Traceability Link Evolution with Version Control

Patrick Mukherjee[1], Karsten Saller[2], Aleksandra Kovacevic[1], Kalman Graffi[1], Andy Schürr[2], Ralf Steinmetz[1]

[1]Multimedia Communications Lab, Technische Universität Darmstadt
Rundeturmstr. 10, 64289 Darmstadt, Germany
{patmuk, sandra, graffi, ralf.steinmetz}@KOM.tu-darmstadt.de

[2]Real-Time Systems Lab, Technische Universität Darmstadt
Merckstr. 25, 64289 Darmstadt, Germany
{saller, schuerr}@ES.tu-darmstadt.de

Abstract: Traceability links enable the possibility to trace the evolution of a project from the earliest requirements engineering phase over the design and implementation to the test phases of a project. Having a link between the artifacts resulting from these phases, a requirement can be traced to, e.g., the test report, which tested the code implementing the requirement. However, as the artifacts evolve into new versions the traceability links between these artifacts evolve as well. Concurrent work on the artifacts might result in conflicts which reflects in conflicting versions of the traceability links. Often the artifacts created in different phases of a development process are stored in different repositories. In this paper we present a version control system capable of controlling the evolutionary changes of traceability links which connect artifacts residing in different repositories and resolve the conflicts caused by concurrent work on artifacts.

1 Introduction

Traditionally (in requirements management) traceability links have been used to connect requirements gathered in the first phase of a project to more detailed requirements resulting from the initial requirements [RSPE97, WN94]. DOORS [IBM] is the most frequently used tool to support this task.

However, traceability links can be created between any items, be it an artifact, folder or module, in any version. Traceability links can be used for various additional features in a project's development. For example, let us observe artifacts from different development phases, such as requirement, design, implementation and testing. By using traceability links, we can trace back a test case from the last phase of the development to related artifacts from earlier phases, such as source code or requirement specifications. This feature could be helpful in deciding which tests are to be executed to prove the implementation of a specific requirement, if budget or time constraints limit the executable tests. Traceability links between modules can be used to store dependency relations among projects, which

enables configuration management to choose specific versions. For example, a project could link to another project, indicating the minimal needed version it depends on.

When using traceability links in a dynamic environment like software development, maintaining traceability links faces many challenges. This paper will focus on three main challenges:

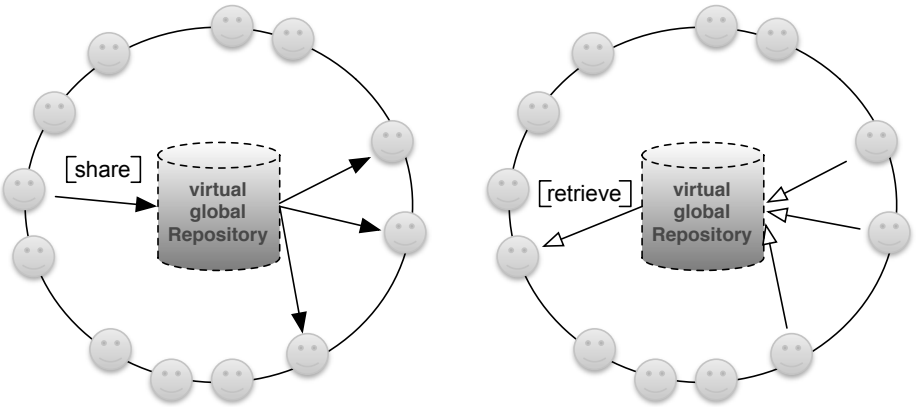
- Traceability links between artifacts should evolve along with the connected artifacts. Like with artifacts this evolution should be tracked, to be able to retrieve the correct state of a traceability link which connects specific versions of artifacts. We present how to overcome this limitation in Section 3.1.
- When storing traceability links in one central place, like most supporting tools do, this place becomes a bottleneck for users which want to access any traceability link, even if the accessed links connect different artifacts. We discuss our approach to overcome this challenge in Section 3.2.
- Often the artifacts created in different phases or by teams in different locations of a development process are stored in different repositories. Our solution to this problem is presented in Section 3.3.
- Concurrent work on connected artifacts might result in conflicts on connecting traceability links. We focus on our solution to this challenge in Section 3.4.

