

Research Proposal for a M.Sc. Thesis

Smart Management of Change on OMDoc Documents

Sönke Holsten

Jacobs University Bremen
D-28759, Bremen, Germany
s.holsten@jacobs-university.de

Advisor: Prof. Dr. Michael Kohlhase
Co-Advisor: Normen Müller

February 5, 2010

Abstract

Change management subsumes and relates to several cutting-edge research topics such as change impact analysis, traceability and round-trip engineering. This research project touches research questions from all of these areas. It aims for incorporating the smart change management features for semi-structured documents such as change detection, change classification and change impact analysis provided by the *locutor* library into the widely-established integrated development environment Eclipse and the version control system Subclipse. The created plug-in will be applied to - and thus evaluated by - use cases in mathematics using OMDoc as the underlying format.

Contents

1	Introduction	3
1.1	Related Fields	3
1.2	Change Management	3
1.3	Application in Software Development	6
2	Research goals	7
2.1	<i>locutor</i> Eclipse plug-in	8
2.2	OMDOC document model	10
3	Progress report and work plan	10
4	Future work	11

1 Introduction

The need for sophisticated change management tools in areas such as software engineering and business process modeling has been identified for a long time, but is still subject to many recent research activities in particular in the fields of **impact analysis** and **traceability**.

To put this research proposal into context I will start off by providing an overview of related research topics in section 1.1 and then introduce the key interest of this project while highlighting the overlap with existing research in section 1.2. A general use case is presented in section 1.3. The research goals with a more detailed description of all deliverables is given in section 2. Finally section 3 presents the current state of the research and section 4 gives an overview of further research questions that are of relevance to this research topic, but are outside the current scope.

1.1 Related Fields

Traceability is the common term for mechanisms to record and navigate relationships between artifacts produced by development processes.[14] It is often used interchangeably with the term **requirements traceability**, which refers to the study of requirements throughout the whole software development life-cycle. The latter links requirements in the requirements analysis to design and code documents that implement the requirements. It also links related requirements in the requirements analysis. An overview of recent traceability techniques in the field of requirements traceability can be found in [4, 9, 8]. One major challenge in this area is overcoming the heterogeneity of the document formats. Different formats are used at each stage of the development process, but the relationships between the different documents are nonetheless manifold. A more general approach to traceability between any heterogeneous artifacts not limited to requirements engineering can be found in [1].

An important distinction made in the field of traceability is the one between **tracable** and **traced** documents. A traced document is a document for which relationships have been identified and are stored visibly, in other words, are made explicit. If the relationships in a document can be deduced from the structure, the document is called tracable [25]. Usually a richer structure makes a document more traceable.

Traceability of a document is often considered a prerequisite for sophisticated **impact analysis**. The term impact analysis (IA) is used in many different contexts and it is not always clear what it comprises. In this proposal IA is taken to generally refer to the identification of potential consequences of a change. One important concept in this area is the **ripple effect**, which occurs when a comparatively small change to a document affects many other parts of the document and/or many parts of external documents. A more detailed explanation of the terminology and differences between IA approaches can be found in [2].

1.2 Change Management

To understand change management it is necessary to first understand and adequately model the nature of change and, more specifically, the process of change.

```

<book isbn="123456789">
  <title>
    On Semi-Structured Documents
  </title>
  <author>
    John Doe
  </author>
</book>

```

Figure 1: A sample XML document

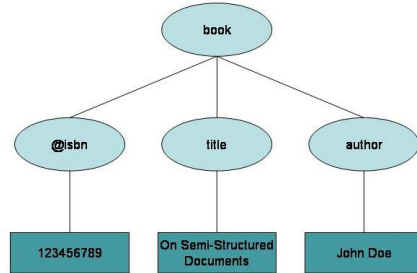


Figure 2: Tree representation of XML document

For this many different concepts have been developed in the area of IA. On a more abstract level, this problem has also been investigated from a business process perspective [10, 21].

In context of this research change is always considered with respect to a specific document. In its most basic interpretation this means that change is the *physical modification, insertion or deletion of a string in a document*. Depending on the type of the document and the exact location of the change its semantics vary greatly.

For instance in the area of software development, modifying the indentation of source code has different implications depending on the programming language used. While in a C program the indentation might be changed to improve readability of the source code, in a Python program a change in indentation might affect the operational semantics of the program itself. Thus in order to reason about the semantics of a change, it needs to be analyzed with respect to the type of document.

