Component Retrieval by Mining CVS Commits in Existing Applications

Annie Chen, Eric Chou, Amir Michail
{anniec,tzuchunc,amichail}@cse.unsw.edu.au
University of New South Wales

Abstract

In the information retrieval approach to component retrieval, component profiles are usually extracted from the natural language documentation of the library. Researchers have observed that this method works well with libraries that include extensive documentation, such as Unix man pages. In this paper, we consider an alternative approach where we use information obtained by examining library usage in existing applications. Specifically, we mine so-called "association rules" in CVS commits associated with the applications. (CVS is a version control system widely used in the open source community.) Our approach has the advantage that library documentation is not required and that as more applications are developed using the library, our system will automatically improve in the kinds of queries that it can answer.

1 Introduction

In this paper, we present a new approach to building component retrieval systems that differs from existing approaches in two fundamental ways: (1) the ranking of library components for a given query is determined by analyzing library usage in existing applications — *not* by looking at the library documentation (whether internal or external); and (2) instead of simply looking at the most recent version of the application code that use the library, we look at the CVS commits associated with these applications. (CVS is a version control system widely used in the open source community [8].)

First, we motivate the advantage of looking at applications that use a library instead of looking at the library itself. First of all, the library may not be well documented, either internally in comments or in external documents. This is often the case with open source code. While it may be true that some applications are not well-documented either, it is unlikely that the majority of applications have no useful documentation — particularly since we look at CVS comments, which tend to be meaningful in the open source community. Second, as more applications are developed using the the library, our system will automatically improve in the kinds of queries that it can answer — including some queries that may not have been anticipated nor documented by the library developers. For example, it may be that the library documentation does not mention “double buffering” anywhere (which is a technique used to reduce screen flicker) but that several applications have used library components to implement double buffering, and those words are used in the CVS commits to describe the double buffering code in the application.

Second, we motivate the reasoning behind looking at CVS commits in the applications. We have observed that open source developers are more likely to write CVS comments than actual code comments. We believe this to be the result of two factors. First, open source developers often use CVS comments as an opportunity to describe their changes to other developers so that everyone is made aware of progress and development directions. Second, CVS comments need not be of as high quality as code comments since few people will see them, so open source developers are more likely to write them quickly without worrying too much about whether they are of sufficient quality to avoid embarrassment.

Our CVS-based approach takes advantage of the fact that: (1) a CVS comment typically describes the lines of code involved in the commit; and (2) that this description will typically hold for many future versions. In other words, we look at previous versions to better understand the current version.

Elaborating on point (1), observe that the comment in a CVS commit not only describes the change made but also indirectly describes the purpose of the lines of code involved in that change (e.g., "added footnote feature" indirectly reveals that the lines involved in the commit have something to do with footnotes). Moreover, we find that CVS commit comments tend to be succinct and right to the point — whereas code comments may vary across many levels of abstraction.

With respect to point (2), we note that the purpose of lines usually does not change often — even if the contents of the lines do. For example, mouse handling code will remain just that in many future versions even if some details change throughout the evolution of an application.

In the open source community, CVS commits tend to be fine grained — particularly in large projects with many developers — since the smaller the commit, the less likely it will yield conflicts with commits from other developers. Consequently, CVS commits tend to characterize one well-defined addition or change to the application. For example, a developer may have added code to support drag and drop, and so the corresponding commit would characterize the concept of drag and drop — both in the commit comment (which says “added drag and drop” or something similar) as well as in the commit code (which makes use of the corresponding library classes and functions necessary to implement drag and drop).[1] Consequently, one can find library classes and functions relevant to the drag and drop query.

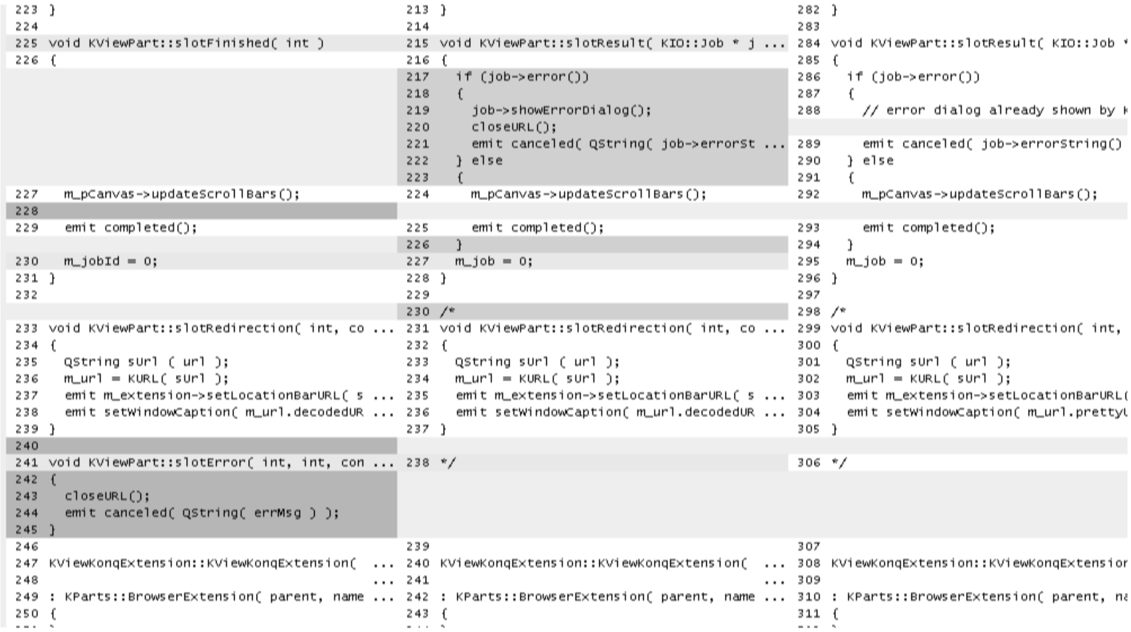
The remainder of the paper is organized as follows. Section 2 explains our technique for building a component retrieval system based on usage of the library in existing applications. Section 3 describes our tool, which we have used on real-life KDE libraries and applications. Section 4 discusses related work. Section 5 summarizes the paper, concluding with future work.

