Mining evolutionary dependencies from web-localization repositories

Huzefa Kagdi and Jonathan I. Maletic

Department of Computer Science, Kent State University, Kent OH, U.S.A.

SUMMARY

An approach for mining repositories of web-based user documentation for patterns of evolutionary change in the context of internationalization and localization is presented. Localized web documents that are frequently co-changed (i.e., an evolutionary dependency) during the natural language translation process are uncovered to support future evolution of the system. A sequential-pattern mining technique is used to uncover patterns from version histories. Characteristics of the uncovered patterns such as size, frequency, and occurrence within a single natural language or across multiple languages are discussed. Such patterns help provide insight as to the effort required in retranslation due to a change in the documentation. The approach is validated on the open source *KDE* system. *KDE* maintains documentation for over 50 different natural languages and presents a prime example of the problem. The technique accurately predicts which documents in *KDE* are retranslated or updated in future versions.

KEYWORDS: software evolution, mining software repositories, localization

1 INTRODUCTION

Open source systems have a global user community that spans across geographical and cultural boundaries. In order to better serve and retain such a diverse user base, open source projects are increasingly developed so that they may adapt to the various natural languages and environments, i.e., they can be *localized*. Products such as *Linux*, *OpenOffice*, and *KDE* provide localization of user interfaces and user documentation, including online help web pages, in several languages and locales (i.e., characters set, date/time, and currency).

It is not uncommon for large-scale software to have a great number of online help manuals localized in multiple languages. For example, Figure 1 shows the *KSpell* (a *KDE* application) online help manuals in US English and German. As the software evolves due to continual changes (e.g., new feature additions or defect corrections) these localized documents also need to be evolved. Unfortunately, software evolution and localized evolution follow disjoint paths. This is in part due to a different set of teams and contributors performing these two different, yet related tasks in isolation. Typically, the software developers rest the responsibility of localization on the translators, and vice versa. Given this situation, a translator or team leader is faced with a common set of questions related to the impact analysis task during evolution of localized documents:

- Given a change in a specific localized document what other localized documents need to be co-changed?

- Localization in how many languages is affected?
- How much time will it take to carry out these changes?

Answers to these questions help determine if changes in localized documents should be planned for an upcoming software release. This also assists translators to identify potential out-of-date parts that need retranslation and discard obsolete documents. This activity in the localization process is termed a *string freeze*. Fortunately, in a distributed collaborative development environment such as open source development, localized documents are managed in a way similar to source code. Typically, multiple teams contributing to the translation into a set of languages are involved in localization. The localized documents, like source code, are stored in repositories that are managed by version-control systems (e.g., *CVS* and *Subversion*). We term repositories that store localized online-help documents as *web-localization repositories*[1]. Such a repository stores every change in every document that is checked-in to a repository. We refer to all the changes made to a specific document during its evolution that are stored in a repository as the *document-history*.

Our premise is that the document histories stored in web-localization repositories can be utilized to extract pertinent information and/or uncover relationships or trends about evolutionary characteristics of localized documents. In order to substantiate this claim, we present an approach that is based on sequential-pattern mining [1] to uncover patterns of documents that frequently co-change in a single language or across multiple languages. These patterns are mined from the web-localization repositories. The recovered patterns provide not only a set of documents that are likely to be retranslated or updated in a single version but also in a series of versions. For example, the pattern *{kexi_basics.po}→{kexi_database.po}* mined from the *KDE* localization-document repository indicates that the localized-document *kexi_basics.po* is changed before the localized-document *kexi_database.po* in two successive versions. This pattern is found in the change history of thirteen translated documents across three different languages. Therefore, it is an indicative of a strong evolutionary dependency among the involved documents. Such patterns could be used to answer the above questions.

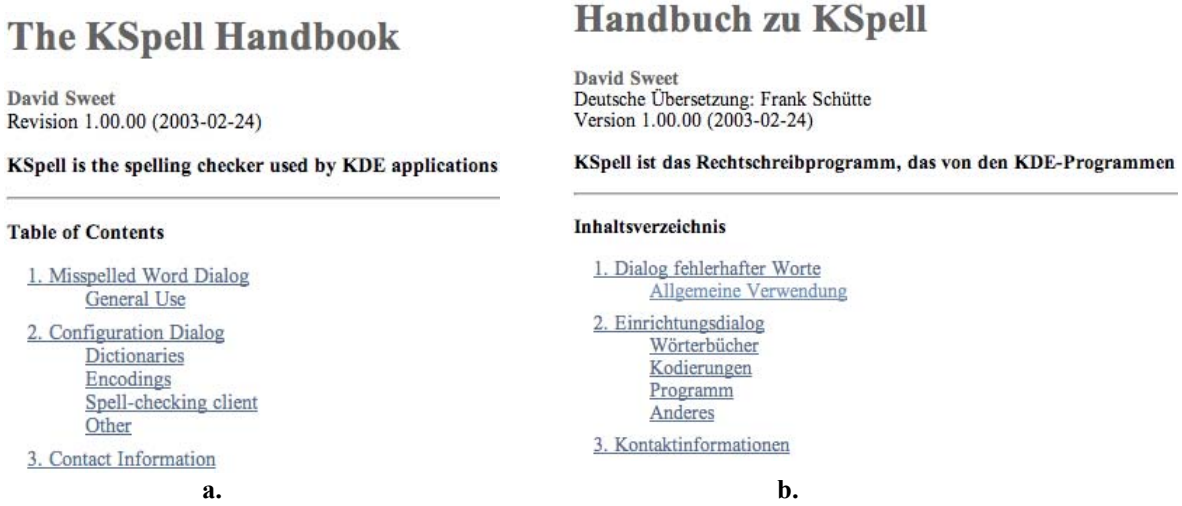


Figure 1. Online manuals for the KDE application KSpell in languages a.) US English and b.) German.

In our previous work [2], we uncovered patterns of documents involved in the translation process including program localization. From this, we formed a conjecture that information embodied in these patterns is useful for supporting localized document evolution. In the work presented here, we provide a rigorous experimental validation of this conjecture in the context of evolving translated online user guides and other web documents. The evaluation will show that the patterns mined from an earlier part of document histories reoccur with similar frequency in a later part of the document histories. As such, our approach directly supports the impact analysis questions stated above. Furthermore, it assists user communities to anticipate updates to the documentation in

[1]Here, use of the terms repository, document, and pattern without additional context refers to web-localization repository, documents involved in the translation process, and sequential-pattern of documents involved in translation respectively.

their respective languages based on the past translation patterns. Additionally, we believe that the patterns discovered by our approach could encourage/aid localization efforts in a new natural language by providing the translation patterns in other languages.

The rest of the paper is organized as follows. In section 2, we examine the document translation paradigm based on *gettext*. Section 3 discusses the evolution data found in the document repository and section 4 presents our pattern mining approach. The evaluation of our approach is given in section 5. We follow that with related work (section 6) and finally our conclusions are given in section 7.

2 DOCUMENT LOCALIZATION: THE GNU MODEL

A number of open source projects use the *gnu gettext* model for localization purposes [3]. Such projects produce source code (i.e., string literals) and documentation in a base language (typically US English), and then extract the strings and documentation that require a language-specific translation, and finally translate them to another language. Localization is a semi-automatic process in which the translation process, i.e., converting text in a document from one language to another, largely remains a human-intensive activity. Actually localization is only a part of the *Native Language Support* (*NLS*) paradigm. Establishing NLS in a system encompasses *Internationalization* (better known as *i18n*) and *localization* (better known as *l10n*) [3]. Internationalization is a generalization process that gives a program the ability to understand and support multiple locales (e.g., interaction messages, input, output, date/currency formats in multiple languages). Localization is a specialization process that produces a locale-specific instance from an internationalized program. Though, NLS (specifically localization) requires a non-trivial human effort, tools such as *gnu gettext* and *KBabel* help simplify the task.

