Mining Software Repositories for Traceability Links

Huzefa Kagdi, Jonathan I. Maletic, Bonita Sharif
Department of Computer Science
Kent State University
Kent Ohio 44242
{hkagdi, jmaletic, bsimoes}@cs.kent.edu

Abstract

An approach to recover/discover traceability links between software artifacts via the examination of a software system's version history is presented. A heuristic-based approach that uses sequential-pattern mining is applied to the commits in software repositories for uncovering highly frequent co-changing sets of artifacts (e.g., source code and documentation). If different types of files are committed together with high frequency then there is a high probability that they have a traceability link between them. The approach is evaluated on a number of versions of the open source system KDE. As a validation step, the discovered links are used to predict similar changes in the newer versions of the same system. The results show highly precision predictions of certain types of traceability links.

1. Introduction

Traceability link recovery has been a subject of investigation for many years within the software engineering community [12, 24]. Having explicit documented traceability links between various artifacts (e.g., source code and documentation) is vital for a variety of software maintenance tasks including impact analysis, program comprehension, and requirements assurance of high quality systems. It is particularly useful to support source code comprehension if links exist between the source code, design, and requirements documentation. These links help explain why a particular function or class exists in the program.

A number of techniques have been proposed to assist in the recovery/discovery of traceability links in existing software systems [24]. However, the problem has proven to be quite difficult and no single approach is, by itself, completely successful or accurate. Many approaches suffer from producing many false positives, suggesting a link when none should exist [2, 17, 18].

Approaches to recovering/discovering traceability links typically analyze a single snapshot (i.e., current version) of a software system to infer links between two or more artifacts. The method presented here to uncovering traceability links differs in that it examines multiple versions (i.e., change history) of the software artifacts to recover traceability links. In [15] we developed the premise that if artifacts of different types (e.g., *src.cpp* and *help.doc*) are co-changed with high frequency over multiple versions, then such artifacts potentially have a traceability link between them.

This history-based approach offers some distinct advantages over single version approaches, namely:

- Traceability links are derived from the actual changes to artifacts, rather than estimations that are based on the analysis of various structural and semantic dependencies between them in a single snapshot of a system.
- The change history represents one of the few sources of information available for recovering traceability links that is manually created and maintained by the actual developers. The commits, and commit messages embody part of the developer's knowledge and experience. The version history may contain domain-specific "hidden" links that program-analysis methods fail to uncover.

To accomplish this, we mine the version history found in software repositories that are maintained by version-control tools such as *Subversion* or *CVS*. Specifically, we use the data mining technique, *sequential-pattern mining* [1], to identify and analyze sets of files that are committed together. This technique produces frequently co-changed files, otherwise known as *change patterns*. Change patterns that include files representing different types of artifacts are considered related via a traceability link. Another approach, *itemset mining* [11], has also been used for mining version histories. Itemset mining produces change patterns that are unordered sets of co-changing artifacts; whereas sequential-pattern mining produces ordered (actually partially ordered) lists of co-changing artifacts. That is, the order in which the artifacts were changed (or committed) is preserved. The ordering information can thus be used to infer directionality of the traceability links.

Our approach is evaluated on the open-source software system *KDE* (K Desktop Environment). The results show that our approach is able to uncover traceability links between various types of software

artifacts (e.g., source code files, change logs, user documentation, and build files) with high accuracy. This approach can readily be applied, in conjunction with other link recovery methods, to produce a more complete picture of the traceability of a software system.

The paper is organized as follows. In Section 2 we discuss traceability in the context of open source development. In Section 3 we discuss our approach to mining traceability patterns from software repositories and our mining tool. The evaluation of the approach is given in Section 4. Related work and conclusions follow that in Sections 5 and 6.

2. Open source and Traceability

Scacchi et al. [21] 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, HowTo 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 project artifacts along with the primary means of coordination and archival.

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. 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. [4] 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 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. [23] 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. [6] describes a tool, namely *Hipikat*, to assist new developers (not necessarily novice) on a project, in performing their current task(s). *Hipikat* recommends artifacts from the project memory that may hold relevance to a task at hand. A developer may ask for the relevant artifacts explicitly in the form of 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 portion of the broad spectrum of documents found in open-source development. 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.