2 Technique

First, we shall give a brief example to motivate our technique. Suppose we would like to know how to do “drag and drop” using the KDE GUI framework. One way to do this is to look at CVS commits of KDE applications. Specifically, we look at two sources of information in each commit: (1) the words in the commit comment; and (2) the words found in the commit code. We would expect to find the words “drag” and “drop” mentioned in the comments, as well as some abbreviations such as “dnd” which may also be used as queries. Moreover, we would expect to find relevant library classes, such as QDropEvent and QDragEnterEvent, and functions, such as acceptDrag() and dragCopy().

As a first approximation, we look for commits with “drag” and “drop” in their comment and then analyze the library classes and functions that tend to be found in the code of such commits. However, we also look for “drag” and “drop” in the code itself because the commit comments are not always adequate. Observe for example that the classes and functions mentioned above all contain “drag” or “drop”

[1] Our approach makes use of *both* the commit comment and commit code for component retrieval. Thus, even if CVS comments are not very good, one can still use the information in the corresponding commit code to search for library components.



```
}

void KViewPart::slotFinished( int )
{
  m_pCanvas->updateScrollBars();

  emit completed();
  m_jobId = 0;
}

void KViewPart::slotRedirection( int, co ...
{
  QString sUrl ( url );
  m_url = KURL( sUrl );
  emit m_extension->setLocationBarURL( s ...
  emit setWindowCaption( m_url.decodedUR ...
}

void KViewPart::slotError( int, int, con ...
{
  closeURL();
  emit canceled( QString( errMsg ) );
}

KViewKonqExtension::KViewKonqExtension(  ...
                                         ...
: KParts::BrowserExtension( parent, name ...
{
```

```
}

void KViewPart::slotResult( KIO::Job * j ...
{
  if (job->error())
  {
    job->showErrorDialog();
    closeURL();
    emit canceled( QString( job->errorSt ...
  } else
  {
    m_pCanvas->updateScrollBars();
    emit completed();
  }
  m_job = 0;
}

/*
void KViewPart::slotRedirection( int, co ...
{
  QString sUrl ( url );
  m_url = KURL( sUrl );
  emit m_extension->setLocationBarURL( s ...
  emit setWindowCaption( m_url.decodedUR ...
}
*/

KViewKonqExtension::KViewKonqExtension(  ...
                                         ...
: KParts::BrowserExtension( parent, name ...
{
```

```
}

void KViewPart::slotResult( KIO::Job *
{
  if (job->error())
  {
    // error dialog already shown by K
    emit canceled( job->errorString()
  } else
  {
    m_pCanvas->updateScrollBars();
    emit completed();
  }
  m_job = 0;
}

/*
void KViewPart::slotRedirection( int,
{
  QString sUrl ( url );
  m_url = KURL( sUrl );
  emit m_extension->setLocationBarURL(
  emit setWindowCaption( m_url.prettyU
}
*/

KViewKonqExtension::KViewKonqExtensior
: KParts::BrowserExtension( parent, na
{
```

Figure 1: Three versions of kview_view.cc are shown: (1) v. 48 before commit on left; (2) v. 49 after commit in middle; and (3) v. 68 (most recent) on right.

in the name, so we would likely find many commits containing drag and drop code even if the corresponding commit comment does not explicitly mention drag and drop.

In what follows, we shall elaborate on two key steps in this process: (1) associating past CVS comments with the most recent version of the code; and (2) the data mining done to rank library classes/functions given the query.

2.1 CVS Comments

The first step in our approach is to produce a mapping between CVS comments and the lines of code to which they refer in applications that make use of the library. Here we are only interested in the lines of code found in the newest version of each file. Observe that a line may be involved in multiple commits in which case it would have multiple CVS comments associated with it. We give a brief overview of the algorithm in what follows.[2]

Consider a file f at version i which is then modified and committed into the CVS repository yielding version $i + 1$. Moreover, suppose the user entered a comment C which is associated with the triple $(f, i, i + 1)$.

By performing a diff[3] between versions i and $i+1$ of f, we can determine lines that are modified or inserted in version $i + 1$; we associate comment C with all such lines. (Figure 1 shows such a diff, visually, between two successive versions of a file where $i = 48$; modified or inserted lines in version 49 are shaded.)

We are interested in a component retrieval system that characterizes library usage in the present, not at some point in the past. For example, library API for drag and drop may change at some point and so application code would have to be adapted. If we sim-

[2] Details can be found in our paper on CVSSearch [6].

[3] We actually use a somewhat more advanced version of GNU diff that detects "similar" lines [6].

ply looked at the code in the past commits, we could possibly return the wrong library classes/functions. However, note that the purpose of drag and drop code would remain the same — even if the code itself changes somewhat — and so the CVS comment describing the code as implementing drag and drop would remain to be correct.