Multilingual online help documents of open source systems typically follow the same internationalization and localization process that is used for the source code of a project. Typically documents are produced in a base language (typically US English). A translation team localizes these base documents to another language (e.g., German). The translation process that follows *gnutext* model uses the *Portable Object* (*PO*) files [3] as a basic representation for localization. A PO file consists of textual entries. Each entry is a pair of untranslated and translated strings. An untranslated document (i.e., the parts of it that require internationalization) is converted to a corresponding PO file. Such PO files typically contain an empty translated string in the pair of strings and are stored as *POT (Portable Object Template)* files. These template files are then used to produce a language-specific translation by manually filling in the empty translated strings. The language-specific copies of these documents are then converted back to the original format of the documents. With the context of documents in the *gnu gettext* model, we define the following terms,

Definition: Base-documents are documents written in a language that does not need translation or localization (typically US English).

Definition: Internationalized-documents are the documents formed from the parts or whole of the base documents that serve as customization templates for translation to a set of other languages (POT files).

Definition: Localized-documents are the documents that are derived from the internationalized documents and translated to a specific language (PO files).

Definition: Translated-documents are language-specific localized documents equivalent to the base documents (e.g., German and Hindi).

Definition: *Message-pair* is a pair of a string in base-language and its translated string in another language (e.g., a pair of *msgid* and *msgstr* in PO/POT files).

We now explain the document localization process by using a simple, albeit concrete, example from the *KDE* project. In the case of *KDE,* the base-documents are written in the US English. The ubiquities *docbook* representation (www.docbook.org) is used for authoring base-documents, and then used for transforming documentation in various formats including the HTML manuals. A number of standard tools (e.g., *XSLT* and *xsltproc*) exist to facilitate this transformation task. The online manuals shown in Figure 1a and Figure 1b are generated with such a transformation from their respective docbooks. Consider a portion of a base-document shown in Figure 2a. The content of the element *title* is in the language US English. In order for this document to support native language, it is converted to an internationalized-document (POT format) as shown in Figure 2b. An entry (*msgid*, *msgstr*) of strings is created. The string prefixed with *msgid* contains the content of the

element *title* from the base-document and the string prefixed with *msgstr* is left empty. This conversion is facilitated by the tool *xml2po*. The tool also inserts additional context information in the form of comments. For example, the comments (lines starting with the #-symbol) provide information about the element name whose content is translated and its location in the original document. An instance of localized-document (i.e., a PO file) is created from the internationalized-document. In this case, the string *msgid* is manually translated to German language and entered in the string *msgstr* as shown in Figure 2c. Finally, the localized-document is converted to the translated-document (*docbook* format with textual contents in German) as shown in Figure 2d.

As the software evolves, the base-documents may need to be changed due to the changes in the software. Now, we discuss the evolution of documents with regards to their localization, where the evolution data is stored, and how that data can be acquired and used to support their further evolution.

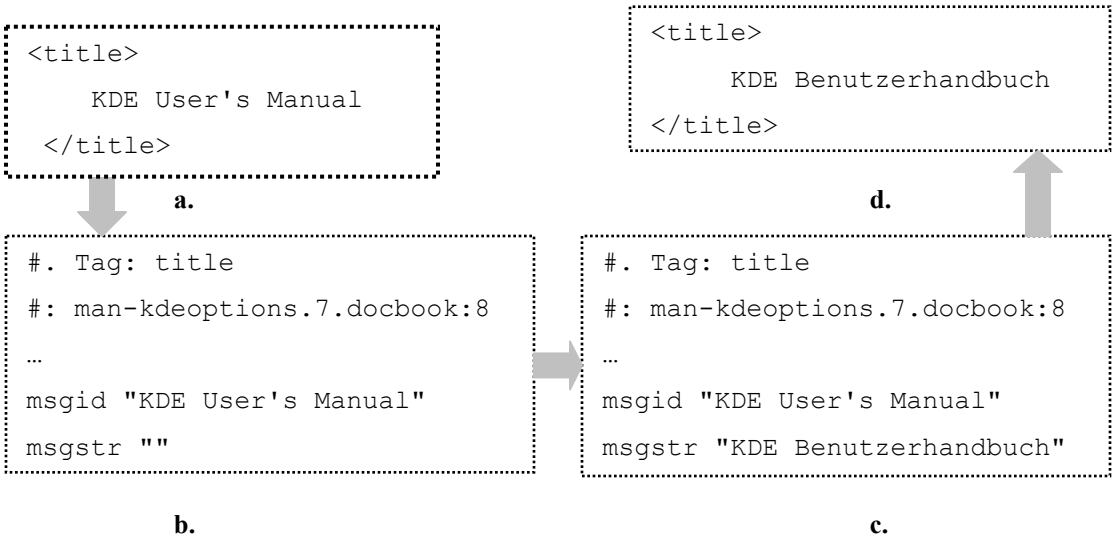


Figure 2. Excerpts of a.) the base-document man-kdeoptions.7.docbook from kdelibs in US English and d) its corresponding translated-document in German. Excerpts of of b.) internationalized-document for translation to other languages (POT file) and c.) localized-document in German (PO) used in the translation process.

3 THE EVOLUTION OF TRANSLATED DOCUMENTS

The evolution of translated web documents is analogous and closely coupled to source code evolution. The task of performing a single high-level logical change in source code is either planned activity (e.g., addition of a new feature or a refactoring), 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). However, such is the nature of software that an extremely well planned change may lead to further unanticipated changes. A single logical change can crosscut multiple source code locations, and require a large sequence of changes in the source code for its complete realization. Consequently, such a single high-level source code change may cause a number of changes to multiple base-documents.

The translation teams work towards keeping the corresponding translated-documents synchronous with the changes in the base-documents. The evolution of documents that are involved in the translation, like source code evolution, is a continuous process in which changes are performed incrementally. That is, (re)translation is often performed for a group of "related" message-pairs, possibly spanning across multiple documents, in a single step. Once again, a sequence of groups, of related message-pairs, may change over time to accommodate the changes in base-documents. Changes in base-documents result in the following change scenarios:

- *Additions (A)*: New internationalized-documents, triggering addition of corresponding new localized-

documents and translated-documents.

- *Deletions (D)*: Obsolete internationalized-documents, eventually causing deletion of corresponding localized-documents and translated-documents.
- *Modifications (M)*: Updates to internationalized-documents that lead to: addition (*M-A*) and/or deletion (*M-D*), and/or modified (*M-M*), message-pairs in corresponding localized-documents.

A change in an internationalized-document (POT file) may ripple changes in multiple localized-documents across a number of languages. While the scenarios *A*, *D*, *M-A*, and *M-D* are generally straightforward to handle, the scenario *M-M* often needs a careful consideration. A given *M-M* change could invalidate translations in existing message-pairs, thus requiring them to be retranslated. The use of tools such as *gettext* and *KBabel* in the document evolution is analogous to the use of tools *diff* and *patch* in the context of source code evolution. They provide assistance in merging the changed message-pairs from the internationalized-documents with the corresponding message-pairs in the localized-documents. These tools help identify message-pairs in the localized-documents that should be considered as retranslation candidates (*M-M*), add new message-pairs that should be translated for the first time (*M-A*), and mark the message-pairs that have possibly become obsolete (*M-D*).

However, the human translators are the ones who finally decide on how to handle the affected message-pairs, and manually carry out the appropriate changes including (re)translation. There is no single standard that is universally accepted and practiced in managing this task. The process in which the affected message-pairs are changed depends largely on the translator's discretion. This includes, for example, checking which localized-documents need changes with regards to changes in the internationalized-documents (with tool support), which message-pairs should be changed together as a single related group, how many groups should be formed, and the order in which these groups should be changed to align with the equivalent changes in base-documents. Clearly, this could vary according to the translator's experience, skills, and overall project state and policies.

Fortunately, the situation is not completely abysmal. Documentation and their localization is a team activity in large open source projects. It is not uncommon to have multiple contributors developing the documentation and translating the same part of the system. Therefore, version-control systems are commonly used to coordinate and manage these efforts. This historical information about the evolution of the translated documents is often captured in web-localization repositories. These web-localization repositories, like source code repositories, are managed by version-control systems such as *Subversion* (http://subversion.tigris.org/) or *CVS* (www.nongnu.org/cvs/). Additionally, metadata such as user-ids, timestamps, and commit comments are often times stored. This metadata can explain some of the why, who, and when characteristics of a performed change. Let us now examine the information found in the repositories and how it can be obtained. We first start with definitions that are relevant to the discussion.

