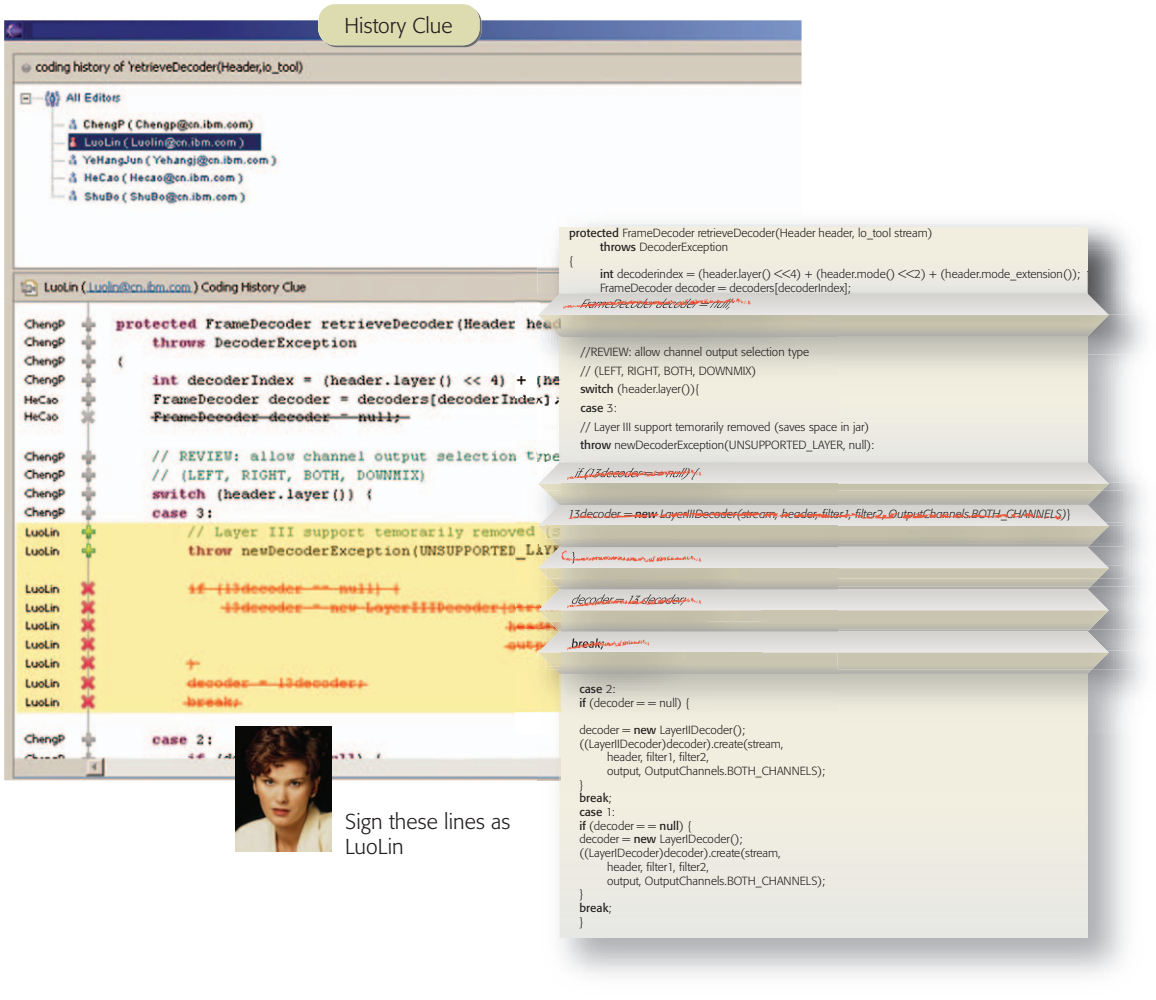

Figure 3
Recording code changes in the history clue: A conceptual view

fragment, a complete line, or a set of consecutive lines, so that the first and last lines may be line fragments. This operation can also be mapped to several *add a line* operations. We should point out that for efficiency we have also developed heuristic algorithms that expand the concept of a line to include semantically connected snippets of code. For example, a loop can be viewed as a single "line".

If all editing events can be viewed as one or more *add a line* operations, then the metadata file can only increase. Each line is associated with a unique developer whose private key is used to digitally sign the contents at the end of the session. In Figure 3 the developer's name appears on the left side of the frame labeled Coding History Clue. Note that a line may involve two developers—the author who creates the line and the editor who imports it (e.g., by pasting it) into the current file. Both of these names are recorded in the metadata. In the case of open-source claims, the original author is responsible for IP-related issues; for software-life-cycle problems, such as tracking software bugs, both the author and the editor are held responsible.

Figure 4 shows the structure of the originality metadata and the associated source code. The metadata includes *delta lines*, the *line number lookup table*, the *editing history*, and the *IP data*. The metadata file is formatted as an XML document. XML security tools[12] are used to map the file to a canonical form and to perform the digital-signing function.