Consequently, we are interested in associating past CVS comments with the most recent version of each file (so that we can identify the most current library usage). This means we need a *propagation phase* during which the comments associated with version $i+1$ of f are "propagated" to the corresponding lines in the most recent version of f, say $j \geq i+1$.This is done by performing diffs on successive versions of f to track the movement of these lines across versions (even in the presence of changes to the lines themselves) until we reach version j. (Figure 1 shows the final outcome of this propagation phase in the third file which has version $j = 68$. Observe how the lines are matched up across versions 49 and 68 even though the line numbers have changed due to deletions/additions of preceding lines in the file over time.)

2.2 Data Mining

Given the information in the commit comments and code, we can now build a component retrieval system. Of course, many information retrieval strategies have been tried in component retrieval systems [3, 5, 9, 12]. However, these typically rely on the availability of high quality external documentation. In our case, the granularity of commits varies (in some cases a commit involves multiple unrelated changes). Moreover, the quality of commit comments and code comments/naming conventions is highly variable. To discover library usage in this noisy data, we use *data mining*, which may be defined as follows:

> The process of nontrivial extraction of implicit, previously unknown and potentially useful information (such as knowledge rules, constraints, regularities) from data [16].

Association Rules We shall mine the CVS commit data in a way that is similar to the mining of *association rules* [1]. Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called *items*. An *association rule* is an implication of the form $(\bigwedge_{x \in X} x) \Rightarrow \left(\bigwedge_{y \in Y} y\right)$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$. For brevity, we also write this as $X \Rightarrow Y$.

For example, suppose that people who purchase bread and butter also tend to purchase milk. In that case, the corresponding association rule is "bread$\wedge$butter$\Rightarrow$milk". The antecedent of the rule X consists of bread and butter and the consequent Y consists of milk.

Such rules are useful for analyzing data. For example, to determine how one might boost the sales of milk, one could look for rules that have "milk" in the consequent. To determine the impact of discontinuing the sale of butter, one could find all rules that have "butter" in the antecedent.

Let D be a set of *transactions*, where each transaction T is a set of items (*itemset*) such that $T \subseteq I$. We say that a rule $X \Rightarrow Y$ holds in transaction set D with *support* $s\%$ if $x\%$ of transactions in D contain $X \cup Y$. Support is an indication of the importance of the rule. If it is only supported by a few transactions, then we are not likely to care much about it. For example, one rule that is supported by 10% of transactions would be considered significantly more important than one supported by only 0.5% of transactions.

We say that a rule $X \Rightarrow Y$ holds in transaction set D with *confidence* $c\%$ if $c\%$ of transactions in D that contain X also contain Y. Confidence is an indication of rule strength. Returning to our example, suppose we find that in 60% of transactions in which customers purchase bread and butter, they also purchase milk.

While confidence is meant to be an indicator of rule strength, it can be misleading. For example, it may be that 65% of transactions involve the purchase of milk, so the presence of bread and butter would actually *decrease* the likelihood of finding milk. Generally, frequent consequents present a problem to the confidence measure.

Consequently, two other measures have been considered for rule strength: *interest* and *conviction* [4]. In probability terms, the confidence of $A \Rightarrow B$ is $P(A,B)/P(A)$, where $P(A,B)$ is the probability of finding items $A \cup B$ in a transaction and $P(A)$ is the probability of finding items A in a transaction. To address the problem of frequent consequents B, the interest measure has been proposed: $\frac{P(A,B)}{P(A)P(B)}$. Observe that interest characterizes the departure from independence of A and B. However, as the formula is symmetric, it is a measure of co-occurrence rather than implication.

The conviction measure addresses the problem of frequent consequents and provides implication. Conviction is defined as $\frac{P(A)P(\neg B)}{P(A,\neg B)}$. Suppose that $P(A) > 0, P(\neg B) > 0$, and whenever A is found in a transaction, we always find B. In that case, $P(A, \neg B) = 0$ and the conviction is ∞, signifying a "perfect" rule. Now if A and B are completely independent, then observe that $P(A, \neg B) = P(A)P(\neg B)$, in which case the conviction is equal to 1, signifying no implication at all. Conviction measures the departure from independence of A and $\neg B$.

We have tried both interest and conviction in our tool. We shall comment on our experience with both measures in Section 3.

Mining Library Usage Now we consider data mining library usage. Suppose we are interested in mining the usage of libraries $L_1, \ldots, L_m$ in applications $A_1, \ldots, A_n$. (An application may use multiple libraries at once.) There are three key steps:

1. identify all classes and functions in libraries $L_1, \ldots, L_m$;

2. identify all CVS commits for applications $A_1, \ldots, A_n$; and

3. given a query Q consisting of words $w_1, w_2, \ldots, w_k$, mine association rules of the form $w_1 \wedge w_2 \wedge \cdots \wedge w_k \Rightarrow a$, where a is a library class or function.

First, we identify all library classes and functions. This is done by running the ctags utility on all library header files that may be included by applications.[4] This is done for all libraries of interest and all the results are put into a single item set $I_{L_1,\ldots,L_m}$. For example, we may have $I_{L_1,\ldots,L_m} = \{QWidget, show(), hide(), \ldots\}$.