Definition: A *change-set* is a set of changed documents that are checked-in together to a repository in a single commit operation.

Definition: A *revision-number* is an identifier used by version-control system to track the state of documents at a given point in time (i.e., a version). Used synonymously with the term *version-number*.

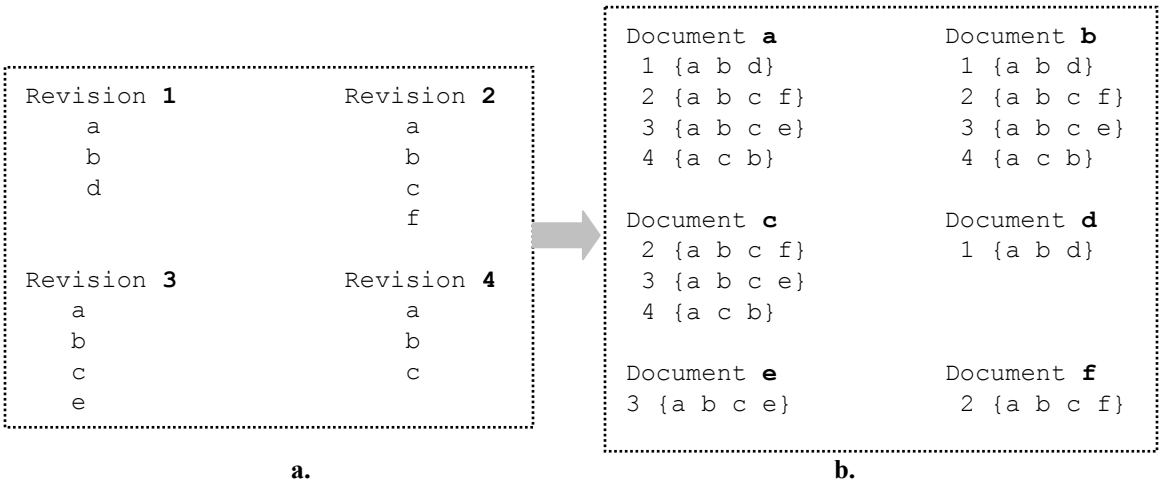
Definition: A *logentry* is a record of metadata associated with the change-set or its member document by a version-control system.

In a repository managed by *Subversion*, version-number assignment and metadata are associated at the change-set level and recorded as a logentry. *Subversion*'s logentries include the attributes *committer* (the person who checked-in the change-set), *date* (the change-set checked-in data and time), and *paths* (files and directories in the change-sets). Each change-set (logentry) is uniquely identified by a revision-number. Figure 3a shows four logentries (i.e., change-sets) of a hypothetical web-localization repository. Revision 1 consists of change-set {*a, b, d*} with three documents *a*, *b*, *d* committed together. Besides metadata, *Subversion* provides access to any version of the document and the difference between any two given versions of a document.

We should note that modern source-control systems, such as *Subversion*, have several improvements over systems such as *CVS* as they preserve the grouping of several changes in multiple files to a single change-set as performed by a committer (i.e., an atomic commit). Preservation of a change-set as an atomic commit in 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. This is a standard *Subversion* policy of the *KDE* project.

The logentries can be readily obtained from the repository via standard version-control system commands. We term this as change-set-oriented view of the change history. However, as we discussed earlier, the granularity and composition of a change-set may vary across tasks, developers, and projects. Therefore, a single high-level change may be completed over multiple change-sets. The number of documents (i.e., size) may vary across the change-sets throughout versions history. On one end, some change-sets may contain only a few documents that are changed slightly (e.g., only a single message-pair is translated in a singe document). This is a case when a single logical change is performed incrementally and is completed by committing multiple versions. On the other end of the spectrum, you can have a change-set that contains a large number of documents that are completely translated. This is a case where the entire task is completed and then all the changed documents are committed in a single version. Also, the order in which these documents appear in a logentry is not necessarily the order in which they were changed. Simply considering a single logentry is insufficient to determine all the related documents that are typically changed together and the specific (temporal) ordering of the documents involved in a change-set.

In order to uncover a complete change performed for a single high-level change, we need to consider changes that spread over a sequence of change-sets. However, the change-sets corresponding to such changes are rarely explicit, at least not directly recorded in the web-localization repositories (or any type of repository generally), or clearly documented. Note that the change-sets stored as atomic commits in web-localization repositories are serialized. The order in which log entries appear in the log files is controlled by 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 web-localization repositories as related to a single high-level change may prove to be meaningless. Next, we describe our approach to uncover evolutionary dependencies from the change-sets in web-localization repositories.



a. **b.**

Figure 3. a.) An example of four revisions available directly from a web-localization repository and b) the change-set-oriented history converted to a document-history.

4 AUTOMATICALLY UNCOVERING EVOLUTIONARY COUPLINGS

Broadly, our investigation is about how we can approximate related changes in documents that represent a single cohesive high-level change along with the ordering information. The information that can be utilized is the (serialized) change-sets committed in a specific temporal order and the metadata in the repository. In an effort to obtain complete coverage of the documents that typically change together over multiple change-sets, we take an alternative view of history compared to what is directly available from the version-control system. Furthermore, on this view we employ sequential-pattern mining to uncover evolutionary couplings.

4.1 DOCUMENT-CHANGE HISTORY

The history of documents available from repository is in a change-set-oriented view. However, as discussed in Section 3, change-set-oriented view may not provide the complete coverage of documents that are changed to realize a single logical change and may produce meaningless patterns. Therefore, we take an alternative approach of rearranging the change-set-oriented view of history into a view that gives the history of a document with regards to all the change-sets in which it is involved. All these change-sets are related as they contain changes to the same document. We term this alternate arrangement of history as document-history.

In this view of document-history, a document contains a sequence of all change-sets in which it is involved. The ordering in a sequence is based on the revision numbers. A change-set in a sequence with a lower revision number is assigned a position before a change-set with a higher revision number. For the example shown in Figure 3a, the corresponding document-history is shown in Figure 3b. The history of document *a* consists of a sequence of change-sets from revisions *1*, *2*, *3*, and *4*. Notice that a change-set in a document-history may contain additional documents that are also co-changed. We now ask the question: Are the co-changes in a single document-history sufficient to infer that such co-changed documents have an evolutionary dependency? Our hypothesis is that if the same co-changed documents occur in a number of document histories (i.e., frequently) then it is likely an evolutionary dependency.

4.2 PATTERN MINING

Our approach automatically uncovers sets of documents that are typically co-changed in the translation process from the document histories. These uncovered evolutionary couplings can be used to support future changes in localized documents by helping to answer the question: Which other documents need to be changed due to a change in a given document? A frequent-pattern mining technique from the data mining community is used in the approach. This technique does not require any predefined rules to infer co-changes. In fact, coming up with predefined rules may not be always practical due to the latent nature of change practices among developers. These practices are rarely documented and enforced, and as such predefined rules may not be kept up-to-date with the system evolution. Moreover, our approach is not restricted to a predefined set of specific documents that are included/excluded for mining. Alternatively, our approach examines the documents in web-localization repositories and automatically infers evolutionary couplings of documents involved in the translation process of a software system. Now, we describe the specific frequent-pattern mining technique used in our approach.

The sequential-pattern mining technique from the data mining community can be adopted to accomplish the task of discovering frequently co-changed documents. The general problem of sequential-pattern mining from any dataset takes a given set of sequences (composed of items) and finds all the frequently occurring subsequences (i.e., ordered patterns) that have at least a user-specified minimum support [4]. 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. Before we describe the sequential-pattern mining approach in the context of our problem, data-mining terminology that is relevant to the discussion is introduced. The input data to frequent-pattern mining algorithms are in the form of *transactions* (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 basic idea is that if the support of a pattern is at least a user specified *minimum support* then it is a frequent pattern in the considered dataset.