3. Uncovering Traceability Links

Our research interest is in uncovering traceability between source code and other artifacts. 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*, *README*, and *INSTALL* guides), progress monitoring documents (*TODO* and *STATUS*), and extensible mechanisms (e.g., *Python*, *Ruby*, and *Pearl* bindings for an *API*). These artifacts can be considered software informalisms [21]

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. We refer to such a pattern as a *traceability pattern*. Our hypothesis is that if the same set of files, of different

types, co-change with a high frequency then there is a potential traceability link between them. For this to be a sound hypothesis, the basic prerequisite is to examine if different types of files are typically changed together. Our study on six open source systems [15] shows that between 28% and 62% of the change-sets (i.e., a set of files checked into a software repository together in a single commit operation) contain two or more types of artifacts. These systems cover a number of application domains, sizes, 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. We believe such change-sets are a valuable source for uncovering traceability links. Using this source, an approach that discovers a specific set of files with traceability links between them can be devised. We first describe how change-sets are stored and represented in software repositories to help facilitate the 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 a log entry.

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. This information can be readily obtained in an XML format by using the command–line client *svn log*. *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 *loader.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 in 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 in *Subversion* repositories. Let us now describe relevant data-mining terminology and the frequent-pattern mining techniques.

```
<?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

3.2. Data Mining Background

The input data to frequent-pattern mining algorithms are in the form of transactions. A transaction refers to a group of items that share a common property or occur in the same event (e.g., customer baskets or items checked-out together in market-basket analysis). The number of transactions in which a pattern occurs is known as its *support*. The support of a pattern is the number of groups in which it appears. So if the support of a pattern is at least a user-specified *minimum support* then it is a *frequent* pattern in the considered dataset. Sequential-pattern mining produces a partially ordered list of files for patterns and as such we term these *ordered patterns*.

Sequential-pattern mining takes a set of sequences and finds all the frequently occurring subsequences (i.e., ordered patterns) that have at least a user-specified minimum support [19]. Here, transactions are in the form of sequences of items. Sequential-pattern mining techniques are typically applied to datasets with temporal or other ordering information. For example, in case of market-basket analysis with the additional timestamp information, sequential patterns such as customers who bought a *camera* are also likely to buy *additional memory* in the next month.

3.3. Mining Ordered Patterns

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 in 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 three 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 [9, 10, 29]. 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 (unlike change-sets in *Subversion*). 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 unique 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 within a given time interval, 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 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 a single 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 the same committer within a time interval 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 a given 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 within a specific 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) [28] 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 by themselves. 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 [16].

4. Evaluation

To evaluate our approach to recovering traceability links we use the open-source system *KDE* (www.kde.org). *KDE* has over 4 million LOC. The *KDE* repository (websvn.kde.org/trunk/KDE) houses around twenty different modules, each containing multiple applications and libraries. The applications in *KDE* represent a wide spectrum of domains, programming languages, size, and developers.

The evaluation methodology is to first mine a portion of the version history for traceability patterns. We call this the training-set. Next we mine a later part of the version history (called the evaluation-set) and see if the results generated from the training-set can accurately predict changes that occur in the evaluation-set.