Change can be classified according to a plethora of characteristics. In the context of software development, Buckley et al.[5] have identified four dimensions of software change: *Temporal properties* (when is it performed?), *System properties* (what is changed?), *Object of Change* (where is the change located?) and *Change Support* (how is the change performed?). Although their further characterization of these dimensions is quite extensive, Buckley et al. acknowledge that there are even more characterizing factors of change. For example the motivation for a change, that is the question of why a change is performed, is not considered in their taxonomy.

Apparently the question whether a change taxonomy is reasonable or not depends on its application domain and the interests of all stakeholders involved. To accomodate this observation I will make use of a formal framework that allows for user-defined change classifications. This framework makes use of the fact that the documents under consideration are **semi-structured** and allows for change classification sensible to the exact location of the change taking the structural information about the type of document into account.

A **semi-structured document** is a document that can be represented in a hierarchical, tree-like structure. XML is one apparent standard for developing such documents as the XML standard already describes a tree representation for XML documents.

Fig. 1 shows an XML snippet describing a book with an ISBN number

modeled as an attribute and a title and an author modeled as child elements of the book element. Fig. 2 shows the corresponding tree representation. Intuitively we consider all types of documents that can be represented as a tree similar to the one depicted in Fig. 2 semi-structured. For a detailed discussion on a formal model for semi-structured documents see [17].

From a document point of view the process of change can be described as follows:

1. maintenance of relationships between and within structured documents,
2. detection of changes between current and base version of the semi-structured document,
3. semantic classification of the detected changes,
4. impact analysis of classified changes and
5. if possible propagation of changes according to previously performed analysis.

In the following I will detail this change process, which is also depicted in Fig. 3.

As mentioned above a major prerequisite for change impact analysis is traceability. In this process model this is reflected in the step of **relationship maintenance** (1). While this step is negligible for documents, which are fully traceable due to their structure, it is expected that in many real life scenarios the structure of the document is not sufficiently rich and manual tracing, that is the explicit determination of relationships between and within documents, needs to be performed.

The second step towards effective management of change is granular **change detection**. The occurrence of a change needs to be detected automatically and localized within a document with as high precision as possible. Typical version control systems like SVN [23] and CVS [6] localize changes by line to line comparison and indicate the location of strings modified, deleted or added by providing its exact starting and ending line and column numbers. Since in the scope of this research it is assumed that the changed document is semi-structured, the localization of the change is performed by indicating its position in the tree representation.

After a change has been detected it needs to be classified (3). At this step the second important characteristic of the change process considered in this proposal comes into play: the automated change detection mechanism is provided with knowledge about equivalencies, thus making it possible to filter out purely syntactic changes automatically. Running the change detection mechanism on a C program, it would for instance recognize that a change

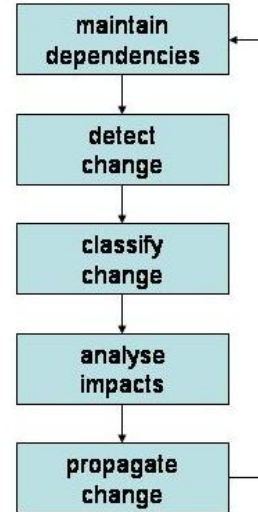


Figure 3: Process of change

in indentation does not affect operational semantics. As an optional step the user performing the change might provide further data about the change. This might be necessary to enable an enhanced impact analysis that can reason more accurately about the precise effect of a change.

Building on the granular change detection and sophisticated change classification, a smart **impact analysis** can be performed (4). The location of the change with respect to the tree representation provides contextual information like the labels of incoming and outgoing arcs of the node in question, which can be taken to represent dependencies. Knowledge about what types of changes influence dependent parts can then be used to calculate the impacts.

Finally during the step of **change propagation** the impacts are resolved manually and/or automatically (5). Change propagation using graphs as an underlying data model has for instance been investigated in [20]. The change might affect the existing dependencies which is why the loop in Fig 1.2 closes with the dependency maintenance.

1.3 Application in Software Development

The initial motivation for this project arises from the area of software development, where change has been identified as a key characteristic of the software development process and heterogeneity of document types is a major issue. It is estimated that often more than 50% of a system's requirements undergo some change before deployment [12]. This has given rise to software development methods that anticipate change during the development process.

