Change Management on Semi-structured Documents[*]

Normen Müller
Jacobs University Bremen
Department of Computer Science
n.mueller@jacobs-university.de

ABSTRACT

Our globalized information society produces, maintains, and publishes about 5 Petabyte (i.e. ca. 3trillion pages) of documents a year for documenting events, plans, and results in society, jurisdiction, economy, politics, technology, and science. Some of these documents — e.g. newswire texts — are just written for the moment, while others — like technical documentation, progress reports, or company mission statements — persist in multiple versions being continually rewritten and adapted to changing situations.

Even, if we assume that only 3% are mission-critical documents that will be maintained over time, we are still faced with a huge management problem, which is aggravated by the fact that documents are inter-related and that changes in one — e.g. a company's mission statement — will make changes in others necessary — e.g. planning documents or the company web site. Such corpora can no longer be authored and maintained by individuals.

This paper applies management of change to semi-structured documents (e.g. XML) to improve evolutionary authoring processes.

Categories and Subject Descriptors

I.7.1 [**Document and Text Editing**]: Document management

General Terms

Management

Keywords

Versioning, Document Model, Change impact analysis, Management of Change

[*]Work in progress to be submitted as Ph.D. thesis at Jacobs University Bremen.

1. INTRODUCTION

The motivation for this work is to improve the maintainability of authoring processes, to optimize the release planning activity and thus reduce the maintenance effort. Reduction in effort can be achieved by minimizing the time between a proposed change, its implementation, and its delivery, while at the same time maintaining quality. It allows the maintainers to assess the consequences of a particular change to the document and can be used as a measure for the effort of a change. The more a change causes other changes to be made by rippling, the higher the cost. Carrying out this analysis before a change is accomplished allows us to assess the cost of the change and allows management to make a trade-off between alternative changes.

However, first we need to understand the *process of change*. Madhaji [15] defines the steps as follows[1]: First identify the need to make a change to an item in the environment (*Identification*), then acquire adequate change related knowledge about the item (*Change classification*), assess the impact of a change on other items in the environment (*Dependency classification*), select or construct a method for the process of change (*Rules*), make changes to all the items and make their inter-dependencies resolved satisfactorily (*Adjustment*), record the details of the changes for future reference, and release the changed item back to the environment (*Version Control*).

One key problem in accommodating changes in an environment is to know all the factors that impact a given change, the consequences of this change and how to adjust the affected items. Seemingly small changes can ripple throughout an entire corpus to cause major unintended impacts[2]. To avoid inefficiencies, conflicts, and delays authors need mechanisms to estimate the impact of a change (*change impact analysis*) and to adjust the affected items (*change management*).

Figure 1 depicts our perception of management of change (MoC) constituents. In contrast to [17], we subsume change impact analysis (CiA) by change management. We consider CiA as the technique to preview effects of changes, whereas the entire machinery of change management serves as the instrument to automatically adapt affected items and to facilitate inconsistency-correction, respectively. The icing on

[1]We map his definition to our illustration in Figure 1
[2]An impact can be thought of as the consequences of a change

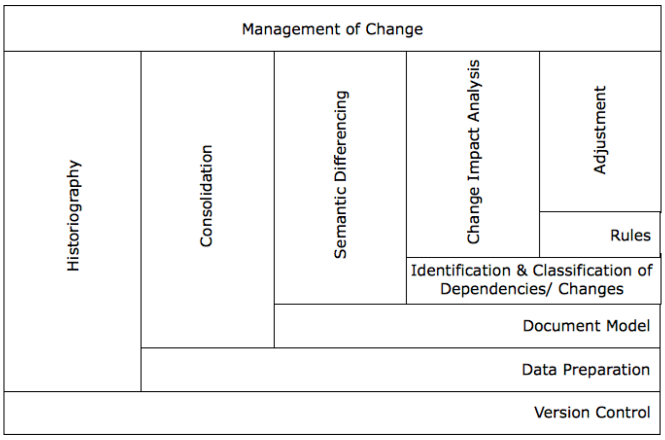

Figure 1: MoC cake.

our MoC cake is the explicit requirement of dependency and change classification. We emphasize that maintenance processes are only feasible if precise and unambiguous information on the potential ripple effects of a change is available. A ripple effect is the "effect caused by making a small change to a system which impacts many other parts of a system" [1].