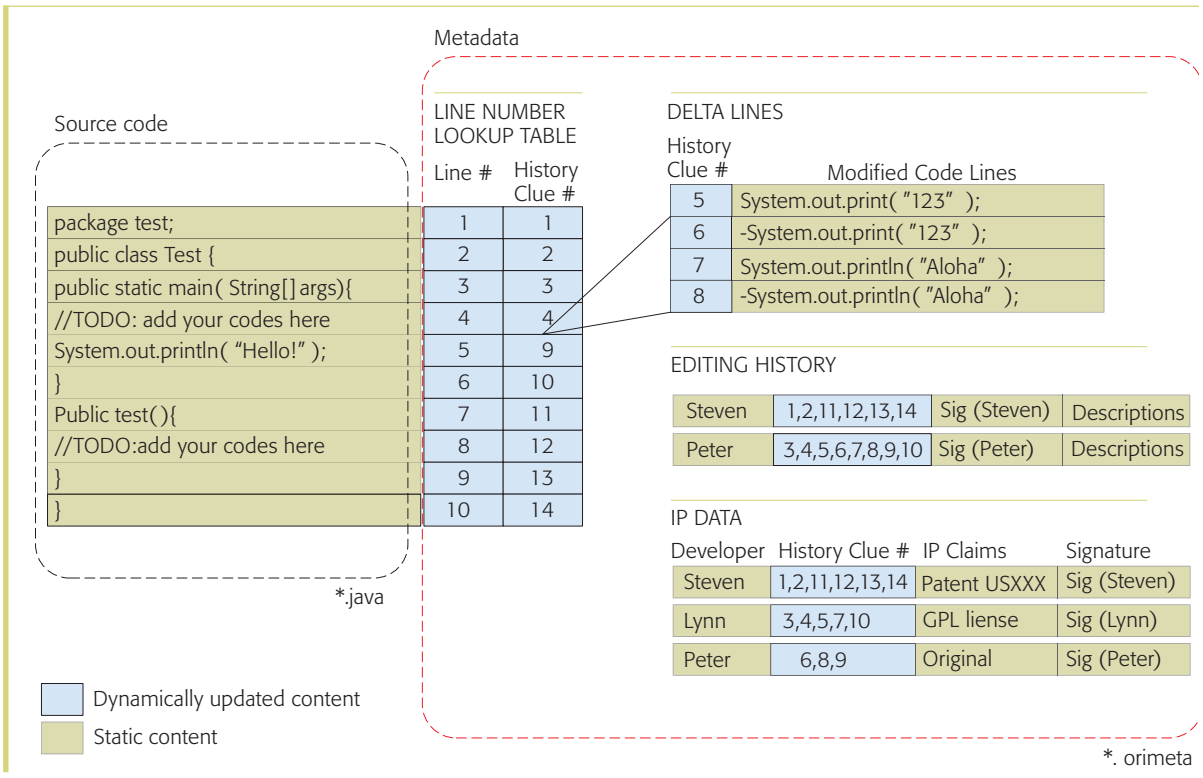

Figure 4
The originality metadata and their relation to the corresponding source code

IP data contains IP claims, such as open-source or ownership-rights claims, for specific lines. The first column contains the author's name, the person responsible for assigning the initial value of the IP-related data. Figure 4 shows Steven, Lynn and Peter as authors—their names appear either in the editing history or in IP data. Sometimes the author creates the content, such as Peter in Figure 4, where the line for Peter in IP data contains "original" in the column IP claims. IP claims are context-independent properties of the code snippets. An author is required to digitally sign the IP data claim and the associated code snippet, and the signature is stored in the IP data.

Editing history contains information on the changes made to the source code. The first column contains the name of the editor, the person responsible for the change. In the example shown in Figure 4, only Steven and Peter are editors. Although the file includes source code authored by Lynn, Lynn is not considered an editor because her code is imported by Peter. Unlike IP data, the editing-history information may be context-dependent. For example, a bug may be caused not by the code snippet itself but by its position in the source-code file. An editor is required to sign the editing history for accountability in the artifact-development life-cycle governance scenarios. The digital signature should cover both his or her input and the semantically relevant context codes in the history clue. In some cases, the context is the entire history clue, as shown in Figure 4.

If developer B imports developer A's code snippet into a work file and if developer C further imports B's source code, which includes A's snippet, then the history clue for this snippet is labeled with author A and editor C. The delta lines in Figure 4 represent the difference between the history clue and the Java file, and correspond to the folded sections of the paper strip in Figure 3. The line-number lookup table holds the mapping of source-code line numbers to history-clue line numbers and is dynamically updated during editing. For example, if a delta line is to be inserted between history-clue line 2 and line 3, the history-clue numbers in the line-number look-up table are automatically updated.

We consider now the ways in which importing a code snippet from a source-code file into the work file affects the author-editor information in the metadata.

