Mining for Co-Changes in the Context of Web Localization

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

Abstract

An approach for mining repositories of web-based user documentation for patterns of evolutionary change in the context of internationalization and localization is presented. Sets of documents that are changed together during the translation process are uncovered and documented to support future evolution of the system. A sequential-pattern mining technique is used to uncover the patterns from Subversion repositories. The approach is applied to the open source KDE system. KDE maintains documentation for over fifty different natural languages and presents a prime example of the problem. Characteristics of the uncovered patterns such as size, frequency, and occurrences within a single 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 and help user communities estimated the progress of documentation in their respective languages.

1. Introduction

Open-source products have a world-wide user base. In order to better serve and retain a diverse user community, open-source projects are increasingly developed with the ability to adapt to a number of natural languages and environments. Products such as *Linux*, *OpenOffice*, and *KDE* provide localization of user interfaces and user documentation (including online help) in several languages and locales (e.g., characters set, date/time, and currency). For example, *KDE* provides online user guides in nearly twenty-six languages [13].

The majority of the open-source projects are based on the *gnu gettext* model for localization purposes [10]. Such projects produce source code (i.e., string literals) and documentation in a base language (typically *US English*), extract the strings and documentation that require a language-specific translation, and translate them to another language. Localization is a semi-automatic process where the translation process (i.e., converting text from one language to another) largely remains a human-intensive activity. It is not uncommon for large projects to have thousands of documents that need to be kept in alignment with the corresponding versions of the software. As such, the evolution process of document translation is similar to the evolution process of source code. There are multiple teams and contributors involved in the evolution process. Therefore, the translation process is managed via versions-control systems in document[1] repositories.

The versions stored in document repositories can be utilized to extract pertinent information and/or uncover relationships or trends about a particular evolutionary characteristic. In this paper, we present an approach for mining the document repositories (web pages) in order to uncover patterns of translated documents that frequently co-change in a single language or across multiple languages. Our approach is applied on over four thousand versions of *KDE* document repository. The *KDE* document repository includes over fifty different languages for localization.

The recovered patterns provide information not only about the 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.po}→{krita.po}* mined from the *KDE* document repository indicates that the documents *kexi.po* and *krita.po* are changed in two successive versions. This pattern is found in the change history of 506 translated documents across 44 languages. Such patterns directly support the evolution of documentation. They can be utilized to assess the impact of changes with regards to translation. This supports the translators and users in the following ways:

- Supports their decision to allow (or not allow) changes in the documentation during the string (hard) freeze,
- Assists translators to plan and examine potential out-of-date/obsolete documents,
- Assists user communities to anticipate updates in the documentation.

The rest of the paper is organized as follows. In section 2, we examine the document translation paradigm

[1] Here document or documentation includes the web pages or sources from which web pages are generated.

based on *gettext*, section 3 discusses the evolution data found in the document repository, section 4 presents our pattern mining approach, section 5 evaluates our approach on *KDE* project, section 6 presents related work, and finally our conclusions and future work are presented in section 7.

2. Document Translation

Open-source systems are very often developed and distributed with the *Native Language Support* (*NLS*) paradigm. The Native-language support encompasses *Internationalization* (better known as *i18n*) and *localization* (better known as *l10n*) [10]. Internationalization is a generalization process that gives a program 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 websites that contain documentation (online help) of open-source systems typically follow the same internationalization and localization process that is established for the source code of a project. For example, in case of *KDE,* the documentation (with internalization support) is written in the *US English* and then translated (localized) to a number of other languages with the help of tools [14]. This is an exact replication of the *KDE* source-code internationalization and localization process.

In the *KDE* project, the online documents produced in the *US English* are said to be *untranslated* documents and serve as templates for translation in other languages. A translation team localizes the untranslated documents to a specific language (e.g., *German*). The documents specialized to a specific language from the untranslated documents are known as *translated* documents.

```
<title>KDE User's Manual</title>
```

Figure 1. A portion of man-kdeoptions.7.docbook from kdelibs – Untranslated document in US English.

The translation process is based on the *Portable Object* (*PO*) files [10]. 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 instance copies of these documents are then converted back to the original format of the documents.

```
#. Tag: title
#: man-kdeoptions.7.docbook:8
…
msgid "KDE User's Manual"
msgstr ""
…
```

Figure 2. A portion of kdelibs_man-kdeoptions.7.pot portable object file from kdelibs – Internationalized template for translation to other languages.