The research described here addresses the problem of change management on semi-structured documents by applying algorithmic dependency and change type analysis techniques on semi-structured documents to discover structural and semantic relationships within and between documents as well as to propagate changes along these relations.

The subsequent sections give an overview of the individual MoC constituents.

2. HISTORIOGRAPHY

Version control systems (VCS) serve as a basis for management of change. A VCS can be characterized as a system which tracks the history of file and directory changes over time. All version controlled files and directories reside as a tree structure in a system distinct repository and both can have version controlled meta-data. Changes to such a tree are transactional resulting in a new snapshot (*aka.* revision) of the whole tree including all recent changes made in that commit operation as well as all previous unchanged data.

Common version control systems try to modify a version-controlled file system tree as safely as possible. Before changing the current tree, the to be committed modifications are written to a log file (*aka.* journal). Architecturally, this is similar to a journaled file system. If an operation is interrupted (e.g, if the process is killed or if the machine crashes), the log files remain on disk. By re-executing the log files, the system can complete the previously started operation, and the file system tree can get itself back into a consistent state.

Consequently version control systems ensure transactional recording of details of the changes for future reference and release the changed item back to the corpora.

3. CONSOLIDATION

For a corpus with many interrelated documents modifying and then re-validating one document is complex: analysis and consistency checking are required for each dependent document and the relations among them. The problem is further compounded because the maintainers are sometimes not the authors and may lack contextual understanding. As a corpus ages and evolves, the task of maintenance becomes more complex and more expensive.

To ease maintenance of complex corpora we first have to identify the coarse-grained constituents, i.e. the version controlled file system trees interrelated to each other. In [21] we elaborated an abstract theory of collaborative content management and version control for collections of semi-structured documents. We identify and consolidate relations between them in a *registry* and, in addition, use our mathematical data structure (the *fs*-tree model) to generalize version controlled file system trees and semi-structured documents alike. This allows us to specify and implement fine-granular management of change algorithms that seamlessly work across the file and file system border. In order to keep the registry consistent, we use the journaling functionality of the underlying version control system, i.e., only committed modifications are recorded.

This data preparation lays the foundation for acquiring adequate change related knowledge about the item subjected to change.

4. SEMANTIC DIFFERENCING

Before we can analyze impacts of changes, we have to identify them — just because everything is different, does not mean anything has changed. Most previous work in change detection has focused on computing differences between flat files. The GNU *diff* utility is probably the most famous one. This algorithm uses the LCS algorithm [23] to compare two plain text files. Version control systems, like CVS [6] and SUBVERSION [27], use *diff* to detect differences between two versions. Chawathe et al. [2] pointed out that these techniques cannot be generalized to handle structured data because they do not understand the hierarchical structure information contained in such data sets. Typical hierarchically structured data, e.g. XML, place tags on each data segment to indicate context. Standard plain-text change-detection tools have problems matching data segments between two versions of data.

In order to design an efficient algorithm to detect changes on XML documents, we first need to understand the hierarchical *structure* in XML. Fortunately, the increasing use of XML over the last years has motivated the development of many differencing tools capable of handling tree-structure documents. However, none of these tools considers *semantics* of XML documents and all of them work with either completely ordered trees [26, 28, 8, 32, 4, 3] or completely unordered trees [33, 29]. Clearly, it is not sufficient to compare XML elements as strings: syntactically different elements may appear to be semantically equivalent.

In [22, 20] we extend the previously mentioned differencing tools by introducing a semantic notion of similarity between individual elements of XML documents. These *document* *models* give us a stronger notion of equality leading to more compact, less intrusive edit script. For instance, if we know that ordering of elements carries no meaning in a document format (think of BibTeX entries), two documents are considered equal, even if they differ in every single line (w.r.t. to the element order). Consequently, with this notion of equality, the computed edit script would be empty. This motivates the need to identify syntactically different but semantically equal document fragments and thus to generate less intrusive edit scripts. Architecturally, a document model is a contravariant type constructor splitting up the vocabulary of an XML document into similarity groups with respect to specific equivalence relations. The problem of assigning elements to the appropriate groups is left to the user. However, this choice is generally made by analyzing the respective schema. Consequently, it allows to reuse the same document model for all documents referring to that schema.

