Software Repositories: A Source for Traceability Links

Huzefa Kagdi and Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242

{hkagdi, jmaletic}@cs.kent.edu

ABSTRACT
This paper analyzes six open source projects in order to assess software repositories, such as those managed by *Subversion*, as a source for uncovering/discovering traceability links between different types of software artifacts. Our finding suggests that software repositories store a variety of artifacts that are central to open source development and use. Furthermore, a heuristic-based approach that uses sequential-pattern mining is presented. This approach analyzes commits in a version history to mine for highly frequent co-occurring changes to different artifacts (e.g., source code and documentation). The hypothesis is if different types of artifacts are committed together frequently then there is a high probability that they have a traceability link between them. Examples of mined traceability links from our preliminary experimentation on mining KDE (K Desktop Environment) repositories are presented.

Categories and Subject Descriptors
D.2.7. [**Software Engineering**]: Distribution, Maintenance, and Enhancement – documentation, enhancement, extensibility, version control.

General Terms
Management, Measurement, Documentation,

Keywords
Traceability, Link Recovery, Mining Software Repositories.

1. INTRODUCTION
Recovery of traceability links has been a subject of investigation for many years within the software engineering community [18, 29]. Various techniques have been proposed to assist in the recovery/discovery of traceability links in existing software systems [29]. However, many approaches suffer from many false positives, suggesting a link when none should exist [4, 23, 24].

TEFSE/GCT'07, March 22-23, 2007, Lexington, KY, USA.

Approaches to recovering traceability links normally analyze only a single snapshot (i.e., current version) of a software system to infer links between two or more artifacts. The research presented here to recover links takes a different approach and examines multiple versions of the software artifacts, stored in software repositories (e.g., *CVS* and *Subversion*). The premise is that artifacts of different types (e.g., *src.cpp* and *help.html*) co-changed in the past potentially have a traceability link between them.

Towards achieving our goal, a set of prerequisite questions needs to be answered: What are the different types of artifacts? Are different kinds of artifacts typically committed together? How many of them are typically committed together? In this paper we investigate the above questions on software repositories of six open source systems. We also present, an approach based on sequential-pattern mining to uncover/discover traceability links from frequently occurring co-changes.

The rest of the paper is organized as follows: Section 2 provides an overview of current state in traceability for open source projects. Section 3 details on our effort on mining traceability links between different types of artifacts from software repositories. Section 4 is a brief discussion on addressing challenges in traceability research. Section 5 presents related work and finally we conclude in Section 6.

2. OPEN SOURCE AND TRACEABILITY
Scacchi et al. [26] observed that requirements elicitation, analysis, and specification of open source system are very different from the traditional approaches (e.g., use of mathematical logic, descriptive schemes, and UML design models) in software engineering. Their requirements are typically implied by discourse of project participants, and after implementation assertions. Different types of informal sources (termed as *software informalisms*) form collective requirements and documentation of an open source project. This includes software repositories, communications, HowT*o* guides, and traditional system documents (e.g., *man* pages). One particular type of requirements that is a common feature in many open source projects is the ability to support extension mechanisms with various programming languages and architecture (e.g., a python binding to the *KDE* libraries). Due to the distributed collaborative nature of open-development, software repositories comprise the primary location of various project artifacts along with the primary means of coordination and archival.

Source-control systems, bug-tracking systems, and archived communications are the main sources used in free and open source software development. Source-control systems are primarily used

for managing evolution of source code artifacts (i.e., files). Bug or issue tracking systems are used to manage the reporting and resolution of requests such as defects, bugs, faults, and feature enhancements. They include priority and severity assignment for a request, and discussion of possible solutions for resolution. For example, *Bugzilla* (a widely used system in the open source development community) is used to manage the life cycle of a request and allows "free-form" textual description and discussion of a request. Archived communications such as email, newsgroups, discussion forums, and instant messages store discussions between project participants, making them sources for information including change rationales. It is not uncommon to have a number of mailing lists for a variety of different purposes.

These repositories vary in their usage, information content, and storage format. The bug/issue tracking repositories and emails can be seen as a source for requirements and corrective-maintenance requests of an open source system. Source-control repositories can be seen as a source of implementation artifacts. However, these repositories are managed and operated (for the most part) in isolation and have no explicit direct relationship with each other. For example, no explicit information is typically maintained between a particular bug in the bug-tracking system and the corresponding source code changes in the source-control repositories that fixed it.