Our approach to maintaining traceability links is integrated in our fully decentralized version control system PlatinVC [Muk10], which is presented in Section 2. Section 3 details how the evolutionary changes of traceability links are handled. We discuss related approaches in Section 4 and close with a conclusion in Section 5.

2 Background: Version Control System PlatinVC

With evolving artifacts in software development a user expects traceability links to be versioned too, to enable all advantages of version control: retrieving an old state of the work to correct mistakes or managing possible conflicts occurred by concurrent work. Version control systems provide the ability to store all changes on artifacts and allow access any instance in the timeline of an artifact later. In this Section we present PlatinVC, a version control system that tracks the evolutionary changes of both, artifacts and the corresponding traceability links between them. In order to understand our approach, we summarize necessary background information about PlatinVC [Muk10].

PlatinVC represents, as a fully decentralized version control system, an opposite approach to the most commonly used client-server based version control systems that rely on a centralized machine to store and manage all artifacts of a project. The obvious drawbacks involved by utilizing an intermediary central server to exchanges and store created versions is having a single point of failure, a central ownership and vulnerability, scalability issues, and increased synchronization as well as communication delays. These drawbacks



(a) retrieve changes (b) share local changes

Figure 1: Sharing and retrieving versions using PlatinVC

are even more emphasized when projects are developed in a globally distributed manner [MKBS08]. Many popular global software projects switched to the emerging distributed version control systems, demonstrating the urgent need for a more suitable version control system. However, distributed version control systems, like, for example, Git [HT] or Mercurial [Mac] lack the global view of centralized systems, where developers do not have to care with whom they exchange their modified artifacts.

The basic features for version control are provided by Mercurial [Mac], the automatic exchange of new versions is carried out by the peer-to-peer middleware implementation FreePastry [FP]. PlatinVCs algorithms care about whom to send and from where to retrieve updates, like presented in Figure 1. Each participating computer, presented as yellow cycle on the circle and called peer, stores the whole repository in form of a mercurial repository. Using the key based routing mechanism [DZD$^+$03] a peer is responsible for any folder, whose hash value calculated using the folders name is similar to the peers identifier (see also [Muk10, MLTS08]. A responsible peer stores the latest versions of all artifacts residing in the folders it is responsible for. If any of these artifacts get updated the resulting snapshot is stored by the responsible peer. When peer retrieves the latest snapshot of all artifacts in a project, PlatinVC delivers them involving the minimum number of responsible peers needed. As the share and retrieve operations work transparent a user experiences the exchange of new versions as if he where communicating with a central repository, called virtual global repository in Figure 1. In this repository several modules can be stored, which consist of artifacts that are versioned using a snapshot. All files are in fact stored by each peer, most likely outdated. The latest snapshots of the artifacts in a folder are stored by its responsible peer, replicated to its direct neighboring peers. Only the peers, whose users retrieved the latest snapshot while no new snapshot was shared, have the latest version of all files.

Despite of its decentralized, peer-to-peer nature, PlatinVC still achieves the complete system view of centralized systems, while overcoming their drawbacks [GKM+07]. All workflows of existing version control systems, centralized or distributed, are supported. PlatinVC even supports a hybrid setup, where other version control systems can interoperate, using PlatinVC as a mediator. Moreover, it introduces an automatic isolation of concurrent work, which separates relevant and required updates from possibly unneeded ones. In this way, branches are only created when necessary. The evolutionary changes of links between files, which can be enhanced by any attributes, are also tracked. That extends snapshot-based version control to other purposes, e.g. traceability of software artifacts. PlatinVC is an alternative to current version control systems, as industry proven components for the most critical parts of the system were reused, and own developed parts where evaluated following [KKM+07].

3 Versioning and Maintaining Traceability Links

Enabling evolution of traceability links rise many questions, which we answer in this Section:

1. How to store them? (Section 3.1)
2. Where to store them? (Section 3.2)
3. What to do when artifacts created in different phases of a development process are stored in different repositories? (Section 3.3)
4. What to do when the linked artifacts change, how to version traceability links and how to resolve concurrent updates on linked artifacts? (Section 3.4)

Answering these questions, we present our approach to the evolution of traceability links in the fully decentralized version control system PlatinVC.