This grouping condenses the required change related knowledge and in turn facilitates to accomplish a more precise impact analysis.

5. CHANGE IMPACT ANALYSIS

In order to calculate the effect of changes on semi-structured documents, we classify dependencies and changes according to different types. Types herein are sets of equivalence relations. A dependency type denotes the equivalence relations sensitive to the dependency, whereas a change type represents the equivalence relations preserved by the modification. A change is only propagated along a dependency relation if its type correlates to the type of the dependency, i.e. if the type intersection is not empty the change has to be propagated.

Regarding dependencies we distinguish between structural and semantic relations. Dependencies between compound fragments are structured and can be taken as a composition of dependencies between their composites. However, documents also have complex associations at the semantic level, which are not hierarchical. This research proposes to use Kleene Algebra with Test (KAT) for change management at the semantic level, a new branch of algebra that lends itself for practical modeling purposes.

6. ADJUSTMENT

Similar to approaches in requirement tracing, document fragments are informed about changes of their environment by activating their triggers. Depending on the nature of the item, triggers activate the adjustment of the corresponding fragment or simply signal the user that he has to update them manually. Triggers are especially useful if the content of a document fragment can be automatically recomputed by inspecting its environment.

Architecturally, we propose to first perform a change impact analysis resulting in a pair lists. The first component represents changes automatically adaptable. For example, think of all syntactical changes, like, renaming. The second component denotes conflicts caused by semantic modifications on the supporter generating (potential) inconsistency in the dependant along the dependency relation. Consequently, an implicit modification does not harm the release work-flow, but just increased the set of affected items. A (semantic) conflict, however, prevents committing the semantically affected items. Such items have to be adjusted by the author.

7. RELATED WORK

Modeling data, control, and component dependency relationships are useful ways to determine change impacts within the set of documents. The basic impact analysis techniques to support these kinds of dependencies are data flow analysis [11, 31, 7], data dependency analysis [18], control flow analysis [13, 16], program slicing [30, 9, 14, 12], test coverage analysis [5, 24, 25], cross referencing, and browsing [1], and logic-based defects detection and reverse engineering algorithms [10].

8. CONCLUSION

We have outlined our approach of change management on semi-structured documents. In particular, we have presented our compilation of MoC constituents and emphasized that a classification of changes is worthwhile compared to time-consuming manual reviews.

To evaluate our concepts and algorithms we have implemented a prototype SUBVERSION client based on the ideas put forward in this paper, in particular, the version control, consolidation, and semantic differencing slices. The *locutor* system [19] has been heavily used in day-to-day work and bears out the expected efficiency and consistency gains. For example, considering software systems as corpora of source code documents[3], we accomplished efficient regression testing by helping testers to decide what objects need to be retested. Consequently our approach is applicable to both domains, authoring process and software engineering.

Current research focuses on the specification and implementation of the remaining slices: change impact analysis and adjustment. By accomplishing these task, we will identify further requirements regarding the correlation of dependency and change types and improve the *locutor* system.

9. REFERENCES

[1] S. A. Bohner. *A Graph Traceability Approach for Software Change Impact Analysis*. PhD thesis, George Mason University, Fairfax VA, 1995.

[2] S. S. Chawathe, A. Rajaraman, H. Garcia-molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

[3] G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *ICDE*, pages 41–52. IEEE Computer Society, 2002.

[4] F. Curbera and D. Epstein. Fast difference and update of XML documents. In *XTech*, San Jose, 1999.

[5] R. A. Demillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. In *IEEE Transactions on Software Engineering*, volume 17, pages 900–910, September 1991.

[6] Free Software Foundation. Concurrent Versions System (CVS), seen May 2009. `http://www.gnu.org/manual/cvs-1.9`.

[7] M. J. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. In *Symposium on*

[3]Source codes are semi-structured documents restricted to a strictly prescribed structure.

Foundations of Software Engineering, pages 154–163, New Orleans, LA, December 1994. ACM SIGSOFT.