Formally, the problem of finding frequent sets of sequences is defined as given a set of items, $\alpha = \{i_1, i_2, ... i_m\}$, and a set of transactions, $\tau = \{T_1, T_2, ..., T_n\}$, find all the sets of sequences, $S = \{S_1, S_2, ...S_o\}$, that co-occur in at least a given number (or percentage) of transactions i.e., it satisfies a given minimum support, σ_{min}. Each Transaction contains an ordered list of events and is identified by an unique id, $T_i = (tid, \varepsilon)$ where $\varepsilon = [E_1, E_2, ..., E_p] \mid \forall_{i,j}\ E_i \rightarrow E_j$ and $\rightarrow$ is a given ordering relation on events. Each event contains a set of items and is identified by a unique id, $E_i = (eid, \subseteq \alpha)$. Each sequence is defined as an ordered list of elements (i.e., itemsets), $S_1 = [I_1 \rightarrow I_2 \rightarrow ..... \rightarrow I_p] \mid \forall I_i \subseteq \alpha$, and each member of an element, $i_j \in I_i$ is defined as an item of a sequence. A mined sequence S_i is called a *frequent sequence.*

We pose the problem of mining patterns of documents that frequently co-change, i.e., evolutionary

dependencies, as an instance of the general problem of sequential-pattern mining. To derive this instance from the general problem, we need to map the general concepts such as item, event, and transaction to their counterparts in our problem's context. Here, an item corresponds to a document that is found in at least one change-set in a considered (part of the) version history, an event corresponds to a single change-set, and a transaction corresponds to a sequence of change-sets of a single document. The ordering of a sequence of change-sets in a transaction is defined by the ordering of revision-numbers. The documents in the same change-set are unordered, whereas, the documents in different change-sets are ordered according to their revision-numbers. For example, the documents *d1* and *d2* in the same change-set with the revision-number *1* occurs before the documents *d3* and *d4* in another change-set with the revision-number *2*. However, documents *d1* and *d2*, and *d3* and *d4* that occur in the same change-set are left unordered.

Application of sequential-pattern mining to our problem automatically produces a set of frequent sequences. That is, a frequent sequence occurs in at least a user specified number of documents histories. Here, an evolutionary dependency between documents is represented as a frequent sequence. We refer to these frequent sequences as ordered patterns. An ordered pattern is made up of ordered elements. Each element is made up of unordered items. The ordering of elements imposes a partial order on the items. For example, the pattern *{d1, d2}→{d3, d4}→{d5}* is made up of 3 elements and 5 items. It indicates that the element *{d1, d2}* happens before the element *{d3, d4}* and the element *{d3, d4}* happens before the element *{d5}*. However, the happens-before relation between items *d3* and *d4* is unknown in the element *{d3, d4}*. Therefore, an ordered pattern can (indirectly) establish both the ordered and unordered relationship between items. In context of our problem, an element of a pattern maps to a change-set or its subset. Therefore, an element of a pattern is the set of documents that change in the same reversion or change-set. Elements are ordered according to their version-numbers. An element with a higher version-number occurs after another element with a lower version-number.

We have developed a sequential pattern-mining tool [5], namely *sqminer*, that is based on the Sequential Pattern Discovery Algorithm (SPADE) [6] 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 item vs. a set of transactions consisting of items) for efficient counting of support values. 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 items are also frequent. The support of a pattern is always less than or equal to the sub-patterns. A common pruning mechanism used in frequent-pattern mining is to eliminate all the sub-patterns that have the same support of the corresponding (larger) pattern. Such sub-patterns are only used with other larger patterns and not independently. Therefore, they give redundant information that may be of a very little meaning. As a result, only disjoint patterns (i.e., patterns with no common calls) that subsumes all the subsets patterns with the same support, and subsets of patterns that have higher support values are retained. Such patterns are known as *closed* patterns. Our approach produces only closed patterns. Frequent-pattern mining algorithms typically report the support of a pattern but not the transactions in which it occurs. The transaction(s) in which a pattern is found is also recorded.

```
path/a 1 a b d          path/b 1 a b d          path/c 2 a b c f
path/a 2 a b c f        path/b 2 a b c f        path/c 3 a b c e
path/a 3 a b c e        path/b 3 a b c e        path/c 4 a c b
path/a 4 a c b          path/b 4 a c b          path/d 1 a b d
path/e 3 a b c e        path/f 2 a b c f
```

Figure 4. Transactions in *sqminer*'s input format. Each event is specified on a separate line and consists of a document name along with its path, version-number, and the set of documents co-changed.

4.3 EXAMPLE

Now we demonstrate the automatic mining of patterns with *sqminer* on the document-history shown in Figure 3b. This document-history is first converted to the input-transaction format of *sqminer*. In this format, transactions are specified in the form of events. Each event is specified as a three-unit tuple on a separate line. The first unit is the document name along with the complete path, the second unit is the version-number of a change-set, and the third unit is the set of all documents in a change-set. Figure 4 shows the transactions of the document-history of Figure 3b. For example, the first event is for the document *path/a* that occurs in the revision *1* consisting of a change-set with the documents *a*, *b*, and *d*. Here, multiple events are specified on the same line for the sake of brevity. Our approach automatically determines that six documents *a*, *b*, *c*, *d*, *e*, *f* are changed in this example of version history and therefore forms six corresponding transactions or document-histories.

The transactions of Figure 4 are then input to *sqminer*. We will use a minimum support of two for a candidate pattern, i.e., at least two document-histories must contain the pattern. Table 1 shows a total of 11 uncovered patterns. The columns *Support*, *Versions*, and *Documents* give the number of document-histories in which a pattern occurs, the number of versions/change-sets committed to complete a pattern, and the number of documents involved in a pattern respectively. Five patterns occur in two document-histories, another two in both three and four document-histories, and one each in five and six document histories. Five patterns take a single change-set or version to complete (i.e., unordered), whereas, others take more than one version (ordered). Also, only one pattern consists of two documents, whereas, others consist of more than two documents. Notice that sub-patterns such as *{a}*, *{b}, {a}→{c}* and*{c}→{a, b}* are not reported individually as they are closed by larger patterns with the same support values.

While the patterns with the same number of documents and support value are quite obvious findings, the patterns with number of documents less or greater than their support values are of greater interest. For examples, the first pattern *{a}→{a, c, b, f}→{a, c, b, e}→{a, c, b}* consists of 5 documents and occurs in two document-histories, and the last pattern *{a, b}* consists of two documents and occurs in six document histories. Such patterns are cases where there are documents in a pattern that tend to change more in relationship with other documents than independently by themselves. The first pattern consists of five documents *a*, *b*, *c, e*, and *f*, but is supported by only the version histories of documents *a* and *b*, and not the version histories of documents *c*, *e*, and *f*. This potentially indicates that the changes to the documents *c*, *e*, and *f* are potentially dependent on the changes to the documents *a* and *b*. The last pattern consists of two documents *a* and *b*, but is supported by the version histories of all the six documents. This potentially indicates the changes to documents *a* and *b* are ubiquitous and perhaps finding such patterns add not much value.

Table 1. Patterns produced by *sqminer* from the transactions of Figure 4.

No.	Pattern	Support	Versions	Documents
1	{a}→{a, c, b, f}→{a, c, b, e}→{a, c, b}	2	4	5
2	{a, b, d}→{a, c, b, f}→{a, c, b}→{a, c, b}	2	4	5
3	{b}→{a, c, b, f}→{a, c, b, e}→{a, c, b}	2	4	5
4	{a, b, d}→{a, c, b, e}→{a, c, b}	2	3	5
5	{d}→{a, c, b, f}→{a, c, b, e}→{a, c, b}	2	4	6
6	{a, b, d}	3	1	3
7	{a, c, b, f}→{a, c, b, e}→{a, c, b}	3	3	5
8	{a, c, b, f}	4	1	4
9	{a, c, b, e}	4	1	4
10	{a, c, b}	5	1	3
11	{a, b}	6	1	2

5 EVALUATION

The assessment of our approach is two fold. First, we show that our approach is able to uncover evolutionary-patterns of documents that are involved in the translation process and show the characteristics of these patterns. Second, we show that the mined evolutionary patterns can be used to support further evolution (i.e., supporting impact analysis and change prediction) of these documents with a high accuracy.