Software development can be seen as a document-centric process as each step in the software development process involves the preparation, manipulation or verification of some - in most cases semi-structured - document [24]. Typical examples of documents produced during the development process are: requirements specification, design document, source code and documentation, all of which may use different underlying document formats.

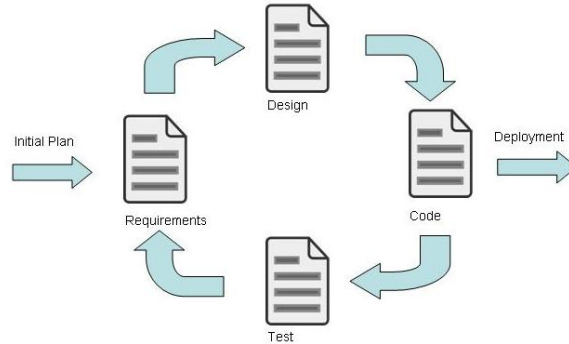


Figure 4: Iterative Development Process

Employing an iterative development method, the process of creation and modification of the different software artifacts can be modeled as depicted in Fig. 4. Starting from an initial requirements specification negotiated with the customer, during each iteration a subset of the requirements is chosen for implementation. The artifacts, design document, program code and test code, are

extended and modified in each iteration. As interim versions of the software are shown to the customer his expectations might change and the requirements specification will change as well.

Clearly, managing changes is a task that needs to be performed very frequently. As the artifacts involved grow in complexity it becomes a more and more complex task. Having well-defined processes and good tool support to accomodate this problem is essential to such a development method.

2 Research goals

The aim of this research project is to provide tool support for managing the change process whereby the focus of this work lies on steps 2, 3 and 4 of the change process as introduced in section 1.2.

To achieve this I build upon existing work, more specifically *locutor* [15, 18, 13]. The *locutor* library is being developed by Normen Müller at Jacobs University. It is a full reimplementaion of the Unix Subversion client focusing on smart management of change. It is able to perform change detection and change classification as outlined in section 1.2. However *locutor* does not come equipped with a graphical user interface.

As heterogeneity is a key issue, integrating *locutor* into an IDE that supports the editing of many different types of documents is desirable. Eclipse [7] has been chosen as an embedding IDE for *locutor* for several reasons. Eclipse is Open Source and has a very well developed plug-in architecture allowing it to be fully customized to ensure a seamless interweaving with *locutor*. Furthermore it can be extended with plug-ins that support the whole software development process from requirements engineering to source code generation, thus it is envisioned that integrating *locutor* in Eclipse will enable users to use change management features provided by *locutor* throughout the whole software development process as outlined in section 1.3. Nonetheless the use of Eclipse does not per se limit its applicability to software engineering.

Also there exists an implementation of the Subversion client for Eclipse, namely Subclipse [22], that is the standard version control features are already supported and need only be extended by the change management features specific to *locutor*. Integrating the *locutor* library in Eclipse also puts the interoperability of *locutor* to a test. The experience gained from implementing the plug-in can be used to improve the integratability of the *locutor* library. A detailed description of the functionality of the envisioned *locutor* plug-in will follow in section 2.1.

To evaluate the usefulness of the implemented mechanisms and the feasibility of the approach as a whole, the plug-in will be tested on mathematical documents represented in OMDOC. Although the real strength of *locutor*'s change management mechanisms exhibits in its ability to combine different document formats, I consider it unrealistic to perform an extensive case study involving several document formats, for instance in the area of software engineering, given the time constraints. Since as of yet there is no native support for OMDOC in *locutor*, an implementation of a document model[16] for OMDOC will also be part of this thesis. A more detailed description of this deliverable will follow in section 2.2.

