University of Konstanz
Department of Computer and Information Science

Master Thesis

Analysing differences in tree-structured data by Visual Analytics

in fulfillment of the requirements to achieve the degree of

Master of Science (M.Sc.)

Johannes Lichtenberger
Matriculation Number :: 01/584875
E-Mail :: ⟨firstname⟩.⟨lastname⟩@uni-konstanz.de

Field of Study :: Information Engineering
Focus :: *Applied Computer Science*
Topic :: *Distributed Systems*

First Assessor :: *Prof. Dr. M. Waldvogel*
Second Assessor :: *Jun.-Prof. Dr. Tobias Schreck*
Advisor :: *M.Sc. Sebastian Graf*

dedicated to...

Abstract. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur non velit eget urna dictum bibendum. Vivamus lacinia nunc non felis. Suspendisse neque. Cras non nulla. In et lorem in nunc aliquet gravida. Morbi venenatis aliquam enim. Aenean ac justo. Mauris pretium varius mi. Proin sagittis gravida lectus. Ut non ante. Praesent tincidunt rutrum augue. Ut dolor. Maecenas est. Integer semper metus et dolor. Sed vitae orci ac risus ultrices vehicula. Duis dolor turpis, pharetra sed, blandit eget, consectetur sit amet, eros. Etiam ultrices velit eu quam. Curabitur laoreet nibh sit amet turpis posuere sagittis. Quisque tellus turpis, ornare vel, mollis sed, tristique eu, orci. Vestibulum sodales nisl vitae diam.
Fusce vitae diam. Aliquam porttitor. Sed neque urna, lobortis sed, pellentesque ac, facilisis id, nibh. Suspendisse mi. Suspendisse diam velit, venenatis a, malesuada sed, faucibus eget, magna. Praesent semper venenatis nisl. In hac habitasse platea dictumst. Suspendisse potenti. Pellentesque interdum, orci eu tristique venenatis, elit neque interdum quam, sit amet semper nisl mi a velit. Praesent a quam nec lacus interdum malesuada. Integer diam. Cras ante nulla, ultrices et, vestibulum id, pulvinar auctor, nisl. Sed ornare aliquet est. Donec interdum tortor at ante. Phasellus tristique viverra lorem. In rutrum viverra velit.

Table of Contents

List of Figures

List of Tables

Todo list

1 Applications

1.1 Introduction

The last chapters described in detail the different pipeline-components involved in the process of detecting differences in tree-structured data by Visual Analytics. This chapter describes some application scenarios and the additional preprocessing steps involved. To demonstrate the feasability of our approach with real world data several independent import mechanisms have been developed. In a nutshell the following three applications have been studied:

- Import of Lexical Functional Grammar (LFG) XML exported files.
- Import of sorted Wikipedia articles by revision-timestamps.
- Directory recursively watched for changes within a filesystem. The XML representation thereof is based on FSML . REFERENZ

1.2 LFG

Linguists often face the problem of having to compare tree-structured data as for instance ASTs (Abstract Syntax Trees). The LFG in particular differenciates between two structures:

1. Constituent structures (c-structures) which "have the form of context-free phrase structure trees".
2. Functional structures (f-structures) "are sets of pairs of attributes and values; attributes may be features, such as tense and gender, or functions, such as subject and object." REFERNZ

We obtained an XML-export of a whole collection of different versions of a c-structure/f-structure combination. In order to import the data in Treetank the FMSE-algorithm described in chapter 2 and 3 is used as the XML-export doesn't contain unique node identifiers. Figure 1 illustrates the visualization of a diff between two revisions.

It is immediately clear that the FSML-algorithm failed to detect the appropriate matching nodes and therefore executes far too many update operations especially in case of the deletion and reinsertion of the old- respectively the new-cstructure subtree. In doing so, the subsequent visualization is impractical, as it just reassembles the update-operations for consecutive versions. As stated in chapter 2 ID-less algorithms work best if leaf nodes and therefore especially text-nodes can be descriminated very well. That is the initial matching of leaf nodes in most if not all cases matches the right nodes, the ones which really haven't changed.

As a direct consequence assuming the export of the XML-documents includes unique identifiers of the nodes, the import requires utilization of the internal diff-algorithm to simply store incremental changes, the changes from the old- to the new-revision. The following steps have to be implemented:

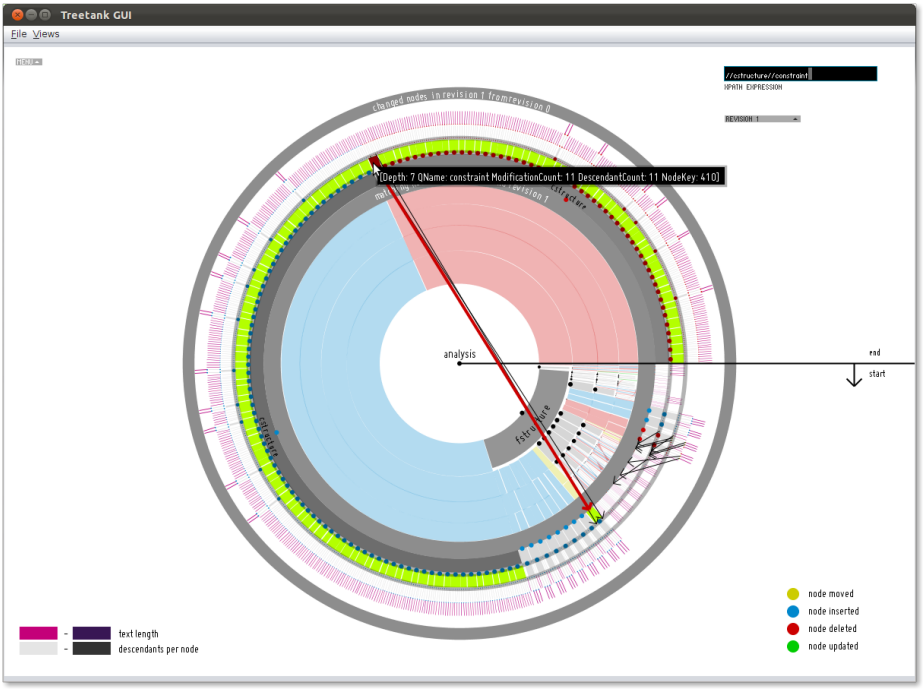

Fig. 1. LFG comparsion

1. Shredding the initial XML-document.
2. Shredding the first updated XML-document to a temporary resource.
3. Utilize the internal Treetank diff-algorithm to determine the changes based on the unique IDs. Whenever an update-operation is encountered the appropriate transaction-method has to be executed. In case of element- and text-nodes it requires how to insert nodes and where, which is very simple due to the `transaction.moveTo(long)`-method which can move the transaction/cursor directly to the left-sibling-node of the node to insert or to the parent. In the first case the node has to be inserted as a right-sibling node, in the latter case as the first child of the parent node.