1. Peter copies a code snippet from file `Source.java`, which he authored, and pastes it into work file `Target.java`. The copied snippet in `Target.java` is labeled with author Peter and editor Peter.
2. Peter copies a code snippet from file `Source.java`, which was authored by Lynn, and pastes it into work file `Target.java`. The copied snippet is labeled with author Lynn and editor Peter.
3. Steven copies a code snippet that has no originality metadata (such as legacy code or code external to Ariadne) and pastes it into the work file. The copied snippet is labeled "Steven" as author and editor, and the IP data contains the IP-related claims entered by Steven.

For signature purposes, the unit of code used should be considered. When a code snippet is to be copied, the smallest signed block that includes the code snippet has to be copied as a whole in order to enable the signature verification that follows. The granularity also depends on the extent of the source code associated with a claim. The entire source-code file can be signed as a single unit, although storage efficiency may deteriorate if the file involves too many codes snippets from other sources. Another possible approach is to partition the source code, either uniformly in smaller blocks or based on semantics (e.g., lines associated with an entire function).

Ariadne client

The Ariadne client is based on Eclipse Version 3.1.x. Eclipse is an open-source platform-independent software framework in which various software tools for application development can be integrated as plug-ins.[10] The implementation of functional modules such as the Login and Signature module, the CoO Generator, and the External Originality Collector module is straightforward and further details are left out. Instead, we focus on the modules that manipulate metadata; Editing Event Monitor, Originality Metadata Analyzer, and Originality Metadata Display; and describe the way we extend the Eclipse Java IDE in order to support the interplay between Java files and originality metadata files.

Figure 5 illustrates the Ariadne client architecture. The Ariadne Developer Environment (ADE) is built on the Basic Eclipse Platform and parallels the following Eclipse components: Java Development Tools (JDT) and Plug-in Developer Environment (PDE). These components streamline the development of plug-ins and extensions. As shown in Figure 5, ADE has three main components—Ariadne Core, Ariadne JDT, and Ariadne UI.

Ariadne Core interprets the data model of the originality metadata, ensures the integrity and validity of the data, ensures that the user actions are reflected in the originality metadata, and provides APIs for the ADE and Ariadne UI components. The class and the associated methods that implement the metadata model are generated by the Eclipse Modeling Framework, which is a Java-framework and code-generation facility for building tools and other applications based on a structured model.[13] Metadata objects are held in the Metadata Pool component. Metadocument Analyzer wraps the originality metadata and provides methods for handling metadata to be used by ADE and by the Ariadne UI components. The Metafile Generator component initializes the metadata file for each source-code file.

Ariadne JDT is an extension of the JDT that includes support for the Ariadne Core functions. Because it is based on the JDT, Ariadne JDT has the same plug-in interfaces and capabilities to manipulate Java code. Besides monitoring editing events in the Java editor, Ariadne JDT also processes the metadata object retrieved from the metadata file and establishes the links between the object and the source file. Because Ariadne JDT is created by modifying only two classes in JDT (which deal with the interplay between Java files and metadata files), it is relatively easy to keep Ariadne JDT updated whenever a new version of JDT is released. In Figure 5, the Ariadne JDT components that are based on JDT are blue.

Ariadne UI implements several functional views, including author view, IP data view, and history view. The architecture is extensible and can accommodate additional views, possibly from third-party vendors.

Figure 6 illustrates the Ariadne JDT monitoring of editing events. As the figure shows, Ariadne JDT event monitoring is an extension of JDT event monitoring. The item labeled JDT Document is