For example, consider a portion of a document in the *docbook* format (www.docbook.org) shown in Figure 1. The content of the element *title* is in the language US English. In order for this document to support native language support, it is converted to the intermediate POT format as shown in Figure 2. An entry (*msgid*, *msgstr*) of strings is created. The string msgid contains the untranslated content of the element *title*, whereas, the string *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.

```
#. Tag: title
#: man-kdeoptions.7.docbook:8
…
msgid "KDE User's Manual"
msgstr "KDE Benutzerhandbuch"
…
```

Figure 3. A portion of kdelibs_man-kdeoptions.7.po portable object file from kdelibs – Specialized portable file for German.

An instance (copy) of the template is created as shown in Figure 3 . In this case, the string *msgid* is translated to the language German and the translated content is manually entered in the string msgstr. Once such a translation is completed, the document in the PO format is converted to the docbook format as shown in Figure 4.

```
<title>KDE Benutzerhandbuch</title>
```

Figure 4. A portion of man-kdeoptions.7.docbook from kdelibs – Translated document to German.

3. Repositories of Translated Documents

The translation of documents is similar to source-code evolution in that it is a continuous process in which changes are performed incrementally. That is, translation

is often performed on a single entity or group of related entities during a single step. Also new untranslated documents (POT files) may be introduced that require creation and translation of new language-specific instances (PO files). Incremental translation of changes is easier to handle when compared with modification of existing template files that require changes in the corresponding language-specific documents. Changes to the template are likely to occur with the changes in the corresponding source code and its user documentation. Such changes may introduce additional entries for translation and invalidate existing entries. Tools such as *gettext* and *KBabel* provide assistance in merging a language-specific document with respect to the changes in the corresponding template file from which it is created. These tools help identify portions of the document that need to be retranslated, contain new untranslated entries, and entries that become obsolete. Nonetheless, the affected entries in the already translated documents may have to undergo changes. Thus, it is clearly evident from the above discussion that the translated document evolves over time.

The history about the evolution of the translated document is often stored in document repositories via version-control systems such as *Subversion* (http://subversion.tigris.org/) or *CVS* (www.nongnu.org/cvs/). Documentation and translation is a team activity in large open-source projects and it is not uncommon to have multiple contributors developing the documentation and/or translating the same part of the system. Therefore, the documentation and their translation are typically managed by versions-control system

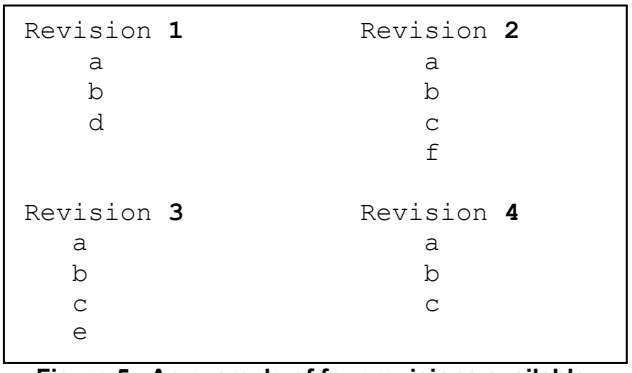

Figure 5. An example of four revisions available directly from a document repository.

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). In *Subversion*, version-number assignment and metadata are associated at the change-set level and recorded as a *logentry*. *Subversion*'s log-entries include the attributes *committer*, *date*, and *paths* (i.e., files) involved in a change-set. Each logentry is uniquely identified by a *revision* number. Figure 5 shows four logentries (i.e., change-sets) of a hypothetical document 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.

Given the evolution history of translation we can now discuss how to mine this information to determine frequently occurring patterns that are useful for understanding the behavior of changes to the system.

4. Mining Patterns from Repositories

Here we are particularly interested in documents that co-change in a specific temporal order (i.e., sequences of documents). These types of change patterns may assist the translators in determining groups of entities that need to be changed together in a single version or multiple versions. We first discuss the information that is directly available from a document repository and the required information to help fulfill our interest.

A logentry from *Subversion* gives us the documents that are committed together as a single change-set. The logentries can be readily obtained from the document repository. We term this as change-set-oriented view of the change history. The size (i.e., number of documents) of the change-sets may vary across the versions history. The size of a change-set depends on the practice of the developer/translator. On one end, some change-sets may contain only a few files that are changed slightly (e.g., only a single entry translated in a singe document). This is a case where a single logical change is performed incrementally and is completed by committing multiple versions. On the other end, some others may contain a large number of files 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 log-entry is not necessarily the order in which they were changed. Therefore, simply considering a single logentry may prove to be insufficient in order to determine all the documents that are typically changed together and the specific (temporal) ordering of the documents involved in a change-set.