Second, we identify all CVS commits for applications $A_1, \ldots, A_n$. (Observe that a commit may involve multiple files.) We denote each commit by a 2-tuple, (M, C), where

- M is the set of CVS commit message words, all stemmed (e.g., "dragged" becomes "drag"), along with all words found in the commit code, including those that are part of identifiers (e.g., "drop" is extracted from QDropEvent), again all stemmed;

- C is the set of code classes and functions in the most recent version of the file(s) that are also present in $I_{L_1,\ldots,L_m}$ (e.g., QDropEvent and acceptDrag()).

These 2-tuples will be our "transactions" T in association rule terminology. The set of all such transactions D contains commits from all applications $A_1, \ldots, A_n$. (The commit code is obtained using the propagation technique described in Section 2.1, so the commit code actually refers to the corresponding code in the most recent version of the files.)

Third, given a query Q consisting of words $w_1, w_2, \ldots, w_k$, with all words w_i stemmed, we

[4] We use the version of ctags from http://ctags.sourceforge.net for C++. Private member functions are omitted from our analysis.

mine association rules of the form $w_1 \wedge w_2 \wedge \cdots \wedge w_k \Rightarrow a$, where a is a class or function in $I_{L_1,\ldots,L_m}$. (As we shall see in Section 3, the query may also consist of library classes and functions; what is described here still applies in that case except that there is no stemming of query words.) Data mining is done *on-the-fly* at query time. Specifically, given the query $Q = \{w_1, \ldots, w_k\}$, we find all commits $S_Q = \{(M_i, C_i) | Q \subseteq M_i\}$. That is, we find all commits whose words include all the query words. From this, we consider the classes and functions a to return, where a is in the code set C_i associated with a commit in S_Q. We return a rule $w_1 \wedge w_2 \wedge \cdots \wedge w_k \Rightarrow a$ for each such a.

Of course, the order in which we return rules $w_1 \wedge w_2 \wedge \cdots \wedge w_k \Rightarrow a$ is important. Our tool gives the user a choice of ranking library classes and functions using either interest or conviction. (With interest, there is no one-way implication, so the rule is more accurately written as $w_1 \wedge w_2 \wedge \cdots \wedge w_k \Leftrightarrow a$. Moreover, some simple algebra shows that, if the query is fixed, one obtains the same ranking using interest as using conviction with backwards implication $w_1 \wedge w_2 \wedge \cdots \wedge w_k \Leftarrow a$; consequently we only consider conviction with forward implication along with interest in our tool.)

3 Tool

We have built a component retrieval tool based on the techniques described in Section 2. Currently, C++ is supported, though other languages can be easily added. The tool was run on 210 real-life KDE applications — including an office suite and web browser — consisting of about 2.7 million lines of code, that make use of the Qt and KDE GUI libraries. (The KDE libraries extend the core functionality offered by the Qt GUI toolkit.) In total, there were approximately 34,000 CVS commits spread across 12,194 application files. The tool found 1182 library classes and 8537 (typically member) library functions used in these applications. All data mining is done at query time, with association rules pruned when their support is less than 10. Typically, the on-the-fly data mining takes under 5 seconds in this data set.

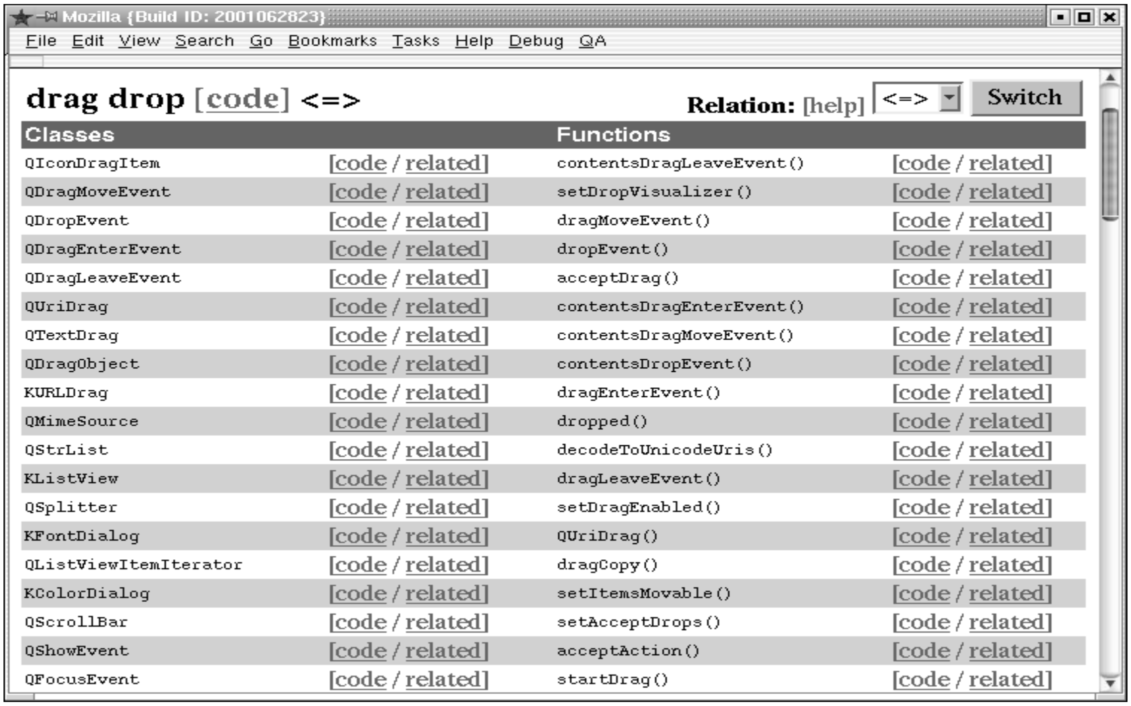
Figure 2, (a) shows the results for the query "drag drop" using interest ranking. (The user can also switch to conviction ranking with forward implication using the Relation drop-down menu.) We observe that the classes and functions returned are indeed highly relevant to the drag drop query.