Furthermore this approach applies to every XML-document which incorporates unique node identifiers.

1.3 Wikipedia

REFERENZ

is dumped as a very large XML-file. "'The XML itself contains complete, raw text of every revision"' and thus is a full dump instead of an incremental- or differential-dump describing the changes an author performed. Moreover WikiText, which is a proprietary markup format is stored as plain text. As a consequence differences between the actual content of several revisions of an article on

a node-granular level can't be found with a state of the art native XML database system, which supports revisioning of the data. The database systems best bet is to generate a delta between the two large text-nodes.

Furthermore due to the fact that XPath 2.0 should be usable without requiring an XPath Full Text 1.0 implementation and our overall goal is to analyse the temporal evolution of the stored data, WikiText markup has to be converted into semistructured XML fragments. To accomplish this the Wiki2XML parser from the MODIS team is used.

REFERENZ

Next the Wikipedia dump isn't sorted. Neither articles are sorted by date nor revisions inside articles are. Thus for the subsequent revisioned import the revisions have to be sorted with their associated article metadata (page id, author...). For the simple reason that we have to deal with large amounts of data external sorting algorithms have to be used. Instead of implementing an own approach, Hadoop is a natural choice. It's *MapReduce* framework is a "programming model and software framework for writing applications that rapidly process vast amounts of data in parallel on large clusters of compute nodes". The overall process of the programming model is divided into two functions.

1. `Map` is a function that has to split the problem into subproblems which can be distributed among worker nodes in a cluster. Logically it is a function of the form $Map(k1, v1) \rightarrow list(k2, v2)$ where $k1$ and $v1$ is an input *key* and *value* and the output from the function is a list of *key*/*value* pairs ($list(k2, v2)$). After that the MapReduce framework groups the values according to the keys.
2. `Reduce` is a function of the form: $Reduce(k2, list(v2)) \rightarrow list(v3)$. It receives a *key* and a list of grouped *values* and returns performs any computation which might be feasable and returns another list of *values*.

Using the map-function means input data has to be split into records. Most MapReduce frameworks rely on a programmable mechanism to do this. In Hadoop the `InputStream` in conjunction with a `RecordReader` takes this responsibility. The following steps describe the implementation of sorting Wikipedia by Hadoop.

1. `XMLInputStream` in conjunction with an `XMLRecordReader` splits records on *revision-elements* with all the *page*-metadata. The **timestamp** of each revision is used the key, the whole subtree of each *revision*-element is saved as the value for the map input. The algorithm is straight forward. A `SAX-Parser` is used to determine starting and ending of each record. The algorithm is outlined in **??**
   Note that the actual implementation uses a configurable record identifier such that it can be adjusted to schema changes regarding the Wikipedia dump. Furthermore it can simple be adapted to split other XML files based on other record identifier nodes as well.
2. `XMLMap` just forwards the received key/value pairs.
3. `XMLReduce` finally merges consecutive revisions with the same *page-id* and **timestamp** by means of Saxon, an XSLT-processor among other things.

The XSLT stylesheet used is pretty small yet maybe not straight forward. It is shown in listing 3.

Listing 1. XSLT stylesheet to combine consecutive pages/revisions

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs">

  <xsl:output method="xml" indent="no"
    omit-xml-declaration="yes" />
  <xsl:strip-space elements="*" />

  <xsl:template match="/">
    <xsl:copy>
      <xsl:for-each-group
        select="descendant-or-self::page"
        group-by="concat(id, revision/timestamp)">
        <page>
          <xsl:copy-of select="* except revision" />
          <xsl:for-each-group
            select="current-group()/revision"
            group-by="xs:dateTime(timestamp)">
            <xsl:apply-templates
              select="current-group()" />
          </xsl:for-each-group>
        </page>
      </xsl:for-each-group>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="revision">
    <xsl:copy-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

Once it is sorted with MapReduce it has to be imported into Treetank. First of all, as we have splitted the data on `revision`-elements and prepended the `page`-metadata we have to construct a new root node, which is simply done by prepending a `mediawiki` start-tag, then writing the result of the Hadoop-run and appending a corresponding end-tag to a new file.

Next, to actually import the data a special Wikipedia-Importer is responsible. It utilizes the FSME-implementation described in detail in chapter 3 to just import incremental changes between the latest stored revision in Treetank and a shredded List of `XMLEvents` in a temporary database/resource. Therefore as data is read from the XML document with a StAX-Parser the page-metadata as

well as the whole revision-subtree is saved in a simple main-memory collection, a `List`, assuming the XML-content of an article can be put in main memory. The assumption holds true as every revision can be parsed and rendered in a web-browser. Everytime a `page` end-tag is encountered the page-id is searched in the already shreddered revision. If it is found the FMSE algorithm is called for the found `page`-element, such that the encountered differences are shreddered subsequently. In case the XPath expression returns no results, that is the page-ID isn't found a new page is going to be appended as a right sibling to the last `page`-element.

The Importer for Wikipedia allows the usage of a timespan (hourly, daily, monthly, yearly) which is used for the revisioning. If a new timestamp is encountered, which differs from the current timestamp regarding the timespan the transaction is commited.

Figure 2 demonstrates the result of importing 50 articles of Wikipedia sorted by article-revisions and an hourly commit. Revisions 141 and 142 are compared, whereas nodes in both revisions are highlighted which are contained in the result-set of the XPath-query
`//revision/id/text()`. The FSME-algorithm used to determine and update the stored Treetank-data with the encountered differences in the first place works reasonably well, as most text-nodes can be distinguished very well. It is fair to say, that for instance the text-nodes below the `id`-element parent could be detected as updates as well if all ancestors and neighbour nodes are checked for equality even though the levenshtein algorithm used to compare text nodes detects too many character edit operations to consider that the nodes are equal.