There is a temporal order between change-sets indirectly imposed by their revision numbers. Change-sets with greater revision numbers occurred after those with lesser revision numbers. Therefore, we can utilize the ordering of change-sets to determine ordering of documents in which they changed. A straightforward approach is to exhaustively list all the patterns of the

documents. For the example shown in Figure 5, change-set {*a, b, d*} occurs before the change-set {*a, b, c, f*}, the possible patterns are {*a*}→{*a*}, {*a*}→{*b*},..,{*b, d*}→{*a*}, and so forth. However, this raises consideration of two major issues: 1) sequences that may not be useful for evolution tasks such as change predication (i.e., false positives) and 2) examination of all possible patterns between a series of change-sets. Notice that the atomic commits are serialized by a versions-control system. The temporal order in which log-entries appear in a log file is at discretion of a versions-control system. As a result successive log-entries may be unrelated in the context of changes performed to the documents (e.g., change-sets of totally independent parts of the system may appear successively only because they happen to be completed in that order and not necessarily due to the change dependency between them). Therefore, it may result in meaningless patterns.

In an effort to obtain complete coverage of the documents that typically change together, we take an alternative view of history compared to what is directly available from the versions-control system. Furthermore, we employ sequential-pattern mining to effectively deal with the combinatorial explosion of search space.

4.1. Grouping Change-sets by Document

The history of documents available from repository is in a change-set-oriented view. However, as discussed in the beginning of section 4, 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. We term this alternate arrangement of history as document-oriented view.

In this view 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 5, the corresponding document-oriented history is shown in Figure 6. The history of document *a* consists of a sequence of change-sets from revisions *1*, *2*, *3*, and *4*.

Once such a document-oriented view of history is available, sequential-pattern mining could be employed to find typical subsequences of documents that occur in a number of documents histories.

```
Document a              Document b
 1 {a b d}               1 {a b d}
 2 {a b c f}             2 {a b c f}
 3 {a b c e}             3 {a b c e}
 4 {a c b}               4 {a c b}

Document c              Document d
 2 {a b c f}             1 {a b d}
 3 {a b c e}
 4 {a c b}

Document e              Document f
 3 {a b c e}             2 {a b c f}
```

Figure 6. The changeset-oriented history converted to a document-oriented history.

4.2. Sequential-Pattern Mining

A sequential 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, a sequential pattern establishes both the ordered and unordered relationship between items. In our case, an element of a pattern maps to a version and an item of a pattern maps to a document. Therefore an element of a sequence indicates the documents that change in the same version, whereas, the happen before relationship between two given elements indicates changes across two different versions.

Mining patterns of documents that typically change together can be considered as an instance of the general problem of sequential-pattern mining (from any type of data) [1]. Before we describe the sequential-pattern mining approach, 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). Here, a transaction corresponds to (a sequence of) change-sets of a single document. 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.

Sequential-pattern mining 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 [15].

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.

We have developed a sequential pattern-mining tool, namely *sqminer*, that is based on the Sequential Pattern Discovery Algorithm (SPADE) [21] 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 call vs. a set of transactions consisting of calls) 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 calls 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 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. Our approach records the transactions in which a pattern is found.

On application of *sqminer* to the document-oriented view of the change-sets, a set of patterns that changes frequently in at least a specified number of documents histories are produced. For the example shown in Figure 6, the pattern $\{a\}\rightarrow\{c\}\rightarrow\{a, b\}$ that occurs in the history of documents *a*, *b*, and *c* is one of the uncovered patterns that is completed in three versions. In this case, the document *a* is changed in a version before a version in which the document *c* is changed. The set of documents *{a, b}* in changed in a version before a version in which the document *c* is changed. This pattern is an example of a partially ordered pattern in which the specific ordering between the documents *a* and *b* in the set $\{a, b\}$ can not be determined from the histories of the documents *a*, *b*, and *c*. This pattern is produced for a minimum support of up to three as it is found in the history of three documents.