Few efforts have been made to infer and then utilize traceability links between artifacts in bug repositories and source code artifacts via Mining Software Repositories (MSR). Canfora et al. [5] used the bug descriptions and the *CVS* commit messages for the purpose of change predictions. Their approach provides a set of files (at line level of granularity) that are likely to change given only the textual description of a new bug (or feature). An information-retrieval method is used to index the changed files in the *CVS* repositories with the textual description of past bug reports in the *Bugzilla* repository and the *CVS* commit messages. A bug report is linked to a *CVS* commit (i.e., a set of changed files) based on the explicit bug identifier found (a common practice in open source development) in that commit message (e.g., bug id 30,000). Sliwerski et al. [28] used a combination of information in the *CVS* log file (commits) and *Bugzilla* to study *fix-inducing* changes. Fix-inducing changes are the changes that introduced new changes to fix an earlier reported problem. Regular-expression matching on the commit messages and text descriptions in *Bugzilla* along with heuristics are used to determine the *CVS* deltas that are related to a change that fixes a bug. Cubranic et al. [9, 10] describes a tool, namely *Hipikat*, to assist new developers (not necessarily novice) on a project, in performing their current task(s). Various artifacts (e.g., source code, emails, and bug reports) produced in the project are integrated to form a repository of explicit information – project memory. A vector-based IR method is used to draw the similarity between artifacts. Heuristics are used to form other relationships between artifacts (e.g., requests in *Bugzilla* are related to the files in *CVS* by matching bug-id in the commit messages). *Hipikat* recommends artifacts from the project memory that may hold relevance to a task at hand. A developer may ask for the relevant artifacts via an explicit query, or the tool can do so automatically based on the current context (e.g., based on the currently open document(s) in the developer's workspace).

In summary, existing MSR approaches have focused on uncovering traceability links between requests in bug-tracking systems and source code. While these are important efforts, they cover only a part of the broad spectrum of documents found in open source development.

3. DIFFERENT ARTIFACTS AND UNCOVERING THEIR TRACEABILITY

Our research interest is in uncovering traceability between source code and other documents such as those reported by Scacchi et al. [26]. This includes:

- User documents (e.g., *HTML*, *XML/docbook*, *LaTeX* and *Doxygen*)
- Build management documents (*automake*, *cmake*, and *makefile*),
- HowTo guides (e.g., *FAQs*),
- Release and distribution documents (e.g., *ChangeLogs*, *whatsNew*, *REAME*, and *INSTALL* guide)
- Progress monitoring (*TODO* and *STATUS*)
- Extensible mechanisms (e.g., *Python*, *Ruby*, and *Pearl* bindings for an *API*)

A sustainable success of an open source project from both development and end use perspectives depends to a large extent on how well they maintain these documents. For example, an application that frequently fails to compile or with very little installation help could have a diminishing effect on the user base. It is important that these documents be kept in alignment with the current state of the source code. Therefore, traceability between them is of desirable interest and value. Accounting these documents along with the requests in bug-tracking systems is a major step towards achieving the complete picture of traceability to source code in the context of open source development.

We now must address a couple of questions. Where are these documents found? How do we uncover the traceability links between them and the source code? Our study (refer Section 3.2) shows that these documents are stored and managed in software repositories along with source code. Change-sets in which they are found along with source code are a valuable source for uncovering traceability links between them. We first describe how change-sets are stored and represented in software repositories to help facilitate following discussion of our mining approach for traceability links.

3.1. Change-sets in Software Repositories

Source code repositories store metadata such as user-IDs, timestamps, and commit comments in addition to the source code artifacts and their differences across versions. This metadata explains the why, who, and when dimensions of a source code change. Modern source-control systems, such as *Subversion*, preserve the grouping of several changes in multiple files to a single change-set as performed by a committer. Version-number assignment and metadata are associated at the change-set level and recorded as an atomic commit.

Figure 1 shows a log entry from the *Subversion* repository of *kdelibs* (a part of *KDE* repository). A log entry corresponds to a single commit operation. *Subversion*'s log entries include the dimensions *author*, *date*, and *paths* involved in a change-set. In this case, the changes in the files *khtml_part.cpp* and l*oader.h* are

committed together by the developer *kling* on the date/time *2005-07-25T17:46:20.434104Z*. The *revision* number *438663* is assigned to the entire change-set (and not to each file that is changed as is the case with some version-control systems such as CVS). Additionally, a text message describing the change entered by the developer is also recorded. Note that the order in which the files appear in the log entry is not necessarily the order in which they were changed. Clearly, a single log entry alone is insufficient to give the temporal ordering in which files were changed. However, there is a temporal order between change-sets. Change-sets with greater revision numbers occur after those with lesser revision numbers. Therefore, we can utilize the ordering of change-sets to determine the ordering of changes between different files. In the rest of the paper we use the term change-sets for the log entries or commits in *Subversion* repositories.

```
<?xml version="1.0" encoding="utf-8"?>
<log>
  <log entry revision="438663">
    <author>kling</author>
    <date>2005-07-25T17:46:20.434104Z</date>
    <paths>
      <path action="M">khtml_part.cpp</path>
      <path action="M">loader.h</path>
    </paths>
    <msg>
       Do pixmap notifications when
       running ad filters.
    </msg>
  </log entry>
</log>
```

Figure 1. A Snippet of kdelibs Subversion Log

A number of approaches in the MSR community [15, 21, 32, 35] have utilized change-sets from versions-control systems to uncover evolutionary patterns or co-changes in source code. Similar approach could be adopted to uncover traceability links between source code and other types of documents. However, for such an approach to work well, it is necessary to determine if this characteristic is exhibited in software repositories (and to what extent).