Our tool not only provides a list of library classes/functions to accomplish a particular task (e.g., drag drop), but also provides numerous code fragments to demonstrate the kind of code people write to accomplish that task. In particular, the applications are not only used to find relevant library classes and functions, but also to demonstrate their usage by example.

For example, clicking on the code link for class QDropEvent results in the list of commits shown in Figure 2, (b). These are commits that contain "drag" and "drop" in the corresponding comment and/or code. Observe that the commit comments are used to give the user an idea of how relevant the commit is with respect to the query given. In this way, the user is more likely to click on a commit with useful code to demonstrate usage of the library classes/functions in question. Clicking on "Browse Code" in the top commit shown in Figure 2, (b) yields the commit code shown in Figure 3, (a). The code shown is from the most recent version of the application, as "propagated" from the original commit. (See Section 2.1.)

It is also possible to use a library class or function as the query. For example, clicking on the "related" link of QMimeSource in Figure 2, (a) uses this class as the query, in which case the system uses the same data mining techniques discussed earlier to find other classes and functions that tend to be found in commits containing QMimeSource. In this case, the query QMimeSource results in the classes and functions shown in Figure 3, (b). By repeatedly clicking on "related", it is possible to browse related concepts of the library much as one browses related concepts on the world wide web. In this case,

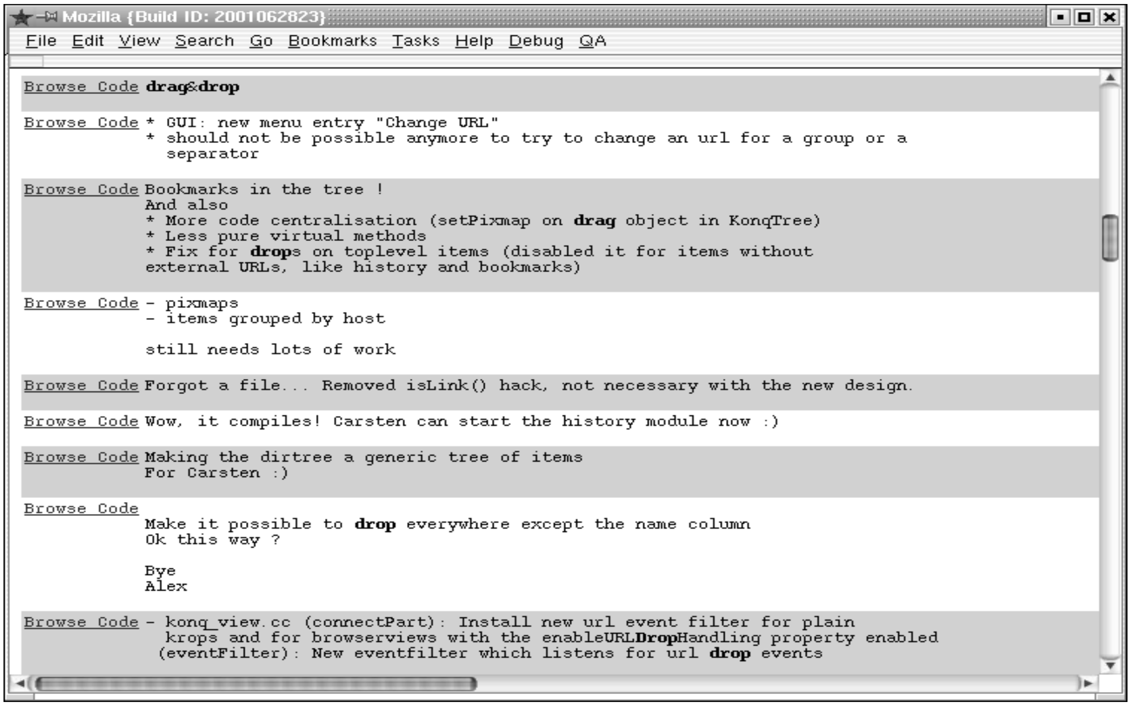
(a)


Mozilla {Build ID: 2001062823}
File Edit View Search Go Bookmarks Tasks Help Debug QA
Relation: [help]
<=>
Switch