5. Evaluation

Our approach is evaluated on the document repository of an open source system. Table 1 shows a subset of the *KDE* document repository *l10n* (localization) that was considered for mining patterns. It gives the number of revisions, the number of (unique) documents involved in the change-sets (i.e., changed documents column), and the number of changes performed in these documents. There are over four thousand change-sets (i.e., revisions) committed in a three-month period. These change-sets consist of over twenty-one thousand documents. These documents are changed over eighty thousand times. This indicates that on an average a document is changed approximately three times and on an average a change-set consists of over five documents. Only change-sets that consisted of less than or equal to ten documents are considered. This is partially done in order to discard noisy change-sets such as those updating the license information and/or performing a merge or branch.

Repository	Period: (01-01-2006 to 03-04-2006)		
	Revisions	Changed-Documents	Documents Changes
l10n	4044	21734	80476

Figure 7. The l10n repository from KDE project considered for mining patterns of changed documents.

The *KDE* document repository is managed by *Subversion*. One approach to extract the log records (i.e., change-sets) from a *Subversion* repository is using the client command *svn log*. However, this approach requires a working copy of the repository. Clearly, this approach is not feasible for use-cases in which only the logentries are needed and not the actual documents. Therefore, we developed the tool *changeextractor* that uses *pysvn* (i.e., *Subversion* API for *Python*) to extract changesets (without a working copy) from the repository. 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. Furthermore, the tool *changeextractor* rearranges the logentries in the document-oriented view of the history.

With the logentries represented in the document-oriented view, the sequential-pattern mining technique is applied to find all the frequently occurring patterns (or subsets of them) in the histories of documents. The document-oriented view of history 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. For further detail on the XML output format of the ordered patterns and rules, we refer to [12].

We configured *sqminer* to perform mining of patterns with a minimum support of 500. That is, in order for a pattern to be considered frequent, it has to occur in the versions histories of at least five-hundred documents translated (not necessarily in the same language). *sqminer* uncovered 127 patterns with 72 single element and 55 binary element patterns respectively. The number of elements of a pattern is the number of versions in which the changes in all its documents are completed.

Table 1. Distribution of patterns with regards to versions and sizes

Version	Size (number of changed documents)			
	1	2	3	4
1	17	34	20	1
2	-	25	25	5

Table 1 shows the distribution of patterns in terms of sizes and versions. This type of information may 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. This information may help decision making with regards to allowing changes during a hard freeze. Singleton patterns form about 13% (i.e., 17 out of 127) of the total patterns. Binary patterns form about 54% (69 out of 127 with the maximum share) of the total patterns. Patterns as large as consisting of 4 documents that are changed together in a single version as well as in a sequence of two versions are uncovered. For example, the pattern *{kword.po, kspread.po, krita.po, kchart.po}* consists of four documents *kword.po*, *kspread.po*, *krita.po*, and *kchart.po* that are changed together in a single version. The pattern *{kexi.po}→{krita.po}* is an example of documents that are changed in a series of two versions. In this case, the document *kexi.po* was changed prior to a version in which the document *krita.po* was changed. All the documents in both of the above patterns correspond to applications in *KOffice*.

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. Patterns occurring in the histories of over a thousand different documents were discovered. The pattern *{kspread.po, krita.po}* is one such example that is common in the histories of 1025 documents. That is, the changes in the documents *kspread.po* and *krita.po* co-occur in a version of all of these 1025 documents history. The patterns *{kword.po, kspread.po, krita.po, kchart.po}* and *{kexi.po}→{krita.po}* are supported by the histories of 552 and 506 documents respectively.

Figure 8 shows the number of documents histories corresponding to the number of patterns they support. It can be seen that there is a wide distribution of number of documents supporting the number of patterns (i.e., 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).

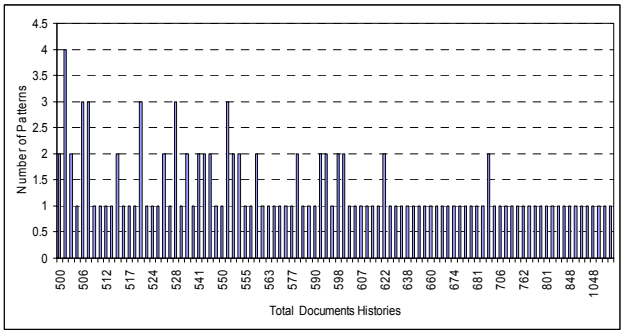

Figure 8. The number of patterns that are found in the number of documents histories.