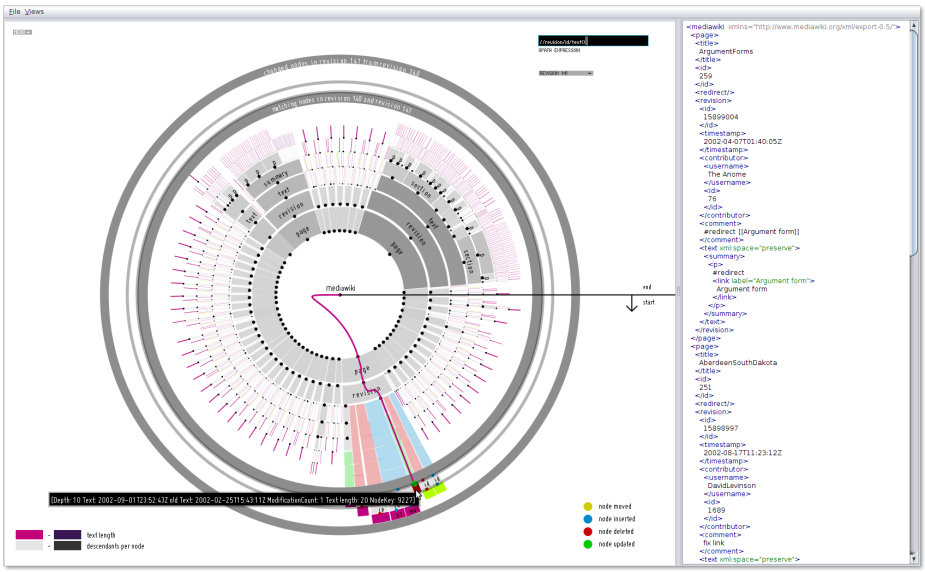

Fig. 2. Wikipedia comparsion

Another visualization (figure 3) depicts what happens if we don't include the number of modifications of a node in it's subtree as another descriminator to scale the create SunburstItems, the segments for each node.

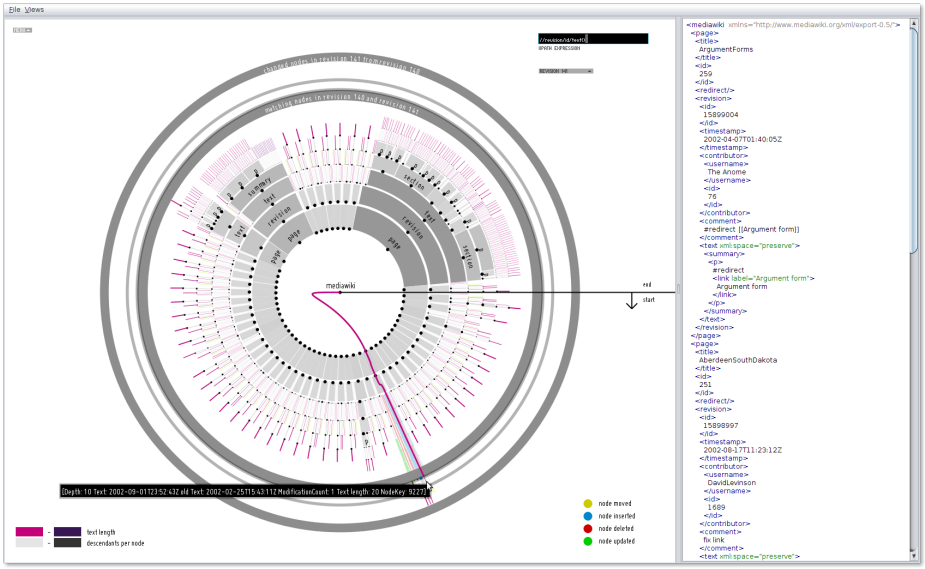

Fig. 3. Wikipedia comparsion without modification weight

To further emphasize the changes and to speed up the calculation we can use the pruning-mechanisms described in chapter 4. Using the pruning based on the itemsize results in figure 4. Instead of larger itemsizes for changed nodes potentially uninteresting nodes are pruned away, which are too small and aren't modified between the two compared revisions. Note that these nodes can still be displayed as a user drills down into the tree by selecting a new root-node, whereas all subtree-nodes are scaled according to the new root.

In order to maximally highligh nodes which are changed between the two revisions fig5 displays the result of applying the hashbased diff-algorithm. Whenever same hashes are encountered for the two nodes of each revisions the whole subtree is skipped for traversal and as such also in the visualization. As a direct consequence maximal performance as well as itemsize for changed nodes is achieved. Context nodes are preserved, such that all siblings of the root-nodes regardings changes are still visible. As a matter of fact we can still enlarge the items which depict changed-nodes by tuning the variable which denotes, how much modifications are weighted in respect to the number of descedants per node in addition to the current node.

In order to compare sequences of revisions it's also possible to view incremental and differential changes between the loaded base-revision and the other

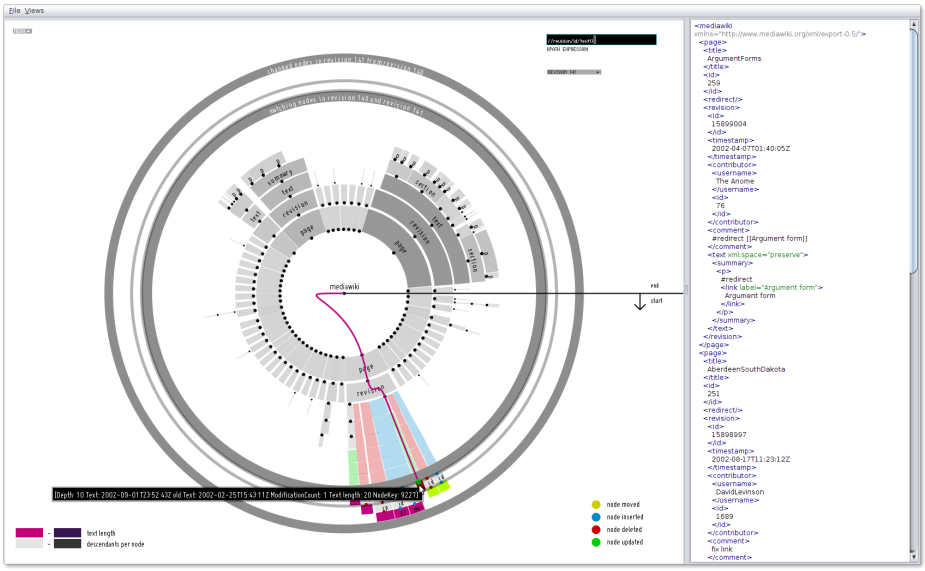

Fig. 4. Wikipedia comparsion pruned by itemsize

revisions. Figure 6 reveals the initial addition of several article, whereas the articles are not present before. In the last comparsion, the lower left SmallMultiple an article is updated instead of added. In order to zoom into interesting regions and to gain further insights the normal view comparing two revisions can be used which is going to be linked with the SmallMultiplesViews.