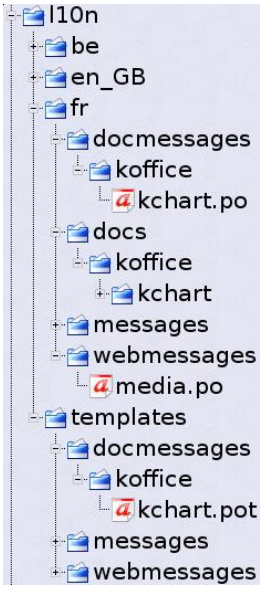


Figure 5. The structure of the KDE localization repository.

5.1 DOCUMENT REPOSITORY AND TOOL SET

The open source *KDE* (K Desktop Environment) system is used for evaluation. The *KDE* system is actually a collection of applications that represent a wide spectrum of domains, programming languages, sizes, and developers. The documents involved in the translation process are stored in the localization repository, called *l10n*, the structure and organization of which is shown in Figure 5. A top-level directory is assigned for each language (e.g., the directory *fr* for the language *French*) and the internationalized-documents (i.e., the directory *templates*). The directories *messages*, *docmessages*, and *docs* are used for the program localized-documents, user guide localized-documents, and the translated-documents respectively. The internalized-documents for program and user guide localizations are stored in the directories *messages* and *docmessages* respectively. Both the internationalized-documents and localized-documents are organized into a separate directory for each package, and a file represents a specific application within a package (e.g., *kchart.pot*, *kchart.po*). The translated-documents for each application are stored in a directory typically with the same name as the localized-document file (e.g., the directory *kchart*). The directory *webmessages* contain documents used in the translation of the *KDE* website (*www.kde.org*).

In this repository over 50 different languages are managed for translation. Online user guides in more than 20 languages are generated for a number of applications. We considered a subset of the *l10n* repository (from the *trunk*) between the time periods of 2006-01-01 and 2006-03-03 for mining patterns. There are over 4,000 change-sets (i.e., revisions) committed during this two-month period. This period represents a typical two-month period in *KDE* development. We picked this time period because it was relatively recent but old enough to allow verification of our findings on more recent versions of the system. The change-sets in this period consist of changes to over 21,000 different documents. These documents are changed over 80,000 times. This indicates that on an average a document is changed approximately three times and an average change-set consists of about five documents. Only change-sets that consisted of less than or equal to ten documents are considered. This is done in order to discard noisy change-sets such as those updating the license information

and/or merging/copying. This is commonly used pruning method for mining software repositories [7-10].

The *KDE* document repository is managed by *Subversion*. We developed a change-set extraction tool, namely *changeextractor* that uses *pysvn* (i.e., *Subversion* API for *Python*) to extract change-sets from a *Subversion* repository without using a working copy. The tool *changeextractor* takes a repository URL, start date, and end date of a history, and extracts the logentries from the repository for a specified period between the start and end dates. Furthermore, the tool *changeextractor* rearranges the logentries in the document-history view of a repository.

With the logentries represented in the document-history view, our sequential-pattern mining technique is applied to find all the frequently occurring patterns in the document-histories. The document-history view of a repository constructed by the *changeextracter* is fed to the mining tool, *sqminer*. Mining frequent ordered patterns with *sqminer* produces a set of closed frequent (ordered) patterns. Additionally, a pattern's support and documents histories in which it is found are reported. The configuration parameters of *sqminer* include support, maximum pattern size (i.e., number of items in a pattern), and output of patterns in a XML format. Further detail on the XML format of the ordered patterns can be found in Kagdi et al. [5].

5.2 MINED PATTERNS AND THEIR CHARACTERISTICS

Since online web documents are of focus here, we considered internationalized-documents, localized-documents, and translated-documents in the directories *docmessage*s, *docs*, and *webmessages* for mining patterns. In our previous work [2] on this topic we also included program-localization documents (e.g., *GUI* messages) in the directory *messages*. We found that program-localization documents exhibit much more frequent change activities than other localized documents. Mining for highly frequent patterns considering both types of localized documents typically results in a very few patterns that are exclusive to user documentation. Therefore, we excluded program-localization documents from the analysis presented here to achieve focused results and analysis of the user-document localization.

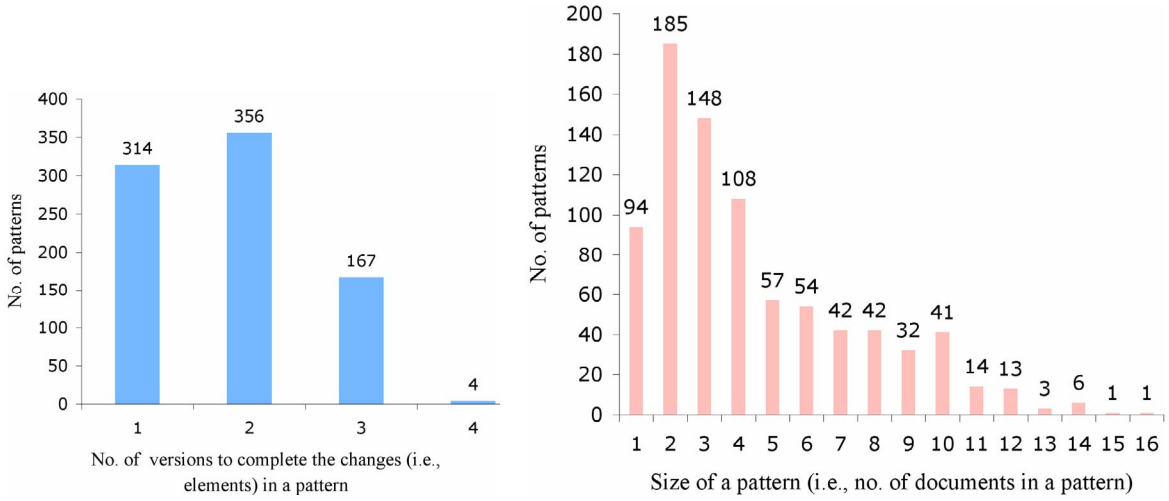


Figure 6. a.) Distribution of patterns with regards to the number of versions taken to complete changes in them and b.) frequency distribution of pattern sizes.

We used *sqminer* to perform mining with a minimum support of two. That is, a pattern must occur in at least two document-histories, not necessarily in the same language, to be considered as frequent, and thus an evolutionary coupling. We uncovered 841 patterns from the considered *KDE l10n* repository. Figure 6a and Figure 6b show the frequency distributions of patterns in terms of versions and sizes. The number of versions (i.e., elements) of a pattern is the total number of versions that were committed to complete all the changes to all its constituent documents (i.e., size). This information about an evolutionary dependency can serve as an

indicator as to how many documents are likely to co-change and the estimated number of versions taken to complete the changes in all the co-change documents. Additionally, this information is important to the decision making process with regards to allowing changes during a string (hard) freeze. That is, how many documents (size) are likely to be involved in a change and how much time (versions) it will take to complete changes in them.

Figure 6a shows that about 37% (314), 42% (356), and 20% (167) of the (841) mined patterns were completed in one, two, and three versions respectively. We also found four patterns that took four versions to complete. As can be seen almost 63% of the mined patterns take more than a version. This supports the existence of evolutionary dependencies that proliferate for longer than a single version. Approaches that consider only a single change-set and are used for software-change prediction, such as [7, 8], do not account for these types of evolutionary dependencies. Figure 6b shows that the patterns consisting of a single document form about 11% (94) and patterns involving two documents form about 22% (185) of the (841) mined patterns. The rest 67% (562) of the patterns consists of more than two documents. Patterns with as many as 16 documents were uncovered. Clearly, this shows that the evolutionary couplings typically consist of more than two documents. In other words, two or more documents frequently co-change. For example, the pattern *{krita.po}→{krita_tutorial-starting.po, krita_tutorial.po, kchart.po}→{krita_introduction.po, krita_tutorial-quick-starts.po, kformula.po, kile.po}* consists of eight localized-documents and takes three versions to implement all changes. The change-set of localized-document *{krita.po}* is changed in the first version, followed by the change-set of localized-documents *{krita_tutorial-starting.po, krita_tutorial.po, kchart.po}* in the second version, and finally the change-set of localized document *{krita_introduction.po, krita_tutorial-quick-starts.po, kformula.po, kile.po}* in the third version. These documents are used to generate translated documents of applications in the *KOffice* package of *KDE*. Notice this evolutionary dependency crosscuts multiple applications (e.g., *krita* a graphics editor and *kchart* a chart and diagram tool).