Another interesting aspect is to examine the different languages that share a common pattern (i.e., a same set of documents are changed in histories of different languages). For examples,

- The pattern *{kword.po, kspread.po, krita.po, kchart.po}* is exhibited in the histories of 40 different language documents,
- The pattern *{kexi.po}→{krita.po}* is exhibited in the histories of 44 different language documents, and
- The pattern *{kspread.po, krita.po}* is exhibited in the histories of 55 different language documents.

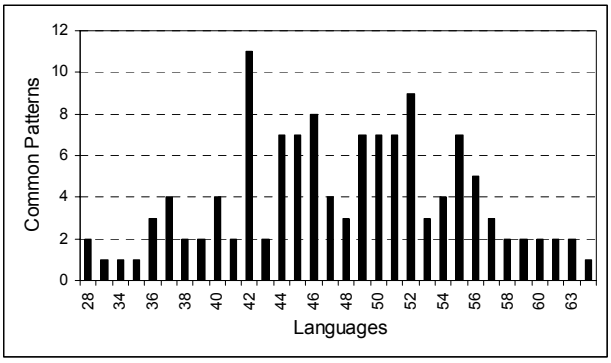

Figure 9. The number of common patterns found in the histories of different languages.

Figure 9 shows the distribution of the 127 uncovered patterns with regards to different languages. There are as

many as 11 patterns that are common in the histories of 42 different languages. There are as many as 63 different languages that have a common pattern. In majority of the cases there is more than one pattern that is common in the histories for a given number of different languages. There is not a single pattern that is confined to a single language. Figure 10 shows the number of patterns found in the history of the individual languages. All the 127 patterns were found in the histories of the languages *Hebrew*, *Romanian*, *Tamil*, and *Hungarian*. Languages *Bengali*, *Telugu*, and *Georgian* supported only a single pattern.

6. Related Work

We discuss the work on 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 [17] 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 [16] 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 [22, 23] 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 [20] used a similar technique for identifying files that frequently change together. Gall et al [8] 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 [11] analyzed *CVS* logs for software-change prediction.

Van Rysselberghe et al [18] utilized *CVS* logs in their approach to find frequently applied changes and presented a 2D visualization technique to help recognize change-relevant information [19]. Bieman et al [3] used logs from software repositories to assist in the computation of metrics for detecting change-prone classes. Burch et al [4] 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 [2].used the log information in visualizing clusters of frequently occurring co-changes. Dinh-Trong et al [6] used *CVS* logs for validating previously developed hypotheses on successful open source development. Chen et al [5] incorporated the *CVS* commit messages in their source-code search tool. El-Ramly et al [7] used sequence mining to detect patterns of user activities from the system-user interaction data.

7. Conclusions and Future Work

We applied a sequential-pattern mining approach to uncover frequently co-changed documents in the context of internationalization and localization. We evaluated our approach on a large open-source system. Characteristics of the patterns with regards to size, frequency, and occurrences within a single language or across multiple languages are presented. The results

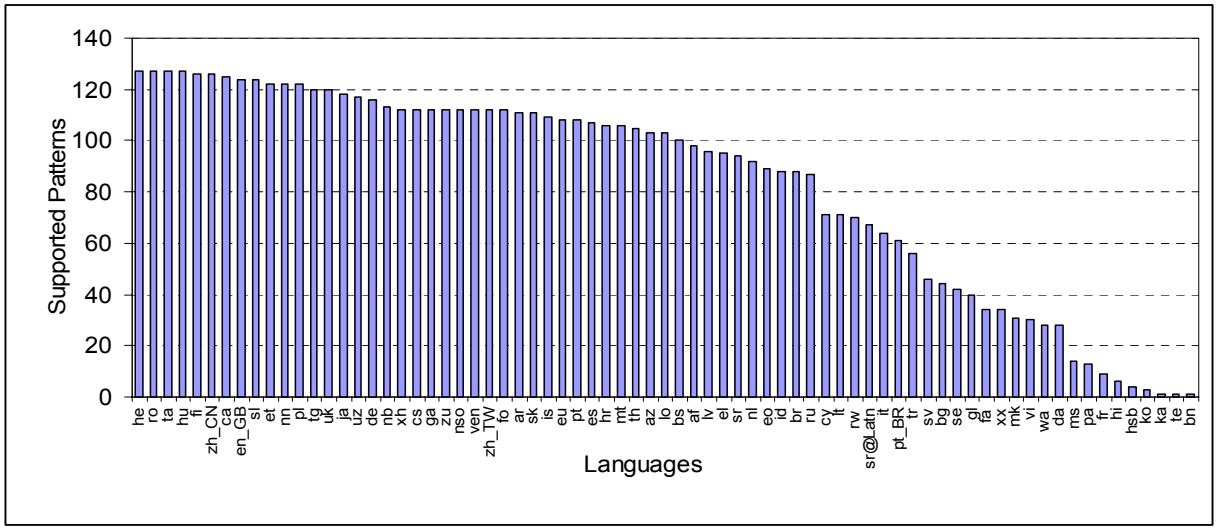

Figure 10. Total number of supported patterns by the history of a language.