Table 1. Six open source systems analyzed for different types of artifacts in change-sets.

System	Period	Total Change-sets	Sample Change-sets
Apache-httpd	[2005-05-01 2007-01-05)	833	84
GCC	[2001-01-01 2007-01-05)	13507	1351
jEdit	[2001-01-01 2007-01-05)	2074	208
Kdelibs	[2005-05-01 2007-01-05)	10339	10334
Koffice	[2005-05-01 2007-01-05)	11767	1177
Python	[2001-01-01 2007-01-05)	3622	363

3.2. Analyses of Change-sets for File Types

In order to establish whether different types of documents are committed in the same change-set six open source systems are examined. These systems cover a number of application domains and are primarily written in *C*, *C++*, and *Java*. *Apache httpd* is a web server, *jEdit* is an editor, *GCC* is a compiler, *koffice* is an office-applications suite, *kdelibs* is a core library for *KDE* (K Desktop Environment), and *Python* is a programming language. All these projects use *Subversion* for managing their repositories. Change-sets committed in periods between one and six years were considered. In some cases only recent history of about a year and half was considered to mitigate the influence of "old" changes that may be irrelevant for the current state of the system and its further evolution. We selected 10% of these change-sets via random sampling. These sampled change-sets were analyzed for the number of different types of artifacts in them.

Figure 2 shows the frequency distribution of the number of different file types with regards to the number of change-sets in the change-set samples. These file types include the ones discussed in Section 3. Our analysis indicates that a substantial proportion of change-sets contain two or more file types in these systems. Change-sets with a fewer number of different file types occur more frequently than those with a larger number of different file types.

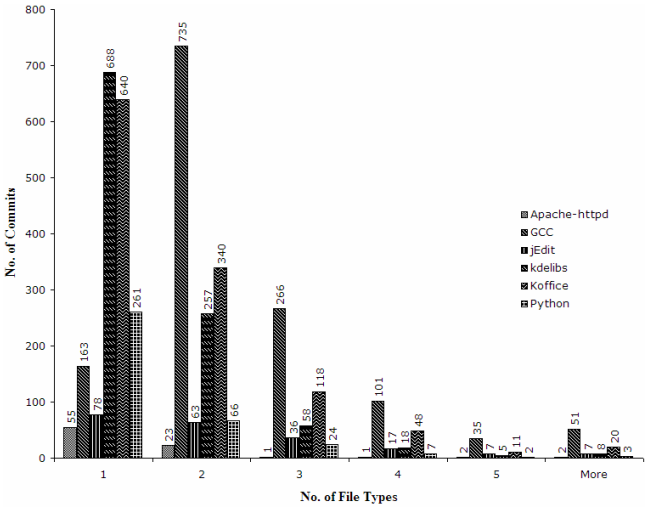

Figure 2. Histogram of the number of different file types with regards to the number of change-sets (i.e., commits) in the six open source systems. It shows two or more file types substantially co-occur in the change-sets in all these systems.

Table 2 provides some descriptive statistics of the sampled change-sets. The proportions of the change-sets with two or more file types (column *Proportion*) are between 28% and 62%. The sample means (column *Mean*) and standard deviations (column *SD*) are also given. Both these measures show that on average a change-set contains more than one file type, and as high as three file types. The standard deviations indicate that change-sets with a number of different file types also appear. We performed an outlier analysis via Inter Quartile Range (IQR) computation to determine the change-sets that deviate from a typically "normal" case (an outlier). That is, they contain a large number of different types than what is typically observed. The limits on the number of different types in a change-set beyond which it can be considered

as a suspect for an outlier are also determined from the samples (Column *Outlier Cut-off*). This analysis suggests that a change-set with the maximum range [3, 6] can well be the "normal" case.

To give an indication as to how well the statistics on the samples represent parametric means of all the change-set (i.e., within and beyond the history period considered for selecting the samples) in these projects, we provide the confidence intervals (column *CI*) for estimating the overall means. The confidence intervals were computed with the confidence level of 95%. That is, we can say with 95% confidence that the mean of all the change-sets in a project will be within the given bounds. As can be seen, the bounds do not vary much the overall mean from the sample means for all the six systems.

The analysis presented in this section shows that there are change-sets with more than one document type in software repositories. Utilizing this information to infer potential traceability link between them is a two-fold issue: 1) Is the presence of different types of documents in the same (and single) commit enough to infer the traceability links between them? 2) How do we account for related documents with potential traceability links committed in a series of multiple change-sets? We now present a heuristic-based approach that uses data mining methodology that addresses theses issues.