The patterns that are common (i.e., supported) in the versions histories of a number of different documents are also of interest. This type of information may support analysis of the amount of retranslation effort required due to potential changes in a common pattern between these document-histories. For example, the pattern *{amarok_config.po, amarok_quick.po}* is common in over 30 document-histories. That is, changes in the localized-documents *amarok_config.po* and *amarok_quick.po* may potentially cause ripple changes in all these documents. The pattern *{kplato_commands.po}→{kplato_options.po, kplato_wbs.po}* is another example that is supported in 20 document-histories and spans two versions. This type of ripple effect may lead to two versions of each of the 20 documents in which this pattern occurred. Figure 7 shows the frequency distribution of document-histories with regards to the number of patterns they support. It can be seen that there is a wide distribution of the number of documents supporting the number of patterns and the majority of the patterns have a unique number of supporting documents history. However, there are cases in which more than one pattern is supported by the histories of same number of documents but not necessarily the same set of documents.

Another interesting aspect is to examine different languages that share a common pattern (i.e., the same set of documents are changed in histories of different languages). This may help perform impact analysis with regards to the number of languages that may be affected due to a change in a common pattern that couples them. For examples, the pattern *{kplato_commands.po}→{kplato_options.po, kplato_wbs.po}* is exhibited in the document-histories in four languages and the pattern *{krita_using-layers.po krita_using-colorspaces.po}* is exhibited in the document-histories of eight languages. Figure 8a shows the frequency distribution of the 841 uncovered patterns with regards to different languages. Approximately 47% (396) of the patterns are confined to a single language and the remaining 53% (445) of the patterns span two or more languages. There are about 37% (310) of the patterns that are common between two languages but not necessarily the same pattern and same languages in every case. We found one pattern that is common in ten languages.

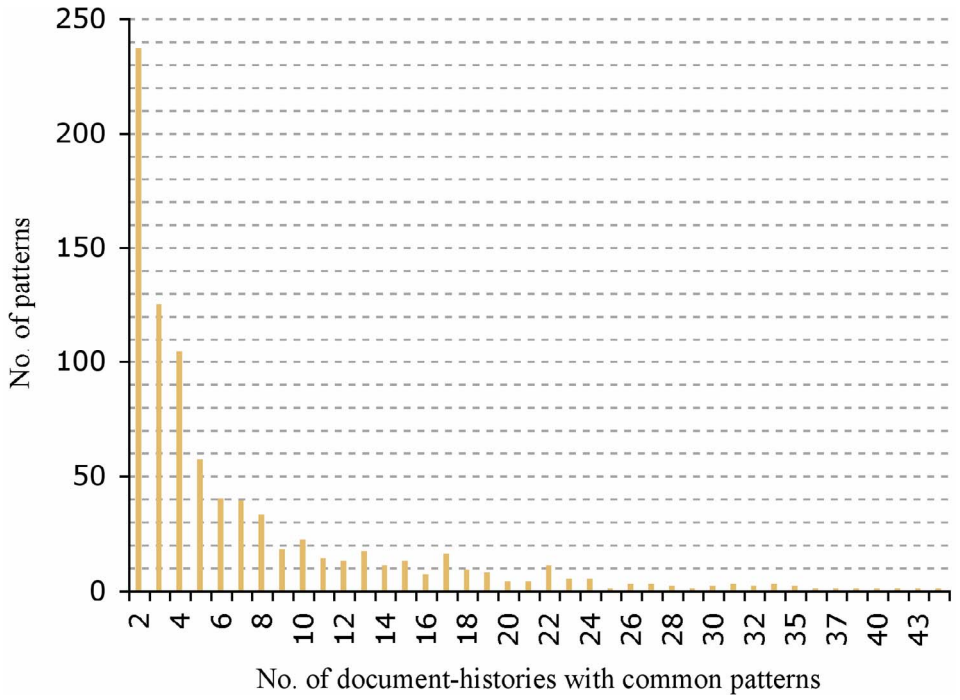


Figure 7. The number of common patterns that are found in the number of documents-histories.

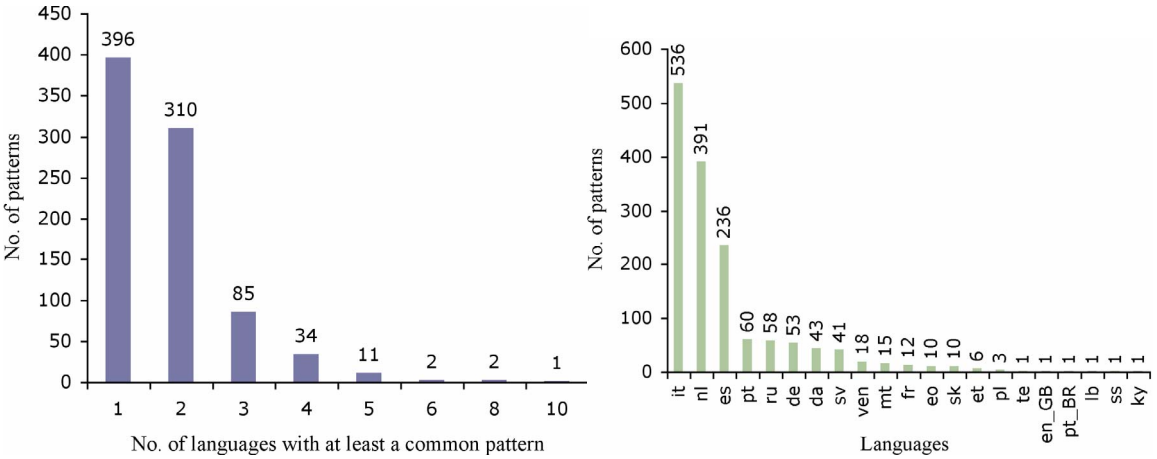


Figure 8. a.) The number of common patterns found in the histories of different languages, b.) total number of supported patterns by the translation history of a specific language.

Figure 8b shows the number of patterns found in the history of the individual languages. Notice that the same pattern may belong to multiple languages. A large number of patterns are found in the languages such as *Italian*, *Dutch*, *Spanish*, *Portuguese*, *Russian*, *German*, *Danish*, *Swedish,* and *French*. This shows frequent change activities in these languages. Therefore, translated-documents in these languages are likely to be abreast to the base-documents. Languages such as *Estonian*, *Polish*, *British English*, and *Brazilian Portuguese* begin to show some change activity. From this data we can surmise that user communities can anticipate translated user guides in all of these languages in the near future.

5.3 VALIDATING SUPPORT FOR EVOLUTION

We have shown that our approach is able to mine patterns with different characteristics such size, frequency, and occurrences within a single natural language or across multiple languages from a document repository. We now evaluate these patterns by examining their usefulness in supporting changes that occur later. More specifically, we validate these patterns for the task of change prediction. Our general evaluation methodology is to first mine a portion of the version history for 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. The training-set is the 841 patterns uncovered from the document-histories in the *KDE l10n* repository between 2006-01-01 and 2006-03-04. We mined the same repository but for the period between 2006-03-04 and 2006-04-03 using the same mining configuration as in training-set. This resulted in 47 patterns being mined from our evaluation-set.

The usefulness of the uncovered patterns can be seen in "how well" the training-set predicts the existence of a 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 with m patterns and evaluation-set with n patterns respectively. Each member ts_j where $1 \le j \le m$ is a pattern in the training-set and each member es_i where $1 \le i \le n$ is a pattern in the evaluation-set. Consider the pattern $es_i = \{f_1\} \rightarrow \{f_2\} \rightarrow, ..., \rightarrow \{f_k\}$ in the evaluation-set consisting of documents $f_1, f_2, ...,$ and f_k that is undergoing changes. To eventually predict this pattern, the training-set can be queried for candidates after each change-set $\{f_i\}$ of the pattern es_i is, or planned to be, performed. Let $Ces_i = Ces_{i1} \cup Ces_{i2} .... \cup Ces_{ik}$ where $Ces_{ij} = \{ts_1, \ , ts_p\}$ be a set of candidate patterns suggested from the training-set after the j^{th} change-set (and all the previous change-sets) 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 the total candidate patterns, both correct and incorrect, 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 $2/3 + 1/2 + 1 = 2.167$.