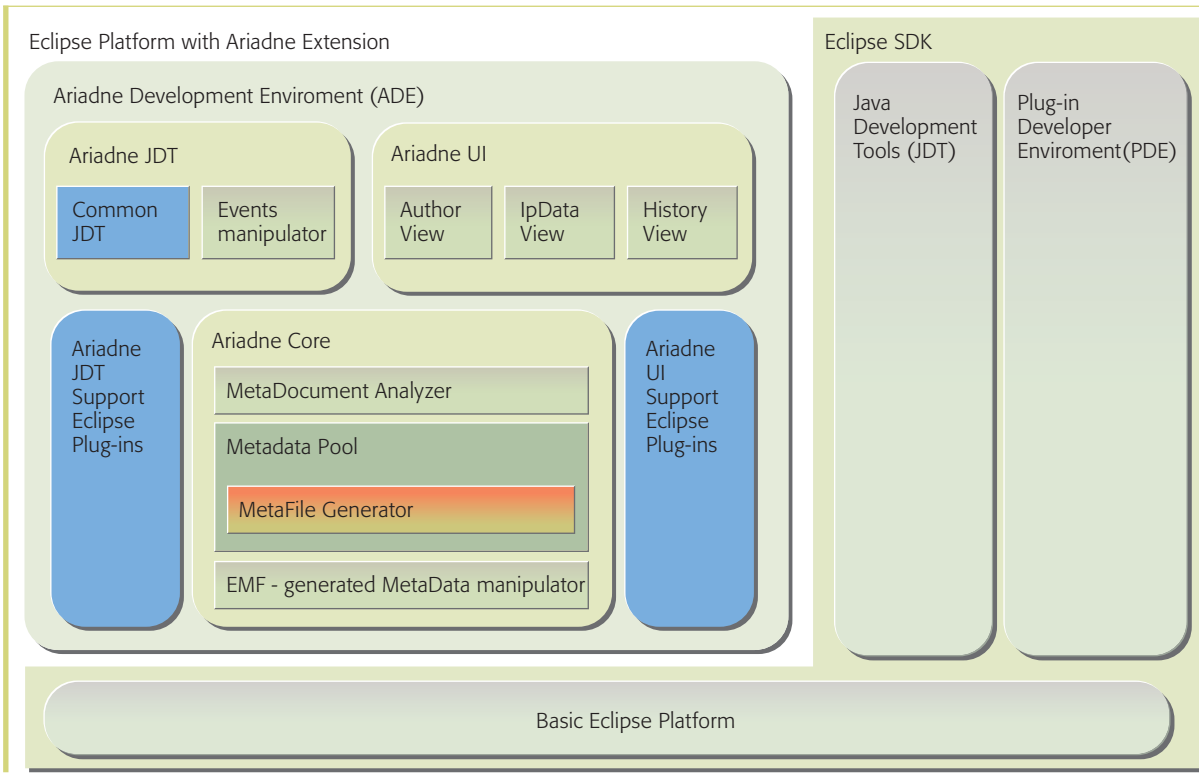

Figure 5
Ariadne client architecture

being monitored by any number of JDT observers and one Ariadne JDT observer (the observers are event driven). Every editing operation performed by the user triggers a JDT-document change event, which is broadcasted to all the registered JDT observers, including the Ariadne observer. The metadocument is then changed accordingly, and the metadocument change events (corresponding to changes to metadata) are broadcasted to any number of meta-observers.

To enable interoperability with other tools, when a developer copies source code in Ariadne JDT, the event-driven Ariadne JDT observer copies the related metadata to a clipboard section separate from the content section. Thus, if the developer pastes code within Ariadne JDT, the corresponding metadata is recognized, and a metadocument change is generated, whereas in standard editors only the source code is displayed.

Because the history clue is a monotonically increasing data structure, the implementation is optimized for storage efficiency. For example, if several consecutive modifications of the source code are made by the same developer, these modifications are aggregated into a single equivalent modification. In addition, the IP data claims that occur frequently are stored in a central repository on the Ariadne compliance server and only their Uniform Resource Identifiers (URIs) are recorded in the metadata file.

In our implementation we also took into consideration the efficiency of metadata loading (from peripheral storage to main memory). Because each source-code file is associated with a metadata file, large amounts of metadata are frequently accessed. A cache-like storage pool mechanism is used to manage the metadata loaded in main memory. The mechanism identifies the metadata objects that are not referenced for a certain period of time and removes them from memory to peripheral storage.

EXPERIMENTAL RESULTS

In this section we first demonstrate Ariadne's functionality in two typical scenarios: tracking of software bugs and generating CoO reports. Then we

discuss preliminary performance results of the Ariadne prototype.

Figure 7 shows a screen capture of an Ariadne editing session. The frame labeled Author View on the right side of the window contains information about the developers associated with the source-code module: David, the current user, Alice, and Tom. The "tool-tip" mechanism (the window that appears by "mousing" over an area of the screen) in the middle of the screen (the editor view) shows the editing history for the highlighted lines. The IP Data view at the bottom of the window contains open-source-related IP claims and applicable third-party legal constraints (under the DRM label; DRM stands for Digital Rights Management) for the code snippets.

Tracking of software bugs

We consider here a scenario involving the tracking of software bugs. A software module authored by David includes code that David copied during an Ariadne editing session from code authored by Alice. In turn, Alice's source code contains code that Alice copied in a previous Ariadne session from code authored by Tom. Thus, the software module contains code authored by Alice, Tom, and David. Some time after the application containing the software module is deployed, David discovers a software bug in lines 15 through 17. During an Ariadne session he opens the source-code file, selects these lines, and retrieves the editing history by "mousing" over these lines. The resulting tool-tip mechanism (see Figure 7) shows who may be responsible for the bug. Although tracing the responsibility for that bug to a single developer is not trivial, Ariadne's editing history facilitates that process. Furthermore, because the local originality information can be sent to the Ariadne compliance server, the server can identify all software components that contain the bug by virtue of the problem code having been propagated to other software modules. Aside from tracking of software bugs, Ariadne can also be used in the related application of measuring the effectiveness of software reuse.

Experimental data from early users of Ariadne helped us improve the UI design. For example, we discovered that developers prefer a simple display that does not include editing history. As a result, we have adopted a design of the UI that exhibits the editing history information only when needed by using the tool-tip mechanism illustrated in Figure 7.