Another view (figure 7) displays the evolution whereas revision 0 is the base revision with just one article added. Afterwards other articles are appended as long as no article has been updated. Articles which haven't change within the two compared revisions are blackened.

1.4 Import of Filesystem-based tree-structure

Filesystems usually organize data in a tree-structured hierarchy whereas a unique *Path* denotes how a file can be located. As a direct consequence it's an optimal use case to demonstarte the feasability of our proposed approach as the tree-structure can't be neglect if we want to quickly obtain differences in the folder/file-structure based on snapshots.

Use cases for quickly gaining knowledge from a comparsion of snapshots are manyfold, ranging from monitoring software-evolution based on the package/-class hierarchy to monitoring if employees stick to organizational policies.

In order to take snapshots of a directory with all subdirectories we study two approachs:

1. FSML representation based on a python script , which has been executed REFERENZ

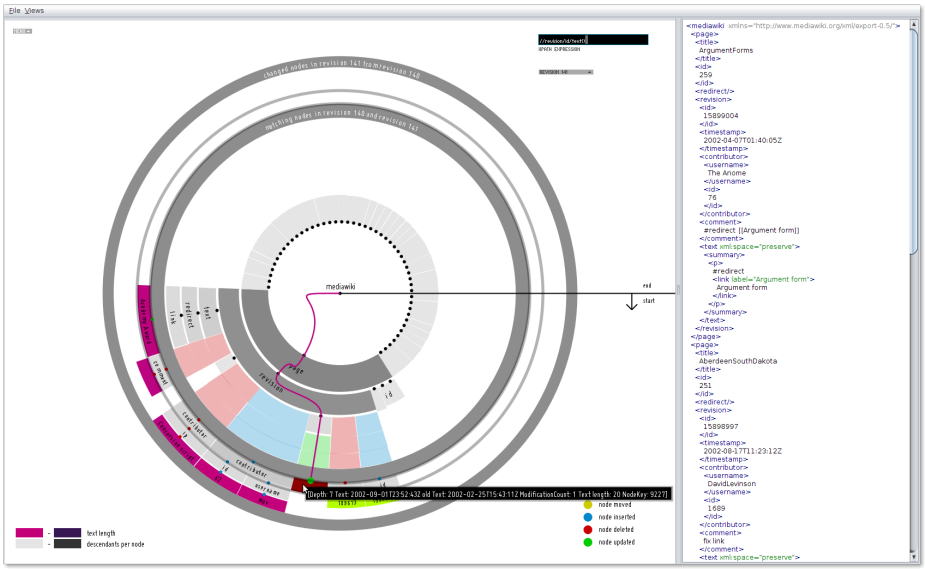

Fig. 5. Wikipedia comparsion pruned by hash-comparsion

several times to obtain snapshots based on an XML-representation which are afterwards imported in Treetank. The differences between the snapshots are calculated based on the FMSE-algorithm described in chapter 2 and 3.

2. FSML representation based on an initial import of a directory with all it's subdirectories. Changes are afterwards monitored.

REFERENZ

The File System Markup Language is an XML-dialect developed by Alexander Holupirek to represent the tree-structure of filesystems by folders and files as well as metadata of certain file-types.

The first approach is considerably simpler than the import of Wikipedia due to the snapshot-creation which results in different files. Therefore we don't have to order `element`-nodes (the revisions) with subtrees at the first place for a subsequent import. Instead we can simply apply the FMSE-algorithm on the latest imported revision and the new XML document.

Figure 8 reveals the differences between revision 0 and 4 on manually taken snapshots of the src-folder of the GUI project. Thus we see possible hotspots of development on the package/class-level. A node is highlighted, which has been deleted and represents the result of the simple XPath-query `//element()[@name='ForkJoinJava']`. The first "/" is not displayed as the query is too long to fit into the text field. Another interesting observation is the recent work on BSplines to use with Hierarchical Edge Bundling desribed in chapter 3 as well as a first addition of a planned view based on TreeMaps.

Notice, that the result represents the real changes in the module. That is the first stage of matching nodes in the FMSE-algorithm works reasonably well on

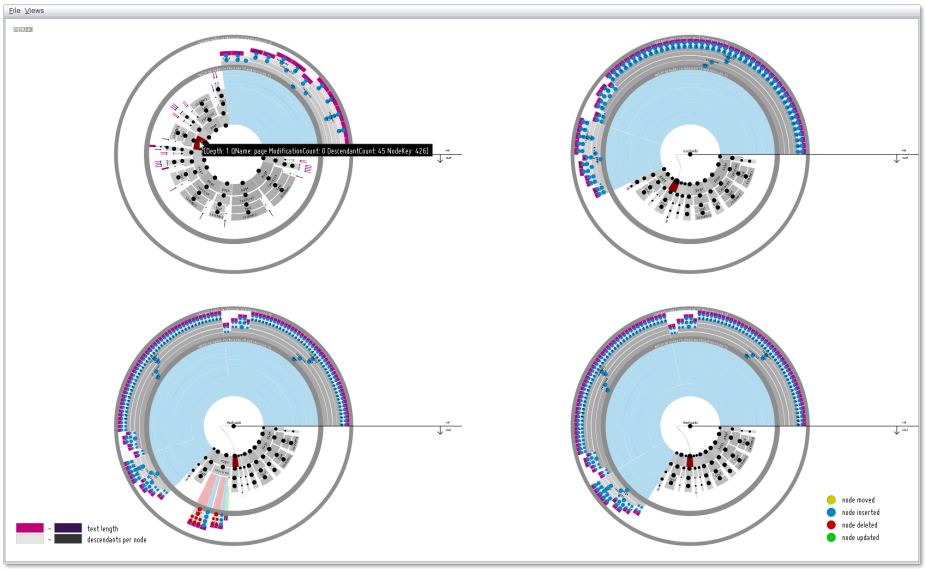

Fig. 6. Wikipedia comparsion (differential smallmultiples view)

FSML-data as no similar text-nodes are included. That said we aim to support the extraction of metadata in the future for several kinds of files which might introduce the possibility of mismatches.

In order to avoid any mismatches and therefore too many update-operations on nodes which haven't changed at all as well as the execution of the costly FMSE-algorithm in the first place we decided to use the capabilities of current filesystems to register for modification-events. The steps are as follows:

1. To obtain the hierarchical structure of filesystems the Java7 Filesystem-Walker API is used instead of the Python-Script. A new database is created in Treetank with a standard resource named "fsml" whereas the hierarchical structure is mapped to the resource while traversing a directory.
2. The new `WatchService` is used to detect subsequent changes in a watched directory. In order to support watching all subdirectories as well a suitable datastructure as for instance an associative array has to be used in the first place. Java doesn't permit recursive watching of all subdirectories, as it's not supported by some filesystems.