Table 2. Statistics of the change-sets analyzed for different types of files in six open source systems

System	Proportion	Mean	SD	Outlier Cut-off	CI
Apache-httpd	34%	1.821	3.0858	3.5	±0.6599
GCC	32%	2.532	1.6547	4.5	±0.0882
jEdit	62%	2.212	1.3489	6.0	±0.1833
kdelibs	33%	1.487	1.0689	3.5	±0.0652
koffice	45%	1.793	1.6448	3.5	±0.0934
Python	28%	1.441	0.9065	3.5	±0.0933

3.3. Mining Ordered Patterns

Our approach is to analyze sets of files that frequently co-occur in change-sets by applying a frequent-pattern mining technique (i.e., sequential pattern mining). We refer to such a set of files as a *change pattern*. These change patterns are then analyzed to uncover patterns that contain source code files and other types of files (e.g., *makefiles, TODO, change logs, HTML/XML/docbook*). We refer to such a pattern as *traceability pattern*. Our hypothesis is that if two files, of different types, co-change with a high frequency than there is a potential traceability link between them.

Software is inherently structured with dependencies among its entities such as call, control, and data dependencies. The task of performing a software change is either planned (e.g., a standard refactoring or a fix for a documented bug), unplanned activity (e.g., fixing an unforeseen side effect due to a change), or a combination of both. A typical planned change is implemented in small increments with the goal of maintaining the overall system in a coherent state (e.g., preserve the build or compile-able state, change source code and documentation in separate steps). These incremental changes corresponding to the change-sets are implicitly ordered. However, such is the nature of software that an extremely well planned change may lead to further unanticipated changes. It is not uncommon to have a bug-fix that introduces a multitude of additional bugs. Often such bugs are discovered only after a fix is committed to a repository and tested by other contributors. Nonetheless, in any case there is a temporal ordering between various change-sets.

Preservation of a change-set as an atomic commit in software repository gives the ability to iterate through the change history at the change-set level (i.e., "undo" at the change-set level rather than the individual file level). This encourages the practice of committing a set of related changes in a single logical change – a standard *Subversion* policy of the *KDE* project. However, the granularity and composition of a change-set may vary across tasks, developers, and projects. For example, consider a refactoring task that requires a series of steps such as *extract method*, *move method*, and so forth. A change-set may correspond to each elementary step or the entire refactoring. In other cases, changes to source code and related documents may be committed at different times even though they represent the same logical change. Therefore, a single high-level change may be completed over multiple change-sets.

In order to mine larger or more complete patterns we need to consider changes that spread over a sequence of change-sets. However, the changes-sets corresponding to such changes are rarely explicit (at least not directly recorded in the software repositories or clearly documented). Notice that the change-sets stored as atomic commits in software repositories are serialized. The order in which log entries appear in the log files is at the discretion of a version-control system. Two unrelated change-sets committed approximately at the same time may appear next to each other. Therefore, treating successive change-sets in the software repositories as related to a single high-level change may be meaningless.

In our approach we use three heuristics to group change-sets. Each heuristic takes a set of change-sets and forms groups of "related" change-sets. From the discussion in Section 3.1, there is a temporal relationship between change-sets. Therefore, each group formed by heuristics is actually a sequence of change-sets. We employ sequential-pattern mining to uncover ordered change patterns from the groups formed by a grouping heuristic. The transactions are the groups (i.e., sequence of change-sets) and the items are the files. The ordered patterns discovered by sequential-pattern mining are the sequences of files (actually a sequence of sets of files) that are found common in at least a user-specified number of groups (i.e., minimum support).

In general an ordered pattern is composed of elements. Each element is composed of unordered items. The ordering of elements imposes a partial order on the items. For example, the ordered pattern *{f1, f2}→{f3, f4}→{f5}* is composed of three elements and five items. It indicates that the element *{f1, f2}* happens before the element *{f3, f4}* and the element *{f3, f4}* happens before the element *{f5}*. However, the happens-before relation between items *f3* and *f4* is unknown in the element *{f3, f4}*. In the context of ordered change patterns, an element in an ordered pattern corresponds to a subset of files changed in a change-set and an item in an element corresponds to a file. Therefore, files in the same element of an ordered pattern indicate files that are likely to change in the same change-set, whereas files in the different elements of an ordered pattern indicate files that are likely to change in different change-sets in the specified order.

For the sake of brevity, ordered change patterns are referred as ordered patterns for the remainder of this discussion.

The support of an ordered pattern is the number of groups in which it occurs. An ordered pattern indicates that if any of its constituent files are found in a change-set then the rest of the files are also likely to occur in the same or different change-set as per their ordering in the pattern. Therefore, an ordered pattern in the context of a software repository could mean a set of files that are likely to be committed in the same revision before a set of files committed in the previous revision.