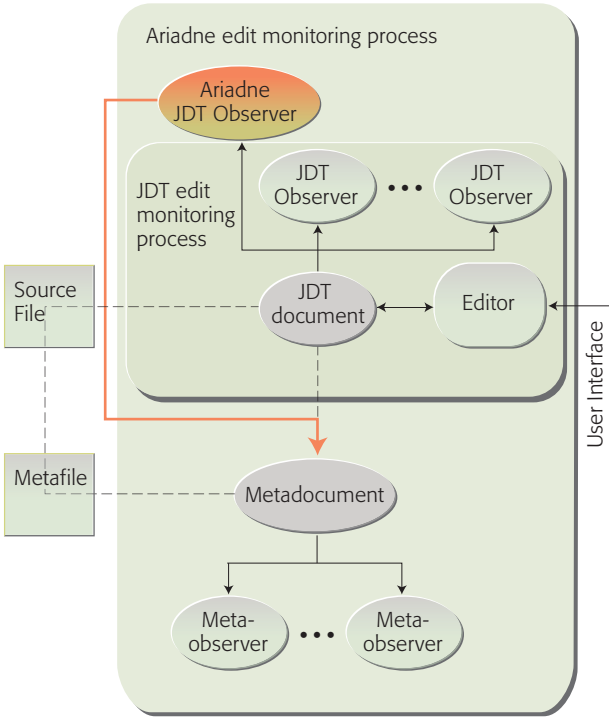

Figure 6
Ariadne event-monitoring process

Generating CoO reports

In the first stage of a customer-facing project, a demo prototype is to be built as soon as possible. Developer Alice embeds open-source code in one of the components of this prototype. She annotates the open-source information as shown in the IP Data view in Figure 7. During another Ariadne session, David makes further use of the source code, copies code authored by Alice together with the associated metadata, and views the relevant IP data in his IP Data view. Although the development of the prototype may involve more developers and many such instances of code reuse, the CoO generation tool will alert the team to inappropriate uses of open-source code.

The CoO report includes multiple sections, two of which are targeted by Ariadne: names of developers and the origin of the source code (we refer to the latter as "origin-of-source-code report"). *Figure 8* shows a sample origin-of-source-code report. In this report, three types of information are shown under "Category": open source, external patents, and under exclusive contract (the last label indicates whether