The `WatchService` detects the following events:

- `ENTRY_CREATE`: New file or directory has been created.
- `ENTRY_DELETE`: File or directory has been deleted.
- `ENTRY_MODIFY`: File has been modified.

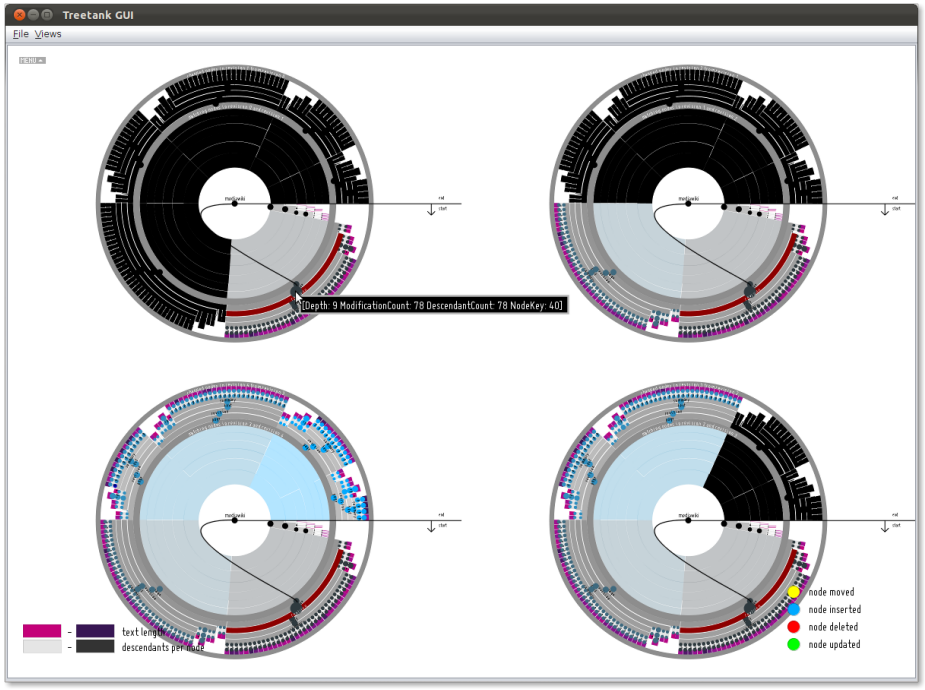

Fig. 7. Wikipedia comparsion (hybrid smallmultiples view)

Therefore moves and renames aren't supported out of the box. The path to the directories and files is translated to an XPath query, which locates the appropriate node in the database. In case a new file or directory has been created a new `ElementNode` is prepended as a first child of the parent node as the trees are unordered. In this case the XPath-query translates the parent-path into the appropriate XPath-query, that is it is of the form `/dir[@name='pathComp1Name']/dir|file[@name='pathComp2Name']`. The path component usually denotes a directory whereas the last component might be a directory or file in case of an `ENTRY_DELETE` event. Deleted directories have to be removed from a WatchService/Path-mapping in an associative array. Furthermore all paths have to be saved in another datastructure to denote if the deleted path pointed to either a file or directory. Thus another associative array is used to save a Path/EPath mapping for inserted nodes, whereas EPath is an enum used to determine the type.

The FSML-subdialect currently used is very simple. Listing 1.4 provides an example of the simple structure. Directories are mapped to `dir`-elements, whereas files are mapped to `file`-elements. Note that the names can't be used to denote the elements, as for instance whitespaces are not permitted in QNames. Thus the labels in the visualizations are optionally based on `name=""`-attributes.

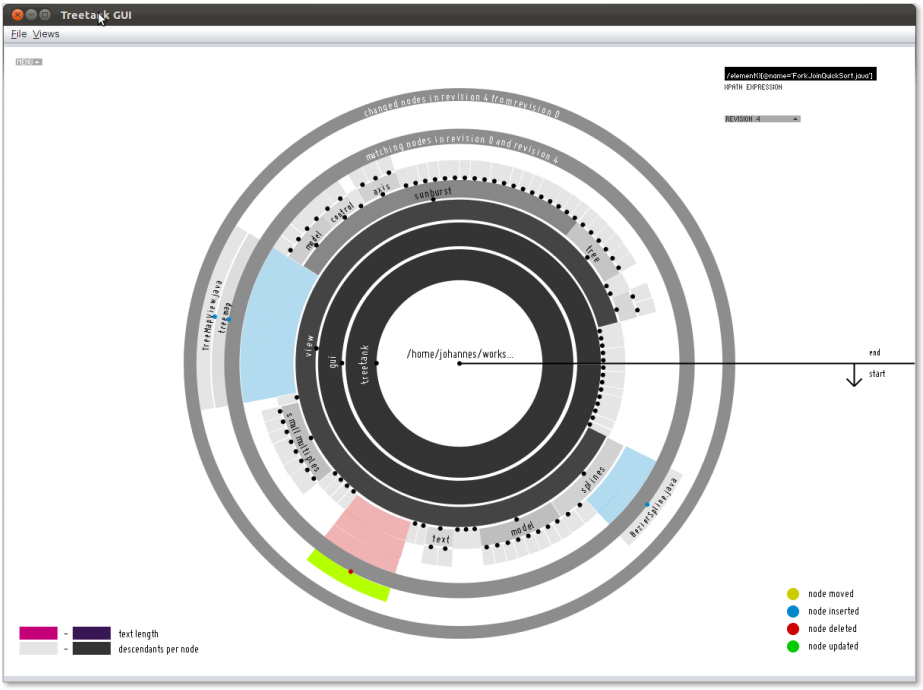

Fig. 8. FSML comparsion

Listing 2. FSML structure

```
<fsml>
  <dir name="Desktop">
    <dir name="Lichtenberger">
      <dir name="Bachelor">
        ...
      </dir>
      <dir name="Master">
        <dir name="Thesis">
          <dir name="figures">
            <file name="fsml-incremental.png" />
            ...
          </dir>
          <dir name="results">
            <file name="1gb-400" />
            ...
          </dir>
          <file name="thesis.tex" />
          <file name="motivation.tex" />
          ...
```

```
20            </dir>
21             ...
22          </dir>
23        </dir>
24         ...
25     </dir>
26 </fsml>
```

While this representation doesn't incorporate most of the strengths FSML usually provides it's easy to provide further metadata about files and to incorporate text-files. Optionally instances of custom classes are pluggable to provide any type of extensibility. Thus it's possible to provide extractors for certain types of files and to incorporate text-files into the FSML-representation itself, possibly by adding a link to another resource in the fsml-database.