3.1 Structure and Form of Traceability Links

A traceability link can be created between any artifacts, folders or modules. A traceability link in PlatinVC can connect any number of items. For each connected item a companion file is created. It has a generic name formed by the "." prefix (to hide the file in unix-like operating systems), the linked item's name and the suffix "<-linkfile". This companion file is stored along with the corresponding artifact, or, in the case the item is a folder, in the folder and in the case it is a linked module, in the module's topmost folder. The companion file contains the names of *links*, which includes the actual information stored by a traceability link. These entries express that a link exists and serve as an index to find the link documents. No further information is stored here. Thus we avoid updating multiple files if a traceability link is changed.

```
<?xml version="1.0" encoding="UTF-8"?>
<linkType>DesignToCode</linkType>
<status>active</status>
<links>
(...)
        <link partner="A">
                <validFrom>3.4</validFrom>
                <direction>from</direction>
                <relation>derived</relation>
                (...)
                <otherAttribute>something</otherAttribute>
        </link>
        <link partner="B">
                <validFrom>1.8</validFrom>
                <direction>to</direction>
                <relation>derived</relation>
                (...)
                <otherAttribute>something</otherAttribute>
        </link>
(...)
</links>
```

Listing 1: Exemplary traceability link file *A<-link->B*

For each traceability link a link document is created, named with a arbitrary name.

If, for example, an artifact named *A* is connected with a traceability link to an artifact named *B*, there exist the following files: In the same folder where *A* is stored a companion file named *.A<-linkfile* is created. it contains the sole entry "A<-link->B". Next to the artifact *B* a file *.B<-linkfile* is created with the same sole entry. All information of the traceability link are stored in the file *A<-link->B*. Please note that the name can be any name, randomly created, created after a fixed scheme or assigned by a user.

The link document (*A<-link->B* in the previous example) is stored in a folder with the identical name (e.g. *A<-link->B* / *A<-link->B*), which is stored in a folder named ".linkdocuments". This extra folder (with the identical name) is necessary to distribute the link documents among different responsible peers, as they are responsible for files in different folders, like explained above. If the link document connects items, which resides in the same module, this folder resides in the topmost folder of that module. If items of different modules are linked the folder is stored in the topmost folder of a third module, which is named after the modules names in which the linked items are stored in. The name is constructed by the modules names with "<-links->" in between. In the latter case the entry in the item's companion file is prefixed with this name.

An exemplary traceability link (link document *A<-link->B* from the previous example) stored in PlatinVC is presented in Listing 1. A link document has two main parts. The

first xml-tags can specify information about the link, like its type. Here anything related to the traceability link itself, but not specific to any artifact, can be stored. The child xml-tags under <*links*> represent each the end of a link. Here the linked items are specified, by their name, version identifier and branch, if the latter is not the default branch. Any additional attributes belonging to the traceability link and specific to the linked item can be stored here as well.

A traceability link is valid for a specific version of the linked artifacts only. However, for the most projects it can be assumed that it is valid for later versions as well, until the link document is updated to a later version, which states new version identifiers for the connected artifacts. A link document can store any metadata about the connection of the linked items, such as integrity constraints, which can be evaluated to prove whether a connection is justified. This validation, however, is not handled by PlatinVC. The system PlatinVC does not control how links are interpreted, it merely tracks their concurrent evolution.

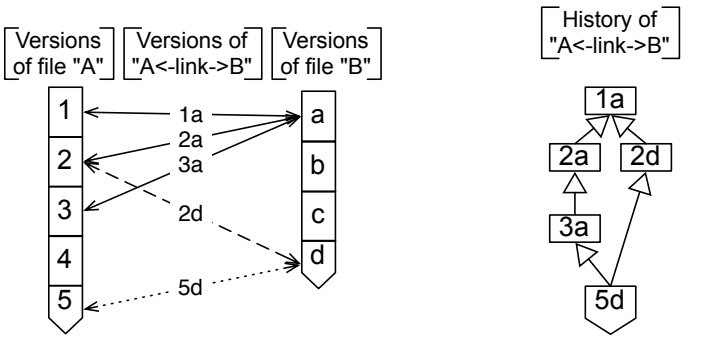
3.2 Distributed storage of traceability links

In opposite to solutions where the whole traceability link information for all files in a project are stored in a central place, the files containing this information (the link documents) are kept in a distributed storage implemented by PlatinVC. Following the key based routing mechanism [DZD+03], the link documents are distributed among different maintaining peers (see also [Muk10]). If two different developers access traceability links, which connect a disjunct set of artifacts, they communicate (most likely) with two different peers. In the rare case, where one peer is responsible for both traceability links and is overloaded by requests, the responsible peer's replicating peers take over.