We considered the change-sets committed in a twelve-month period in the *KDE* repository from 2005-05-01 to 2006-04-30. *KDE* migrated from *CVS* to *Subversion* around May 2005 and this was our primary reason for picking this particular time frame. We allocated 2/3rd of this version history to the training-set: an eight-month period of history starting at 2005-05-01 and ending on 2005-12-26. This training-set contains 14,939 change-sets consisting of 13,037 files. The remainder was allocated to the evaluation-set: a four-month period of history starting at 2005-12-27 and ending at 2006-04-30. This evaluation-set contains 9,008 revisions (i.e., change sets) consisting of 9,070 files. Only change-sets that consisted of ten files or less are considered. This avoids change-sets such as updating the license information on every file or performing merging and copying.

First we need to extract relevant information from the *KDE* repository. A straightforward approach to extract the log entries from a *Subversion* repository is to use the client command *svn log* from a working copy of the repository. This approach is not feasible for use-cases in which only the logs stored in software repositories are needed and not the contents of the committed documents. We developed the tool *changeextractor* that uses *pysvn* (a *Subversion* module for Python) to extract changesets, without using a working copy, from the repository. *changeextractor* takes a repository URL, a start date, and an end date of a history, and extracts the change-sets from the repository logs for a specified period.

The change-sets extracted from the repository are then grouped into sequences according to the grouping heuristics by the tool *groupchanges* (another Python script). We choose a calendar day as the time interval for the heuristic *Time Interval*. A committer that contributed a change is mapped to a group for the heuristic *Committer*. The number of groups and the total number of change-sets involved in these groups for the three heuristics are shown in Table 1.

Heuristics	Groups
Day	237
Committer	330
CommitterDay	4,884

Table 1. Groups formed from the change-sets extracted from the *KDE* repository by the different heuristics.

Heuristics	CP	TP	TP/CP%
Day	5,839	1,851	37.10
Committer	718	54	7.52
CommitterDay	2,372	277	11.68

Table 2. Change patterns (CP) and traceability patterns (TP) uncovered from the *KDE* repository. All patterns have a minimum support of five.

4.1. Uncovering Traceability Patterns

The groups constructed by the tool *groupchanges* are fed to the sequential-pattern mining tool *sqminer*. Mining frequent ordered patterns with *sqminer* produces a set of closed change patterns. We configured *sqminer* to mine patterns with a minimum support of five for all our defined heuristics.

Table 2 shows the uncovered change patterns and the traceability patterns uncovered by the heuristics. The

traceability pattern consists of at least one source code file and at least one file of another type. C++ is the primary programming language for *KDE*. The files with extensions {.h, .cpp, .cc, .c, .hxx, .cxx } were considered to be containing source code, whereas those that are not source code files are considered as other artifacts (e.g., with extensions .docbook, .xml, and .html, and ChangeLog). We do not restrict other artifacts to a particular set of types; rather we discover them.

The heuristic *Committer* uncovered the minimum number of change patterns, traceability patterns, and percentage of the traceability patterns in the change patterns. The heuristic *Day* uncovered the maximum number of change patterns, traceability patterns, and percentage of the traceability patterns.

The traceability patterns mined are not restricted to binary patterns. Table 3 shows the minimum, maximum, and average number of files in the traceability patterns uncovered from the training-set. Traceability patterns with as many as seven files were uncovered with the heuristic *CommitterDay*. Now that our approach is able to find potential traceability patterns, a measure of the "goodness" is needed to evaluate the traceability patterns. We validate our approach with three metrics.

Heuristic Queries	Number of files		
	Min	Max	Avg
Day	2	6	2.83
Committer	2	5	3.30
CommitterDay	2	7	3.34

Table 3. The minimum, maximum, and average number of files in the traceability patterns in the training-set

4.2. Validation Metrics

We refer to the traceability patterns in Table 2 as the *training-set* and the traceability patterns in Table 4 as the *evaluation-set*. The usefulness of the uncovered traceability patterns can be seen in "how well" the training-set predicts the existence of a traceability pattern in the evaluation-set. We validate this using the metrics *coverage*, *recall*, and *precision*.

Let $T=\{ts_1, ts_2, ...ts_m\}$ and $E=\{es_1, es_2, ...es_n\}$ be the training-set and evaluation-set respectively. Consider the pattern $es_i = \{f_1\}\rightarrow\{f_2\}\rightarrow,...,\rightarrow\{f_k\}$ in the evaluation-set that is undergoing changes. To eventually predict this pattern, the training-set can be queried for candidates after each element $\{f_i\}$ of the pattern es_i is changed (or planned to be changed). Let $Ces_i = Ces_{i1} \cup Ces_{i2} ....\cup Ces_{ik}$ where $Ces_{ij} = \{ts_1, \quad ,ts_p\}$ be a set of candidate patterns suggested from the training-set after changing the j^{th} element (and previous elements) of the pattern es_i.