Figure 9 is an example of mapping a Desktop-folder to a database in Treetank. Subsequent revisions are commited every five minutes.

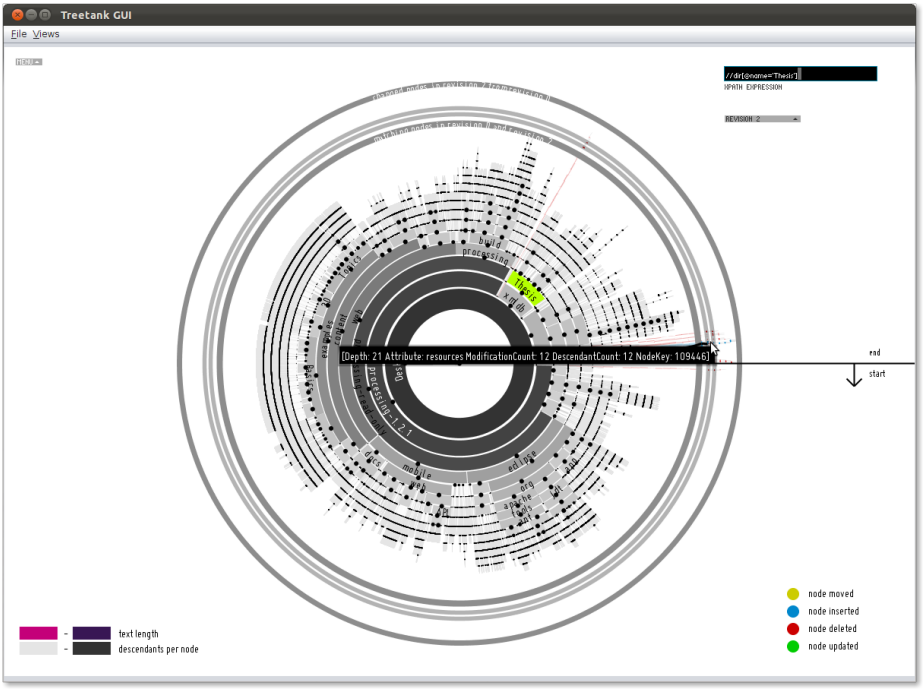

Fig. 9. FSML comparsion of "/home/johannes/Desktop"

Most files and directories are unchanged. The changed items are really small, as the subtrees are rather small compared to the unchanged parts. Results from adjusting the modification-weight are displayed in figure 10. The changed sub-

trees are much better visible. Two subdirectories matching the simple XPath-query `//dir[@name='wiki-sorted']` are highlighted. It's immediately obvious that a subtree rooted at the directory named "wiki-sorted" has been deleted and another one has been inserted. The equal subtree-size suggests that the subtree has been moved. On close inspection, for instance using the subtrees rooted at "wiki-sorted" we see that the subtrees are isomorph (figure 11). This might be due to the directory traversal which seems to be non-deterministic. We can suggest that the directory therefore has been moved. However in future work we aim to detect moves in the first place and therefore also use the appropriate visualization through splines explained in chapter 3. To compare the two subtrees we used two Treetank-GUI instances on the same database/resource. This is possible due to Treetank permitting multiple read-transactions.

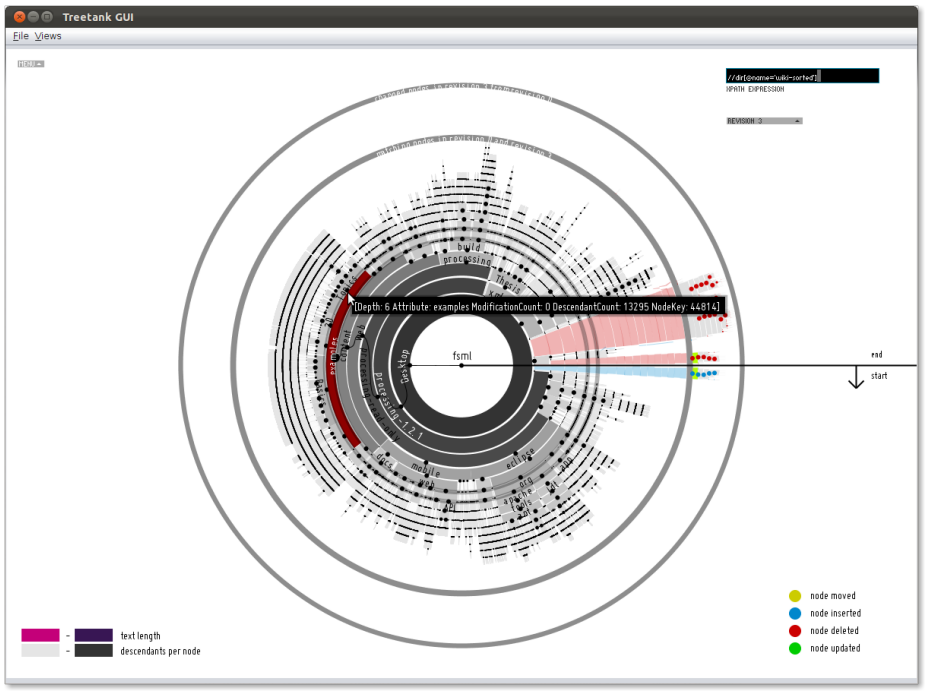

Fig. 10. FSML comparsion of "/home/johannes/Desktop"

To reduce the number of *SunburstItem*s and the runtime of the creation of the items the tree can be pruned by itemsize, just like in the case of the wikipedia-import. Items which are too thin and thus don't add value to the visualization are pruned. The default modification-weight is used, which isn't enough to adjust and enhance the sizes of changed subtrees. Furthermore this type of pruning doesn't impose any positive effect on the subtree-size of changes. Therefore we