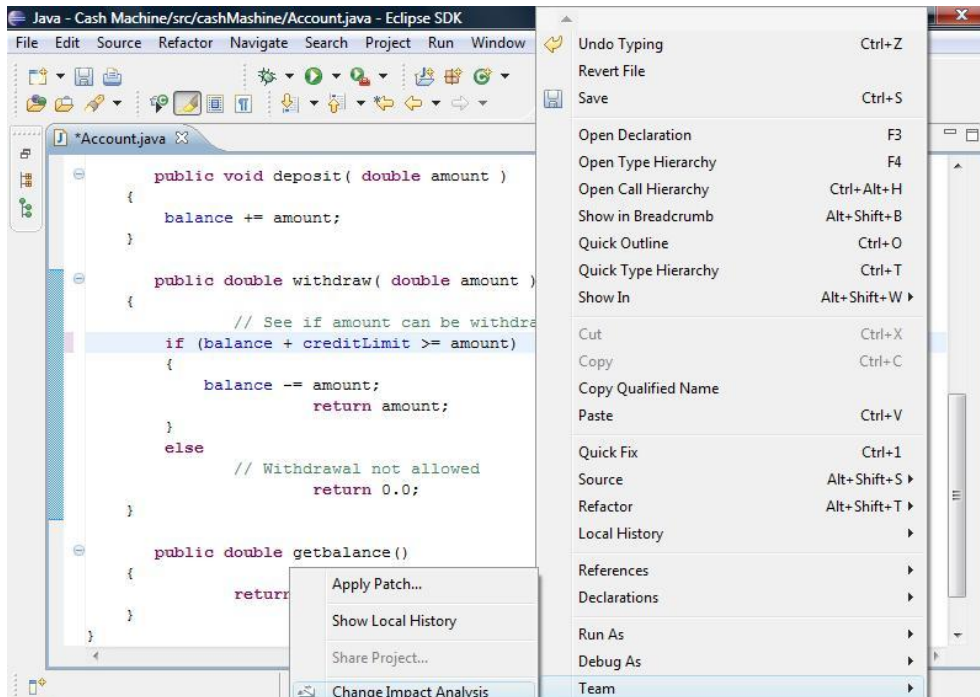


Figure 5: Before performing change impact analysis.

2.1 *locutor* Eclipse plug-in

There is a number of prerequisites that need to be fulfilled in order for the change impact analysis to function properly. Firstly a technical description of the underlying document format, which I refer to as a **document model**, needs to be provided. In case there are several types of documents in the project folder, one document model for each type needs to be provided. If one is missing, a default document model needs to be specified.

The exact representation of such a document model is subject to on-going research and dependent on the *locutor* library. Essential to the concept of a document model is the contained classification of changes and equivalencies for the document format in question.

Secondly a dependency graph[3] for the document to be analyzed needs to be provided. Again the exact representation of such a dependency graph is subject to on-going research. It is however envisioned that a textual representation of a directed, labeled graph will be suitable for this purpose. It might be possible to generate basic dependency graphs automatically by exploiting the structural information of the document, but the user needs to add further semantic relations manually.

Thirdly a base version to which the current version of the document should be compared needs to be provided. As the *locutor* plug-in should be used in conjunction with Subclipse, the base version in the local repository seems to be a reasonable candidate for the comparison. However if the user does not commit his changes after impact analysis and change propagation, the question of how to maintain a comparative version of the document requires further attention.

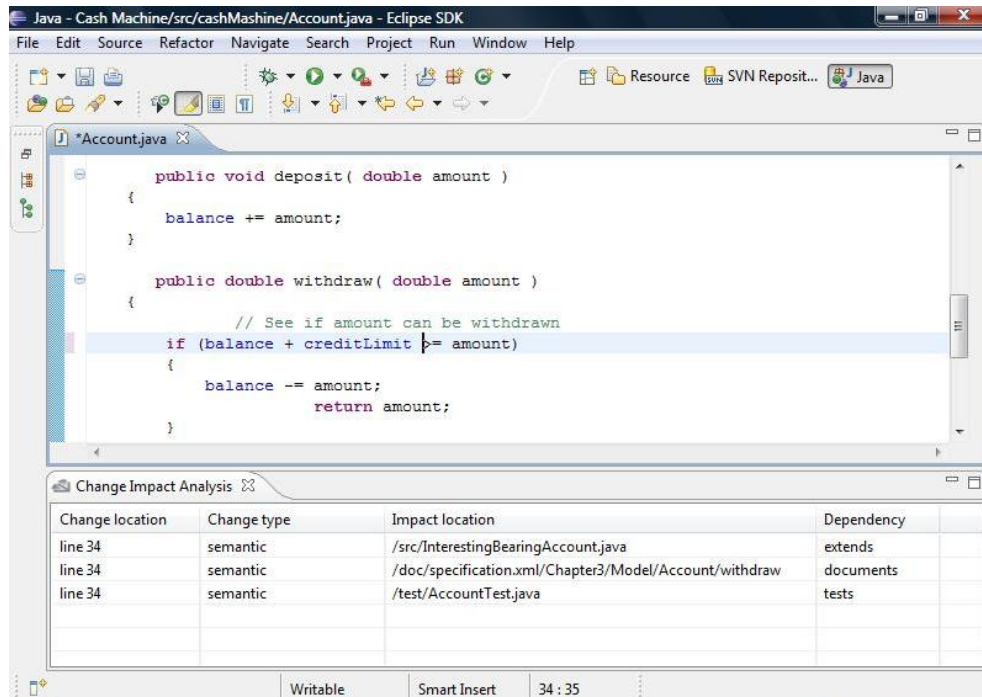


Figure 6: Result of change impact analysis.