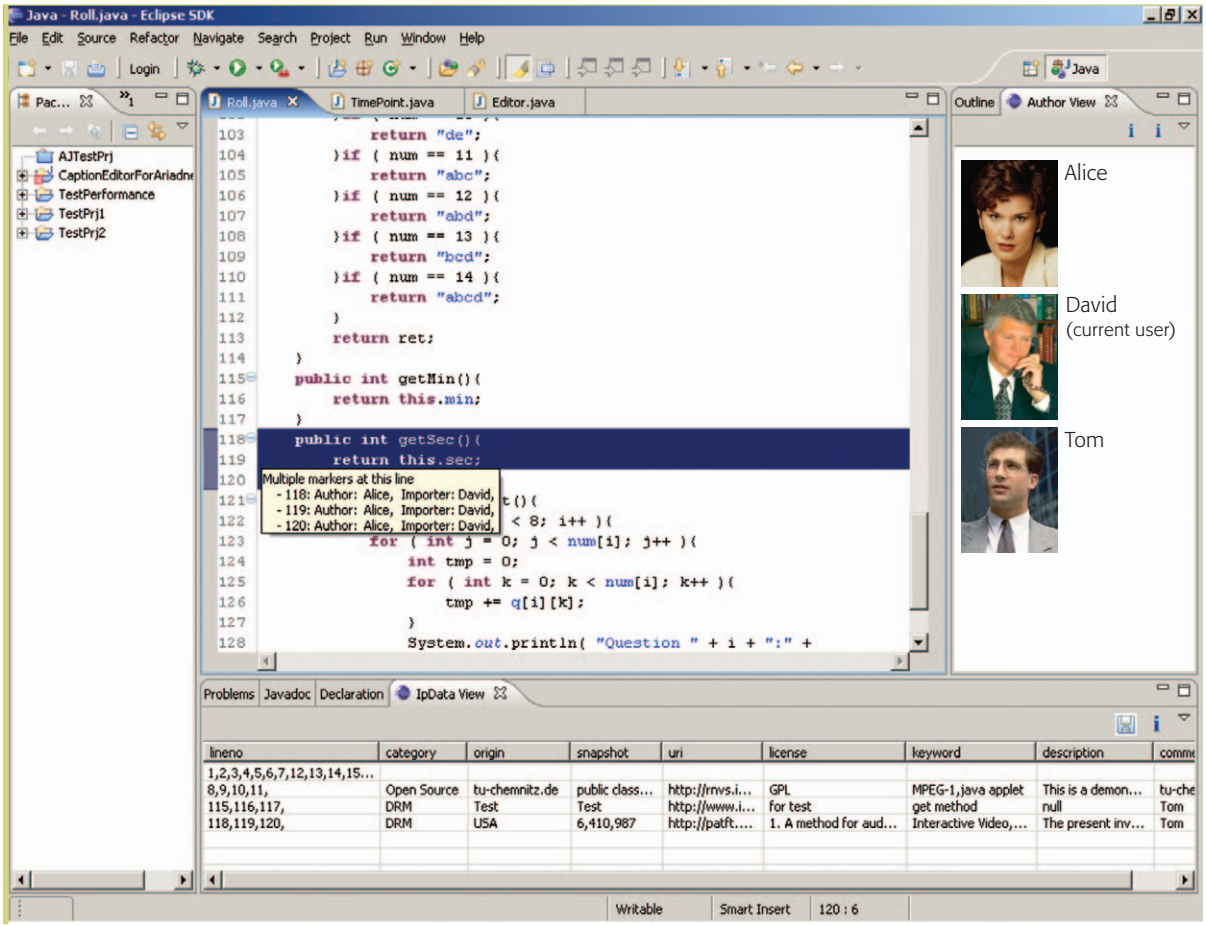

Figure 7
A screen capture of an Ariadne editing session

the use of the code is limited to a single customer). Under the label "open source" are shown three types of open-source license; the hyperlinks lead to details of these licenses and the associated source-code files. External patent information indicates that the algorithms implemented by some source code may be subject to existing patents, which is intended as a warning for the IP staff. Ariadne thus helps detect cases of noncompliance with open-source restrictions or other IP-related constraints.

Early use of the Ariadne prototype shows that making sure developers initialize originality information in their source code is an important requirement for the success of an originality tracking program. A procedure and guidelines for initializing metadata should be instituted by the development organization. The digital signature technology should be used in order to keep everyone accountable. In addition, technology for detecting code cloning could be used in order to minimize the manual effort required for this task.

Performance analysis

In order to test the storage and processing performance of the Ariadne client we used as an experimental vehicle, the source code of the Apache search engine Lucene Version 2.0.0, which consists of 541 Java files.[14] The average size of the Java file is 6224.95 bytes, the average number of code lines per file is 192.78, and the average number of characters per code line is 30.11 bytes.

The storage needed to store the metadata is the sum of the storage for the four components of the metadata: delta lines, line-number lookup table, IP data, and editing history (see Figure 4). The storage required includes both content and XML tags.