3.3 Traceability links between different repositories

Often the artifacts created in different phases of a development process are stored in different repositories. This happens, when the staff working in one phase is different from the staff working in another phase, most often because they are employed by different companies. The challenge of maintaining traceability links along with the artifact evolution is even bigger when the repositories belong to different version control systems (e.g. subversion [Pil04], CVS [CVS], Mercurial [Mac]). Thus the requirements might be compiled by a traveling worker, in direct exchange with the customer, managed by a local version control system such as mercurial. The designers of company A create the software architecture, version controlled by CVS, while the developers of company B prefer Subversion for their source code. Lastly, the testers of a different company use their own CVS server to maintain the test reports.

As already explained in Section 2, PlatinVC is based on Mercurial which can use adaptors to many version control systems [O'S09]. Therefore, PlatinVC is capeable to handle



(a) Files and versions

(b) Created history of the link document

Figure 2: Concurrent updates on a linkdocument

several CVS and Subversion repositories at the same time.

3.4 Concurrent updates on linked artifacts

Changing the information stored in a traceability link updates its link document to a new version. Pointing to a different version of a linked artifact changes the link document as well. As a convention a link should only point to more recent versions of the linked artifacts in an update. However, when two different developers first update a linked artifact and update the link document subsequently, without having the latest version created by the other developer, the link's versions cross each other, like depicted in Figure 2(a).

Figure 2(a) illustrates an exemplary situation. The left column represents five versions of artifact "A", the right column four versions of a linked artifact "B", and the arrows in between represents the versions of the linking document (representing a binary traceability link). The version of both artifacts are created by different users. Alice works on "A" and Bob on "B". First Alice creates the artifact "A" and links it to version a of artifact "B". She updates "A" to version 2, updates the link to version 2a, and repeats the same process later to create version 3 of artifact "A" and version 3a of the link document. The versions 1a, 2a, and 3a of the link document are represented with the solid arrow lines. Meanwhile Bob updates artifact "B" to the versions b, c, and d. He updates the link to connect version d of artifact "B" and version 2 of artifact "A", which is the latest version he fetched.

Alice created the version 3 of artifact "A" and updated the link from version 2a to version 3a at the same time when Bob created the versions b, c and d of the artifact "B" and updated the link form version 2a to 2d. Hereby a conflict in the traceability link's version history arose. This conflicting history is stored by PlatinVC in an unnamed branch (as illustrated in Figure 2(b)). Version 3a and 2d are both based on version 2a. This branch can be merged

by pointing to a more recent version of the connected artifacts. Thereby link version 5d can be based on 3a and 2d to merge the diverging history lines. To avoid unnamed branches, updated links should always point to the latest version of the linked artifacts. When the network is seperated into partitions this is not always possible immediately. We integrated the version control aware instant messenger ASKME [MKS08] in order to notify users of a detected conflict.

4 Related Work

Much effort has been spent in the research of traceability links. Most of the research handles the problem of recovering traceability links to identify semantic connections between source-code and documentation [MM03] [ACC+02] [NG06]. None of these approaches can handle the evolution of software projects over time. The links which are identified during the recovery process are valid only for a certain version of the software project. During the development, the software artifacts are changing constantly which is why simple methods for link recovery are not efficiently applicable.

An approach that is leveraging traceability link recovery methods to provide consistent and correct traceability links in an evolving environment is described in [JNC+08] by Hsin-yi et. al. He describes TLEM, a automatic traceability link evolution management tool, that uses an incremental version of the Latent Semantic Indexing algorithm [yJNCD07] to re-evaluate traceability links after each change in an incremental fashion. The presented method re-uses all available information before a change is being triggered and only evaluates the impact of a change on the traceability. Therefore the costs of a full traceability link recovery is being avoided for every new version.