3.4. Change-set Grouping Heuristics

We present a number of heuristics for grouping related change-sets formed from version history metadata found in software repositories (i.e., developer, time, and changed files). These heuristics can be considered similar to the fixed and sliding window techniques [15, 17, 35]. These techniques are used to group changed files into a single change-set typically applied to CVS repositories as they lose the atomicity of original change-sets. Our heuristics combine change-sets into groups in order to account for related changes committed across multiple change-sets.

3.4.1. Time Interval

This grouping heuristic is based on the premise that the change-sets committed during a given time-interval are related, and change-sets committed outside this interval are unrelated. All the change-sets committed in a given time duration are placed in a single group. The number of groups is equal to the number of time intervals over which the change-sets were committed. This heuristic covers related change-sets that are committed by different developers but during the same time interval. The ordered patterns found using this heuristic implies that if a file is modified in a particular pattern on a given day, the following (or preceding) files are likely to be modified on the same day.

For example, the pattern *{khtml_part.h}* → *{ChangeLog}* was found from mining the change-sets in the KDE *Subversion* repository (under *kdelibs/khtml/*) committed between May 2005 and December 2005. In this case, a group in this case was formed for the change-sets committed in one calendar day. This pattern is found to occur in five groups. On each of these five days, the file *khtml_part.h* was in a change-set that was committed before the change-set in which the file *ChangeLog* was committed. This is a traceability pattern showing that changes are documented after an interface file is changed. The pattern

{kdeedu/kalzium/src/kalzium.cpp, kdeedu/kalzium/src/pse.cpp} → *{kdesdk/doc/scripts/kdesvn-build/index.docbook}*

is another example pattern that occur in change-sets committed in each of five different days. This pattern shows that the documentation is updated after performing changes to the source code. However, the order in which the two source code files were changed cannot be determined (i.e., a partially ordered pattern).

3.4.2. Committer

This heuristic is based on the premise that the change-sets committed by the same developer are related and the change-sets committed by different committers are unrelated. This defines an order on the change-sets by a committer. Therefore, all the change-sets committed by a given committer are placed in a single group.

The number of groups is equal to the number of unique committers. This heuristic covers related change-sets that are committed in different time intervals but by the same author. The ordered pattern found using this heuristic implies that if a file is modified in a pattern by a committer, the following (or preceding) files are likely to be modified by the same committer.

The pattern *{khtml_part.h}→{ChangeLog}* was found from mining the change-sets in the KDE *Subversion* repository committed between May 2005 and December 2005. A group in this case was formed for the change-sets committed by the same developer. This pattern is found to occur in five groups. In the case of each committer, the file *kdelibs/khtml/khtml_part.h* was in a change-set that was committed before the change-set in which the file *kdelibs/khtml/ChangeLog* was committed. The same pattern was found by grouping change-sets by the heuristic Time interval (see Section 3.4.1). This further strengthens that this is a change dependency between these artifacts and not an unrelated dependency due to a development practice of a developer or unusual changes made during a particular day. The pattern

{kdeedu/kalzium/src/kalziumtip.cpp} → *{kdeedu/kalzium/src/detailinfodlg.cpp}* → *{kdeedu/kalzium/src/Makefile.am}* → *{kdeedu/kalzium/src/kalzium.cpp, kdeedu/kalzium/src/kalzium.h}*

is another example pattern that is found in the change-sets committed by five developers. This pattern shows that a build file is updated both before and after changing the source code.

3.4.3. Committer + Time Interval

This heuristic is based on the premise that the change-sets committed by a committer in the same time are related, and the change-sets committed by the same or different committers in different time intervals are unrelated. This defines an order on the change-sets by a committer. Therefore, all the change-sets committed by the same committer within the same time interval are placed in a single group. The number of groups is equal to the number of unique committers and time interval combinations. This heuristic restricts related change-sets to the change-sets committed by an author in a time period. The ordered pattern found using this heuristic implies that if a file is modified in a pattern by a committer the following (or preceding) files are likely to be modified by the same committer in the same time interval.

For example, the pattern *{TODO}→{pse.cpp}* was found from mining the change-sets in the KDE *Subversion* repository committed between May 2005 and December 2005. A group in this case was formed for the change-sets committed by the same developer on the same calendar day. This pattern is found to occur in ten groups. In each combination of committer and day, the file *kdeedu/kalzium/TODO* was in a change-set that was committed before the change-set in which the file *kdeedu/kalzium/src/pse.cpp* was committed. The pattern

{kdeedu/kalzium/src/kalziumui.rc} → *{kdeedu/kalzium/src/pse.h, kdeedu/kalzium/src/pse.cpp}*

is another example pattern that is found in the change-sets committed by seven different committer-day combination. This

pattern shows that a particular user-interface file is changed before modifying the code.

3.5. Frequent Pattern-Mining Tool

We have developed a sequential pattern-mining tool, namely *sqminer*, that is based on the Sequential Pattern Discovery Algorithm (SPADE) [33] which utilizes an efficient enumeration of ordered patterns based on common-prefix subsequences and division of search space using equivalence classes. Additionally, it utilizes a vertical input-transaction format (i.e., a set of transactions for each file vs. a set of transactions consisting of files) for efficiency.