Given these prerequisites the basic workflow from the user perspective with the core version of the *locutor* plug-in is as follows:

1. Perform a change to the document
2. Start change impact analysis from the context submenu for SVN ("Team")
3. Manually lookup change impacts displayed in view and resolve them
4. Commit consistent version to SVN repository

This workflow is illustrated in figures 5 and 6. Fig. 5 shows a Java class `account` representing a bank account, which has been modified by the programmer. More specifically, the programmer has modified the constraint in the method `withdraw` to not only take the `balance` of the account into consideration, but also a possible `creditLimit`. Having performed this change he would like to know about the impacts his change has on other documents in the current project and clicks on the action **Change Impact Analysis**.

The result of this action is shown in Fig. 6. A new view has opened in which all impacts of the change are shown. The first column of the view shows the result of the change detection, that is the change location. The second column displays the result of the change classification, which has identified the change as having an impact on the operational semantics. In the third and fourth column information about the impacts is shown. The third column tells the user exactly what document was affected and the fourth column gives context information about the dependency due to which the change has an effect on the mentioned artifact.

The change has an impact on the artifacts located at `src/InterestingBearingAccount.java`, `doc/specification.xml` and `test/AccountTest.java`. Note that the second impact does not stop at indicating a document, but further specifies a part contained inside the document. This is possible due to the similarity of the internal representation of semi-structured documents used and file systems as was discussed in more detail in [17]. The situation described would be typical for the software development process as outlined in section 1.3.

The *locutor* plug-in will be made available on a *locutor* update site. Being distributed as an Eclipse feature it can be installed on any OS that is supported by Eclipse.

2.2 OMDoc document model

OMDOC [11] is a strong semantic markup for mathematical theories. It allows to represent mathematics on three different levels, the *object level* (e.g. variables), the *statement level* (e.g. proofs) and the *theory level* (e.g. the theory of a monoid).

OMDOC has a rich structure and it will be an interesting challenge to investigate how it can be represented in a document model. Gaining hands-on experience on implementing a document model for *locutor* will allow me to evaluate the usability and usefulness of the *locutor* plug-in.

Since OMDoc is based on XML no problems are expected when defining the tree representation. However there does not exist any extensive change classification for OMDoc theories. Defining useful change types will require me to understand the major use cases for OMDoc.

3 Progress report and work plan

Up to now I have created a mock-up of the Eclipse plug-in for *locutor* from which I have taken the screenshots (Fig. 5 and Fig. 6). The results displayed in the Change Impact Analysis-view have been hardcoded into the plug-in. In fact the *locutor* library is as of yet neither used nor linked in the *locutor* plug-in. Currently it can be installed from <http://kwarc.info/projects/locutor-plugin/v1.x/>.

The first milestone for the implementation which at the same time will make up for the core of the *locutor* plug-in is rudimentary support for steps 2 to 4 of the change process specifically for OMDoc documents using a minimalistic document model. One major challenge that can be foreseen for this milestone is maintenance of the state of the base version to which the current version of the document should be compared and keeping track of resolved impacts. I am estimating that this initial version can be completed by March 2010.

After finishing the core version of the plug-in I will start developing the OMDoc document model and designing sample documents for testing purposes. At the same time I will gradually refine the plug-in eliminating bugs and making it more customizable, so it can be used for other document formats as well.

4 Future work

In section 2 I presented first steps to support smart change management in Eclipse whose implementation seems reasonable given the timetable for my research. As it has become apparent from this proposal, the plug-in would not support the whole change process. Thus, a natural continuation of this research would be to extend it to cover steps 1 and 5 of the change process.

Step 1 would comprise means to view and navigate dependencies, changes and impacts using the *locutor* plug-in as well as maintaining them. While the former can be solved by improving the implementation of the existing views, the latter would require the implementation of an editor from scratch.