<u>Definition:</u> *Covered pattern* is a traceability-pattern in the evaluation-set for which there is at least one candidate pattern suggested from the training-set,

$$Covered\ Patterns = |\{\ \forall es_i \in E \Rightarrow |Ces_i| > 0\}|$$

<u>Definition:</u> *Coverage* is the percentage of the total number of covered patterns to the total number of patterns in the evaluation-set,

$$Coverage = \frac{CoveredPatterns}{|E|} \times 100\%$$

<u>Definition:</u> *Correctly covered pattern* is a covered pattern with at least one suggested candidate pattern from the training-set that is the same (completely identical) or its sub-pattern (partially identical).

<u>Definition:</u> *Recall* is the percentage of the total number of correctly covered patterns to the total number of patterns in the evaluation-set.

$$Recall = \frac{CorrectlyCoveredPatterns}{|E|} \times 100\%$$

Coverage and recall are indicative of the completeness of the training-set in predicting the evaluation-set. Coverage describes how many traceability patterns, a developer can expect to be recommended from the patterns mined in the training-set. Recall describes how may of these recommendations are "correct". Ideally, both coverage and recall should be 100% (all patterns in the evaluation-set are correctly predicted in training-set).

Coverage and recall give only one measure of usefulness of the traceability patterns for software-change prediction. An arguably more important measure is how many total candidate patterns, both correct and incorrect, are suggested from the training-set that require examination for a covered pattern in the evaluation-set.

<u>Definition:</u> *Relevant patterns* of a covered pattern are the number of correctly covered patterns suggested after a change in its given element.

For example let $es_i = \{f_1\}\rightarrow\{f_2\}\rightarrow\{f_3\}$ be a covered pattern. If the candidate patterns suggested from the training-set are $\{f_1\}\rightarrow\{f_2\}\rightarrow\{f_3\}$, $\{f_1\}\rightarrow\{f_2\}\rightarrow\{f_4\}$, and $\{f_1\}\rightarrow\{f_5\}\rightarrow\{f6\}$ after a change in file $\{f_1\}$ in es_i, the relevant patterns are two (out of the three). After a change in the file $\{f_2\}$ in es_i (i.e., $\{f_1\}\rightarrow\{f_2\}$) the suggested candidate patterns are $\{f_1\}\rightarrow\{f_2\}\rightarrow\{f_3\}$ and $\{f_1\}\rightarrow\{f_2\}\rightarrow\{f_4\}$. The relevant pattern is one (out of two).

<u>Definition:</u> The *relevance ratio* of a covered pattern is the sum of the ratios of the number of relevant patterns over the number of suggested candidates of all its elements. The relevance ratio in our example is 2/3 + 1/2 + 1 = 2.167.

$$Relevance\ Ratio\ (es_i) = \sum \frac{relevantPatterns}{|Ces_i|}$$

<u>Definition:</u> *Precision* of a covered pattern is the percentage of relevance ratio weighted over its number of elements. Let $|es_i|$ be the number of elements in es_i,

$$Precision\ (es_i) = \frac{relevanceRatio}{|es_i|} \times 100\%$$

Precision of our example is (2.167/3) x 100 = 72%. In the best case, for any given covered pattern in the evaluation-set, only that pattern is suggested from the training-set after changes to any of its elements (i.e., precision is 100%). Using these metrics, we can evaluate our approach on the evaluation-set of our *KDE* study.

Heuristics	CP	TP	TP/CP%
Day	1112	143	12.86
Committer	304	26	8.55
CommitterDay	835	8	0.9

Table 4. Traceability patterns (TP) from the *KDE* repository that are used as evaluation data-set for the traceability patterns (TP) in Table 2. All patterns have a minimum support of five.

Heuristic Queries	Coverage (%)	Recall (%)
Day	44	8
Committer	57	11
CommitterDay	25	25

Table 5. Coverage and Recall for the evaluation-set (Table 2) compared against the training-set (Table 4).

4.3. Predicting Changes in the Evaluation-set

We use the traceability patterns in the training-set for software-change prediction on the evaluation-set. A similar process to that of analyzing the training-set was followed in mining traceability patterns from the evaluation-set. Table 4 shows the uncovered traceability patterns from the evaluation-set. A total of 177 traceability patterns were uncovered in the evaluation-set. We evaluated all these patterns with the patterns in training-set with regards to coverage, recall, and precision. The traceability patterns mined with our heuristics can be considered as representing the following types of software-change prediction questions (i.e., the complete change due to prediction will eventually form a traceability pattern):

- *Day* – If the file *f* is changed on the day *d*, what other files are likely to change on the day *d*?
- *Committer* – If the developer *c* changes the file *f*, what other files are likely to be change by *c*?
- *CommitterDay* – If the developer *c* changes the file *f1* on the day *d*, what other files will *c* likely change along with the file *f1* on the same day *d*?