To help prune the number of candidate patterns produced by the mining techniques, patterns with redundant information are eliminated. A pattern that is frequent means that all possible patterns formed from the subsets of its files are also frequent. The support of a pattern is always less than or equal to the subset patterns. A common pruning mechanism used in frequent-pattern mining is to eliminate all the subset patterns that have the same support of the corresponding larger pattern. Such subset patterns are only used with other larger patterns and not in isolation. Therefore, they give redundant information that may be of very little meaning. As a result, only disjoint patterns (i.e., patterns with no common files) that subsume all subsets of patterns with the same or higher support are retained. Such patterns are known as *closed* patterns. Our tool produces only closed patterns.

Frequent-pattern mining algorithms typically report the support of a pattern but not the transactions in which it occurs. Our tool records the transactions in which a pattern is found. For uncovering both unordered and ordered change patterns, we use the same underlying mining algorithm. The tool *sqminer* can also be used for frequent itemset mining. In this case the transactions are formed with no ordering information of items. The configuration parameters of *sqminer* include support, maximum number of items in a pattern, mining of sequence (association) rules, and output in both a flat-file and XML format. For further detail on the XML output format of the ordered patterns and rules, we refer to [21].

4. TRACEABILITY CHALLENGES

We believe that our approach provides at least partial answers to a subset of the many grand challenges identified in the area of traceability research [1]. Our approach is based on the evolutionary information (i.e., actual changes) recorded in the software repositories. The traceability links recovered could be used as a validation mechanism for traditional approaches (in addition to manual efforts). Therefore, versions history can be a useful source for establishing benchmarks in traceability (L-GC2).

Software repositories are managed by versions-control systems that are integrated in modern development environments (e.g., *Eclipse* and *KDevelop*). We believe that our approach can seamlessly operate with such tools to uncover and enforce traceability links before or after a change is committed (e.g., as a *pre-hook* or *post-hook* monitor). This directly addresses the issue of tools that support incremental traceability recovery and integration with other development tools (C-GC2).

The approach that is based on evolutionary couplings, such as ours, can be used to a certain extent in establishing dependencies between artifacts without even analyzing their actual contents. This provides a starting point (minimally) in uncovering possible traceability links across heterogeneous document types including those that are semi-structured, binaries, and graphics files (C-GC3 and E-GC3.2).

5. RELATED WORK

There are two distinct areas of research that are directly related to the work presented here, namely mining software repositories and traceability-link recovery.

We briefly discuss a few early approaches utilizing information found in source code repositories maintained by tools such as *CVS* and *Subversion* with a focus on co-changes and analysis. Zimmerman et al [34, 35] used *CVS* logs for detecting evolutionary coupling between source code entities. They employed sliding window heuristics to estimate the atomic commits (change-sets). Association-rules based on itemset mining were formed from the change-sets and used for change-prediction. Yang et al [32] used a similar technique for identifying files that frequently change together. Gall et al [15] used window-based heuristics on *CVS* logs for uncovering logical couplings and change patterns, and German et al [16] for studying characteristics of different types of changes. Hassan et al [19] analyzed *CVS* logs for software-change prediction.

The work presented in this paper is closely related to the works by Zimmermann et al [34, 35] and Canfora et al [5]. However our work has important distinctions. Zimmermann et al [34, 35] focused on uncovering source-code-to-source-code change dependencies using itemset mining. They considered only the files committed together in a single change-set (approximated via sliding-window technique). Our focus is on uncovering traceability links between source code and other types of artifacts (note that we also uncover source-code-to-source-code change dependencies). We also consider files committed over a sequence of change-sets (and not just a single change-set). We use sequential-pattern mining to uncover the ordering information of committed files (and not just unordered sets of files). Yang et al [32] used a similar technique as Zimmermann et al [34, 35] for identifying files that frequently change together. Canfora et al [5] work is based on the textual similarity of different bug reports (and commit messages) in the change history. An information-retrieval method is used to index the changed files in the *CVS* repositories with the textual description of past bug reports in the *Bugzilla* repository and the *CVS* commit messages. Our work is based on a common set of files that is changed multiple times in the change history. As such, their work is dependent on the "quality" of the textual description. Additionally, they are only able to find traceability between bugs/features and source code files for those bugs/features entered in the bug-tracking system. However, we found that there are many instances in large open source projects (e.g., KDE) where a feature/bug is implemented without a corresponding entry in the bug-tracking system. Our approach handles such situations.