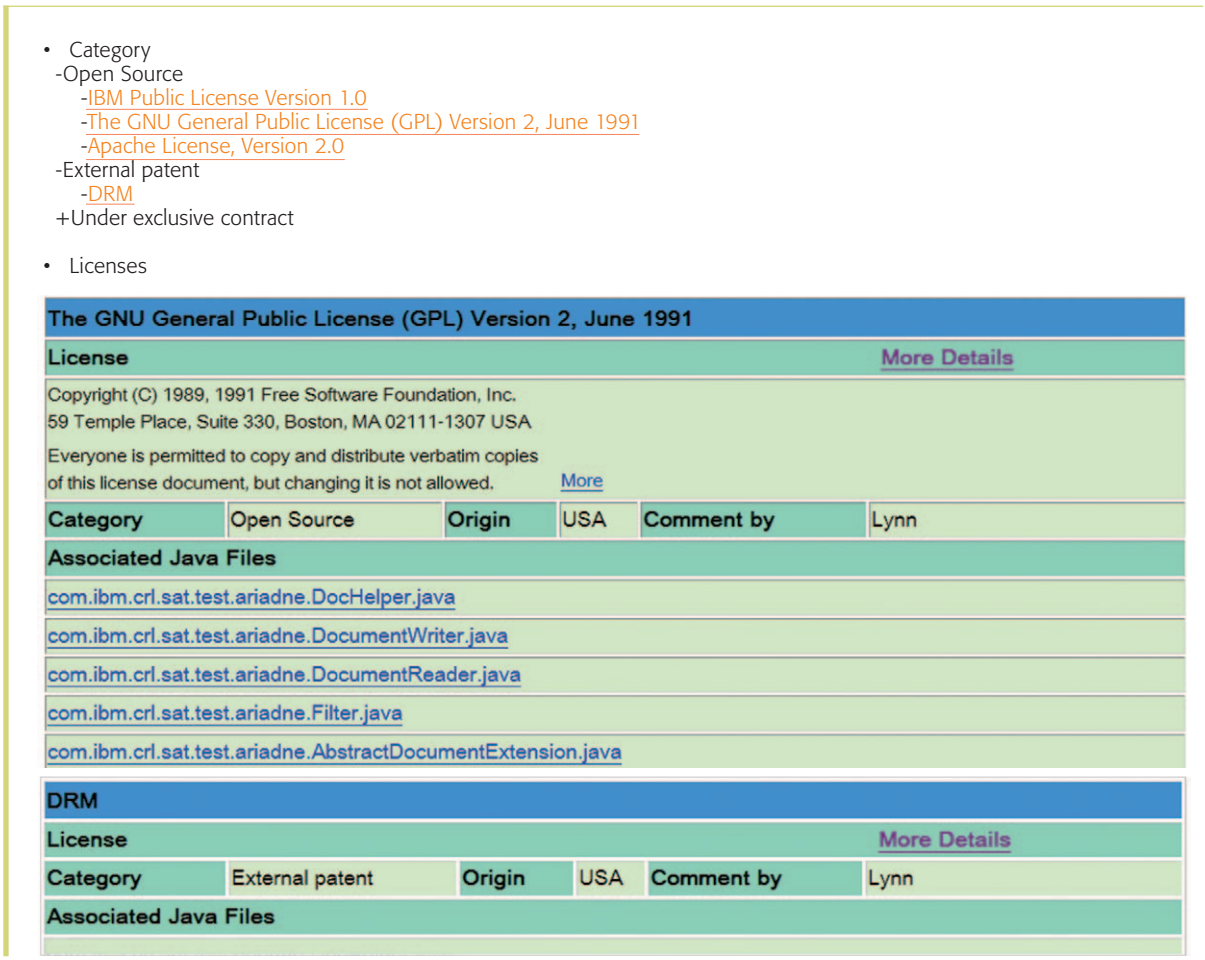

Figure 8
Origin-of-source-code report for a Certificate of Originality

Because the XML tags can be customized or encoded, we include in this discussion only the storage needed for the raw contents. The total storage S_{meta} is given by

$$S_{meta} = S_{delta} + S_{lookup} + S_{ipdata} + S_{history}$$

$S_{history}$ is roughly proportional to the number of times that the source code is modified (number of editing sessions). In general we can assume an empirical constant for it.

S_{ipdata} depends on the number of IP claims. License descriptions often require significant storage space. If the license descriptions are held in a central location, a short URI can be used instead of the full license description. When most of the source code is internally created, the storage needed for IP data is minimal.

S_{lookup} is proportional to the number of history-clue lines, $Num_{ClueLines}$. A row in the lookup table has two columns whose entries are integers; thus, the size of S_{lookup} is 8 times $Num_{ClueLines}$.

S_{delta} depends on the number of deletions and modifications that have been performed and the number of source-code lines affected. Let us assume S_{delta} is 0.5 times S_{source}, which corresponds to a case with many deletions and modifications, in which $Num_{ClueLines}$ is about 1.5 times $Num_{JavaLines}$. Hence, the size of the metadata can be written as:

$$S_{meta} = 0.5 * S_{source} + 8 * 1.5 Num_{JavaLines} + S_{ipdata} + S_{history}.$$

Using the previously stated statistics for Apache Lucene,

$$Num_{JavaLines} = S_{source}/30.11,$$

and thus,

$$S_{meta} = 0.90 * S_{source} + S_{ipdata} + S_{editing\,history}.$$

This value does not include the space taken by XML tags, which may be substantial and may even double the storage size. We can abbreviate or encode the XML tags in order to decrease the size of the metadata file.

We measured the metadata loading time at 213.21 msec per source-code file on the average. Although we have not measured the user response time for the Ariadne client, early experiments show it is quite adequate.

CONCLUSION AND FUTURE WORK

In this paper we introduce Ariadne, a system for tracking originality information in collaborative software development environments. Although our Eclipse-based implementation prototype is designed to handle Java source code, our approach is extensible to other kinds of software artifacts. We describe the history-clue data structure, which is used for managing originality metadata consistently and reliably. Digital signatures are used to validate the integrity of the metadata-handling process. We demonstrate the benefits of Ariadne in two typical scenarios: tracking of software bugs and generating CoO reports.

We are collaborating with software development teams and service teams within IBM in the deployment of Ariadne in a number of internal projects. From these projects we expect to collect data on user experience with the tool, on generating CoO reports, and on managing the software life cycle. In addition, we will gather related information such as the impact that company regulations have on software development, how often source code is collaboratively edited, and how often source code is imported from open-source or legacy sources.

Project managers care not only about source-code level originality, but also about coarser-grain software artifacts such as binary libraries, software components, and even copyrighted images. We plan to extend Ariadne to cover component-level originality information and IP requirements. We also plan to enhance the server-side functions, especially functions for managing the software life cycle, such as checking the compliance of submitted source code with company regulations. In order to integrate Ariadne with existing source-control tools, such as CVS, the Ariadne Client has to support version management for both source files and metafiles.

Developers of open-source software can use Ariadne to ensure the trustworthiness of their code so that other will be more willing to reuse originality-cleared open source in their projects. Unlike the commercial environment, the originality information in the open-source community is less reliable, and there is a strong need to protect the asset owner's originality information from being intentionally deleted. Technologies such as code watermarking could be used to hide the originality-related information in the software artifacts. Code-cloning and plagiarism-detection techniques can be used as auditing mechanisms within the Ariadne framework.

ACKNOWLEDGMENTS

We thank Brent Hailpern, Harold Ossher, Sridhar Iyengar, Mark Wegman, Scott Rich, John Wiegand, Dave Thomson, Hang Jun Ye, and Ling Shao for many fruitful discussions, and Ping Cheng, Chao He, Bo Shu, and Harry Pendergrass for their help in the initial tool development and for many useful discussions.

CITED REFERENCES

1. L. Rosen, *Open Source Licensing: Software Freedom and Intellectual Property Law*, Prentice Hall PTR, Upper Saddle River, NJ (2004).
2. B. Perens, "The Monster Arrives: Software Patent Lawsuits Against Open Source Developers," http://technocrat.net/d/2006/6/30/5032.
3. T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large-Scale Source Code." *IEEE Transactions on Software Engineering* **28**, No. 7, 654–670 (2002).
4. S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document fingerprinting,"

Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (2003), pp. 76–85.