Table 5 shows the coverage and recall of the traceability patterns in the evaluation-set from the traceability patterns in the training-set. The results show that the heuristic *Committer* provides the maximum coverage, however, at the cost of a lower recall. Similarly, the heuristic *Day* provides a high coverage but also at the cost of a lower recall. The heuristic *CommitterDay* provides the highest recall, however, at the cost of a very low coverage.

Table 6 shows the precision of the traceability patterns in the evaluation-set from the traceability patterns in the training-set. Only the top ten frequent patterns are recommended as candidates for an element change. The minimum, maximum, and average precision of all the traceability patterns are reported. Notice that precision is measured per pattern; coverage and recall are measured for the entire evaluation-set. The results show that the heuristic *CommitterDay* is likely to provide higher precision than the others.

Heuristic Queries	Precision (%)		
	Min	Max	Avg
Day	50	100	63
Committer	55	75	67
CommitterDay	100	100	100

Table 6. Precision for the evaluation-set (Table 2) compared against the training-set (Table 4).

The overall results show that there is no single heuristic that outperforms others in terms of coverage, recall, and precision. The heuristic *CommitterDay* is likely to produce better recall and precision. It is safe to say that any predictions made by this approach about the existence of traceability links are quite precise. The heuristics *Day* and *Commiter* overall provide reasonable coverage and precision, but recall is low.

4.4. Threats to Validity

With regards to internal validity, we are able to uncover traceability patterns from the change-sets committed in the *KDE* repository in various periods of time and sizes (a minimum of seven days) between 2005/05 and 2006/08. We also show our approach effective in recovering traceability links with only a few days (versions) of change history.

Our results are obtained from the change history of over twenty packages in the *KDE* repository. We also applied our approach to Apache *httpd* software repository and found similar results. Both systems are prime examples of successful open-source development, representing different domains and sizes but we do not claim that our results would generalize to any given system (e.g., commercially developed software). We believe that our approach is applicable to any software development with a practice of committing related files together in a software repository. Also, our validation results are within the context of only the frequent patterns that reoccur, and not for every change, in the later part of the history.

5. Related Work

There are two distinct areas of research that are directly related to our work, namely Mining Software Repositories (MSR) and traceability link recovery.

Kagdi et al. [14] present a survey of MSR approaches in the context of software evolution. Our work closely relates to the works by Zimmermann et al. [29] and Canfora et al. [4], however with important distinctions. Zimmermann et al. [29] mainly focused on uncovering source-code-to-source-code change dependencies using itemset mining. They considered only the source code entities committed together in a single change-set (approximated via sliding-window technique). Our focus is on uncovering the 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 in a single change-set. We use sequential-pattern mining to uncover the ordering information of committed files. Yang et al [27] used a similar technique as Zimmermann et al. [29] for identifying files that frequently change together. Canfora et al. [4] 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. Our approach can operate without this information.

Spanoudakis and Zisman [24] conducted a comprehensive study of various methods for link recovery that utilize such things as information retrieval, test-cases, and design patterns. Marcus and Maletic [18] used Latent Semantic Indexing (LSI) to recover links, with better precision, from documentation to source code on the same set of case studies done by Antoniol et al. [2] using vector space IR models. A number of other researchers [7, 13, 17, 22, 26] have also applied IR methods for traceability link recovery. The results of these studies demonstrate the usefulness of IR methods for link recovery. These 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 two versions at a time, Antoniol et al. [3] establish traceability links between software releases of an object oriented system to determine inconsistencies.

Egyed [8] use program run-time information using test cases to uncover traceability information between software artifacts. Spanoudakis and Zisman [25, 30] 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. [5] propose an event based traceability approach in establishing traceability links between requirements and performance models using an event-notifier design pattern. Murphy et al. [20] introduce software reflexion models for identifying links between high-level models and source code.

6. Conclusions and Future Work