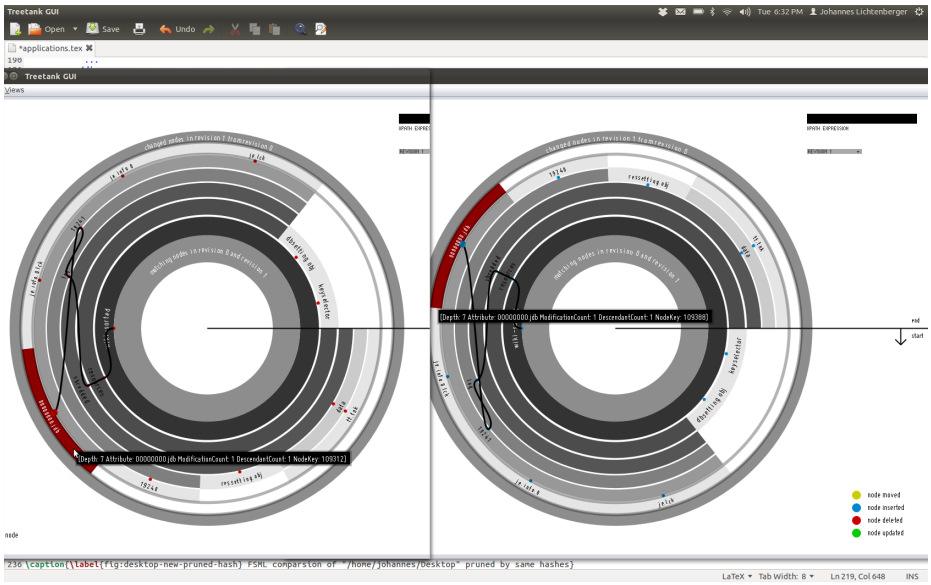

Fig. 11. FSML comparsion of wiki-sorted subtrees

use the pruning by hash-values (figure 12), which skips any unchanged subtrees. The number of created items is reduced drastically without causing any negativ effect. The context is still visible. Furthermore the items are big enough to be readable on the highest level without zooming into subtrees by selecting a new subtree-root node.

A sequence of revisions can be viewed with the differential (figure 13) and incremental view (figure 14). We used the hash-based pruning to keep the number of items to a minimum. Due to the fact that we use the hash-based pruning the differential view is a much better idea, as it's otherwise very hard to keep track of the structure. In so doing, subtrees are skipped for unchanged nodes with same hash-values. Thus it's hard to keep track of unchanged nodes across several revisions. However, it's the nature of incremental changes to easily track changes within each two compared revisions.

We are able to deduce from the two views, that an xmldiff-folder has been deleted in the first two revisions and the wikipedia-sorted folder has been moved from a directory in the folder named "Lichtenberger" to the "Desktop". In the next revision a few files or folders (with no content) have been deleted. The comparsion with the last revision which is displayed in the lower right corner reveals, that two files or folders have been added to the "figures" folder. In order to zoom in and to reveal further insights the standard comparsion views which have been displayed earlier in this chapter are very useful. Furthermore disabling the attribute-based labels reveals if the deleted or inserted nodes denote files or

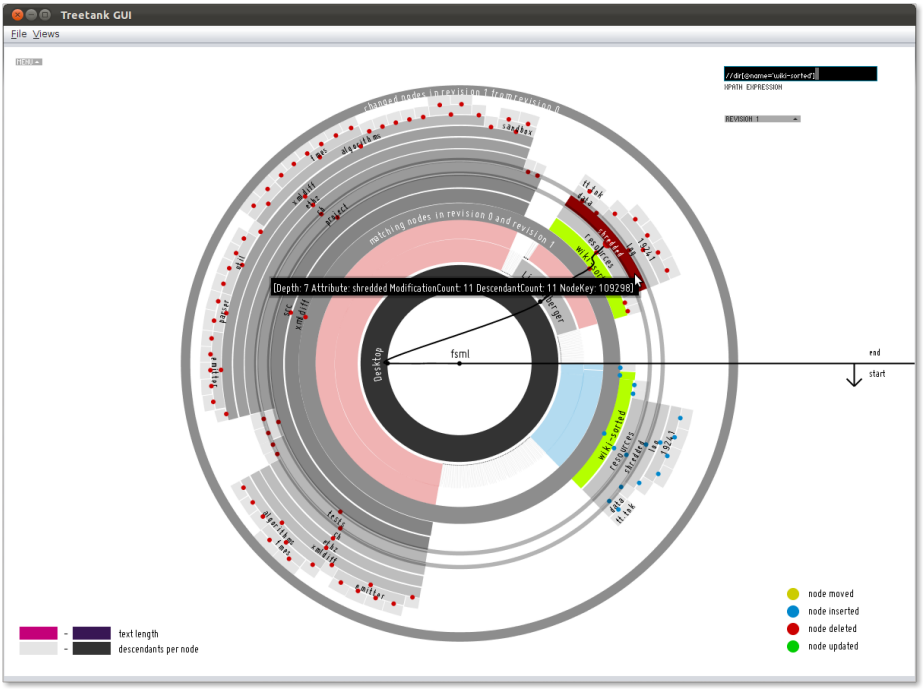

Fig. 12. FSML comparsion of "/home/johannes/Desktop" pruned by same hashes

directories (figure 15). Furthermore it is easily possible to adjust the item-color to map to the text color, usually used for text-nodes.

1.5 Summary

This chapter introduced a few use cases for revealing differences in hierarchically structured tree-data. We pointed out that depending on the used views different characteristics are revealed.

It is for instance ideal to use the SmallMultiples Differential-View with hash-based pruning to track which subtrees are unchanged and to reveal differences to the opened base-revision whereas the SmallMultiples Incremental-View is better to reveal changes in the data between each two revisions whereas the overall structure is neglegable. In case the tree is rather small, it's possible to use the Incremental-View without pruning or prune by itemsize in which case the overall structure is also much better reflected as the angles of the items just merely change. The third SmallMultiples-View is a kind of hybrid which reveals differences between several revisions as well as helps to keep the whole tree in view. One drawback of the current implementation is, that intermediate insertions and deletions of nodes are not visible in this view.