Another interesting approach to handle traceability links in evolving software projects is ArchTrace [MHW06]. Murta describes how a policy based system of rules can automatically manage traceability links between documentation and the corresponding source-code. Instead of reconstructing the traceability links after a certain amount of changes or time, ArchTrace updates these links after every commit operation from a user. ArchTrace is only capeable of maintaining existing traceability links, which means that they have to be created manually by the developers or by a traceability recovery method. The described approach uses a policy-based infrastructure to determine the according actions during an update of a software artifact. The policy rule-set has to be configured by the developer to be more accurate in the managing of traceability links. A conflict arises if more than one policy or no policy is triggered by an update. This has to be resolved manually by the developer. However, it may happen, depending on the configuration of the policies, that the wrong actions are triggered after an update (e.g. adding a new traceability link to another artifact) without any knowledge of the developer. The current implementation of ArchTrace only supports Subversion as version control system and has problems if the software architecture is branched.

A considerably different approach to handle evolving traceability links is described in [CHCC03] by Cleland-Huang et. al. This approach uses a scalable publish-subscribe

paradigm to handle the traceability links by event-notification. Source-code and documentation are linked by publish-subscribe relationships where updated artifacts take the role of publishers and the depending artifacts take the role of subscribers. When e.g. a requirement changes all depending artifacts are notified by specific notification messages which contain detailed information about each update event to support the update process of depending artifacts. Thus, this approach tries to support the overall project management with a distributed project-wide process guidance system. The owners of an artifact which has been notified by a change event are being supplied with the necessary information to update the corresponding traceability links. Hence, the traceability links have to be maintained manually.

Existing approaches to maintain traceability links lack the possibility to operate in distributed environments and are therefore limited in their usability in globally distributed software development environments, in contrast to PlatinVC which was specifically developed for such environments. Operating in distributed environments also enables PlatinVC to maintain traceability links over several repositories. For example, the requirement documentation may be stored in a CVS repository and the source-code in SVN repository. In this case, the traceability links may be stored in a third, independent repository. By utilizing mercurial, PlatinVC is compatible to all version control systems which are compatible to Mercurial (e.g. Subversion or CVS). Another advantage of PlatinVC compared to the presented approaches, is its simplicity: Manual interaction is only needed, if two traceability links are created in parallel with conflict to each other. Both links are kept separately in two separated versions until the developer resolves this conflict.

5 Conclusion

Traceability links bring many advantages to the software development process and their evolution is important and a challenging task. In this paper we presented an approach to version and maintain traceability links in an even more challenging environment where all communication is fully decentralized.

PlatinVC enables the tracking for the evolutionary changes of traceability links between artifacts in different modules, even when stored by different version control systems. It handles and maintains updates of traceability links and their version history. Concurrent updates are handled automatically and a version history for each traceability link is maintained. However, there is currently no support for creating links automatically or assisted. Links can currently only be created automatically via Piki [MLS08], our peer-to-peer based wiki engine.

In future our solution for tracking the changes of traceability links could be combined with a solution to create and change these links automatically. An approach as described in [Kön08] could be used to create or update links between artifacts (semi-)automatically, which could be stored in a decentralized fashion using PlatinVC. However, as a short term goal we strive to implement a distributed access control similar to the concept presented in [GMM+09] for the peer-to-peer social network *LifeSocial* [GGM+10].

References

[ACC+02] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering Traceability Links between Code and Documentation. *IEEE Trans. Softw. Eng.*, 28:970–983, October 2002.

[CHCC03] Jane Cleland-Huang, Carl K. Chang, and Mark Christensen. Event-Based Traceability for Managing Evolutionary Change. *IEEE Trans. Softw. Eng.*, 29:796–810, September 2003.

[CVS] CVS - Concurrent Versions System. `http://www.nongnu.org/cvs/`.

[DZD+03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, pages 33–44, February 2003.

[FP] FreePastry. `http://www.freepastry.org/FreePastry/`.

[GGM+10] Kalman Graffi, Christian Groß, Patrick Mukherjee, Aleksandra Kovacevic, and Ralf Steinmetz. LifeSocial.KOM: A P2P-based Platform for Secure Online Social Networks. In *Proceedings of the 10th IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 161–162, August 2010.

[GKM+07] Kalman Graffi, Aleksandra Kovacevic, Patrick Mukherjee, Michael Benz, Christof Leng, Dirk Bradler, Julian Schroeder-Bernhardi, and Nicolas Liebau. Peer-to-Peer Forschung - Überblick und Herausforderungen. *it - Information Technology (Methods and Applications of Informatics and Information Technology)*, 49(5):272–279, September 2007.