Spanoudakis and Zisman conducted a comprehensive study of various methods for link recovery in [29]. These methods utilize such things as information retrieval, test-cases, and design patterns. Antoniol et al. [2] recover links between source code and documentation using a probabilistic and a vector space IR models. Marcus and Maletic [24] use another IR technique namely, Latent Semantic Indexing (LSI) to recover links from documentation to source code on the same set of case studies done

by Antoniol et al with better precision. A number of other researchers [11, 12, 20, 22, 23, 27, 31] have also applied IR methods for traceability link recovery. The results of these studies demonstrate the usefulness of IR methods for link recovery; however the approaches do not consider, or depend on, multiple versions of a software system to construct the links. In one of the rare studies that examined multiple versions, Antoniol et al. [3] establish traceability links between software releases of an object oriented system to determine inconsistencies. However, their work only looks at two versions at a time.

Egyed takes a scenario based semi-automated approach to uncover traceability information between software artifacts [13, 14]. Test cases are used to generate trace information produced during program execution. Spanoudakis and Zisman [30, 36] use heuristics for automatic generation of traceability links between requirements and the UML object model as well as between different parts of a requirements document. Cleland- Huang et al. [6-8] propose an event based traceability approach in establishing traceability links between requirements and performance models using an event-notifier design pattern. Murphy et al. [25] introduce software reflexion models to automatically identify links between high level models and source code.

6. CONCLUSIONS AND FUTURE WORK

We empirically analyzed six open source projects to make the case for software repositories as a potential source for uncovering traceability links between various types of software artifacts. This includes source code from/to user documentation traceability links. A heuristic based approach that uses frequent-pattern mining is presented as one such effort. An uncovered ordered pattern gives a traceability link between multiple documents with the ordering information implying the potential directionality. Our work compounded with the existing approaches in uncovering traceability between requests in bug repositories and source code expands the horizons of traceability research via mining software repositories and overall generally. While the discussion here may seem restricted to the open source development, we believe that our approach is equally applicable in any other development methodology that exhibits different types of artifacts in the same change-sets.

In future we plan to evaluate the "goodness" of our approach. The general evaluation methodology will be to first mine a portion of the version history for traceability patterns. Then mine a later part of the version history and see how accurately the prior traceability patterns hold. Additional heuristics for grouping related change-sets such as textual similarity of commit messages are also investigated. Also, we are working on recovering fine-grained traceability link (e.g., at class and method levels) by leveraging *srcML* infrastructure and standard differencing tools (e.g., *diff*).

7. REFERENCES

[1] Antoniol, G., Berenbach, B., Eyged, A., Ferguson, S., Maletic, J., and Zisman, A. Problem Statements and Grand Challenges. Center of Excellence for Traceability, Lexington, KY September, 10 2006.

[2] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, 28, 10 (October 2002), 970-983.

[3] Antoniol, G., Canfora, G., and Lucia, A. D. Maintaining Traceability During Object-Oriented Software Evolution: A Case Study in Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM) (August, 1999), 211-219.

[4] Antoniol, G., Canfora, G., Lucia, A. D., and Merlo, E. Recovering Code to Documentation Links in Object-oriented Systems in Proceedings of 6th IEEE Working Conference on Reverse Engineering (Atlanta, Georgia, October 1999, 1999), 136-144.

[5] Canfora, G. and Cerulo, L. Impact Analysis by Mining Software and Change Request Repositories in Proceedings of 11th IEEE International Symposium on Software Metrics (METRICS'05) (Como, Italy, September, 19-22 2005), 29-37.

[6] Cleland-Huang, J., Chang, C. K., Sethi, G., Javvaji, K., Hu, H., and Xia, J. Automating Speculative Queries through Event-Based Requirements Traceability in Proceedings of IEEE Joint International Conference on Requirements Engineering (RE) (Sept, 2002), 289-296.

[7] Cleland-Huang, J. and Chang, K. C. Supporting Event Based Traceability through High-Level Recognition of Change Events in Proceedings of 26th Annual International Computer Software and Applications Conference (Oxford, England, Aug, 2002), 595-600.

[8] Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhan, E., and Christina, S. Goal Centric Traceability for Managing Non-Functional Requirements in Proceedings of International Conference on Software Engineering (ICSE) (St Louis, USA, May, 2005), 362-371.

[9] Cubranic, D. and Murphy, G. C. Hipikat: Recommending Pertinent Software Development Artifacts in Proceedings of 25th International Conference on Software Engineering (ICSE'03) (Portland, Oregon, May 6-8, 2003), 408-418.

[10] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S. Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering*, 31, 6 (2005), 446-465.

[11] DeLucia, A., Fasano, F., Oliveto, R., and Tortora, G. Enhancing an Artefact Management System with Traceability Recovery Features in Proceedings of 20th IEEE International Conference on Software Maintenance (Chicago, Illinois, September, 2004), 306-315.

[12] DeLucia, A., Fasano, F., Oliveto, R., and Tortora, G. Can Information Retrieval Techniques Effectively Support Traceability Link Recovery? in Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC'06) (Athens, Greece, June 14-16 2006), 307-316.

[13] Egyed, A. A Scenario-Driven Approach to Traceability in Proceedings of 23rd International Conference on Software Engineering (ICSE) (Toronto, Canada, May, 2001), 123-132.