We used versions history to uncover traceability patterns consisting of source code files and other artifacts. A heuristic based approach that uses frequent-pattern mining is presented. The recovered patterns give the specific order in which the files in a pattern were changes. This ordering information can be utilized to infer the directionality (i.e., change causality) of the traceability link (e.g., source→documentation). We showed that these traceability links support software-change prediction with high precision, if a similar pattern frequently occurred in past versions.

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. We feel that the choice of the appropriate duration of history that maximizes accuracy for software-change prediction remains an issue of future investigation. Additional heuristics for grouping related change-sets such as textual similarity of commit messages are also being investigated. We feel that mining at a finer granularity for patterns (e.g., class or method level to paragraph) would produce better results; at least in terms of better context to developers. We are developing tools in this direction. Further, we plan to integrate our traceability tools directly into a version-control system.

7. References

[1] Agrawal, R. and Srikant, R., "Mining Sequential Patterns", in Proceedings of 11th International Conference on Data Engineering, Taipei, Taiwan, March 1995.

[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, vol. 28, no. 10, October 2002, pp. 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, pp. 211-219.

[4] 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, Sept, 19-22 2005, pp. 29-37.

[5] 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, pp. 362-371.

[6] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S., "Hipikat: A Project Memory for Software Development", IEEE Trans on Soft Engineering, vol. 31, no. 6, 2005, pp. 446-465.

[7] 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, pp. 307-316.

[8] Egyed, A., "A Scenario-Driven Approach to Trace Dependency Analysis", IEEE Transactions on Software Engineering, vol. 29, no. 2, 2004, pp. 116-132.

[9] 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, MD, Nov, 16-20 1998, pp. 190-199.

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

[11] Goethals, B., "Frequent Set Mining", in The Data Mining and Knowledge Discovery Handbook, Springer, 2005, pp. 377-397.

[12] Gotel, O. C. Z. and Finkelstein, A. C. W., "An Analysis of the Requirements Traceability Problem", in Proceedings of 1st IEEE International Conference on Requirements Engineering, Colorado Springs, CO, USA, April 18-22 1994, pp. 94-101.

[13] 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, pp. 138-147.

[14] Kagdi, H., Collard, M. L., and Maletic , J. I., "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution", Journal of Software Maintenance and Evolution: Research and Practice, vol. 19, no. 2, 2007, pp. 77-131.

[15] Kagdi, H. and Maletic, J. I., "Software Repositories: A Source for Traceability Links", in Proceedings of 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSE'07), Lexington, KY, USA, March 22-23 2007, pp. 32-39.

[16] 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, pp. 47-53.

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

[18] 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, pp. 125-137.

[19] Masseglia, F., Teisseire, M., and Poncelet, P., "Sequential Pattern Mining: A Survey on Issues and Approaches ", in Encyclopedia of Data Warehousing and Mining nformation Science Publishing, 2005.

[20] Murphy, G. C., Notkin, D., and Sullivan, K., "Software Reflexion Models: Bridging the Gap between Source and High-Level Models", in Proceedings Symposium on Foundations of Software Engineering, NY, NY, October 1995, pp. 18-28.

[21] Scacchi, W., "Understanding the Requirements for Developing Open Source Software Systems", IEE Proceedings--Software, vol. 149, no. 1, February 2002, pp. 24-39.

[22] 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 Int Workshop on Principles of Software Evolution (IWPSE) Kyoto, Japan, Sept 6-7 2004, pp. 49-54.

[23] 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, pp. 24-28.

[24] 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.

[25] Spanoudakis, G., Zisman, A., Perez-Minana, E., and Krause, P., "Rule-based generation of requirements traceability relations", The Journal of Systems and Software, vol. 72, no. 2004, 2004, pp. 105-127.

[26] Sundaram, S., Hayes, J. H., and Dekhtyar, A., "Baselines in Requirements Tracing", in Proceedings of Workshop on Predictive Models of Software Engineering (PROMISE) 2005, St. Louis, MO, May 2005, pp. 12-17.

[27] 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, vol. 30, no. 9, September 2004, pp. 574-586

[28] Zaki, M. J., "SPADE: An Efficient Algorithm for Mining Frequent Sequences", Machine Learning, vol. 42, no. 1-2, January 2001, pp. 31 - 60.

[29] Zimmermann, T., Zeller, A., Weißgerber, P., and Diehl, S., "Mining Version Histories to Guide Software Changes", IEEE Trans on Soft Engineering, vol. 31, no. 6, 2005, pp. 429-445.

[30] 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, pp. 448-455.