[GMM+09] Kalman Graffi, Patrick Mukherjee, Burkhard Menges, Daniel Hartung, Aleksandra Kovacevic, and Ralf Steinmetz. Practical Security in P2P-based Social Networks. In *Proceedings of the 34th Annual IEEE Conference on Local Computer Networks (LCN)*, pages 269–272, October 2009.

[HT] Junio Hamano and Linus Torvalds. Git - Fast Version Control System. `http://git-scm.com/`.

[IBM] IBM. Rational DOORS - Requirements Management. `http://www-01.ibm.com/software/awdtools/doors/`.

[JNC+08] Hsin-Yi Jiang, T. N. Nguyen, Ing-Xiang Chen, H. Jaygarl, and C. K. Chang. Incremental Latent Semantic Indexing for Automatic Traceability Link Evolution Management. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 59–68, Washington, DC, USA, 2008. IEEE Computer Society.

[KKM+07] Aleksandra Kovacevic, Sebastian Kaune, Patrick Mukherjee, Nicolas Liebau, and Ralf Steinmetz. Benchmarking Platform for Peer-to-Peer Systems. *it - Information Technology (Methods and Applications of Informatics and Information Technology)*, 49(5):312–319, September 2007.

[Kön08] Alexander Königs. *Model Integration and Transformation – A Triple Graph Grammar-based QVT Implementation*. PhD thesis, Real-Time Systems Lab (ES), Technische Universität Darmstadt, Germany, 2008. Supervisor: Andy Schürr.

[Mac] Matt Mackall. Mercurial, a Distributed SCM. `http://selenic.com/mercurial/`.

[MHW06] Leonardo G. P. Murta, Andr Hoek, and ClÂudia M. L. Werner. ArchTrace: PolicyBased Support for Managing Evolving Architecture-to-Implementation Traceability Links. In *In ASE*, pages 135–144. IEEE, 2006.

[MKBS08] Patrick Mukherjee, Aleksandra Kovacevic, Michael Benz, and Andy Schürr. Towards a Peer-to-Peer Based Global Software Development Environment. In *Proceedings of the Software Engineering Conference SE2008*, pages 204–216, February 2008.

[MKS08] Patrick Mukherjee, Aleksandra Kovacevic, and Andy Schürr. Analysis of the Benefits the Peer-to-Peer Paradigm brings to Distributed Agile Software Development. In *Proceedings of the Software Engineering Conference (SE)*, pages 72—77, Februar 2008.

[MLS08] Patrick Mukherjee, Christof Leng, and Andy Schürr. Piki - A Peer-to-Peer based Wiki Engine. In *Proceedings of the International Conference on Peer-to-Peer Computing (P2P)*, pages 185–186, September 2008.

[MLTS08] Patrick Mukherjee, Christof Leng, Wesley W. Terpstra, and Andy Schürr. Peer-to-Peer based Version Control. In *Proceedings of the 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 829–834, December 2008.

[MM03] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[Muk10] Patrick Mukherjee. *A Fully Decentralized, Peer-to-Peer based Version Control System*. PhD thesis, Real-Time Systems Lab (ES), Technische Universität Darmstadt, Germany, December 2010. Supervisor: Andy Schürr.

[NG06] Christian Neumuller and Paul Grunbacher. Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learne. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 145–156, Washington, DC, USA, 2006. IEEE Computer Society.

[O'S09] Bryan O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly & Associates Inc, 2009.

[Pil04] Michael Pilato. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[RSPE97] Balasubramaniam Ramesh, Curtis Stubbs, Timothy Powers, and Michael Edwards. Requirements traceability: Theory and practice. *Ann. Softw. Eng.*, 3:397–415, 1997.

[WN94] Robert Watkins and Mark Neal. Why and How of Requirements Tracing. *IEEE Software*, 11(4):104–106, 1994.

[yJNCD07] Hsin yi Jiang, Tien N. Nguyen, Carl K. Chang, and Fei Dong. Traceability Link Evolution Management with Incremental Latent Semantic Indexing. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '07, pages 309–316, Washington, DC, USA, 2007. IEEE Computer Society.