[14] Egyed, A. A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Transactions on Software Engineering*, 29, 2 (2004), 116-132.

[15] Gall, H., Hajek, K., and Jazayeri, M. Detection of Logical Coupling Based on Product Release History in Proceedings of 14th IEEE International Conference on Software Maintenance (ICSM'98) (Bethesda, Maryland, November, 16-20, 1998), 190-199.

[16] German, D. M. An Empirical Study of Fine-Grained Software Modifications in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September, 11-17, 2004), 316-325.

[17] German, D. M. Mining CVS Repositories, the SoftChange Experience in Proceedings of 1st International Workshop on Mining Software Repositories (MSR'04) (Edinburph, Scotland, 2004), 17-21.

[18] Gotel, O. C. Z. and Finkelstein, A. C. W. An Analysis of the Requirements Traceability Problem in Proceedings of 1st IEEE International Conference on Requirments Engineering (Colorado Springs, April, 1993), 94-101.

[19] Hassan, A. E. and Holt, R. C. Predicting Change Propagation in Software Systems in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinios, September, 11-17, 2004), 284-293.

[20] Hayes, J. H., Dekhtyar, A., and Osborne, J. Improving Requirements Tracing via Information Retrieval in Proceedings of 11th IEEE International Requirements Engineering Conference (RE) (Washington, D.C, USA, September, 2003), 138-147.

[21] Kagdi, H., Yusuf, S., and Maletic, J. I. Mining Sequences of Changed-files from Version Histories in Proceedings of 3rd International Workshop on Mining Software Repositories (MSR'06) (Shanghai, China, May 22-23, 2006, 2006), 47-53.

[22] Lormans, M. and Van Deursen, A. Reconstructing Requirements Coverage Views from Design and Test using Traceability Recovery via LSI in Proceedings of 3rd ACM International Workshop on Traceability in Emerging Forms Of Software Engineering (Long Beach, California, USA, Nov 8th, 2005), 37-42.

[23] Lormans, M. and Van Duersen, A. Can LSI help Reconstructing Requirements Traceability in Design and Test? in Proceedings of Conference on Software Maintenance and Reengineering (CSMR'06) (March, 2006), 47-56.

[24] Marcus, A. and Maletic, J. I. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing in Proceedings of 25th IEEE/ACM International Conference on Software Engineering (ICSE'03) (Portland, OR, May 3-10, 2003), 125-137.

[25] Murphy, G. C., Notkin, D., and Sullivan, K. Software Reflexion Models: Bridging the Gap between Source and High-Level Models in Proceedings of 3rd ACM Symposium on Foundations of Software Engineering (New York, NY, October, 1995), 18-28.

[26] Scacchi, W. Understanding the Requirements for Developing Open Source Software Systems. *IEE Proceedings--Software*, 149, 1 (February 2002), 24-39.

[27] Settimi, R., Cleland-Huang, J., Khadra, O. B., Mody, J., Lukasik, W., and DePalma, C. Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts in Proceedings of 7th International Workshop on Principles of Software Evolution (IWPSE) (Kyoto, Japan, Sept 6-7, 2004), 49-54.

[28] Sliwerski, J., Zimmermann, T., and Zeller, A. When do changes induce fixes? in Proceedings of 2nd International Workshop on Mining Software Repositories (MSR'05) (St. Louis, Missouri May 17, 2005), 24-28.

[29] Spanoudakis, G. and Zisman, A., "Software Traceability: A Roadmap", in *Handbook of Software Engineering and Knowledge Engineering*, Chang, S. K., Ed. World Scientific Publishing Co, 2005, pp. 395-428.

[30] Spanoudakis, G., Zisman, A., Perez-Minana, E., and Krause, P. Rule-based generation of requirements traceability relations. *The Journal of Systems and Software*, 72, 2004 (2004), 105-127.

[31] Sundaram, S. K., Hayes, J. H., and Dekhtyar, A. Baselines in Requirements Tracing in Proceedings of Predictor models in software engineering (PROMISE'05) (St. Louis, Missouri, USA, 2004), 1-6.

[32] Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering*, 30, 9 (September 2004), 574-586

[33] Zaki, M. J. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning*, 42, 1-2 (January 2001), 31 - 60.

[34] Zimmermann, T., Weißgerber, P., Diehl, S., and Zeller, A. Mining Version Histories to Guide Software Changes in Proceedings of 26th International Conference on Software Engineering (ICSE'04) (2004), 563-572.

[35] Zimmermann, T., Zeller, A., Weißgerber, P., and Diehl, S. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31, 6 (2005), 429-445.

[36] Zisman, A., Spanoudakis, G., Perez-Minana, E., and Krause, P. Tracing Software Requirements Artifacts in Proceedings of 2003 International Conference on Software Engineering Research and Practice (SERP'03) (Las Vegas, Nevada, USA, 2003), 448-455.