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

Definition: *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%).

Top 10 Translations Teams – trunk

Position	Team Name	Translated	%	Fuzzy	%	Untranslated	%	Total
1	Portuguese	79107	100.00	0	0.00	0	0.00	79107
2	Swedish	78036	98.65	68	0.09	1003	1.27	79107
3	Danish	77912	98.49	0	0.00	1195	1.51	79107
4	Spanish	76285	96.12	450	0.57	2632	3.32	79367
5	Estonian	54665	68.32	7162	8.95	18184	22.73	80011
6	Italian	53190	67.24	5363	6.78	20554	25.98	79107
7	German	52720	66.28	5246	6.60	21575	27.12	79541
8	Russian	50528	63.61	5512	6.94	23392	29.45	79432
9	Brazilian Portuguese	50352	62.85	6917	8.63	22849	28.52	80118
10	French	50011	62.76	6295	7.90	23378	29.34	79684
11	Dutch	47594	59.51	5313	6.64	27073	33.85	79980
12	British English	17988	22.72	3449	4.36	57746	72.93	79183
13	Catalan	15214	19.21	1483	1.87	62506	78.92	79203
14	Slovak	8579	10.84	3523	4.45	67065	84.71	79167
15	Polish	7713	9.74	1632	2.06	69810	88.19	79155

Figure 9. A screen shot of the KDE web page showing the status of translated help documents in the trunk of the l10n repository ordered by language on February 3, 2007.

Using these metrics, we can evaluate the "goodness" of our approach on the evaluation-set of our *KDE* study. Notice that precision is measured per pattern. Therefore we give its minimum, maximum, and average values. Coverage and recall are measured for the entire evaluation-set. The coverage and recall of the patterns in the evaluation-set with regards to the patterns in the training-set were found to be 34% and 25% respectively. Only the top ten candidate patterns were suggested for each prediction. The minimum, maximum, and average precision of the patterns were found to be 100%. This result shows that our approach has a high precision in predicting frequent changes. Also restricting candidates to the top ten candidates help prunes false positives. Notice that coverage and recall has to be the same for precision to be 100% otherwise. Coverage and recall can be considered as low. This could be partially attributed to the general limitation of a purely history-based model that might fail to predict changes unseen in the past. However, a high precision of our approach shows that if candidates are suggested, they are less likely to be false positives. Therefore, we believe that our approach will directly benefit translators, including novice contributors, in their localization efforts.

End users will benefit from a tool that allows them to foresee the availability of help documents in their respective local languages. Information derived from the mined patterns, such as shown in Figure 8b, is an indicator that translated help-documents for a particular language are likely to be soon available. Eleven languages: *Italian*, *Danish*, *Swedish, Portuguese*, *Dutch*, *Spanish*, *Russian*, *German*, *French, Catalan,* and *Norwegian Bokmål* were found in the evaluation-set. Based on the prediction from our training-set, nine out of these eleven languages with the exception of *Catalan* and *Norwegian Bokmål* (82%) were predicted to have

updated translations.

To further validate this finding, we consulted the web page that gives the localization status in the trunk of the l10n repository (http://l10n.kde.org/stats/doc/trunk/toplist.php) on February 3, 2007. A screen shot of this web page is shown in Figure 9. Comparing this with the languages we found in our training-set (almost 11 months earlier), it is clear that our approach was able to accurately predict the "top-ten" languages, along with a number of languages further down the list, well in advance.

Table 2. Accuracy of patterns with different history period and mining parameter combinations.

Training-set			Evaluation-set			Coverage (%)	Recall (%)	Precision (%)		
Period	MS	Patterns	Period	MS	Patterns			Min	Max	Mean
2006-01-01 to 2006-02-01	2	139	2006-02-01 to 2006-03-04	2	339	46.59	28.48	3.70	100.00	69.55
	4	101		4	202	35.15	26.73	5.56	100.00	91.74
	8	58		8	117	23.93	23.93	100.00	100.00	100.00
	2	139		8	117	39.32	36.75	100.00	100.00	100.00
	4	101		8	117	36.75	34.18	100.00	100.00	100.00
	2	139	2006-03-04 to 2006-04-03	2	47	14.90	14.90	100.00	100.00	100.00
	4	101		4	27	18.51	18.51	100.00	100.00	100.00
	8	58		8	12	25.00	25.00	100.00	100.00	100.00
	2	139		8	12	33.33	33.33	100.00	100.00	100.00
	4	101		8	12	33.33	33.33	100.00	100.00	100.00
2006-02-01 to 2006-03-04	2	337	2006-03-04 to 2006-04-03	2	47	29.79	21.27	100.00	100.00	100.00
	4	202		4	27	25.93	25.93	100.00	100.00	100.00
	8	117		8	12	25.00	25.00	100.00	100.00	100.00
	2	337		8	12	33.33	33.33	100.00	100.00	100.00
	4	202		8	12	33.33	33.33	100.00	100.00	100.00
2006-01-01 to 2006-03-04	2	841	2006-03-04 to 2006-04-03	2	47	34.04	25.53	100.00	100.00	100.00
	4	479		4	27	29.63	29.63	100.00	100.00	100.00
	8	239		8	12	25.00	25.00	100.00	100.00	100.00
	2	841		8	12	41.67	41.67	100.00	100.00	100.00
	4	479		8	12	41.67	41.67	100.00	100.00	100.00

5.4 THREATS TO VALIDITY

We discuss the internal and external validities of our approach with regards to the results obtained from its evaluation.

Internal validity refers to addressing the possible factors in our evaluation that bias the obtained results. Our evaluation was on a portion of the history of documents involved in the translation process. Also, the number of mined patterns depends on the user specified minimum support value. In order to assess the impact of the specific period and amount of history, and the user specific minimum support on the accuracy of change prediction, we further validated our approach with different values of these parameters and their combinations.

Table 2 shows the coverage, recall, and precision values for different training-sets and evaluation-sets. In the majority of these cases, our approach has high precision values. In only two cases is the mean precision below 100%. Coverage and recall are reasonable but cannot be considered as particularly high. An interesting observation is at higher values of minimum supports (e.g., 4 and 8), coverage and recall are the same. This indicates candidates suggested by our approach in case of prominent patterns are mostly correct. Also, at lower minimum support values, the difference between coverage and recall cannot be considered high. Improvements in coverage tend to show a decrease in recall and precision. Based on our experimentation results, we can assert that if our approach predicts a change to follow a specific pattern, it is generally precise for a given combination of the above parameters.

External validity refers to addressing the general applicability of our approach and conclusions to any given dataset. We validated our approach on a large open source system (*KDE*) that actively maintains localized documents in a number of languages. Also, our subject system follows the *gnu gettext* model with a particular

organization and structure of localized documents in the repository. While this system is a prime representative, we do not claim that our results will generalize to any given system (e.g., commercially developed software with a different repository structure). However, we believe that our mining toolsets are generic enough to be applicable to any translation process with a practice of committing related documents together to a repository.

6 RELATED WORK

We discuss the work on the analysis of multilingual web sites and web usage data. Also, we briefly discuss approaches utilizing information found in source code repositories maintained by tools such as *CVS* and *Subversion* with a focus on software changes.

Tonella et al. [11] used a combination of text and structure comparison technique to recover traceability links between multilingual documents. They use a lightweight text comparison approach based on the number of errors produced by applying the *UNIX* tool *spell* on a document. The structure comparison is based on the AST and edit distances of web pages. The goal of this work is to support consistency in information provided by a website in multiple languages. Niu et al. [12] used sequential-pattern mining to recover usage patterns from the user-session log information of web sites. The goal of this work is to use these patterns to support web site organization to help user-specific navigation and web page recommendation.

Zimmerman et al. [7, 8] 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. [13] used a similar technique for identifying files that frequently change together. Gall et al. [14] used window-based heuristics on *CVS* logs for uncovering logical couplings and change patterns, and German et al. [9] for studying characteristics of different types of changes. Hassan et al. [10] analyzed *CVS* logs for software-change prediction.