5. "Code Search for Developers," Krugle Inc., http://www.krugle.com.
6. "Google Code Search," Google, Inc., http://www.google.com/codesearch.
7. "ProtexIP™/development Software Compliance Management System," Black Duck Software Inc., http://www.blackducksoftware.com/products/_protexip.html.
8. "IP Amplifier Overview," Palamida, Inc., http://www.palamida.com/products/ipamp/overview.
9. M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," *Proceedings of the 2004 ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2004)* (2004), pp. 83–92.
10. Eclipse SDK, The Eclipse Foundation, http://www.eclipse.org/downloads/.
11. B. Hammond and M. Atreya, *Digital Signatures*, RSA Press, Osborne/McGraw-Hill, New York (2002).
12. *XML Security*, The Apache Software Foundation, http://xml.apache.org/security/.
13. Eclipse Modeling Framework, The Eclipse Foundation, http://www.eclipse.org/emf/.
14. Apache Lucene, Apache Software Foundation, http://lucene.apache.org/java/docs/index.html.

Accepted for publication November 20, 2006.
Published online April 11, 2007.

Lin Luo
IBM Research Division, China Research Lab, Building 19, Zhongguancun Software Park, 8 Dongbeiwang West Road, Haidian District, Beijing 100094, China (luolin@cn.ibm.com). Dr. Luo is a research staff member in the Service Asset Technology department at the China Research Laboratory. She received a Ph.D. degree in electrical engineering from the University of Science and Technology of China in 2003. She subsequently joined IBM at the China Research Laboratory, where she has worked on digital media communication and content protection. Her current research interest is in service asset protection and governance.

Da Ming Hao
IBM Research Division, China Research Lab, Building 19, Zhongguancun Software Park, 8 Dongbeiwang West Road, Haidian District, Beijing 100094, China (haodm@cn.ibm.com). Mr. Hao is an R & D engineer in the Service Asset Technology department at the China Research Laboratory. He received a Master's degree in electrical engineering from the Xi'an Jiaotong University in 2005 and subsequently joined IBM at the China Research Laboratory, where he has worked on content protection and security enhancement technologies. His current research interest is in service asset protection and governance.

Zhong Tian
IBM Software Group, China Development Laboratories, Building 19, Zhongguancun Software Park, 8 Dongbeiwang West Road, Haidian District, Beijing 100094, China (tianz@cn.ibm.com). Dr. Tian is a Senior Architect at the Laboratory-Based Services department of the China Development Laboratories. He received a Ph.D. degree in computer science from Fudan University, Shanghai, China in 1995. He subsequently joined IBM at the China Research Laboratory, where he worked on SOA service discovery and patterns, business-process modeling and integration, B2B eCommerce, and the digital library. His currently interests are in SOA solution architecture through process-modeling and patterns.

Ya Bin Dang
IBM Research Division, China Research Laboratory, Building 19, Zhongguancun Software Park, 8 Dongbeiwang West Road, Haidian District, Beijing 100094, China (dangyb@cn.ibm.com). Mr. Dang is an R & D engineer in the Service Asset Technology department at the China Research Laboratory. He received a Master's degree in electrical engineering from the Xi'an Jiaotong University in 2004. Since joining IBM two years ago, he has worked on digital rights management and content protection. His current research interest is in service asset protection and governance.

Bo Hou
State Key Laboratory of Networking and Switching, Beijing University of Posts and Telecommunications (BUPT), 187#, 10 Xitucheng Road, Haidian District, Beijing 100876, China (polluxplus@gmail.com). Bo Hou is a postgraduate student in BUPT, majoring in computer science. He received his Bachelor's degree in computer science and technology from BUPT in 2005. His current research interest is in next generation network and next generation Internet (NGN & NGI).

Peter Malkin
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 USA (malkin@us.ibm.com). After a one year ITT fellowship at the Australian National University in Canberra Australia, where he worked on the Automated Reasoning Project, Peter Malkin earned a Master's degree in computer science from Brown University in 1987. In 1988, Peter joined the IBM Thomas J. Watson Research Center, and in 2002, he became a Master Inventor by developing and helping others develop intellectual property for IBM. He is now a member of the Enhanced Web Experience Project, working with the SPARCLE Privacy Policy Project, developing a privacy-policy-authoring and a compliance-auditing workbench.

Shun Xiang Yang
IBM Research Division, China Research Laboratory, Building 19, Zhongguancun Software Park, 8 Dongbeiwang West Road, Haidian District, Beijing 100094, China (yangsx@cn.ibm.com). Mr. Yang is a research staff member and the manager of the Service Asset Technology department at the China Research Lab. He received his Master's degree in computer science from Beihang University in 2001. He subsequently joined IBM at the China Research Laboratory, where he has worked on business-process modeling and integration and enterprise-architecture and business-transformation technologies. His current research interests lie mainly in service-asset harvesting, reuse, and governance. ■