[8] C. M. Hoffmann and M. J. O'Donnell. Pattern Matching in Trees. *J. ACM*, 29(1):68–95, 1982.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *ACS Transactions on Programming Languages and Systems*, volume 12, pages 26–60, January 1990.

[10] Y.-F. Hwang. *Detecting Faults In Chained-Inference Rules In Information Distribution Systems.* PhD thesis, George Mason University, Fairfax VA, 1997.

[11] J. Keables, K. Roberson, and A. von Mayrhauser. Data Flow Analysis and its Application to Software Maintenance. In *Proceedings of the Conference on Software Maintenance*, pages 335–347, Los Alamitos, CA., October 1988. IEEE CS Press.

[12] B. Korel and J. Laski. Dynamic Slicing of Computer Programs. *The Journal of Systems and Software*, 13(3):187–195, November 1990. Elsevier North Holland Inc.

[13] J. P. Loyall and S. A. Mathisen. Using Dependence Analysis to Support the Software Maintenance Process. In *Conference on Software Maintenance*, pages 282–291, Los Alamito, CA, September 1993. IEEE CS Press.

[14] J. R. Lyle, D. R. Wallance, J. R. Graham, K. B. Gallagher, J. P. Poole, and D. W. Binkley. *Unravel: A CASE Tool to Assist Evaluation of High Integrity Software Volume 1: Requirements and Design.* National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD 20899, 1990.

[15] N. H. Madhavji. Environment Evolution: The Prism Model of Changes. *IEEE Transaction on Software Engineering*, 18(5):380–392, May 1992.

[16] I. McCabe & Associates. *Battlemap Analysis Tool Reference Manual.* McCabe & Associates, Inc., Twin Knolls Professional Park, 5501 Twin Knolls Road, Columbia, December 1992.

[17] R. Moreton. A Process Model for Software Maintenance. *Journal Information Technology*, 5:100–104, 1990.

[18] L. E. Moser. Data Dependency Graphs for Ada Programs. In *IEEE Transactions on Software Engineering*, volume 16, pages 498–509, May 1990.

[19] N. Müller. locutor - An Ontology-driven Management of Change System, seen May 2009. `http://locutor.kwarc.info/`.

[20] N. Müller. *Change Management on Semi-structured Documents.* PhD thesis, Jacobs University Bremen, 2010. Work in progess.

[21] N. Müller and M. Kohlhase. Fine-Granular Version Control & Redundancy Resolution. In J. Baumeister and M. Atzmüller, editors, *LWA Conference Proceedings (FGWM)*, pages 1–8. Universität Würzburg, 2008.

[22] N. Müller and M. Wagner. Towards Improving Interactive Mathematical Authoring by Ontology-driven Management of Change. In A. Hinneburg, editor, *LWA*, pages 289–295. Martin-Luther-University Halle-Wittenberg, 2007.

[23] E. W. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, 1:251–266, 1986.

[24] A. J. Offutt. An Integrated Automatic Test Data Generation System. *Journal of Systems Integration*, pages 391–409, November 1991.

[25] A. J. Offutt and A. Irvine. Testing Object-Oriented Software Using the Category-Partition Method. In *Seventeenth International Conference on Technology of Object-Oriented Languages and Systems, (TOOLS USA '95)*, pages 293–304, Santa Barbara, CA, August 1995.

[26] S. M. Selkow. The Tree-to-Tree Editing Problem. *Inf. Process. Lett.*, 6(6):184–186, 1977.

[27] Subversion, seen May 2009. `http://subversion.tigris.org/`.

[28] K.-C. Tai. The Tree-to-Tree Correction Problem. *J. ACM*, 26(3):422–433, 1979.

[29] Y. Wang, D. J. DeWitt, and J. yi Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 519–530. IEEE Computer Society, 2003.

[30] M. Weiser. Program Slicing. In *IEEE Transactions on Software Engineering*, volume 10, pages 352–357, July 1984.

[31] L. J. White. A Firewall Concept for both Control-Flow and Data-Flow in Regression Integration Testing. *IEEE Transactions on Software Engineering*, pages 171–262, 1992.

[32] K. Zhang and D. Shasha. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

[33] K. Zhang, R. Statman, and D. Shasha. On the Editing Distance Between Unordered Labeled Trees. *Inf. Process. Lett.*, 42(3):133–139, 1992.