indicate that using the historical information in versions archive is a promising source for supporting internationalization and localization of web sites.

In future we plan to examine the effectiveness of this approach for tasks such as change prediction. One approach is to validate the patterns mined from prior versions of document repositories with the subsequent versions. Furthermore, other open-source systems such as *OpenOffice* and *Apache* will be investigated. The approach will be extended to produce finer granularity patterns (e.g., at entry level of PO files) than document level.

8. References

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

[2] Beyer, D. and Noack, A., "Clustering Software Artifacts Based on Frequent Common Changes", in Proceedings of 13th International Workshop on Program Comprehension (IWPC'05), St. Louis, Missouri, USA, May 15-16 2005, pp. 259-268.

[3] Bieman, J. M., Andrews, A. A., and Yang, H. J., "Understanding Change-Proneness in OO Software Through Visualization", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), 2003, pp. 44-53.

[4] Burch, M., Diehl, S., and Weißgerber, P., "Visual Data Mining in Software Archives", in Proceedings of Proceedings of the 2005 ACM symposium on Software visualization, St. Louis, Missouri, May 14-15 2005, pp. 37-46.

[5] Chen, A., Chou, E., Wong, J., Yao, A. Y., Zhang, Q., Zhang, S., and Michail, A., "CVSSearch: Searching through Source Code using CVS Comments", in Proceedings of Proceedings IEEE International Conference on Software Maintenance (ICSM'01), 2001, pp. 364-373.

[6] Dinh-Trong, T. T. and Bieman, J. M., "The FreeBSD Project: a Replication Case Study of Open Source Development", IEEE Transactions on Software Engineering, vol. 31, no. 6, 2005, pp. 481-494.

[7] El-Ramly, M. and Stroulia, E., "Mining Software Usage Data", in Proceedings of International Workshop on Mining Software Repositories (MSR'04), 2004, pp. 64-8.

[8] Gall, H., Hajek, K., and Jazayeri, M., "Detection of Logical Coupling based on Product Release History", in Proceedings of International Conference on Software Maintenance (ICSM'98), 1998, pp. 190-199.

[9] German, D. M., "An Empirical Study of Fine-Grained Software Modifications", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004, pp. 316-25.

[10] GNU, "GNU gettext ", Free Software Foundation, Date Accessed: May, 25, 2006, http://www.gnu.org/software/gettext/manual/gettext.html, 2006.

[11] Hassan, A. E. and Holt, R. C., "Predicting Change Propagation in Software Systems", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004, pp. 284-93.

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

[13] KDE, "KDE Documentation", Date Accessed: May, 25, 2006, http://docs.kde.org/, 2006.

[14] KDE, "KDE Localization", Date Accessed: May, 25, 2006, http://l10n.kde.org/, 2006.

[15] 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.

[16] Niu, N., Stroulia, E., and El-Ramly, M., "Understanding Web Usage for Dynamic Web-Site Adaptation: A Case Study ", in Proceedings of Fourth International Workshop on Web Site Evolution (WSE'02), Montréal, Canada, October, 2 2002, pp. 53-62.

[17] Tonella, P., Ricca, F., Pianta, E., and Girardi, C., "Recovering Traceability Links in Multilingual Web Sites ", in Proceedings of 3rd International Workshop on Web Site Evolution (WSE'01), Florence, Italy, November, 10 2001, pp. 14-21.

[18] Van Rysselberghe, F. and Demeyer, S., "Mining Version Control Systems for FACs (Frequently Applied Changes)", in Proceedings of International Workshop on Mining Software Repositories (MSR'04), May 25 2004, pp. 48-52.

[19] Van Rysselberghe, F. and Demeyer, S., "Studying Software Evolution Information By Visualizing the Change History", in Proceedings of 20th IEEE International Conference on Software Maintenance, 2004, pp. 328-37.

[20] 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.

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

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

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