drag drop [code] **<=>**

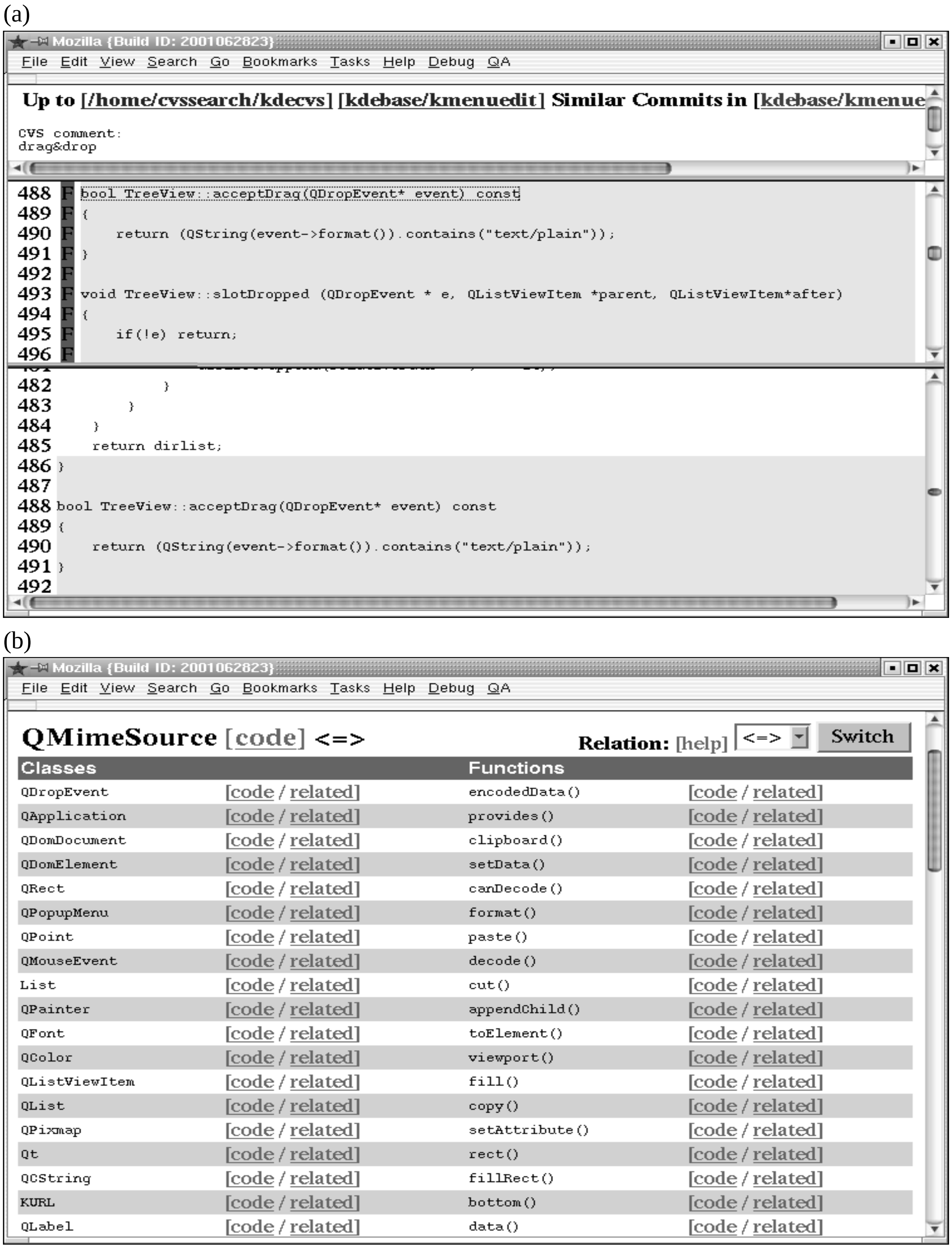
Classes		Functions	
QIconDragItem	[code / related]	contentsDragLeaveEvent()	[code / related]
QDragMoveEvent	[code / related]	setDropVisualizer()	[code / related]
QDropEvent	[code / related]	dragMoveEvent()	[code / related]
QDragEnterEvent	[code / related]	dropEvent()	[code / related]
QDragLeaveEvent	[code / related]	acceptDrag()	[code / related]
QUriDrag	[code / related]	contentsDragEnterEvent()	[code / related]
QTextDrag	[code / related]	contentsDragMoveEvent()	[code / related]
QDragObject	[code / related]	contentsDropEvent()	[code / related]
KURLDrag	[code / related]	dragEnterEvent()	[code / related]
QMimeSource	[code / related]	dropped()	[code / related]
QStrList	[code / related]	decodeToUnicodeUris()	[code / related]
KListView	[code / related]	dragLeaveEvent()	[code / related]
QSplitter	[code / related]	setDragEnabled()	[code / related]
KFontDialog	[code / related]	QUriDrag()	[code / related]
QListViewItemIterator	[code / related]	dragCopy()	[code / related]
KColorDialog	[code / related]	setItemsMovable()	[code / related]
QScrollBar	[code / related]	setAcceptDrops()	[code / related]
QShowEvent	[code / related]	acceptAction()	[code / related]
QFocusEvent	[code / related]	startDrag()	[code / related]

(b)


Mozilla {Build ID: 2001062823}
File Edit View Search Go Bookmarks Tasks Help Debug QA

```
Browse Code drag&drop

Browse Code * GUI: new menu entry "Change URL"
            * should not be possible anymore to try to change an url for a group or a
              separator

Browse Code Bookmarks in the tree !
            And also
            * More code centralisation (setPixmap on drag object in KonqTree)
            * Less pure virtual methods
            * Fix for drops on toplevel items (disabled it for items without
            external URLs, like history and bookmarks)

Browse Code - pixmaps
            - items grouped by host

            still needs lots of work

Browse Code Forgot a file... Removed isLink() hack, not necessary with the new design.

Browse Code Wow, it compiles! Carsten can start the history module now :)

Browse Code Making the dirtree a generic tree of items
            For Carsten :)

Browse Code
            Make it possible to drop everywhere except the name column
            Ok this way ?

            Bye
            Alex

Browse Code - konq_view.cc (connectPart): Install new url event filter for plain
              krops and for browserviews with the enableURLDropHandling property enabled
             (eventFilter): New eventfilter which listens for url drop events
```

Figure 2: Results for “drag drop” query shown in (a), ranked by interest; clicking on “code” link for QDropEvent shows the list of corresponding commits in (b).