Step 5 depends on the means provided by the *locutor* library. Given that the employed change calculus allows to detect and classify changes that can be propagated automatically, mechanisms to guide through this process could be implemented in the plug-in. These would be similar to the refactoring mechanisms for Java already supported in Eclipse.

From a conceptual viewpoint it is interesting to note that there already exists an OMDOC ontology [19]. As there is a semantic overlap between ontologies and document models, it can be interesting to study this overlap by means of comparing the OMDOC document model with the OMDOC ontology.

Acknowledgements

I would like to thank my supervisors Michael Kohlhase and Normen Müller for their support and incredible patience.

References

- [1] K. M. Anderson, S. A. Sherba, and W. V. Lepthien. Towards large-scale information integration. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 524–534, New York, NY, USA, 2002. ACM.
- [2] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 292–301, Washington, DC, USA, 1993. IEEE Computer Society.
- [3] F. Balmas. Displaying dependence graphs: a hierarchical approach. *J. Softw. Maint. Evol.*, 16(3):151–185, 2004.
- [4] M. Bashir and M. Qadir. Traceability techniques: A critical study. *Multi-topic Conference, 2006. INMIC '06. IEEE*, pages 265–268, Dec. 2006.
- [5] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, 2005.
- [6] CVS - Open Source Version Control. <http://www.nongnu.org/cvs/>, seen at January 2010.

- [7] Eclipse. <http://www.eclipse.org/>, seen at January 2010.
- [8] A. Espinoza, P. P. Alarcon, and J. Garbajosa. Analyzing and systematizing current traceability schemas. *Software Engineering Workshop, Annual IEEE/NASA Goddard*, 0:21–32, 2006.
- [9] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. pages 94–101, 1994.
- [10] E. Hu and Y. Liu. It project change management. *Computer Science and Computational Technology, International Symposium on*, 1:417–420, 2008.
- [11] M. Kohlhase. OMDoc – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, 2006.
- [12] G. Kotonya and I. Sommerville. *Requirements Engineering - Processes and Techniques*. John Wiley & Sons, 1998.
- [13] locutor - A library for management of change on semi-structured documents. <http://code.google.com/p/locutor/>, seen at January 2010.
- [14] P. Mason, K. Cosh, and P. Vihakapirom. On structuring formal, semi-formal and informal data to support traceability in systems engineering environments. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 642–651, New York, NY, USA, 2004. ACM.
- [15] N. Müller. An Ontology-Driven Management of Change. In *Wissens- und Erfahrungsmanagement LWA (Lernen, Wissensentdeckung und Adaptivität) conference proceedings*, pages 186–193, 2006.
- [16] N. Müller. Change Management on Semi-Structured Documents. 2009.
- [17] N. Müller and M. Kohlhase. Fine-granular version control & redundancy resolution. In J. Baumeister and M. Atzmüller, editors, *LWA*, volume 448 of *Technical Report*, pages 1–8. Department of Computer Science, University of Würzburg, Germany, 2008.
- [18] N. Müller and M. Wagner. Towards Improving Interactive Mathematical Authoring by Ontology-driven Management of Change. In A. Hinneburg, editor, *Wissens- und Erfahrungsmanagement LWA (Lernen, Wissensentdeckung und Adaptivität) conference proceedings*, pages 289–295, 2007.
- [19] OMDoc Document Ontology. <http://kwarc.info/projects/docOnto/omdoc.html>, seen at January 2010.
- [20] V. Rajlich. A model for change propagation based on graph rewriting. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 84–91, Washington, DC, USA, 1997. IEEE Computer Society.
- [21] S. Ramzan and N. Ikram. Making decision in requirement change management. In *ICICT 2005. First International Conference on Information and Communication Technologies, 2005.*, pages 309–312, 2005.

- [22] Subclipse. <http://subclipse.tigris.org/>, seen at January 2010.
- [23] Subversion. <http://subversion.tigris.org/>, seen at January 2010.
- [24] J. Welsh and J. Han. Software documents: Concepts and tools. *Software - Concepts and Tools*, 15(1):12–25, 1994.
- [25] R. J. Wieringa. Traceability and modularity in software design. In *IWSSD '98: Proceedings of the 9th international workshop on Software specification and design*, page 87, Washington, DC, USA, 1998. IEEE Computer Society.