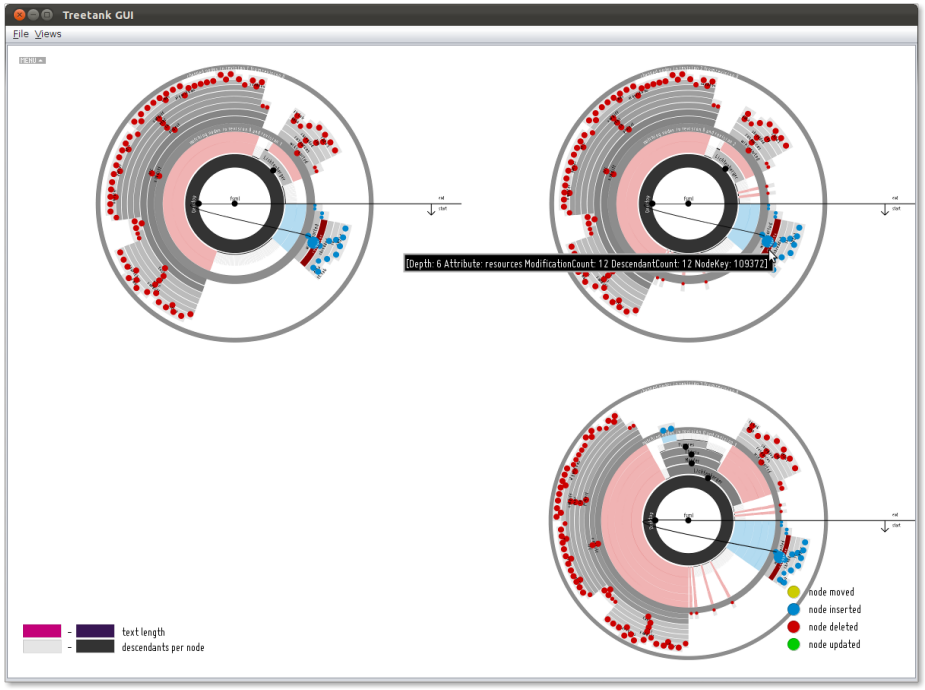

Fig. 13. FSML comparsion of "/home/johannes/Desktop" (SmallMultiples differential view)

The standard SunburstView reveals all kinds of differences and furthermore allows two zooming operations, the normal zoom which transforms the viewing-coordinates and can be used if the number of items is small and therefore animations are possible whereas a left-mouseclick on an item transforms the item into a new root-node. As already explaned in chapter 3 either a real transformation based on node-copies and adjusting of angles is done or in some circumstances the algorithm to create the items in the first place must be used.

We have merely scratched the surface of what's possible with the current approach and future work. The SmallMultiple-Views will greatly benefit from a temporal XPath-extension. Some proposed axis as `next::`, `previous::`, `future::`, `past::`, `revision::` already have been implemented but have to be incorporated in either the Saxon-Parser or our XPath 2.0-parser. Furthermore adding the possibility to execute temporal XPath-queries and the highlighting of resulting nodes in the SmallMultiple-Views will allow to track specific kinds of nodes over several revisions. The simple query
`//dir[@name='wiki-sorted']/future::node()` will reveal nodes with the attribute `name='wiki-sorted'` in all future revisions. The context will be based on the currently opened base-revision.

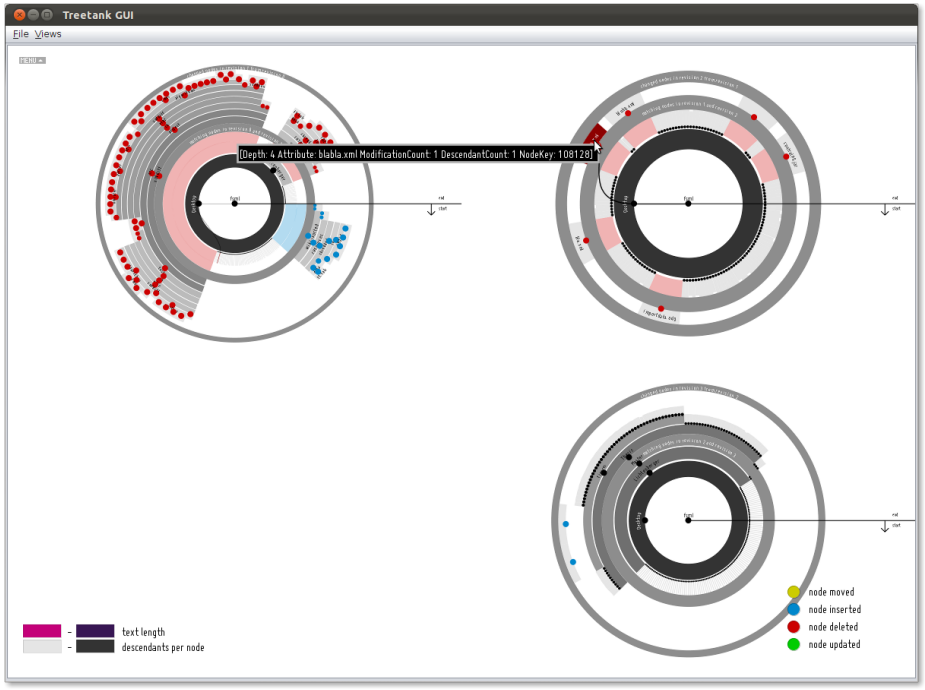

Fig. 14. FSML comparsion of "/home/johannes/Desktop" (SmallMultiples incremental view)

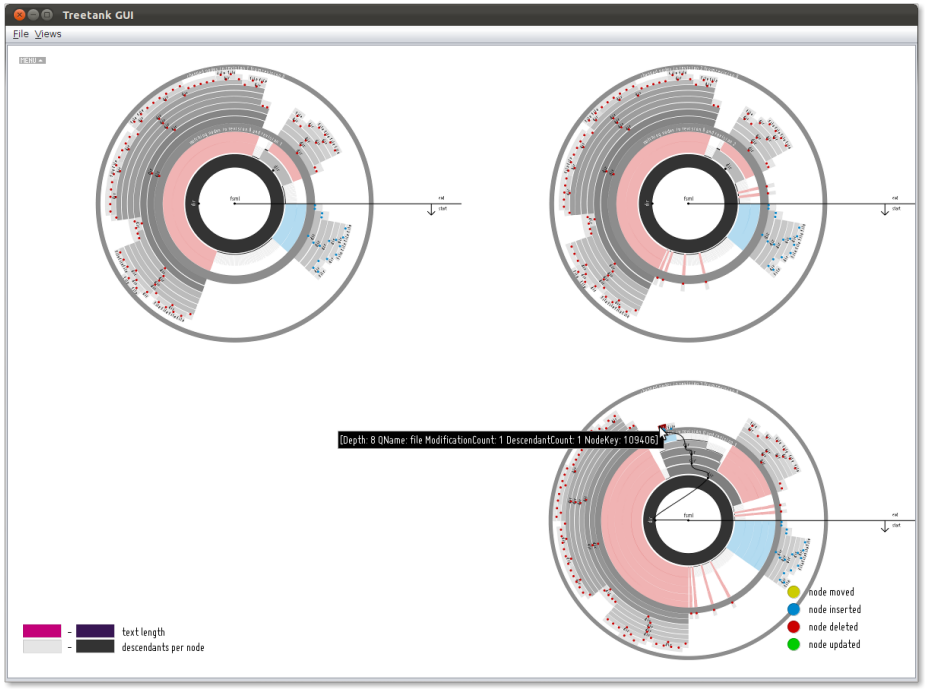

Fig. 15. FSML comparsion of "/home/johannes/Desktop" (SmallMultiples differential view without using attributes to plot the labels)

A Acknowledgements

References