(a)

Mozilla {Build ID: 2001062823}
File Edit View Search Go Bookmarks Tasks Help Debug QA

Up to [/home/cvssearch/kdecvs] [kdebase/kmenuedit] Similar Commits in [kdebase/kmenue

```
CVS comment:
drag&drop
```

```
F bool TreeView::acceptDrag(QDropEvent* event) const
F {
F     return (QString(event->format()).contains("text/plain"));
F }
F
F void TreeView::slotDropped (QDropEvent * e, QListViewItem *parent, QListViewItem*after)
F {
F     if(!e) return;
F
```

```
            }
        }
    }
    return dirlist;
}

bool TreeView::acceptDrag(QDropEvent* event) const
{
    return (QString(event->format()).contains("text/plain"));
}

```

(b)

Mozilla {Build ID: 2001062823}
File Edit View Search Go Bookmarks Tasks Help Debug QA

QMimeSource [code] **<=>**

Relation: [help] <=> Switch

Classes		Functions	
QDropEvent	[code / related]	encodedData()	[code / related]
QApplication	[code / related]	provides()	[code / related]
QDomDocument	[code / related]	clipboard()	[code / related]
QDomElement	[code / related]	setData()	[code / related]
QRect	[code / related]	canDecode()	[code / related]
QPopupMenu	[code / related]	format()	[code / related]
QPoint	[code / related]	paste()	[code / related]
QMouseEvent	[code / related]	decode()	[code / related]
List	[code / related]	cut()	[code / related]
QPainter	[code / related]	appendChild()	[code / related]
QFont	[code / related]	toElement()	[code / related]
QColor	[code / related]	viewport()	[code / related]
QListViewItem	[code / related]	fill()	[code / related]
QList	[code / related]	copy()	[code / related]
QPixmap	[code / related]	setAttribute()	[code / related]
Qt	[code / related]	rect()	[code / related]
QCString	[code / related]	fillRect()	[code / related]
KURL	[code / related]	bottom()	[code / related]
QLabel	[code / related]	data()	[code / related]

Figure 3: Clicking on "Browse Code" for first commit in Figure 2, (b) yields the commit code in (a); clicking on the "related" link for QMimeSource in Figure 2, (a) yields (b).

starting with the “drag drop” query, we have found the QMimeSource class, which represents an abstract piece of formatted data. Using QMimeSource as the query yields not only drag and drop related classes/functions, but also those related with cut and paste. Indeed, both drag/drop and cut/paste are related concepts and operate on formatted data, as encapsulated in QMimeSource.

Typically, an object-oriented library has many more functions than it has classes, and finding the right function to use can be especially difficult. To give us some idea of the kinds of functions returned by the tool, we ran it on queries corresponding to the most widely used 100 library classes (as determined by the number of commits that contain those classes in the code). For example, from the class QDropEvent, we would consider the query “drop event”. We call QDropEvent the *query class* since the query was derived from this class. We then examined the functions returned by the tool. Averaged across all such queries, we observed the following:

Top	Conviction	Interest
10	27.1%	22.2%
20	23.7%	20.4%
30	20.4%	18.5%
40	18.7%	17.2%
50	17.2%	16.0%

The table above shows the percentage of functions returned that are members (defined/inherited) of the query class corresponding to the query performed. We see that this percentage tends to be slightly greater for the conviction measure than it is for the interest measure, and that the percentages goes down as the number of results returned increases. The majority of functions returned are those not defined/inherited by the query class and that tend to be used with the query class in practice — these are usually shown in manually constructed library documentation under the “see also” section.

While we have not formally evaluated the tool, we have observed some differences between the interest and conviction rankings after manual analysis of numerous queries. Since conviction models unidirectional implication, it is possible to have uninteresting classes/functions show up for a particular query simply because those classes/functions show up in almost all commits. For example, the conviction ranking for “drag drop” has QString ranked 2nd and QWidget ranked 3rd. Such classes are widely used and are not so interesting in the context of drag and drop. While interest does not have this problem, it’s rankings of relevant classes/functions may not be intuitive if those classes/functions tend to be used in other contexts where the query words do not appear — in which case they will be penalized in their ranking even if they are highly relevant to the query. In future work, we shall consider using a combination of the two measures as well as other completely different ranking methods.

4 Related Work

Our component retrieval tool extends our previous work on CodeWeb [13, 14] and CVSSearch [6]. The work on CodeWeb introduced the notion of library reuse patterns, which are similar to our queries with classes and functions (e.g., the “related” link). However, CodeWeb does not allow arbitrary queries with on-the-fly data mining nor does it make use of CVS commits. CVSSearch is a general purpose search tool for code based on CVS commits but it has no notion of libraries — that is, CVSSearch is not a component retrieval system. The work in this paper builds on CVSSearch to make into one.

Of course, researchers have taken numerous approaches to addressing the component retrieval problem. We discuss some of these below and point out how our approach differs. As done elsewhere by other authors, we shall also divide our discussion into information retrieval and domain-specific approaches [11].