Van Rysselberghe et al. [15] utilized *CVS* logs in their approach to find frequently applied changes and presented a 2D visualization technique to help recognize change-relevant information [16]. Bieman et. al. [17] used logs from software repositories to assist in the computation of metrics for detecting change-prone classes. Burch et al. [18] presented a tool that supports visualization of association rules and sequence rules. However, very little information is provided on how *CVS* transactions are processed and sequences are mined. Beyer et al. [19] used the log information in visualizing clusters of frequently occurring co-changes. Dinh-Trong et al. [20] used *CVS* logs for validating previously developed hypotheses on successful open source development. Chen et al. [21] incorporated the *CVS* commit messages in their source code search tool. El-Ramly et al. [22] used sequence mining to detect patterns of user activities from the system-user interaction data. Recently, Kagdi et al. [23] surveyed and classified a number of Mining Software Repositories (MSR) approaches for various purposes in the context of software evolution. To our knowledge, uncovering evolutionary dependencies of localized documents with a MSR approach has not been investigated previously.

7 CONCLUSIONS AND FUTURE WORK

The work presented here applies sequential-pattern mining to uncover frequently co-changed web-documents (i.e., evolutionary dependencies) in the context of internationalization and localization. A large open source system, *KDE*, was used to evaluate the method. The mined patterns from the historical information in version archives proved to be beneficial for supporting the task of internationalization and localization of web-based documents. This is one of the first published works on the use of sequential-pattern mining for supporting the evolution of documents involved in the natural language translation process.

The performed evaluation shows that our approach accurately predicts the (partially) ordered sets of documents impacted due to a change in a given document, if a similar change pattern had occurred in the history. This is beneficial to developers and translators for the purpose of planning and maintenance as these patterns embody information such as the number of documents, time needed, and the specific order of a proposed change. We envision that our tool could be integrated into a version-control system in the form of *commit hooks* to support developers and translators. The method also demonstrated to be useful to end users interested in languages that would be better supported in future releases of the system. Our approach accurately

predicted, almost a year in advance, which languages would be better supported through updated translations in *KDE*.

For future work we plan to further assess our approach on open source systems such as *OpenOffice* and *Apache*. Additionally, we are working on extending our approach to produce finer granularity patterns than the document level and feel that mining patterns at the entry level of PO files should produce even more useful results. Also, translation processes that use different models of translation, such as *XLIFF* model (www.oasis-open.org/committees/xliff/documents/xliff-specification.htm) for localization, and projects with different organization of translated documents in the repository (e.g., *Apache* with translated web pages suffixed with the language code) will be investigated to understand the generality of our approach.

REFERENCES

1. Agrawal, R. and Srikant, R. Mining Sequential Patterns. *Proceedings 11th International Conference on Data Engineering*. IEEE Computer Society: Los Alamitos CA, 1995.
2. Kagdi, H. and Maletic , J. I. Mining for Co-Changes in the Context of Web Localization. *Proceedings International Symposium on Web Site Evolution (WSE'06)*. IEEE Computer Society: Los Alamitos CA, 2006; 50-57.
3. GNU. GNU gettext Free Software Foundation, http://www.gnu.org/software/gettext/manual/gettext.html [May, 25, 2006]
4. Masseglia, F., Teisseire, M., and Poncelet, P. Sequential Pattern Mining: A Survey on Issues and Approaches *Encyclopedia of Data Warehousing and Mining*. Information Science Publishing, 2005.
5. Kagdi, H., Yusuf, S., and Maletic, J. I. Mining Sequences of Changed-files from Version Histories. *Proceedings 3rd International Workshop on Mining Software Repositories (MSR'06)* ACM Press: New York NY, 2006; 47-53.
6. Zaki, M. J. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning* 2001; **42**(1-2):31 - 60.
7. Zimmermann, T., Weißgerber, P., Diehl, S., and Zeller, A. Mining Version Histories to Guide Software Changes. *Proceedings 26th International Conference on Software Engineering (ICSE'04)*. IEEE Computer Society: Los Alamitos CA, 2004; 563-572.
8. Zimmermann, T., Zeller, A., Weißgerber, P., and Diehl, S. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering* 2005; **31**(6):429-445.
9. German, D. M. An Empirical Study of Fine-Grained Software Modifications. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society: Los Alamitos CA, 2004; 316-325.
10. Hassan, A. E. and Holt, R. C. Predicting Change Propagation in Software Systems. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society: Los Alamitos CA, 2004; 284-293.
11. Tonella, P., Ricca, F., Pianta, E., and Girardi, C. Recovering Traceability Links in Multilingual Web Sites *Proceedings 3rd International Workshop on Web Site Evolution (WSE'01)*. IEEE Computer Society: Los Alamitos CA, 2001; 14-21.
12. Niu, N., Stroulia, E., and El-Ramly, M. Understanding Web Usage for Dynamic Web-Site Adaptation: A Case Study *Proceedings 4th International Workshop on Web Site Evolution (WSE'02)*. IEEE Computer Society: Los Alamitos CA, 2002; 53-62.
13. 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* 2004; **30**(9):574-586
14. Gall, H., Hajek, K., and Jazayeri, M. Detection of Logical Coupling Based on Product Release History. *Proceedings 14th IEEE International Conference on Software Maintenance (ICSM'98)*. IEEE Computer Society: Los Alamitos CA, 1998; 190-199.
15. Van Rysselberghe, F. and Demeyer, S. Mining Version Control Systems for FACs (Frequently Applied Changes). *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 48–52. Available at: http://plg.uwaterloo.ca/~aeehassa/home/pubs/MSR2004ProceedingsFINAL IEE Acrobat4.pdf [1 February 2007].
16. Van Rysselberghe, F. and Demeyer, S. Studying Software Evolution Information By Visualizing the Change History. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society: Los Alamitos CA, 2004; 328-37.
17. Bieman, J. M., Andrews, A. A., and Yang, H. J. Understanding Change-Proneness in OO Software Through Visualization. *Proceedings 11th IEEE International Workshop on Program Comprehension (IWPC'03)*. IEEE Computer Society: Los Alamitos CA, 2003; 44-53.
18. Burch, M., Diehl, S., and Weißgerber, P. Visual Data Mining in Software Archives. *Proceedings ACM Symposium on Software Visualization (SoftVis'05)*. ACM Press: New York NY, 2005; 37-46
19. Beyer, D. and Noack, A. Clustering Software Artifacts Based on Frequent Common Changes. *Proceedings 13th International Workshop on Program Comprehension (IWPC'05)* IEEE Computer Society: Los Alamitos CA, 2005; 259-268.
20. Dinh-Trong, T. T. and Bieman, J. M. The FreeBSD Project: A Replication Case Study of Open Source Development. *IEEE Transactions on Software Engineering* 2005; **31**(6):481-494.
21. Chen, A., Chou, E., Wong, J., Yao, A. Y., Zhang, Q., Zhang, S., and Michail, A. CVSSearch: Searching through Source Code using CVS Comments. *Proceedings 17th IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer

Society: Los Alamitos CA, 2001; 364-373.
22. El-Ramly, M. and Stroulia, E. Mining Software Usage Data. *Proceedings 1st International Workshop on Mining Software Repositories (MSR'04)*. University of Waterloo: Waterloo ON, 2004; 64-68. Available at: http://plg.uwaterloo.ca/~aeehassa/home/pubs/MSR2004ProceedingsFINAL IEE Acrobat4.pdf [1 February 2007].
23. 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* 2007; **19**(2):77-131.

Huzefa Kagdi is a Doctoral Candidate in the Department of Computer Science at Kent State University in Ohio, USA. His research interests are in mining software repositories, source code representations and analysis, and UML visualization for supporting evolution of large-scale software systems. He received the M.S. in Computer Science from Kent State University, USA and the B.E. in Computer Engineering from Birla Vishwakarma Mahavidyalaya, India.

Jonathan I. Maletic is an Associate Professor in the Department of Computer Science at Kent State University in Ohio, USA. His research interests are centered on software evolution and he has authored over 60 refereed publications in the areas of analysis, transformation, comprehension, traceability, and visualization of software. He received the Ph.D. and M.S. in Computer Science from Wayne State University and the B.S. in Computer Science from The University of Michigan-Flint.