4.1 Information Retrieval Approaches

In the information retrieval approach, all information is provided by natural language documentation of the library components. Researchers have observed that this method works well with libraries that include extensive documentation, such as Unix man pages [3, 5, 9, 12]. Then one can rely on regularities in the text such as relative word frequencies or lexical affinities [12]. The advantage of this approach is that no semantic knowledge is used and no interpretation of the components and their documentation is made. The goal is to characterize the component documents rather than to understand them.

Our approach differs in several ways. Since we cannot be sure of the quality of any particular CVS comment, we rely instead on data mining patterns across thousands of CVS comments. Moreover, we do not use the library documentation at all. Rather, we use the CVS comments associated with applications developed using the library. This has the added advantage that as more applications are developed for the library, our component retrieval system will improve — by better handling typical queries and even unusual queries that the authors of the library may not have thought about (or documented).

4.2 Domain-Specific Approaches

Interestingly, domain specific approaches have been much more popular than information retrieval methods for component retrieval in the software reuse community [2, 7, 10, 17, 18, 19, 15]. In this approach, it is typically the case that a domain expert manually provides descriptions of each component.

For example, in the faceted approach to component retrieval [17], an expert evaluates a component with respect to several facets or viewpoints. As an example, in a data structures library, an expert may decide to describe the classes by facets allocation = {bounded, unbounded} and iterator = {none, supplied}. Using these facets, the expert may then describe a linked-list class by the facet pairs (allocation, unbounded) and (iterator, supplied) say. A similarity measure can be be defined over the facets to determine which classes best match the user's query.

While such systems can be more accurate and complete than those based on information retrieval, there is a price to be paid: they require extensive domain analysis to produce the high quality semantic descriptions of the components. Information retrieval methods are fully automated and are consequently much cheaper and more scalable.

5 Conclusions

In this paper, we have presented an alternative approach to constructing component retrieval systems that makes use of the information associated with the applications that use a library rather than information associated with the library itself. As far as we know, this is the first component retrieval system of its kind.

As we have discussed, this approach is particularly useful when the library documentation is poor — as is often the case in the open source community — and also has the advantage that as more applications are developed, the component retrieval system improves automatically in the kinds of queries it can handle — and can even answer queries that the library developers did not anticipate nor document (e.g., double buffering) as long as some application(s) written using the library implement the concept and contain the corresponding concept keywords in the commit comments and/or code.

Another novelty of our approach is the fact the we make use of CVS commits, which tend to contain code fragments relevant to only one or two concepts (and so it is easier to identify related library classes/functions as well as those relevant to the query), and are typically described by succinct and meaningful CVS comments. However, even if the CVS comments are of poor quality, our approach also considers words in the commit code.

We have presented a ranking system that works by mining association rules. Two well-known measures, interest and conviction, were tried to indicate the strength of the mined rules. Preliminary observations seem to indicate that both measures can be useful — and perhaps should be combined. For future work, it would be interesting to attempt a more formal evaluation of these two measures as well as to try completely different data mining and/or information retrieval methods in our application-based framework for component retrieval.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th Very Large Data Bases Conference*, pages 487–499, 1994.

[2] B. P. Allen and S. D. Lee. A knowledge-based environment for the development of software parts composition systems. In *Proceedings of the 11th International Conference on Software Engineering*, pages 104–112, 1989.

[3] S. P. Arnold and S. L. Stepoway. The reuse system: Cataloging and retrieval of reusable software. In *Software Reuse: Emerging Technology*, pages 138–141, 1987.

[4] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD*, pages 255–264, 1997.

[5] B. A. Burton, R. Wienk Aragon, S. A. Bailey, K. D. Koelher, and L. A. Mayes. The reuse system: Cataloging and retrieval of reusable software. In *Software Reuse: Emerging Technology*, pages 129–137, 1987.

[6] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through source code using CVS comments. To appear in *International Conference on Software Maintenance*, 2001. Available from http://cvssearch.sourceforge.net.

[7] P. Devanbu, P. G. Selfridge, B. W. Ballard, and R. J. Brachman. A knowledge-based software information system. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, pages 110–115, 1989.

[8] K. F. Fogel. *Open Source Development with CVS*. Coriolis Inc., 2000.

[9] W. B. Frakes and B. A. Nejmeh. Software reuse through information retrieval. In *20th Hawaii International Conference on System Sciences*, pages 530–535. IEEE, 1987.

[10] D. Gangopadhyay and A. R. Helm. A model-driven approach for the reuse of classes from domain specific object-oriented class repositories. Technical Report RC14510, IBM Rsearch Division, 1989.

[11] R. Helm and Y. S. Maarek. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–61, 1991.

[12] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[13] A. Michail. Data mining library reuse patterns in user-selected applications. In *14th IEEE International Conference on Automated Software Engineering*, pages 24–33, 1999.

[14] A. Michail. Data mining library reuse patterns using generalized association rules. In *Pro-*

ceedings of the 22nd International Conference on Software Engineering, 2000.

[15] A. Moormann-Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.

[16] G. Piatetsky-Shapiro and W. J. Frawley. *Knowledge Discovery in Databases*. AAAI/MIT Press, 1991.

[17] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, 1987.

[18] W. F. Tichy, R. L. Adams, and L. Holter. NLH/E: A natural-language help system. In *Proceedings of the 11th International Conference on Software Engineering*, pages 364–374, 1989.

[19] M. Wood and I. Sommerville. An information retrieval system for software components. *SIGIR Forum*, 22(3,4):11–25, 1988.